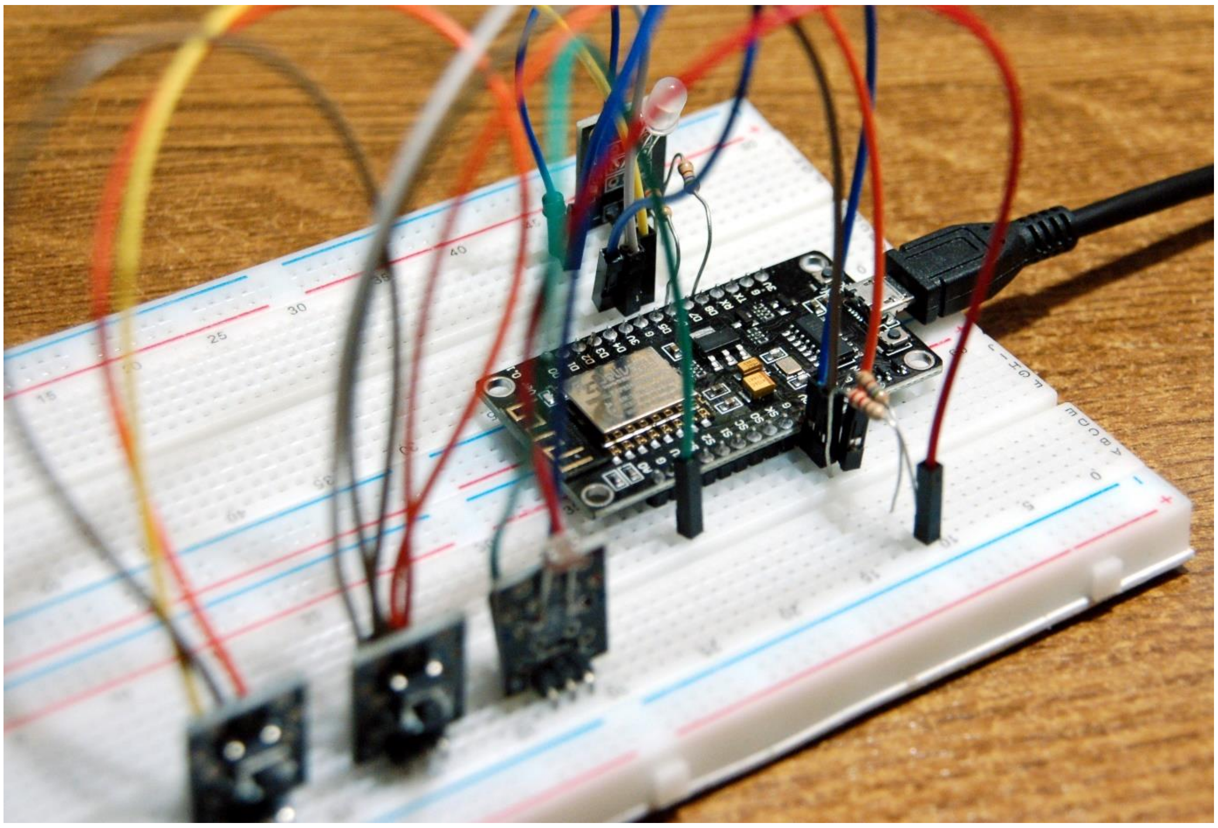


MicroPython mit dem ESP32 und ESP8266 – Teil3

Module und Klassen

Welcome to the third part of MicroPython with the ESP32/ESP8266. This time we will discuss the use of additional hardware and, above all, take a closer look at the subject of modules and classes, which I think is very interesting.

The following hardware is used, some of which you already have if you have read and edited the first two blog posts.

- 1x ESP32 NodeMCU Module WLAN WiFi Development Board oder
- 1x ESP-32 Dev Kit C V4 oder
- 1x ESP32 D1 R32 oder
- 1x ESP8266 LoLin V3 oder ähnlich

- 1x 0,91 Zoll OLED I2C Display 128 x 32 Pixel für Arduino und Raspberry Pi oder

- 1x 0,96 Zoll OLED I2C Display 128 x 64 Pixel für Arduino und Raspberry Pi

- 1x Widerstand 680Ohm

- 1x Widerstand 220Ohm

- 1x KY-011 Bi-Color LED Modul 5mm und
- 2x Widerstand 680 Ohm für LED oder

- 1x KY-009 RGB LED SMD Modul und
- 3x Widerstand 1,0kOhm für LED

- 2x KY-004 Taster Modul

- 1x KY-018 Foto LDR Widerstand

- 2x Mini Breadboard 400 Pin mit 4 Stromschienen für Arduino und Jumper Kabel

- 1x Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F

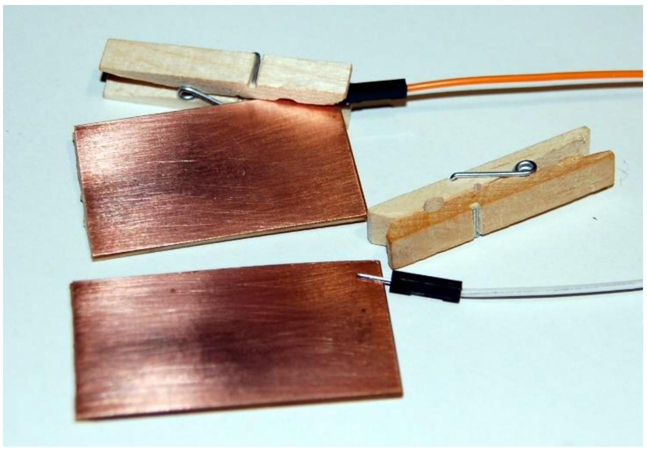
- 2x Blech ca. 20 x 20 mm (nicht Aluminium!) oder Platinenreste

- einige Steckstifte 0,6x0,6x12mm

The 35 in 1 Arduino sensor kit, module and accessory kit for Arduino and other microcontrollers, which I got some time ago, is ideal for experimentation. In addition to the KY modules mentioned above, there are a lot of other interesting sensors.

You need some tools:
• Soldering iron
• solder

Pins for the connection of jumper cables must be soldered to the sheet metal or circuit board pieces. Therefore, no aluminum should be used here, because it can only be soldered with a trick that requires the use of chemistry. So use copper, brass or bare tinplate. If you don't have a soldering tool to hand, you can use two clothespins to fix the pins of the jumper cables to the sheet metal parts. That's enough for the first attempts.

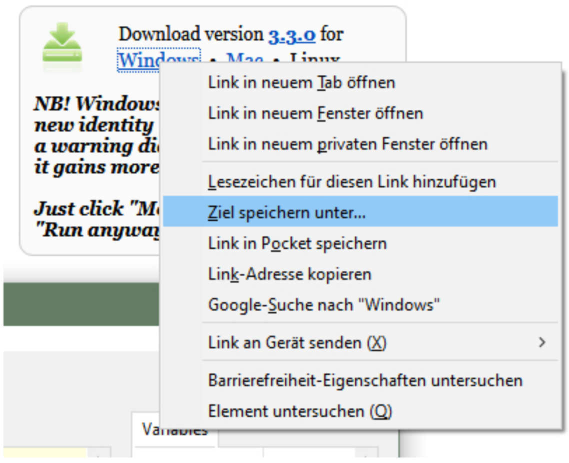


Before we deal with the hardware and its programming, I'll introduce you to Thonny, another IDE for MicroPython. Since the focus of this article, in addition to the integration of hardware, is to be on the creation and use of modules, we will immediately start with a few experiments on this topic.

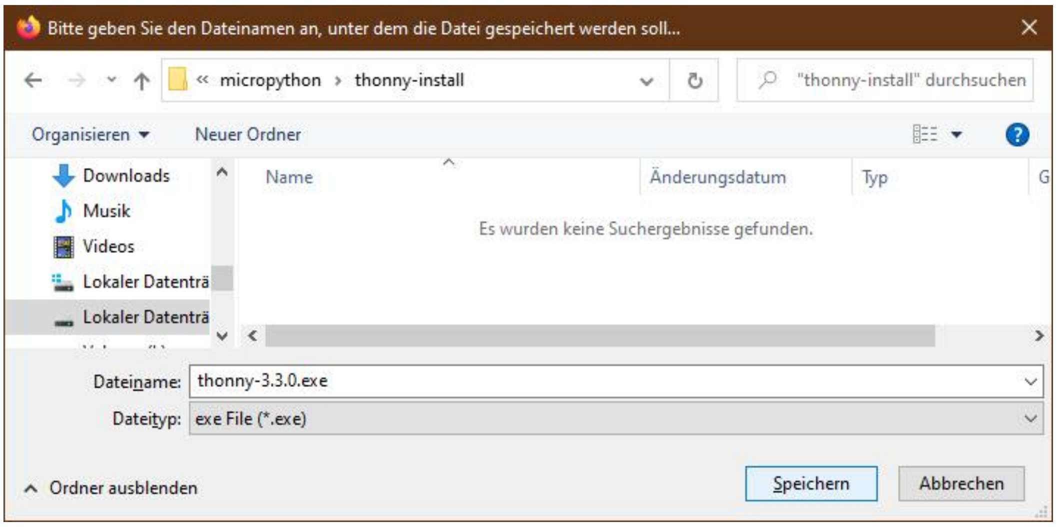
How to install and use Thonny

Thonny is another IDE for programming in MicroPython. It is developed and maintained as free software at the University of Tartu, Estonia, and is available with a variety of language options, including in German (since rev. 3.3 as beta!). The language can be selected after the installation at the first start or afterwards via Tool - Options.

Let's start!
The resource for Thonny is the file thonny-3.3.x.exe, the latest version of which can be downloaded directly from the product page. There you can also get an initial overview of the features of the program.



Right-click on Windows and save target as to download the file to any directory of your choice. Alternatively, you can also follow this direct link.

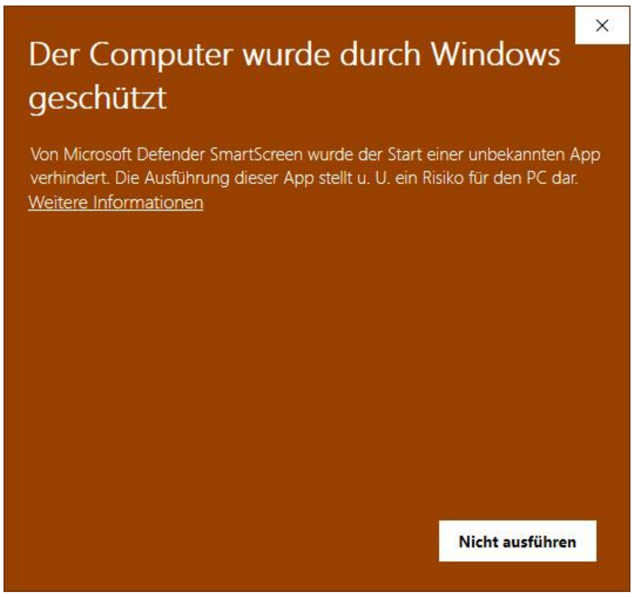

Bitte geben Sie den Dateinamen an, unter dem die Datei gespeichert werden soll...
« micropython › thonny-install
"thonny-install" durchsuchen
Organisieren
Neuer Ordner
Downloads
Musik
Videos
Lokaler Datenträ
Lokaler Datenträ
Name
Änderungsdatum
Typ
Es wurden keine Suchergebnisse gefunden.
Ordner ausblenden
Speichern
Abbrechen

Dateiname: thonny-3.3.0.exe
Dateityp: exe File (*.exe)

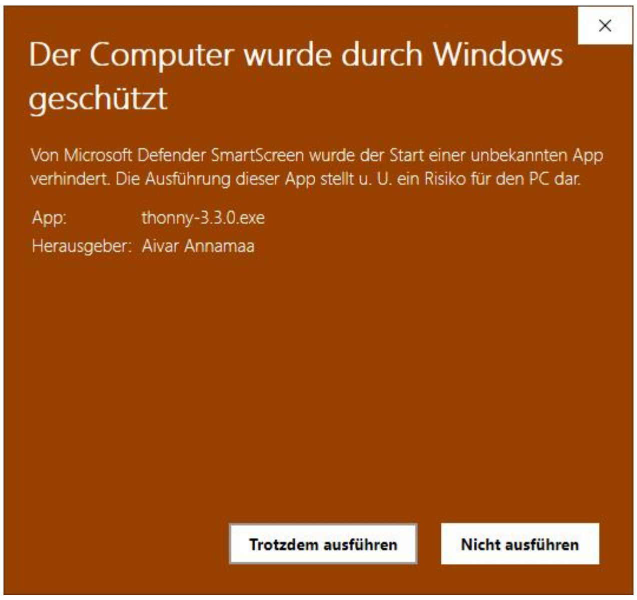
In addition to the IDE itself, the Thonny bundle also includes Python 3.7 for Windows and esptool.py. Python 3.7 (or higher) is the basis for Thonny and esptool.py. With this python program you can flash the firmware on the ESP32 / ESP8266 as an alternative to µPyCraft. Thonny himself also offers the possibility to flash the MicroPython.Firmware, albeit well hidden.

Now start the installation by double-clicking your downloaded file if you only want to use the software for yourself. If Thonny & Co. is to be available to all users, you must run the exe file as an administrator. In this case right click on the file entry in Explorer and select Run as administrator.

Most likely, Windows Defender (or your antivirus software) will respond. Click on more information and in the window that opens click on Run anyway.


Der Computer wurde durch Windows geschützt

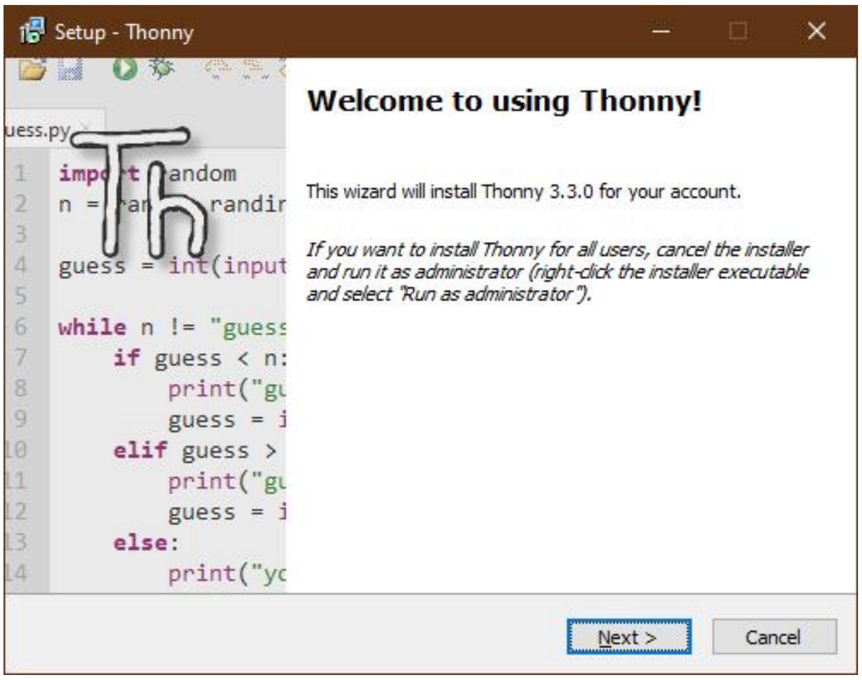
Von Microsoft Defender SmartScreen wurde der Start einer unbekannten App verhindert. Die Ausführung dieser App stellt u. U. ein Risiko für den PC dar.
Weitere Informationen


Der Computer wurde durch Windows geschützt

Von Microsoft Defender SmartScreen wurde der Start einer unbekannten App verhindert. Die Ausführung dieser App stellt u. U. ein Risiko für den PC dar.

App: thonny-3.3.0.exe
Herausgeber: Aivar Annamaa

You will be greeted warmly, click on Next.

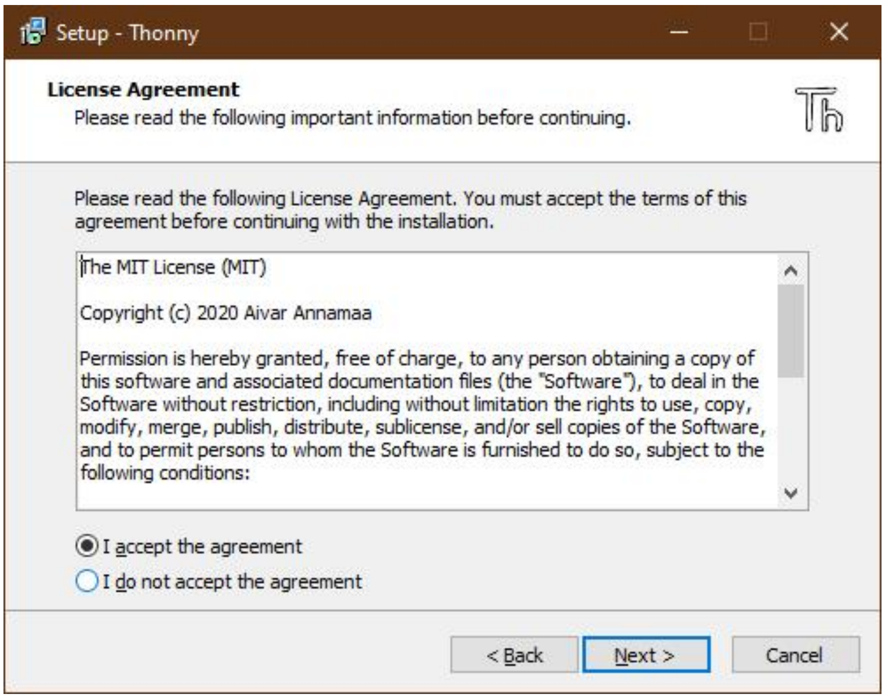


Welcome to using Thonny!

This wizard will install Thonny 3.3.0 for your account.

If you want to install Thonny for all users, cancel the installer and run it as administrator (right-click the installer executable and select 'Run as administrator').

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**



License Agreement
Please read the following important information before continuing.

Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.

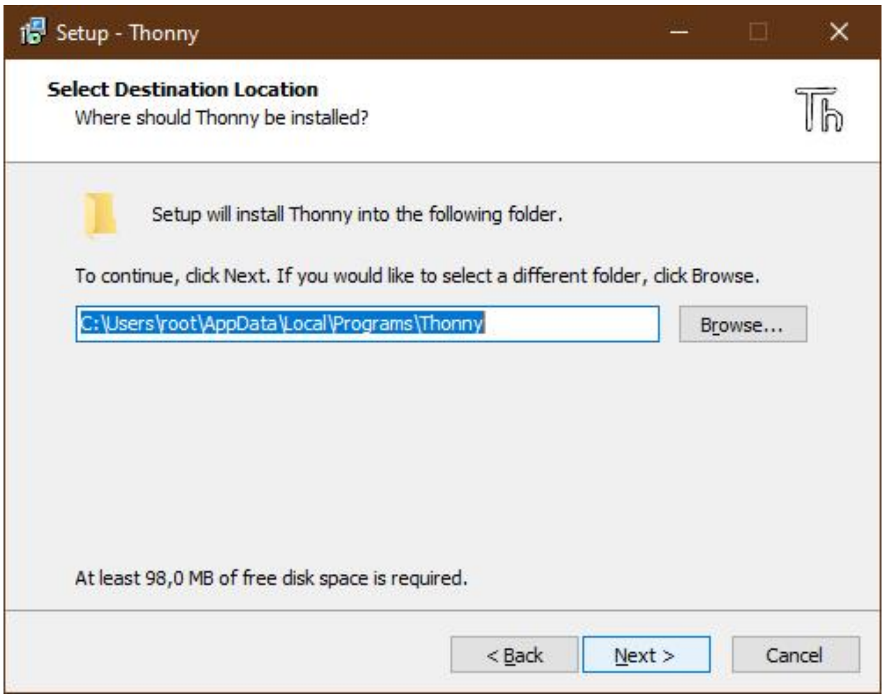
The MIT License (MIT)

Copyright (c) 2020 Aivar Annamaa

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

[✓ I accept the agreement]
I do not accept the agreement

I recommend confirming the destination folder - Next.



Select Destination Location
Where should Thonny be installed?

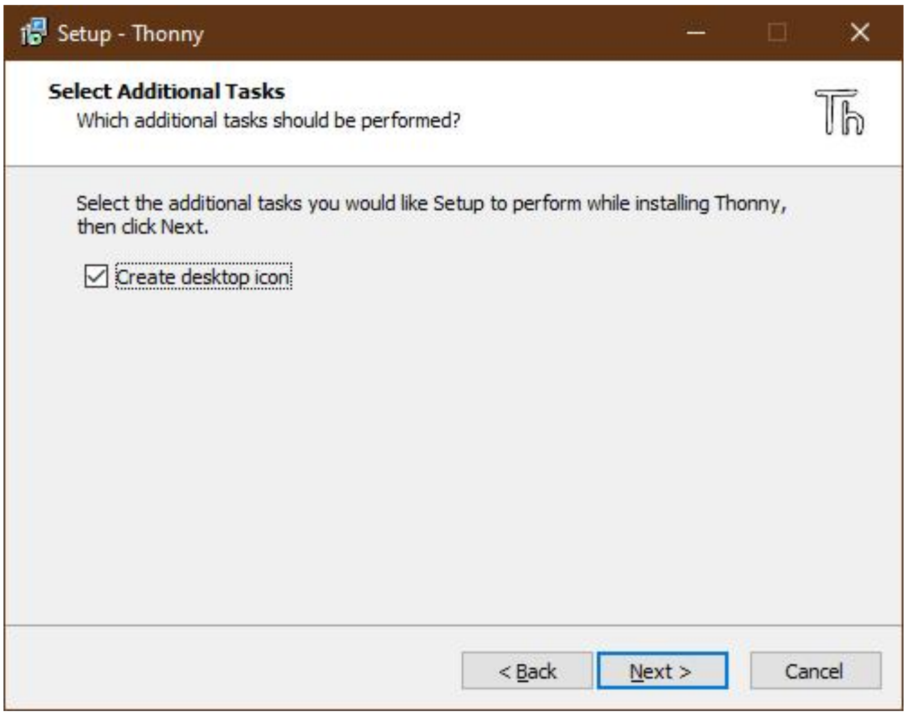
Setup will install Thonny into the following folder.

To continue, click Next. If you would like to select a different folder, click Browse.

C:\Users\root\AppData\Local\Programs\Thonny

At least 98,0 MB of free disk space is required.

Have a desktop icon created, you can later move it to any start folder - Next.


Setup - Thonny

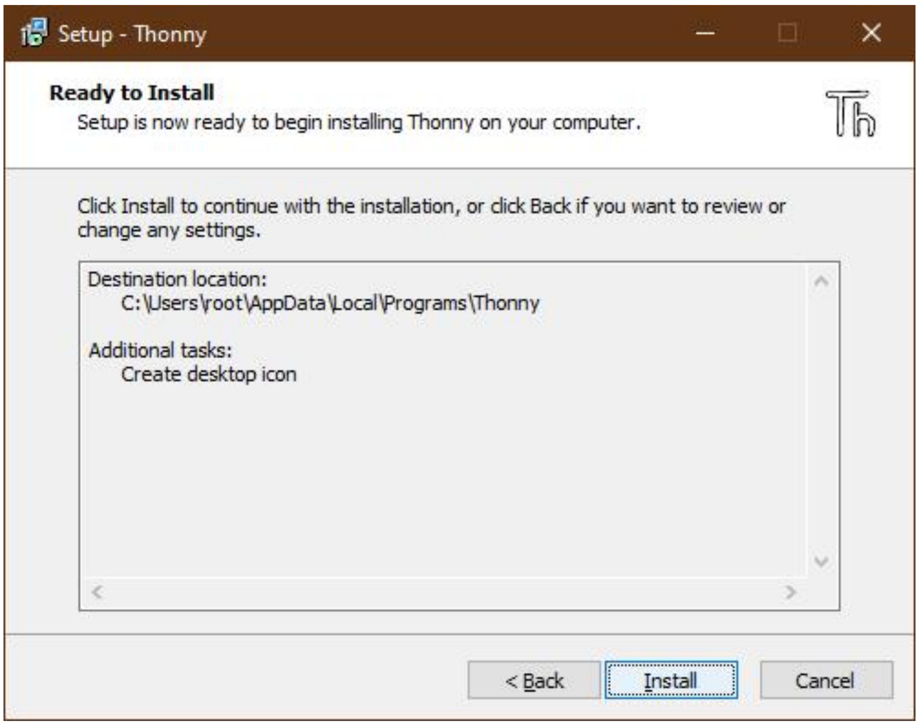
Select Additional Tasks

Which additional tasks should be performed?

Select the additional tasks you would like Setup to perform while installing Thonny, then click Next.

☑ Create desktop icon

The installation process takes a few minutes, depending on the computer. Start it by clicking Install.


Setup - Thonny

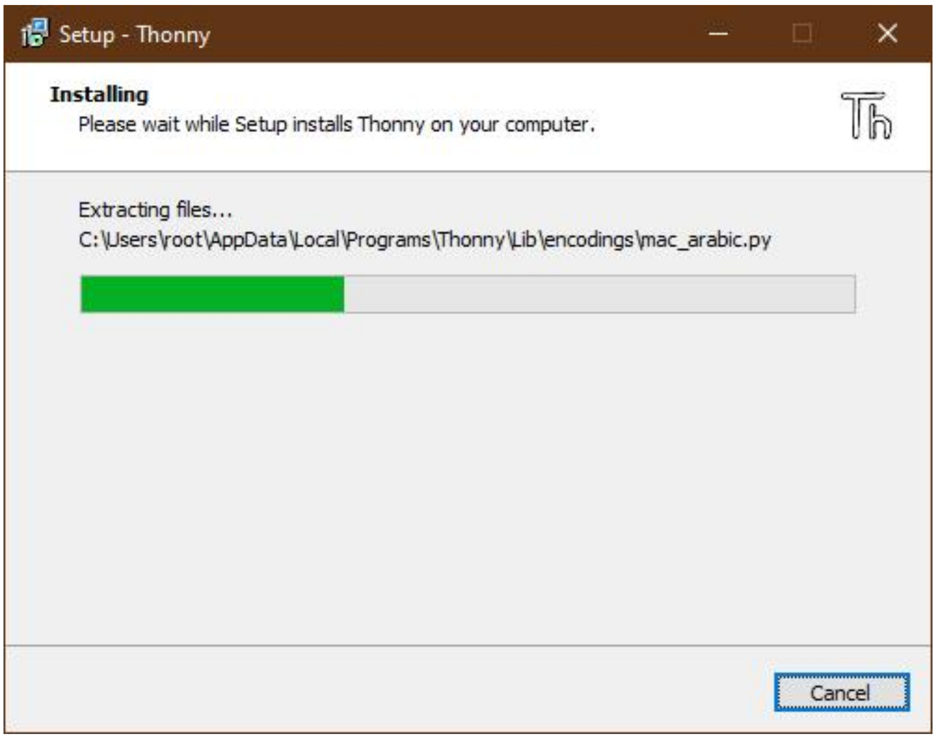
Ready to Install

Setup is now ready to begin installing Thonny on your computer.

Click Install to continue with the installation, or click Back if you want to review or change any settings.

Destination location:
C:\Users\root\AppData\Local\Programs\Thonny

Additional tasks:
Create desktop icon

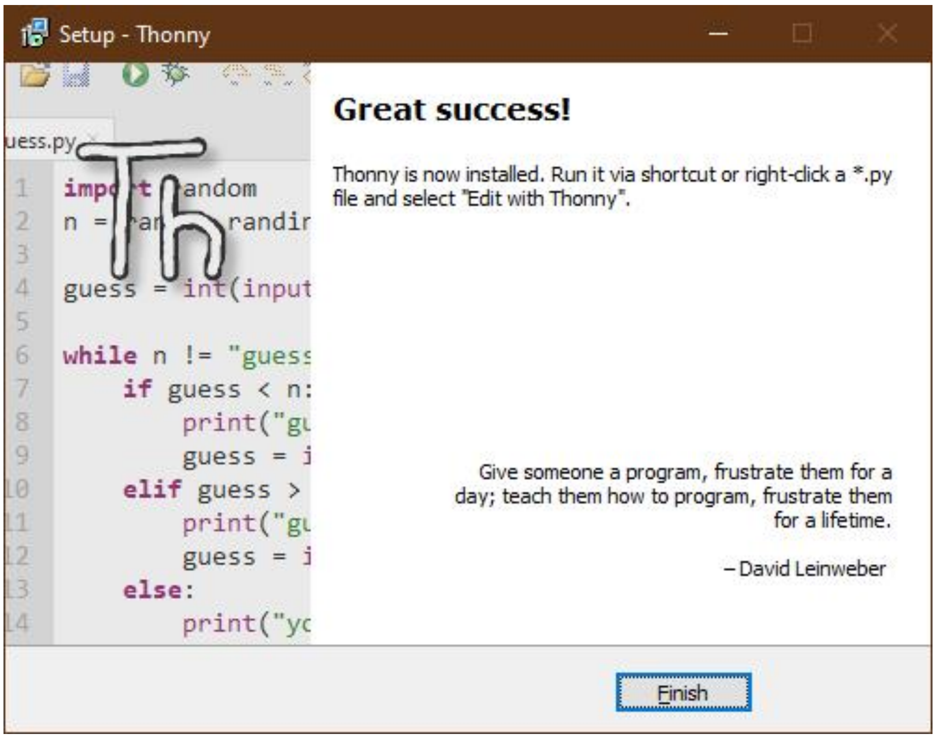

Setup - Thonny

Installing

Please wait while Setup installs Thonny on your computer.

Extracting files...
C:\Users\root\AppData\Local\Programs\Thonny\Lib\encodings\mac_arabic.py

If the following window looks like this, you are done -Finish.


Setup - Thonny

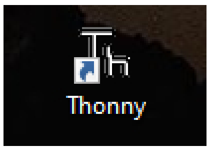
Great success!

Thonny is now installed. Run it via shortcut or right-click a *.py file and select "Edit with Thonny".

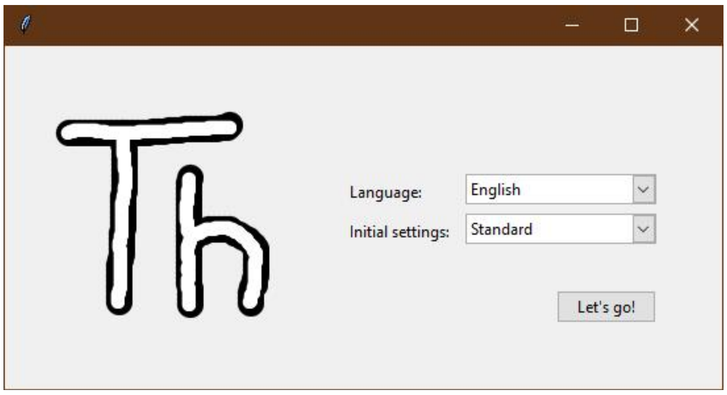
Give someone a program, frustrate them for a day; teach them how to program, frustrate them for a lifetime.

– David Leinweber

This is what the result looks like on the desktop.



Start Thonny with a double click, there are still a few settings to be made.

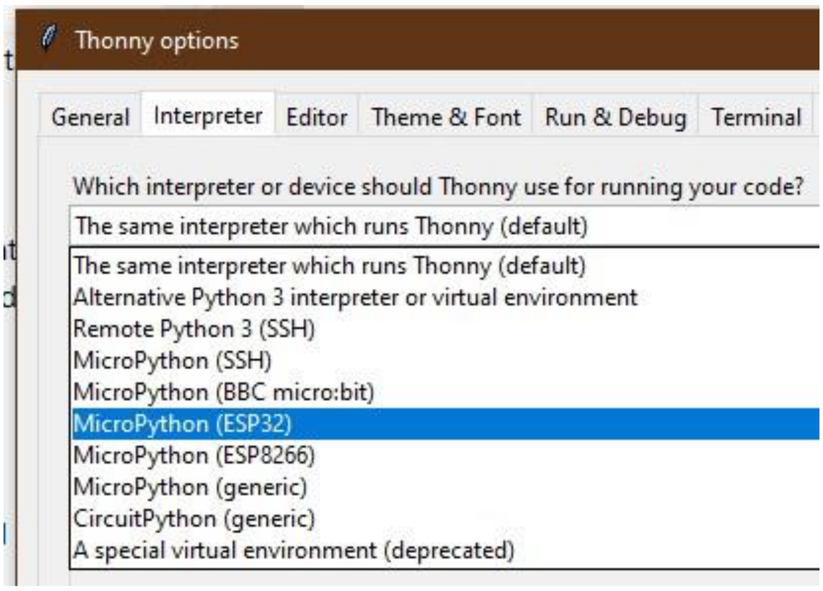

Language: English

Initial settings: Standard

Since the operator language German is still in the beta phase, I recommend using English - Let's go !.

Now is the time to plug the ESP32 into the USB port if it is not already there. Thonny usually recognizes the corresponding connection itself.

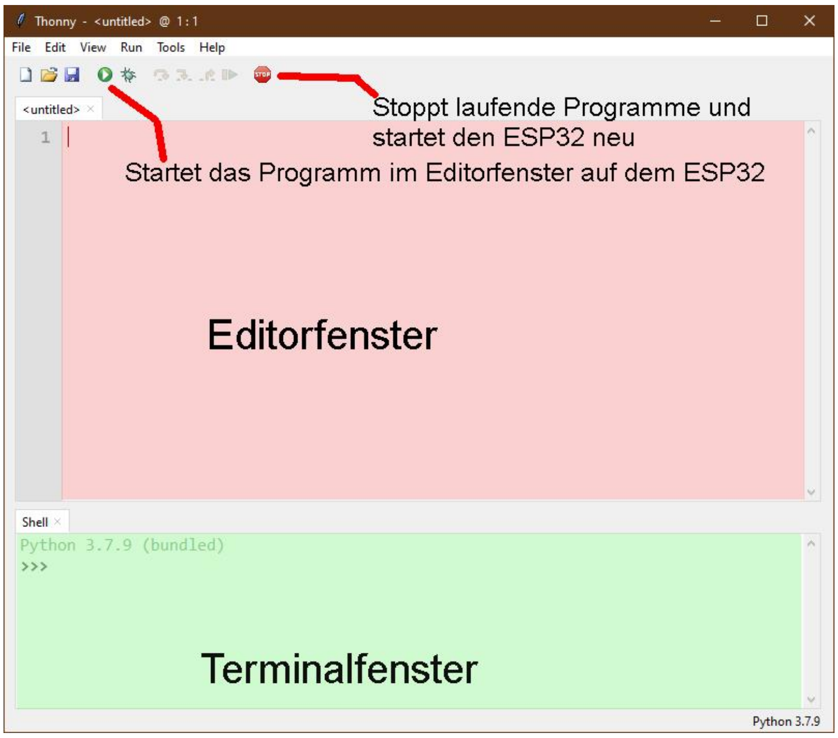
Editor and terminal windows have the same position as with µPyCraft. The menu and buttons are both at the top of the window. Except for Device, the menus are richly equipped with commands. The first three icons on the toolbar are self-explanatory and the other two are labeled. The debug-ger is called with the little bug. It works if The same interpreter which runs Thonny is selected in the settings (Run - Select Interpreter…). This setting may supply incorrect data to the program to be tested if controller and board-specific information, such as the platform, are requested. It is best to set ESP32 or ESP8266, depending on what you are currently working with.



Which interpreter or device should Thonny use for running your code?

The same interpreter which runs Thonny (default)

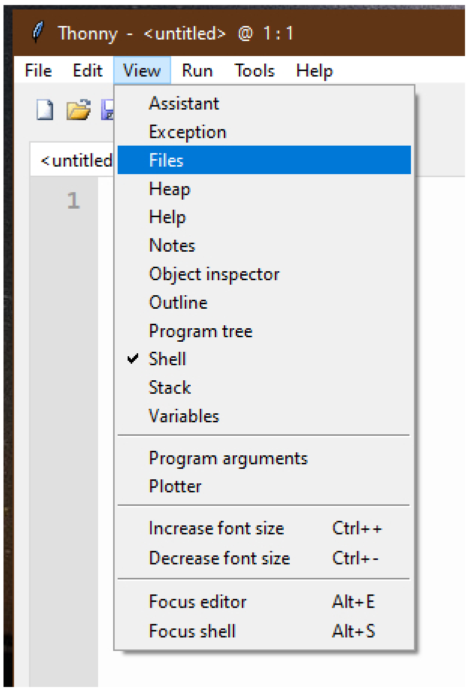
The same interpreter which runs Thonny (default)
Alternative Python 3 interpreter or virtual environment
Remote Python 3 (SSH)
MicroPython (SSH)
MicroPython (BBC micro:bit)
MicroPython (ESP32)
MicroPython (ESP8266)
MicroPython (generic)
CircuitPython (generic)
A special virtual environment (deprecated)

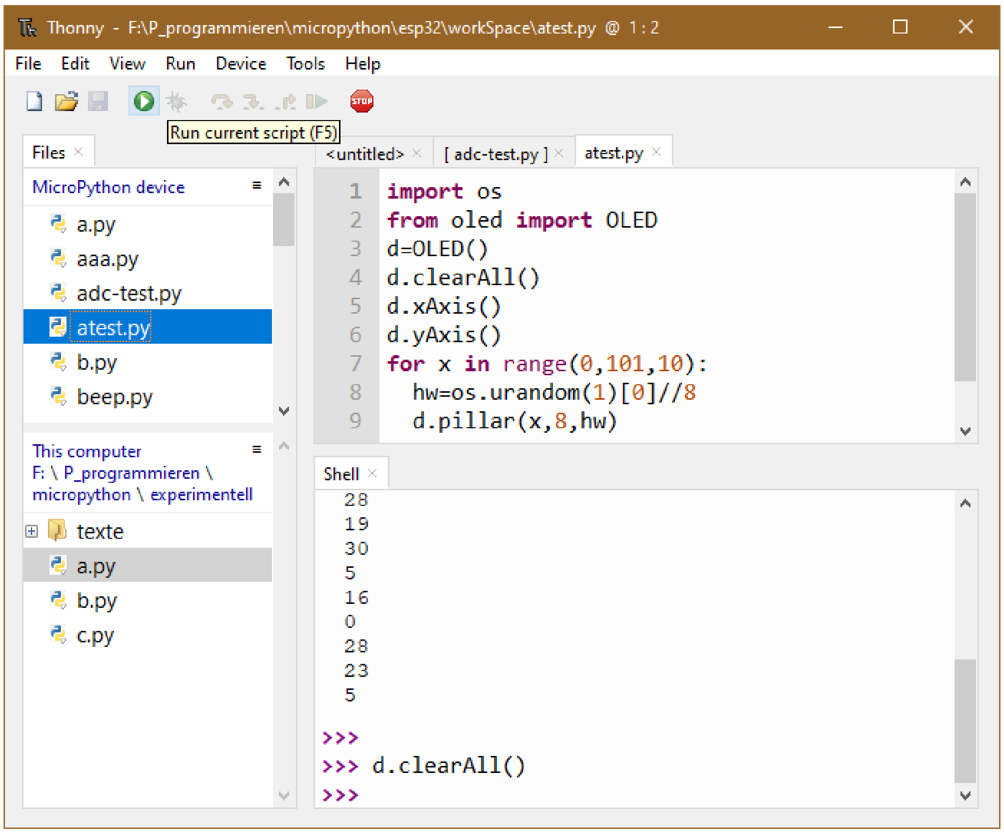

Thonny - <untitled> @ 1 : 1
File Edit View Run Tools Help
<untitled>
Stoppt laufende Programme und startet den ESP32 neu
Startet das Programm im Editorfenster auf dem ESP32
Editorfenster
Shell
Terminalfenster
Python 3.7.9

```
Python 3.7.9 (bundled)
>>>
```

To work, however, the overview of the workSpace and the device directory is missing.

Click on View and then click on Files



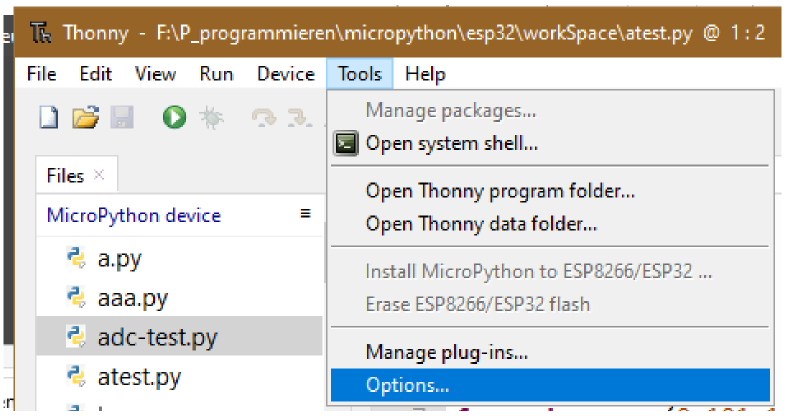


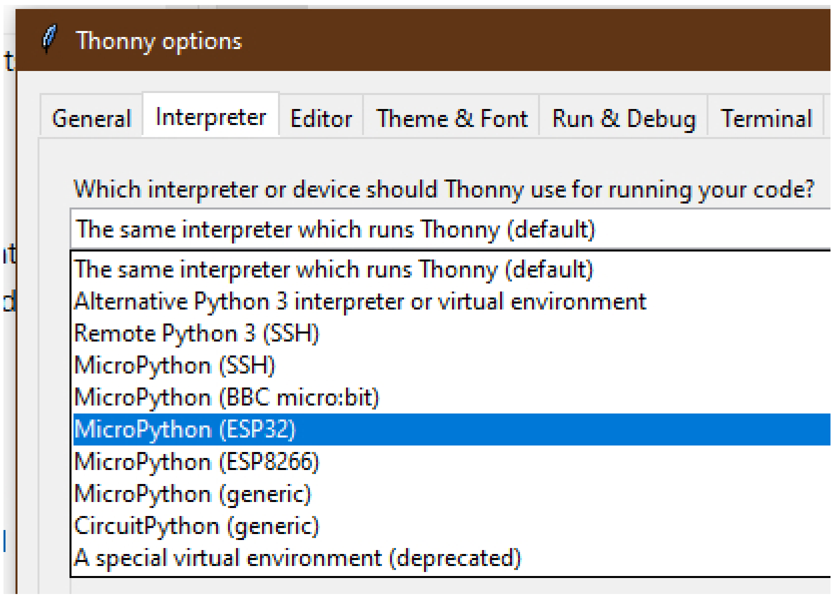
```
import os
from oled import OLED
d=OLED()
d.clearAll()
d.xAxis()
d.yAxis()
for x in range(0,101,10):
  hw=os.urandom(1)[0]//8
  d.pillar(x,8,hw)
```

```
28
19
30
5
16
0
28
23
5

>>>
>>> d.clearAll()
>>>
```

Now it looks very good, the only thing missing is the setting for the board used, if you haven't already done this above. You can choose from Tools - Options. It is best to set ESP32 or ESP8266, depending on what you are currently working with.


Thonny - F:\P_programmieren\micropython\esp32\workSpace\atest.py @ 1 : 2
File
Edit
View
Run
Device
Tools
Help
Files
MicroPython device
a.py
aaa.py
adc-test.py
atest.py
Manage packages...
Open system shell...
Open Thonny program folder...
Open Thonny data folder...
Install MicroPython to ESP8266/ESP32 ...
Erase ESP8266/ESP32 flash
Manage plug-ins...
Options...



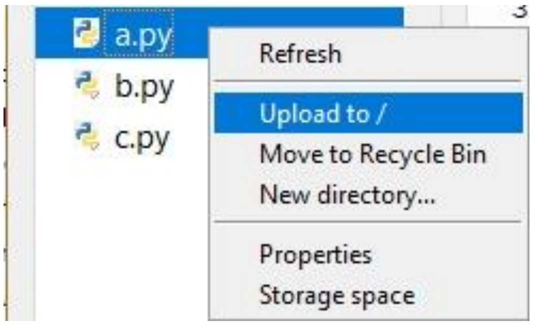
Which interpreter or device should Thonny use for running your code?

The same interpreter which runs Thonny (default)

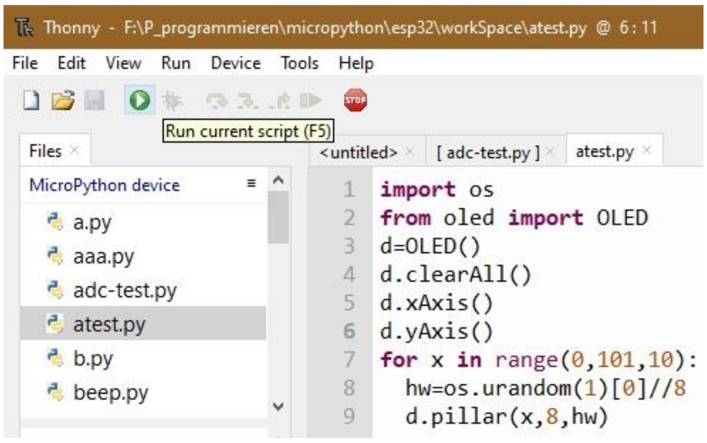
- The same interpreter which runs Thonny (default)
- Alternative Python 3 interpreter or virtual environment
- Remote Python 3 (SSH)
- MicroPython (SSH)
- MicroPython (BBC micro:bit)
- MicroPython (ESP32)
- MicroPython (ESP8266)
- MicroPython (generic)
- CircuitPython (generic)
- A special virtual environment (deprecated)

After OK you can send the first commands to the ESP32 via the terminal.

In the editor you write your first program in Thonny or choose one from the workSpace of µPyCraft. Simply navigate there in the lower part of the Files column. A double click on the file entry opens the file in the editor window. Ctrl + S saves the file. To upload to the ESP32, right click on the file entry and select Upload to /.



To start a file on the ESP32, select the script in the editor and click the green button or simply press F5. With the stop sign you restart the ESP32. Ctrl + C ends the running script. If nothing works, use the reset button on the ESP32.



```
import os
from oled import OLED
d=OLED()
d.clearAll()
d.xAxis()
d.yAxis()
for x in range(0,101,10):
  hw=os.urandom(1)[0]//8
  d.pillar(x,8,hw)
```

By right-clicking on a file entry in the device directory and downloading to…. you can copy a file from there to the workSpace. The respective target directory is also displayed.

You can tell whether you have opened a file from the workSpace or from the device directory in the editor by the Thonny title bar and the notation of the file name in the label of the editor window. Atest.py is open in the workSpace on drive F: in the picture above. The square brackets around adc-test.py tell you that the file was loaded from the device directory. In this case, the headline would also provide information.

In addition to the files overview, other columns can be displayed, e.g. for to get an overview of the variables and their assignment in your program.

Unfortunately, some things in Thonny don't work that well, e.g. the docking of columns via View. Unfortunately, the program also hangs up from time to time, which reduces programming fun. It is therefore advisable to save the work in good time! Another disadvantage is that the firmware has to be flashed separately via the command line with esptool.py. But that is seldom necessary.

The disadvantages are well balanced by the more pleasant workflow. The file transfer is easier than with µPyCraft and the workSpace is much easier to change. With thonny's own interpreter it is even possible to debug scripts step by step. After testing the IDEs, you decide which one best suits your needs. I don't want to talk you into anything.

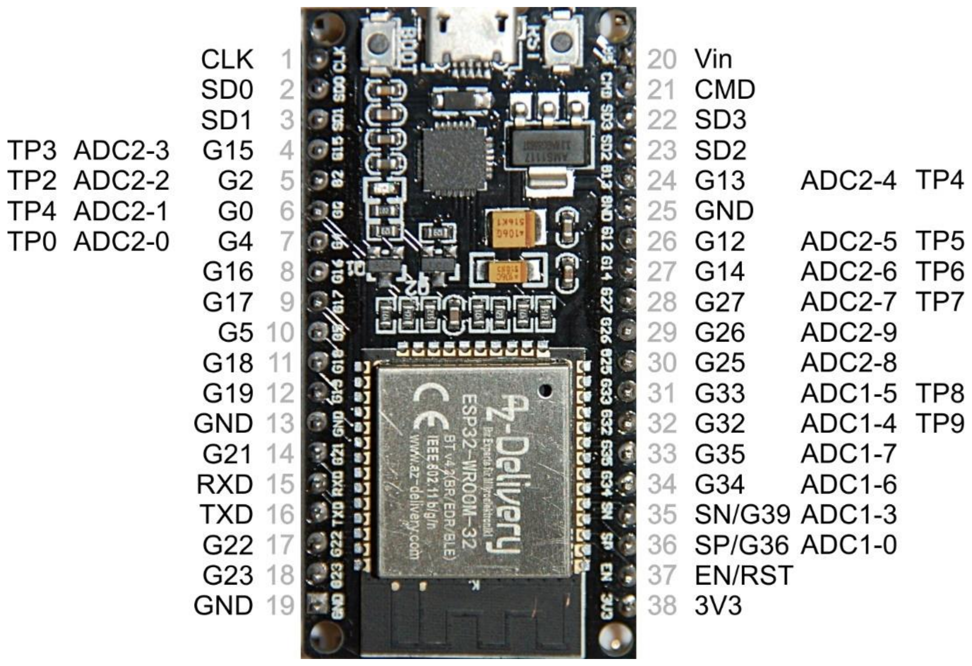
At the moment you have an ESP32 with running firmware and if necessary you can re-flash it with µPyCraft if everything goes wrong. In the next episode I will also show you how the firmware is flashed with esptool.py and the configuration options. At this point, I put you off until the next post so that the hardware can come into play.

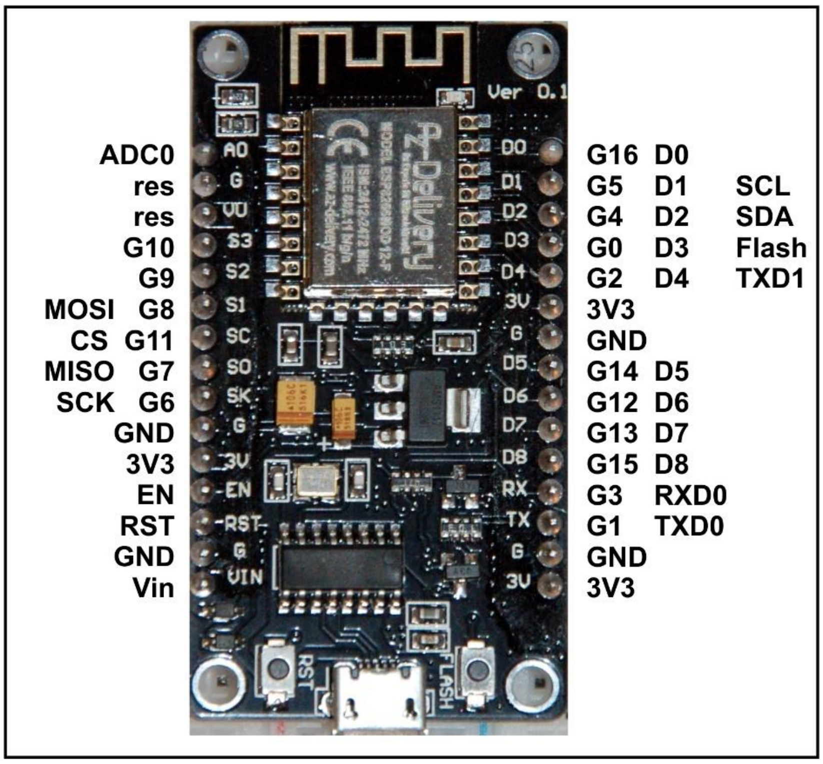
Measure analog signals

While the ESP8266 can only have one analog input, the ESP32 has 16 pins that can (also) be used as analog inputs. These pins are internally connected to two ADCs, ADC1 and ADC2. ADC stands for Analog Digital Converter. These modules convert an applied analog voltage into an integer value, i.e. a whole number. You can read about the technology behind it at Wikipedia. For us this is not so important now and here, we just want to apply it. Therefore I am only giving a superficial description of what happens in the converter.

The ADC module of the ESP32 compares the voltage level at the input with a series of voltages that are generated internally. This comparison voltage is increased in stages. If the levels of the external and internal voltage are the same, the "level number" is written to a memory as the conversion value. This type of memory is called a register. From there it can be called up using appropriate methods. Other methods are used to set the functionality of the ADC.

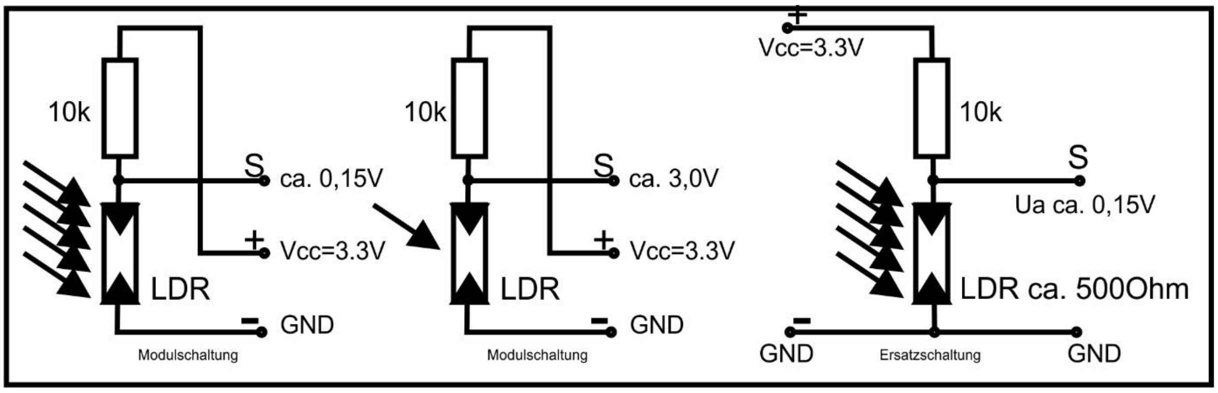
Of the two ADC groups, only ADC1 can be used in MicroPython. The ADC2 unit has other tasks to perform. Therefore, according to the quick overview for the ESP32, only pins 32 to 39 are specified as analog inputs. There is no statement about pins 37 and 38 because they are not led out to the connector strips. The following figure shows other possibilities, but our own tests have shown that in MicroPython only the pins that are connected to ADC1 can actually be addressed. Pins that cannot be used are rejected by the interpreter with an error message. This means that there are no longer 6 analog channels as with the Arduino UNO. However, the AD converters of the ESP32 have a higher bit width of up to 12 bits. Like the Arduino UNO, the ESP8266 works with 10 bits.



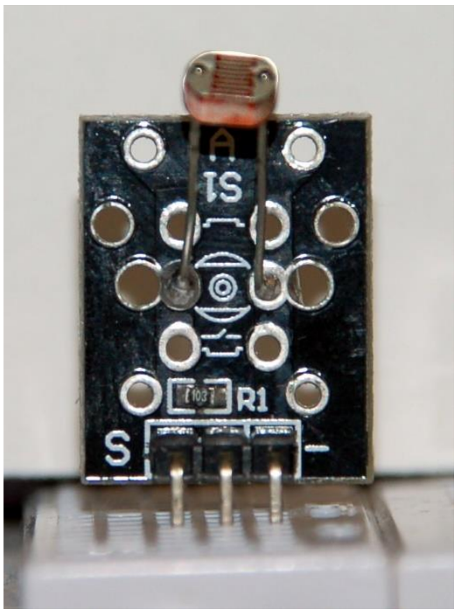


I also found that the measuring accuracy and the linearity of the transducers are not of the best quality. In the project that I will present in one of the next episodes, however, this does not play a role because no absolute values have to be recorded there.

Now build the necessary circurity for the next experiments. You need the ESP32/ESP8266 on the breadboards, the LDR module KY-018 photo LDR resistor, and three jumper cables (male-male). LDR is an acronym for Light Depended Resistor. This means that the resistance value depends on the illuminance. Here is the circuit of the module. First of all I have drawn in the polarity as it corresponds to the imprint on the circuit board.



Pay close attention to the voltages and polarities indicated in the sketches. The number of arrows shows the intensity of the incident light. In addition to the LDR, the board contains a fixed resistor R1 of 10kOhm, which is connected in series with the LDR. This circuit is also known as a voltage divider circuit because the voltage that can be measured between the two resistors is always lower than that which is applied to the ends of the circuit. There are two connection options for the PCB.

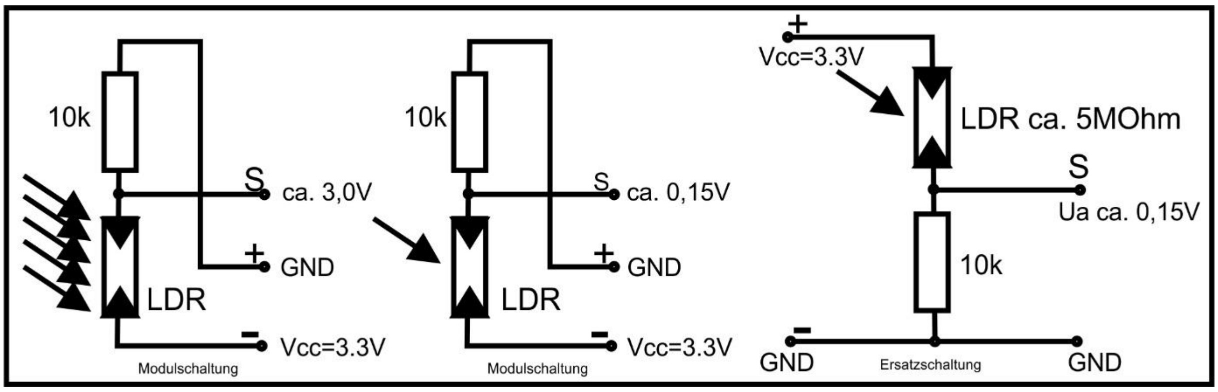


If you stick to the labeling on the circuit board, the positive connection is in the middle and the GND connection is on the right-hand pin labeled "-". This corresponds to the circuit diagram above.

Because the resistance of the LDR drops when the lighting is bright, a relatively low voltage will be set at the voltage divider consisting of the 10k resistor and the LDR at the center tap S, measured against GND. Because the voltages at the resistors behave like the resistance values themselves. That means low resistance value, low voltage, high resistance value higher voltage.

If little light reaches the LDR, its resistance increases up to the megohm range, and the voltage at S goes towards Vcc = 3.3V (above circuit diagram on the left). If high light intensity hits the LDR, its resistance drops well below 500 ohms and the voltage is in the millivolt range (above circuit diagram, middle). This is clearer if you draw the circuit of the module as it is usual - Vcc above, GND below and signal output in the middle (above circuit diagram on the right).

Unfortunately, that's not the behavior I want. I would like to have high voltages and thus high samples when there is a lot of light and, well, vice versa. Therefore I reverse the polarity of the supply voltage at the connections of the board. This is easy with this module because the optical component is actually a "normal" resistor and not a diode or transistor. With the latter, polarity reversal would not be possible, the positions of the components would have to be swapped, i.e. re-soldered. Here is my variant. The arrangement of the components has remained the same, but the polarity of the supply voltage Vcc has been swapped with GND. The equivalent circuit on the right now shows the relationships better.



To explain the behavior of this circuit, I'll take a quick look at physics and math. According to what is commonly referred to as Ohm's law, the voltage U applied to a conductor and the strength I of the current flowing through it are directly proportional to one another. The quotient U / I is therefore a constant (which one likes to define as electrical resistance). This helps us to roughly calculate what to expect. The resistor R1 on the board has a fixed value of 10kOhm = 10000Ohm.

Because the resistors are connected in series, the same current must flow through both. Ohm's law resolved according to I applies to the sum of the resistances and the supply voltage Vcc = 3.3V.

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

However, it also applies to the partial resistance R1 and the voltage Ua against GND. Here, too, the same current I must come out.

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

If the current strength is the same, the two fractions can also be equated.

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

From this equation we calculate Ua.

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

At this point the paths for ESP32 and ESP8266 separate. Read on here if you are working with an ESP32. For the ESP8266 I explain the facts below.

Analoge Signale beim ESP32

Let's use the values for bright lighting, LDR is approx. 500 ohms and less.

HELL

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

$$Ua = \frac{3{,}3V \cdot 10000\Omega}{10000\Omega + 500\Omega}$$

$$Ua = \frac{3{,}3V \cdot 10000\Omega}{10500\Omega}$$

$$Ua = 3{,}3V \cdot \frac{10000}{10500}$$

$$Ua = 3{,}3V \cdot 0{,}95 = 3{,}14V$$

DUNKEL

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

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

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

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

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

For little light, the resistance of the LDR increases to high values up to the megohm range. In the calculation example I used 5MΩ. M = mega stands for the factor 1,000,000.

This is what it looks like on our breadboard. The LDR module was added below and wired according to my variant.

The preparation and the acquisition of measurements for this experiment can be done with a few simple commands. We import the Pin and ADC classes from the machine module and the time and sleep classes from time.

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

We create an instance ldr of the ADC class. The name is not important. But the call to the ADC class constructor is interesting. The parameter is not a simple pin number but an instance of the machine.Pin class. Pin (34) creates a pin object that is passed to ADC .__ init__. GPIO34 is now linked as an analog input with ADC1 of the ESP32.

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

Querying the analog value is even simpler and is similar to reading in digital values. Instead of value () it says read (). The output value 523 is the result according to the Delfault settings for the ADC object, which I have not changed

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

The ESP32 has two more delicacies that the ESP8266 does not have to offer. With the latter there is only the analog input A0. The resolution there is fixed at 1024 bits and the voltage is limited to a maximum of 1V according to the Quick reference for the ESP8266. I describe the procedure for the ESP8266 further below.

With the ESP32 you can already specify the resolution in 4 levels. With the help of 4 constants that are defined in the ADC class, switching takes place using the instance method width (). Each instance can therefore be assigned a different value.

Tabelle 1

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

Furthermore, the internal attenuator provides 4 ranges for the input voltage, which can also be switched using the atten method using parameters. The referencing of the constants via the class name ADC also tells us that they must be class attributes. The error message when trying to assign a value to the attribute from the instance ldr supports us in this view.

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

The listing of the members of the ADC class clearly states it. Here we also see that apart from read, read_u16, atten and width, no other methods are available in this class.

```
>>> 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']
```

Table 2 shows the value range of the input voltage and the values hidden behind the name for the attributes.

Tabelle 2

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 Kon-stante	0	1	2	3

In practice it looks like this. With the same ambient brightness, I received the specified values for the following settings

```
>>> 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
```

If you put these values in relation to the set bit width (512, 1024, 2048, 4 96), the result is an almost constant value of 0.157 +/- 0.004. Regardless of the resolution, the measured value can be reproduced very precisely, provided that several measurements are made.

If you multiply the quotient 0.157 by the set maximum detectable voltage of 3.3V, you get the voltage actually applied to pin32: 3.3V * 0.157 = 0.518V. Unfortunately, this value is "slightly" off. The DVM (digital volt meter) says 0.68V. That is a 25% deviation below. At least the deviation in the upper range is reduced to 5%. That means that the scale is then not even linear.

Correcting this is another construction site that may one day be the subject of a blog post of its own.

Let us note the following formula at this point, which allows the voltage at the analog input to be calculated (approximately) from the digital measured value. If the input voltage exceeds the maximum measuring range, the ADC result is always bit width 1. Attention: The absolute maximum voltage at the inputs is 3.6V. If you go beyond that, the ESP32 will say goodbye to nirvana.

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

$$\text{Bitbreite} = [512, 1024, 2048, 4096]$$

For the main project in the last part, the inaccuracy found is of no interest. Much more interesting, however, is how many measurements the ADC can achieve per second or better per millisecond. This is determined by the following program adc.py.

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")
```

You are all familiar with the structures in it, so I leave the explanation to you. If you remove the diamond in the two lines that have been commented out (correct the indentation), the program is indeed slowed down enormously, but you get a lot of ADC values in the terminal window and can thus check whether the LDR is working properly. To do this, shade the LDR or illuminate it with a bright lamp. In this way you can estimate under which light conditions which measured values come out.

For me, the result of the speed measurement is between 12,500 and 13,500 samples per second with both 9-bit and 12-bit resolution, i.e. approx. 13 samples per ms. It's not great, but it is good enough for my purposes.

Analog input signals for the ESP8266

This is about the handling of the analog input and the circuit on the ESP8266. Because there are no configuration options for the ADC, this is done quickly. But there is one important circuit change from the ESP32.

A major difference to the ESP32 is that the constructor for the ADC object does not expect a pin object as a parameter, but simply a 0. The following instructions demonstrate how to do this.

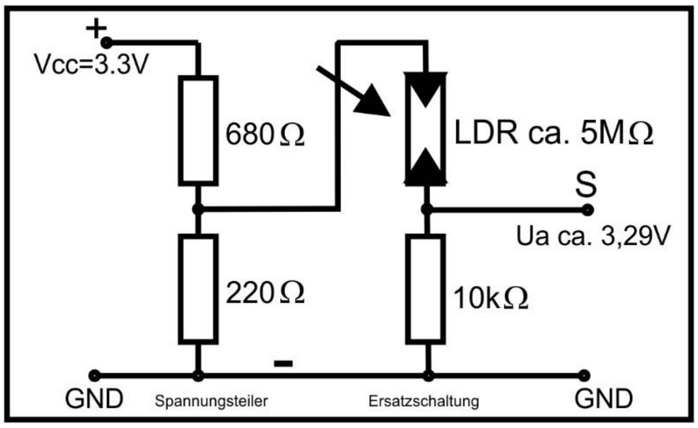
```
>>> from machine import ADC
>>> a = ADC (0)
>>> a.read ()
156
```

So you only have one analog input, which is assigned to object a in the example. As with ESP32, the ADC method read () reads in the value between 0 and 1023. But there is one thing you need to consider.

The maximum voltage at pin A0 is 1V

At least that's what the Quick reference for the ESP8266 says. Appropriate protective measures such as voltage dividers and limiting diodes can be used to ensure that this condition is reliably met. Therefore the circuit of the ESP32 has to be changed. The circuit board of the LDR must not get a higher voltage than 1V, then it cannot deliver a higher voltage.

I solved this with a relatively low-resistance voltage divider.



The calculation of the voltage between the resistors of 680Ω and 220Ω gives a value of 1.07V.

Caused by the tolerances in the resistors, I measure a voltage of 0.91V in my circuit. But a value of 1.07V is also acceptable, because a further reduction in the voltage is to be expected from the voltage divider from LDR and R1 and values around 500Ω are only obtained at the LDR with very bright lighting.

HELL

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

$$Ua = \frac{1{,}07V \cdot 10000\Omega}{10000\Omega + 500\Omega}$$

$$Ua = \frac{1{,}07V \cdot 10000\Omega}{10500\Omega}$$

$$Ua = 1{,}07V \cdot \frac{10000}{10500}$$

$$Ua = 1{,}07V \cdot 0{,}95 = 1{,}01V$$

DUNKEL

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

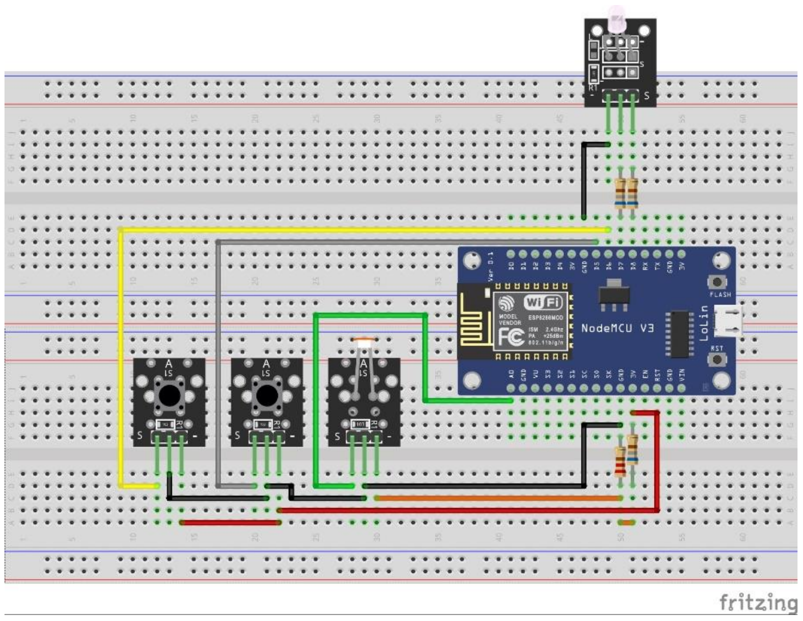
$$Ua = \frac{1{,}07V \cdot 10000\ \Omega}{10000\Omega + 5.000.000\ \Omega}$$

$$Ua = \frac{1{,}07V \cdot 10000\ \Omega}{5.010.000\ \Omega}$$

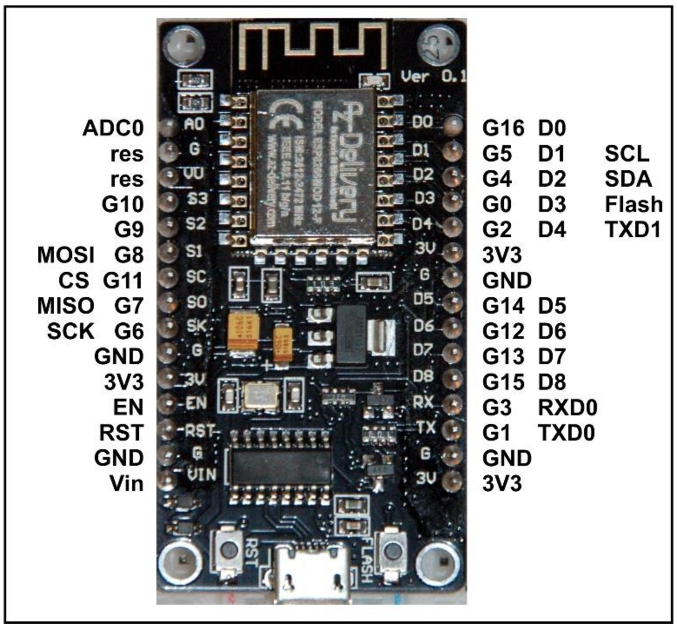
$$Ua = 1{,}07V \cdot \frac{10.000}{5.010.000}$$

$$Ua = 1{,}07V \cdot 0{,}02 = 0{,}007V$$

The circuit for the ESP8266 also looks different from the ESP32 due to the different pin offer. In anticipation of the next chapter, the buttons that replace the touchpads on the ESP32 are also built in. They are connected to GPIO12 and GPIO14. Instead of the RGB-LED, I use the duo-LED here. The two anodes are each connected to GPIO13 (red) and GPIO15 (green) via 680Ω. The connections GPIO5 (SCL) and GPIO4 (SDA) will be used later for the I2C bus. GPIO2 is internally connected to the built-in LED.



For better orientation, here is the pin assignment of the ESP8266 again.



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")
```

The speed measurement of the ADC gave me approx. 5000 samples per second. That is significantly less than the approximately 13,000 for the ESP32.

I noticed something else. I was able to cheer up the ADC values through extremely bright lighting up to a maximum of 290, then it was over. This aroused my suspicion that the LoLin V3 I was using at A0 had a maximum voltage of 3.3V. The extrapolation of 0.91V at the voltage divider and the measured value 290 to the maximum possible of 1023 confirmed this suspicion with 3.21V. This means that I can connect the LDR board to Vcc = 3.3V and connect its output directly to A0 of the ESP8266.
But before you do the same with me, try out the circuit presented above as a precaution.

A subsequent closer examination with the magnifying glass and the DVM revealed that all my ESP8266 boards (LoLin V3, Amica, D1 mini) have a voltage divider with 220kΩ and 100kΩ on the A0 connection of the board. The voltage is divided well into thirds with the board's own resources so that it does not overload the actual input on the ESP12F. The input impedance is fixed by the two resistors to 320kΩ and thus lower than that of the high-impedance input of the ESP-12F.

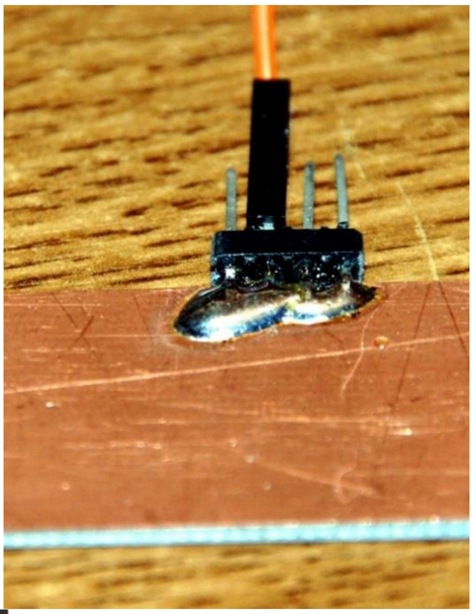
Touchpads

If you are using an ESP8266, I have bad news. This controller has no sensor inputs for touchpads. This means that the parts of this chapter that only work in conjunction with touchpads cannot be reproduced experimentally with the little brother of the ESP32.

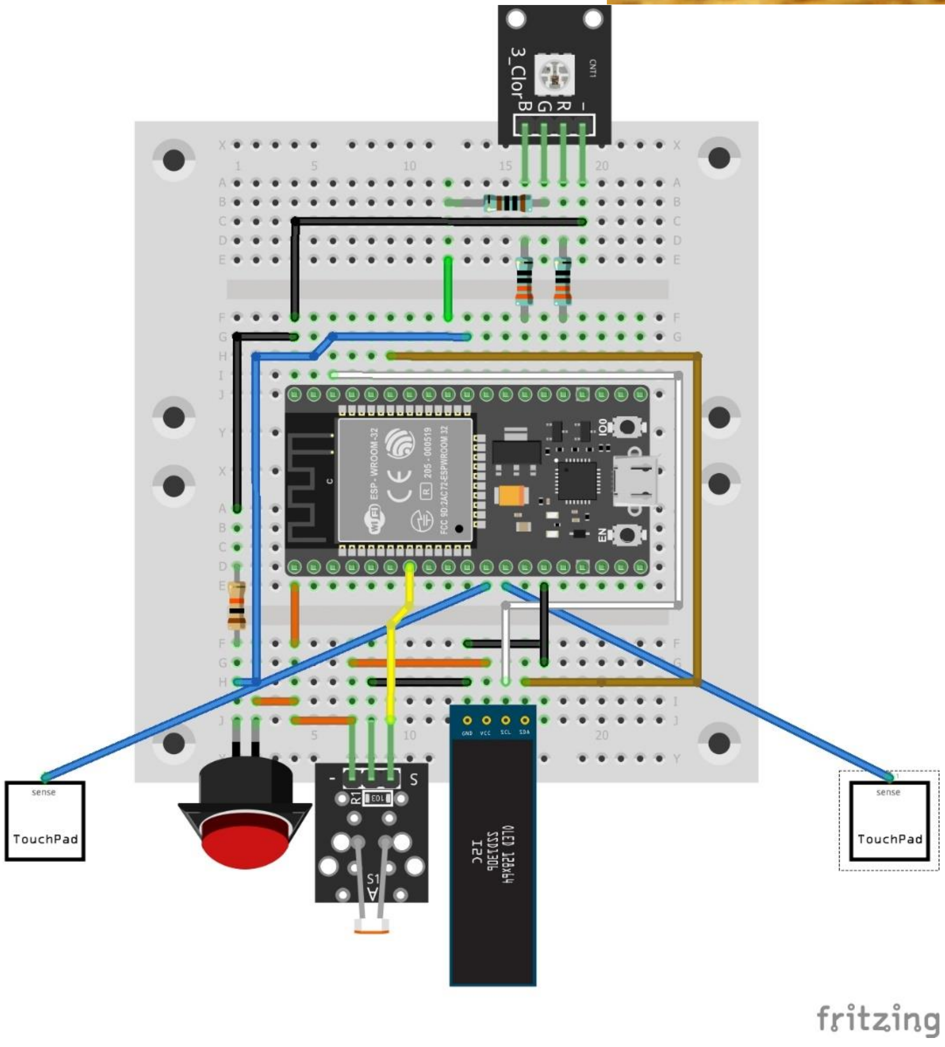
But there is also good news. The essential functions from the touch8266 module adapted for the ESP8266 also work with normal buttons. These two functions will be used in the next blog post to control the OLED display. The use of normal buttons will be discussed shortly after handling the touchpads for the ESP32.

Touchpads am ESP32

Now it is time for ESP32 users to provide the pieces of sheet metal or remnants of circuit boards with the connector strips, because we now use them as buttons that react to mere touch. Then connect one part to pin GPIO27 and the second to GPIO14, right next to it. At the beginning of this article, I already told you the trick with the clothespins when no soldering tool is within reach. By the way, a paper clip as a fastening clip is also possible.



The next picture shows the connection of the pads in the circuitry.

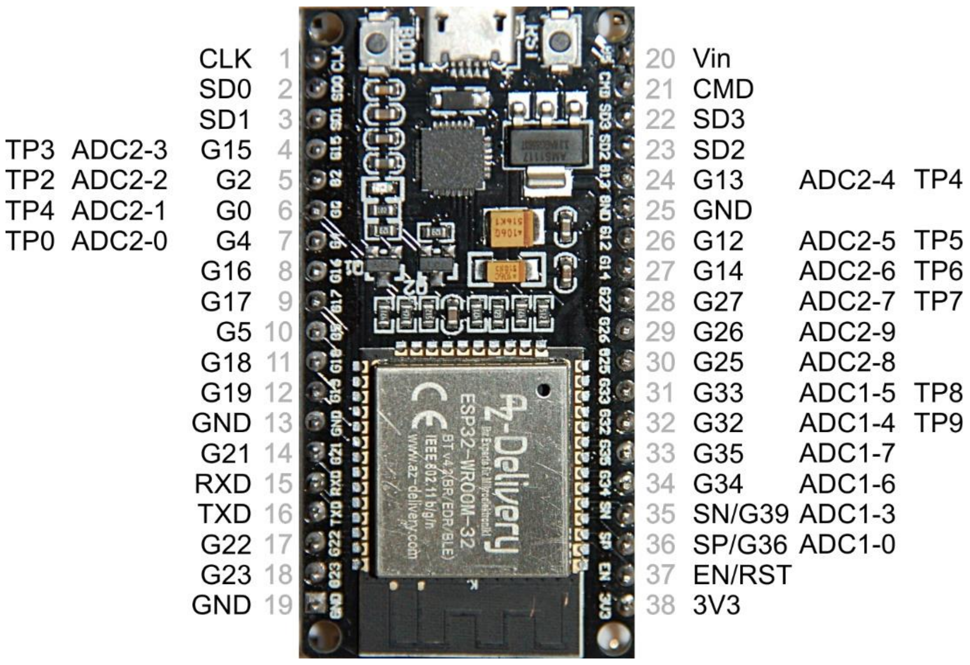


Different GPIO pins can be defined as a touchpad. If you simply connect a piece of sheet metal or another conductive surface to these inputs, you can determine that touching this surface with your finger when you query the pin as a decrease in the query value. Preliminary tests are necessary to determine a limit value for the response. Because the switching threshold is not firmly defined. The value without touch flows almost smoothly into the value with touch. For an application it is therefore essential to set an integer value as the limit value, which is not in the gray area around maximum and minimum and can therefore guarantee reliable switching behavior. If the value is set too low, the switching process may no longer be triggered; if it is defined too high, random disturbance events can trigger the switching process unintentionally. The latter would be fatal in the case of door opening control. The use of touchpads could therefore be classified in the area of fuzzy logic, where there can be not only 0 and 1 but also any number of intermediate values.

Not all GPIOs are touch-compatible, just the following. The graphic also shows the location, TP stands for touchpin.

Tabelle 3

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



Here is a program that you can use to test touchpads.

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()
  # gegf. Fehler abfangen
  except ValueError:
    print("ValueError while reading touch_pin")
  print(touch_val)
  utime.sleep_ms(200)
```

The TouchPad class is part of the machine module and must be imported for the use of touchpads, as well as the Pin class. We create a pin object on GPIO27 and assign it to the touch object using the constructor of the TouchPad class.

Because it can be read at various points that errors can occur when querying touch-pads, which hang up the whole system, we secure this with error handling. I will explain how they work in more detail below. Then output the read-in touch value in the terminal, wait 0.2 seconds and do the whole thing all over again. This time I use a delay timer in milliseconds, sleep_ms ().

Untouched I get a value of over 400, with touch it is around 60 to 80. For this hardware I would therefore set a threshold of 150, which is well below 400 and also well over 60.

We will use the two pads in the next episode to control the contrast of an OLED display that we build into the hardware. To make it easier to use, I wrote a class TP and saved the whole thing as a module in the touch.py file. In addition to the constructor, the TP class provides three useful methods. But one after anonther. First of all, here is the text of the touch module.

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 < threshould 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.threshould:
        return val
      current = time()
      if delay==0:
        end=current+10
    return None

  # delay = 0 wartet ewig und gibt gegf. einen Wert > threshould 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.threshould:
        return val
      current = time()
      if delay==0:
        end=current+10
    return None
```

Before I discuss the module and class TP, a few comments on modules and classes in advance. I'm just scratching the surface here, in the next episode we'll work a little deeper into the matter.

In the second part you learned how to use the modules and classes built into Micro-Python. Now we will deal with the first example of a custom class that we have crea-ted. I have named it TP and it is contained in a module called touch. The touch mo-dule is represented by the touch.py file of the same name. The file name is always the name of the module.

If you combine variables, constants and functions in such a file and save them, it is a module, but by no means a class. You will learn a lot more about the subtleties in the next episode. At this point, I'll just introduce you to the two essential terms class and self so that you can understand how touch and TP work.

The reserved word class introduces a class definition, just as def introduces the defi-nition of a function. What then follows is the blueprint mentioned in the second part.

Our blueprint begins with the declaration and definition of the boundary attribute. Li-mit provides a default of 150 for the limit value when detecting contact. The class at-tribute is available in all objects and has the same value everywhere. This is specified once when the class is defined and should not be changed manually or by instance methods. "Should" is at this point, because while it is possible in MicroPython to change the value of class attributes, doing so can potentially bring disaster and ugly errors. MicroPython is just not C / C ++.

self - is a reserved word that puzzled me the most a long time ago when I first looked at OOP (Object Oriented Programming). What is self doing?

Well, self simply indicates that a method or attribute refers to an object itself and not to the class. Instance or object attributes can, despite having the same name, take on completely different values from object to object that do not interfere with one another. Such instance attributes are declared in a special method and assigned va-lues. This method is named __init__ and is called a constructor. This has no other task than to declare instance attributes and assign them start values. These values can themselves be references, aka references, to other objects. init stands between two opening and closing underlines. We will see this in a moment using the example of class TP.

In the case of attributes, this self-reference is indicated by prefixing self. specified. For functions, aka methods, self is the first entry in the parameter list. When calling functions within the class definition, as with attributes, self. placed in front of the func-tion name.

When using, aka referencing, methods and attributes outside of the class definition, self. replaced by the name of the object that was created by calling the constructor. Now you can see the connection. self simply replaces the names of objects in the class definition that may not be known at this time.

The constructor TP (), in the form of the method __init__, has a position parameter and an optional parameter in addition to self. The first parameter takes the GPIO number used and must always be specified. The optional second parameter can have a different value for the response limit. If it is omitted, the default is used in limit. The two instance attributes number and threshould are set up for the query or assign-ment in a program. tpin is used internally in method definitions of the class and direct access to the touch object. threshould can be queried during a program run independently of other objects of class TP, but it can also be assigned new values. The programmer or user is responsible for ensuring that no nonsensical values are given (not negative, no floating point values, not greater than 255). This will also be the subject of a homework assignment.

By the way, everything that is in a class definition is visible to the outside, variables and methods. The term private does not exist in MicroPython. I have already stated above that MicroPython is not C / C ++. Now we come to the discussion of the other methods that the TP class makes available.

The getTouch () method tries to read in a touch value and returns it if the request was successful. Otherwise an error message will appear on the terminal and the return will be None. This is another demonstration of how to catch an error with try and except. The following way of speaking gives an impression of the function of the structure. "If the attempt to get a value from the touch pin is successful, skip the except part, otherwise, if a value error (ValueError) has occurred, do what is in the Except block." ValueError can also be omitted, then except jumps to all possible errors.

waitForTouch () and waitForRelease () work in a similar structure. The delay parameter is mandatory and must therefore be specified. If 0 is passed as the value, the method waits until the day of the day for the touch or the release. In a conditional expression, the end of the waiting time, end, is set to the start value increased by delay, if delay is greater than 0. Otherwise start + 10 is assigned. The running time is set to start and then it goes into the loop. We pick up the actual value on the pad. If the value is valid, i.e. not None, and also less than the limit value (for waitForTouch ()) or greater than this (for waitForRelease ()), then we are done and the read integer value is returned. If the event does not occur within the time specified with delay, the return value is None. The calling program can thus check whether the expected event has taken place.

If the value of the pad is None or is it in the wrong range, we adjust the running time and then check whether the delay is approximately zero. If not, the loop starts a new run until the current time is greater than or equal to the value in end. This will never happen if delay = 0, because then the end is pushed out 10 seconds with each run and thus never reached.

Of course we will test the functionality of our first module right away. In the following I use Thonny. In direct mode (command line) I first import the class TP into the global namespace of REPL and declare an object t, which I define for GPIO27. Declaration and definition are different terms in OOP. With the declaration, the MicroPython interpreter (in the Arduino IDE, the compiler) reserves memory space for an object that does not yet have a defined content. This is only ensured by the definition with a value assignment. While the two steps can be carried out separately in the Arduino IDE, in Python they have to be done in one go, as is the case here with the generation of t = TP (27). With this definition, the limit value is automatically set to 150 because I did not specify a second parameter. I check this by querying the instance attribute threshould.

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

I don't touch my touchpad and read the value.

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

Now I touch the piece of board and get the value again.

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

Now I set the limit value for this configuration to the arithmetic mean of the two values. So I assign this value to the threshould attribute of object t. Because the result has to be an integer (aka whole number), I use integer division with // as an operator. Then I ask for the value again.

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

Now I create another touch object s and immediately check the limit value, which, as expected, has the value 150. The limit of t is still at 270.

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

>>> t.threshould
270
```

I can query the class attribute limit for each object but also for the class TP itself.

```
>>> t. limit
150
>>> TP limit
150
```

Now it's getting exciting. I'm doing something that shouldn't be done. I am assigning a new value to the class attribute Limit. Set and query, of course!

```
>>> TP.limit = 120
>>> TP limit
120
```

Query about the object t, of course.

```
>>> t. limit
120
```

If I can query the value via the object, what happens now?

```
>>> t limit = 200
>>> t. limit
200
```

Instead of 200, there should have been an error message that this is not possible. How can that be explained? The next two queries again fit perfectly into the scheme.

```
>>> see limit
120

>>> TP limit
120
```

Well, a class variable such as limit cannot be changed from within an object, that would be fatal because the other objects of the same rank would not notice anything.

```
>>> t limit = 200
```

This expression creates a new instance variable in the object t, the value of which can be queried there. Of course, neither the class nor other objects of the same class have access to this value. The listing of the attributes of t and s together with their values shows this clearly. The variable __dict__ contains a list of name-value pairs of the attributes and objects of the instance of a class. Such a list is called in the Python Dictionary. The content is in curly brackets.

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

In the excerpt from the list for class TP you can see that the value of the class attribute limit was set to 120.

```
>>> TP .__ dict__
{'waitForTouch': <function waitForTouch at 0x3ffe5840>,…, 'Limit': 120,…}
```

If I now create another touch object, I get this seemingly strange result.

```
>>> a = TP (4)
>>> at the limit
120
>>> a.threshould
150
>>> a .__ dict__
{'threshould': 150, 'number': 4, 'tpin': <TouchPad>}
```

It is clear that there is no border in the object. But why does threshould now have the value 150 instead of 120? The contradiction, which is not one, can be easily explained. During the import, the interpreter translates the module line by line into a program that can run on the ESP32. At this point in time, a reference to the limit value is permanently built into the parameter list, which at this point is 150 and is stored somewhere in the RAM memory. MicroPython refers to exactly this value, not to the new value 120 of limit stored elsewhere in the memory, when a new object is derived from the class and no value is specified for the second parameter.

To avoid this confusion, class attributes should not be changed manually afterwards, if possible, but in the class definition if necessary.

Let's conclude this series of MicroPython with the ESP32 with a test of the time-controlled TP methods.

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

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

In the first case the touchpad was touched before the 5 seconds had elapsed, in the second case not. Thus None was returned by the method and nothing was output in the terminal.

Stop! I almost forgot the ESP8266. What does the touch8266 module look like?

Touchpad replacement for the ESP8266

Instead of the more complex query of the sensor value in getTouch (), there is a simple reading of the digital button input pin. There is no limit value, which simplifies the corresponding conditionals in the wait functions. The constructor is simpler and so is the getTouch () method.

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 < threshould 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 > threshould 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
```

As I have already mentioned, the buttons are located on GPIO pins 12 and 14. The use of the touch8266 module and its TP class is almost identical to that of the touch module for the ESP32. Except that the constructor doesn't have a second parameter. The methods have the same name and function. Only the attribute threshould does not exist.

MicroPython even provides the option of automatically recognizing the controller used and then addressing the correct module. The short test program for the ESP8266 makes use of this. It also works for the ESP32 without modification.

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.threshould
```

Kaffee-Pause!

****

In the next post we will connect an OLED display and an active buzzer to the ESP32 and control the contrast function of the display with the help of the touchpads and the LDR. Of course there will be more modules and we will continue to deepen the information about classes. I also promised you to explain flashing with esptool.py.

Here are the links to the first two posts in case you want to read something there again.

Folge 1
Folge 2

Episode 3 is also available as a PDF for Download.

New Homework

1. Write a sequence that checks the limit value parameter in the __init __ () method for valid information. The value must be an integer, positive, and less than 256.

2. Create a plausibility check in __init__, which checks the validity of the GPIO number for the touchpin before the object is created. For this purpose, define a list with the valid pin numbers. Use the keyword in to check whether the entry corresponds to a value in the list. Example for the command line:

```
>>> input = 4
>>> touchlist = [15,2,0,4,13,12,14,27,33,32]
>>> entry in touch list
True
```

3. Can you change waitForTouch so that if you touch it within the runtime, instead of the value of the touchpad, the delay from the start of the method to the time of touch is returned?

4. Expand exercise 3 into a reaction test device by lighting up an LED at the beginning of the runtime, which is switched off again after touching it.

Form of increase:

Let one of three LEDs light up randomly, whereby only one may be touched. If you type on the wrong LED, you will be degraded. The right touches are counted.

5. What happens with the ESP32 and the ESP8266 if you uncomment the last line in touchtest8266.py?

6. Find a way to catch the error that occurs when querying the threshould attribute when using the touch8266 module.

Solutions to the homework from part 2

• What actually happens if you don't enter a slash in the URL? Which value is then in slashPos?

The server script server1.py is completely insensitive here because it does not parse the request. You always get the same answer, regardless of whether you append a path or parameter to the URL or nothing at all. The only important thing is the port number, otherwise the server will not feel addressed.

In server2.py, a (pseudo-) path specification is required for the switching process - / on or / off. If you do not specify a path or even omit the slash, the same thing happens as with any other path specification that does not begin with / on or / off. The server intercepts this with an error message. A missing slash in the URL line is automatically added by the browser, so it always arrives at the server and is therefore after the line

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

the value of slashPos always equals 4. Because no error can occur due to a non-existent slash, you don't have to do anything special to handle it. The 3 characters following the slash in the 'GET / HTTP / 1.1…' request are 'HT', and they are also output with the error message: Invalid command: HT.

• If you have already knitted HTML pages by hand. Can you make a simple page with links that can be used to switch the LED?

Add the following lines to the HTML text in the <body>… </body> area.

```
<a href=http://8.8.8.8:9192/ein> Switch on the LED </a> <br>
<a href=http://8.8.8.8:9192/aus> Switch off the LED </a> <br>
```

• Can you include the IP and port number of the client in the response text from the server?

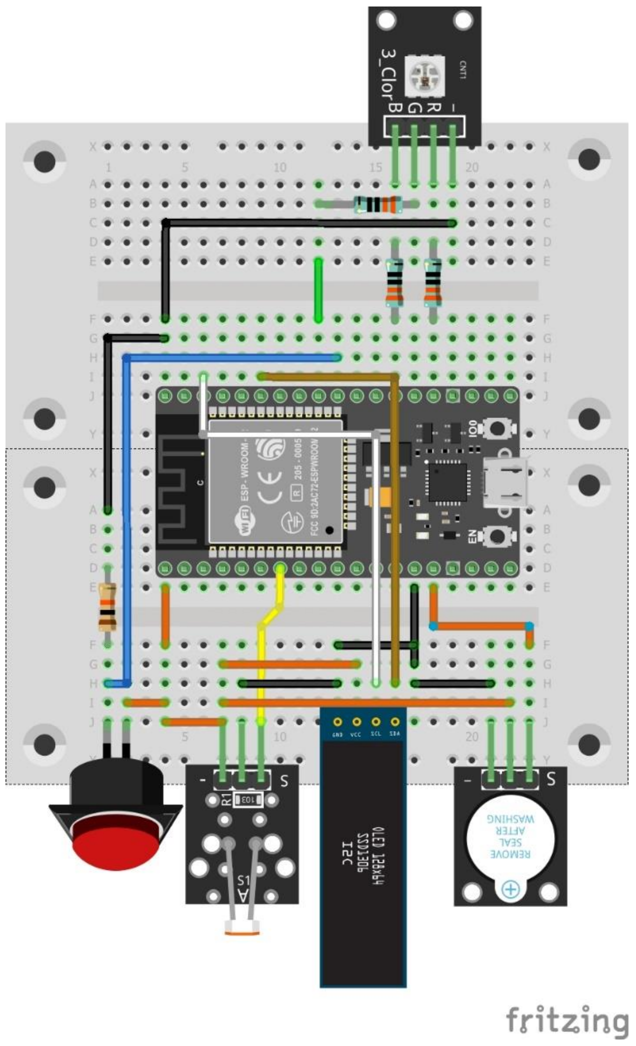
Replace the structure of the html variable in the webPage () function with the following text. Please pay attention to the correct indentations. You can use the string of the print instruction from the server loop directly.

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

  html = html + "" "<br> <a href=http://8.8.8.8:9192/ein> switch on the LED </a> <br>
  <a href=http://8.8.8.8:9192/aus> Switch off the LED </a> <br>
  "" "html = html + act + '. </body> </html>'
```

• Add a second LED to the structure that flashes when a request is received on the server.

For this purpose I took an RGB LED and connected it to pins GPIO2 (red), 4 (blue) and 18 (green). server2.py was added as follows.



```
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 of the server loop
…
  c.close()
  connPin.off()
```

Now every time a request is received by the ESP32 server, the blue LED flashes briefly.