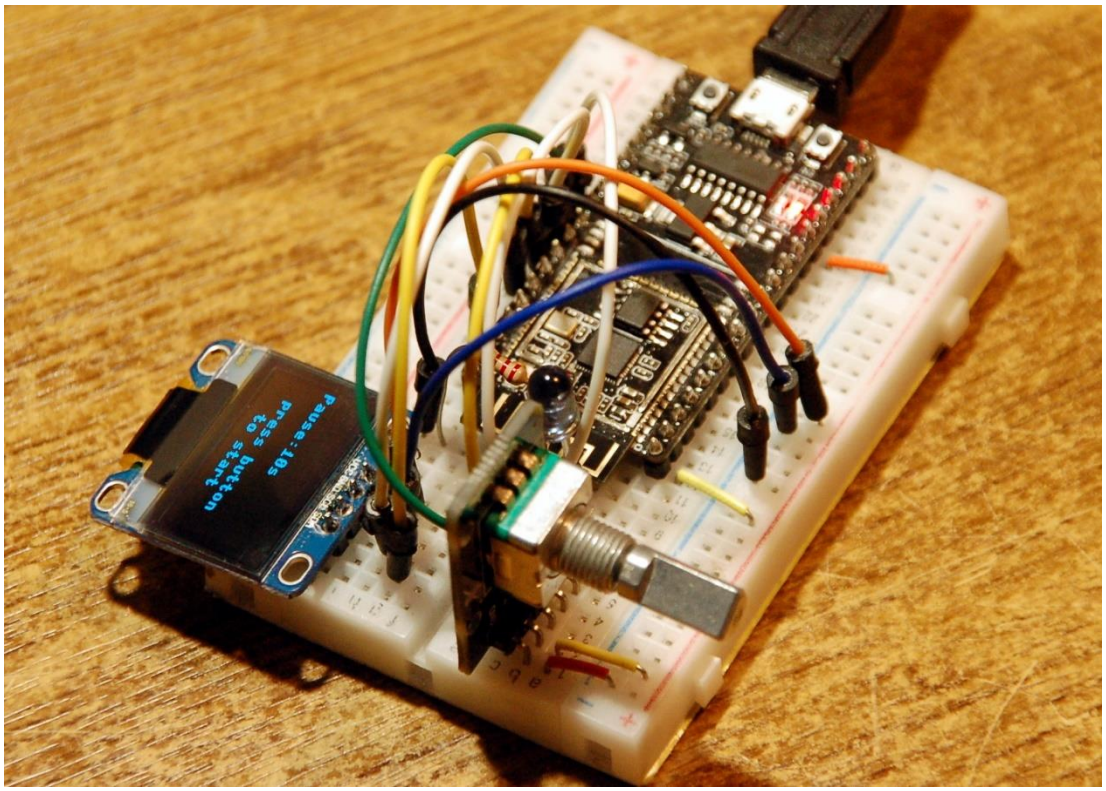

Nikon-Timer

Diese Anleitung gibt es auch als PDF-Dokument

Für meine ersten Fotoapparate hatte ich, um Verwacklungen auf dem Stativ zu vermeiden, einen Drahtauslöser. Um auf größere Entfernungen aufs Knöpfen drücken zu können, gesellte sich später ein pneumatischer Auslöser mit 3m Schlauch dazu. Heute ersetze ich Draht und Schlauch durch Infrarotlicht und Handy. Außerdem lasse ich intervallgesteuert den Auslöser betätigen, das liefert Serienaufnahmen, die man gut zu einem Zeitrafferfilmchen zusammensetzen kann. Die Schaltung besteht aus grade mal vier Teilen, und auch das Programm für einen ESP32 oder ESP8266 ist kurz und sehr übersichtlich, also ein schnelles Projekt für Einsteiger aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der Foto-Timer für die Nikon

Der Urvater dieses Projekts stammt aus dem Jahr 2013. Damals hatte ich einen AT-Tiny2313 mit Assembler darauf programmiert, eine IR-LED mit genau denselben Signalen anzusteuern, wie es die Original Nikon-RC tut. Natürlich wurde auch bereits die Timer-Funktion mit integriert. Wie man von den ersten Messungen zum fertigen Produkt kommt, das zeige ich heute mit einem ESP32S. Das Programm lässt sich aber auch ohne Änderungen genauso gut mit einem ESP8266 D1 mini und seinen Verwandten verwirklichen, mit Ausnahme des ESP8266-01, denn der Käfer hat zu wenig Beinchen.

Hardware

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F oder ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	KY-040 Drehwinkelgeber Drehgeber Rotary Encoder Modul
1	KY-005 IR Infrarot Sender Transceiver Modul
1	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
1	KY-022 Set IR Empfänger Infrarot Receiver CHQ1838
diverse	Jumperkabel
1	Minibreadboard oder Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins
1	Logic Analyzer

ESP32 oder ESP8266 sind beide mit einer Ausnahme gleichermaßen für dieses Projekt geeignet. Die Ausnahme ist der ESP8266-01, weil der einfach zu wenig herausgeführte GPIO-Leitungen hat.

Die Software

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

Verwendete Firmware für den ESP8266/ESP32:

MicropythonFirmware
Bitte eine Stable-Version aussuchen
ESP8266 mit 1MB Version 1.18 Stand: 25.03.2022 oder
ESP32 mit 4MB Version 1.18 Stand 25.03.2022
Die Versionsnummer ist entscheidend für die Umsetzung des Projekts.

Die MicroPython-Programme zum Projekt:

oled.py OLED-Frontend
ssd1306.py OLED-Treibermodul
rotary.py Treiber für Winkel-Encoder portübergreifend
rotary_irq_esp.py Treiber für ESP32/ESP8266
ir_ausloeser.py Testprogramm für die Auslösesequenz
nikon_timer.py Betriebssoftware

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.

Kontaktaufnahme mit (Micro)-Python

Wenn sie bereits mit der Arduino-IDE gearbeitet haben, hatten sie vielleicht hin und wieder das Verlangen einen Befehl auszuprobieren, um dessen Verhalten zu untersuchen. Aber dort ist eine interaktive Arbeit mit dem System nicht vorgesehen, ich habe das weiter oben schon erwähnt.

MicroPython als Abkömmling der Interpretersprache C-Python verhält sich da anders. Sie arbeiten in einer IDE, einer integrierten Entwicklungs- Umgebung, mit direktem Draht zu einem Python-Interpreter. In Python nennt sich dieses Tool **REPL**. Das ist ein Akronym für Read – Evaluate - Print – Loop. Sie machen am Terminal eine Eingabe, der MicroPython-Interpreter liest sie ein, wertet die Eingabe aus, gibt eine Antwort zurück und wartet auf die nächste Eingabe. Die Eingabeaufforderung von Windows, früher MS-DOS, arbeitet nach dem gleichen Schema.

Es gibt eine ganze Reihe von IDEs, Thonny, µPyCraft sind zwei davon, Idle und MU sind zwei weitere. Ich arbeite gerne mit Thonny, weil es außer dem Terminal (REPL) und einem komfortablen Editor zum Erstellen und Testen von Programmen auch noch über 12 Assistenten, einen Plotter und ein Tool zum Flashen der Firmware verfügt. Mit REPL ist es ein Leichtes, jede Anweisung einzeln zu testen. Wir werden das an verschiedenen Stellen in dieser Anleitung ausprobieren.

Noch etwas ist bei Python anders als bei C oder anderen Compiler-Sprachen. Es gibt keine Typfestlegung beim Deklarieren von Variablen oder Funktionen. MicroPython findet den Typ selbst heraus. Das testen wir doch gleich einmal. Starten Sie schon mal Thonny. **Eingaben** am REPL-Prompt >>> formatiere ich im Folgenden **fett**, die *Antworten* vom System *kursiv*.

Bezeichner für Variablen, Konstanten, Funktionen und andere Objekte beginnen stets mit einem Buchstaben oder mit einem Unterstrich, nie mit einer Ziffer. MicroPython ist case sensitive, es unterscheidet also Groß- Kleinschreibung. Hier weise ich der Variable x den Wert 129 zu, und der Interpreter erkennt den Typ Ganzzahl (integer oder int), der Typ von y ist float, also Fließkommazahl.

>>> x = 129
>>> type(x)
<class 'int'>

>>> y=23.7
>>> type(y)
<class 'float'>

Alles in MicroPython ist ein Objekt. x ist ein Objekt der Klasse int, und y eines der Klasse float. Genau genommen sind die Bezeichner x und y nur Referenzen auf die eigentlichen Daten, also die Zahlenwerte im Speicher. Es gibt noch eine ganze Reihe von weiteren "Geheimnissen" zum Thema Variablen, aber das soll fürs Erste an Infos zu den Hintergründen genügen.

Natürlich gibt es auch Zeichenketten oder Strings.

```
>>> t="Hallo Freunde!"
>>> type(t)
<class 'str'>
```

Ein ganz besonderer Datentyp ist **None**. Ein Objekt davon, auch Instanz genannt, verweist auf nichts. Man verwendet None immer dann, wenn man eine Variable deklarieren, aber deren Wert noch nicht festlegen möchte.

```
>>> r=None
>>> type(r)
<class 'NoneType'>
```

Nach den ersten Gehversuchen erstellen wir eine Anwendung, die mit einem ESP8266 oder ESP32, einem Drehwinkelgeber, einer IR-Sendediode und einem OLED-Display zeitgesteuert eine Nikon-Kamera auslösen kann.

Wir vermessen den "Original"-Nikon-Fernauslöser

Wenn man wissen will, welche Signale eine IR-Fernsteuerung aussendet, baut man rudimentär einen Empfänger nach und bestrahlt diesen mit den Signalen des Originalgeräts. Das liefert eine Folge von Impulsen verschiedener Länge. In deren Abfolge ist die übertragene Nachricht codiert. Aber ganz so einfach ist die Sache dann doch nicht, da brauchen wir noch ein wenig Hintergrund Informationen, und damit fangen wir einmal an.

Würden einfach nur Rechteckimpulse von einer RC (Remote Control aka Fernsteuerung) abgestrahlt, dann stünde die Übertragung wegen des Einflusses von Tages- und Raumlicht auf sehr unsicheren Beinen. Deshalb sendet die RC sogenannte Bursts, das sind Pakete eines höherfrequenten Trägers, in unserem Fall 38kHz. Die Dauer der Bursts wird durch die Länge der zu übertragenden Rechtecksignale vorgegeben. Die Impulse der Trägerfrequenz schalten die IR-LED für die kurze Zeit von ca. 13µs ein und genauso lang wieder aus.

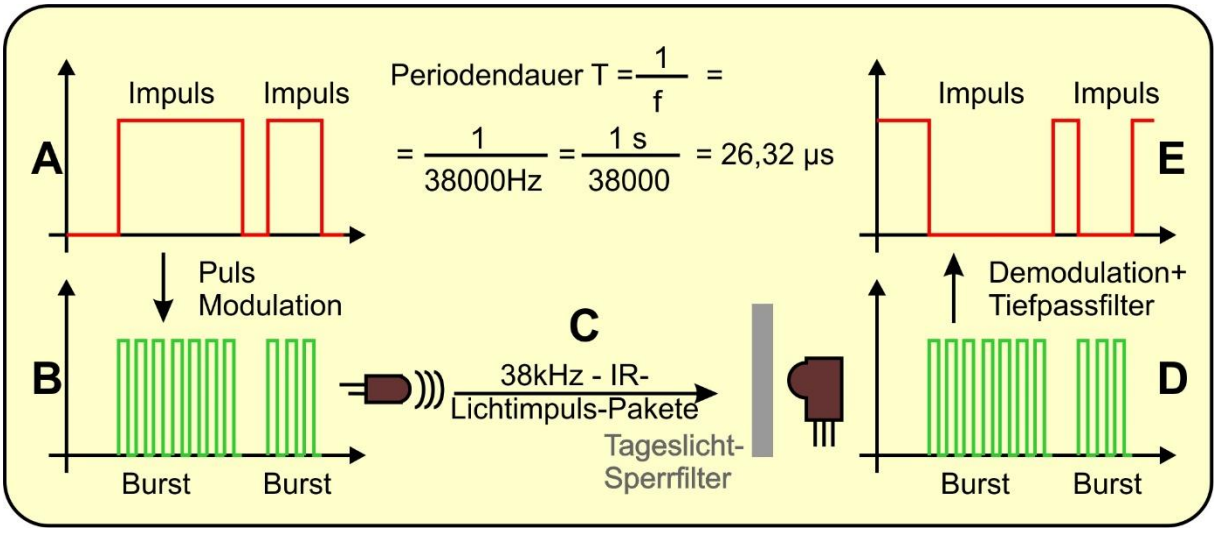

Abbildung 1: Optischer Informationsfluss mit 38kHz Trägersignal

Nur Impulspakete mit 38kHz-Träger werden vom Empfänger wieder in ein Rechtecksignal gleicher Breite mit invertierten Pegeln umgesetzt. Fremdlicht kann durch das IR-Durchlassfilter am Empfängerbaustein zudem wirkungsvoll unterdrückt werden.

Mit Hilfe des Logic-Analyzers (LA) werden wir jetzt die bei E empfangenen Signale sichtbar machen und vermessen. Die Schaltung ist einfach, der Aufbau auch.

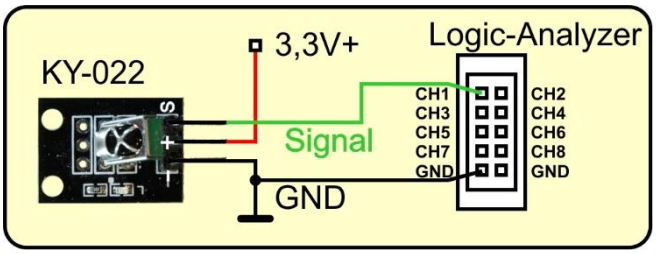

Abbildung 2: Signalabtastung der Nikon-Fernsteuerung

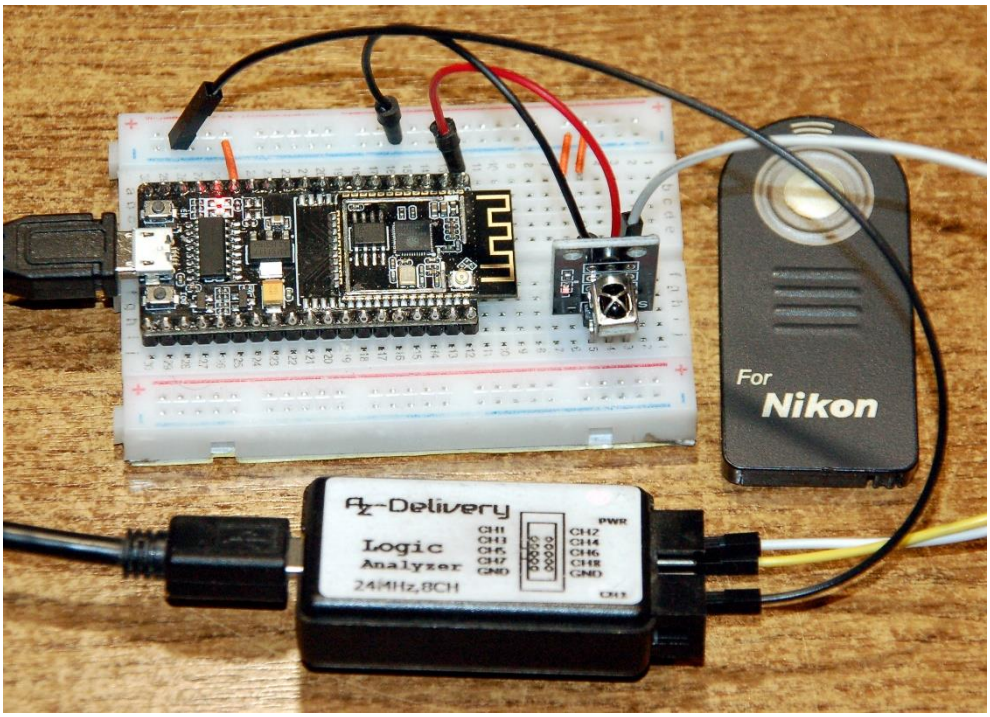

Abbildung 3: Nikon-RC auslesen

Als Betriebssoftware für den LA habe ich das empfohlene Programm von SALEAE verwendet. Es spricht den LA über eine USB-Verbindung an. Eine gute Beschreibung zur Installation und Inbetriebnahme finden Sie in dem Beitrag von Bernd Albrecht "Logic Analyzer - Teil 1: I2C-Signale sichtbar machen". So sieht das Impulsdiagramm der Nikon-RC aus.

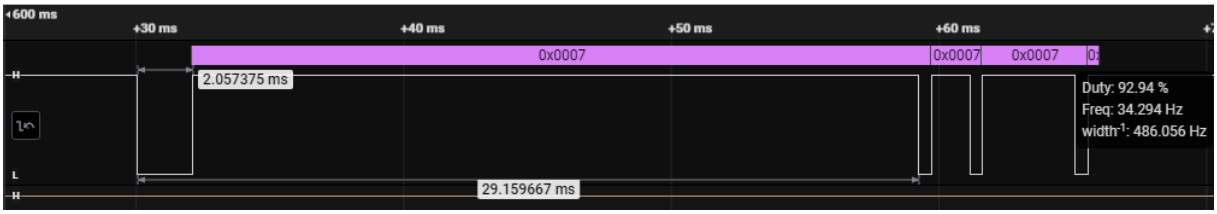

Abbildung 4: Originalsignal

Das Signal ist bereits demoduliert, die Hochfrequenz herausgefiltert.

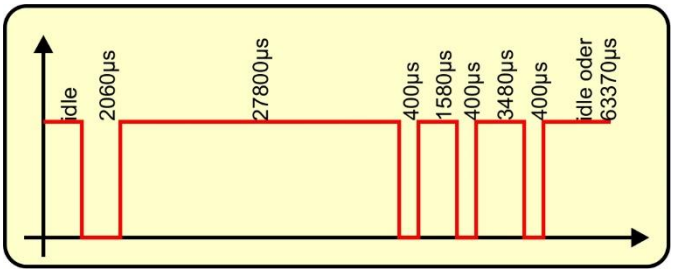

Abbildung 5: Signalabtastung der Nikon-Fernsteuerung

Der Ruhezustand des Empfängersignals ist 3,3V entsprechend einer logischen 1. Während der LOW-Pulse feuert die LED des Handsenders.

Der ESP-Clone des Handsenders

Das Impulsdiagramm gilt es nun nachzubilden. Dabei müssen wir die Pegel invertieren, das heißt, zu den Zeiten, zu denen der Pegel in Abbildung 5 auf LOW liegt, muss unser Controller mit 38kHz-Bursts feuern.

Dazu hilft uns ein Feature, das in der Firmware-Version 1.15 von MicroPython noch nicht vorhanden war. In diesem Projekt ist daher die Firmware 1.18 aufwärts für das Funktionieren essentiell. Das betrifft sowohl den ESP32 als auch den ESP8266.

In diesem Release gibt es nämlich im Modul **machine** die Funktion **bitstream**(), die zumindest in den Versionen bis einschließlich 1.15 noch nicht enthalten ist. Genau diese Funktion versorgt uns nämlich mit dem 38kHz-Trägersignal. In Abbildung 1 habe ich bereits die Periodenlänge zu ca 26,32µs =26320ns berechnet. Puls und Pause sind jeweils gleich lang also 13160ns. Jetzt schauen wir uns auf dieser Basis einmal die Syntax der Parameterliste der Funktion bitstream() an.

Bitstream(**outPin**, **config**, (**high_time_0**, **low_time_0**, **high_time_1**, **low_time_1**),**data**)

outPin: GPIO-Pin für die Taktausgabe
config: 0 gegenwärtig die einzig wählbare
data: zu codierende Datenbytes

Das nachfolgende Tuple enthält zwei Paare von Zeitvorgaben. 0 und 1 werden in der Regel durch unterschiedlich lange Pulse und Pausen codiert. Die folgenden vier Werte dürfen keinesfalls als HIGH-Byte und LOW-Byte (MSB und LSB) eines 16-Bitwortes verstanden werden. Die 32-Bit-Integer-Werte geben Zeiten in Nanosekunden ($1 \bullet 10^{-9}$ s= 0,000000001s) an.

high_time_0: die Pulsdauer für eine 0
low_time_0: die Pausendauer für eine 0

high_time_1: die Pulsdauer für eine 1
low_time_1: die Pausendauer für eine 1

In den von mir ausgewählten Daten kommt zwischen zwei 1-en keine 0 vor, bestenfalls am Ende der Bytes-Folge nach der letzten 1. Die Belegung des 4-er-Tuples ist demnach wie folgt:

(0,0,13160,13160).

Die Anzahl der 1-en im Bytes-Objekt data bestimmt die Breite des 38kHz-Bursts. 1 Byte der Form 0xFF enthält 8 Einsen und steht daher für 8 Perioden der Länge 26,32µs. Für eine Breite von 2060µs brauche ich also

2060µs / 26,32µs = 78,27 1-er-Bits,

mit 8 Bits pro Byte daher

78,27 / 8 = 9,78 Byte.

0,78 Byte entspricht

0,78 • 8 = 6,26 Bit.

6 Einser-Bits werden mit 2 Nullen aufgefüllt und das ergibt

0b1111 1100 = 0xFC.

Die Kontrolle am DSO ergab, dass dieser Burst etwas zu lang ausfiel. Die Genauigkeit wird für den ESP32 oder den ESP8266 mit +/-30ns angegeben. Ich habe den Wert also auf 4 1-en verkürzt zu 0xF0. Folgende Anweisung setzt somit am Ausgang GPIO17 einen Burst von ca. 2060µs ab:

bitstream(out,0,(0,0,13160,13160),(b'\xff'*9)+b'\xf0')

Für die Pausen zwischen den Bursts lasse ich den Controller einfach ein paar Microsekunden schlafen. Die Sequenz für die Auslösung eines Fotos sieht jetzt so aus:

ir_ausloeser.py

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

out=Pin(17,Pin.OUT,value=0)

bitstream(out,0,(0,0,13160,13160),(b'\xff'*9)+b'\xf0')
sleep_us(27800)
bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
```

```
sleep_us(1580)
bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
sleep_us(3480)
bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
sleep_us(63100)
```

Zum Testen brauchen wir eine Testschaltung. Die ist immens aufwendig, sie besteht nämlich grade mal aus drei Teilen. Die Anode der IR-Sendediode liegt am Anschluss S, die Kathode an -. Für den Test benutze ich GPIO17 als Steuerausgang, das werde ich später noch ändern.

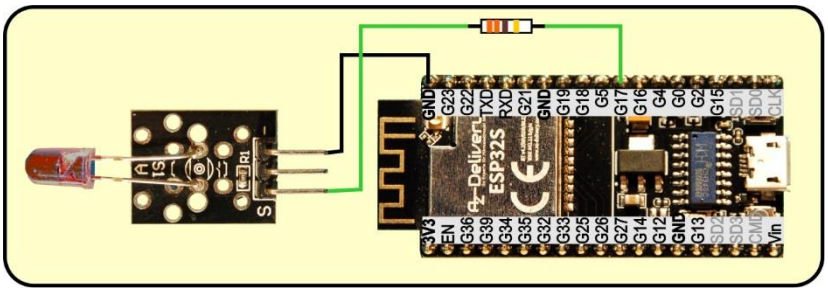

Abbildung 6: Nikon-IR-Auslöser - Testschaltung:

Laden Sie jetzt das Programm in ein Editorfenster. Schalten Sie die Kamera ein und aktivieren Sie die IR-Fernauslösung. Wenn Sie die IR-Diode auf den IR-Sensor der Nikon richten, wird mit dem Start des Programms eine Aufnahme ausgelöst. Ist das nicht gespenstisch?

Das Programm könnten wir bereits jetzt unter dem Namen boot.py abspeichern und in den Flash des ESP32/ESP8266 schicken. Wird die Schaltung dann mittels einer Taste mit Spannung versorgt, startet das Programm automatisch und bedient den Auslöser an der Kamera. Das könnten wir machen, tun wir aber nicht, denn wir haben noch etwas viel Besseres vor.

Der Nikon-Timer

Es wäre doch auch nicht schlecht, wenn unser IR-Auslöser mit Hilfe einer Zeitsteuerung Serienaufnahmen machen könnte. Natürlich sollte man verschiedene Zeitintervalle einstellen können. Das ginge mit zwei Tasten für rauf und runter, aber noch schöner geht das mit einem Winkelencoder. Das sind die Teile, die aussehen wie ein Potentiometer, bei deren Bedienung man aber beim Drehen eine Rasterung spürt. Hier wird kein Widerstand eingestellt, sondern es werden Schaltkontakte in besonderer Weise zeitversetzt geschlossen und geöffnet. Was dabei an den beiden Anschlüssen der Schalter in Verbindung mit dem Massekontakt herauskommt, ist der Graycode der Zahlen 0 bis 3. Die Besonderheit des Graycodes ist, dass sich beim Hochzählen stets nur genau ein Bit ändert.

Nun möchten wir natürlich mehr als nur vier Zustände haben, denn wir wollen auch mehr als nur vier Zeitintervalle auswählen können. An dieser Stelle kommt uns ein MicroPython-Modul namens **rotary** zu Hilfe. Durch die darin enthaltenen Klassen erhalten wir einen Zähler, der beliebig weit hoch- und herunterzählt. Das Modul steht unter der MIT-Lizenz und ist kostenlos nutzbar. Zwei Modi bieten sich für unseren Zweck an.

- Wir könnten die Zeitintervalle sekundengenau durch Drehen am Knopf einstellen. Das ist für Zeiten bis etwa 20s oder 30s brauchbar, scheidet aber für längere Zeiten aus.
- Die andere Variante, die ich gewählt habe, erlaubt die Auswahl einer Zeitspanne aus einer Vorgabe von 24 Zeiten im Raster von Sekunden, Minuten und Stunden.

Und damit man sehen kann, was eingestellt wurde, habe ich der Schaltung noch ein OLED-Display spendiert. Alles zusammen sieht im Schaltbild dann für den ESP32 und den ESP8266 so aus.

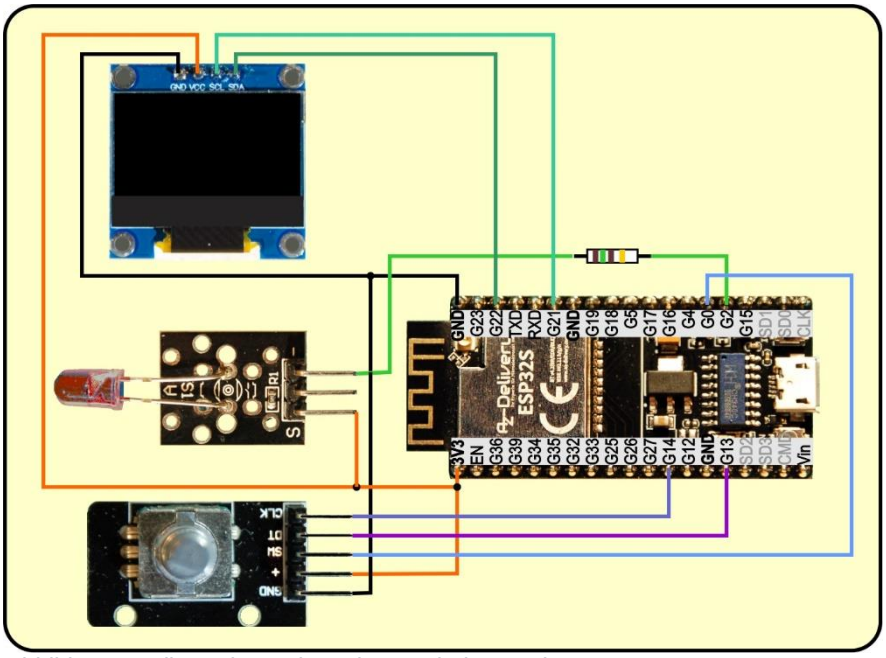

Abbildung 7: Nikon-Timer - komplette Schaltung mit ESP32

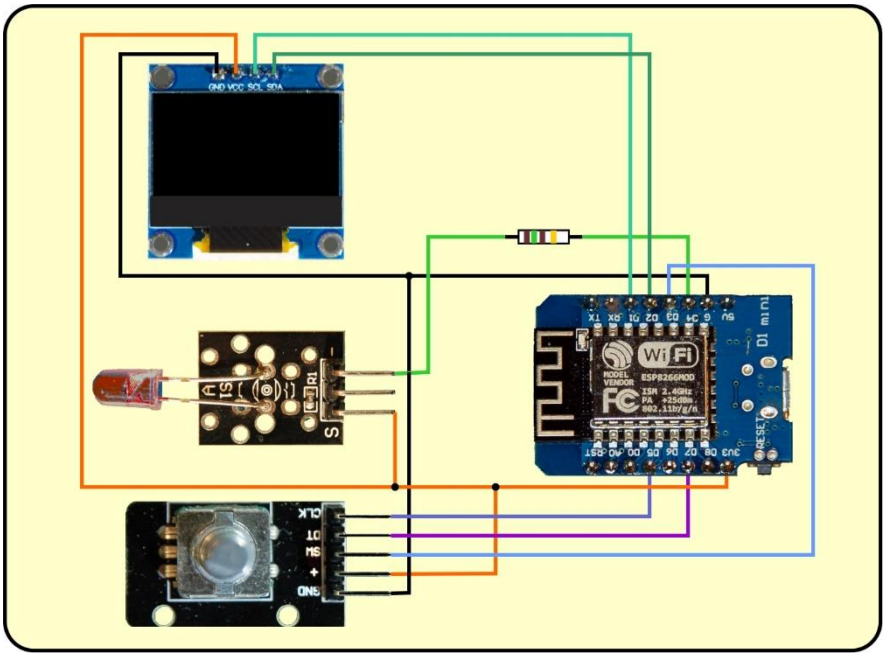

Abbildung 8: Nikon-Timer - komplette Schaltung mit ESP8266

Damit der Entwurf sowohl für den ESP32 als auch für den ESP8266 nur eine Programmversion benötigt, musste darauf geachtet werden, dass die ausgewählten Pins auf beiden Controllern verfügbar sind und, dass sie die gleiche Funktionalität aufweisen. Beim ESP8266 ist das nicht so einfach, weil verschiedene Anschlüsse während der Boot-Phase bestimmte Pegel führen müssen, damit der Controller startet. Für die Abfrage des Winkelencoders müssen die GPIO-Pins außerdem interrupttauglich sein. Da bleibt beim ESP8266 nicht mehr viel übrig. Die Tabelle gibt Auskunft über die Pinbelegung beim ESP8266 D1 Mini.

Beschriftung	GPIO	Input	Output	Bemrkung	benutzt als
D0	GPIO16	kein IRQ	kein PWM, kein bitstream, sonst OK	HIGH beim booten Verbindung mit RST zum Aufwachen von DEEP SLEEP	
D1	GPIO5	OK	OK	SCL	I2C
D2	GPIO4	OK	OK	SDA	I2C
D3	GPIO0	pulled up	OK	Beim Start offen oder auf HIGH, Flashtaste gegen GND	taste
D4	GPIO2	pulled up	OK	HIGH beim Start, liegt über LED und Widerstand an +Vcc	Pulsaus-gang
D5	GPIO14	OK	OK	SPI (SCLK)	CLK
D6	GPIO12	OK	OK	SPI (MISO)	
D7	GPIO13	OK	OK	SPI (MOSI)	DT
D8	GPIO15	pulled down	OK	SPI (CS), muss beim Start LOW sein	
RX	GPIO3	OK	RX pin	HIGH beim Start, serial REPL	
TX	GPIO1	TX pin	OK	HIGH beim Start, serial REPL	
A0	ADC0	Analog Input	-	analoger Eingang bis 3,2V	Vorteiler 220kΩ:100kΩ

Als Pulsausgang für die Burstsignale habe ich GPIO2 gewählt. Der Anschluss liegt über einen Widerstand und eine LED auf +Vcc = 3,3V. Dazu parallel liegen der Widerstand von 180 Ohm und mit ihm die IR-LED. Die blaue LED auf dem D1-Board blitzt kurz auf, wenn auch die IR-LED feuert. Zur Funktionskontrolle reicht das.

Die beiden Schalter im Drehwinkelgeber sind mit je einem 10kΩ-Widerstand gegen Vcc gezogen. Wenn ein Schalter schließt, wird der herausgeführte Anschluss auf GND-Potenzial gelegt.

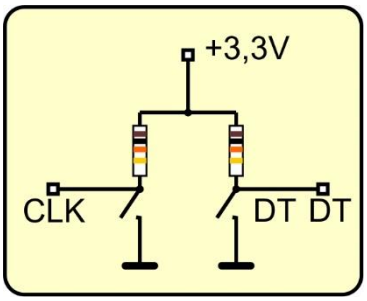

Abbildung 9: Interne Schaltung des Winkelencoder-Moduls

Bei einer Rechtsdrehung ergibt sich folgendes Impulsdiagramm. Oben liegt der Anschluss CLK, unten DT.

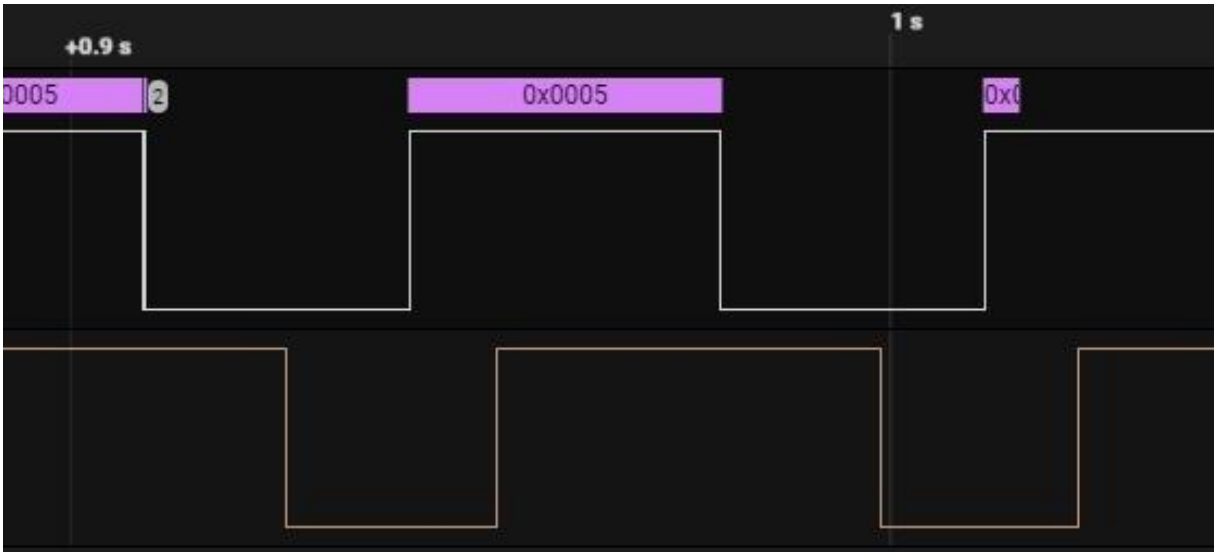

Abbildung 10: Rechtsdrehung

Die Module **rotary_irq_esp.py** und **rotary.py** übernehmen die Decodierung der Drehrichtung und liefern einen Zähler, dessen Eigenschaften beim Aufruf des Konstruktors der Klasse **RotaryIRQ** verändert werden können. Das sind die Modi:

1. Zählen ohne Grenzen
2. Zählen mit Unter- und Obergrenze
3. Anhalten beim Erreichen einer Grenze
4. Umkehren der Zählung beim Erreichen einer Grenze

Ich habe mich für den Modus 3 entschieden, weil dadurch ein Überschreiten der Indexgrenzen meiner Liste **delay** automatisch verhindert wird.

Jetzt wird es Zeit für den Aufbau der Hardware. Laden Sie dann die Module am besten schon einmal herunter und kopieren Sie dieselben in das Arbeitsverzeichnis von Thonny. Von dort müssen alle vier Dateien in den Flash des Controllers verfrachtet werden. Dazu die Dateien mit gedrückter STRG-Taste markieren, dann Rechtsklick auf eine davon und im Kontextmenü **Upload to /** klicken, mit OK bestätigen.

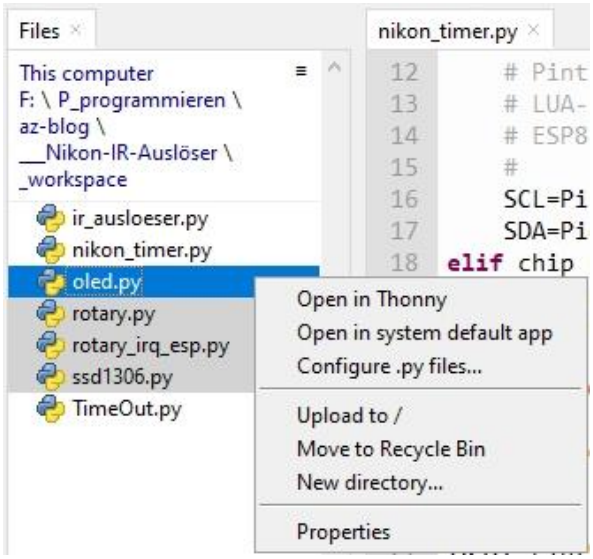

Abbildung 11: Module in den Flash transferieren

Dann geht es an die Besprechung des Programms **nikon_timer.py**. Wir starten, wie immer, mit dem Importgeschäft.

```
# nikon_timer.py
#
import sys
from rotary_irq_esp import RotaryIRQ
from time import sleep_ms,ticks_ms, sleep_us
from machine import SoftI2C, Pin, bitstream
from oled import OLED
from ssd1306 import SSD1306_I2C
```

rotary_irq_esp lädt selbsttätig das Modul **rotary** nach, um davon zu erben. Von **time** holen wir uns ein paar Routinen zur zeitlichen Verzögerung und Zeiterfassung. Die Methode **bitstream**() holen wir neben dem I2C-Interface von **machine**. **oled** und **ssd1306** brauchen wir zur Ansteuerung des Displays.

Dann stellen wir den Typ des Controllers fest und belegen entsprechend die Variablen zur Einrichtung der I2C-Schnittstelle.

```
chip=sys.platform
if chip == 'esp8266':
    # Pintranslator fuer ESP8266-Boards
    # LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
    # ESP8266 Pins 16  5  4  0  2 14 12 13 15
    #                 SC SD
    SCL=Pin(5) # S01: 0
    SDA=Pin(4) # S01: 2
elif chip == 'esp32':
    SCL=Pin(21)
    SDA=Pin(22)
else:
    raise OSError ("Unbekannter Port")
```

Mit dem eingerichteten I2C-Interface instanziieren wir gleich das OLED-Display-objekt.

```
i2c=SoftI2C(SCL,SDA)
d=OLED(i2c)
```

Es folgt die Deklaration der GPIO-Pins für die Taste am Rotary-Encoder, den Impulsausgang für die IR-LED und die Impulsgeber am Encoder.

```
taste=Pin(0,Pin.IN)  #  D3
out=Pin(2,Pin.OUT,value=1)  # D4
CLK=14  # D5
DT=13   # D7
```

Die Liste **delay** definiert die Zeitintervalle für den Timer in Sekunden.

```
delay=[
    86400,
    5,    10,    20,    30,    40,    50,    60,
    90,  120,   180,   240,   300,   360,   480,  600,
    900,1200,  1800,  2400,  3000,  3600,  5400, 7200,
    ]
```

Mit Hilfe der Funktion **TimeOut**() erzeuge ich einen nichtblockierenden Softwaretimer für die Intervallsteuerung. Dazu übergebe ich die Zeitdauer in Millisekunden. In der lokalen Variablen **start** wird der aktuelle Zeitpunkt in Millisekunden abgelegt. TimeOut gibt statt eines Werts eine Referenz auf die lokal definierte Funktion **compare**() zurück. compare() ist eine sogenannte Closure und die bewirkt, dass die Variable **start** und der Parameter **t** auch nach dem Verlassen der umgebenden Funktion TimeOut() bis zu weiteren Aufrufen von compare() erhalten bleiben. Weiter unten sehen Sie die Anwendung. Normalerweise, werden ja lokal erzeugte Objekte nach dem Verlassen der Funktion eingestampft, aber eben nicht so bei einer Closure.

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

Die Sequenz in der Funktion **ausloesen**() kennen wir bereits. Die Schaltung der IR-LED in Abbildung 6 ist HIGH-aktiv. Das heißt, die LED leuchtet, wenn auch unsichtbar, wenn der Ausgang logisch auf 1 und elektrisch auf 3,3V liegt. Nach dem Ende der Burstsequenz bleibt der Ausgang aber auf 0. An GPIO2 hätte das zur Folge, dass die LED auf dem ESP8266-Board quasi während des Pausen ständig leuchtet. Deshalb habe ich die Schaltung der IR-LED von HIGH-aktiv auf LOW-aktiv für den Anschluss an GPIO2 geändert. Nun bliebe aber nach Abschluss eines Bursts mit dem Ausgang auf LOW-Potenzial die IR-LED eingeschaltet. Das ist schlecht, weil

dann auch während der Pausen zwischen dem Auslösen der Kamera die IR-LED in Betrieb ist und eine höhere Stromaufnahme veranlasst. Aus diesem Grund habe ich nach jedem Burst den Ausgang auf 1 gelegt. Damit liegen beide Anschlüsse der LED auf +Vcc und die LED ist aus, ebenso wie die blaue LED auf dem Board. Letztere blitzt kurz auf, wenn auch die IR-LED feuert. So ist das gut!

Weiter im Programm, ich lösche die OLED-Anzeige und gebe die Startmeldung aus.

```
d.clearAll()
d.writeAt("NIKON-TIMER",2,0,False)
d.writeAt("press button",2,4,False)
d.writeAt("to start",4,5)
```

Es wird Zeit für die Instanziierung des Drehwinkelgeber-Objekts, Zuweisung der Pins und Vorgabe der Grenzwerte. Ich habe wohl die GPIOs verkehrt herum angeschlossen, deshalb sage ich reverse=True, weil ich keine Lust habe, die Schaltskizzen ein viertes Mal zu ändern. Im Uhrzeigersinn drehen zählt jetzt auf jeden Fall hoch.

Ich hole den Startwert des Encoders nach **indexOld** und starte den Intervalltimer mit der Intervallzeit für einen Tag, 86400 Sekunden. Das ist nötig, damit **jetzt**() als Funktion initialisiert wird. Die Referenz auf die Funktion **compare**() weise ich dem Bezeichner **jetzt** zu. Wenn ich danach **jetzt**() aufrufe, rufe ich eigentlich **compare**() auf. Die Rückgabe dieser Funktion ist jetzt so lange **False**, bis seit dem Zeitpunkt in **start t** Millisekunden vergangen sind. Danach bleibt der Rückgabewert **True**.

```
indexOld = r.value()
jetzt=TimeOut(delay[0])
while True:
    index = r.value()

    if indexOld != index:
        indexOld = index
        d.clearFT(0,2,d.width-1,2,False)
        d.writeAt("Pause:{}s".format(delay[index]),0,2)
    if taste.value() == 0:
        jetzt=TimeOut(delay[index]*1000)
        ausloesen()
        print("Foto")
        sleep_ms(1000)
    if jetzt():
        jetzt=TimeOut(delay[index]*1000)
        ausloesen()
        print("Foto")
    sleep_ms(100)
```

Es geht in die Mainloop, und wir holen erneut einen Wert vom Drehwinkelgeber. Wenn sich dieser Wert vom zuvor gemerkten in indexOld unterscheidet, merken wir

uns den neuen, löschen die Zeile 2 der Anzeige von Anfang bis Ende, lesen den Zeitwert zum neuen Index aus der Liste **delay** und schreiben ihn ins Display.

Wurde die Taste am Rotary-Encoder gedrückt, starten wir den Intervalltimer mit dem 1000-fachen des Werts aus der Liste **delay**, **compare**() zählt ja in Millisekunden. Die Variable **start** wird damit auf den jetzt aktuellen Zeitstempel gesetzt. Eine Burstsequenz wird angestoßen, das Terminal meldet "Foto" und wir schicken den Controller für eine Sekunde schlafen.

Mit der Taste können also auch Fotos ohne den Timer aufgenommen werden. Damit der nicht dazwischenfunkt, ist es angebracht die Intervalldauer in diesem Fall auf einen Tag = 86400 Sekunden zu setzen.

Mit **jetzt**() frage ich den Zustand des Timers ab. Erst, wenn der Timer abgelaufen ist, meldet **jetzt**() ein **True** zurück. Dann stellen wir den Timer neu, lösen eine Aufnahme aus und schicken "Foto" ans Terminal.

100 Millisekunden Generalpause und eine weitere Schleifenrunde beginnt.

Hier ist der gesamte Programmtext noch einmal im Zusammenhang. Sie können das Programm nikon_timer.py natürlich auch herunterladen. Öffnen Sie es in einem Editorfenster von Thonny und starten Sie es mit der Funktionstaste F5.

```
# nikon_timer.py
#
import sys
from rotary_irq_esp import RotaryIRQ
from time import sleep_ms,ticks_ms, sleep_us
from machine import SoftI2C, Pin, bitstream
from oled import OLED
from ssd1306 import SSD1306_I2C

chip=sys.platform
if chip == 'esp8266':
    # Pintranslator fuer ESP8266-Boards
    # LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
    # ESP8266 Pins 16  5  4  0  2 14 12 13 15
    #                 SC SD
    SCL=Pin(5) # S01: 0
    SDA=Pin(4) # S01: 2
elif chip == 'esp32':
    SCL=Pin(21)
    SDA=Pin(22)
else:
    raise OSError ("Unbekannter Port")

i2c=SoftI2C(SCL,SDA)
d=OLED(i2c)

taste=Pin(0,Pin.IN)  #  D3
out=Pin(2,Pin.OUT,value=1)  # D4
```

```
CLK=14  # D5
DT=13   # D7

delay=[
    86400,
    5,    10,    20,    30,    40,    50,    60,
    90,  120,   180,   240,   300,   360,   480,  600,
    900,1200,  1800,  2400,  3000,  3600,  5400, 7200,
    ]

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

def ausloesen():
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*9)+b'\xf0')
    out(1)
    sleep_us(27800)
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
    out(1)
    sleep_us(1580)
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
    out(1)
    sleep_us(3480)
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
    out(1)
    sleep_us(63100)

d.clearAll()
d.writeAt("NIKON-TIMER",2,0,False)
d.writeAt("press button",2,4,False)
d.writeAt("to start",4,5)

r = RotaryIRQ(pin_num_clk=CLK,
              pin_num_dt=DT,
              min_val=0,
              max_val=24,
              reverse=True,
              range_mode=RotaryIRQ.RANGE_BOUNDED)

indexOld = r.value()
jetzt=TimeOut(delay[0])
while True:
    index = r.value()

    if indexOld != index:
        indexOld = index
        d.clearFT(0,2,d.width-1,2,False)
        d.writeAt("Pause:{}s".format(delay[index]),0,2)
    if taste.value() == 0:
```

```
        jetzt=TimeOut(delay[index]*1000)
        ausloesen()
        print("Foto")
        sleep_ms(1000)
    if jetzt():
        jetzt=TimeOut(delay[index]*1000)
        ausloesen()
        print("Foto")
    sleep_ms(100)
```

Mit dem Drehwinkelgeber stellen Sie eine Intervallzeit von 5 Sekunden ein und drücken die Taste. Nun blinkt die blaue LED am ESP8266 D1 mini alle 5 Sekunden kurz auf und wenn eine Nikon im Strahlbündel der IR-LED eingeschaltet und auf Fernauslöser gestellt ist, wird alle 5 Sekunden eine Aufnahme ausgelöst.

Dieses Selfie von sich hat der Nikon-Timer übrigens selbst aufgenommen – OK, OK, ich hab die Kamera gehalten, aber er hat immerhin selbst aufs "Knöpfchen gedrückt".

Ausblick

Neulich hatten wir eine Familienfeier und einer der Gäste demonstrierte seine neueste Errungenschaft – fotografieren mittels Handy. Natürlich nicht mit dem Handy, das kann ja jeder, sondern fotografieren mit der Spiegelreflex vermittels des Handys. Damit sind wir zurück bei der Einleitung des heutigen Posts. Nur wird der Pneumatik-Auslöser durch den zeitgemäßeren Funkauslöser ersetzt. In der nächsten Folge werden wir die WLAN-Eigenschaft der ESPs nutzen um mit Hilfe einer Handy-App einen Fernauslöser zu konzipieren.

Sicher fallen Ihnen da noch eine Menge Sachen ein, für die man einen Timer, handbedient oder per Funksteuerung, einsetzen kann. Ich denke da zum Beispiel auch an einen Timer zur Belichtung von fotobeschichteten Platinen zur Herstellung gedruckter Schaltungen.

Aber jetzt wünsche ich erst einmal viel Vergnügen beim Experimentieren mit dem Drehencoder und dem Foto-Timer.

Bis dann!