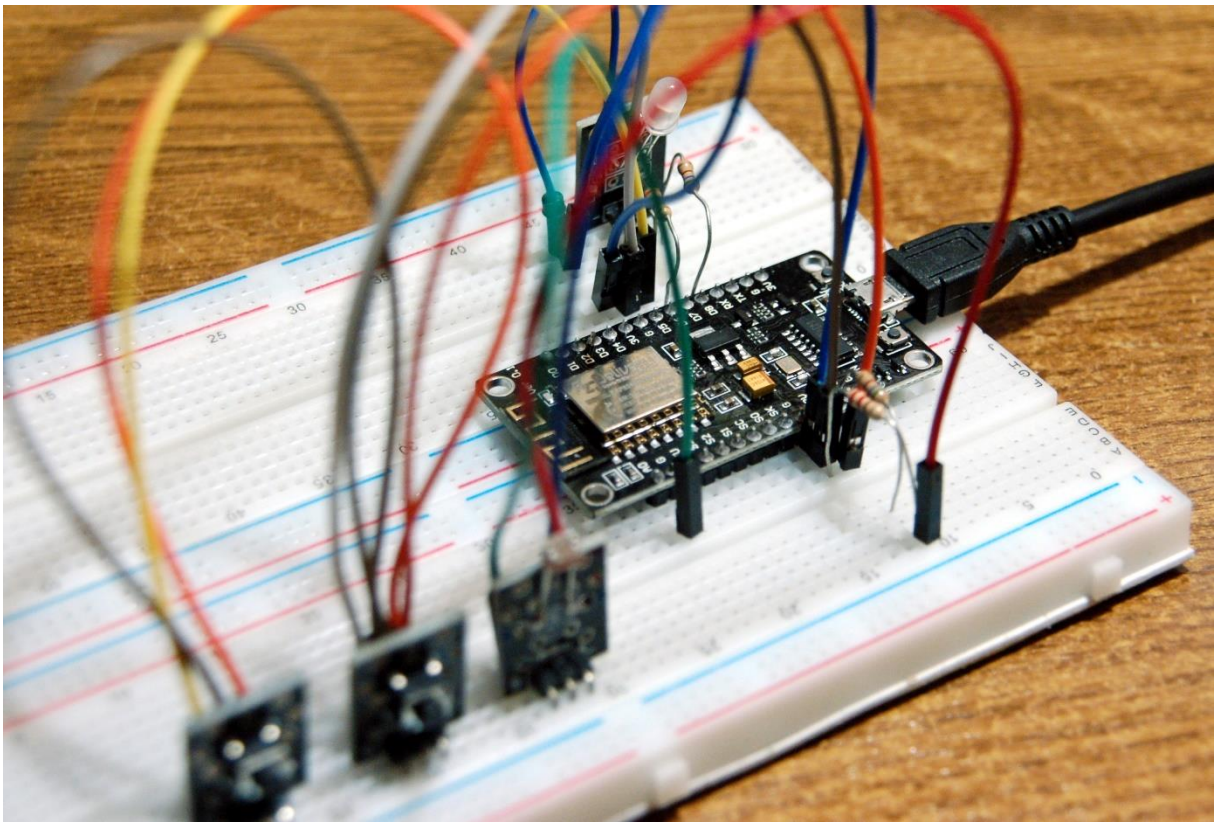


MicroPython mit dem ESP32 – Teil 3

Module und Klassen

Herzlich willkommen zum dritten Teil von MicroPython mit dem ESP32. Dieses Mal werde ich Ihnen den Einsatz weiterer Hardware vorstellen und vor allem das Thema Module und Klassen näher beleuchten; eine sehr interessante Materie, wie ich meine.

An Hardware kommt folgendes Material zum Einsatz, einen Teil davon haben Sie ja bereits, falls Sie die ersten beiden Blogbeiträge gelesen und bearbeitet haben.

1	ESP32 NodeMCU Module WLAN WiFi Development Board oder
1	ESP-32 Dev Kit C V4 oder
1	ESP32 D1 R32 oder
1	ESP8266 LoLin V3 oder ähnlich
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel für Arduino und Raspberry Pi oder
1	0,96 Zoll OLED I2C Display 128 x 64 Pixel für Arduino und Raspberry Pi
1	Widerstand 680 Ohm
1	Widerstand 220 Ohm
1	KY-011 Bi-Color LED Modul 5mm und
2	Widerstand 680 Ohm für LED oder
1	KY-009 RGB LED SMD Modul und
3	Widerstand 1,0kOhm für LED

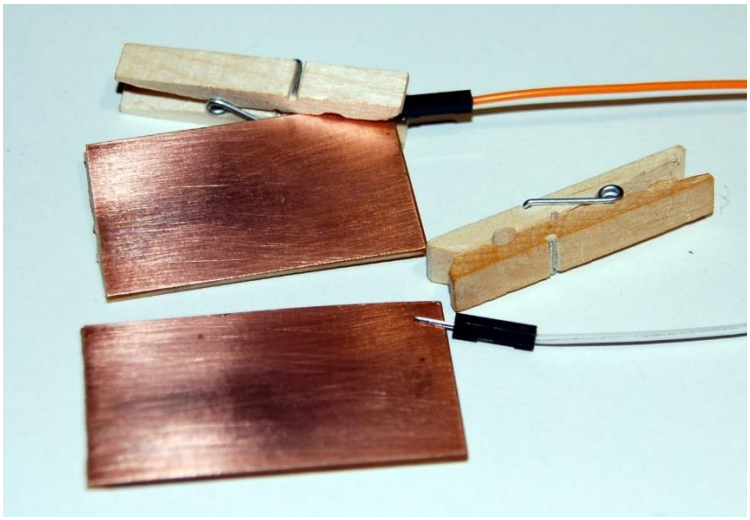
2	KY-004 Taster Modul
1	AZ-Delivery-Keypad-ttp224-1x4
1	KY-018 Foto LDR Widerstand
2	Mini Breadboard 400 Pin mit 4 Stromschienen für Arduino und Jumper Kabel
1	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F
2	Blech ca. 20 x 20 mm (nicht Aluminium!) oder Platinenreste
	einige Steckstifte 0,6x0,6x12mm

Ideal zum Experimentieren ist das **35 in 1 Arduino Sensorenkit Modulkit und Zubehörkit für Arduino und andere Mikrocontroller**, das ich mir vor einiger Zeit besorgt habe. Darin sind neben den oben genannten KY-Modulen noch eine Menge weiterer interessanter Sensoren.

An Werkzeug brauchen Sie:

- Lötkolben
- Lötzinn

An die Blech- oder Platinenstücke müssen Steckstifte für den Anschluss von Jumperkabeln gelötet werden. Deshalb sollte hier auch kein Aluminium hergenommen werden, weil sich das nur mit einem Trick verlöten lässt, der den Einsatz von Chemie nötig macht. Also Kupfer, Messing oder blankes Weißblech nehmen. Falls kein Lötwerkzeug griffbereit ist, können Sie sich mit zwei Wäscheklammern behelfen, um die Stifte der Jumperkabel auf den Blechteilen zu befestigen. Für die ersten Versuche reicht das.

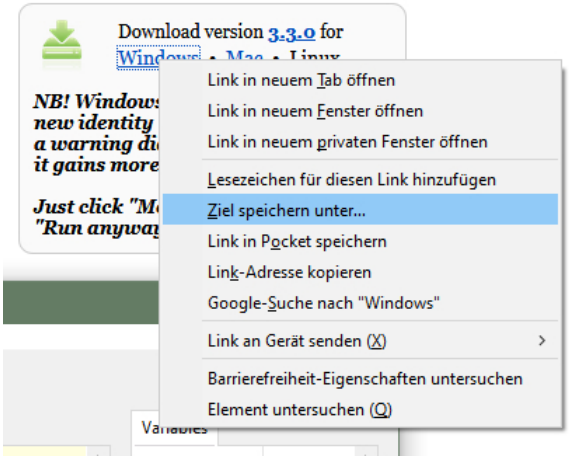


Bevor wir uns mit der Hardware und deren Programmierung befassen, stelle ich Ihnen mit Thonny eine weitere IDE für MicroPython vor. Nachdem der Schwerpunkt in diesem Beitrag neben der Einbindung von Hardware auf dem Erstellen und dem Einsatz von Modulen liegen soll, starten wir gleich danach mit ein paar Experimenten zu diesem Thema.

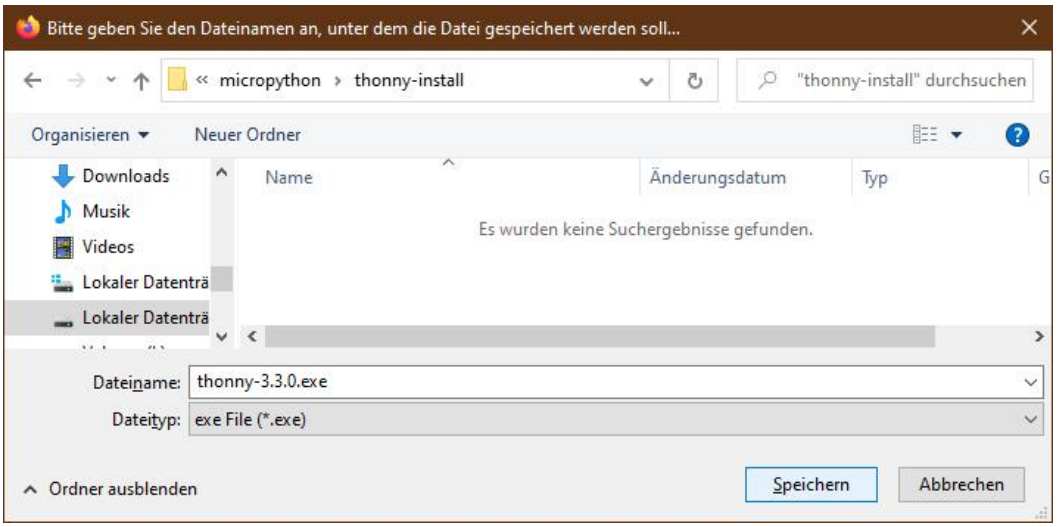
So installieren und benutzen Sie Thonny

Thonny ist eine weitere IDE zum Programmieren in MicroPython. Es wird als freie Software an der Universität von Tartu, Estland, entwickelt und gepflegt und ist mit vielfältiger Sprachanpasssung verfügbar, unter anderem in Deutsch (seit Rev. 3.3 als Beta!). Die Sprache ist nach der Installation beim ersten Start oder nachträglich über **Tool – Options** wählbar.

Legen wir los!
Die Ressource zu Thonny ist die Datei thonny-3.3.x.exe , deren neuste Version direkt von der Produktseite heruntergeladen werden kann. Dort kann man sich auch einen ersten Überblick über die Eigenschaften des Programms holen.



Mit Rechtsklick auf **Windows** und **Ziel speichern unter** laden Sie die Datei in ein beliebiges Verzeichnis Ihrer Wahl herunter. Alternativ können Sie auch diesem Direktlink folgen.

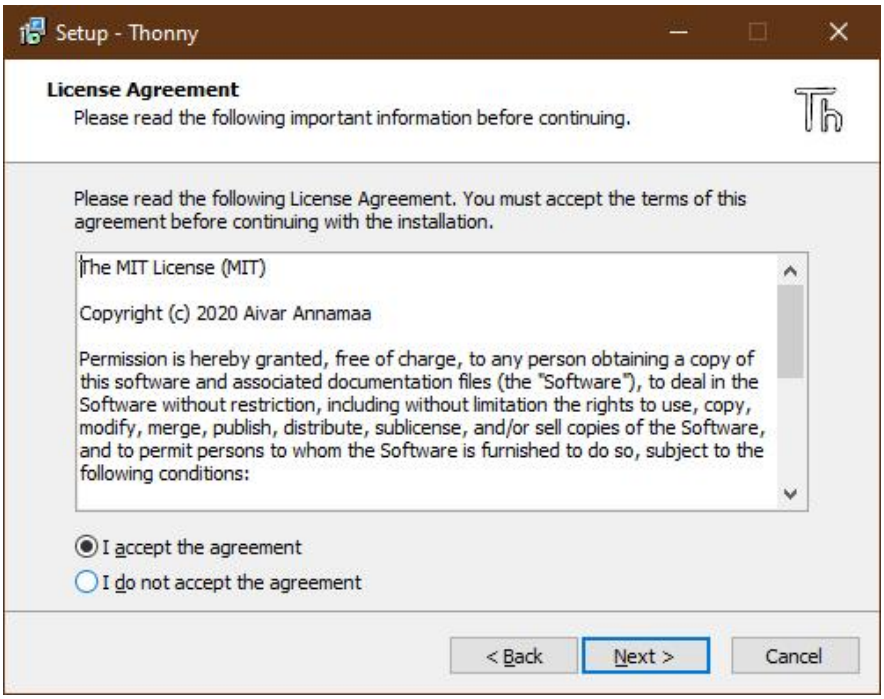


Im Bundle von **Thonny** sind neben der IDE selbst auch **Python 3.7** für Windows und **esptool.py** enthalten. Python 3.7 (oder höher) ist die Grundlage für Thonny und esptool.py. Mit diesem Pythonprogramm können Sie alternativ zu µPyCraft die Firmware auf dem ESP32/ESP8266 flashen. **Thonny selbst bietet keine Möglichkeit, die Firmware zu übertragen**, sondern setzt auf eine bereits installierte Version von MicroPython auf dem ESP32 auf.
Starten Sie jetzt die Installation durch Doppelklick auf ihre heruntergeladene Datei, wenn Sie die Software nur für sich selbst nutzen möchten. Wenn Thonny & Co. allen Usern zur Verfügung stehen soll, müssen Sie die exe-Datei als Administrator ausführen. In diesem Fall klicken Sie rechts auf den Dateieintrag im Explorer und wählen **Als Administrator ausführen**.

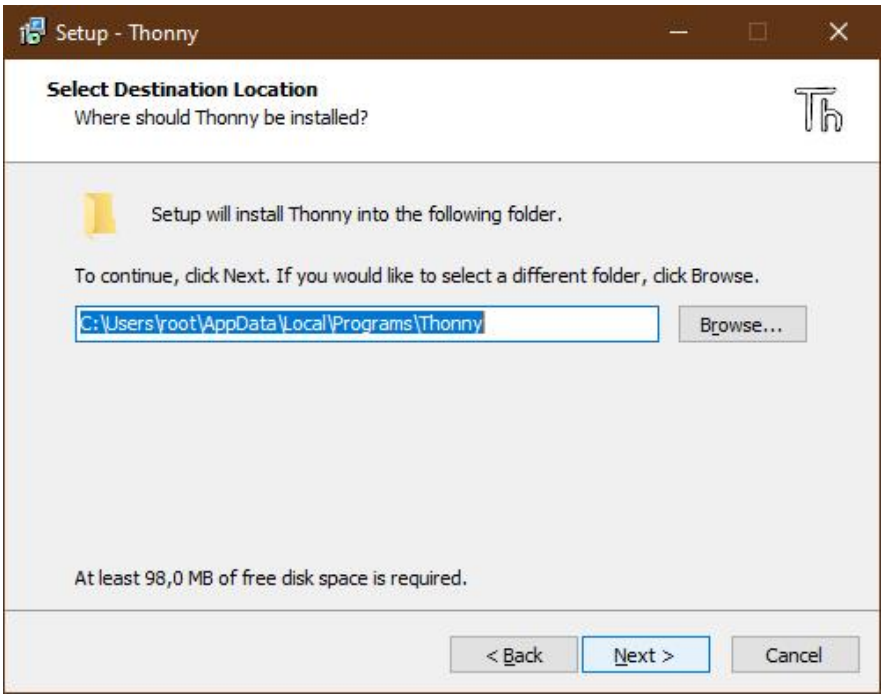
Sehr wahrscheinlich meldet sich der Windows Defender (oder Ihre Antivirensoftware). Klicken Sie auf **weitere Informationen** und im folgenden Fenster auf **Trotzdem ausführen** .

Sie werden freundlich begrüßt, Klick auf **Next**.

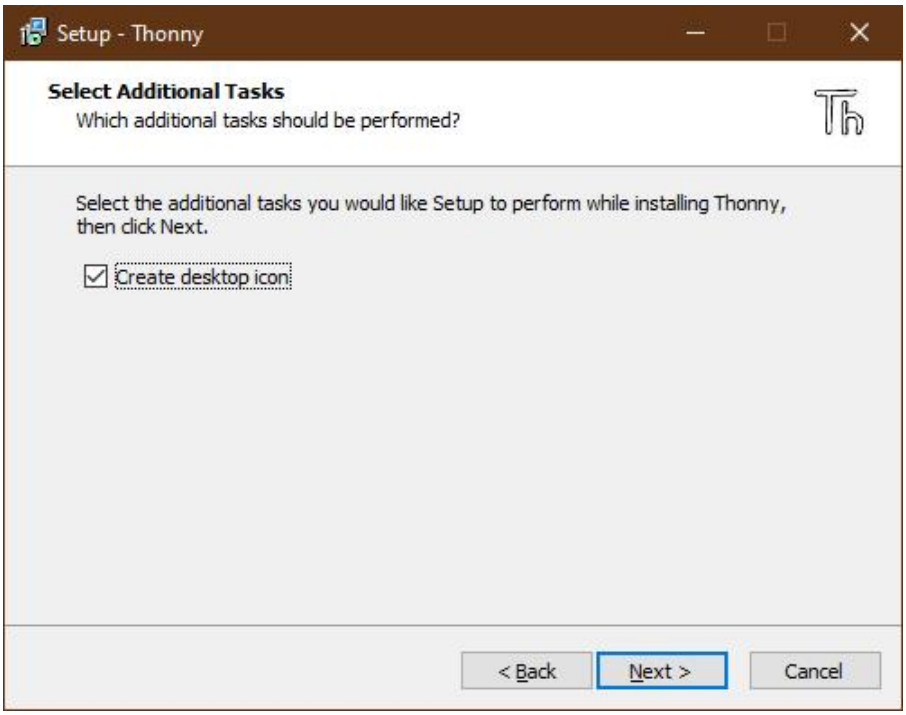
Sie müssen das Lizenzabkommen vom MIT (Masschusetts Institute of Technology) akzeptieren. Diese Lizensierung ist vergleichbar mit GPL. Die Software ist frei und kostenlos und kann auch frei weitergegeben werden. - **Next**



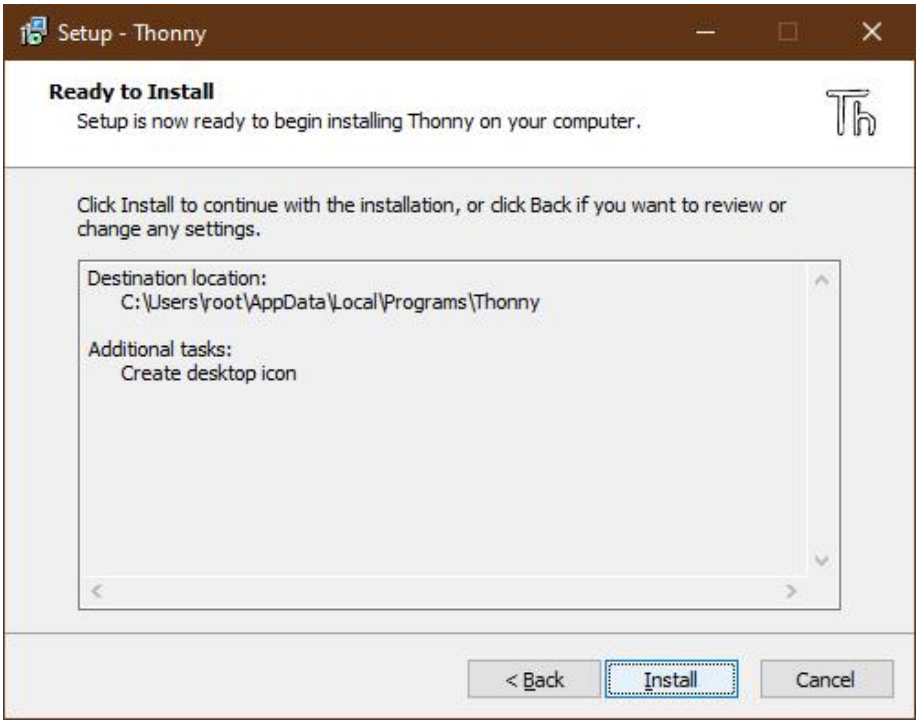
Ich empfehle, den Zielordner zu bestätigen – **Next**.

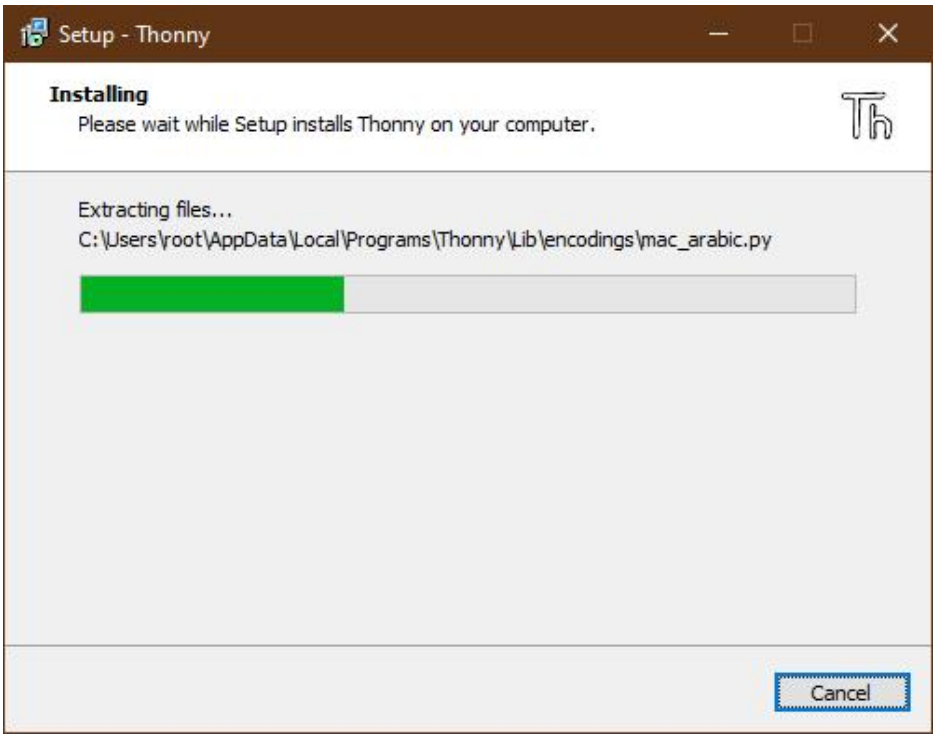


Lassen Sie ein Desktop-Icon erstellen, Sie können es später in einen beliebigen Startordner verschieben – **Next**.



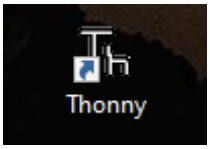
Der Installationsprozess dauert ein paar Minuten, je nach Rechner. Starten Sie ihn mit Klick auf **Install**.



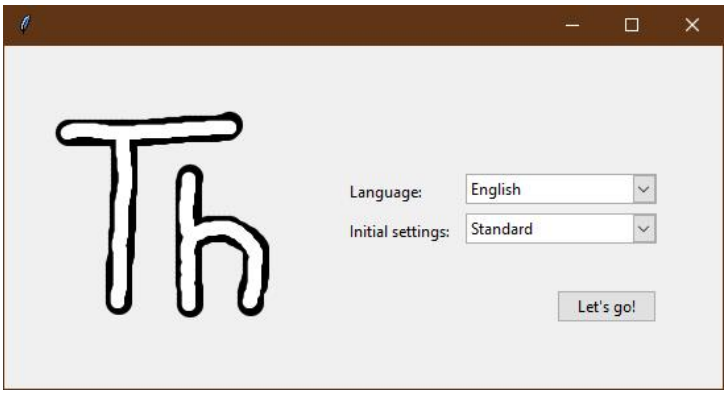


Wenn das folgende Fenster so aussieht, haben Sie es geschafft -**Finish**.

So sieht das Ergebnis auf dem Desktop aus.



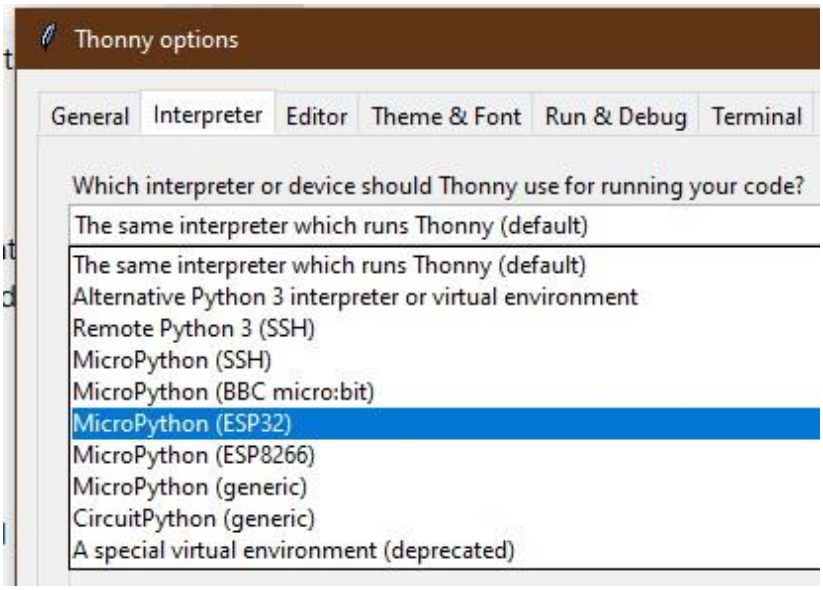
Starten Sie Thonny mit Doppelklick, es gibt noch einige Einstellungen zu machen.

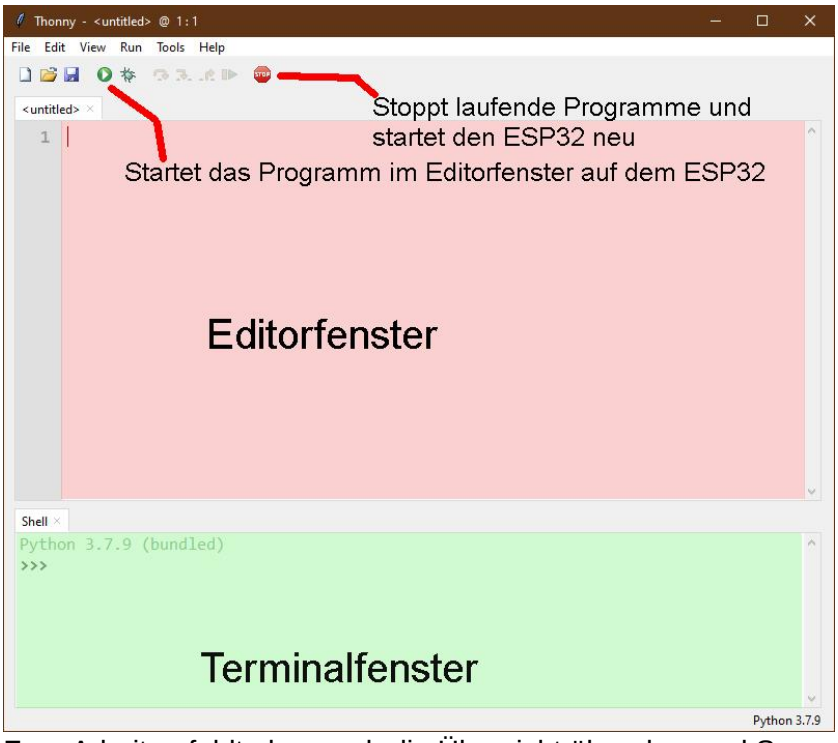


Nachdem die Bedienersprache Deutsch noch in der Betaphase ist, empfehle ich, Englisch zu benutzen – **Let's go!**.

Jetzt ist es an der Zeit, den ESP32 an den USB-Port zu stecken, wenn er sich nicht bereits dort befindet. Thonny erkennt in der Regel den entsprechenden Anschluss selbst.

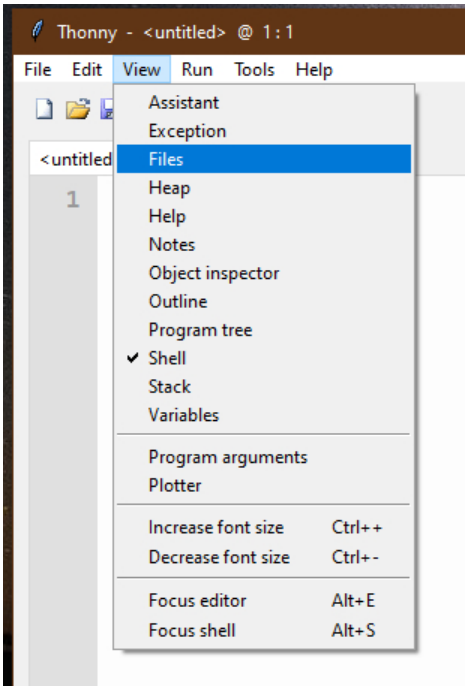
Editor- und Terminalfenster haben die gleiche Position wie bei µPyCraft. Menü und Buttons befinden sich beide am oberen Rand des Fensters. Die Menüs sind bis auf **Device** reich mit Befehlen bestückt. Die ersten drei Icons der Schalterleiste erklären sich selbst und die beiden anderen sind beschriftet. Mit der kleinen Wanze (Bug) wird der Debugger aufgerufen. Er funktioniert, wenn in den Einstellungen (**Run – Select Interpreter …**) **The same interpreter which runs Thonny** ausgewählt ist. Diese Einstellung liefert unter Umständen falsche Daten an das zu testende Programm, wenn Controller- und boardspezifische Informationen, wie z. B. die Plattform, angefragt werden. Stellen Sie am besten **ESP32** oder **ESP8266** ein, je nachdem, womit Sie grade arbeiten.

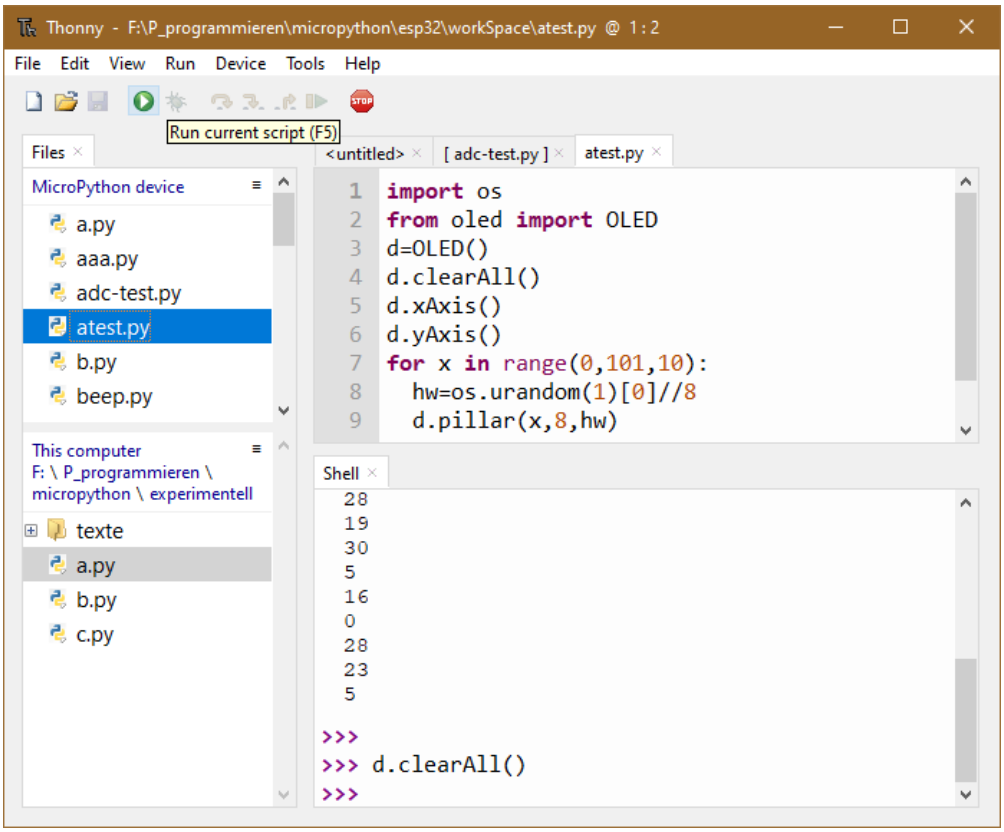




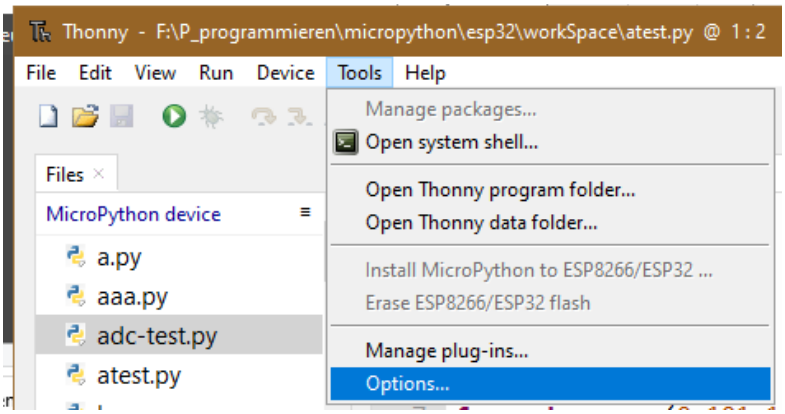
Zum Arbeiten fehlt aber noch die Übersicht über den workSpace und das device Directory.

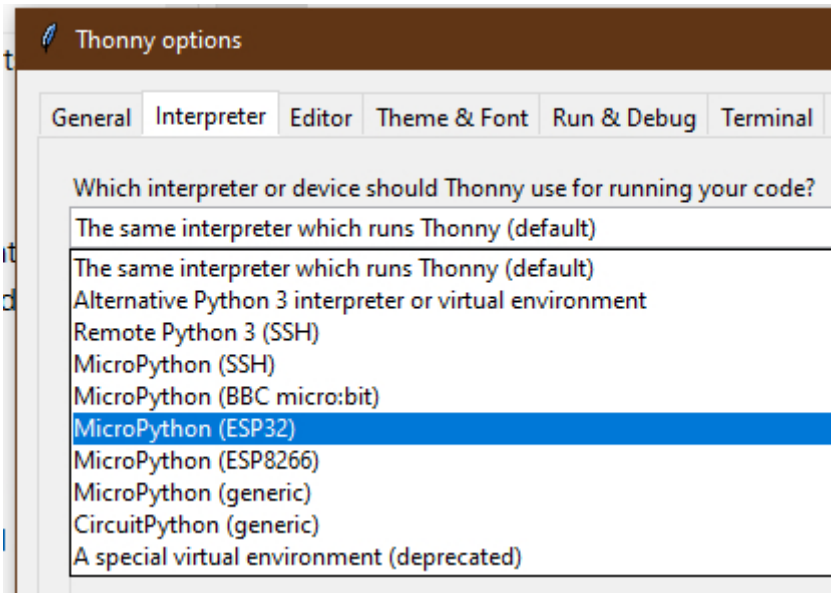
Klicken Sie auf **View** und dann auf **Files**





Jetzt sieht das schon sehr gut aus, es fehlt nur noch die Einstellung für das verwendete Board, wenn Sie das nicht oben schon erledigt haben. Zur Auswahl kommen Sie über **Tools** – **Options**. Stellen Sie am besten ESP32 oder ESP8266 ein, je nachdem, womit Sie grade arbeiten.

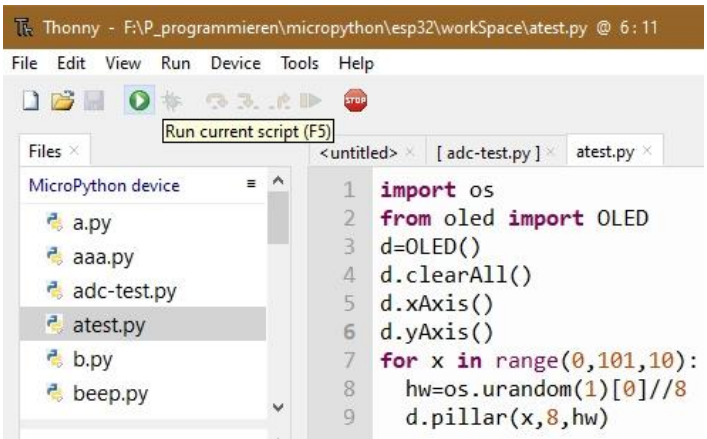




Nach OK können Sie über das Terminal die ersten Befehle an den ESP32 senden.

Im Editor verfassen Sie ihr erstes Programm in Thonny oder wählen eines aus dem workSpace von µPyCraft. Navigieren Sie einfach im unteren Teil der Files-Spalte dort hin. Ein Doppelklick auf den Dateieintrag öffnet die Datei im Editorfenster. **Strg + S** speichert die Datei. Zum Hochladen auf den ESP32 machen Sie einen **Rechtsklick** auf den Dateieintrag und wählen **Upload to/.**

Zum Starten einer Datei auf dem ESP32 **wählen Sie das Script im Editor aus** und klicken auf den **grünen Button** oder drücken Sie einfach **F5**. Mit dem **Stoppschild** starten Sie den ESP32 neu. **Strg+C** beendet das laufende Script. Wenn gar nichts mehr geht, benutzen Sie den Resetknopf auf dem ESP32.



Durch Rechtsklick auf einen Dateieintrag im device Directory und **Download to ….** können Sie eine Datei von dort in den workSpace kopieren. Das jeweilige Zielverzeichnis wird mit angezeigt.

Ob Sie eine Datei aus dem workSpace oder aus dem device Directory im Editor geöffnet haben, erkennen Sie an der Titelzeile von Thonny und an der Notation des Dateinamens im Label des Editorfensters. **atest.py** ist im obigen Bild im workSpace auf Laufwerk F: geöffnet. Die eckigen Klammern um **adc-test.py** sagen Ihnen, dass die Datei aus dem device Directory geladen wurde. Darüber würde in diesem Fall auch die Titelzeile informieren.

Neben der Files-Übersicht lassen sich weitere Spalten anzeigen, wodurch Sie z. B. einen Überblick über die Variablen und deren Belegung in Ihrem Programm erhalten.

Manche Dinge in Thonny funktionieren leider nicht so ganz, z. B. das Andocken von Spalten über View. Leider hängt sich das Programm auch von Zeit zu Zeit auf, was den Programmierspaß mindert. Rechtzeitiges Speichern der Arbeit ist daher in jedem Fall anzuraten! Nachteilig ist auch, dass die Firmware separat über die Kommandozeile mit esptool.py zu flashen ist. Aber das ist ja nur selten erforderlich.

Die Nachteile werden aber durch den angenehmeren Workflow gut aufgewogen. Der Filetransfer ist einfacher als bei µPyCraft und der workSpace ist viel einfacher zu wechseln. Mit dem thonny-eigenen Interpreter ist sogar das schrittweise Debuggen von Scripten möglich. Sie selbst entscheiden nach dem Test der IDEs, welche davon Ihre Bedürfnisse am besten deckt. Ich will Ihnen nichts aufschwatzen.

Im Moment haben Sie ja einen ESP32 mit laufender Firmware und gegebenenfalls können Sie ihn mit µPyCraft neu flashen, wenn alles schief geht. In der nächsten Folge werde ich Ihnen dann auch noch zeigen, wie die Firmware mit esptool.py geflasht wird und welche Möglichkeiten der Konfiguration es da gibt. An dieser Stelle vertröste ich Sie auf den nächsten Beitrag, damit jetzt die Hardware zum Zuge kommt.

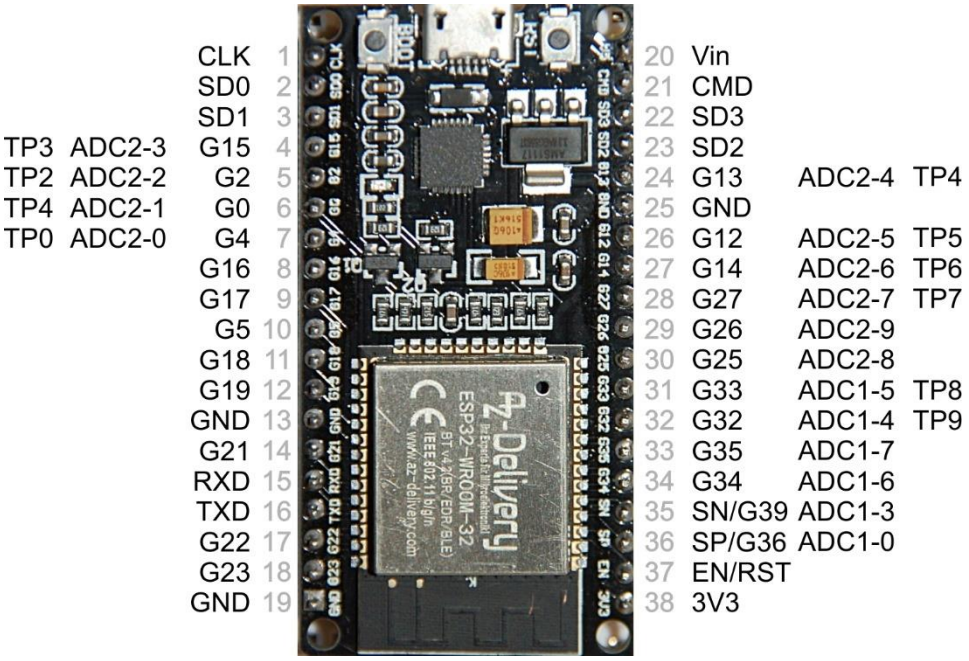
Analoge Signale messen

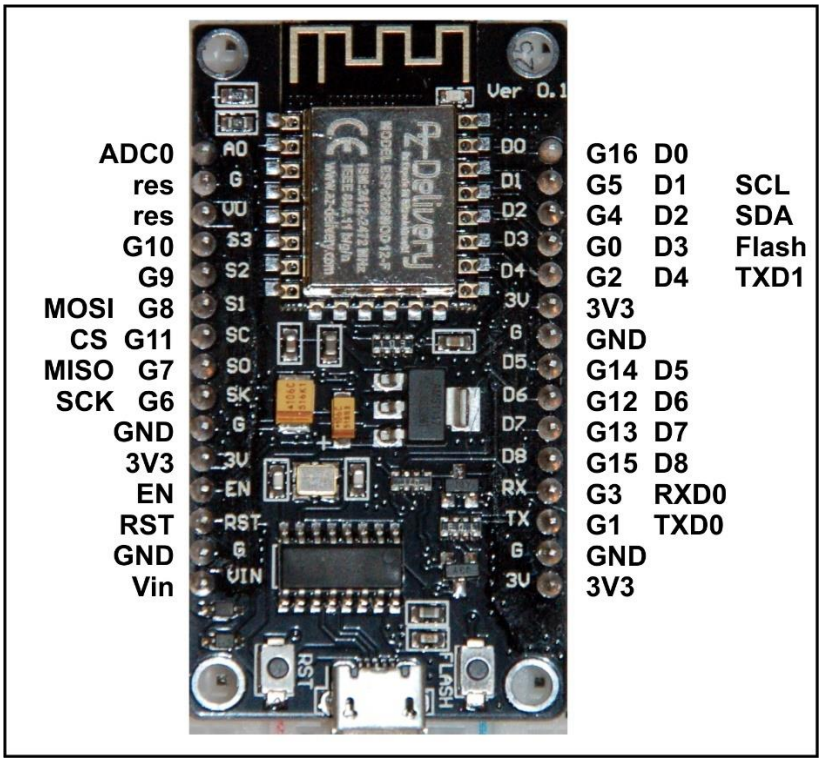
Während der ESP8266 nur einen analogen Eingang aufweisen kann, besitzt der ESP32 16 Pins, die (auch) als analoge Eingänge genutzt werden können. Diese Pins sind intern an zwei ADCs, ADC1 und ADC2 gelegt. ADC steht für **A**nalog **D**igital **C**onverter. Diese Baugruppen wandeln eine angelegte Analogspannung in einen Integerwert, also eine ganze Zahl um. Über die dahinter versteckte Technologie können Sie bei Wikipedia nachlesen. Für uns ist das jetzt und hier nicht so wichtig, wir wollen das ja nur anwenden. Daher gebe ich hier auch nur eine ganz oberflächliche Beschreibung dessen, was im Wandler passiert.

Das ADC-Modul des ESP32 vergleicht den am Eingang liegenden Spannungspegel mit einer Reihe von Spannungen, die intern erzeugt werden. Diese Vergleichsspannung wird stufenweise erhöht. Sind die Pegel von externer und interner Spannung gleich, wird die "Stufennummer" als Wert der Wandlung in eine Speicherstelle (diese Art Speicher nennt man Register) geschrieben. Von dort kann er durch entsprechende Methoden abgerufen werden. Weitere Methoden dienen zum Einstellen der Funktionsweise des ADC.

Von den beiden ADC-Gruppen ist in MicroPython nur ADC1 verwendbar. Die Einheit ADC2 hat andere Aufgaben zu erfüllen. Deshalb sind gemäß der Schnellübersicht zum ESP32 nur die Pins 32 bis 39 als analoge Eingänge angegeben. Über die Pins

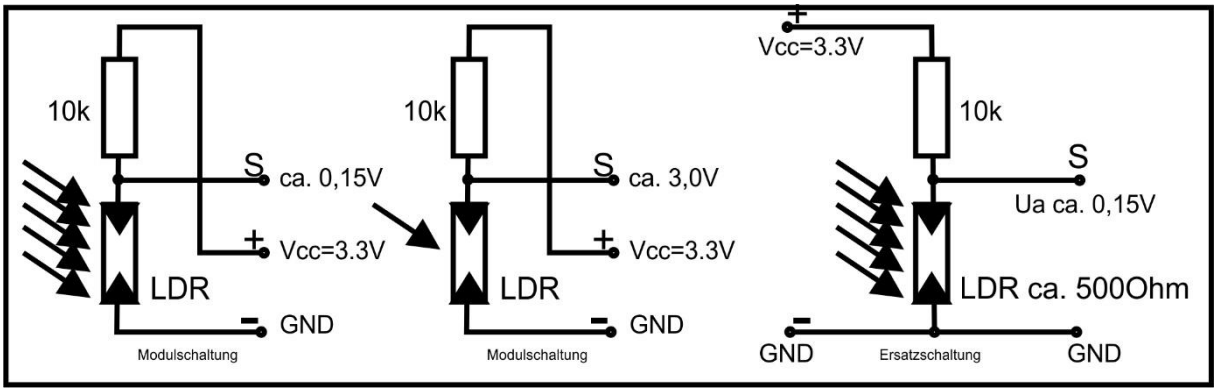
37 und 38 wird dort keine Aussage gemacht, weil diese nicht auf die Steckleisten herausgeführt sind. Weitere Möglichkeiten zeigt zwar die folgende Abbildung, aber eigene Tests haben ergeben, dass in MicroPython tatsächlich nur die Pins ansprechbar sind, die mit ADC1 verbunden sind. Nicht verwendbare Pins werden vom Interpreter mit einer Fehlermeldung zurückgewiesen. Somit verbleiben mit 6 Analog-Kanälen auch nicht mehr wie beim Arduino UNO. Allerdings haben die AD-Wandler des ESP32 mit bis zu 12Bit eine höhere Bitbreite. Der ESP8266 arbeitet, wie der Arduino UNO, fest mit 10Bit.



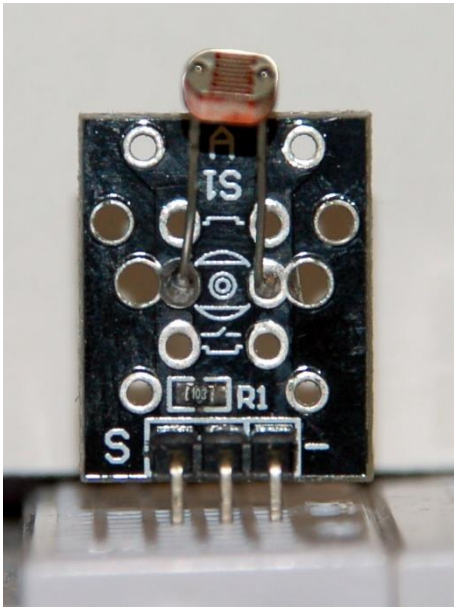


Weiter habe ich festgestellt, dass die Messgenauigkeit und die Linearität der Wandler nicht von bester Qualität sind. Im Projekt, das ich in einer der nächsten Folgen vorstellen werde, spielt das aber keine Rolle, weil dort keine Absolutwerte erfasst werden müssen.

Stellen Sie nun für die nächsten Versuche die nötige Schaltung her. Sie benötigen den ESP32 auf den Breadboards, das LDR-Modul KY-018 Foto LDR Widerstand, und drei Jumperkabel (male-male). LDR ist ein Akronym für **L**ight **D**ependent **R**esistor = Lichtabhängiger Widerstand. Das bedeutet, dass der Widerstandswert von der Beleuchtungsstärke abhängt. Hier ist die Schaltung des Moduls. Die Polung habe ich erst einmal so eingezeichnet, wie sie dem Aufdruck auf der Platine entspricht.



Achten Sie genau auf die angegebenen Spannungen und Polungen in den Skizzen. Die Anzahl der Pfeile gibt die Intensität des einfallenden Lichts wieder. Neben dem LDR enthält das Platinchen einen Festwiderstand R1 von 10kOhm, der mit dem LDR in Reihe geschaltet ist. Diese Schaltung wird auch als Spannungsteilerschaltung bezeichnet, weil die Spannung, die man zwischen den beiden Widerständen messen kann, stets kleiner ist wie diejenige, die an die Enden der Schaltung angelegt wird. Es ergeben sich zwei Anschlussmöglichkeiten des Platinchens.

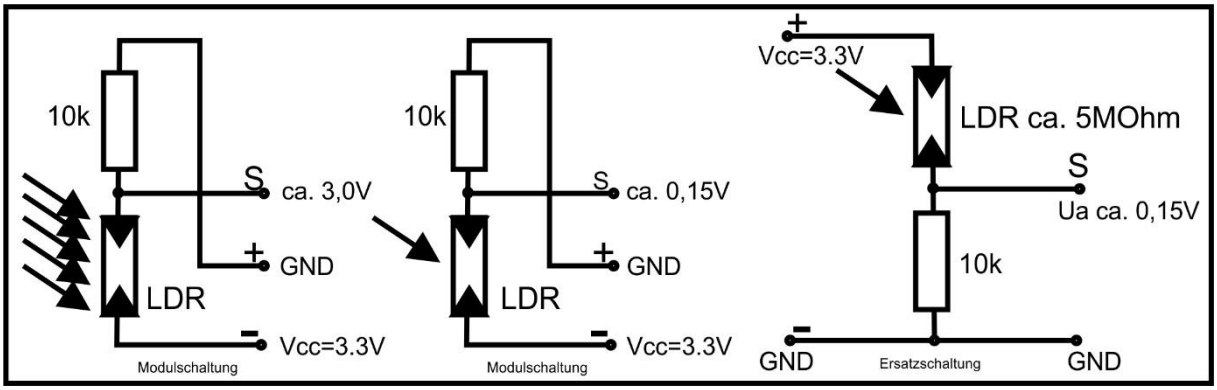


Wenn Sie sich an die Beschriftung auf der Platine halten, liegt der positive Anschluss in der Mitte und der GND-Anschluss an dem rechten mit "-" beschrifteten Stift. Das entspricht der obigen Schaltskizze.

Weil der Widerstand des LDR bei heller Beleuchtung sinkt, wird sich am Spannungsteiler aus dem 10k-Widerstand und dem LDR am Mittelabgriff S, gegen GND gemessen, eine relativ niedrige Spannung einstellen. Denn die Spannungen an den Widerständen verhalten sich wie die Widerstandswerte selbst. Das heißt kleiner Widerstandswert, niedrige Spannung, hoher Widerstandswert höhere Spannung.

Erreicht den LDR wenig Licht, steigt sein Widerstand bis in dem MegaOhm-Bereich, und die Spannung an S geht gegen Vcc = 3,3V (obige Schaltskizze links). Trifft hohe Lichtintensität auf den LDR, sinkt sein Widerstand deutlich unter 500 Ohm und die Spannung liegt im Millivoltbereich (obige Schaltskizze Mitte). Das ist einsichtiger, wenn man die Schaltung des Moduls so zeichnet wie es üblich ist – Vcc oben, GND unten und Signalausgang in der Mitte (obige Schaltskizze rechts).

Leider ist das nicht das von mir gewünschte Verhalten. Ich möchte hohe Spannungen und somit hohe Abtastwerte haben, wenn viel Licht auftrifft und, na ja, halt auch umgekehrt. Deswegen drehe ich die Polung der Versorgungsspannung an den Anschlüssen des Platinchens um. Das geht bei diesem Modul ohne Weiteres, weil das optische Bauteil eigentlich ein "normaler" Widerstand und keine Diode oder kein Transistor ist. Bei Letzteren wäre die Umpolung nicht möglich, da müsste man die Positionen der Bauteile tauschen, also umlöten. Hier ist meine Variante. Die Anordnung der Bauteile ist gleichgeblieben, aber die **Polung der Versorgungsspannung Vcc ist mit GND vertauscht**. Die Ersatzschaltung rechts zeigt auch jetzt die Zusammenhänge besser auf.



Zur Erklärung des Verhaltens dieser Schaltung ziehe ich kurz etwas Physik und Mathematik zu Rate. Nach dem, was man landläufig als Ohmsches-Gesetz bezeichnet, sind die an einem Leiter anliegende Spannung U und die Stärke I des durch ihn fließenden Stroms zueinander direkt proportional, sie wachsen gleichmäßig. Der Quotient U / I ist also eine Konstante (, die man gerne als elektrischen Widerstand definiert). Das hilft uns, grob zu berechnen, was uns erwartet. Der Widerstand R1 auf dem Platinchen hat den festen Wert 10 kOhm = 10000 Ohm.

Weil die Widerstände in Reihe geschaltet sind, muss durch beide der gleiche Strom fließen. Das nach I aufgelöste Ohmsche-Gesetz gilt für die Summe der Widerstände und die Versorgungsspannung Vcc = 3,3V.

$$R1 + LDR = \frac{Vcc}{I} \leftrightarrow I = \frac{Vcc}{R1 + LDR}$$

Es gilt aber auch für den Teilwiderstand R1 und die gegen GND anliegende Spannung Ua. Auch hier muss die gleiche Stromstärke I herauskommen.

$$R1 = \frac{Ua}{I} \leftrightarrow I = \frac{Ua}{R1}$$

Wenn die Stromstärke gleich ist, kann man auch die beiden Brüche gleichsetzen.

$$\frac{Vcc}{R1 + LDR} = \frac{Ua}{R1}$$

Aus dieser Gleichung berechnen wir Ua.

$$\frac{Vcc \cdot R1}{R1 + LDR} = Ua$$

An dieser Stelle trennen sich die Wege für ESP32 und ESP8266. Lesen Sie hier weiter, wenn Sie mit einem ESP32 arbeiten. Für den ESP8266 erkläre ich den Sachverhalt im Anschluss.

Analoge Signale beim ESP32

Setzen wir die Werte für helle Beleuchtung ein, LDR ist ca. 500Ohm und weniger.

$$Ua = \frac{Vcc \bullet LDR}{R1 + LDR} \quad \text{DUNKEL}$$

$$Ua = \frac{3{,}3V \bullet 10000\,\Omega}{10000\Omega + 5.000.000\,\Omega}$$

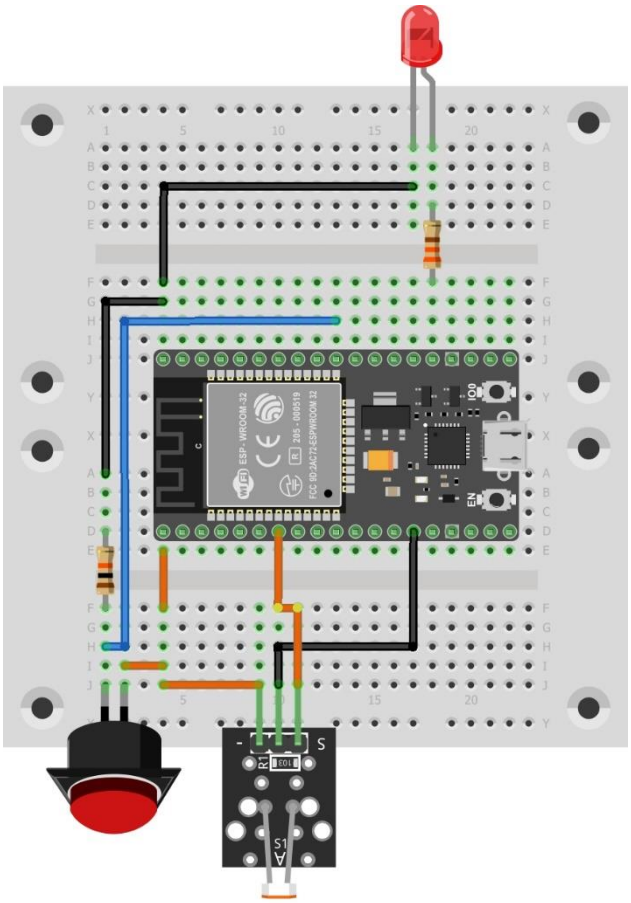
$$Ua = \frac{3{,}3V \bullet 10000\,\Omega}{5.010.000\,\Omega}$$

$$Ua = 3{,}3V \bullet \frac{10.000}{5.010.000}$$

$$Ua = 3{,}3V \bullet 0{,}02 = 0{,}007V$$

Für wenig Licht steigt der Widerstand des LDR auf hohe Werte bis in den MegaOhm-Bereich. Im Rechenbeispiel habe ich 5MΩ hergenommen. M = Mega steht für den Faktor 1.000.000.

So sieht das auf unserem Breadboard aus. Unten wurde der LDR-Baustein ergänzt und nach meiner Variante verdrahtet.



Die Vorbereitung und die Erfassung von Messwerten für dieses Experiment sind mit ein paar simplen Befehlen zu erledigen. Wir importieren die Klassen **Pin** und **ADC** vom Modul **machine** und von **time** die Klassen **time** und **sleep**.

```
>>> from machine import ADC, Pin
>>> from time import time, sleep
```

Wir erzeugen eine Instanz ldr der ADC-Klasse. Der Name ist unwichtig. Aber der Aufruf des Konstruktors der ADC-Klasse ist interessant. Der Parameter ist keine einfache Pinnummer, sondern eine Instanz der Klasse **machine.Pin**. Pin(34) erzeugt ein Pinobjekt, das an ADC.__init__ übergeben wird. GPIO34 wird also jetzt als Analogeingang mit dem ADC1 des ESP32 verknüpft.

```
>>>ldr = ADC(Pin(34))
```

Die Abfrage des Analogwerts ist noch simpler und ähnelt dem Einlesen von digitalen Werten. Statt value() heißt es hier **read**(). Der Ausgabewert 523 ist das Resultat gemäß der Default-Einstellungen für das ADC-Objekt, an denen ich nichts verändert habe.

```
>>> ldr.read()
523
```

Der ESP32 wartet noch mit zwei Leckerbissen auf, die der ESP8266 nicht zu bieten hat. Bei Letzterem gibt es nur als einzigen, den Analogeingang A0. Die Auflösung dort ist fest 10 Bit, also max. 1024 Werte, und die Spannung ist laut Quick reference for the ESP8266 auf maximal 1V beschränkt. Das Vorgehen für den ESP8266 beschreibe ich weiter unten.

Beim ESP32 kann man schon einmal die Auflösung in 4 Stufen angeben. Mittels 4 Konstanten, die in der Klasse ADC definiert sind, erfolgt die Umschaltung durch die Instanz-Methode **width**(). Jeder Instanz kann deshalb ein anderer Wert zugewiesen werden.

Bitbreite	512	1024	2048	4096
Konstante	ADC. WIDTH_9Bit	ADC. WIDTH_10Bit	ADC. WIDTH_11Bit	ADC. WIDTH_12Bit
Wert der Konstante	0	1	2	3

Ferner stehen durch den internen Abschwächer 4 Bereiche für die Eingangsspannung zur Verfügung, die auch durch die Methode **atten** (von attenuation=Verminderung, Abschwächung) über Parameter umgeschaltet werden können. Die Referenzierung der Konstanten über den Klassennamen ADC sagt uns außerdem, dass es sich um Klassenattribute handeln muss. Die Fehlermeldung beim Versuch, dem Attribut einen Wert von der Instanz ldr aus zuzuweisen, bestärkt uns in dieser Ansicht.

```
>>> ldr.WIDTH_12BIT=4
AttributeError: 'ADC' object has no attribute 'WIDTH_12BIT'
```

Die Auflistung der Members der Klasse ADC sagt es eindeutig. Hier sehen wir auch, dass außer read, read_u16, atten und width keine weiteren Methoden in dieser Klasse verfügbar sind.

>>> **dir(ADC)**
['__class__', '__name__', 'read', '__bases__', '__dict__', 'ATTN_0DB', 'ATTN_11DB', 'ATTN_2_5DB', 'ATTN_6DB', 'WIDTH_10BIT', 'WIDTH_11BIT', **'WIDTH_12BIT'**, 'WIDTH_9BIT', 'atten', 'read_u16', 'width']

Tabelle 2 gibt zu den Attributen den Wertebereich der Eingangsspannung und die hinter dem Namen verborgenen Werte wieder.

maximaler Messbereich	1,2V	1,5V	2,5V	3,3V
Konstante	ADC. ATTN_0dB	ADC. ATTN_2,5dB	ADC. ATTN_6dB	ADC. ATTN_11dB
Wert der Konstante	0	1	2	3

In der Praxis seht das so aus. Bei gleicher Umgebungshelligkeit habe ich zu folgenden Einstellungen die angegeben Werte erhalten:

```
>>> ldr.atten(ADC.ATTN_11DB)
>>> ldr.width(ADC.WIDTH_9BIT)
>>> ldr.read()
81
>>> ldr.width(ADC.WIDTH_10BIT)
>>> ldr.read()
164
>>> ldr.width(ADC.WIDTH_11BIT)
>>> ldr.read()
315
>>> ldr.width(ADC.WIDTH_12BIT)
>>> ldr.read()
642
```

stellt man diese Werte ins Verhältnis zur eingestellten Bitweite (512, 1024, 2048, 4 96), ergibt sich ein annähernd konstanter Wert von 0,157 +/- 0,004. Egal mit welcher Auflösung, der Messwert ist sehr genau reproduzierbar, mehrere Messungen jeweils vorausgesetzt.

Multipliziert man nun den Quotienten 0,157 mit der eingestellten maximal erfassbaren Spannung von 3,3V, dann erhält man die tatsächlich am Pin32 anliegende Spannung: 3,3V * 0,157 = 0,518V. Leider ist dieser Wert "leicht" daneben. Das DVM (**D**igitales **V**olt-**M**eter) sagt nämlich 0,68V an. Das sind 25% Abweichung nach unten. Immerhin verringert sich die Abweichung im oberen Bereich auf 5%. Das bedeutet, dass die Skala dann aber nicht einmal linear ist.

Das zu korrigieren ist eine andere Baustelle, die vielleicht einmal Gegenstand eines eigenen Blogbeitrags wird.

Merken wir uns an dieser Stelle folgende Formel, die es erlaubt, die Spannung am Analogeingang aus dem digitalen Messwert (näherungsweise) zu berechnen. Übersteigt die Eingangsspannung den maximalen Messbereich, ist das ADC-Ergebnis stets Bitbreite-1. **Achtung: Das absolute Spannungsmaximum an den Eingängen ist 3,6V**. Gehen Sie darüber hinaus, verabschiedet sich der ESP32 ins Nirvana.

$$U_{Pin} = \frac{ldr.read()}{Bitbreite - 1} \bullet Messbereich[V]$$

$$Bitbreite = [512, 1024, 2048, 4096]$$

Für das Hauptprojekt im letzten Teil interessiert die festgestellte Ungenauigkeit nicht. Viel interessanter ist dagegen, wie viele Messungen der ADC pro Sekunde oder besser pro Millisekunde schaffen kann. Dies stellt das folgende Programm **adc.py** fest.

Download: Programmtext adc.py

```
from machine import Pin, ADC
from time import sleep, time
ldr = ADC(Pin(34))
ldr.atten(ADC.ATTN_0DB) #Full range: 3.3v
ldr.width(ADC.WIDTH_9BIT) # ldr Maximum ist 511
n = 0
now=time()
dauer = 5  # Programmlaufzeit ca. 5 Sekunden
then=now+dauer
actual=now
while actual <= then:
  ldr_value = ldr.read()
  n+=1
  actual=time()
  # print(ldr_value)
  # sleep(0.1)
print(n/dauer,"Messungen pro Sekunde")
```

Die darin vorkommenden Strukturen kennen Sie schon alle, deshalb überlasse ich die Erklärung einmal Ihnen. Wenn Sie die Raute in den beiden auskommentierten Zeilen entfernen (Indentation korrigieren), damit wird das Programm zwar enorm ausgebremst, dafür bekommen Sie eine Menge an ADC-Werten im Terminalfenster und können auf diese Weise kontrollieren, ob der LDR richtig arbeitet. Schatten Sie den LDR dazu ab oder beleuchten Sie ihn mit einer hellen Lampe. So können sie abschätzen, bei welchen Lichtverhältnissen welche Messwerte herauskommen.

Das Ergebnis der Geschwindigkeitsmessung liegt bei mir sowohl mit 9Bit wie mit 12Bit Auflösung zwischen 12500 und 13500 Samples pro Sekunde, also ca. 13 Samples pro ms. Das ist nicht berauschend, aber es reicht für meine Zwecke gut aus.

Analoge Eingangssignale beim ESP8266

Hier geht es um die Behandlung des Analogeingangs und der Schaltung beim ESP8266. Weil es keine Konfigurationsmöglichkeiten für den ADC gibt, ist das rasch erledigt. Aber es gibt eine wichtige Änderung der Schaltung gegenüber dem ESP32.

Ein wesentlicher Unterschied zum ESP32 ist, dass der Konstruktor für das ADC-Objekt als Parameter nicht ein Pin-Objekt, sondern schlicht und einfach nur eine 0 erwartet. Die folgenden Anweisungen demonstrieren das Vorgehen.

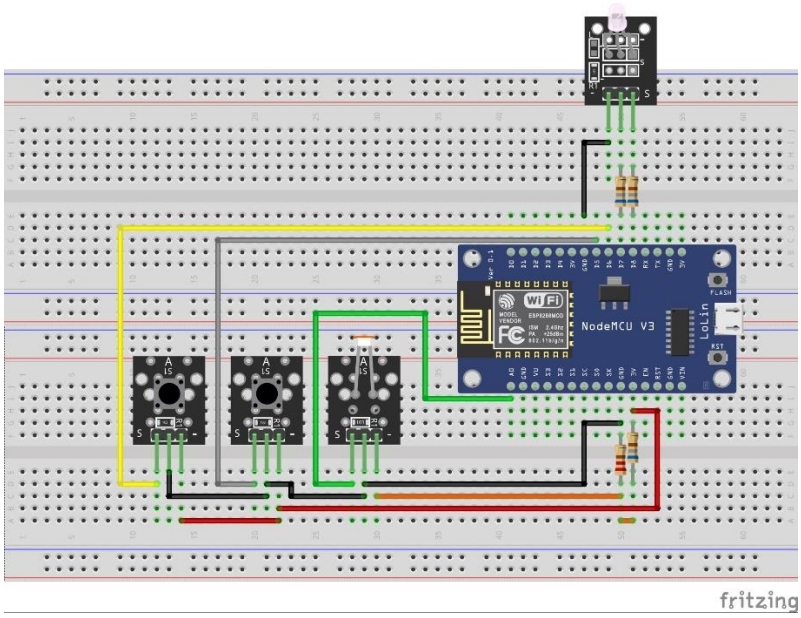
```
>>> from machine import ADC
>>> a=ADC(0)
>>> a.read()
156
```

Man hat also nur einen analogen Eingang, der im Beispiel dem Objekt a zugewiesen wird. Mit der ADC-Methode **read**() wird wie bei ESP32 der Wert eingelesen, der sich zwischen 0 und 1023 bewegt. Eine Sache mahnt zur Vorsicht.

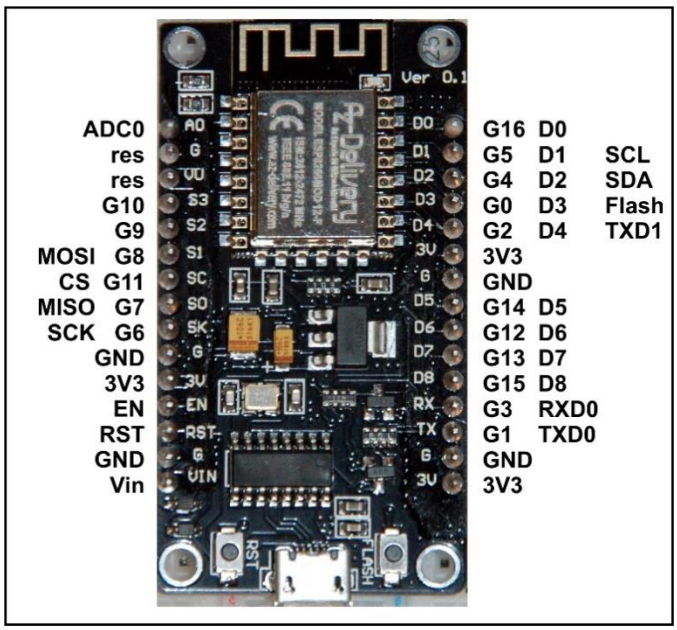
Die maximale Spannung am Pin A0 ist 1V

Das sagt zumindest die Quick reference for the ESP8266. Durch entsprechende Schutzmaßnahmen wie Spannungsteiler und Begrenzungsdioden kann man dafür sorgen, dass diese Bedingung sicher eingehalten wird. Mit der 1V-Grenze ist aber wie sich herausstellte, wirklich nur der Eingang der ESP8266-Baugruppe selbst gemeint. Eine nähere Untersuchung mit der Lupe und dem DVM brachte zum Vorschein, dass bei allen meinen ESP8266-Boards (LoLin V3, Amica, D1 mini) bereits ein Spannungsteiler mit 220kΩ und 100kΩ am A0-Anschluss der Platine liegt. Es wird demnach mit boardeigenen Mitteln die Spannung an A0 gut gedrittelt damit sie den eigentlichen Eingang am ESP8266-12F nicht überlastet. Die Eingangsimpedanz ist durch die beiden Widerstände auf 320kΩ festgeschrieben und damit niedriger als die des hochohmigen Eingangs des ESP8266-12F.

Die Schaltung für den ESP8266 sieht aufgrund des anderen Pinangebots auch anders aus als beim ESP32. Im Vorgriff auf das nächste Kapitel sind die Taster, welche die Touchpads beim ESP32 ersetzen, auch gleich mit eingebaut. Sie liegen an GPIO12 und GPIO14. Statt der RGB-LED habe ich hier die Duo-LED im Einsatz. Die beiden Anoden sind jeweils über 680Ω mit GPIO13 (rot) und GPIO15(grün) verbunden. Die Anschlüsse GPIO5 (SCL) und GPIO4 (SDA) werden später für den I2C-Bus gebraucht. GPIO2 ist intern mit der built-in-LED verbunden.



Zur besseren Orientierung hier noch einmal die Pinbelegung des ESP8266-12F.



Das Programm zum Testen des Analogeingangs fällt beim ESP8266 einfacher aus als beim ESP32.

Download adc8266.py

```
from machine import Pin, ADC
from time import sleep, time
ldr = ADC(0)
n = 0
now=time()
dauer = 5  # Programmlaufzeit ca. 5 Sekunden
then=now+dauer
actual=now
while actual <= then:
  ldr_value = ldr.read()
  n+=1
  actual=time()
  #print(ldr_value)
  #sleep(0.1)
print(n/dauer,"Messungen pro Sekunde")
```

Die Geschwindigkeitsmessung des ADC ergab bei mir ca. 5000 Abtastungen pro Sekunde. Das ist deutlich weniger als die rund 13000 beim ESP32.

Touchpads

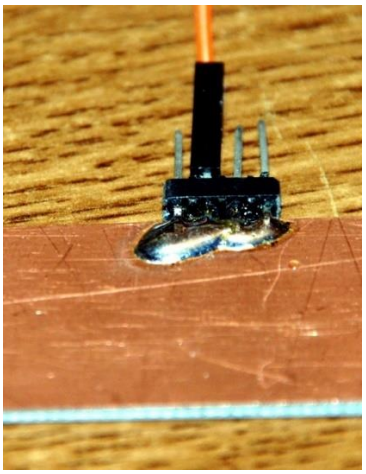
Falls Sie einen ESP8266 benutzen, habe ich jetzt eine schlechte Nachricht. Dieser Controller hat keine Sensoreingänge für Touchpads. Das bedeutet, dass die Teile dieses Kapitels, die nur in Zusammenwirken mit Touchpads funktionieren, mit dem kleinen Bruder des ESP32 nicht experimentell nachvollzogen werden können.

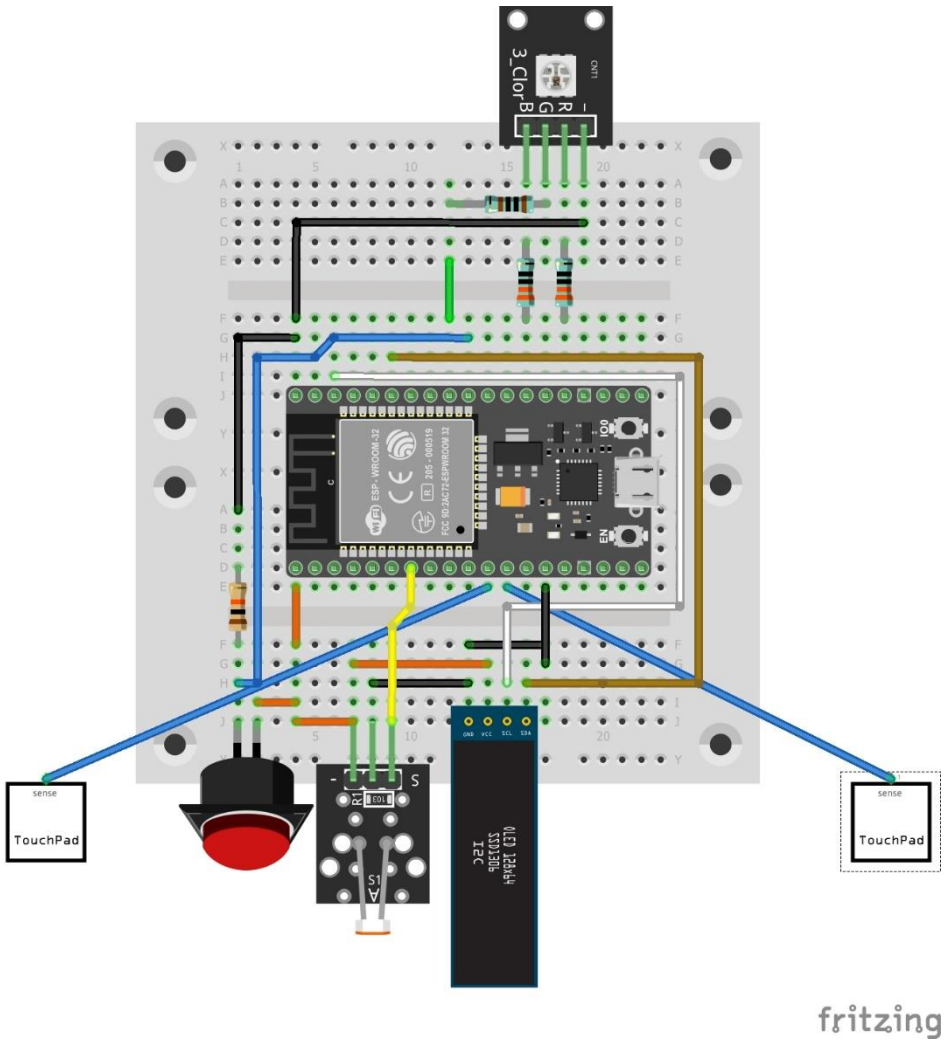
Eine gute Nachricht gibt es aber auch. Die wesentlichen Funktionen aus dem, für den ESP8266 angepassten, Modul **touch8266** funktionieren auch mit normalen Tastern. Diese beiden Funktionen werden im nächsten Blogbeitrag für die Steuerung des OLED-Displays gebraucht. Auf den Einsatz von normalen Tastern gehe noch kurz nach der Behandlung der Touchpads für den ESP32 ein.

Eine gute Alternative für mechanische Taster wäre auch der Einsatz von Touchpad-Modulen, die als Ausgangssignal die Logikpegel 0 und 1 liefern wie ein Taster. Ein solches Modul mit 4 Pads, das Keypad-ttp224, habe ich in der Materialliste am Beginn dieses Beitrags aufgelistet. Der Anschluss erfolgt analog zu den mechanischen Tastern.

Touchpads am ESP32

Jetzt ist es für ESP32-User an der Zeit, die Blechstücke oder Platinenreste mit den Steckleisten zu versehen, denn die benutzen wir nun als Schaltflächen, die auf bloße Berührung reagieren. Schließen Sie dann so ein Teil am Pin GPIO27 an und das zweite an GPIO14, gleich daneben. Am Beginn dieses Beitrags habe ich Ihnen ja schon den Trick mit den Wäscheklammern verraten, wenn kein Lötwerkzeug greifbar ist. Übrigens eine Büroklammer als Befestigungsklipp geht auch.



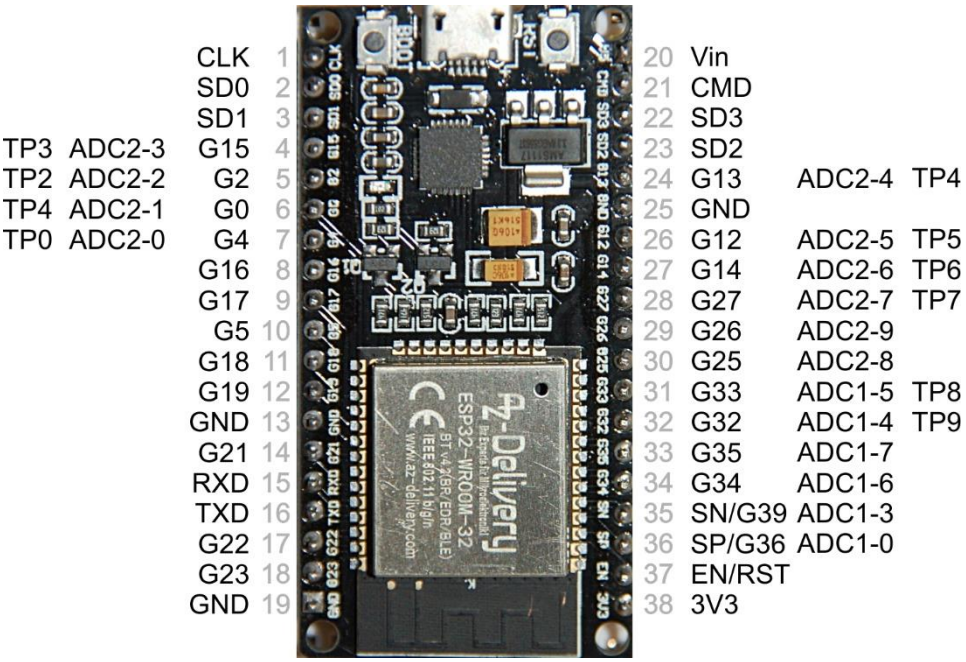


Verschiedene GPIO-Pins können als Anschluss für ein Touchpad definiert werden. Schließt man an diese Eingänge einfach ein Stück Blech oder eine andere leitfähige Oberfläche an, dann kann man eine Berührung dieser Fläche mit dem Finger bei der Abfrage des Pins als Senkung des Abfragewerts feststellen. Vorversuche sind notwendig, um einen Grenzwert für das Ansprechen festzulegen. Denn die Schaltschwelle ist nicht fest definiert. Der Wert ohne Touch geht fast fließend in den mit Touch über. Für eine Anwendung ist es daher essentiell, als Grenzwert einen Integerwert festzulegen, der nicht in der Grauzone um Maximum und Minimum liegt,

und daher ein sicheres Schaltverhalten garantieren kann. Legt man den Wert zu niedrig fest, kann der Schaltvorgang möglicherweise nicht mehr ausgelöst werden, definiert man ihn zu hoch, können zufällige Störereignisse den Schaltvorgang ungewollt auslösen. Letzteres wäre im Fall einer Türöffnungssteuerung fatal. Man könnte den Einsatz von Touchpads also in den Bereich der Fuzzy-Logic einordnen, bei der es nicht nur 0 und 1, sondern auch jede Menge Zwischenwerte geben kann.

Nicht alle GPIOs sind touch-tauglich, nur die folgenden. Die Grafik zeigt auch die Lage, TP steht für Touchpin.

GPIO	15	2	0	4	13	12	14	27	33	32
Touchpad	3	2	1	0	4	5	6	7	8	9



Hier ist ein Programm, mit dem Sie Touchpads testen können.

touch1.py Download

```
from machine import Pin, TouchPad
import utime
#
touch_pin = 27 # GPIO-Nummer des Eingangs
touch = TouchPad(Pin(touch_pin))
while True:
  # versuche den Wert einzulesen
  try:
    touch_val = touch.read()
  # ggf. Fehler abfangen
  except ValueError:
    print("ValueError while reading touch_pin")
  print(touch_val)
  utime.sleep_ms(200)
```

Die Klasse **TouchPad** ist ein Teil des Moduls **machine**, und muss für die Verwendung von Touchpads importiert werden, ebenso die Klasse **Pin**. Wir erzeugen ein Pin-Objekt auf GPIO27 und weisen es über den Konstruktor der **TouchPad**-Klasse dem Objekt **touch** zu.

Weil an diversen Stellen zu lesen ist, dass es bei der Abfrage von Touchpads zu Fehlern kommen kann, die das ganze System aufhängen, sichern wir das mit einer Fehlerbehandlung ab. Deren Arbeitsweise erkläre ich weiter unten genauer. Dann, eingelesenen Touchwert im Terminal ausgeben, 0,2 Sekunden warten und das Ganze von vorn. Diesmal benutze ich einen Delaytimer in Millisekunden, **sleep_ms**().

Untouched ergibt sich bei mir ein Wert von über 400, mit Berührung liegt er um die 60 bis 80. Für diese Hardware würde ich daher einen Schwellenwert (Threshold) von 150 festlegen, das ist deutlich unter 400 und ebenfalls gut über 60.

Die beiden Pads werden wir in der nächsten Folge benutzen, um den Kontrast eines OLED-Displays zu steuern, das wir in die Hardware mit einbauen. Damit die Anwendung erleichtert wird, habe ich dazu eine Klasse **TP** verfasst und das Ganze als Modul in der Datei **touch.py** abgelegt. Die TP-Klasse stellt neben dem Konstruktor drei nützliche Methoden bereit. Aber der Reihe nach. Hier ist erst einmal der Text des Moduls **touch**.

touch.py Download

```
from machine import Pin, TouchPad
from time import time

class TP:
  # ************************************************
  # touch related values
  # Default-Grenzwert fuer Beruehrungsdetermination
  Grenze = const(150)

  # touch related methods
  # ************************************************
  def __init__(self, pinNbr, grenzwert=Grenze):
    self.number = pinNbr
    self.tpin = TouchPad(Pin(pinNbr))
    self.threshould = grenzwert

  # Liest den Touchwert ein und gibt ihn zurueck. Im Fehlerfall wird
  # None zurueckgegeben.
  def getTouch(self):
    # try to read touch pin
    try:
      tvalue = self.tpin.read()
    except ValueError:
      print("ValueError while reading touch_pin")
      tvalue = None
    return tvalue
```

```
# delay = 0 wartet ewig und gibt gegf. einen Wert < threshold zurueck
# delay <> 0 wartet delay Sekunden, wird bis dann kein Touch bemerkt,
# wird None zurueckgegeben, sonst ein Integer, der der Beruehrung enspricht

def waitForTouch(self, delay):
  start = time()
  end = (start + delay if delay >0 else start + 10)
  current = start
  while current < end:
    val = self.getTouch()
    if (not val is None) and val < self.threshold:
      return val
    current = time()
    if delay==0:
      end=current+10
  return None

# delay = 0 wartet ewig und gibt gegf. einen Wert > threshold zurueck
# delay <> 0 wartet delay Sekunden, wird bis dann kein Release bemerkt,
# wird None zurueckgegeben, sonst ein Integer, der dem Leerlauf enspricht

def waitForRelease(self, delay):
  start = time()
  end = (start + delay if delay >0 else start + 10)
  current = start
  while current < end:
    val = self.getTouch()
    if (not val is None) and val > self.threshold:
      return val
    current = time()
    if delay==0:
      end=current+10
  return None
```

Bevor ich das Modul und die Klasse TP bespreche, ein paar Bemerkungen zu Modulen und Klassen vorweg. Ich kratze hier nur an der Oberfläche, in der nächsten Folge arbeiten wir uns dann etwas tiefer in die Materie hinein.

Im zweiten Teil haben Sie gelernt, die in MicroPython eingebauten Module und Klassen zu benutzen. Jetzt befassen wir uns mit dem ersten Beispiel einer eigenen, selbst erstellten Klasse. Ich habe sie TP genannt, und sie ist in einem Modul namens touch enthalten. Das Modul **touch** wird durch die gleichnamige Datei **touch.py** vertreten. Der Dateiname ist also stets auch der Name des Moduls.

Wenn man in so einer Datei Variablen, Konstanten und Funktionen zusammenfasst und abspeichert, ist das ein Modul, aber noch lang keine Klasse. Über die Feinheiten erfahren sie in der nächsten Folge viel mehr. An dieser Stelle mache ich Sie nur mit den beiden wesentlichen Begriffen class und self bekannt, damit Sie verstehen können, wie touch und TP arbeiten.

Das reservierte Wort **class** leitet eine Klassendefinition ein, so wie **def** die Definition einer Funktion einleitet. Was daraufhin folgt, ist der im zweiten Teil angesprochene **Bauplan**.

Unser Bauplan beginnt mit der Deklaration und Definition des Attributs **Grenze**. **Grenze** liefert eine Vorgabe von 150 für den Grenzwert bei der Erkennung einer Berührung. Das Klassenattribut ist in allen Objekten verfügbar und hat überall den gleichen Wert. **Dieser wird einmal bei der Definition der Klasse vorgegeben und sollte händisch oder durch Instanzmethoden nicht verändert werden**. "sollte" steht an dieser Stelle, weil es zwar in MicroPython möglich ist, den Wert von Klassenattributen zu verändern, doch bringt es möglichweise Unheil und beschwört hässliche Fehler herauf, wenn man das tut. MicroPython ist eben nicht C/C++.

self – ist ein reserviertes Wort, das mich vor langer Zeit, als ich mich das erste Mal mit OOP (**O**bjekt **o**rientierte **P**rogrammierung) beschäftigt habe, am meisten verwirrt hat. Was macht self?

Nun, self deutet ganz einfach an, dass sich eine Methode oder ein Attribut auf ein **Objekt** selbst bezieht und eben nicht auf die Klasse. Instanz- oder Objektattribute können, trotz gleichem Namen, von Objekt zu Objekt völlig unterschiedliche Werte annehmen, die sich gegenseitig nicht stören. Solche Instanzattribute werden in einer besonderen Methode deklariert und mit Werten belegt. Diese Methode hat den Namen __**init**__ und wird als **Konstruktor** bezeichnet. Diesem steht keine andere Aufgabe zu, als Instanzattribute zu deklarieren und ihnen Startwerte zuzuweisen. Diese Werte können selbst wieder Bezüge, aka Referenzen, auf andere Objekte sein. **init** steht zwischen zwei öffnenden und schließenden **Unterstreichungen**. Wir werden das gleich am Beispiel der Klasse TP sehen.

Bei Attributen wird dieser Selbstbezug durch Voranstellen von **self.** angegeben. Bei Funktionen, aka Methoden, ist **self** die erste Angabe in der Parameterliste. Beim Aufruf von Funktionen innerhalb der Klassendefinition wird, wie bei Attributen, **self.** vor den Funktionsnamen gesetzt.

Bei der Verwendung, aka Referenzierung, von Methoden und Attributen außerhalb der Klassendefinition wird **self.** durch den Namen des Objekts ersetzt, welches durch den Aufruf des Konstruktors erzeugt wurde. Jetzt erkennen Sie sicher den Zusammenhang. **self** ersetzt in der Klassendefinition einfach den **Namen von Objekten**, die zu dieser Zeit ja noch nicht bekannt sein können.

Der Konstruktor **TP()**, in Form der Methode __init__ hat neben self einen Positionsparameter und einen optionalen Parameter. Der erste Parameter nimmt die verwendete GPIO-Nummer und ist stets anzugeben. Der optionale zweite Parameter kann einen anderen Wert für die Ansprechgrenze annehmen. Wird er weggelassen, dann wird die Vorgabe in **Grenze** hergenommen. Für die Abfrage bzw. Belegung in einem Programm werden die beiden Instanzattribute **number** und **threshold** eingerichtet. **tpin** dient der internen Verwendung in Methodendefinitionen der Klasse und dem direkten Zugriff auf das Touchobjekt. **threshold** kann während eines Programmlaufs unabhängig von anderen Objekten der Klasse TP abgefragt aber auch mit neuen Werten belegt werden. Der Programmierer oder Nutzer ist selbst dafür verantwortlich, dass keine unsinnigen Werte angegeben werden (nicht negativ,

keine Fließkommawerte, nicht größer als 255). Das wird auch Thema einer Hausaufgabe sein.

Übrigens, alles was in einer Klassendefinition steht, ist nach außen sichtbar, Variablen und Methoden. Den Begriff private gibt es in MicroPython nicht. Ich habe weiter oben bereits festgestellt, MicroPython ist eben nicht C/C++. Kommen wir zur Besprechung der weiteren Methoden, welche die Klasse TP zur Verfügung stellt.

Die Methode **getTouch**() versucht, einen Touchwert einzulesen, und gibt ihn zurück, wenn das Ansinnen erfolgreich war. Sonst erfolgt eine Fehlermeldung am Terminal, und die Rückgabe ist **None**. Hier wird noch einmal demonstriert, wie das Abfangen eines Fehlers mit try und except erfolgt. Die folgende Sprechweise vermittelt einen Eindruck von der Funktion der Struktur. "*Wenn der Versuch, einen Wert vom Touchpin zu bekommen, geglückt ist, überspringe den except-Teil, sonst mache, falls ein Wertfehler (ValueError) aufgetreten ist, das, was im Except-Block steht.*" **ValueError** kann auch weggelassen werden, dann springt except auf alle möglichen Fehler an.

waitForTouch() und **waitForRelease**() funktionieren von der Struktur her ähnlich. Der Parameter **delay** ist obligatorisch, muss also angegeben werden. Wird als Wert 0 übergeben, wartet die Methode bis zum St. Nimmerleinstag auf die Berührung oder das Loslassen. In einer **conditional expression** wird das Ende der Wartezeit, **end**, auf den um **delay** erhöhten Startwert gesetzt, falls delay größer 0 ist. Sonst wird **start + 10** zugewiesen. Die laufende Zeit wird auf Start gesetzt und dann geht es in die Schleife. Wir holen den Istwert am Pad ab. Ist der Wert gültig, also nicht **None**, und außerdem kleiner als der Grenzwert (für waitForTouch()) oder größer als dieser (für waitForRelease()), dann sind wir fertig und der eingelesene Integerwert wird zurückgegeben. Falls in der mit **delay** angegebenen Zeit das Ereignis nicht eingetreten ist, ist der Rückgabewert None. Das aufrufende Programm kann damit prüfen, ob das erwartete Ereignis stattgefunden hat.

Ist der Wert vom Pad None oder liegt er im falschen Bereich, stellen wir die laufende Zeit nach und prüfen danach, ob **delay** etwa gleich Null ist. Wenn nein, startet die Schleife einen neuen Durchgang solange, bis die laufende Zeit größer oder gleich dem Wert in **end** ist. Das wird nie geschehen, falls **delay = 0** ist, denn dann wird nämlich bei jedem Durchlauf das Ende um 10 Sekunden weiter hinausgeschoben und somit nie erreicht.

Natürlich testen wir gleich die Funktion unseres ersten Moduls. Ich verwende im Folgenden Thonny. Im Direktmodus (Kommandozeile) importiere ich zunächst die Klasse TP in den globalen Namensraum von REPL und deklariere ein Objekt t, das ich für GPIO27 definiere. Deklaration und Definition sind in der OOP verschiedene Begriffe. Mit der Deklaration reserviert der MicroPython-Interpreter (in der Arduino-IDE der Compiler) Speicherplatz für ein Objekt, der aber noch keinen definierten Inhalt hat. Dafür sorgt dann erst die Definition mit einer Wertzuweisung. Während die beiden Schritte in der Arduino-IDE getrennt durchgeführt werden können, müssen sie in Python in einem Zug passieren, wie hier bei der Erzeugung von t = TP(27). Bei dieser Festlegung wird automatisch der Grenzwert auf 150 gesetzt, weil ich keinen zweiten Parameter angegeben habe. Ich überprüfe das durch Abfragen des Instanzattributs **threshold**.

```
>>> from touch import TP
>>> t=TP(27)
>>> t.threshould
150
```

Ich berühre mein Touchpad nicht und lasse den Wert einlesen.

```
>>> t.getTouch()
474
```

Jetzt berühre ich das Platinenstück und hole erneut den Wert.

```
>>> t.getTouch()
67
```

Nun setze ich den Grenzwert für diese Konfiguration auf das arithmetische Mittel aus den beiden Werten. Ich weise also dem Attribut threshould des Objekts t diesen Wert zu. Weil das Ergebnis ein Integerwert (aka ganze Zahl) sein muss, verwende ich die Ganzzahl-Division mit // als Operator. Danach frage ich den Wert wieder ab.

```
>>> t.threshould = (474 + 67)//2
>>> t.threshould
270
```

Nun erzeuge ich ein weiteres Touchobjekt s und überprüfe sogleich den Grenzwert, der wie erwartet den Wert 150 hat. Der Grenzwert von t ist immer noch bei 270.

```
>>> s=TP(14)
>>> s.threshould
150

>>> t.threshould
270
```

Das Klassen-Attribut Grenze kann ich über jedes Objekt aber auch über die Klasse TP selbst abfragen.

```
>>> t.Grenze
150
>>> TP.Grenze
150
```

Jetzt wird es spannend. Ich mache etwas, das man nicht tun sollte. Ich weise dem Klassenattribut Grenze einen neuen Wert zu. Setzen und Abfrage, klar!

```
>>> TP.Grenze=120
>>> TP.Grenze
120
```

Abfrage über das Objekt t, klar.

```
>>> t.Grenze
120
```

Wenn ich den Wert über das Objekt abfragen kann, was passiert nun?

```
>>> t.Grenze=200
>>> t.Grenze
200
```

Statt 200 hätte hier eigentlich eine Fehlermeldung kommen müssen, dass das nicht möglich ist. Wie kann man das erklären? Die beiden nächsten Abfragen passen wieder einwandfrei ins Schema.

```
>>> s.Grenze
120

>>> TP.Grenze
120
```

Nun, aus einem Objekt heraus kann eine Klassenvariable wie Grenze nicht verändert werden, das wäre fatal, weil die anderen gleichrangigen Objekte davon nichts mitbekämen.

```
>>> t.Grenze=200
```

Dieser Ausdruck erzeugt eine neue Instanzvariable im Objekt t, deren Wert man dort abfragen kann. Natürlich haben weder die Klasse und schon gar nicht andere Objekte der gleichen Klasse Zugriff auf diesen Wert. Die Auflistung der Attribute von t und s zusammen mit ihren Werten zeigt dies deutlich. Die Variable __dict__ enthält eine Liste aus Name-Wert-Paaren der Attribute und Objekte der Instanz einer Klasse. So eine Liste nennt man in Python **Dictionary**. Der Inhalt steht in **geschweiften Klammern**.

```
>>> t.__dict__
{'threshould': 270, 'Grenze': 200, 'number': 27, 'tpin': <TouchPad>}
>>> s.__dict__
{'threshould': 150, 'number': 14, 'tpin': <TouchPad>}
```

In dem Ausschnitt aus der Liste für die Klasse TP erkennt man, dass der Wert des Klassenattributs **Grenze** auf 120 gesetzt wurde.

```
>>> TP.__dict__
{'waitForTouch': <function waitForTouch at 0x3ffe5840>,…,'Grenze': 120, … }
```

Erzeuge ich jetzt ein weiteres Touch-Objekt, dann bekomme ich aber dieses scheinbar seltsame Ergebnis.

```
>>> a=TP(4)
```

```
>>> a.Grenze
120
>>> a.threshould
150
>>> a.__dict__
{'threshould': 150, 'number': 4, 'tpin': <TouchPad>}
```

Dass Grenze im Objekt nicht vorkommt, ist klar. Aber warum hat threshould jetzt wieder den Wert 150 statt 120? Der Widerspruch, der keiner ist, lässt sich ganz einfach erklären. Beim Import wird das Modul Zeile für Zeile vom Interpreter in ein, für den ESP32 lauffähiges Programm übersetzt. Zu diesem Zeitpunkt wird in der Parameterliste eine Referenz auf den Wert Grenze fest eingebaut, der zu diesem Zeitpunkt auf 150 steht und irgendwo im RAM-Speicher abgelegt ist. Auf genau diesen Wert, nicht auf den andernorts im Speicher abgelegten neuen Wert 120 von Grenze, bezieht sich MicroPython, wenn ein Neues Objekt aus der Klasse abgeleitet und kein Wert für den zweiten Parameter angeben wird.

Um dieses Durcheinander zu vermeiden, sollte man Klassenattribute möglichst nicht händisch im Nachhinein verändern, sondern falls nötig vorab in der Klassendefinition.

Schließen wir diese Folge von MicroPython mit dem ESP32 mit einem Test der zeitgesteuerten TP-Methoden ab.

```
>>> t.waitForTouch(5)
104

>>> t.waitForTouch(5)
```

Im ersten Fall wurde das Touchpad vor Ablauf der 5 Sekunden berührt, im zweiten Fall gar nicht. Somit wurde **None** von der Methode zurückgegeben und eben auch **nichts** im Terminal ausgegeben.

Halt! Hätte ich doch fast den ESP8266 vergessen. Wie sieht denn das Modul touch8266 aus?

Touchpad-Ersatz für den ESP8266

An die Stelle der aufwendigeren Abfrage des Sensorwerts in getTouch() tritt ein einfaches Einlesen des digitalen Taster-Eingangspins. Es gibt keinen Grenzwert, damit vereinfachen sich die entsprechenden Conditionals in den wait-Funktionen. Der Konstruktor fällt einfacher aus und die Methode getTouch() auch.

Download touch8266.py

```
from machine import Pin
from time import time

class TP:
  # touch related methods
  # ***********************************************
  def __init__(self, pinNbr):
    self.number = pinNbr
    self.tpin = Pin(pinNbr,Pin.IN)

  def getTouch(self):
      return self.tpin.value()

  # Es wird ein no-Taster vorausgesetzt, der einen Pullup-Widerstand gegen
  # GND zieht, wenn die Taste gedrückt wird.
  # delay = 0 wartet ewig und gibt gegf. einen Wert < threshold zurueck
  # delay <> 0 wartet delay Sekunden, wird bis dann kein Touch bemerkt,
  # wird None zurueckgegeben, sonst ein Integer, der der Beruehrung enspricht
  def waitForTouch(self, delay):
    start = time()
    end = (start + delay if delay >0 else start + 10)
    current = start
    while current < end:
      val = self.tpin.value()
      if val==1:
        return val
      current = time()
      if delay==0:
        end=current+10
    return None

  # Es wird ein no-Taster vorausgesetzt, der einen Pullup-Widerstand gegen
  # GND zieht, wenn die Taste gedrückt wird.
  # delay = 0 wartet ewig und gibt gegf. einen Wert > threshold zurueck
  # delay <> 0 wartet delay Sekunden, wird bis dann kein Release bemerkt,
  # wird None zurueckgegeben, sonst ein Integer, der dem Leerlauf enspricht
  def waitForRelease(self, delay):
    start = time()
    end = (start + delay if delay >0 else start + 10)
    current = start
    while current < end:
      val = self.tpin.value()
      if val==0:
        return val
      current = time()
      if delay==0:
        end=current+10
    return None
```

Die Taster liegen, wie ich schon erwähnt habe, an den GPIO-Pins 12 und 14. Die Verwendung des Moduls **touch8266** und seiner Klasse **TP** ist fast identisch mit der des Moduls **touch** für den ESP32. Nur dass es eben beim Konstruktor keinen zweiten Parameter gibt. Die Methoden haben den gleichen Namen und die gleiche Funktion. Lediglich das Attribut threshould gibt es nicht, weil es zwischen 0 und 1 keine Zwischenwerte gibt.

MicroPython stellt sogar eine Möglichkeit bereit, den verwendeten Controller automatisch zu erkennen und daraufhin das richtige Modul anzusprechen. Das kurze Testprogramm für den ESP8266 mach sich das zunutze. Es funktioniert ohne Änderung auch für den ESP32.

Download touchtest8266

```
import sys
port = sys.platform
if port == 'esp8266':
  from touch8266 import TP
  down = 14
  up   = 12
elif port == 'esp32':
  from touch import TP
  down = 27
  up   = 14
print(port,"erkannt")


t=TP(down)
s=TP(up)

if t.waitForTouch(5) == None:
    print("Taste nicht gedrückt")
#t.threshold
```

Kaffee-Pause!

****

Ausblick

Im nächsten Beitrag werden wir ein OLED-Display und einen aktiven Buzzer am ESP32 anschließen und die Kontrastfunktion des Displays mit Hilfe der Touchpads und dem LDR steuern. Natürlich wird es weitere Module geben und wir werden die Informationen über Klassen weiter vertiefen. Außerdem habe ich Ihnen versprochen, das Flashen mit esptool.py zu erklären.

Anmerkungen nach Redaktionsschluss:

Das Kapitel zum esptool.py musste leider noch einmal um einen Blog verschoben werden und erscheint aus Platzgründen daher erst im fünften Teil.

Die Version 3.0 war extrem buggy und wurde bald von der 3.1 abgelöst, die wesentlich stabiler läuft. Mittlerweile ist 3.3 aktuell, wurde aber von mir noch nicht getestet.

Hier sind noch die Links auf die ersten beiden Beiträge, falls Sie dort noch einmal etwas nachlesen möchten.
Folge 1
Folge 2

Die Folge 3 gibt es auch als PDF zum Download.

Neue Hausaufgaben

1. Schreiben Sie eine Sequenz, die in der Methode __init__() den Parameter grenzwert auf gültige Angaben überprüft. Der Wert muss ganzzahlig, positiv und kleiner als 256 sein.

2. Erstellen Sie eine Plausibilitätskontrolle in __init__, welche die Gültigkeit der GPIO-Nummer für den Touchpin überprüft bevor das Objekt erzeugt wird. Definieren Sie hierfür eine Liste mit den gültigen Pinnummern. Ob die Eingabe einem Wert der Liste entspricht, prüfen Sie mit dem Schlüsselwort **in** nach. Beispiel für die Kommandozeile:

   ```
   >>> eingabe = 4
   >>> touchliste = [15,2,0,4,13,12,14,27,33,32]
   >>> eingabe in touchliste
   True
   ```

3. Können Sie waitForTouch so verändern, dass bei Berührung innerhalb der Laufzeit statt dem Wert des Touchpads die Verzögerung von Start der Methode bis zum Zeitpunkt des Berührens zurückgegeben wird?

4. Bauen Sie Aufgabe 3 zu einem Reaktionstestgerät aus, indem Sie zu Beginn der Laufzeit eine LED aufleuchten lassen, die nach dem Berühren wieder ausgemacht wird.

   Steigerungsform:
   Lassen Sie zufallsgesteuert eine von drei LEDs aufleuchten wobei nur bei einer getouched werden darf. Tippt man bei einer falschen LED, wird man degradiert. Die richtigen Touches werden mitgezählt.

5. Was passiert, beim ESP32 und beim ESP8266, wenn Sie die letzte Zeile in touchtest8266.py entkommentieren?

6. Finden Sie eine Möglichkeit, den Fehler aufzufangen, der beim Abfragen des Attributs threshould auftritt, wenn man das Modul touch8266 benutzt.

Lösungen der Hausaufgaben aus Teil2

- **Was passiert eigentlich, wenn Sie in der URL keinen Slash eingeben? Welcher Wert steht dann in slashPos?**

Das Serverscript server1.py ist hier völlig unempfindlich, weil es die Anfrage nicht parst. Man bekommt stets die gleiche Antwort, egal ob man einen Pfad oder Parameter an die URL dranhängt oder gar nichts. Wichtig ist nur die Portnummer, sonst fühlt sich der Server nicht angesprochen.

In server2.py ist für den Schaltvorgang eine (Pseudo-)Pfadangabe nötig - /ein oder /aus. Gibt man keinen Pfad an oder lässt gar den Slash weg, passiert das Gleiche wie bei einer beliebigen andern Pfadangabe, die nicht mit /ein oder /aus beginnt. Der Server fängt das mit einer Fehlermeldung ab. Ein fehlender Slash in der URL-Zeile wird vom Browser automatisch ergänzt, der kommt also immer beim Server an und deshalb ist nach der Zeile

```
slashPos= request.find("/")
```

der Wert von slashPos immer gleich 4. Weil kein Fehler durch einen nicht vorhandenen Slash auftreten kann, muss man auch nichts Besonderes zu dessen Behandlung unternehmen. Die auf den Slash folgenden 3 Zeichen in der Anfrage 'GET / HTTP/1.1…' sind ' HT', und die werden auch mit der Fehlermeldung ausgegeben: **Ungueltiger Befehl: HT.**

- Falls Sie schon einmal HTML-Seiten von Hand gestrickt haben. Können Sie eine einfache Seite mit Links herstellen, mittels deren die LED geschaltet werden kann?

Fügen Sie die folgenden Zeilen in den HTML-Text im Bereich <body> … </body> ein.

```
<a href=http://8.8.8.8:9192/ein> LED einschalten </a><br>
<a href=http://8.8.8.8:9192/aus> LED ausschalten </a><br>
```

- Können Sie in den Antworttext vom Server die IP und die Portnummer des Clients mit einbauen?

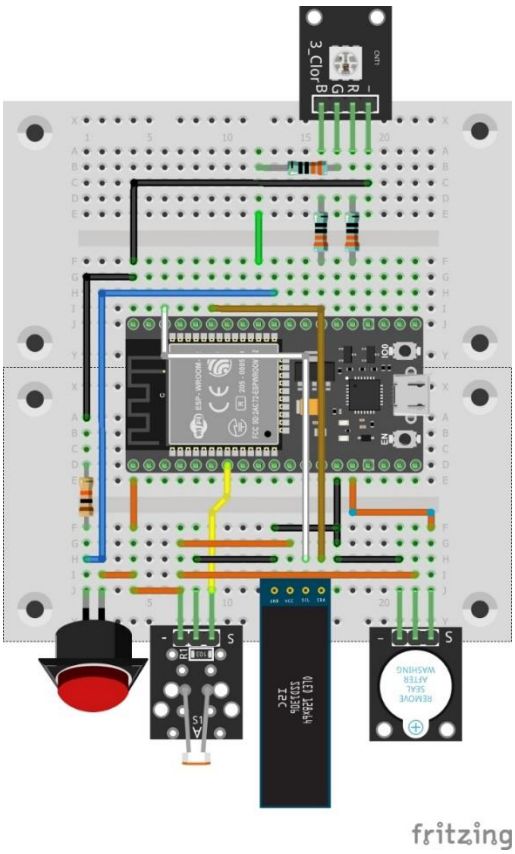
Ersetzen Sie den Aufbau der Variable html in der Funktion webPage() durch folgenden Text. Achten Sie bitte auf die korrekten Einrückungen. Sie können direkt den String der Printanweisung aus der Serverschleife verwenden.

```
html = """<html><head><meta name="viewport"
content="width=device-width, initial-scale=1"></head>
<body><h1>LED-Schalter</h1> """
html = html + 'Got a connection from %s' % str(addr)

html = html + """ <br><a href=http://8.8.8.8:9192/ein> LED einschalten </a><br>
<a href=http://8.8.8.8:9192/aus> LED ausschalten </a><br>
"""
html = html + act+'.</body></html>'
```

- **Fügen Sie dem Aufbau eine zweite LED hinzu, die aufblinkt, wenn eine Anfrage am Server eingeht.**

Ich habe für diesen Zweck eine RGB-LED genommen und an die Pins GPIO2 (rot), 4 (blau) und 18 (grün) angeschlossen. **server2.py** wurde wie folgt ergänzt.



*

```
from machine import Pin

conn=4
connPin=Pin(conn,Pin.OUT,)
connPin.off()
portNum=9192

  connPin.on()
  print('Got a connection from %s' % str(addr))
…
Rest der Serverschleife
…
  c.close()
  connPin.off()
```

Jetzt blitzt jedes Mal, wenn eine Anfrage am ESP32-Server eingeht, die blaue LED kurz auf.