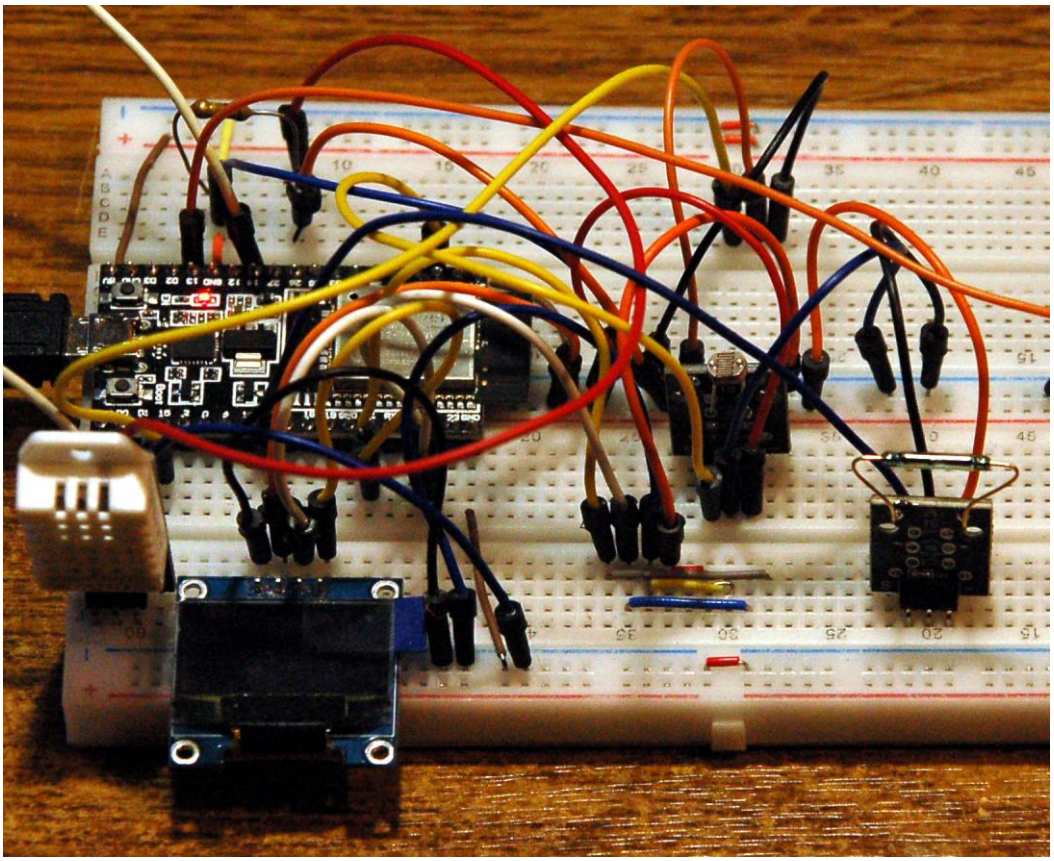

Aufbau der Steuereinheit

Diesen Beitrag gibt es auch als PDF-Dokument.

Beim Post zu If-This-Then-That hatten wir Daten vom ESP32 oder ESP8266 an IFTTT geschickt, die uns der Server dann als e-Mail zugesandt hat. Dann hatten wir uns die Messdaten von BME280 und BH1750 durch Thingspeak grafisch darstellen lassen. In dieser Episode wird unser ESP32 Kontakt mit einem Bot auf **Telegram** aufnehmen. Kein ESP8266? Nein, leider nicht. Warum? Weil das Programm und die Module recht niedlich sind, habe ich es als Erstes mit einem ESP8266 D1 mini probiert. Alles lief perfekt, bis der Kleine einen POST-Request an den Telegram-Bot absetzen sollte. Ich bekam permanent den OSError -40, egal was ich probierte. Im Web wurde ich zu dem Fehler auch nicht fündig. Nach einem vergeblichen Vormittag baute ich die Schaltung für einen ESP32 um. Sieh da - dasselbe Programm, dieselben Module und dieselben Baugruppen – alles schnurrte wie ein Uhrwerk. Irgendetwas im Kernel des ESP8266 muss wohl anders ticken wie beim ESP32, der tickt richtig.

Im Zusammenhang mit dem Telegram-Bot werden wir uns ein wenig mit JSON-Code beschäftigen (JSON = Java Script Objekt Notification). Es gibt starke Ähnlichkeiten mit MicroPython-Objekten. Außerdem bietet das MicroPython-Modul **Telegram_ext** zwei interessante Features, um eine Klasse zur Laufzeit um anwendungsspezifische Funktionen zu erweitern, ohne in den Programmtext der Klasse eingreifen zu müssen. Klingt interessant? Gut dann legen wir los mit einer neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Telegram und der ESP32

Mit Telegram erhalten wir die Möglichkeit, nicht nur Nachrichten auf dem PC oder dem Handy zu empfangen, sondern auch noch unseren ESP32 interaktiv zu steuern. Damit schalten wir eine LED, rufen Werte vom DHT22 ab und lassen uns ereignisgesteuerte Sensordaten senden, ohne vorher eine Anfrage zu starten. Wenn Sie also über keinen eigenen internettauglichen Webserver verfügen, dann ist dieser Post der Schlüssel für den weltweiten Zugriff auf Ihre ESP32-Einheit.

Hardware

Um den Zustand der Schaltung jederzeit auch direkt vor Ort einsehen zu können, habe ich dem ESP ein Display spendiert. Über eine Taste ist ein geordneter Abbruch des Programms möglich, falls zum Beispiel Aktoren sicher ausgeschaltet werden müssen, wie hier die LED.

1	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	KY-004 Taster Modul
1	KY-021 Magnet Schalter Mini Magnet Reed Modul Sensor oder 10x N/O Reed Schalter Magnetische Schalter 2 * 14mm Magnetischer Induktion Schalter für Arduino + 1 Widerstand 10kΩ
1	DHT22 AM2302 Temperatursensor und Luftfeuchtigkeitssensor
1	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
1	LED, zum Beispiel rot
1	Widerstand 330 Ω
1	Widerstand 10 kΩ
2	MB-102 Breadboard Steckbrett mit 830 Kontakten
diverse	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F evtl. auch 65Stk. Jumper Wire Kabel Steckbrücken für Breadboard
optional	Logic Analyzer

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

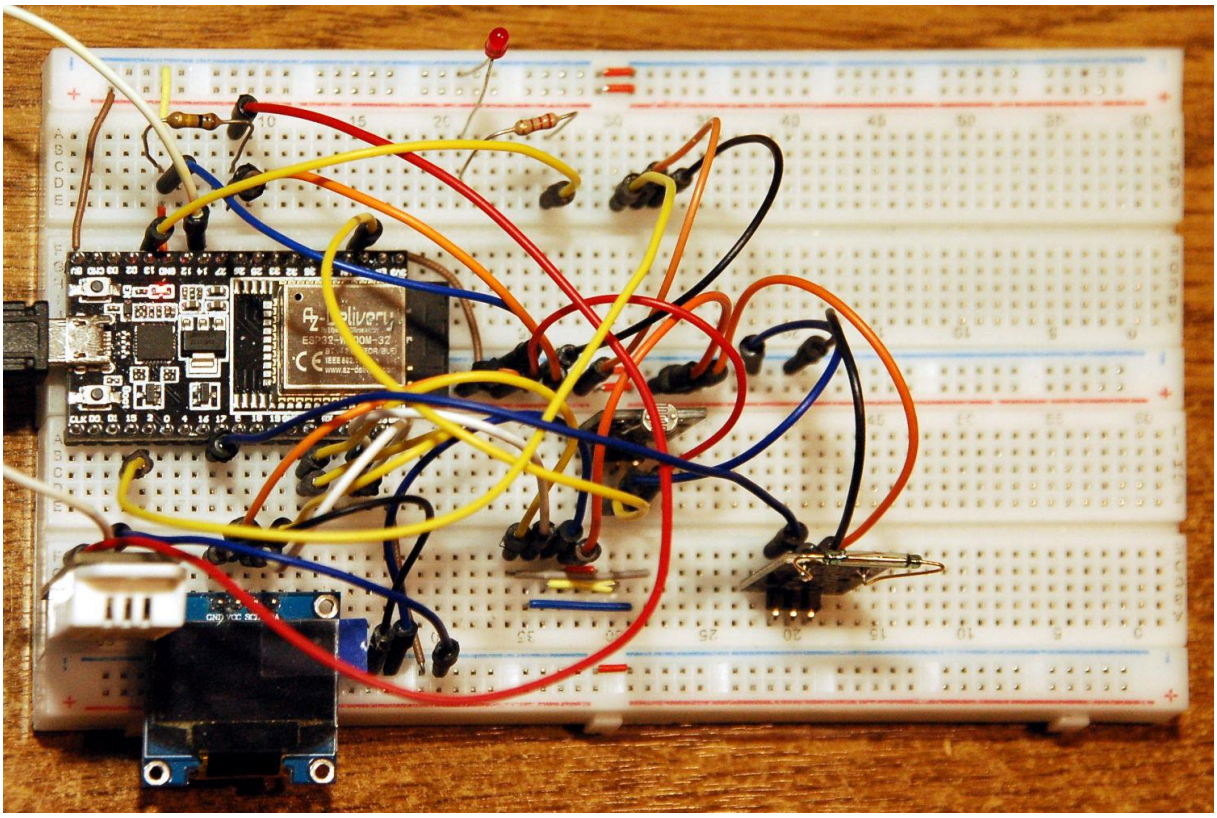

Abbildung 1: Zwei Breadboards für den ESP32

Eine besondere Erwähnung bedarf der Reed-Kontakt. Er soll am Kontakt S 3,3V liefern, wenn der Kontakt geschlossen ist. Deshalb müssen wir die Anschlüsse an dem Platinchen "+" und "-" so zuordnen wie in Abbildung 2 rechts. Falls Sie sich für die lose Form des Bauteils entschieden haben, es ist halt unauffälliger anzubringen, dann müssen Sie einen extra 10kΩ-Widerstand mit einbauen, auf dem Modul ist bereits einer vorhanden.

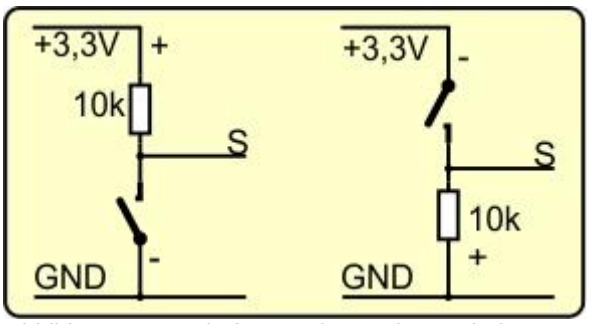

Abbildung 2: So wir der Reed-Kontakt geschaltet

Der Reed-Kontakt wird durch das Annähern eines Magneten zum Durchschalten gebracht. Seine federnden Kontaktstreifen sind ferromagnetisch. In einem Magnetfeld werden die Streifen also selbst magnetisch und ziehen sich gegenseitig an, wodurch der Kontakt geschlossen wird. Entfernt man das Magnetfeld, treiben die Federkräfte die Streifen wieder in ihre Ausgangslage zurück, der Kontakt öffnet.

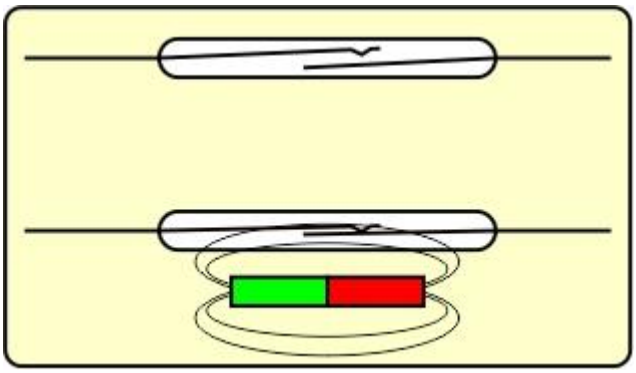

Abbildung 3: So arbeitet ein REED-Kontakt

Ein wichtiger Tipp! Seien Sie ganz vorsichtig, wenn Sie die Anschlussdrähte biegen möchten. Sie sind sehr starr. Biegt man sie zu nah am Glaskörper, bricht dieser aus und das Bauteil ist in den ewigen Jagdgründen.

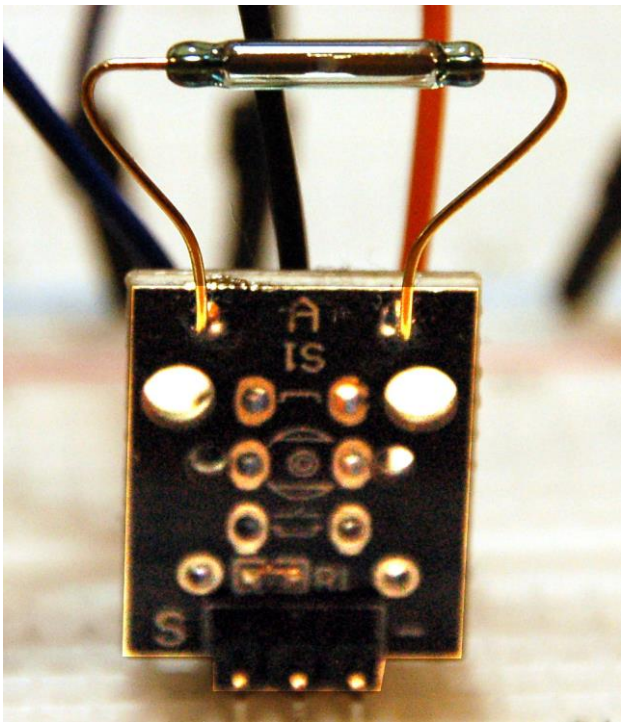

Abbildung 4: Reed-Kontakt

Hier kommt die Schaltung für das Projekt.

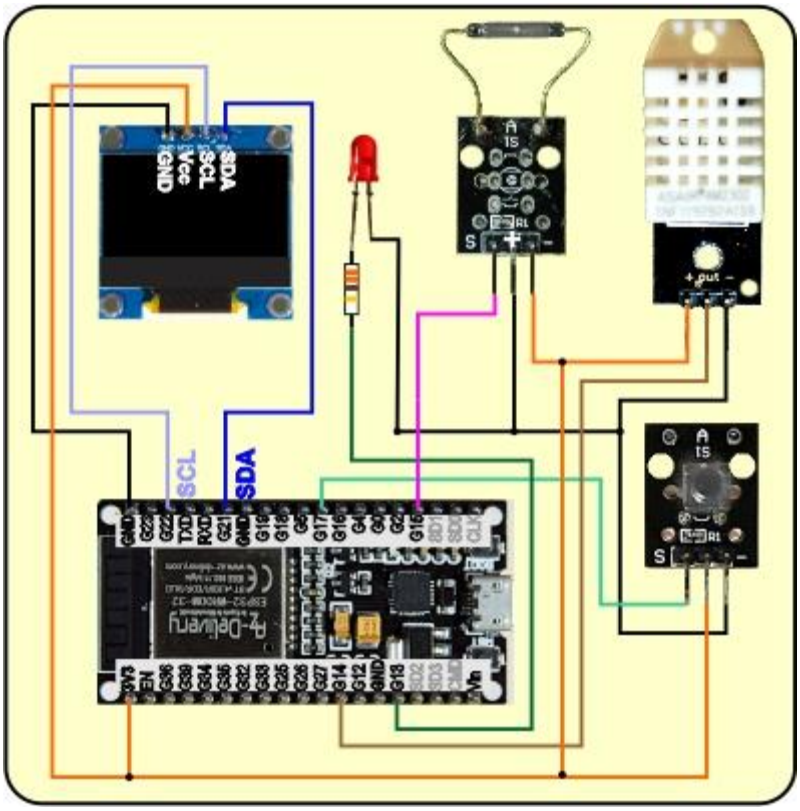

Abbildung 5: Schaltung mit Reed-Modul

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
telegram_ext.py API für den Telegram-Traffic
telegram.py Betriebssoftware des Projekts
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.

Telegram – Apps und Bot bauen

Einen Telegram-Account erzeugen

1. Den Telegram-Account legen Sie am besten mit dem Handy oder Tablet an. Dazu laden und installieren Sie die App vom Playstore.
2. Starten Sie die App und tappen Sie **Jetzt beginnen**.
3. Erteilen Sie Telegram die Berechtigung, Sie anrufen zu dürfen.
4. Geben Sie Ihre Telefonnummer ein und tappen Sie **Weiter**.
5. Erlauben Sie Telegram auf Ihre Telefonkontaktliste zuzugreifen.
6. Geben Sie Vor- und Nachname an und tappen Sie den **blauen Pfeil**.
7. Starten sie die App neu.

Telegram-Software für den PC

Für den PC gibt es eine Software, um Kontakt mit Telegram aufzunehmen. So können Sie sie herunterladen und installieren. Folgen Sie dem Link. Er führt Sie auf diese Seite.

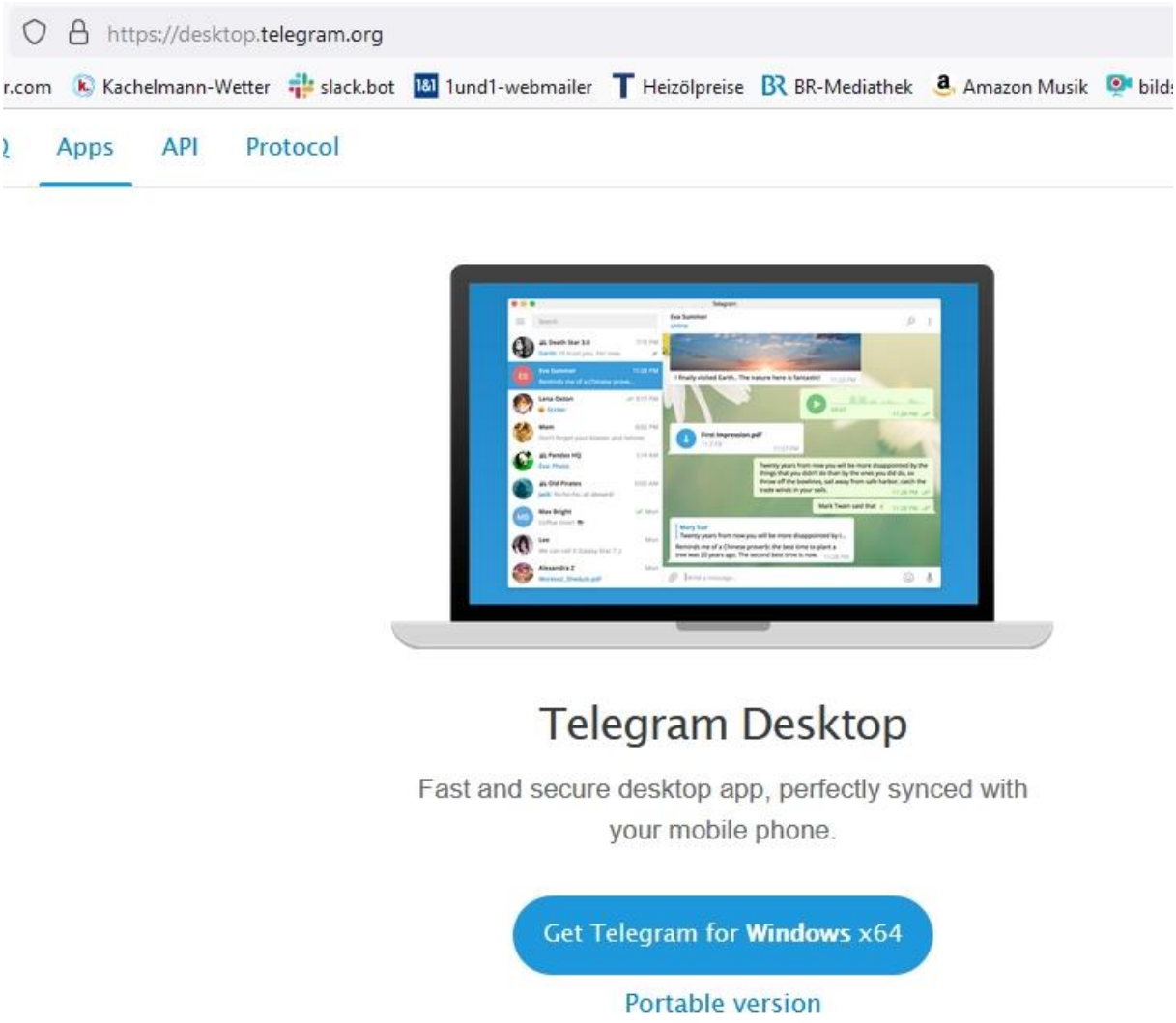

Abbildung 6: Telegram-App für den Desktop-PC_herunterladen

Ich habe die **Portable version** gewählt, die kann man auch gut auf einem Stick entpacken. Speichern Sie das zip-Archiv in einem beliebigen Verzeichnis und entpacken Sie es dort.

Abbildung 7: Portable-Version entpacken

Im Ordner Telegram finden Sie die Datei **Telegram.exe**. Erzeugen Sie davon eine Verknüpfung.

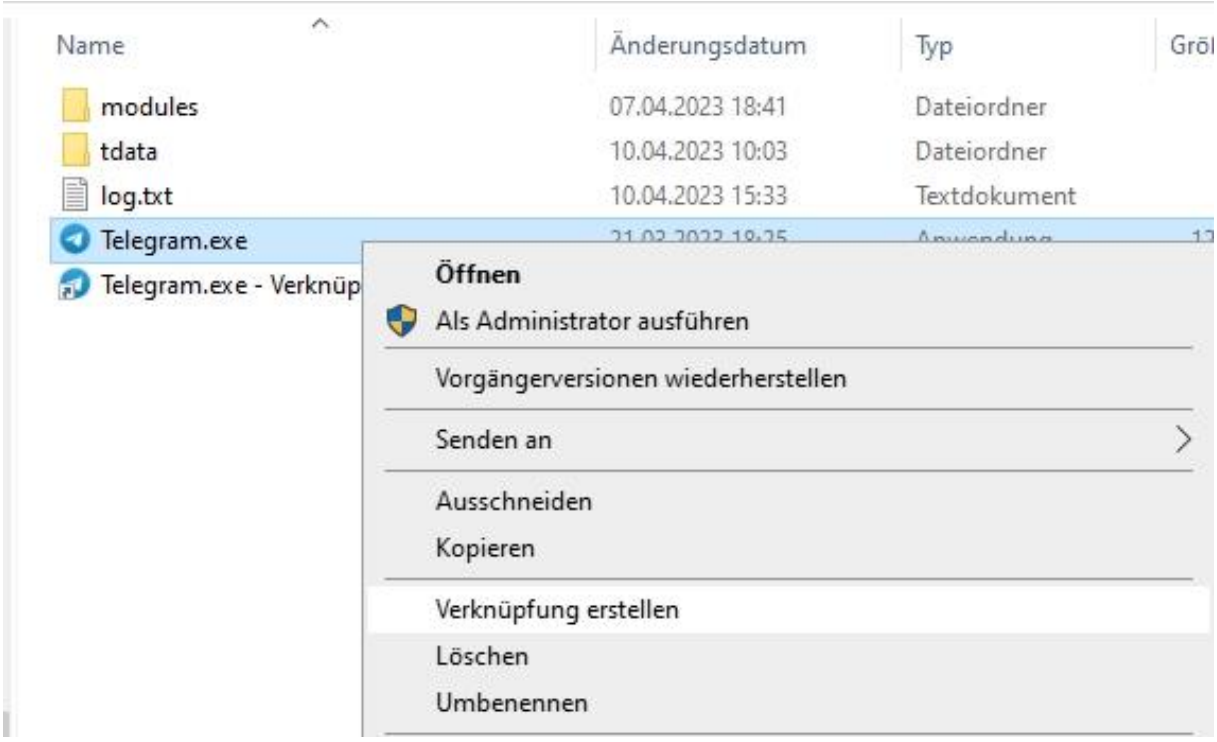

Abbildung 8: Von der Anwendung einen Link erzeugen

Die Verknüpfung ziehen Sie auf den Desktop.

Abbildung 9: Verknüpfung zur App auf dem Desktop

Beim Start bringt Windows eine Sicherheitswarnung, klicken Sie auf **Ausführen**.

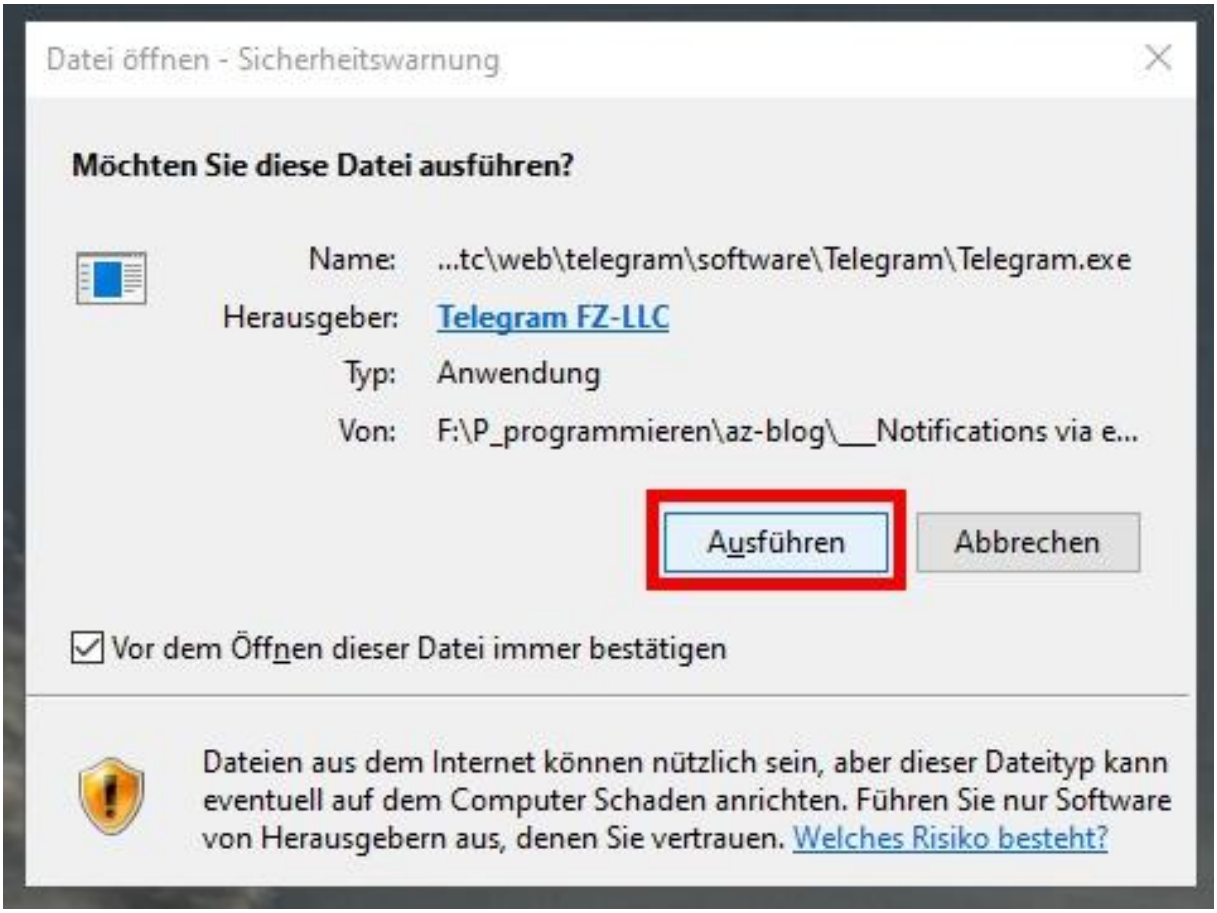

Abbildung 10: Nach dem Start

Wir bauen einen Bot

Im Suchfeld links oben im Startfenster geben wir **BotFather** ein und klicken auf den ersten Listeneintrag.

Abbildung 11: Suche botfather

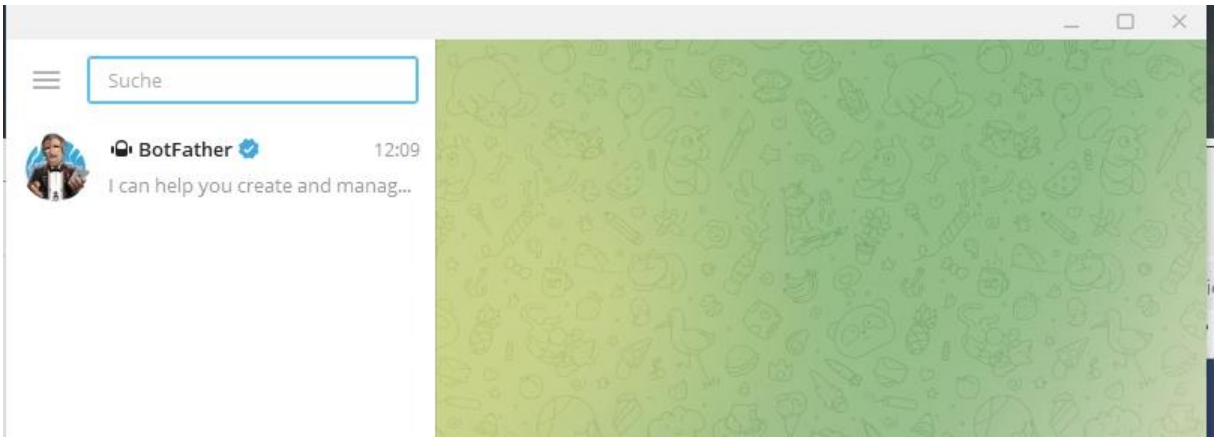

Abbildung 12: botfather

Mit Klick auf **/newbot** geht es weiter.

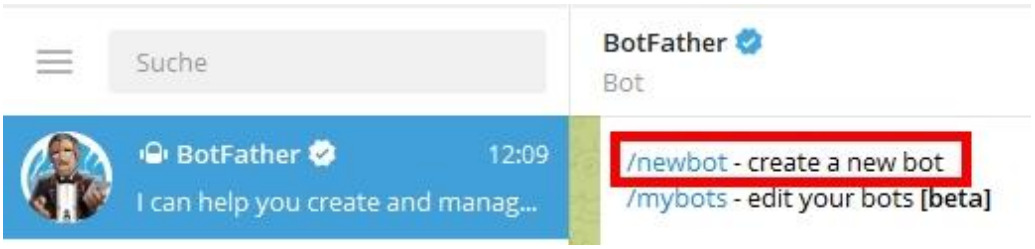

Abbildung 13: Neuen Bot erzeugen

Wir geben in der Eingabezeile den Namen für den neuen Bot ein und schicken die Nachricht mit dem kleinen blauen Dreieck unten rechts an botfather ab.

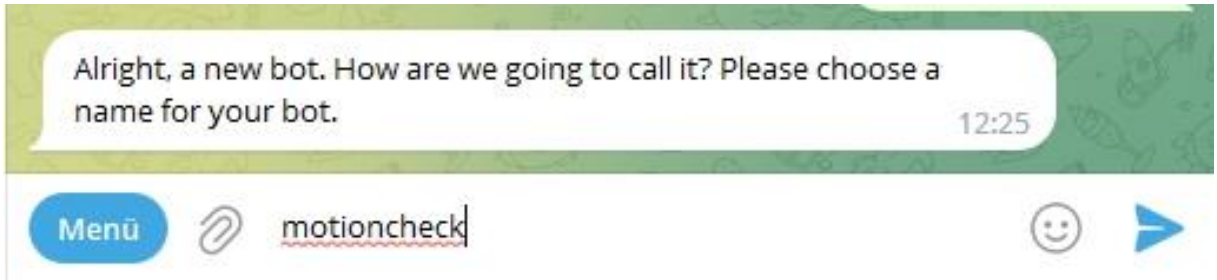

Abbildung 14: Bot benennen

Der Username für den Bot kann beliebig gewählt werden, muss aber auf bot enden.

Abbildung 15: Benutzername für den Bot festlegen

Nach dem Abschicken der Nachricht, Bekommen Sie das Token für den API-Zugriff. Kopieren Sie die Zeichenfolge und legen Sie sie an einer sicheren Position ab, wir brauchen sie später.

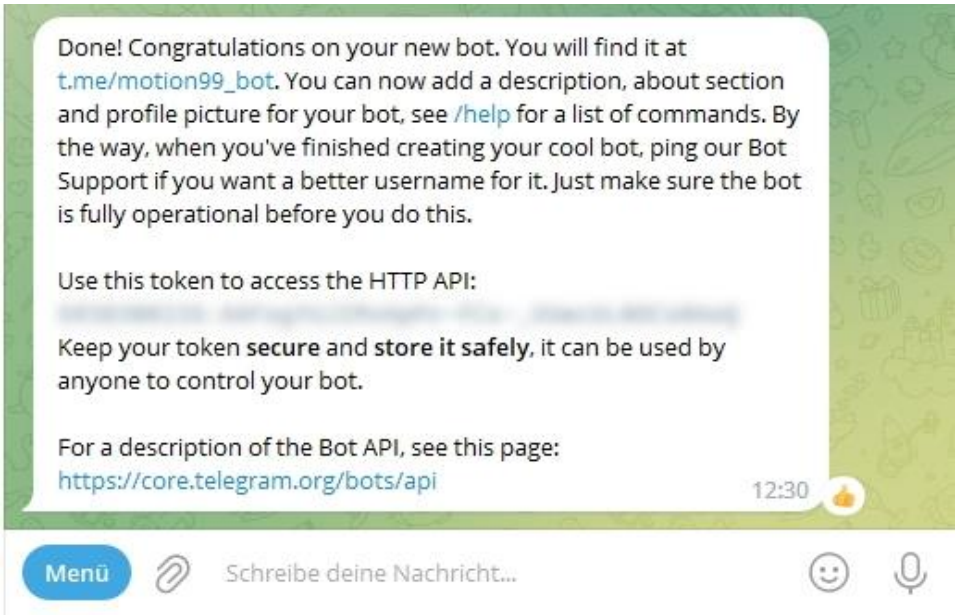

Abbildung 16: Neuer Bot ist angelegt

Für den Zugriff brauchen Sie ferner die Bot-ID. Sie bekommen diese über den IDBot, den wir zur Fahndung ausschreiben.

Abbildung 17: IDBot suchen

Klicken Sie auf den Listeneintrag und dann auf Starten.

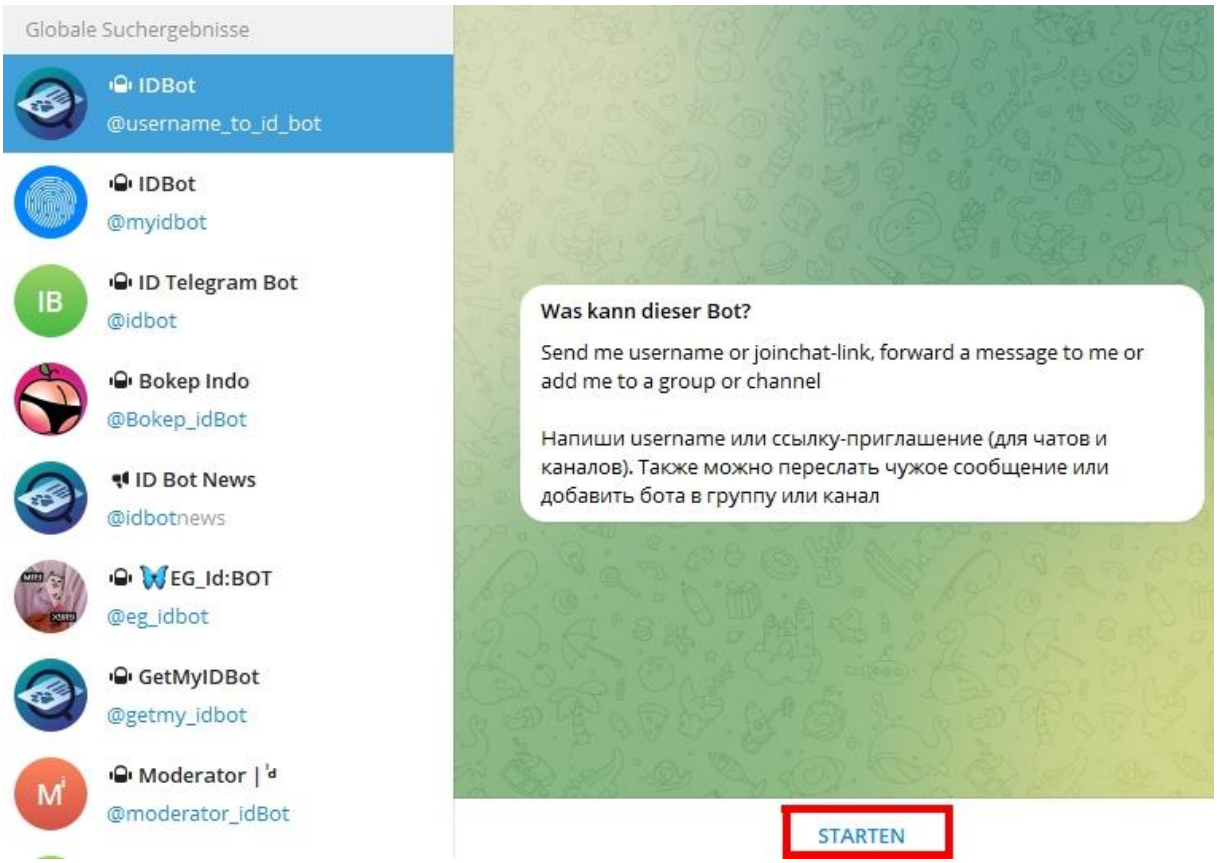

Abbildung 18:IDBot starten

Abbildung 19: User-ID als Antwort

Auch die ID benötigen Sie später.

Sie können nun über den neuen Bot einen ersten Kontakt zum ESP32 aufnehmen, wenn darauf die Anwendung für Ihr Projekt läuft. Damit das läuft, sprechen wir jetzt das Programm dazu durch.

Abbildung 20: Schalten und Werte abrufen

Das Chat-Programm

Für das Programm habe ich das Modul **utelegram.py** benutzt, das ich auf GitHub gefunden habe. Für meine Zwecke habe ich zwei Erweiterungen hinzugefügt und die Methode **listen**() etwas aufgemufft.

```
    def setCallback (self, f):
        self.cb = f

    def setExit(self,f):
        self.beenden = f

    def listen(self):
        while True:
            self.read_once()
            if self.cb is not None:
                self.cb()
            if self.beenden is not None:
                self.beenden()
            time.sleep(self.sleep_btw_updates)
            gc.collect()
```

In Python kann man die Referenz auf eine Funktion weitergeben. Diesen Umstand nutzt man bei Closures. Aber auch hier kann man eine tolle Sache damit anstellen.

Üblicherweise läuft die Hauptschleife im Hauptprogramm, wo man sie nach persönlichen Bedürfnissen gestalten kann. Unsere Hauptschleife ist aber die Methode **listen**() im Modul utelegram.py. Wir könnten unsere Wünsche natürlich umsetzen, indem wir den Code des Moduls verändern. Das müssten wir aber jedes Mal erneut tun, wenn wir das Modul für einen anderen Zweck einsetzen wollen.

Es gibt einen einfacheren Weg, um das Verhalten der Klasse **utelegram.ubot** zu verändern, ohne den Klassen-Code zu verändern. Die Klasse kann das in zweierlei Formen, die beide denselben Hintergrund haben, Callback-Funktionen. Schauen wir uns zuerst die einfachere, statische Version an.

```
    def setCallback (self, f):
        self.cb = f
```

Ich definiere hier eine Methode **setCallback**(), die die Referenz auf eine Funktion des Hauptprogramms als Argument nimmt und dem Attribut **cb** zuweist. Damit versetze ich **cb** in die Lage als Funktion aufrufbar zu sein und den Code der referenzierten Funktion im Hauptprogramm auszuführen. Weil **cb** innerhalb **ubot** deklariert ist, kann ich die Funktion überall in der Klasse verwenden. Das tue ich in der **listen**-Schleife.

```
            if self.cb is not None:
                self.cb()
            if self.beenden is not None:
                self.beenden()
```

Beim Programmstart weiß **listen**() aber noch nicht, was ausgeführt werden soll. Daher belege ich im Konstruktor **cb** und **beenden** mit **None** vor. In **listen**() passiert nichts, solange diese Belegung aktiv ist.

```
        self.cb = None
        self.beenden=None
```

Im Hauptprogramm übergebe ich schließlich die Namen der im Hauptprogramm deklarierten Funktionen.

```
    bot.setCallback(warnDoorTemp)
    bor.setExit(cancelProgram)
```

Wenn ich andere Funktionen ausführen lassen möchte, oder wenn Änderungen am Code der beiden Funktionen nötig sind, übergebe ich einfach nur die neuen Namen, oder ädere eben den Code. Aber ich muss nicht das Modul **ubot** neu hochladen, sondern nur das Hauptprogramm neu starten. Und ich kann ubot ohne Veränderung für neue Projekte einsetzen.

Die dynamische Methode, Callback-Funktionen zu nutzen bedient sich eines Dictionaries. Als Schlüssel wird der Name einer Aktion gespeichert und als Wert der Name einer Funktion im Hauptprogramm. Dadurch können wir Befehlscode, den wir per Telegram an den ESP32 gesendet haben, in der Empfangsmethode **read_once**() parsen und die entsprechende Routine im Hauptprogramm ausführen lassen. Das macht die Methode **message_handler**(). Brauchen wir einen neuen Befehl, hängen wir den Namen und die auszuführende Funktion einfach an das Dictionary an. Genial, oder?

```
    bot.register('/values', sendValues)
    bot.register('/on',switchOn)
    bot.register('/off',switchOff)
    bot.set_default_handler(get_message)
```

Durch dieses Vorgehen, können wir das Modul **utelegram**.py jetzt sogar compilieren, denn wir müssen ja nichts mehr daran ändern. Das spart Platz im Flash und schützt unseren Code.

Das MicroPython-Programm für den Bot

Nur drei Module in der Importliste sind externe Dateien: **timeout.py**, **telegram_ext.py** und **oled.py**. Letzteres Modul importiert **ssd1306.py**. Diese vier Dateien müssen also zum ESP32 hochgeladen werden.

```
from timeout import *
from time import sleep
from machine import Pin, SoftI2C
from oled import OLED
import dht
import network
import Telegram_ext
import sys
```

Für SSID, Passwort, Token und PID setzen Sie bitte Ihre Daten ein.

```
mySSID = 'EMPIRE_OF_ANTS'; myPass = "nightingale"
token = 'here goes your token'
PID = "here goes your ID"
```

Der folgende Block ermittelt automatisch die korrekten Anschlüsse für den I2C-Bus, abhängig vom Controllertyp und legt ein I2C-Objekt an.

```
if sys.platform == "esp8266":
    i2c=SoftI2C(scl=Pin(5),sda=Pin(4))
elif sys.platform == "esp32":
    i2c=SoftI2C(scl=Pin(22),sda=Pin(21))
else:
    raise RuntimeError("Unknown Port")
```

Das Bus-Objekt übergeben wir an den Konstruktor der OLED-Klasse, stellen die Helligkeit ein und geben die Überschrift aus.

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

Dann instanziieren wir das DHT22-Objekt, Eingänge für den Reed-Kontakt und die Taste und einen Ausgang für die LED. Damit nicht ununterbrochen Warnungen zur Temperaturüberschreitung oder offenstehenden Tür eintreffen definieren wir eine Auszeit von zunächst 10 Sekunden. Den zugehörigen Software-Timer stellen wir auf 1 Sekunde, damit die Überwachung sofort startet.

```
amb=dht.DHT22(Pin(14)) # D5
reed=Pin(15,Pin.IN)  # D3
led=Pin(13,Pin.OUT,value=0) # D7
taste=Pin(17,Pin.IN, Pin. PULL_UP)
deadTime=10 # Sekunden Totzeit zwischen Warnungen
sendNow=TimeOut(1)
```

Das Dictionary **connectStatus** übersetzt die Nummerncodes, die wir von **nic.status**() erhalten, in Klartext.

```
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",
    }
```

Von der Funktion **hexMac**() erfahren wir die MAC-Adresse des Station-Interfaces. Diese Adresse muss der WLAN-Router kennen, damit er dem ESP32 Zugang gewährt.

```
def hexMac(byteMac):
  """
  Die Funktion hexMAC nimmt die MAC-Adresse im Bytecode
  entgegen und bildet daraus einen String fuer die Rueckgabe
  """
  macString =""
  for i in range(0,len(byteMac)):     # Fuer alle Bytewerte
    macString += hex(byteMac[i])[2:]  # ab Position 2 bis Ende
    if i <len(byteMac)-1 :            # Trennzeichen
      macString +="-"
  return macString
```

Jetzt kommen die Deklarationen der Callback-Routinen. Sie werden von den Methoden **listen**() und **message_handler**() aus der Klasse **ubot** aufgerufen (indirect call).

get_message() wird von ubot.**set_default_handler**() gerufen, wenn Telegram eine Nachricht erhält, die keiner spezifischen Kennung entspricht und daher keiner speziellen Aktion zugeordnet werden kann. In **message** wird der empfangene Nachrichtenblock in JSON-Notierung übergeben. Die beiden print-Anweisungen dienen der Darstellung während der Entwicklungszeit und können im Produktionsbetrieb gelöscht oder auskommentiert werden. Die Textnachricht wird aus dem **message**-Block herausgefiltert und in Großbuchstaben an den PC und ans Handy gesendet.

```
def get_message(message):
    print("Message:\n",message,"\n")
    print("Nachricht:\n",message['message']['text'])
    bot.send(message['message']['chat']['id'],
             message['message']['text'].upper())
```

Die Syntax der Methode **send**() ist im Grunde sehr simpel, wirkt aber durch die Struktur des **message**-Blocks mächtig kompliziert.

bot.send(Telegram-ID, Textnachricht)

Um Licht ins Dunkel zu bringen, schauen wir uns so einen Block näher an. Die JavaScript Object Notation, kurz JSON, erinnert stark an die MicroPython-Objekte Liste, Dictionary und String.

JSON-Element-Typen
[ ] kennzeichnet ein Array (Liste in MicroPython)
{ } kennzeichnet ein Objekt (in MicroPython ein Dictionary)
" " oder ' ' begrenzt eine Zeichenkette (in MicroPython eine Stringkonstante)
Zahlen enthalten die Ziffern 0 bis 9, "-"-Zeichen und Dezimalpunkt "."
true und **false** sind boolsche Werte (in MicroPython True und False)
null ist der Nullwert (in MicroPython None)

Die Elemente von Arrays und Objekten sind durch Kommas getrennt. Dem Komma und dem Doppelpunkt in den Name-Wert-Paaren von Objekten folgt ein Leerzeichen.

Die Methode ubot. **read_once**() liest vorhandene Nachrichtenblöcke in ein Array ein und übergibt den letzten Eintrag an ubot.**message_handler**().

```
messages = self.read_messages()
```
…
```
self.message_handler(messages[-1])
```

Schauen wir uns nun **messages** und einen **message**-Block näher an.

{'update_id': 209726430, 'message': {'message_id': 140, 'from': {'username': 'user16', 'is_bot': False, 'last_name': 'Mustermann', 'language_code': 'de', 'id': 1234567890, 'first_name': 'Max'}, 'text': '/Gru\xdf vom ESP32', 'date': 1681825527, 'chat': {'first_name': 'Max', 'last_name': 'Mustermann', 'type': 'private', 'id': 1234567890, 'username': 'user16'}}}

Abbildung 21:Ein message-Objekt

Sie meinen, das ist auch nicht weniger verwirrend? Da liegen Sie genau richtig. Ich habe mir mal die Mühe gemacht, das Ganze in eine strukturierte Form zu bringen. Jetzt ist alles ganz einfach. Das Array **messages** enthält eine **message**, die mit dem Name-Wert-Paar **'update_id': 'xxxxxxxxxx'** beginnt. Der Wert des zweiten Paars mit dem Namen **message** ist ein Objekt das Objekte, einfache Name-Wert-Paare und ein Array enthält.

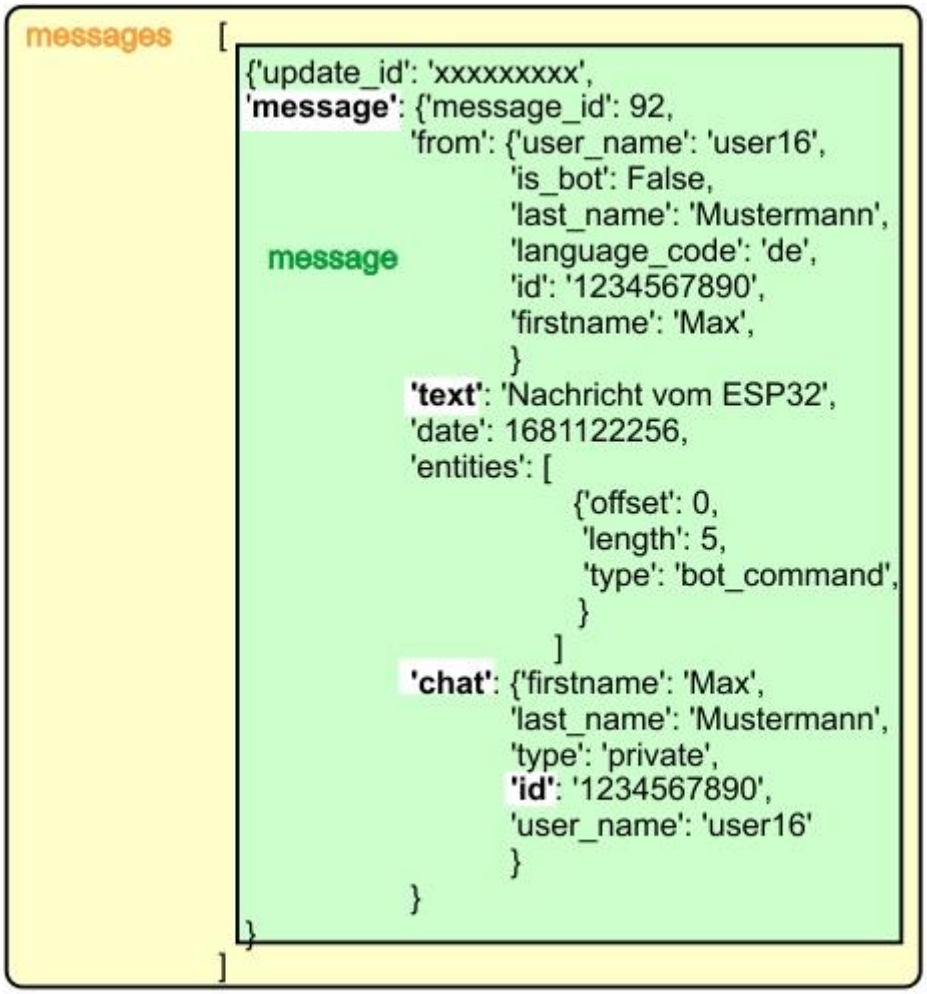

Abbildung 22: Hierarchischer Aufbau von messages im JSON-Format

Dann ist message['message']['chat']['id'] '1234567890' und message['message']['text'] ist 'Nachricht vom ESP32'. Die ID hätte man auch so abrufen können: message['message']['from']['id']. Jetzt stimmen Sie mir sicher zu, dass es einfach ist, Informationen aus dem Block zu bekommen.

Wenn wir an unseren Bot das Kommando **/on** über die Telegram-App senden, wird die Funktion **switchOn**() ausgeführt. Wir lassen uns den Wert von message['message']['text'] in Großbuchstaben wandeln, im Terminal ausgeben und, nachdem wir die LED eingeschaltet haben, zurücksenden.

```
def switchOn(message):
    print(message['message']['text'])
    code=message['message']['text'].upper()
    print(code)
    led.on()
    bot.send(message['message']['chat']['id'], code)
```

Analog verhält sich die Funktion **switchOff**().

```
def switchOff(message):
    print(message['message']['text'])
    code=message['message']['text'].upper()
    print(code)
    led.off()
    bot.send(message['message']['chat']['id'], code)
```

cancelProgram() wird von **ubot.listen**() aufgerufen, um zu testen, ob die Abbruchtaste gedrückt ist und dann gegebenenfalls das Programm zu beenden. **listen**() selbst kann die Taste nicht abfragen, weil die Methode vom Vorhandensein des Tastenobjekts nicht den blassesten Schimmer hat. Aber **listen**() weiß, dass eine Funktion **bot.beenden**() zyklisch aufgerufen werden muss und **bot.beenden** kennt die Hausnummer von **cancelProgram**() und weiß daher, wo angeklopft werden muss. **cancelProgram**() wiederum kennt das Objekt **taste** sehr gut und kann den Zustand daher abfragen. Kennen Sie die Story "Der Bauer schickt den Jockel aus…"?

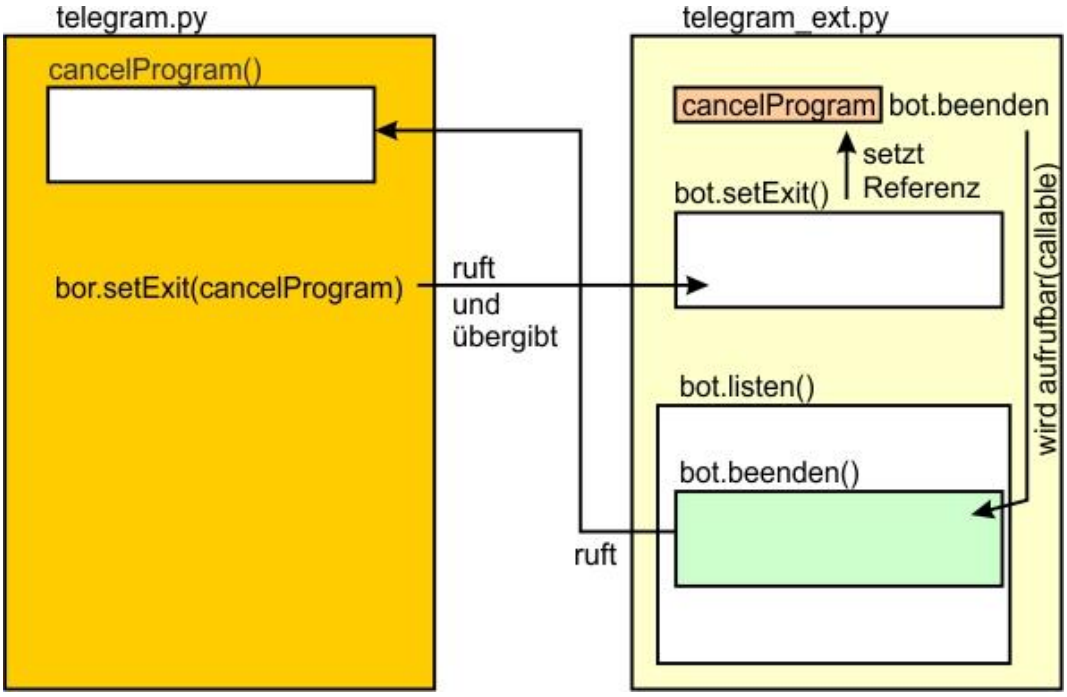

Abbildung 23: Callback Ablauf

Die Funktion **warnDoorTemp**() ist wieder eine Spur delikater. Warum ist die Funktion **sendNow** als global ausgewiesen und **taste.value** in **cancelProgram**() nicht? Ganz einfach, weil wir eine Fehlermeldung bekämen, wenn wir sendNow nicht als global deklarieren:

Traceback (most recent call last):
File "<stdin>", line 174, in <module>
File "telegram_ext.py", line 69, in listen
File "<stdin>", line 97, in warnDoorTemp
NameError: local variable referenced before assignment

```
def warnDoorTemp():
    global sendNow
    amb.measure()
    temp=amb.temperature()
    message=""
    if sendNow():
        if reed.value()==0:
            message="Door open! "
        if temp > 20:
            message+="Temperatur zu hoch: {:.2f}°C".\
                     format(temp)
        if message != "":
            bot.send(PID,message)
        sendNow=TimeOut(deadTime)
```

taste.value() wird nur referenziert aber nicht in der Funktion neu deklariert. In Zeile 6 rufen wir **sendNow**() auf, aber in der letzten Zeile deklarieren wir **sendNow**() durch den Aufruf von **TimeOut**() neu. Der MicroPython-Interpreter stuft damit **sendNow** als lokale Variable ein und erkennt, dass diese referenziert wird, bevor ihr die Referenz auf die Closure **compare**() zugewiesen wird, die in **TimeOut**() deklariert wird. Dadurch, dass wir **sendNow** als globales Objekt ausweisen, machen wir die Abfrage möglich, bevor wir einen neuen Wert zuweisen. Der Zugriff erfolgt jetzt auf die globale Referenz von **sendNow**. Der Interpreter stößt sich nicht mehr an der nachträglichen Wertzuweisung.

```
def TimeOut(t):
    start=time()

    def compare():
        nonlocal start
        if t==0:
            return False
        else:
            return int(time()-start) >= t

    return compare
```

In **warnDoorTemp**() holen wir den Temperaturwert und bereiten den Ergebnisstring vor. Falls der Timer abgelaufen ist, prüfen wir den Zustand des Reed-Kontakts und den Wert der Temperatur. Nur wenn einer der beiden Events getriggert wurde, wird eine Nachricht an unseren Telegram-Account gesendet. Danach stellen wir den Wecker neu.

```
def sendValues(message):
    print(message['message']['text'])
    code=message['message']['text'].upper()
    print(code)
    amb.measure()
    temp=amb.temperature()
    hum=amb.humidity()
    d.clearFT(0,1,15,2,False)
    d.writeAt("Temp: {:.2f}".format(temp),0,1,False)
    d.writeAt("rHum: {:.2f}".format(hum),0,2,False)
    code="Temperatur: {:.2f} *C\n".format(temp)
    code+="Rel. Feuchte: {:.2f}%\n".format(hum)
    bot.send(message['message']['chat']['id'], code)
```

sendValues() wird gerufen, wenn wir **/values** an den Bot senden. Messung in Auftrag geben, Temperatur und relative Luftfeuchte abrufen und im Display ausgeben. In **code** wird die Nachricht zusammengestellt und schließlich versandt.

Nun stellen wir noch die Verbindung zum WLAN-Router her.

```
nic=network.WLAN(network.AP_IF)
nic.active(False)

nic = network.WLAN(network.STA_IF)  # erzeugt WiFi-Objekt nic
nic.active(True)                    # nic einschalten
sleep(1)

MAC = nic.config('mac') # binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)    # in eine Hexziffernfolge umgewandelt
print("STATION MAC: \t"+myMac+"\n") # ausgeben

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='')
d.writeAt(STAconf[0],0,0)
d.writeAt(STAconf[1],0,1)
d.writeAt(STAconf[2],0,2)
sleep(3)
d.clearAll()
d.writeAt("AMBIANT DATA",2,0)
```

Es fehlt nur noch die Instanziierung des Bots und die Registrierung der Callback-Funktionen, falls eine Verbindung zum Router zustande gekommen ist. **register**() baut das Dictionary ubot.**commands** auf. **set_default_handler**(), **setCallback**() und **setExit**() machen die entsprechenden Funktionen in ubot bekannt. Final wird mit bot.**listen**() die Empfangsschleife betreten.

```
if nic.isconnected():
    bot = telegram_ext.ubot(token)
    bot.register('/values', sendValues)
    bot.register('/on',switchOn)
    bot.register('/off',switchOff)
    bot.set_default_handler(get_message)
    bot.setCallback(warnDoorTemp)
    bot.setExit(cancelProgram)
    print('BOT LISTENING')
    bot.listen()
else:
    print('NOT CONNECTED - aborting')
```

Das Programm wird beendet, falls keine Verbindung zum Router zustande gekommen ist.

Wrapping up

Hier noch einmal alle Schritte, um das Projekt zum Laufen zu bringen.

1. Schaltung aufbauen
2. Telegram-App aufs Handy laden und installieren
3. Telegram-Account über das Handy einrichten
4. Die Telegram-App für den Desktop-PC herunterladen und entpacken
5. Botfather aufrufen und einen Bot einrichten
6. Die Dienstprogramme herunterladen und zum ESP32 hochladen
   ssd1306.py
   oled.py
   telegram_ext.py
   timeout.py
7. Telegram-App starten (PC und oder Handy)
8. Einen Magneten an den Reed-Kontakt heften
9. Das Programm telegram.py im Editor starten

10. In der Eingabezeit der Telegram-App eines der Kommandos eingeben und abschicken
11. Die Reaktionen des ESP32 kontrollieren (LED)
12. Magnet entfernen, kommt die Nachricht vom Bot?
13. Kommt die Nachricht, wenn die Temperatur den Grenzwert überschreitet?
14. Wenn alles funktioniert, das Programm telegram.py als main.py im Flash des ESP32 speichern.
15. ESP32 neu starten

Bis jetzt sind aller guten Dinge drei, in der nächsten Folge zum Thema Messaging behandeln wir abschließend den Dienst **Whatsapp**.

Bis dann, bleiben Sie dran!