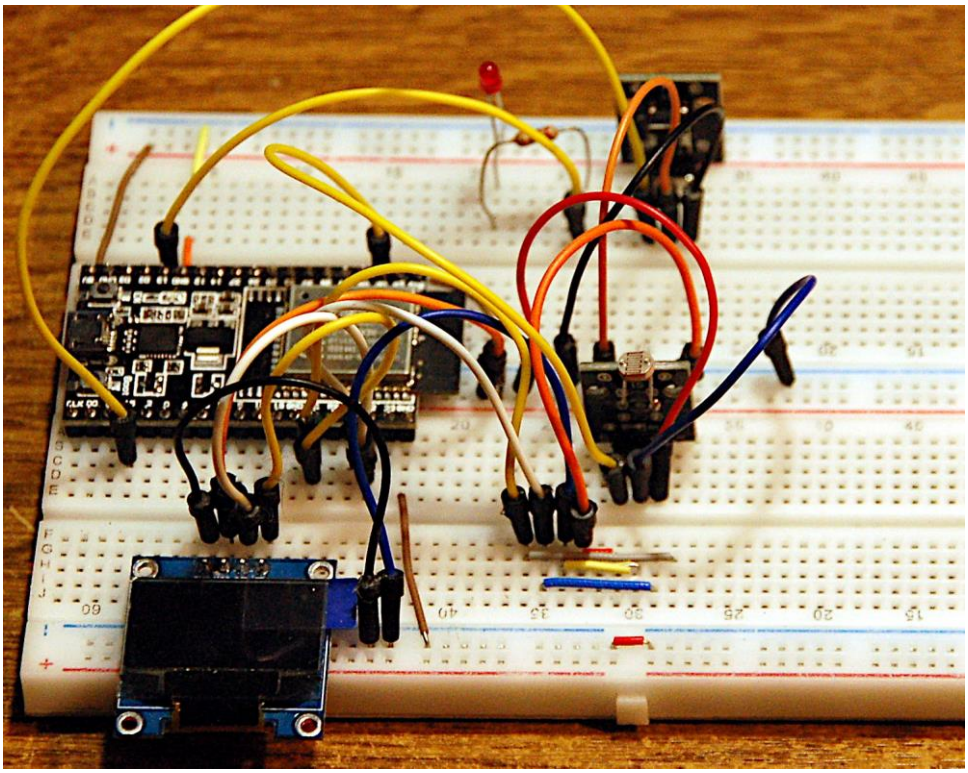

Schaltung mit dem ESP32 und LDR-Modul

Diesen Beitrag gibt es auch als PDF-Dokument.

Neben IFTTT, ThingSpeak und Telegram bietet auch WhatsApp die Möglichkeit, mittels Bot Nachrichten zu verschicken. Die verwendeten Module sind nicht sehr anspruchsvoll, was den Speicherbedarf angeht. Und so sind unsere Controller ESP32 und ESP8266 wieder beide mit dabei. Die Textnachricht, die wir an CallMeBot senden, muss URL-encoded sein. Das heißt, dass Sonderzeichen wie "äöüß" durch Hexadezimalcodes ersetzt werden müssen. Das macht mein Modul **urlencode**.py. Dank **urequests**.py brauchen wir zur Übermittlung auch bei diesem Projekt nur eine Zeile für den Transfer der Daten zum Server. Wie Sie einen WhatsApp-Account bekommen und wie ein Bot eingerichtet wird, erfahren Sie in diesem Beitrag aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

ESP32 und ESP8266 senden Werte über WhatsApp

Vergessen, das Kellerlicht auszuschalten? Ja, das passiert immer mal wieder. Gut, wenn unser ESP das registriert und uns nach einer gewissen Zeit über WhatsApp eine Nachricht zukommen lässt. Der Schaltungsaufwand ist denkbar gering. In der Magerausbaustufe genügt ein Controller, ein LDR (Light dependend resistor = Fotowiderstand) und ein einfacher Widerstand von 10kΩ oder das Modul KY-018, auf dem beides schon montiert ist. Wer möchte, kann ein Display mit dazunehmen, damit

Systemmeldungen angezeigt werden können. Die Schaltung wird im Einsatz ja sehr wahrscheinlich nicht an den PC angeschlossen sein.

Hardware

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder D1 Mini V3 NodeMCU mit ESP8266-12F oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI oder NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI oder ESP32 Dev Kit C unverlötet oder ESP32 Dev Kit C V4 unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder NodeMCU-ESP-32S-Kit oder ESP32 Lolin LOLIN32 WiFi Bluetooth Dev Kit
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	KY-018 Foto LDR Widerstand oder Fotowiderstand Photo Resistor plus Widerstand 10 kΩ
2	MB-102 Breadboard Steckbrett mit 830 Kontakten
1	KY-004 Taster Modul
diverse	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F evtl. auch 65Stk. Jumper Wire Kabel Steckbrücken für Breadboard

Damit neben dem Controller noch Steckplätze für die Kabel frei sind, habe ich zwei Breadboards, mit einer Stromschiene dazwischen, zusammengesteckt.

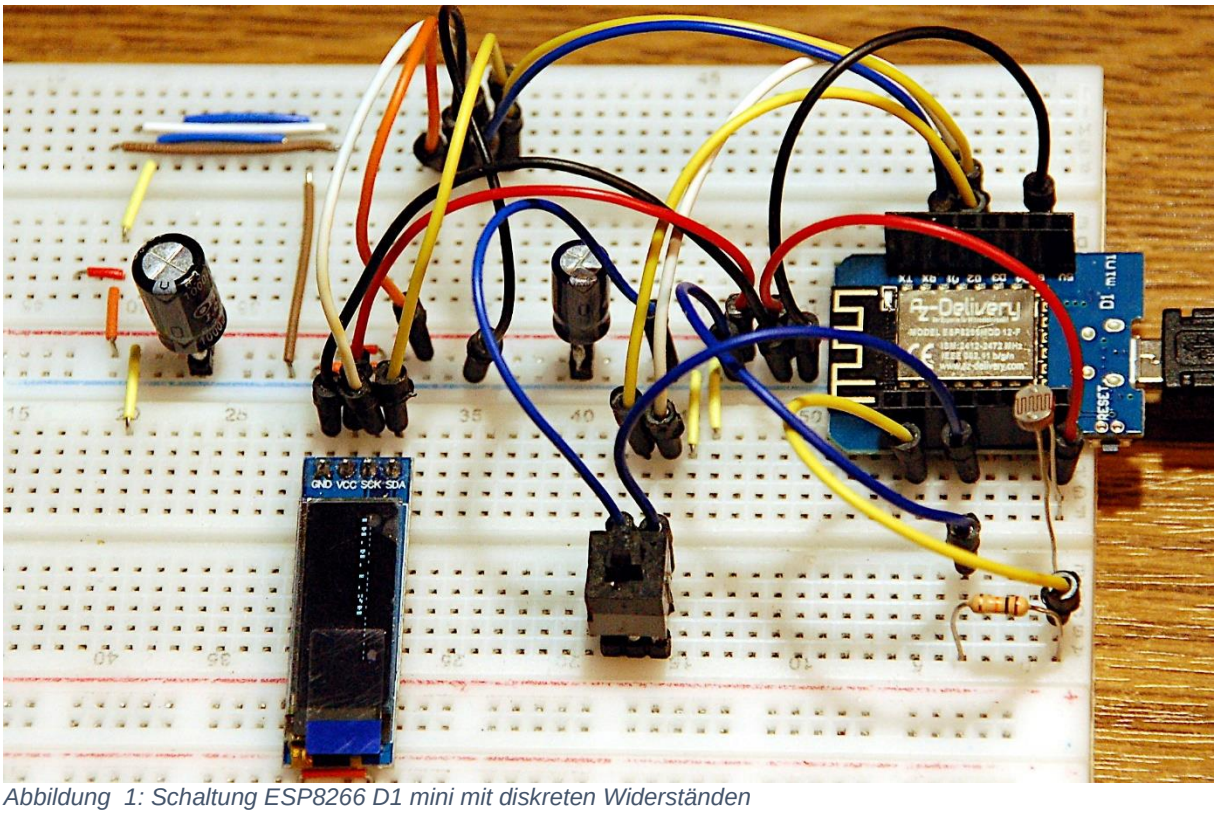

Abbildung 1: Schaltung ESP8266 D1 mini mit diskreten Widerständen

Der LDR verringert bei zunehmender Helligkeit seinen Widerstandswert. Er ist mit einem Festwiderstand von 10kΩ in Reihe geschaltet. Die beiden bilden zusammen einen Spannungsteiler. Am Mittelkontakt S liegen Spannungen von etwas über 0V bis nahe 3,3V an. Damit bei viel Licht auch eine höhere Spannung an S auftritt, muss der LDR an 3,3V und der 10kΩ-Widerstand an GND liegen. Wenn man den losen LDR verwendet kann man das leicht realisieren.

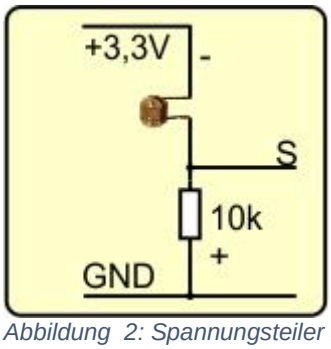

Abbildung 2: Spannungsteiler aus Einzelteilen

Beim Modul KY-018 liegt der LDR aber gegen Masse, wenn man es so anschließt, wie es die Beschriftung sagt. Damit sich das Teil genauso verhält, wie wir es wünschen, muss man GND an den mittleren Stift legen und +3,3V an den rechten. S verbinden wir mit dem Analogeingang des Controllers, GPIO36 beim ESP32 und A0 beim ESP8266.

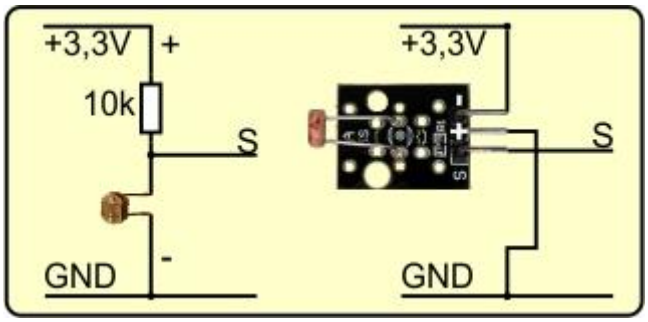

Abbildung 3: Schaltung des Moduls KY-018

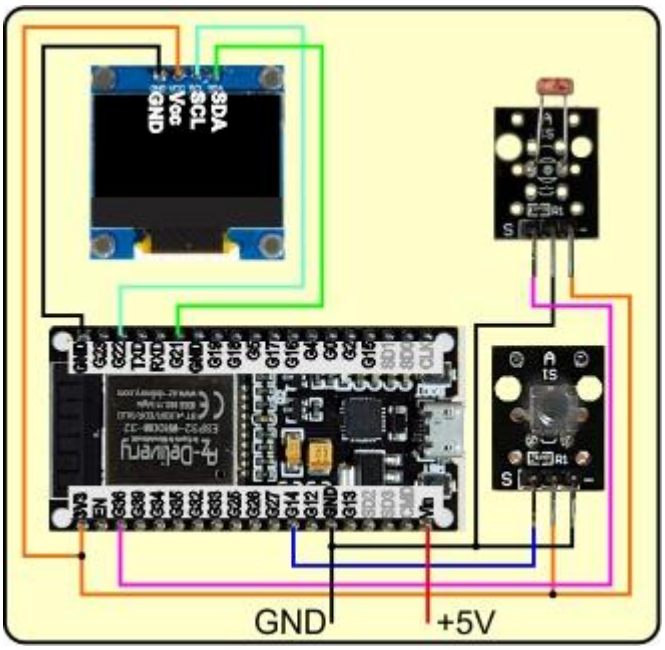

Abbildung 4: Schaltung mit ESP32 und LDR-Modul

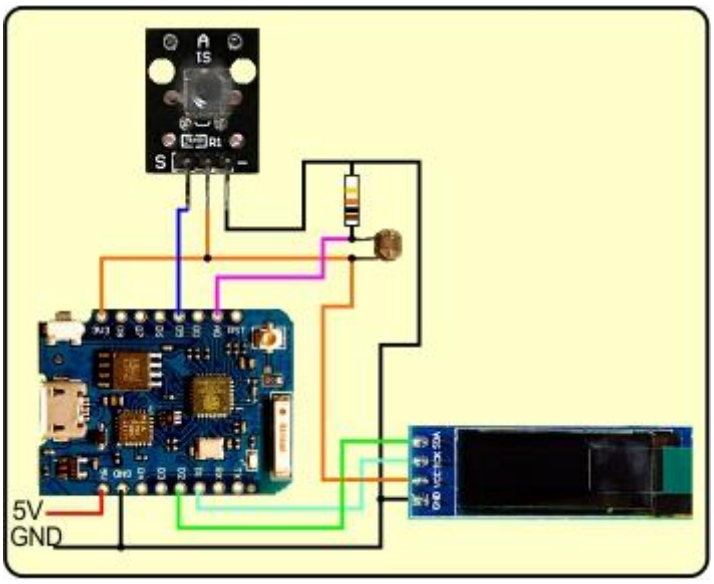

Abbildung 5: Schaltung mit ESP8266 und Einzelwiderständen

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft

Verwendete Firmware für den ESP32:

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

Verwendete Firmware für den ESP8266:

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

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für das OLED-Display
urequests.py Treibermodul für den HTTP-Betrieb des ESP8266
urlencode.py URL-Encoder-Modul
timeout.py Softwaretimer-Modul
WhatsApp.py Demoprogramm für den e-Mailversand

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.

So kommen Sie zu einem WhatsApp-Account

1. WhatsApp aus dem Google Playstore (App Store bei iPhone) herunterladen
2. Die App installieren
3. WhatsApp starten
4. Handy-Rufnummer eingeben. Die übliche 0 fällt weg, dafür +49 verwenden. Also etwa nach diesem Muster: +49 512 123456789
5. Auf die SMS mit dem Bestätigungs-Code von WhatsApp warten – Code eingeben - Fertig tappen.
6. WhatsApp gestatten, auf die Kontaktliste zugreifen zu dürfen
7. Die App hat jetzt alle Kontakte importiert, die auch WhatsApp haben

WhatsApp für den PC

Für den PC gibt es auch eine Windows-App bei portapps.io. Das Entpacken nach dem Start der heruntergeladenen EXE-Datei läuft reibungslos. Allerdings bringt das Programm nach dem Start eine Fehlermeldung und bricht ab.

Alternativ kann man eine App auch bei Chip bekommen. Sie arbeitet aber stets mit dem Handy zusammen, die Geräte müssen gekoppelt werden. Nach dem Start der sofort lauffähigen Datei wird folgendes Fenster eingeblendet.

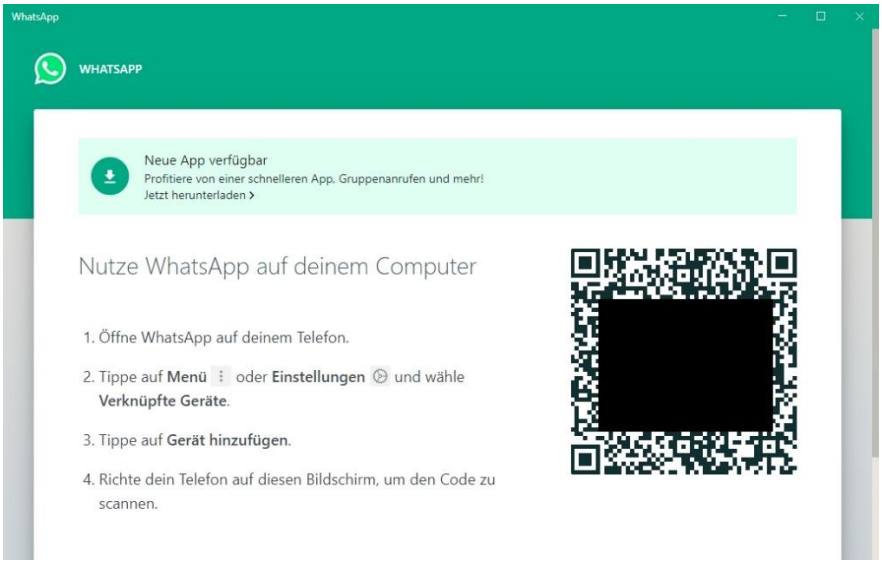

Abbildung 6: WhatsApp portable für den PC

Öffnen Sie WhatsApp auf dem Handy. Gehen Sie ins Menü und tappen Sie Verknüpfte Geräte. Scannen Sie nun den QR-Code mit dem Handy ab. Kurz darauf bekommen Sie ein zweigeteiltes Fenster, in dem Sie links den Chat auswählen. Rechts sehen Sie die Nachrichten.

So erstellen Sie einen Bot

1. Fügen Sie auf dem Handy folgende Nummer zu den Kontakten hinzufügen: +34 644 51 95 23. Geben Sie einen beliebigen Namen dazu ein, er tut nichts zur Sache.
2. Senden Sie über WhatsApp folgenden Text an den neuen Kontakt:
   I allow callmebot to send me messages
3. Kurze Zeit später bekommen Sie eine Nachricht von WhatsApp mit Ihrem API-Key.

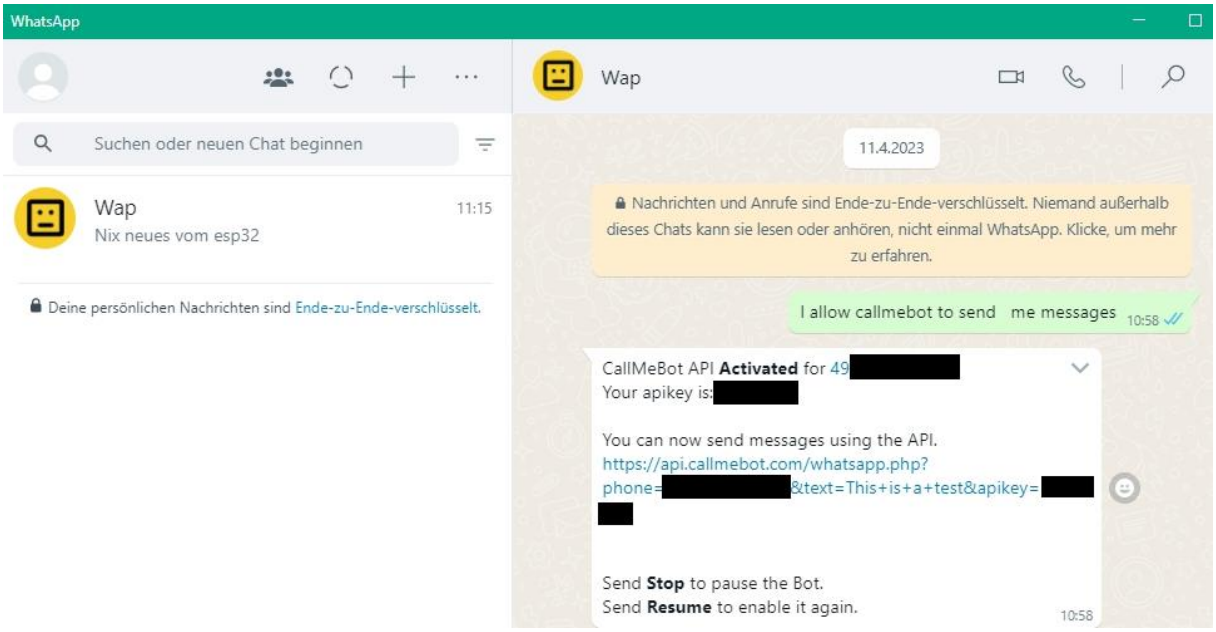

Abbildung 7: API-Key für den neuen Bot

4. Tappen Sie auf die URL-Zeile.
5. Nach ein paar Sekunden bekommen Sie eine Testnachricht von WhatsApp.

Abbildung 8: Test-Nachricht vom Bot

Das MicroPython-Programm zum WhatsApp-Bot

Das Importgeschäft ist von etwas größerem Stil. Pin, SoftI2C und ADC kommen vom Modul **machine**. **time** liefert sleep. Software-Timer für Microsekunden, Millisekunden und Sekunden, die das Programm nicht blockieren, liegen in **timeout** bereit. **urlencode** bietet die Funktion **URLEncode**(), die, zusammen mit den Listen q und z, spezielle Zeichen in Hexadezimalcode übersetzt.

```
from machine import Pin, SoftI2C, ADC
from time import sleep
from timeout import *
from urlencode import *
import urequests as requests
from oled import OLED
import network
import sys
import gc
```

HTTP-Anfragen werden mit **urequests** sehr vereinfacht. Die Klasse **OLED** ist die API für das Display, **network** macht die Verbindung zum WLAN-Router. **sys** nutzen wir für die Abfrage des Controllertyps und für einen gesicherten Programmausstieg. **gc** steht für **Garbage collection** und räumt nicht mehr benötigten Datenmüll weg.

```
trigger=500 # counts
warnLevel=60 # Sekunden
```

Der ADC-Level **trigger**, der hell von dunkel trennt muss für jeden Fall individuell eingestellt werden, ebenso wie **warnLevel**, der Wert für die Wiederholung des Nachrichtenversands.

```
mySSID = 'EMPIRE_OF_ANTS'; myPass = "nightingale"
key="1234567"
phone="+49123456789"
```

Für **mySSID** und **myPass** setzen Sie bitte die Credentials für Ihren Router ein. Gleiches gilt für API-Key und Rufnummer.

Der nächste Block erkennt den Controllertyp und instanziiert dementsprechend ein I2C-Objekt und den ADC.

```
if sys.platform == "esp8266":
    i2c=SoftI2C(scl=Pin(5),sda=Pin(4))
    adc=ADC(0)
elif sys.platform == "esp32":
    i2c=SoftI2C(scl=Pin(22),sda=Pin(21))
    adc=ADC(Pin(36))
    adc.atten(ADC.ATTN_11DB)
    adc.width(ADC.WIDTH_10BIT)
else:
    raise RuntimeError("Unknown Port")
```

Die I2C-Bus-Instanz übergeben wir an den Konstruktor des Display-Objekts, stellen die Helligkeit ein und geben die Titelzeile aus.

```
d=OLED(i2c,heightw=64) # 128x64-Pixel-Display
d.contrast(255)
d.writeAt("Kellerlicht",2,0)
```

Die Taste legen wir an GPIO14 und aktivieren den Pullup-Widerstand.

```
taste=Pin(14,Pin.IN,Pin.PULL_UP)
```

Das Dictionary **connectStatus** übersetzt die Nummern-Codes, die uns **network.status**() liefert in Klartext. Der ESP32 liefert andere Nummern als der ESP8266.

```
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN",
    0: "STAT_IDLE",
    1: "STAT_CONNECTING",
    5: "STAT_GOT_IP",
    2:  "STAT_WRONG_PASSWORD",
    3:  "NO AP FOUND",
    4:  "STAT_CONNECT_FAIL",
    }
```

Damit das Station-Interface vom WLAN-Router den Zugang erhält, muss die MAC-Adresse davon dem Router bekannt sein, falls dort MAC-Filterung aktiviert ist. Es ist übrigens keine gute Idee, die Filterung auszuschalten, weil sich dann beliebige WLAN-Nomaden leichter am Router einloggen können.

Die Anmeldesequenz zum Router habe ich dieses Mal in eine Funktion verpackt, die das Station-Objekt zurückgibt. Solange noch keine Verbindung besteht, wird im Sekundentakt ein Punkt im Terminal und am Display ausgegeben.

```
def connect2router():
    # ************** Zum Router verbinden ******************
    nic=network.WLAN(network.AP_IF)
    nic.active(False)

    nic = network.WLAN(network.STA_IF)  # erzeugt WiFi-Objekt
    nic.active(True)                    # nic einschalten
    sleep(1)
    MAC = nic.config('mac')# binaere MAC-Adresse abrufen und
```

```
    myMac=hexMac(MAC)      # in Hexziffernfolge umwandeln
    print("STATION MAC: \t"+myMac+"\n") # ausgeben
    sleep(1)
    if not nic.isconnected():
      nic.connect(mySSID, myPass)
      print("Status: ", nic.isconnected())
      d.writeAt("WLAN connecting",0,1)
      points="..........."
      n=1
      while nic.status() != network.STAT_GOT_IP:
        print(".",end='')
        d.writeAt(points[0:n],0,2)
        n+=1
        sleep(1)
    print("\nStatus: ",connectStatus[nic.status()])
    d.clearAll()
    STAconf = nic.ifconfig()
    print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
          STAconf[1], "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
    print()
    d.writeAt(STAconf[0],0,0)
    d.writeAt(STAconf[1],0,1)
    d.writeAt(STAconf[2],0,2)
    return nic
```

Weil der ESP8266 bereits beim Einschalten eine Funkverbindung zu dem Router aufbaut, mit dem er schon einmal eine Verbindung hatte, erscheint kein Punkt. Folgender Test bestätigt das. Verbinden Sie einen solchen ESP8266 mit dem PC und starten Sie Thonny. Im Terminal geben Sie folgende Anweisungen ein.

>>> import network
>>> nic = network.WLAN(network.STA_IF)
>>> nic.active(True)
>>> nic.isconnected()
True

Das ist auch der Grund, warum der ESP8266 manchmal ständig neu bootet. Er versucht, den Router zu kontaktieren, wenn das nicht gelingt, macht er einen Neustart. Mitunter hilft es dann, webrepl, das Funkterminal, auszuschalten.

Nach dem Flashen der Firmware auf dem ESP8266:
>>> import webrepl_setup
> d fuer disable
Dann RST; Neustart!

Der ESP32 zeigt dieses merkwürdige Verhalten nicht. Deshalb erscheinen hier auch um die drei bis fünf Punkte.

Um das Rauschen der ADC-Werte zu verringern, ermittelt die Funktion **messen**() den Mittelwert von n Messungen. Der Rückgabewert ist 1, wenn der Messwert größer als der Grenzwert in **trigger** ist, sonst 0. Uns interessiert nicht der Quasi-Helligkeitswert, sondern nur ob das Licht an (1) oder aus (0) ist.

```
def messen(n):
    val=0
    for _ in range(n):
        val+=adc.read()
    val//=n
    return 1 if val > trigger else 0
```

Die Hauptschleife wird klarer, wenn man Jobs wie **messen**() oder Ereignishandler in Funktionen auslagert. Auch das Senden der Nachricht an WhatsApp ist deshalb als Funktion codiert.

```
def sendeNachricht(text):
    url="https://api.callmebot.com/whatsapp.php?phone="+\
         phone+"&text="+URLEncode(text)+"&apikey="+key
#     print(url)
    resp=requests.get(url)
    if resp.status_code == 200:
        print("Transfer OK")
    else:
        print("Transfer-Fehler")
    resp.close()
    gc.collect()
```

Alles an Information wird in die Variable **url** verpackt, Protokoll, Server, Rufnummer, der urlcodierte Textund und der API-Schlüssel. Zur Kontrolle kann das Ergebnis ausgedruckt werden. Dann schicken wir die URL mit der Methode GET auf die Reise. Das Attribut **status_code** des Response-Objekts **resp** sagt uns, ob der Transfer erfolgreich war oder nicht. In jedem Fall schließen wir den Socket und räumen den Speicher auf.

```
nic=connect2router()
sys.exit()
```

Die Verbindung wird aufgebaut. Während der Entwicklungsphase brechen wir an dieser Stelle das Programm ab. Wir haben jetzt eine funktionierende Netzwerkverbindung, außerdem sind die ganzen Objekte, Variablen und Funktionen deklariert. So können wir die einzelnen Komponenten händisch testen.

>>> messen(20)
1
Jetzt den LDR abdecken

>>> messen(20)
0

>>> sendeNachricht("Morgenstund ist aller Laster Anfang")
Transfer OK

Ein paar Sekunden später melden sich das Handy und die Windows-App.

Abbildung 9: Erste Nachricht vom ESP32

Wenn alles geklappt hat, kommentieren wir **sys.exit**() aus.

Der Wecker für den nächsten Scan wird auf 20 Millisekunden gestellt. Dann holen wir den Lichtwert, die Weckzeit für **warnen** stellen wir auf unendlich.

```
nextScan=TimeOutMs(20)
alt=messen(10)
warnen=TimeOut(0)
if alt==1:
    start=time()
    warnen=TimeOut(warnLevel)
```

Falls das Licht bereits an ist, merken wir uns den Zeitpunkt und stellen den Warn-Timer auf **warnlevel**. **alt** ist der Zustand der zurückliegenden Messung.

Dann geht es in die Hauptschleife. Wenn der Timer für die nächste Abtastung des LDR abgelaufen ist, holen wir uns den aktuellen Zustand. Abhängig vom alten und neuen Zustand können drei Situationen entstehen.

Das Licht war aus und ist jetzt an.
Wir stellen den Timer **warnen** auf **warnLevel** und merken uns die Zeit. Keine weitere Aktion.

Das Licht war und ist noch immer an.
Ist der Timer **warnen**() abgelaufen, dann wird es Zeit, eine Nachricht abzufeuern. Wir stellen den Timer **warnen** erneut auf **warnLevel**.

Das Licht war an und ist jetzt aus.
Wir nehmen erneut die Zeit und berechnen daraus die Einschaltdauer. Mit einer neuen Nachricht geben wir Entwarnung.

```
while 1:
    if nextScan():
        neu=messen(10)
        if alt==0 and neu==1:
            warnen=TimeOut(warnLevel)
            start=time()
        elif alt==1 and neu==1:
```

```
            if warnen():
                sendeNachricht("Licht ist seit {} s an.".\
                               format(warnLevel))
                warnen=TimeOut(warnLevel)
        elif alt==1 and neu==0:
            ende=time()
            dauer=ende-start
            if dauer > warnLevel:
                sendeNachricht("Licht ist nach {} s aus.".\
                                format(dauer))
```

In jedem Fall wird der neue Lichtwert auf den alten übertragen. Weil **nextScan**() abgelaufen war, stellen wir den Timer neu. Den print-Befehl kann man im Produktionsbetrieb löschen oder auskommentieren

Bleibt noch die obligatorische Tastenabfrage.

Zum Testen wird das Programm whatsapp.py neu gestartet. Jetzt müssen Nachrichten verschickt werden, wenn das Licht länger als **warnLevel** Sekunden an ist oder/und wenn das Licht ausgemacht wird.