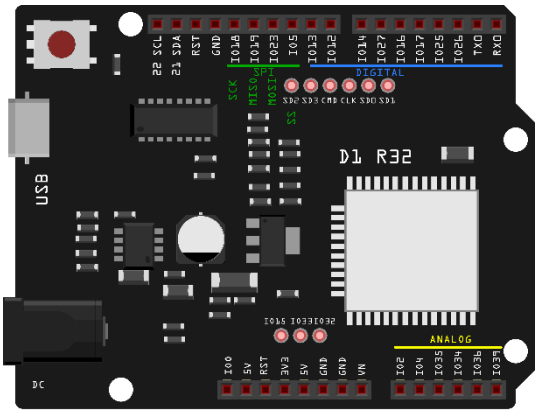
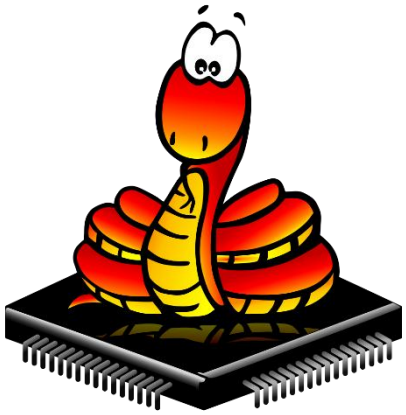
ESP32 mit MicroPython



+



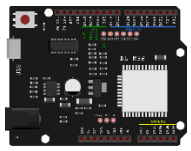
- Unterrichtsmaterial für den NwT Unterricht in Klasse 9 –

Microcontroller als Steuereinheit

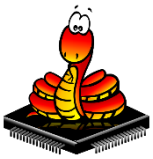
Microcontroller sind in vielen **elektronischen Geräten** enthalten. Sie **steuern Abläufe**. Du findest sie zum Beispiel in **Waschmaschinen** zur Programmsteuerung, in **Saugrobotern** zur Navigation, in **Airbagsystemen** zur Auslösung im richtigen Moment oder in **automatischen Türen**, damit diese sich rechzeitig öffnen.
Es gibt ganz **verschiedene Microcontroller** und man kann sie mit **unterschiedlichen Programmiersprachen** steuern. Dieses Material führt dich an die Programmierung des **ESP32-Controllers** mit der Sprache **MicroPython** heran. Es sind keine Grundkenntnisse notwendig und die verschiedenen enthaltenen Arbeitsblätter bauen aufeinander auf.

➔ Dieses Material ist ein Projekt in Entwicklung – Rückmeldungen sind erwünscht!

04.2026 (Sch)



ESP32 mit MicroPython



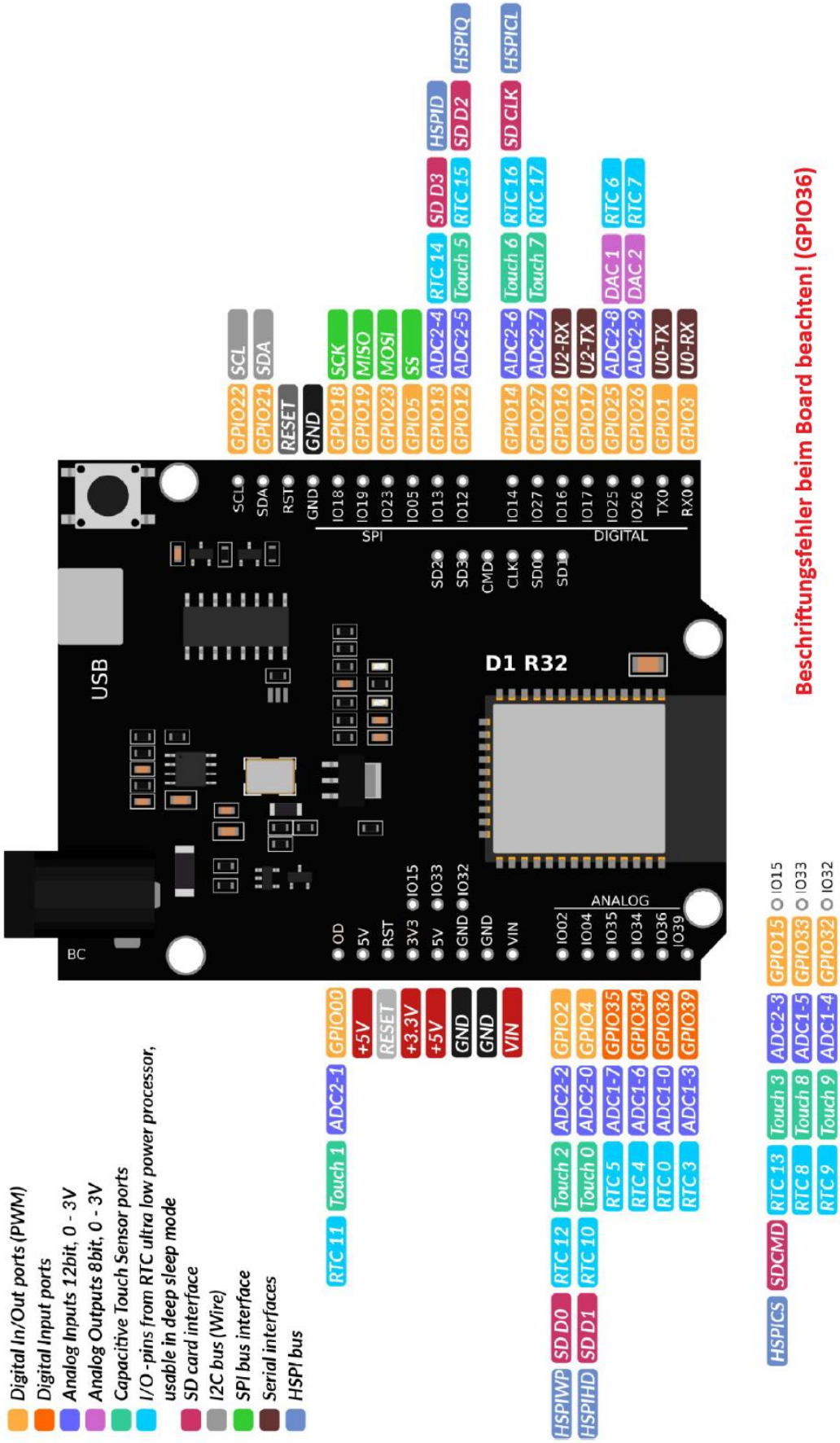
Inhaltsverzeichnis:

AB-01: Gleichstrom, Wechselstrom & Gleichstromkreise
AB-02: Aufbau ESP32
AB-03: Stromkreis mit LED und Vorwiderstand
AB-04: Die Programmierumgebung „Thonny“
AB-05: LED blinken lassen
AB-06: Elektrische Widerstände
AB-07: Programmierübungen mit LEDs
AB-08: Programmablaufpläne
AB-09: Unterprogramme
AB-10: Textausgabe
AB-11: Tonausgabe
AB-12: Variablen und Datentypen
AB-13: Tasten und Entprellung
AB-14: Schleifen
AB-15: Zufallsgenerator
AB-16: Nicht-blockierendes Warten
AB-17: TM1637-Display
AB-18: Der Reaktionstester
AB-19: Das Reaktionsspiel
AB-20: Das Reaktionsspiel für zwei Personen
AB-21: OLED-Display (SH1106)
AB-22: LCD-Display (1602)
AB-23: Potentiometer
AB-24: Lichtabhängige Widerstände (LDR)
AB-25: Farbdisplay (ST7735)
AB-26: ESP-NOW Funkübertragung (WLAN)

04.2026 (Sch)

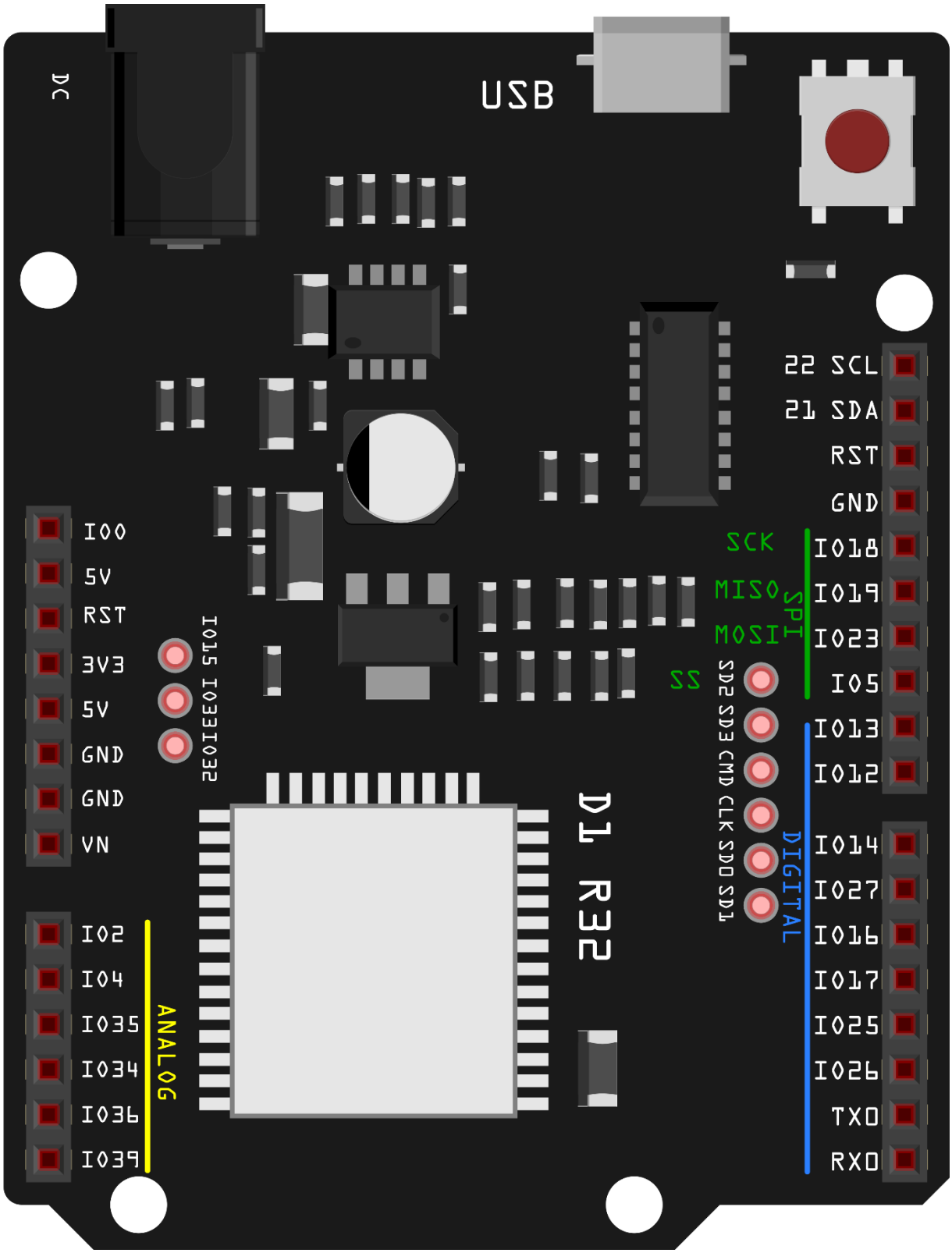
AB-27:	**Projekt:**	**Taschenrechner**
AB-28:	**Projekt:**	**RGB-Mixer**
AB-29:	**Projekt:**	**Kombination von Listen**
AB-30:	**Projekt:**	**Form-Demo mit Animation**
AB-31:	**Projekt:**	**NeoPixel LED-Ring**
AB-32:	**Projekt:**	**Max7219 8x8-Pixel Matrix**
AB-33:	**Projekt:**	**Temperatur- und Feuchtigkeitssensor (DHT11)**
AB-34:	**Projekt:**	**Echtzeituhr (DS3231)**
AB-35:	**Projekt:**	**Entfernungsmessung mit Licht (VL5310X)**
AB-36:	**Projekt:**	**Laufschriftanzeige (Max7219)**

ESP32 Pinout



04.2026 (Sch)

ESP32 Pinout



Microcontroller-Programmierung mit MicroPython (ESP32)

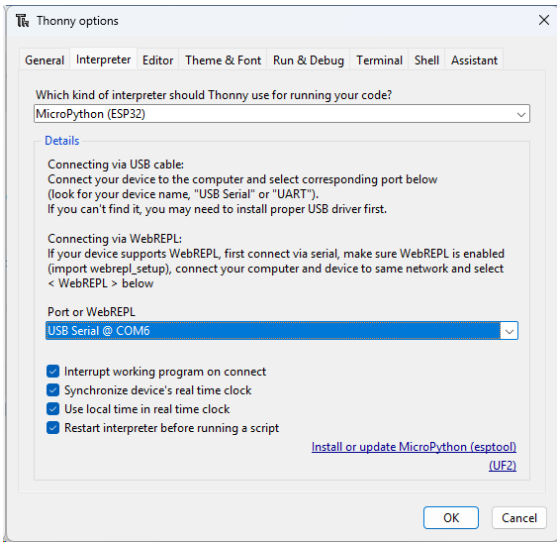
Installation von MicroPython auf dem ESP (Thonny)

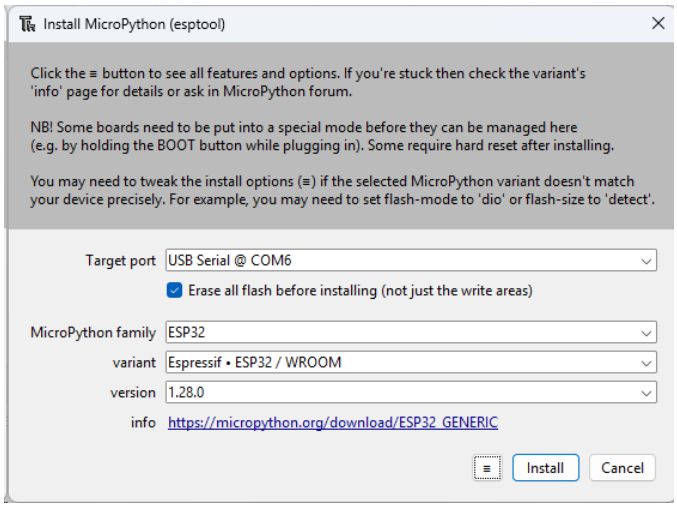
Thonny und der ESP32:

Um MicroPython auf dem ESP32 nutzen zu können, muss zunächst der Python-Interpreter auf dem ESP32-Board installiert werden. Dafür ist es notwendig genau zu wissen, welches ESP32 Board man nutzen möchte (es gibt viele Varianten!).

Einzelschritte:

- Thonny installieren (https://thonny.org)
- USB-Bridge-Treiber installieren (CH341/340 oder CP210x)
- ESP32-Controller per USB anschließen
- Menüauswahl: Run → Configure Interpreter
- Kind of Interpreter: MicroPython (ESP32) auswählen
- Port auswählen
- Install or update MicroPython (esptool)
- Port, Family, Variant und Version auswählen (je nach Chip)
- Installieren





Achtung USB-Bridge Treiber!

Auf unterschiedlichen ESP-Boards kommen unterschiedliche **USB-Bridge Chips** zum Einsatz. Daher werden **unterschiedliche Treiber** benötigt. Häufig ist es entweder der **CH341/340** und **CP210x**. Beide Treiber können in Thonny Probleme erzeugen.

-Beim **CH341/340** ist bei **Windows11** teilweise nur die **Version** von **2014** nutzbar!

-Beim **CP210x** funktioniert das **STOP-Kommando** nicht, man muss **STRG-C** nutzen!

04.2026 (Sch)

Info:

AC oder DC – was ist der entscheidende Unterschied?

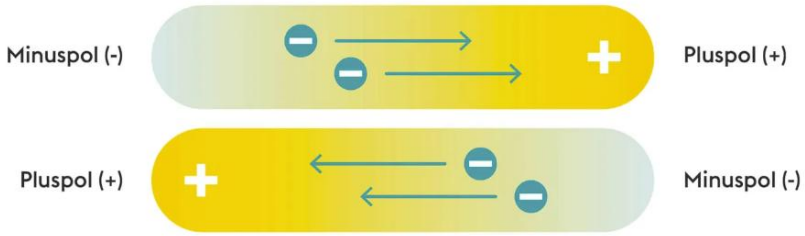
Wer schon einmal einen Blick auf die Rückseite seines Laptop-Netzteils geworfen hat, ist dabei vielleicht über die Abkürzung **AC** gestolpert. AC steht für **„alternating current“**. Auf Deutsch bedeutet das **„Wechselstrom“**. Wechselstrom ist eine Art von elektrischem Strom, bei dem die Elektronen ständig **ihre Richtung ändern**.

Das Gegenteil von AC ist **DC**, der sogenannte **„direct current“** beziehungsweise **Gleichstrom**. Hier fließen die kleinen Ladungsträger die ganze Zeit über konstant in **dieselbe Richtung**.

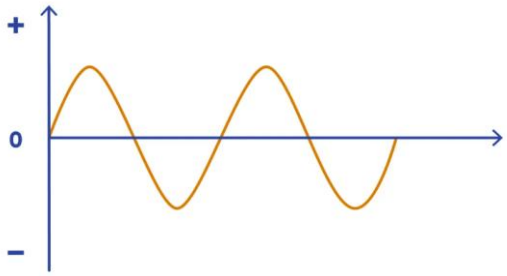
Elektronen sind winzige negativ geladene elektrische Teilchen, die für die Erzeugung von elektrischem Strom verantwortlich sind. Sie erzeugen Elektrizität beziehungsweise leiten den elektrischen Strom durch Bewegung.

Was ist Wechselstrom genau?

Wechselstrom (AC) **ändert seine Richtung periodisch**, also in festen zeitlichen Abständen. Dabei wechselt die Stromstärke immer wieder vom Plus- zum Minuspol. Optisch kann man sich das etwa so vorstellen:



Die freien Elektronen bewegen sich konstant zwischen Plus- und Minuspol hin und her. Im zeitlichen Verlauf entsteht durch das Hin- und Herwechseln zwischen dem Plus- und dem Minuspol eine Wellenbewegung oder **Sinuskurve**:



Wie häufig die Stromrichtung pro Sekunde wechselt, wird dabei mit der Frequenz in der Einheit Hertz (Hz) angegeben. **Ein Beispiel:** Die Stromnetze in Europa werden mit **50 Hz** betrieben. **Das bedeutet, die Richtung der Elektronen ändert sich 50-mal pro Sekunde**.

04.2026 (Sch)

Wie entsteht Wechselstrom?

Wechselstrom entsteht, wenn eine Spannungsquelle, zum Beispiel ein Generator, ein **Magnetfeld erzeugt und eine Spule sich darin dreht**. Die Bewegung der Spule erzeugt dann elektrischen Strom, der seine Richtung ständig ändert – also Wechselstrom.

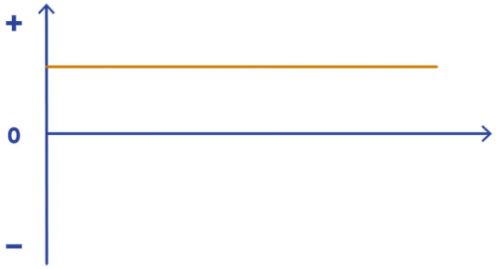
Ein Beispiel:

Ein Fahrraddynamo besteht hauptsächlich aus einer Kupferdrahtspule und einem Magneten, der sich durch die Befestigung am Rad ständig im Kreis dreht. Der rotierende Magnet mit seinem Plus- und Minuspol sorgt dafür, dass die Elektronen in der Spule durch das veränderte Magnetfeld ständig ihre Richtung ändern. Die Elektronen im Kupferdraht pendeln hin und her. Dadurch entsteht Wechselstrom, und Licht wird erzeugt.

Was genau ist Gleichstrom?

Beim Gleichstrom (DC) bewegen sich die Elektronen innerhalb eines Stromleiters **immer in dieselbe Richtung**, nämlich vom Minuspol zum Pluspol.

Die freien Elektronen bewegen sich ausschließlich vom Minus- zum Pluspol.
Den zeitlichen Verlauf des Gleichstroms kann man sich folglich wie eine **gerade Linie** vorstellen – keine Kurve wie beim Wechselstrom:



Wie entsteht Gleichstrom?

Gleichstrom lässt sich beispielsweise mit Batterien erzeugen. Verbindet man die Gleichstromquelle mit einem Stromkreis, werden die freien Elektronen im elektrischen Leiter vom Pluspol angezogen. Dies führt zu einer konstanten Bewegung der Ladung – das Resultat ist Gleichstrom.

Gleich- und Wechselstrom am jeweiligen Zeichen erkennen

Nicht immer werden die unterschiedlichen Stromarten durch ihre Abkürzungen AC oder DC, beispielsweise auf Elektrogeräten, gekennzeichnet. Oft finden sich auch nur die entsprechenden Zeichen für Gleich- und Wechselstrom:

Dieses Zeichen steht für **Wechselstrom** und erinnert an die Sinuskurve, die entsteht, wenn die Elektronen zwischen den Polen hin- und herwandern.

Alles geradlinig: Beim **Gleichstrom** fließen die Elektronen immer in dieselbe Richtung – dafür steht dieses Symbol.

Welcher Strom kommt aus der Steckdose – AC oder DC?

Aus einer haushaltsüblichen Steckdose in Europa kommt **Wechselstrom mit einer Spannung von 230 Volt**. Viele Elektrogeräte werden mit Wechselstrom betrieben (siehe weiter unten). Für den **Herd mit Backofen**, die **Wärmepumpe** zum Heizen des Hauses oder das Laden des **E-Autos** (siehe unten) darf es allerdings gerne etwas mehr sein: Diese Geräte und Vorgänge benötigen eine Wechselspannung von **400 Volt** und werden deshalb an den **Dreiphasenwechselstrom** (umgangssprachlich **Starkstrom**) angeschlossen, der über das Niederspannungsnetz – also das lokale Verteilnetz – zu den Wohngebäuden transportiert wird.

Wie oben beschrieben, steckt Gleichstrom vor allem in Batterien und somit in allen batteriebetriebenen Geräten. Aber auch viele moderne elektrische Geräte wie Laptops benötigen Gleichstrom. Damit sie trotz des Wechselstroms aus der Steckdose verwendet werden können, kommt ein **Netzteil** zum Einsatz, das den **Wechselstrom in Gleichstrom umwandelt**. Manche Geräte haben auch einen integrierten Akku, der diese Aufgabe übernimmt.

Folgende Geräte nutzen typischerweise Gleich- beziehungsweise Wechselstrom:

Gleichstrom

- Batteriebetriebene Haushaltsgeräte, z. B. Fernbedienung oder Taschenlampe
- Unterhaltungs- und Kommunikationsgeräte wie Smart-TV, Handy, Laptop

Wechselstrom

- Elektro-Großgeräte, z. B. Kühlschrank, Waschmaschine, Backofen
- Küchengeräte wie Küchenmaschine, Waffeleisen, Kaffeemaschine
- Föhn

Quelle: https://www.enercity.de/magazin/mein-leben/fakten-wechselstrom-gleichstrom

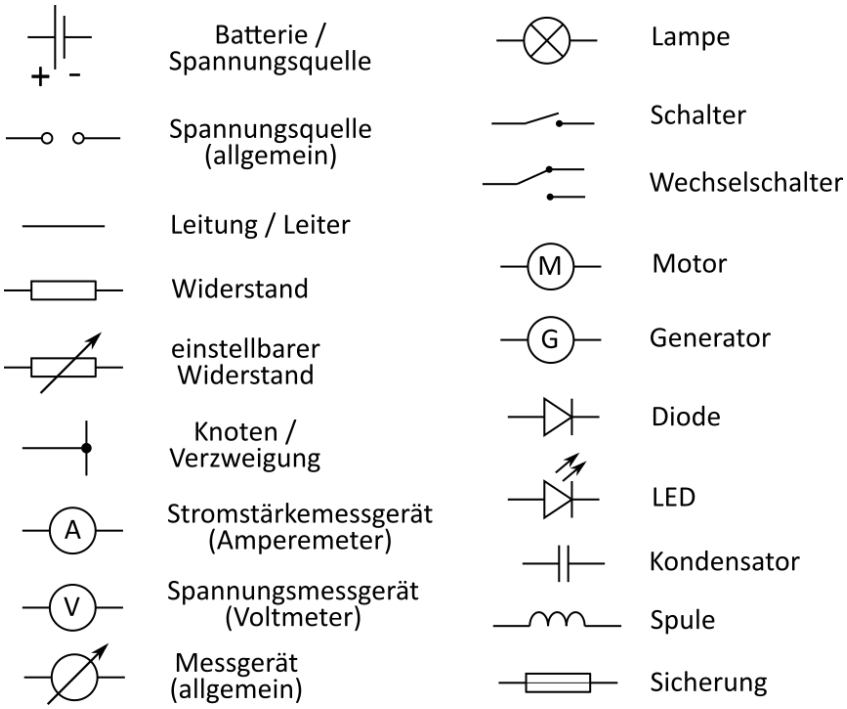
04.2026 (Sch)

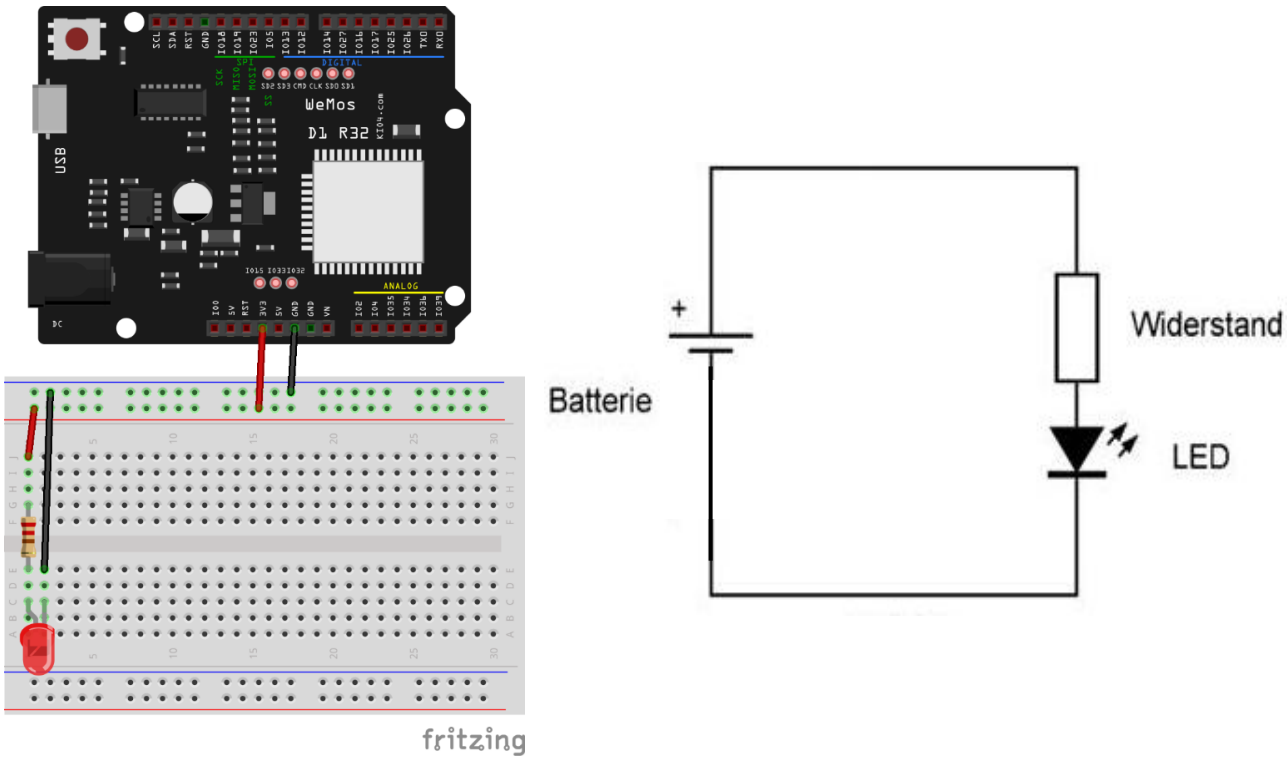
Microcontroller-Programmierung mit MicroPython (ESP32)

AB-01: Gleichstrom, Wechselstrom & Gleichstromkreise

Info:

Bei der Erstellung von **Gleichstromkreisen** gibt es eine Liste von Symbolen, die zu verwenden sind. Dazu sollte beachtet werden, dass nur senkrechte oder waagerechte Linien gezeichnet werden.





Quelle: https://mondbrand.de/gleichstromkreise.htm

04.2026 (Sch)

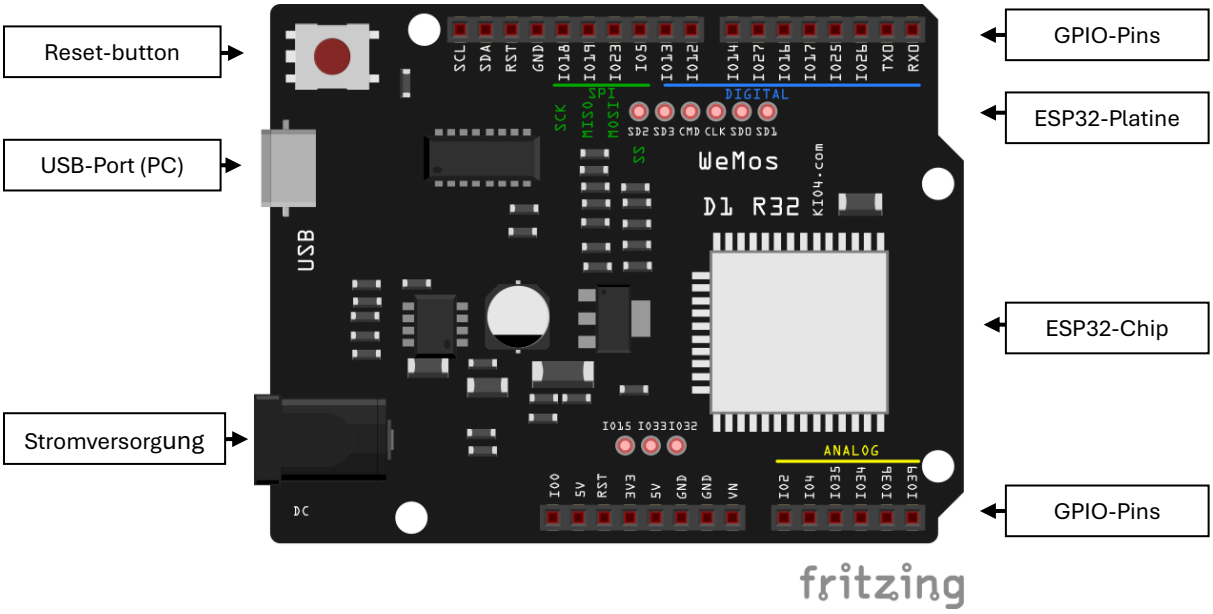
Microcontroller-Programmierung mit MicroPython (ESP32)

AB-02: Aufbau ESP32

Info:

Der **ESP32** ist eine Platine mit einem kleinen **Mikrocontroller**, die sich mittels eines USB-Kabels an den Computer anschließen lässt. Mit dort **geschriebenen Programmen** lassen sich über den Mikrocontroller **Bauteile steuern** und **Sensoren auslesen**. Dadurch lassen sich kleine oder auch etwas größere Elektronikprojekte realisieren. Ein **Programm** besteht aus einer **Reihe von Anweisungen**, die nacheinander abgearbeitet werden.

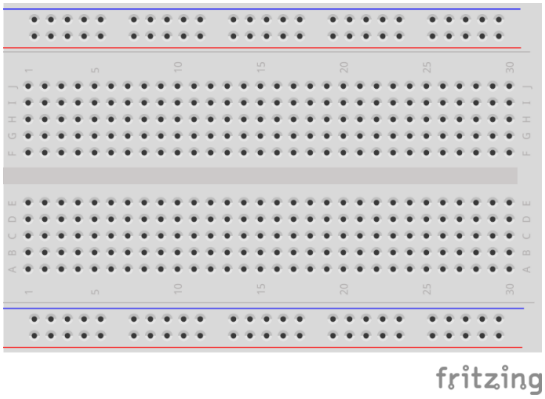
Aufbau:

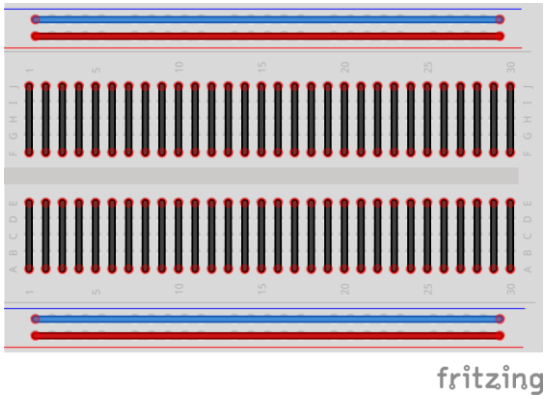


Der Arduino verfügt über eine **Platine**, auf der oben **digitale Anschlüsse** und unten **analoge Anschlüsse** (**GPIO**s) festgelötet sind. Außerdem gibt es einen **Reset-Knopf**, **USB-Anschluss** und einen **Batterieanschluss**. Das größte sichtbare Bauteil ist der eigentliche **Mikrocontroller**, der die Datenverarbeitung übernimmt.

Breadboard:

Ein Breadboard (Steckbrett) ist ein Bauteil um eigene Schaltungen zu stecken ohne sie zu löten. Auf dem Breadboard sind immer mehrere ports miteinander verbunden:





04.2026 (Sch)

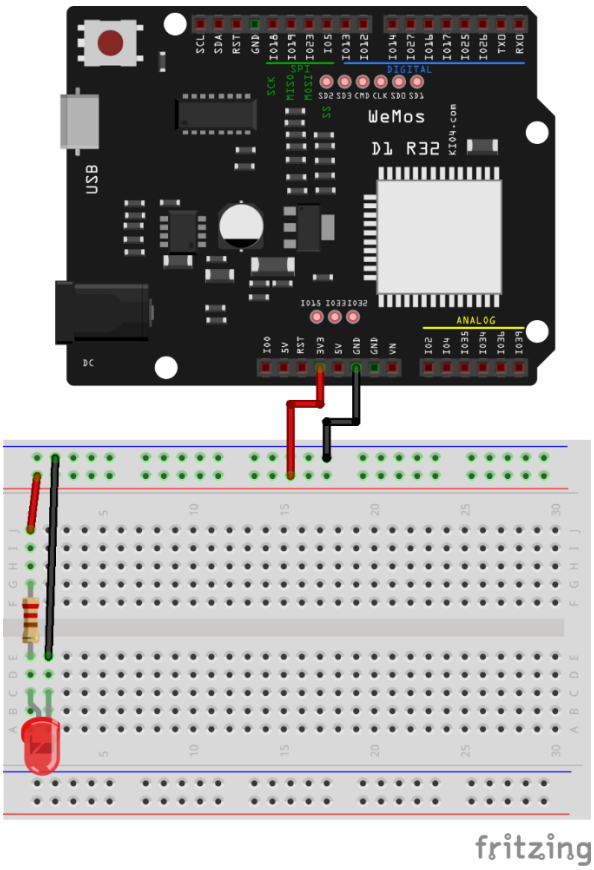
Microcontroller-Programmierung mit MicroPython (ESP32)

AB-03: Aufbau ESP32

Komponenten:

- ESP32 Board, Breadboard, Verbindungskabel (x4), LED (1x), Widerstand (1x 220 Ω)

Aufbau:



Hier wird der ESP32 als **Spannungsquelle** für eine LED verwendet. Der **Vorwiderstand (220 Ω)** muss sich am Pluspol (+) oder am Minuspol (-) der LED im Stromkreis befinden.

Der (-) Pol entspricht der Erdung (GND) des ESP und alle GND-Pins sind verbunden!

➔ **Ohne Vorwiderstand nimmt die LED in der Regel sofort Schaden!**

Achtung! Bei einem geschlossenen Stromkreis, ohne dass ein Widerstand eingebaut ist entsteht ein Kurzschluss. Dies muss unbedingt verhindert werden!

Aufgabe:

4.1) Zeichne in das Bild den geschlossenen Stromkreis ein!

Achtung, sehr wichtig!

Der ESP32 liefert **3,3 V und 5 V an zwei unterschiedlichen Ports**. Wir verwenden für die Aufgaben im Unterricht **ausschließlich 3,3 V**. Schließe **niemals** ein Kabel an den **5 V** Port des ESP32 an, außer du wirst dazu aufgefordert. **Der ESP32 kann Schaden nehmen!**

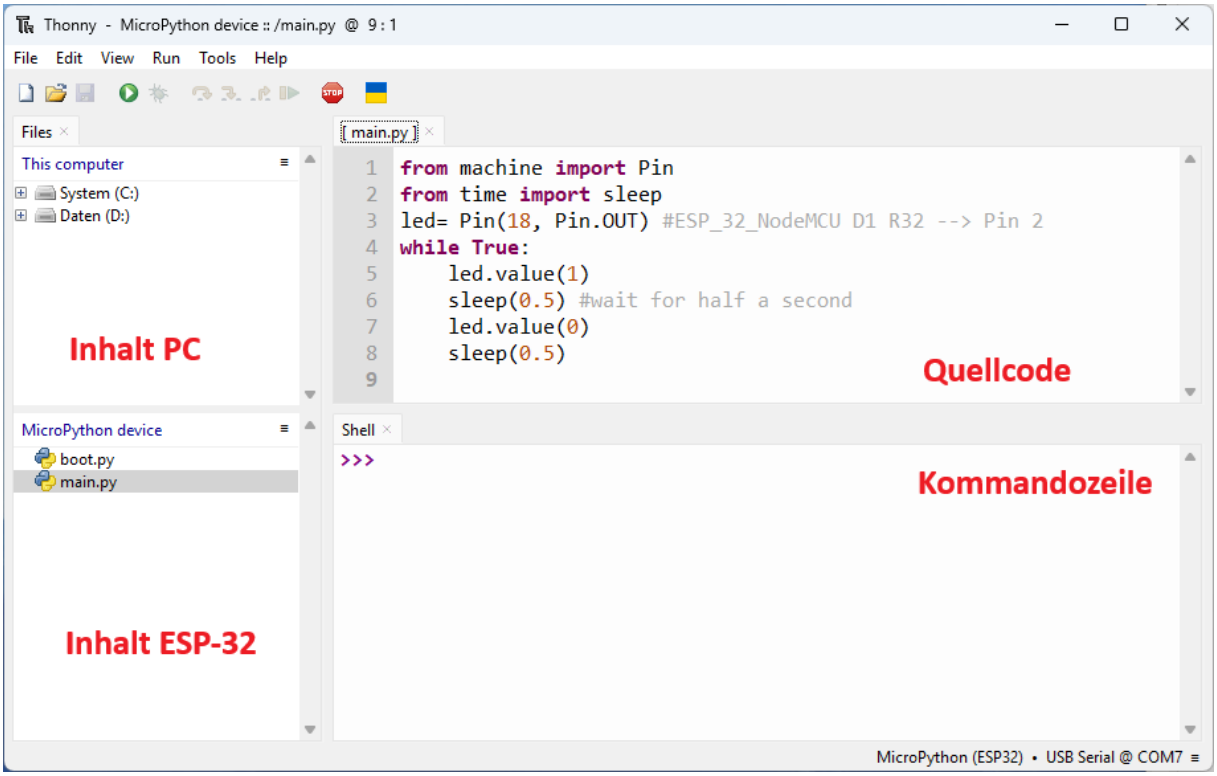
04.2026 (Sch)

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-04: Die Programmierumgebung „Thonny“

Info:

Für die Programmierung eines ESP32-Microcontrollers verwenden wir die freie IDE („Integrated Developer Environment“) **Thonny**. (https://thonny.org)



Hier siehst du auf einen Blick alles, was du zum Programmieren brauchst:

- Links oben: Hier kannst du auf Dateien auf deinem PC zugreifen
- Links unten: Hier siehst du alle auf dem ESP-32 gespeicherten Dateien
- Rechts oben: Hier siehst du den aktuell geöffneten Quellcode
- Rechts unten: Hier kannst du dem ESP32 Befehle geben und Meldungen lesen

Der **ESP32** wird in unserem Fall mit der **Programmiersprache Python** programmiert. Dies ist eine **interpretierte** Programmiersprache. Das bedeutet, dass der ESP32 Befehle **in Echtzeit verarbeiten** (interpretieren) kann, die man ihm entweder als Datei oder als Kommando über die Kommandozeile erteilt. Dies steht im Gegensatz zu **anderen Programmiersprachen**, wo Quellcode erst **kompiliert** (in Maschinensprache übersetzt) werden muss.

Aufgabe:

2.1) Gib dem ESP32 ein paar einfache Rechenaufgaben in der Kommandozeile.

04.2026 (Sch)

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-04: Die Programmierumgebung „Thonny“

Programmaufbau:

Ein **Python-Skript** ist sehr einfach aufgebaut. Eigentlich handelt es sich nur um **eine Liste von Befehlen**, die nacheinander vom Python-Interpreter, der auf dem ESP32 installiert ist, abgearbeitet werden.

Um ohne PC ausgeführt werden zu können, muss das Skript **main.py** heißen und **auf dem ESP32 gespeichert** werden. Der ESP32 kann aber auch andere Python-Skripte ausführen, die in Thonny geöffnet sind und auch anders heißen. Dafür muss der ESP32 aber dauerhaft mit einem PC verbunden sein.

Ein einfaches Skript (Quellcode) kann zum Beispiel so aussehen:

[main.py]

```
1  from machine import Pin
2  from time import sleep
3  led= Pin(18, Pin.OUT) #ESP_32_NodeMCU D1 R32 --> Pin 2
4  while True:
5      led.value(1)
6      sleep(0.5) #wait for half a second
7      led.value(0)
8      sleep(0.5)
9
```

- In den Zeilen 1 & 2 werden **Module importiert** (Bibliotheken) um bestimmte (zusätzliche) Befehle verwenden zu können. Es handelt sich dabei um .py-Dateien.
- In Zeile 3 wird ein **Pin** (18) als Ausgangspin **definiert** und mit einem **eindeutigen Bezeichner** (led) versehen, der im folgenden Code verwendet werden kann
- In Zeile 4 wird eine **while-Schleife** gestartet, deren Inhalt dauerhaft läuft, solange der ESP32 mit einer Stromquelle (Netzteil, Batterie oder PC) verbunden ist
- Die Zeilen 5-8 sind **eingerückt**, weil hier der **Inhalt der Schleife** (Block) steht

→ Ohne die Schleife würde der ESP32 das Skript nur ein einziges Mal abarbeiten!

Ausführung/Anhalten eines Skripts:

- Ein Skript wird über die Schaltfläche **PLAY** gestartet.
- Mit der Schaltfläche **STOP** wird die Ausführung des Skripts beendet. Dies ist nötig bevor ein verändertes Skript ausgeführt oder auf dem ESP32 gespeichert werden kann.

Wichtig:

- Die **Groß- und Kleinschreibung** sind bei Python sehr wichtig
- Die **Einrückungen** (siehe Zeile 5ff) sind bei Python ebenfalls sehr wichtig
- Mit einem **#** werden **Kommentare** eingeleitet, die der Interpreter nicht beachtet

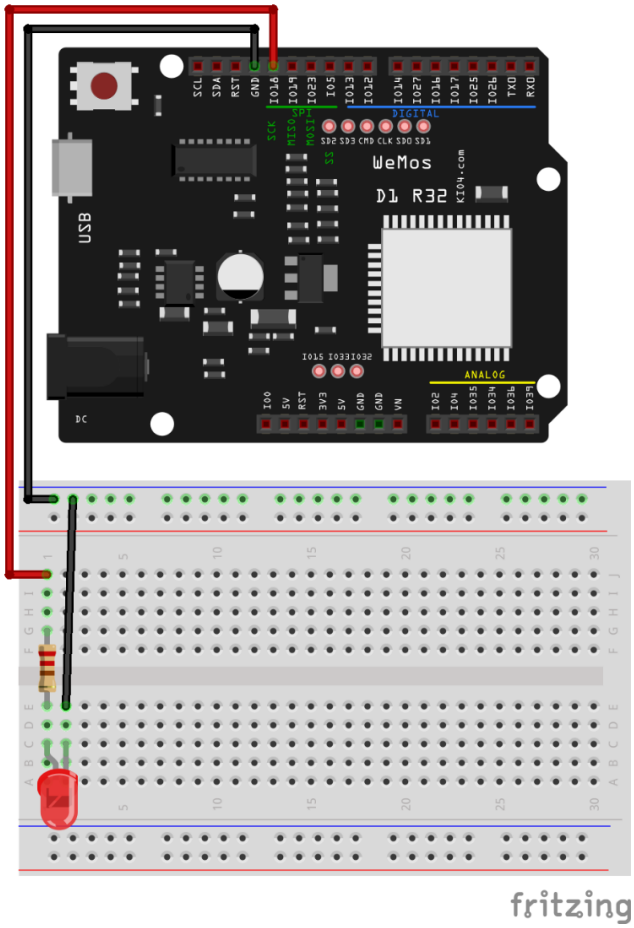
Microcontroller-Programmierung mit MicroPython (ESP32)

AB-06: Elektrische Widerstände

Komponenten:

- ESP32 Board, Breadboard, Verbindungskabel (x3), LED (1x), Widerstand (1x 220 Ω)

Aufbau:



Befehle:

led ist variabel und entspricht der vorher **zugewiesenen Bezeichnung** eines Pins (Zeile 3).

value hat die Zustände **1** oder **0**. Das entspricht **Strom** oder **kein Strom**.

sleep pausiert die komplette Verarbeitung des ESP32 für eine gewisse Zeit. Die Einheit hierbei sind **Sekunden**. **1** steht dabei für **eine Sekunde** und **0.5** für **eine halbe Sekunde**.

sleep muss aus **time** importiert werden:

Achtung!

a) `2 from time import sleep`

Aufruf über: `6 sleep(0.5)`

→ einzelner Befehl aus time wird importiert

→ bequemer, da kürzerer Aufruf

b) `2 import time`

Aufruf: `6 time.sleep(0.5)`

→ ganzes Modul wird importiert

→ Herkunft des Befehls direkt nachvollziehbar

Bei gleichen Namen kann es Konflikte geben!

Code:

```
1 from machine import Pin
2 from time import sleep
3 led= Pin(18, Pin.OUT) #onboard LED --> Pin 2
4 while True:
5     led.value(1) #turn power on for pin 18
6     sleep(0.5)   #wait for half a second
7     led.value(0) #turn power off for pin 18
8     sleep(0.5)   #wait for half a second
```

Aufgaben:

5.1) Baue die Schaltung auf und schreibe den Code ab.

5.2) Verändere den Code, sodass die LED schneller bzw. langsamer blinkt.

5.3) Verringere die sleep-Zeit, bis du das Blinken nicht mehr erkennen kannst.

5.4) Verändere den Code unter Verwendung von „not“. `5 led.value(not led.value())`

5.5) Verändere den Code unter Verwendung von „toggle“. `5 led.toggle()`

5.6) Passe den Code aus **5.5)** mit `1 import machine` und `2 import time` an.

04.2026 (Sch)

Was machen elektrische Widerstände?

Ein elektrischer Widerstand (engl: „resistor“) ist ein Bauelement, das den **Fluss des elektrischen Stroms** in einem Stromkreis **begrenzt oder reguliert**. Er wandelt elektrische Energie in Wärme um, wodurch die **Stromstärke verringert** und die **Spannung reduziert** wird. Dies schützt Komponenten (z. B. LEDs) vor zu hohen Strömen.

Wenn du dir Strom wie Wasser in einem Rohr vorstellst, dann wirkt ein Widerstand wie eine Engstelle im Rohr. Je enger die Stelle, desto schwieriger kommt das Wasser durch. Genauso wird auch der elektrische Strom reduziert.

Farbcodes von Widerständen:

(Braun, Schwarz, Rot, Gold)

4 Ringe

1.000 Ω ± 5 %

Farbe	1. Ring	2. Ring	3. Ring	Multiplikator	Toleranz
Schwarz	0	0	0	× 1 Ω	
Braun	1	1	1	× 10 Ω	± 1 %
Rot	2	2	2	× 100 Ω	± 2 %
Orange	3	3	3	× 1.000 Ω (1 kΩ)	
Gelb	4	4	4	× 10.000 Ω (10 kΩ)	
Grün	5	5	5	× 100.000 Ω (100 kΩ)	± 0,5 %
Blau	6	6	6	× 1.000.000 Ω (1 MΩ)	± 0,25 %
Lila	7	7	7	× 10.000.000 Ω (10 MΩ)	± 0,1 %
Grau	8	8	8		± 0,05 %
Weiß	9	9	9		
Gold				× 0,1 Ω	± 5 %
Silber				× 0,01 Ω	± 10 %

5 Ringe

2.700 Ω ± 1 %

(Rot, Lila, Schwarz, Braun, Braun)

Stromkreis und Widerstand

Die Vorgänge in einem Stromkreis sind manchmal schwer zu verstehen, da man Strom nicht sehen kann. Als Modell wird häufig ein Wasserkreislauf verwendet, auf den man die Bauteile und Vorgänge überträgt. Stelle in der Tabelle zu den Begriffen aus dem Stromkreis wie Stromstärke und Spannung sowie Bauteile eines Stromkreises (Batterie, Lampen, Widerstände) den passenden Bauteilen eines Wasserkreislaufs gegenüber.

Stromkreis	**Wassermodell**
Batterie	
Kabel	
Lampe	
Widerstand	
Spannung	
Stromstärke	

➔ Die Pins unseres ESP32 liefern eine Spannung von 3,3V – 5V. Die LEDs können jedoch nur mit Spannungen von 2 bis 3 V arbeiten, alles darüber kann sie zerstören. Daher werden Schutzwiderstände benötigt.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-07: Programmierübungen LED

Aufgaben:

7.1) Windkraftanlagen müssen ein rotes Blinklicht an der Spitze haben um Flugzeuge zu warnen. Das Blinklicht hat einen besonderen Rhythmus: **1 Sekunde an, dann 0,5 Sekunden aus, dann wieder eine Sekunde an, dann aber 1,5 Sekunden aus.** Programmiere ein solches Blinklicht.

7.2) Wie lange leuchtet in diesem Programm die LED jeweils und wie lange ist sie aus?

```
from machine import Pin
from time import sleep
led= Pin(2, Pin.OUT) #onboard LED --> Pin 2
while True:
    led.value(1)
    sleep(0.4)
    led.value(0)
    sleep(0.2)
    led.value(1)
    sleep(0.3)
```

7.3) Bei einem dieser Programme blinkt die LED nur einmal, beim anderen leuchtet sie nur dauerhaft. Welches ist welches und warum?

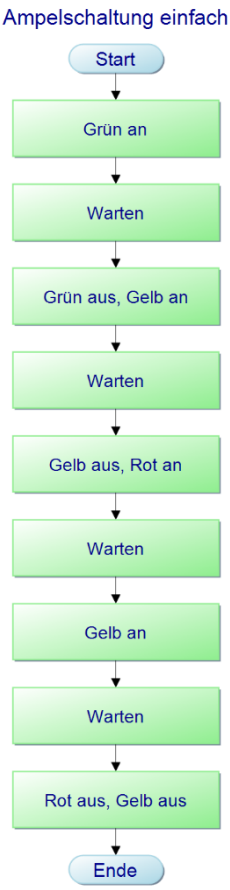
a)

```
from machine import Pin
from time import sleep
led= Pin(18, Pin.OUT)
while True:
    led.value(1)
    sleep(0.5)
    led.value(0)
```

b)

```
from machine import Pin
from time import sleep
led= Pin(18, Pin.OUT)
led.value(1)
sleep(0.5)
led.value(0)
while True:
    sleep(1)
```

7.4) Realisiere mit einer roten, gelben & grünen LED eine Ampelschaltung.



Microcontroller-Programmierung mit MicroPython (ESP32)

AB-08: Programmablaufpläne

Info:

Programmablaufpläne sind **visualisierte Darstellungen** für die Abläufe von Programmen. Die **Handlungsanweisungen** in einem Programm (**Algorithmen**) werden mit Hilfe von standardisierten **Symbolen** darstellt. Algorithmen sind mit Kochrezepten vergleichbar.

Symbole:

➔ Die Abläufe/Zusammenhänge werden mit Hilfe von Pfeilen dargestellt:

Vorarbeit:

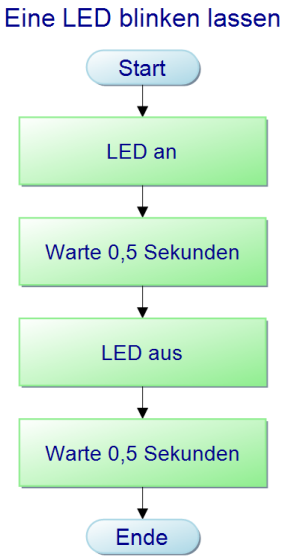
Bevor ein Programmablaufplan (PAP) erstellt wird, erfolgt die verbale Beschreibung der Abläufe eines Programms:

a) Eine LED soll blinken. Dafür soll sie jeweils eine halbe Sekunde an und eine halbe Sekunde aus sein.

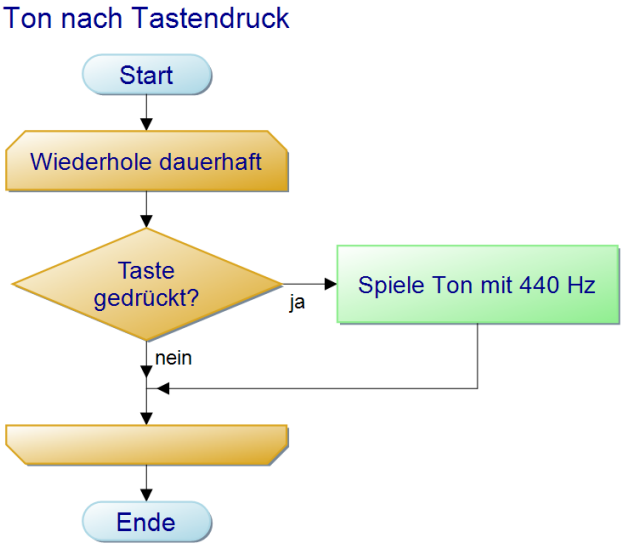
b) Wenn eine Taste gedrückt wird, soll ein Ton gespielt werden (440 Hz). Sobald die Taste nicht mehr gedrückt wird, soll der Ton ausgehen.

Beispiele: (PapDesigner)

a)



b)



ablone lass

Wichtig: Ein Programmablaufplan zeigt <u>was</u> passiert, <u>nicht wie</u> es konkret realisiert wird!

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-08: Programmablaufpläne

Übungsaufgabe 1:

Wir wollen einen Reaktionstester bauen. Folgender Ablauf soll dabei abgebildet werden:

Mit einer Verzögerung von 2-5 Sekunden nach dem Start des Programms soll eine LED aufleuchten. Es soll dann innerhalb von 10 Sekunden eine Taste gedrückt werden. Die LED soll mit dem Drücken des Schalters ausgehen und die gemessene Zeit vom Aufleuchten der LED bis zum Drücken des Schalters angezeigt werden. Falls nach 10 Sekunden kein Tastendruck erfolgt ist, soll dies ebenfalls rückgemeldet werden.

Schreibe einen Programmablaufplan!

Übungsaufgabe 2:

Wir wollen einen Notenwürfel bauen. Folgender Ablauf soll dabei abgebildet werden:

Der Benutzer soll eine Taste zu drücken. Nach dem Drücken der Taste soll eine zufällige Schulnote gewürfelt werden. Wenn diese Note zwischen 1 und 4 liegt, dann soll die Meldung „Bestanden“ ausgegeben werden. Bei einer schlechteren Note soll eine rote Lampe aufleuchten und die Rückmeldung „Nicht bestanden“ ausgegeben werden.

Schreibe einen Programmablaufplan!

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-09: Unterprogramme

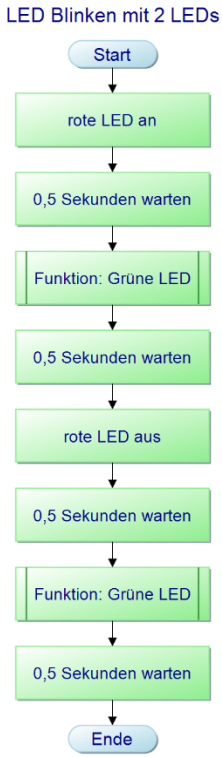
Info:

Wenn in einem Programm häufig eine Reihe von gleichen Anweisungen benötigt wird, ist es hilfreich diese Anweisungen in ein Unterprogramm (function) auszulagern. Diese Praxis führt zu übersichtlicherem Code und spätere Veränderungen lassen sich leichter umsetzen. Das Unterprogramm kann nach Bedarf aufgerufen werden.

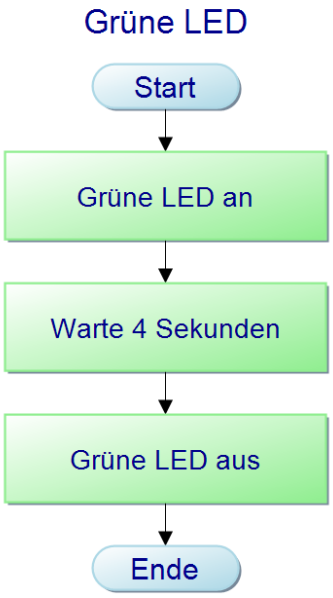
Symbol: 

Beispiel:

Hauptprogramm:



Unterprogramm (Grüne LED):



In Python wird ein Unterprogramm folgendermaßen **definiert**:

```
6  def green(): #function is defined
7      led_green.value(1)
8      sleep(4)
9      led_green.value(0)
```

Der **Aufruf** geschieht dann einfach über:

```
14     green() #function is called
```

➔ **Wichtig: Das Unterprogramm muss im Skript vor dem Aufruf definiert werden!**

Aufgabe:

9.1: Baue in deine Ampelschaltung aus 7.4) Unterprogramme ein, die den verschiedenen Ampelphasen entsprechen. (*Tipp: Es gibt 4 verschiedene Ampelphasen.*)

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-10: Textausgabe

Info:

Um in der Kommandozeile einen Text ausgeben zu können braucht es den Befehl **print**.

Beispiele:

a)
```
1 print("Hallo Welt!")
```

➔ Hier wird in der Kommandozeile rechts unten „Hallo Welt“ ausgegeben.

b)
```
1 print("Hallo Welt!")
2 print("Ich bin ein einfaches Python-Skript.")
```

➔ Hier werden zwei Textzeilen ausgegeben.

➔ Merke: Python fängt bei jedem **print** Befehl eine neue Zeile an (Zeilenumbruch).

c)
```
1 import time
2 print("Aktuelles Datum und die Zeit:")
3 print(time.localtime())
```

➔ Hier wird die aktuelle Systemzeit ausgegeben (unformatiert).

d)
```
1  namen = ("Max","Peter","Monika","Jana")
2
3  def sage_hallo(name):
4      print("Hallo "+name)
5
6  for name in namen:
7      sage_hallo(name)
8      sleep(1)
9
10 print("Ende")
11
```

➔ Hier werden die Einträge einer Liste im Sekundentakt angezeigt.

Befehle:

print kann **Text** (String) ausgeben, dieser wird mit Anführungszeichen markiert: `1 print("Text")`

print kann auch den **Inhalt einer Variablen** mit dem Namen **text** ausgeben: `1 print(text)`

➔ Mehr zu **Variablen** auf dem **AB-12**!

Aufgaben:

10.1) Ergänze dein Ampelprogramm, dass du in **9.1)** mit Unterprogrammen ausgestattet hast um eine **Textausgabe**. In jeder Ampelphase soll ausgegeben werden **welche LED jeweils leuchten**. *(Beispiel Rotphase: „red on“)*

10.2) Prüfe ob die Textausgabe mit dem Leuchten der LEDs übereinstimmt!

04.2026 (Sch)

Microcontroller-Programmierung mit MicroPython (ESP32)

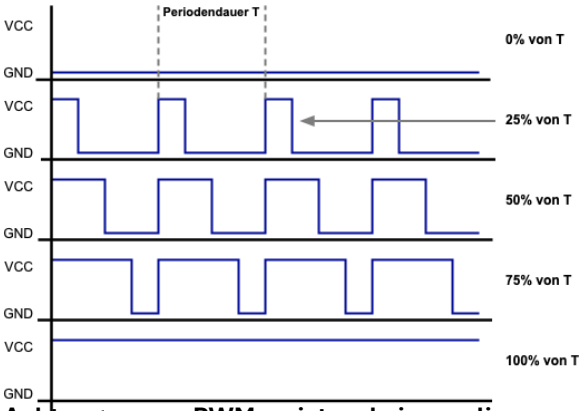
AB-11: Tonausgabe

Info:

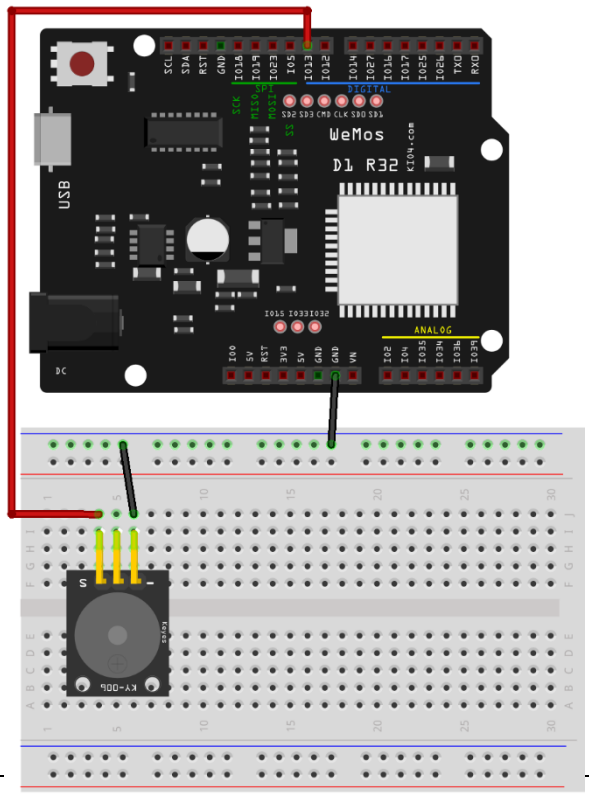
Mit einem **passiven Piezo-Buzzer** (Mini-Lautsprecher), können über den ESP32 einfache **Piep-Töne** in unterschiedlicher Höhe ausgegeben werden. Ein Piezo-Buzzer reagiert auf **Spannungsänderungen** und dafür wird **Pulsweitenmodulation (PWM)** eingesetzt.

PWM erklärt:

Die Spannung wird über den **Duty Cycle** (0-1023) gesteuert.



Achtung: PWM ist keine lineare Lautstärkeregelung! Ein Buzzer ist etwa bei einem mittleren Duty Cycle am lautesten. Wir arbeiten mit einem Wert von **512**.



Verwendung:

Im folgenden Beispiel wird ein Ton im Sekunde

Beachte: PWM muss als Bibliothek geladen werden!

```
1  from machine import Pin, PWM
2  from time import sleep
3
4  buzzer = PWM(Pin(13))    # PWM defines Pin as OUT
5  buzzer.freq(440)         # Frequency of tone must not be 0
6
7  while True:
8      buzzer.duty(512)         # PWM-duty cycle (leave at 512)
9      sleep(1)
10     buzzer.duty(0)           # Stop Tone
11     sleep(1)
```

Befehle:

PWM(Pin(13))
Pin 13 wird als Ausgangspin mit 1024 Stufen definiert

buzzer.freq(440)
bestimmt die Tonhöhe (A)

buzzer.duty(512)
bestimmt Verhältnis an/aus

Bei PWM an einem Pin muss dieser nicht zusätzlich als OUT deklariert werden. Es ist aber auch möglich dies der Übersicht halber zu tun: `4 buzzer = PWM(Pin(13, Pin.OUT))`

Auf diese Weise können **mit Listen** sogar **Melodien** wiedergegeben werden.

04.2026 (Sch)

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-11: Tonausgabe

Auf diese Weise können **mit Listen** sogar **Melodien** wiedergegeben werden.

a)

```
from machine import Pin, PWM
from time import sleep

buzzer = PWM(Pin(13))

melody = [262, 294, 330, 349, 392, 440, 494, 523]  # C-Major Scale

for tone in melody:
    buzzer.freq(tone)
    buzzer.duty(512)
    sleep(0.5)

buzzer.duty(0)
```

→ In diesem Beispiel wir eine C-Dur-Tonleiter wiedergegeben.

---

b)

```
from machine import Pin, PWM
from time import sleep

buzzer = PWM(Pin(13))

# C-Mayor Scale
melody = [
    (262, 0.3),  # C
    (294, 0.3),  # D
    (330, 0.3),  # E
    (349, 0.3),  # F
    (392, 0.3),  # G
    (440, 0.3),  # A
    (494, 0.3),  # H
    (523, 0.3)   # C
]

for freq, duration in melody:
    buzzer.freq(freq)
    buzzer.duty(512)
    sleep(duration)
    buzzer.duty(0)
    sleep(0.05)
```

→ In diesem Beispiel wird mit einem zweiten Wert die Länge des Tones ergänzt.

Aufgabe:

11.1) Programmiere die Melodie von „Alle meine Entchen“.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-12: Variablen und Datentypen

Info:

Variablen sind unverzichtbare Elemente beim Programmieren. Eine Variable ist ein **Platzhalter** bzw. ein **Speicherplatz für Daten** (z.B. Zahlen oder Texte). Man kann sich das vorstellen wie eine **Box**, in der **ein bestimmter Wert abgelegt** ist. In MicroPython ist der Umgang mit Variablen einfach. Im Gegensatz zu anderen Sprachen **muss kein bestimmter Datentyp angegeben werden**, sondern der **Typ wird automatisch erkannt**.

Dennoch müssen Variablen **deklariert** (erstellt) und **initialisiert** (mit Inhalt versehen) werden. Dies passiert bei MicroPython aber in einem Schritt:

Beispiel: **X=10**
Das bedeutet, es wird die **Variable** mit dem **Namen X** erstellt und mit dem **Inhalt 10** befüllt. MicroPython erkennt, dass es sich hierbei um **ganze Zahlen** (ohne Komma) handeln muss und folglich ist **X ein Integer (int)**.

Wichtig: Der Operator = weist der linken Seite etwas zu!

Typischerweise findet das relativ am Anfang eines Scripts statt, z.B. nach der Initialisierung von Hardware wie LEDs, Tastern oder Displays.

Zentrale Datentypen in MicroPython:

X=10	**integer**	Ganze Zahlen
X=3.14	**float**	Kommazahlen (Vorsicht: „Punkt“, statt „Komma“!)
X=“Hallo“	**string**	Text (Vorsicht: Anführungszeichen beachten!)
Status=True	**boolean**	Wahrheitswert, nur **True/False** (Schreibweise!)
Hinweis:	**print(type(X))**	Gibt Typ der Variable zurück, falls man unsicher ist

Wichtig: **Datentypen in MicroPython können sich jederzeit ändern.**
Die jeweils letzte Initialisierung bestimmt deren aktuellen Typ/Inhalt!

Beispielcode:

```
1 X=10
2 X="Hallo"
3 print(type(X))
```

Ausgabe:

```
>>> %Run -c $EDITOR_CONTENT

  MPY: soft reboot
  <class 'str'>

>>>
```

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-12: Variablen und Datentypen

Beispielprogramme:

a)

```
1 A=4
2 B=5
3 C=A*B
4 D="*"
5 E="ergibt"
6 print(A,D,B,E,C)
```

b)

```
1 a=5
2 b=17-8
3 c=2*a+5-14
4 a=7
5 print("a =",a)
6 print("b =",b)
7 print("c =",c)
```

Aufgabe:

12.1) Was geben die Programme in **a)** und **b)** in der Kommandozeile aus?

Operatoren in Python:

Mathematische Operatoren:

+	→ Addition
-	→ Subtraktion
*	→Multiplikation
/	→Division

Vergleichsoperatoren:

==	→ Beide Seiten gleich?
!=	→ Beide Seiten ungleich?
<	→ kleiner als
>	→ größer als
<=	→ größer oder gleich
>=	→ kleiner oder gleich

Allgemeine Speicherung von digitalen Daten:

Ein Computer speichert Daten binär, also als eine Kette von 1ern und 0ern.

1 Bit	kleinste Einheit	1 oder 0
1 Byte	entspricht 8 Bit	10101101 (= 173, siehe Tabelle)
1 KByte	1000 Byte	→ x1000: Megabyte, Gigabyte, Terabyte, usw...

➔ Ein Byte (8 Bit) kann 256 verschiedene Zustände speichern (0-255).

Umrechnung:

Dezimal:	128	64	32	16	8	4	2	1
Binär:	1	0	1	0	1	1	0	1

Aufgabe:

12.2) Wie viele Zustände kann man mit 4Bit speichern?

04.2026 (Sch)

Microcontroller-Programmierung mit MicroPython (ESP32)

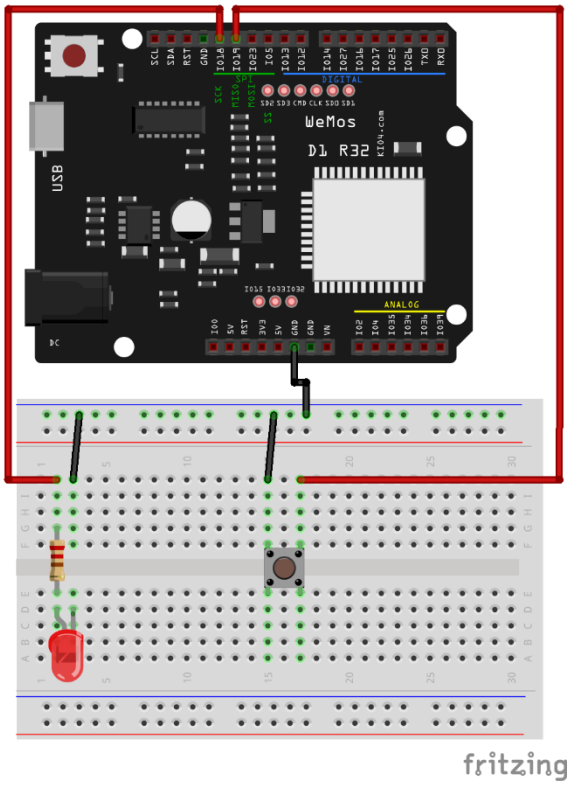
AB-13: Taster und Entprellung

Info:

Eine weitere Standardkomponente neben der LED ist ein **Taster** (button). Die Taster, die wir verwenden, **schließen einen Stromkreis** nur so lange, wie sie gedrückt gehalten werden (momentary button). Der Taster verfügt über 4 Anschlüsse. Die sich gegenüberliegenden Anschlüsse sind jeweils elektrisch verbunden auch wenn der Taster nicht gedrückt ist. **Technisch gehen, gibt es daher nur zwei Anschlüsse, die auf jeder Seite nebeneinander liegen**. Einer davon ist an einem GPIO (Pin) des ESP angeschlossen und der andere an der Erdung (GND**). Wird der Taster gedrückt, wird der Stromkreis geschlossen** und es fließt Strom.

Wichtig:	Intern wird der GPIO, der im Code für einen Taster vorbereitet ist, mit einem sogenannten **Pull-UP Widerstand** verbunden. Das bedeutet, dass sich die **Logik des Schalters umkehrt**. Der Pin des Schalters hat daher im Normalzustand den Zustand 1 und **im gedrückten Zustand den Wert 0!**

Beispielaufbau:



a)

```
from machine import Pin
button = Pin(19, Pin.IN, Pin.PULL_UP)
led= Pin(18, Pin.OUT)
while True:
    if button.value()==0: #wenn Taste gedrückt
        led.value(1)      #LED geht an
    else:                 #wenn Taste nicht gedrückt
        led.value(0)      #LED geht aus
```

b)

```
from machine import Pin
button = Pin(19, Pin.IN, Pin.PULL_UP)
led= Pin(18, Pin.OUT)
while True:
    if button.value()==0: #wenn Taste gedrückt
        led.value(1)      #LED geht an
    if button.value()==1: #wenn Taste nicht gedrückt
        led.value(0)      #LED geht aus
```

Tipp:	*Diesen Schalter-LED-Aufbau kann man ganz einfach mit einem Kabel an Pin19 und der internen LED an Pin2 testen.*

Im Code müssen wir nun mit einer **Bedingung** arbeiten um den **Zustand des Schalters abzufragen**. Der Ablauf des Codes wird **verzweigt**, da **verschiedene Fälle** betrachtet werden müssen. Hierfür brauchen wir eine **if-else Struktur** (a). Grundsätzlich können Zustände auch **nur mit if** geprüft werden. Das hat aber den Nachteil, dass immer **alle if Zustände geprüft werden** müssen (b). Bei **boolschen Zuständen** (1 oder 0) wie bei Tastern und LEDs halten wir uns daher immer an eine **if-else** Struktur wie in **a)**.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-13: Taster und Entprellung

Rastende Taster und Entprellung:

Der gerade eingeführte Taster hat den großen **Nachteil**, dass er **gedrückt gehalten werden muss** um etwas auszulösen. Um dieses Problem zu lösen, brauchen wir einen **rastenden Schalter** (toggle-Schalter). Grundsätzlich ist der Code dafür sehr einfach:

c)

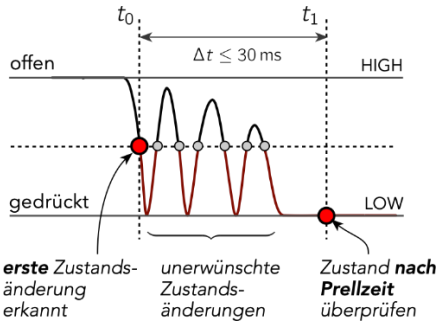
```
from machine import Pin
from time import sleep
button = Pin(19, Pin.IN, Pin.PULL_UP)
led=Pin(18, Pin.OUT)
while True:
    if button.value() == 0:
        led.toggle()
        sleep(0.2) #Einfache Entprellung
```

else ... **if** wird abgekürzt zu **elif** →

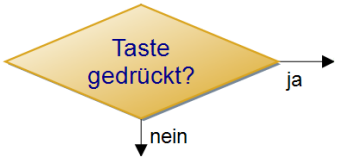
d)

```
from machine import Pin
from time import sleep
button = Pin(19, Pin.IN, Pin.PULL_UP)
led=Pin(18, Pin.OUT)
pressed=False
while True:
    if button.value()==0 and pressed==False:#Hineindrücken der Taste
        led.toggle()                        #LED wird umgeschaltet
        pressed=True                        #Zustandsvariable schalten
        sleep(0.2)                          #kurze Wartezeit
    elif button.value()==1 and pressed==True: #Loslassen der Taste
        pressed=False                       #Ausgagszustand herstellen
```

Ein Problem bei Tastern ist es, dass sie **nicht sofort beim Drücken von einem Zustand zum anderen übergehen**, sondern **„prellen“**. Das bedeutet, dass sie innerhalb eines kurzen Zeitraumes **mehrfach den Zustand wechseln**. Man muss einen Taster daher **„entprellen“**. Am einfachsten geht das wie in **c)** mit einer **kurzen Wartezeit**.



Taster werden in **PAPs** mit Verzweigungen dargestellt:



(Bildquelle: https://spacehal.github.io/docs/arduino/debounce)

Der Nachteil der Variante in **c)** ist es, dass man durch dauerhaftes Drücken oder eine zu kurz gewählte Entprellzeit noch immer ungewollte Zustandsänderungen erhält. Daher verwenden wir eine Entprellung mit einer Zustandsvariablen wie in **d)**. Man kann die Variante aus **d)** noch etwas **vereinfachen**:

e)

```
from machine import Pin
from time import sleep
button = Pin(19, Pin.IN, Pin.PULL_UP)
led=Pin(18, Pin.OUT)
pressed=False
while True:
    if not button.value() and not pressed:  #Hineindrücken der Taste
        led.toggle()                        #LED wird umgeschaltet
        pressed=True                        #Zustandsvariable schalten
        sleep(0.2)                          #kurze Wartezeit
    if button.value() and pressed:          #Loslassen der Taste
        pressed=False                       #Ausgagszustand herstellen
```

Aufgaben:

13.1) Zeichne jeweils einen Programmablaufplan der Varianten a), c) und d).

13.2) Lasse die Kommandozeile den Zustand der LED anzeigen. (z.B. „Licht an“)

04.2026 (Sch)

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-14: Schleifen

Info:

Beim Programmieren tritt immer wieder die Situation auf, dass man **Teile des Codes wiederholen** will. Für diesen Zweck bieten sich **Schleifen (loops)** an, die ihren Inhalt mehrfach oder dauerhaft wiederholen. Es gibt **zwei verschiedene Typen von Schleifen.**

for-Schleife:

a)

```
1 from time import sleep
2 for x in range (10): #Schleife läuft 10x (0-9)
3     print(x)         #Rundennummer wird angezeigt
4     sleep(1)         #Eine Sekunde Wartezeit
```

Die Schleife **läuft insgesamt 10 Runden** (siehe Zeile 2). **Der Inhalt der Schleife wird eingerückt** und die Schleife gibt lediglich ihre Rundennummer in der Kommandozeile im Abstand von einer Sekunde aus. **(Beachte: Die 10 wird nicht erreicht, da die 0 mitzählt!)**

b)

```
1 from time import sleep
2 fruits = ["apple", "banana", "cherry", "strawberry", "kiwi"]
3 for x in fruits:
4     print(x)
5     sleep(1)
```

Man kann auch die **Inhalte einer Liste** ausgeben. Wie in a) passiert das im Sekundentakt.

c)

```
1 from machine import Pin
2 from time import sleep
3 led= Pin(2, Pin.OUT)
4 for x in range (10):
5     led.toggle()     #LED schalten
6     sleep(0.5)       #eine halbe Sekunde warten
7     led.toggle()     #LED schalten
8     sleep(0.5)       #eine halbe Sekunde warten
```

Diese Schleife lässt die interne **LED 10x blinken**. Beachte, dass die LED dafür innerhalb eines Schleifenzyklus **zwei Mal geschaltet** werden muss (an und wieder aus)!

while-Schleife:

Im Unterschied zur for-Schleife läuft die while-Schleife so lange wie ihre Bedingung wahr (True) ist. Diese Art Schleife kann auch vorzeitig abgebrochen (break) werden. Häufig wird die while-Schleife eingesetzt um den Inhalt eines Programms dauerhaft laufen zu lassen.

d)

```
1 from machine import Pin
2 from time import sleep
3 led= Pin(2, Pin.OUT)
4 while True: #ist immer "wahr"
5     led.toggle() #LED schalten
6     sleep(0.1)
```

e)

```
1  from machine import Pin
2  from time import sleep
3  led= Pin(2, Pin.OUT)
4  button = Pin(19, Pin.IN, Pin.PULL_UP)
5  while True: #Bedingung "True" ist immer wahr
6      led.toggle() #LED schalten
7      sleep(0.1)
8      if button.value()==0:    #Wenn Taste gedrückt
9          print("User abort") #Meldung "Abbruch"
10         break               #Abbruch der Schleife
```

Während in d) die LED ständig blinkt, kann das in e) durch Tastendruck abgebrochen werden. Beachte: Durch die Verwendung von sleep wird ein Tastendruck möglicherweise nicht zuverlässig erkannt wenn er zu schnell erfolgt, da der ESP kurzzeitig blockiert ist.

04.2026 (Sch)

f)

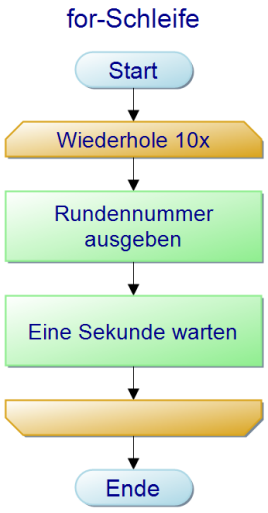
```
from time import sleep
x=0
while x<10:  #Schleife läuft solange x<10
    x+=1     #ist das Gleich wie x=x+1
    print(x) #Variable wird ausgegeben
    sleep(1) #Wartezeit von einer Sekunde
print("end of while-loop") #Aktion nach Schleife
```

Man kann auch andere Bedingungen für eine while-Schleife festlegen. In f) verhält sich die Schleife **wie eine for-Schleife**, denn ihre Bedingung ist, dass x<10 ist, damit sie läuft. Wenn jetzt in jeder Runde x um 1 erhöht wird, dann ist die Bedingung nach 10 Runden ungültig und die Schleife endet. In diesem Fall wird dann das Kommando direkt nach der Schleife ausgeführt.

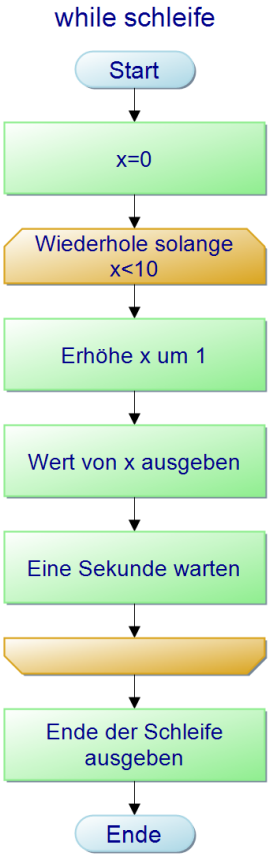
Schleifen im Programmablaufplan (PAP):

Die Schleifen in **a)** und **f)** lassen sich auch mit Programmablaufplänen visualisieren:

a)



b)



Microcontroller-Programmierung mit MicroPython (ESP32)

AB-15: Zufallsgenerator

Info:

Sehr hilfreich beim Programmieren sind **zufällig generierte Werte**. Damit lassen sich **Vorgänge unvorhersehbar gestalten**, was beispielsweise bei Spielen nötig wird. Der **Zufallsgenerator** beim ESP32 funktioniert vergleichsweise einfach.

Die Funktion **randint(a,b)** wird aus **random** importiert und hat zwei Argumente (a,b).

Die Argumente **a und b** sind die jeweiligen **Grenzwerte** des Zahlenbereiches. Wenn also **a=randint(1,100)** ausgeführt wird, schreibt der Zufallsgenerator eine ganze Zahl **zwischen 1 und 100** in die Variable a. Die **Grenzwerte sind eingeschlossen (1 & 100).**

Beispiele:

a)

```
1 from random import randint
2 for x in range(5):
3     number = randint(1,100)
4     print(number)
```

In diesem Beispiel würfelt eine for-Schleife fünf Mal eine zufällige Zahl zwischen 1 und 100. Die Zahl wird anschließend in der Kommandozeile ausgegeben.

Aufgaben:

Erstelle ein Notenwürfelprogramm. Folgender Ablauf soll dabei realisiert werden:

Wenn ein (entprellter) **Taster gedrückt** wird, soll eine **zufällige Schulnote** zwischen 1 und 6 **gewürfelt** werden. Wenn die Note 5 oder 6 ist, soll eine vorhandene **LED aufleuchten** und die **Meldung** „durchgefallen“ ausgegeben werden. Bei Noten zwischen 1 und 4 soll die LED nicht leuchten und die Meldung „bestanden“ kommen.

15.1) Erstelle einen Programmablaufplan (PAP).
15.2) Schreibe das Programm und prüfe seine Zuverlässigkeit.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-16: Nicht-blockierendes Warten (ticks_ms & ticks_diff)

Info:

Wenn wir mit dem ESP32 **eine gewisse Zeit warten** wollen, haben wir bisher das Kommando **sleep** verwendet. Das funktioniert für reine Wartezeiten zwar zufriedenstellend, aber hat den entscheidenden **Nachteil**, dass während dieser Zeit der ESP32 nicht verwendet werden kann – die **Programmausführung ist blockiert**. Um zu warten und **gleichzeitig Aktionen** ausführen zu können, verwenden wir **ticks_ms** und/oder **ticks_diff**.

Beispiele:

a)

```
1  from time import sleep
2  from machine import Pin
3  button = Pin(18, Pin.IN, Pin.PULL_UP)
4  print("You have 5 seconds to press the button")
5  sleep(5)
6  if button.value()==0:
7      success = True
8      print("success!")
9  else:
10     print("loser!")
```

b)

```
1  from time import sleep, ticks_ms
2  from machine import Pin
3  button = Pin(18, Pin.IN, Pin.PULL_UP)
4  print("You have 5 seconds to press the button")
5  success=False
6  start=ticks_ms()
7  while start+5000>ticks_ms():
8      if button.value()==0:
9          success = True
10         print("success!")
11         break
12 if not success:
13     print("loser!")
```

In diesem Beispiel hat der Nutzer **5 Sekunden Zeit einen Taster zu drücken**. Wenn er es schafft, dann wird die Meldung „success!“ ausgegeben, wenn er es nicht schafft „loser!“. Das Problem in **a)** ist, dass **erst nach Ablauf der 5 Sekunden kurz geprüft wird**, ob die Taste zu diesem Zeitpunkt gedrückt (gehalten) wird. Das Programm **reagiert bei Tastendrücken davor nicht**, da der ESP32 durch sleep(5) **blockiert** ist.

In **b)** erfolgt die Zeitmessung über **ticks_ms**. Die Funktion muss **zuvor importiert** werden. In ticks_ms können jederzeit die **Millisekunden seit dem Start des ESP32** ausgelesen werden. Der Wert in ticks_ms kann nicht beeinflusst werden. Die while-Schleife läuft so lange wie der ausgelesene **Startwert und die Wartezeit größer sind wie ticks_ms**. Das ist auch eine grundsätzlich praktikable Lösung, allerdings **läuft ticks_ms nach etwa 12 Tagen über**.

Überlauf: Variablen können überlaufen, wenn ihr Speicherbereich ausgeschöpft ist. Die Variable **ticks_ms** ist auf **30 Bit** beschränkt (2^{30} =1.073.741.824 ms = 12,4 Tage). Wenn eine Variable überläuft, läuft sie von ihrem Startwert (0) weiter.

Die solide Lösung ist hier **c)**. In dieser Variante brauchen wir noch zusätzlich **ticks_diff**, das wir ebenfalls noch **importieren** müssen. Mit **ticks_diff** können wir **vergangene Zeit berechnen**, indem der aktuelle ticks_ms Wert mit einem vorher ausgelesenen und in einer Variable gespeicherten Wert vergleichen wird (siehe Zeile 7). Im Gegensatz zu ticks_ms **erkennt ticks_diff einen Überlauf** und funktioniert trotzdem noch. Daher ist diese Lösung **die bevorzugte Variante**.

c)

```
1  from time import sleep, ticks_ms, ticks_diff
2  from machine import Pin
3  button = Pin(18, Pin.IN, Pin.PULL_UP)
4  print("You have 5 seconds to press the button")
5  success=False
6  start=ticks_ms()
7  while ticks_diff(ticks_ms(),start)<5000:
8      if button.value()==0:
9          success = True
10         print("success!")
11         break
12 if not success:
13     print("loser!")
```

Vorsicht:

ticks_ms und **ticks_diff** arbeiten mit **Millisekunden**, im Gegensatz zur **sleep**, das mit **Sekunden** arbeitet.

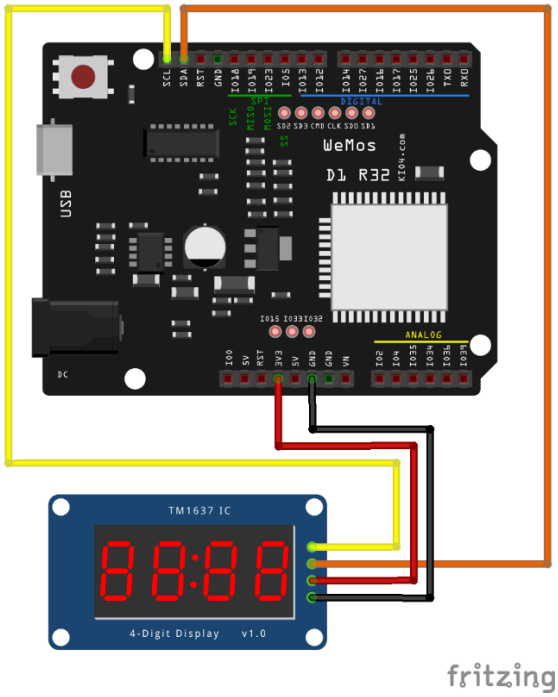
Microcontroller-Programmierung mit MicroPython (ESP32)

AB-17: TM1637 Display

Info:

Bisher erfolgen **Rückmeldungen vom ESP32** primär über die Kommandozeile in Thonny. Das hat den Nachteil, dass immer ein PC benötigt wird, was vor allem portable Projekte erschwert. Mit einem **Display** lässt sich die **Rückmeldung überall** umsetzen und es ist nur noch eine einfache Stromversorgung (z.B. Batterie) nötig.

Aufbau:



Hinweis:

Das Display kann mit **VCC** wahlweise an **3,3V** oder **5V** angeschlossen werden. Auf 5V ist das Display dann heller.

Tipp:

Mit **display.show("Text")** kann auch Text angezeigt werden. Manche Buchstaben werden dabei an das Display angepasst.

TM1637-Display:

Das TM1637-Display ist ein einfaches **Display mit vier Ziffern** von 1-9 und einem Doppelpunkt in der Mitte. Für die **Anzeige von Zahlen** ist es ideal geeignet. Um es ansteuern zu können, ist eine **externe Bibliothek** nötig. Diese muss auf den ESP32 als **py-Datei** geladen werden. Dann muss sie nur noch **importiert** werden.

a)

```
from machine import Pin
from time import sleep
import tm1637
display = tm1637.TM1637(clk=Pin(22), dio=Pin(21))

while True:
    display.numbers(23, 59, colon=True)
    sleep(0.5)
    display.write([0,0,0,0]) #clear Display
    sleep(0.5)
```

Im Skript müssen die **CLK** (Clock) und **DIO** (Data Input-Output) korrekt angegeben werden. In diesem Beispiel blinkt das Display dann wie ein klassischer Digital-Wecker.

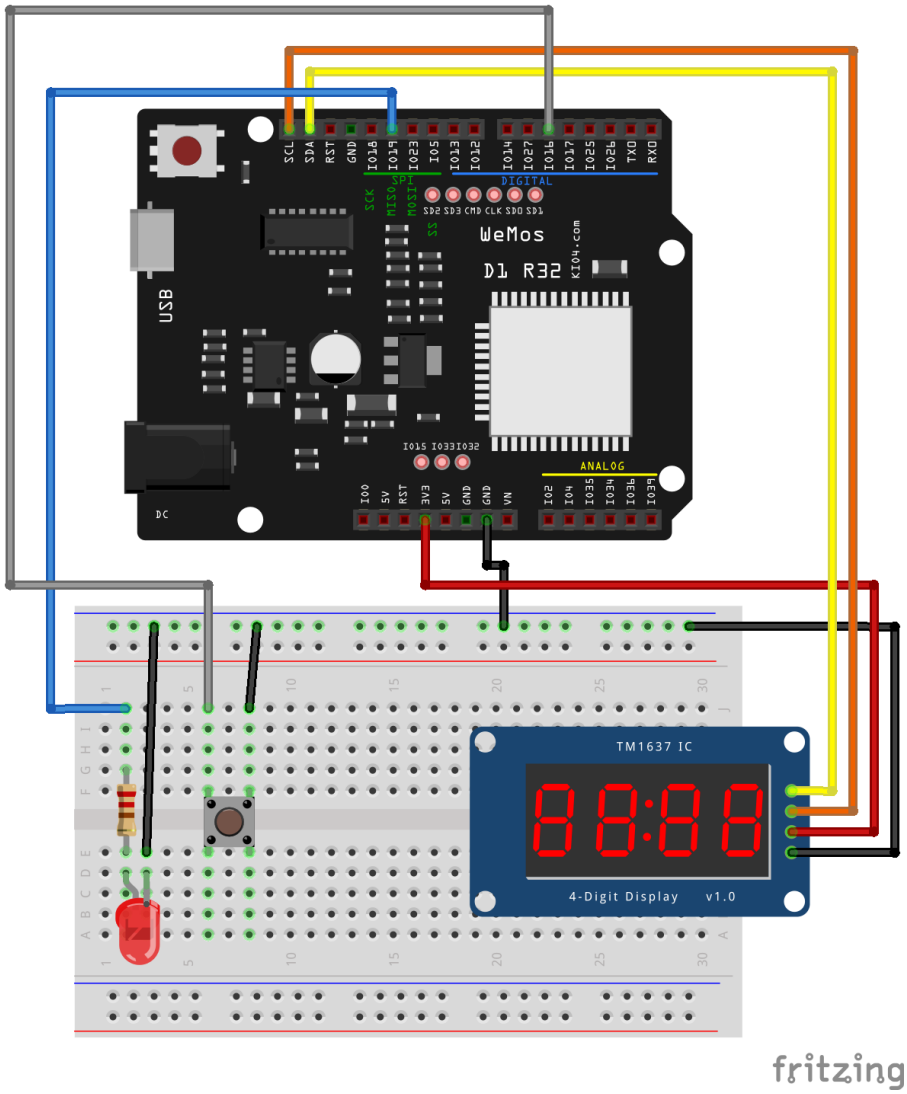
Microcontroller-Programmierung mit MicroPython (ESP32)

AB-18: Der Reaktionstester

Funktionsweise des Reaktionstesters:

2-5 Sekunden nach dem Start des Programms soll eine LED aufleuchten. Es soll dann innerhalb der nächsten 10 Sekunden eine Taste gedrückt werden können. Die LED soll mit dem Drücken des Schalters ausgehen und die gemessene Zeit vom Aufleuchten der LED bis zum Drücken des Schalters auf dem Display angezeigt werden. Falls nach 10 Sekunden kein Tastendruck erfolgt ist, soll dies ebenfalls angezeigt werden (SLOU).

Aufbau:



Aufgaben:

18.1) Erstelle einen Programmablaufplan zur Funktionsweise des Programms.
18.2) Versuche das Programm zu schreiben, zunächst ohne Display.
18.3) Baue das TM1637 Display ein.
18.4) Verändere das Programm, sodass mit dem Taster der Test neu gestartet wird.
18.5) Passe das Programm an, sodass es ein Unterprogramm enthält.

04.2026 (Sch)

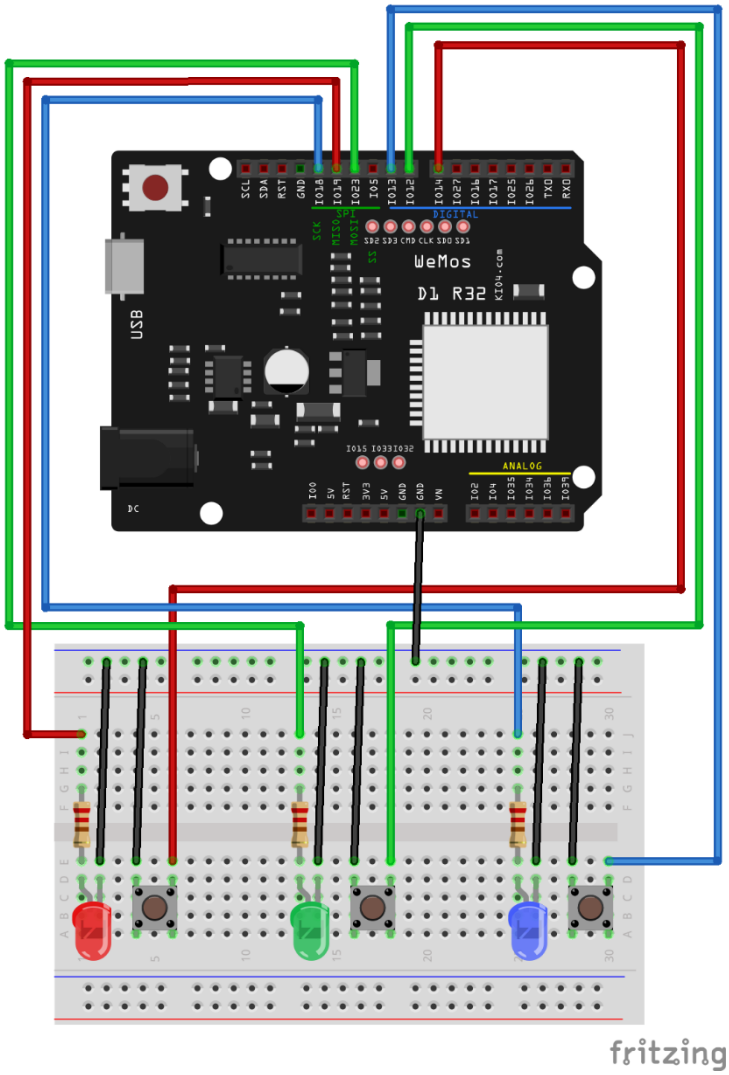
Microcontroller-Programmierung mit MicroPython (ESP32)

AB-19: Das Reaktionsspiel

Funktionsweise des Reaktionsspiels:

Das **Reaktionsspiel** besteht aus drei unterschiedlich gefärbten LEDs und drei Tastern.

Das Spiel soll insgesamt **10 Runden** laufen. In jeder Runde soll **eine der drei LEDs zufällig ausgewählt** werden und nach einer **zufälligen Wartezeit** (2-5 Sekunden) für eine **zufällige Zeit (ca. 0,3 bis 1 Sekunde) aufleuchten**. Während des Leuchtens soll der Spieler **per Tastendruck auf die der jeweiligen LED zugeordnete Taste reagieren** können. Nur wenn er **rechtzeitig drückt**, bekommt er einen Punkt. Der **Punktestand** und andere Meldungen sollen zunächst über die **Kommandozeile** ausgegeben werden.



Aufgaben:

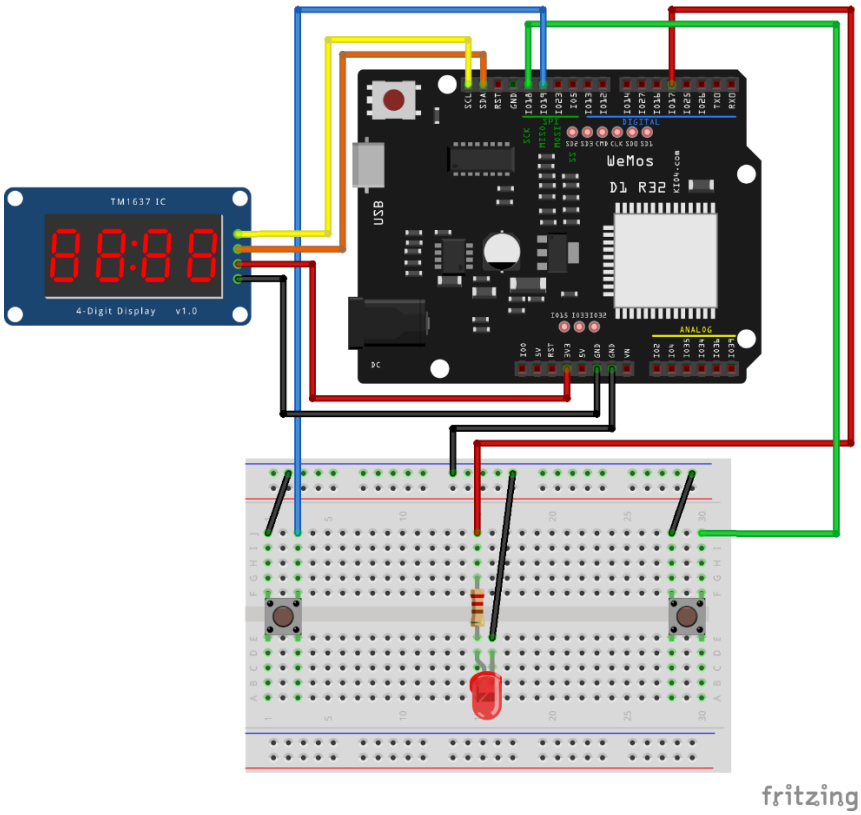
19.1) Erstelle einen **Programmablaufplan**.
19.2) Baue das Spiel nach der Darstellung oben auf.
19.3) Programmiere das Spiel. Nutze **for**- und **while-Schleifen**, **ticks_ms** & **ticks_diff**.
19.4) Baue ein **TM1637**-Display ein um das Spiel ohne PC nutzbar zu machen.

04.2026 (Sch)

Funktionsweise des Reaktionsspiels für zwei Personen:

Das **Reaktionsspiel** besteht aus einer LEDs und zwei Tastern.

Das Spiel soll insgesamt **10 Runden** laufen. In jeder Runde soll **die LED nach einer zufälligen Wartezeit** von 2-5 Sekunden **aufleuchten.** Während des Leuchtens sollen beide Spieler **per Tastendruck** einen Punkt machen können. Die LED geht sofort aus, nachdem einer der Spieler seine Taste gedrückt hat, sodass nur jeweils einer der Spieler punkten kann. Der **Punktestand** soll zunächst über die **Kommandozeile** ausgegeben werden. Das Ende des Spiels soll klar signalisiert werden.



Aufgaben:

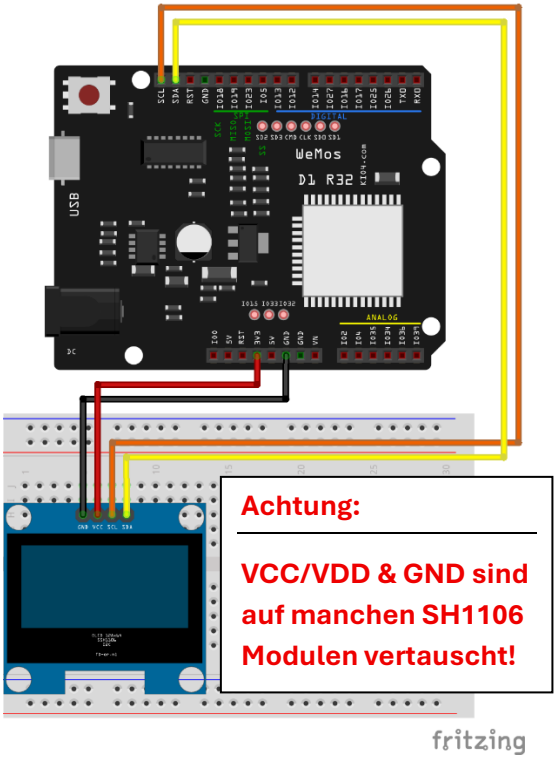
20.1) Erstelle einen **Programmablaufplan**.
20.2) Baue das Spiel nach der Darstellung oben auf (zunächst ohne TM1637-Display).
20.3) Programmiere das Spiel. Nutze **for**- und **while-Schleifen**, **ticks_ms** & **ticks_diff**.
20.4) Baue ein **TM1637**-Display ein um das Spiel ohne PC nutzbar zu machen.
20.5) Bestrafe ein **zu frühes Drücken** der Spielertasten mit einem **Minuspunkt**.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-21: OLED-Display (SH1106)

Info:

Wenn um **Text und Grafik** geht, dann ist ein **OLED-Display** ideal. Die OLED-Technik zeichnet sich durch **gute Ablesbarkeit** auch bei kleinen Bildschirmen aus. Ein **Schwarz-Weiß-Display** ist meist vollkommen ausreichend. Die häufigste Auflösung sind **128x32** und **128x64** Pixel. Das hier verwendete Display benötigt einen **SH1106-Treiber** und ist etwas größer als vergleichbare Displays mit gleicher Auflösung.



I2C-Bus:

Das **SH1106-Display** kommuniziert mit dem ESP32 über den **I2C-Bus**. Das ist eine Schnittstelle, die mit **nur zwei Datenleitungen** auskommt. In der Regel sind diese mit **SCL** (Clock) und **SDA** (Data) gekennzeichnet. Der ESP32 hat für diesen Bus spezielle Ports, auch wenn durchaus **andere Pins verwendet werden können**. Es können **mehrere Geräte gleichzeitig** an diesem Bus angeschlossen werden. Dann ist aber darauf zu achten, dass es **keine Adresskonflikte** gibt. Für diesen Zweck kann ein **I2C-Scanner** (siehe Rückseite) verwendet werden um **angeschlossene Geräte** und **potentielle Konflikte** zu erkennen.

Es gibt verschiedene SH1106-Treiber. Man kann die zur Verfügung stehenden Funktionen unter Verwendung von künstlicher Intelligenz erweitern.

SH1106 → VDD= 3,3V/5V!

Textdarstellung:

```
from machine import Pin, I2C
from sh1106 import SH1106_I2C

i2c = I2C(0,sda=Pin(21), scl=Pin(22))
oled = SH1106_I2C(128, 64, i2c)
oled.flip(True)
oled.invert(False)
oled.reset()

oled.fill(0)

oled.text('Line 1 16 Chars!',0,0)
oled.text('Line 2 16 Chars!',0,8)
oled.text('Line 3 16 Chars!',0,16)
oled.text('Line 4 16 Chars!',0,24)
oled.text('Line 5 16 Chars!',0,32)
oled.text('Line 6 16 Chars!',0,40)
oled.text('Line 7 16 Chars!',0,48)
oled.text('Line 8 16 Chars!',0,56)

oled.show()
```

Hinweis:

Das Display kann in **8 Zeilen jeweils 16 Zeichen** anzeigen. Somit sind insgesamt **128 Zeichen** darstellbar. Die jeweilige **Darstellungsposition** muss über **X/Y-Koordinaten** angegeben werden. Der **Ursprung (0/0) ist oben links**. Falls das Display **um 180 Grad gedreht** darstellt, hilft **oled.flip(True)** weiter.

oled.invert(True) invertiert die Farben.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-21: OLED-Display (SH1106)

Grafikdarstellung:

```
from machine import Pin, I2C
from sh1106 import SH1106_I2C

i2c = I2C(0,sda=Pin(21), scl=Pin(22))
oled = SH1106_I2C(128, 64, i2c)
oled.flip(True)
oled.invert(False)
oled.reset()

oled.fill(0)

oled.rect(8,8,16,16,1)
oled.fill_rect(8,32,16,16,1)

oled.ellipse(50,15,8,8,1)
oled.fill_ellipse(50,39,8,8,1)

oled.triangle(95,23,75,23,85,8,1)
oled.fill_triangle(95,47,75,47,85,32,1)

oled.show()
```

Hinweis:

Mit einem **modifizierten Treiber** lassen sich nicht nur Rechtecke darstellen, sondern wie hier im Beispiel auch Ellipsen (Kreise) und Dreiecke.

Rechteck: rect(x, y, w, h, color)

Ellipse: ellipse(x, y, xr, yr, color)

Dreieck: triangle(x0, y0, x1, y1, x2, y2, color)

→ Bei einem S/W-Display ist die **Farbe=1**.

I2C-Scanner:

```
import machine
i2c = machine.I2C(scl=machine.Pin(22), sda=machine.Pin(21))

print('Scan i2c bus...')
devices = i2c.scan()

if len(devices) == 0:
  print("No i2c device !")
else:
  print('i2c devices found:',len(devices))

  for device in devices:
    print("Decimal address: ",device," | Hexa address: ",hex(device))
```

Der I2C-Scanner meldet zurück, wie viele I2C-Geräte gefunden wurden und welche Adressen sie haben. Er funktioniert natürlich nur bei der korrekten Angabe der Pins.

Beispiel:

```
MPY: soft reboot
Scan i2c bus...
i2c devices found: 1
Decimal address:  60  | Hexa address:  0x3c
```

Aufgabe:

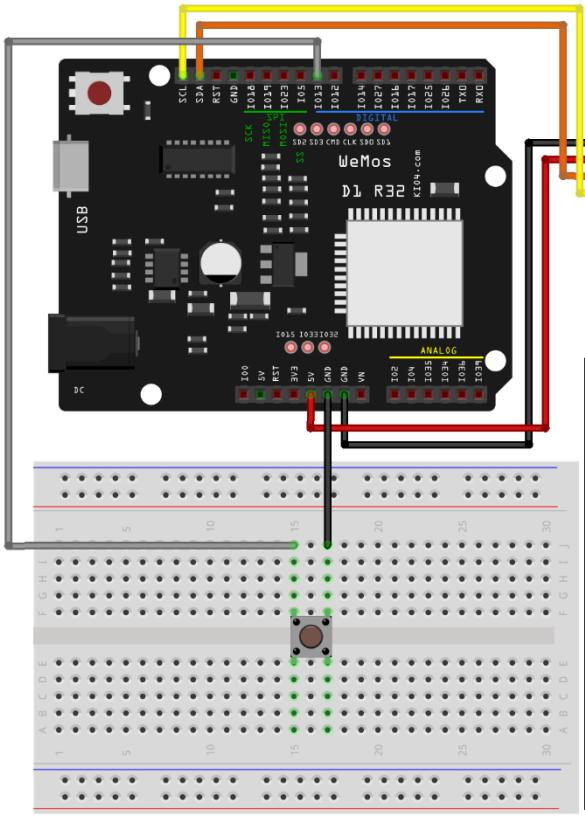
21.1) Baue in deine Ampel mit Unterprogrammen aus 9.1) ein SH1106 mit der grafischen Darstellung von Ampelphasen ein. Ergänze dazu auch eine Textausgabe.

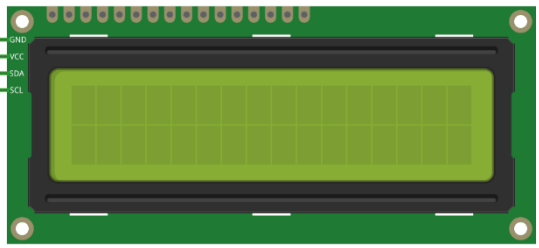
Microcontroller-Programmierung mit MicroPython (ESP32)

AB-22: LCD-Display (1602)

Info:

Wenn ein **Display nur Text** darstellen soll, dann ist meist ein **1602-LCD** ausreichend. Es kann jeweils **16 Zeichen in 2 Zeilen** darstellen. Es gibt auch **Varianten mit 20x4** Zeichen (2004-LCD). Das Display ist groß und **leicht ablesbar**, wenn der **Kontrast** mit dem Regler an der Rückseite korrekt eingestellt ist. Der **Nachteil** im Vergleich zum OLED-Display ist der **höhere Stromverbrauch**, denn die Beleuchtung des Displays ist im Dauerbetrieb.





1602-Treiber notwendig!
Das **1602-Display** benötigt **5V**!

Hinweis:

Diese LCD-Displays werden **mit und ohne Adapter** angeboten. Der Adapter macht das Display erst **I2C**-fähig (4 Pins). Ein Betrieb ohne den Adapter empfiehlt sich nicht. Wenn das Display keinen Adapter hat, muss er meist nachträglich verlötet werden, es gibt aber auch 1602-Module ohne (sichtbaren) Adapter, die nur vier Pins haben und I2C-fähig sind.

```
from machine import Pin, I2C
from time import sleep
from random import *
from lcd1602 import LCD
i2c = I2C(0, sda=Pin(21), scl=Pin(22), freq=400000)
lcd = LCD(i2c)
lcd.clear() # '\n' creates a new line.
lcd.message("Random 0-99:\nNumber:00")
button=Pin(13,Pin.IN, Pin.PULL_UP)
led=Pin(5, Pin.OUT)
number=0
x=False
while True:
    if button.value()==0 and x==False:
        number = randint(0,99)
        print(number)
        if number<10:
            lcd.write(0,1, "Number:"+"0"+str(number))
            led.on() # row 2, pos 1, str-to-int
        if number>9:
            lcd.write(0,1, "Number:"+str(number))
            led.off() # row 2, pos 1, str-to-int
        x=True
    if button.value()==1 and x==True:
        x=False
        sleep(0.2)
```

Beispielprogramm:

Beim Drücken auf einen entprellten Taster wird eine Zufallszahl von 0-99 gewürfelt und dann auf dem Display ausgegeben. Wenn die Zahl kleiner als 10 ist, leuchtet zusätzlich eine LED auf.

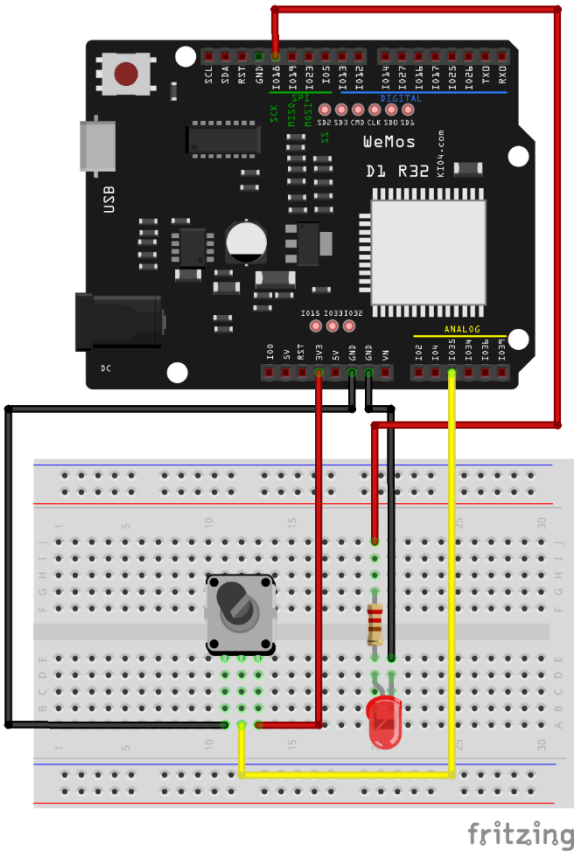
Ähnlich wie beim OLED-Display kann man nur Teile des Displays aktualisieren. In diesem Beispiel wird immer nur die zweite Zeile aktualisiert, da sich die erste Zeile inhaltlich nicht ändert.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-23: Potentiometer

Info:

Mit einem **Potentiometer** (Drehregler) lassen sich **Lautstärkeregelungen** oder **Dimmer** für LEDs realisieren. Durch die Auflösung des Potentiometers (12 Bit in 4096 Stufen) lässt sich eine **stufenlos wirkende Regelung** erreichen. Ein Potentiometer ist ein **verstellbarer elektrischer Widerstand**. Man kann es auch als einen einstellbaren Spannungsteiler verstehen, mit dem man Spannungen zwischen 0V - 3,3V einstellt.



Wichtig!

Das Potentiometer **muss immer an 3,3V** angeschlossen werden. Es darf **keinesfalls an 5V** angeschlossen werden. Die GPIOs des ESP32 sind nicht 5V-tolerant. **Der ESP32 kann Schaden nehmen!**

```
1  from machine import ADC, Pin, PWM
2  from time import sleep
3
4  pot = ADC(Pin(35))
5  pot.atten(ADC.ATTN_11DB) #for range 0-3.3V
6
7  led= Pin(18, Pin.OUT)
8  led = PWM(led)  # PWM-Pin für LED
9
10 while True:
11     val_pot = pot.read()           # 0-4095 (Pot)
12     led.duty(val_pot // 4)         # 0-1023 (PWM)
13     a=val_pot
14     print(a)
15     sleep(0.1)
```

Die äußeren Anschlüsse werden mit **GND** und **3,3V** (siehe Infokasten) verbunden. Wenn sie verpolt (vertauscht) werden, invertiert sich die Drehrichtung des Potentiometers. Der mittlere Kontakt wird mit einem **Analog-In GPIO** verbunden (untere Kontaktleiste). Für die korrekte Funktion des Potentiometers muss im Skript das **Modul ADC** (Analog-Digital-Conversion) importiert werden.

Das ausgelesene Signal in unserem Beispiel liefert **4096 Stufen** (12 Bit=2^{12}). Da PWM an unserer LED nur **1024 Stufen** liefert (10 Bit=2^{10}) **muss erst umgerechnet werden**, bevor der ausgelesene Wert weiterverwendet wird (Zeile12). Die LED lässt sich jetzt in ihrer Helligkeit (nahezu) stufenlos regeln.

Aufgabe:

23.1) Baue ein Stroboskop. Das Potentiometer soll die Blinkfrequenz regulieren.

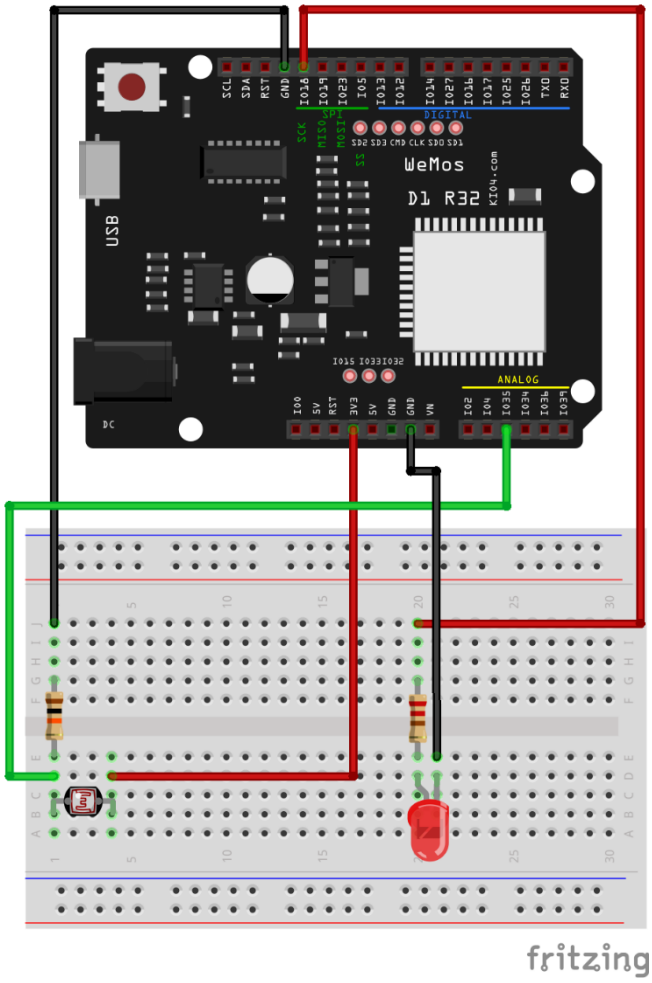
Microcontroller-Programmierung mit MicroPython (ESP32)

AB-24: Lichtabhängiger Widerstand (LDR)

Info:

Ein LDR (Light-Dependent-Resistor) ist ein lichtabhängiger Widerstand. Ist viel Licht vorhanden, ist der Widerstand klein (gute Leitfähigkeit) und wenn wenig Licht vorhanden ist, wird der Widerstand groß. Mit einem LDR kann man Schaltungen bauen, die auf Helligkeit reagieren wie automatische Lampen oder Nachtlichter. Auch Alarmanlagen sind machbar, die darauf reagieren, wenn ein Lichtstrahl unterbrochen wird.

Während ein Potentiometer ein manuell verstellbarer Spannungsteiler ist, funktioniert ein LDR als ein automatischer Spannungsteiler für Licht. Ein LDR ist somit ein Sensor.



Wichtig!

Der LDR **muss immer an 3,3V** angeschlossen werden. Er darf **keinesfalls an 5V** angeschlossen werden. Die GPIOs des ESP32 sind nicht 5V-tolerant. **Der ESP32 kann Schaden nehmen!**

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

ldr = ADC(Pin(35))
ldr.atten(ADC.ATTN_11DB)

led= Pin(18, Pin.OUT)

while True:
    val_ldr = ldr.read()
    if val_ldr>1000:
       led.value(1)
    else:
        led.value(0)
    print("LDR:  "+str(val_ldr))
    sleep(0.1)
```

Aufgabe:

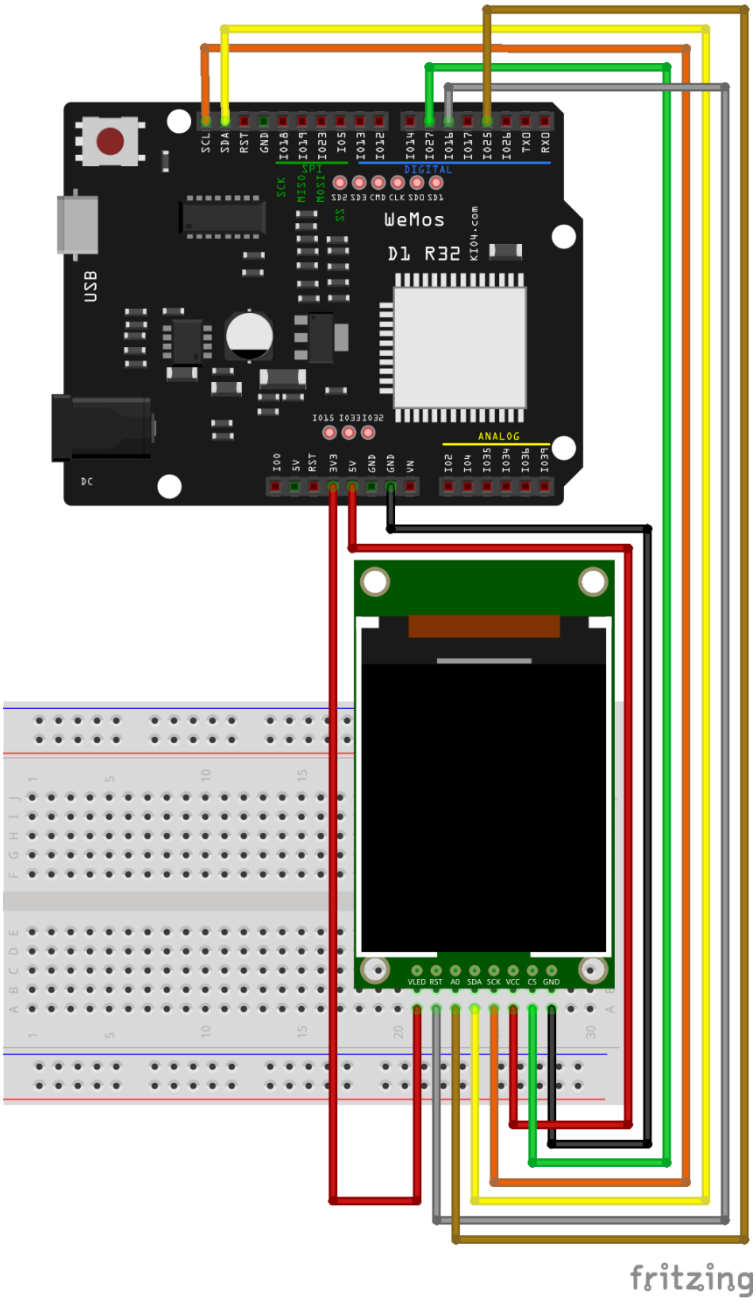
24.1) Baue ein Potentiometer ein, mit dem sich der Schaltwert der LED regulieren lässt.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-25: Farbdisplay (ST7735)

Info:

Das ST7735-Display ist ein relativ günstiges **1,8-Zoll Farbdisplay**. Damit lassen sich auch Bilder anzeigen (16-Bit). Für dieses Display ist ein **Treiber-Ordner** auf dem ESP32 zu hinterlegen, der neben dem eigentlichen Treiber auch noch eine **Schriftart** (Font) enthält.



Die **Verkabelung aufwändiger** im Vergleich zu S/W-Displays. Manche Displays dieser Art haben auch einen SD-Kartenleser, der über zusätzliche Pins angesprochen werden kann. Um Bilder zu laden, müssen diese zuvor auf den ESP-32 als 24-Bit Bitmap-Datei geladen werden (siehe Beispiel).

AB-25: Farbdisplay (ST7735)

a) Textbeispiel:

```
from st7735 import TFT
from st7735 import sysfont
from machine import SPI,Pin

def tftprinttest(font):
    tft.fill(TFT.BLACK);
    v = 0
    tft.text((0, v), "Hello World!", TFT.RED, font, 1, nowrap=True)
    v += font["Height"]
    tft.text((0, v), "Hello World!", TFT.YELLOW, font, 2, nowrap=True)
    v += font["Height"] * 2
    tft.text((0, v), "Hello World!", TFT.GREEN, font, 3, nowrap=True)
    v += font["Height"] * 3
    tft.text((0, v), str(1234.567), TFT.BLUE, font, 4, nowrap=True)

spi = SPI(1, baudrate=20000000, polarity=0, phase=0, sck=Pin(22), mosi=Pin(21), miso=None)
tft=TFT(spi,25,26,27) #A0,Reset,CS,sda=mosi,led=3.3V
tft.initr()
tft.rgb(True)
tft.rotation(1) #add rotation 0-3
tftprinttest(sysfont.sysfont)
```

b) Bild aus Datei laden:

```
from st7735 import TFT, TFTColor
from machine import SPI, Pin

spi = SPI(1, baudrate=20000000, polarity=0, phase=0, sck=Pin(22), mosi=Pin(21), miso=None)
tft=TFT(spi,25,26,27)
tft.initr()
tft.rgb(True)
tft.fill(TFT.BLACK)

# Note BMP file needs to be in 24 bit format
f = open("python.bmp", "rb")
if f.read(2) == b"BM":  # header
    dummy = f.read(8)  # file size(4), creator bytes(4)
    offset = int.from_bytes(f.read(4), "little")
    hdrsize = int.from_bytes(f.read(4), "little")
    width = int.from_bytes(f.read(4), "little")
    height = int.from_bytes(f.read(4), "little")
    if int.from_bytes(f.read(2), "little") == 1:  # planes must be 1
        depth = int.from_bytes(f.read(2), "little")
        if (
            depth == 24 and int.from_bytes(f.read(4), "little") == 0
        ):  # compress method == uncompressed
            print("Image size:", width, "x", height)
            rowsize = (width * 3 + 3) & ~3
            if height < 0:
                height = -height
                flip = False
            else:
                flip = True
            w, h = width, height
            if w > 128:
                w = 128
            if h > 160:
                h = 160
            tft._setwindowloc((0, 0), (w - 1, h - 1))
            for row in range(h):
                if flip:
                    pos = offset + (height - 1 - row) * rowsize
                else:
                    pos = offset + row * rowsize
                if f.tell() != pos:
                    dummy = f.seek(pos)
                for col in range(w):
                    bgr = f.read(3)
                    tft._pushcolor(TFTColor(bgr[2], bgr[1], bgr[0]))
spi.deinit()
```

Microcontroller-Programmierung mit MicroPython (ESP32)

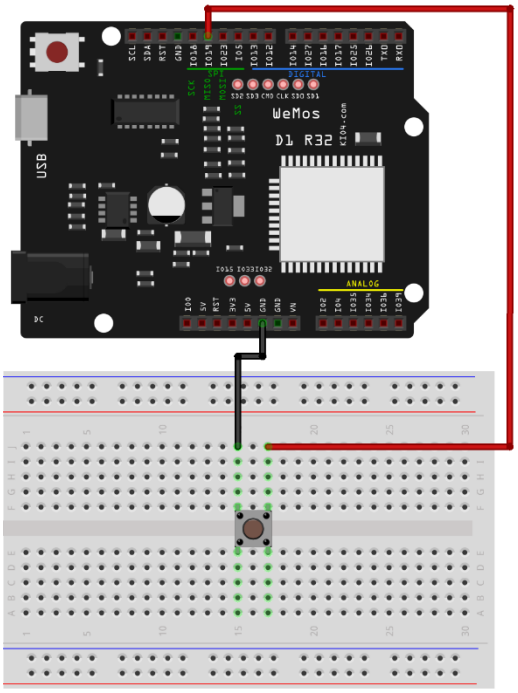
AB-26: ESP-NOW Funkübertragung

Info:

Das **ESP-NOW** Funkprotokoll ermöglicht eine **drahtlose Funkkommunikation** ohne Router. Damit können **Fernsteuerungen** realisiert oder **Sensoren ausgelesen** werden. Die Reichweite beträgt etwa 10-30 Meter. Es sind **keine zusätzlichen Bibliotheken** erforderlich. Im folgenden Beispiel wird über einen Taster eine LED ferngesteuert geschaltet. Die **MAC-Adresse** des Empfängers muss vorher über ein kleines Skript ausgelesen werden:

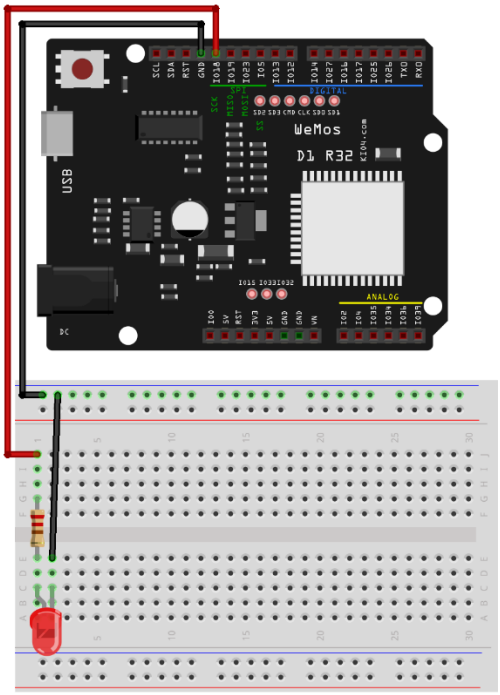
```
1  import machine as mc
2  print(mc.unique_id())
```

Sender:




```
1  from machine import Pin
2  from network import WLAN
3  from espnow import ESPNow
4  from time import sleep
5  button=Pin(19,Pin.IN, Pin.PULL_UP)
6
7  wlan = WLAN(WLAN.IF_STA)
8  wlan.active(True)
9
10 esp = ESPNow()
11 esp.active(True)
12
13 peer = b'\x88W!\x8eh\x14'
14 esp.add_peer(peer)
15 pressed=False
16
17 while True:
18     if button.value()==0 and not pressed:
19         esp.send(peer, "toggle")
20         pressed=True
21     if button.value()==1 and pressed:
22         pressed=False
23         sleep(0.1)
```

Empfänger:




```
1  from machine import Pin
2  from network import WLAN
3  from espnow import ESPNow
4  from time import sleep
5  led= Pin(18, Pin.OUT)
6
7  wlan = WLAN(WLAN.IF_STA)
8  wlan.active(True)
9
10 esp = ESPNow()
11 esp.active(True)
12
13 while True:
14     host, msg = esp.recv(100)
15     if msg == b'toggle':
16         led.toggle()
17     if msg is None:
18         sleep(0.01)
19         continue
```

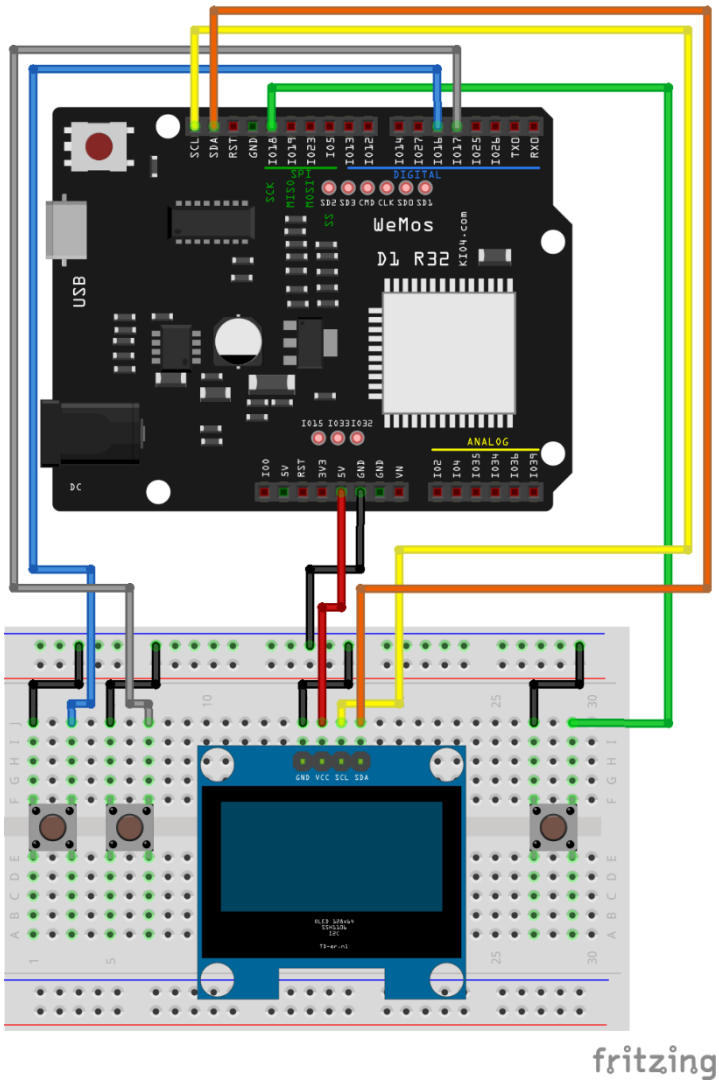
Der **Inhalt** der übermittelten Nachricht ist in diesem Beispiel “toggle“. Der Inhalt der Nachricht ist grundsätzlich **beliebig**!

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-27: Taschenrechner

Projekt:

Wir wollen **einen Taschenrechner** bauen. Dafür gehen wir **minimalistisch** vor und verwenden **nur 3 Taster und ein Display**. Mit zwei Tasten sollen die jeweiligen Zahlen ausgewählt werden können, die dritte Taste erfüllt die Funktion einer „Bestätigen“-Taste.



Aufgaben:

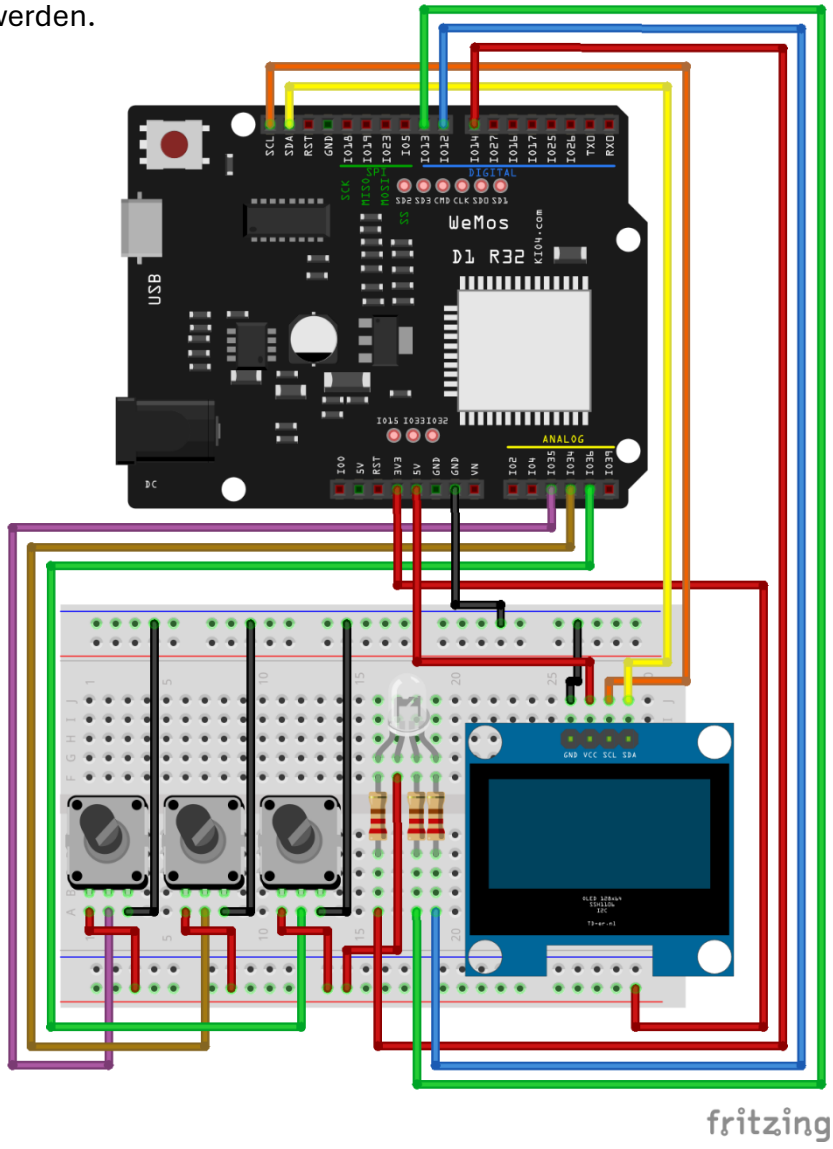
27.1) Erstelle einen Programmablaufplan.
27.2) Verwende den abgebildeten Aufbau.
27.3) Erstelle das Programm in MicroPython.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-28: RGB-Mixer

Projekt:

Bildschirmfarben sind am PC immer eine **Mischung aus Rot, Grün und Blau** (RGB). Wir wollen **LED-RGB-Mixer** bauen. Mit drei Potentiometern sollen die Farben einzeln zusammengemischt werden können, damit eine RBG-LED die Mischung anzeigen kann. Zusätzlich soll das Mischungsverhältnis in einem **Display** in Stufen zwischen 0-255 dargestellt werden.



Achtung: Es gibt **RBG-LEDs** mit **gemeinsamer Anode(+) oder Kathode(-)**. Die Verkabelung weicht bei einer gemeinsamen Kathode(-) von der Darstellung oben ab: Statt mit 3.3V ist der vierte Pin der LED dann mit GND verbunden.

Aufgaben:

28.1) Erstelle einen Programmablaufplan.
28.2) Verwende den abgebildeten Aufbau.
28.3) Erstelle das Programm in MicroPython.

04.2026 (Sch)

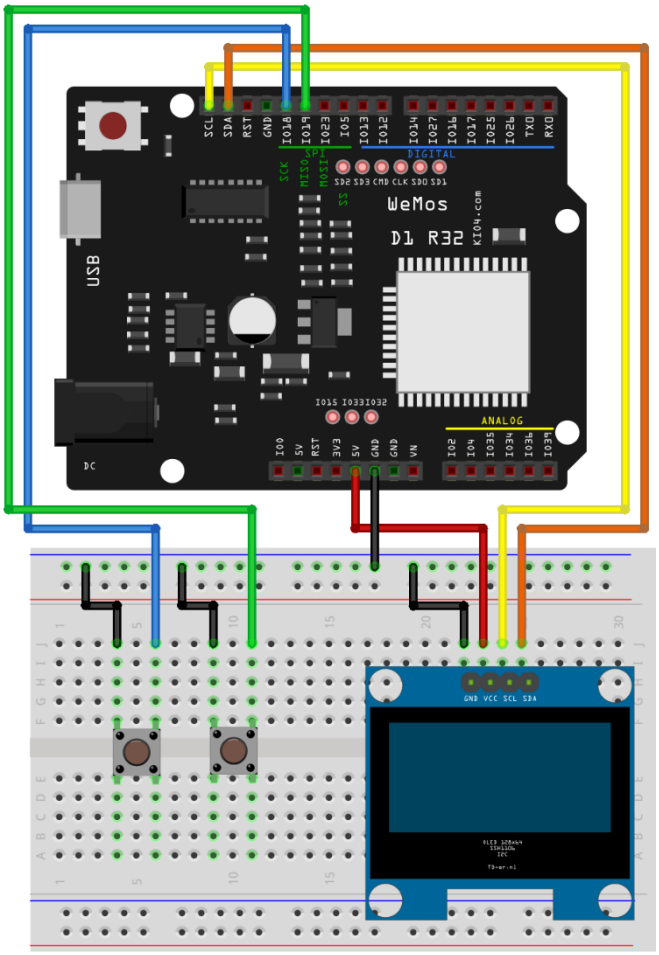
Microcontroller-Programmierung mit MicroPython (ESP32)

AB-29: Zufällige Kombination von Listen

Projekt:

Mit mehreren **Listen von Strings** (Texten) können **zufällige Sätze** generiert werden. Ebenso ist es auf diese Weise beispielsweise möglich **zufällig Dienste** im Haushalt zu verteilen. Nach folgendem Prinzip soll vorgegangen werden:

➔ **Name** (Liste1) + **is/was** + **Adjektiv** (Liste2). (**Peter is/was happy.**)



➔ Hier kann auch zunächst nur mit Kommandozeile gearbeitet werden.

Aufgaben:

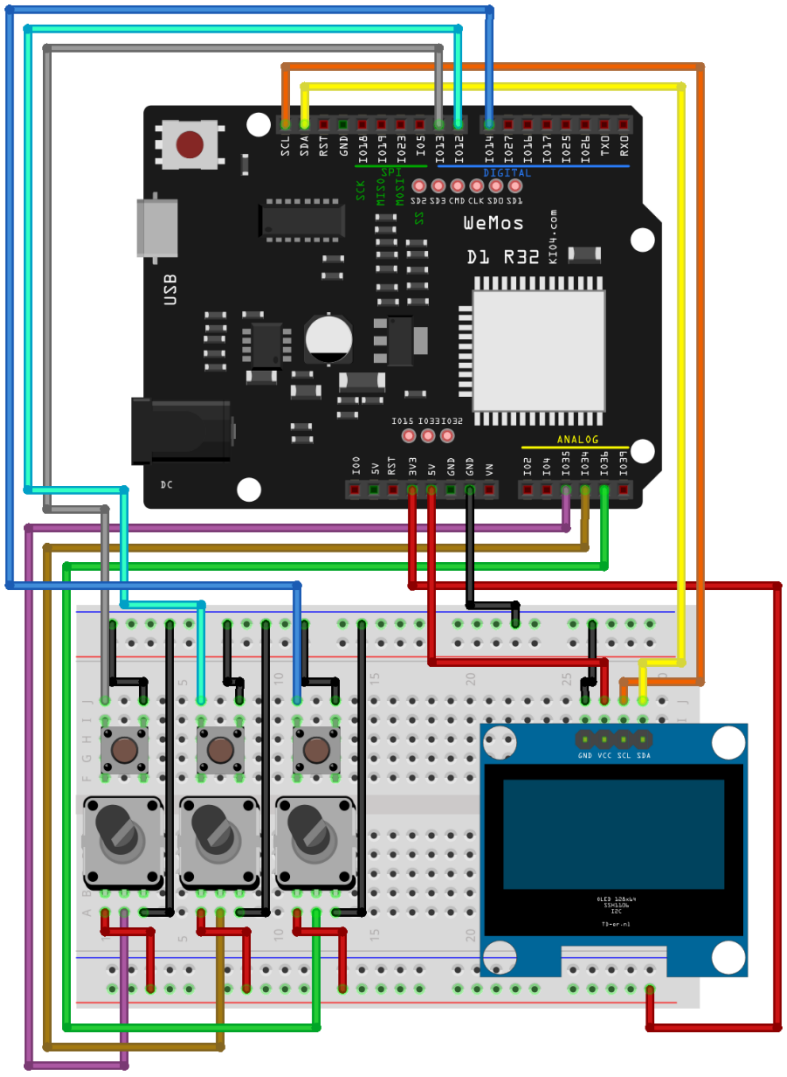
29.1) Erstelle einen Programmablaufplan.
29.2) Verwende den unten abgebildeten Aufbau.
29.3) Erstelle das Programm in MicroPython.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-30: Form-Demo mit Animation

Projekt:

Das **SH1106-OLED-Display** kann nicht nur **Formen** darstellen, sondern diese auch bewegen. Wir wollen in diesem Projekt mit einem **Quadrat** und einem **Kreis** eine kleine **Animation** ausprobieren, wie man sie von **klassischen Bildschirmschonern** kennt. Der Aufbau besteht aus **drei Potentiometern**, **drei Tastern** und einem **Display**. Mit dem ersten Taster wird zwischen Quadrat und Kreis umgeschaltet. Der zweite Taster schaltet zwischen dem Umriss der Form und einer gefüllten Form um. Der dritte Taster startet eine Animation, in der die ausgewählte Form immer wieder vom Displayrand abprallt. Die drei Potentiometer steuern die Größte der Form, die X-Koordinate und die Y-Koordinate. Die Potentiometer sind nur in Funktion, wenn die Animation gerade nicht läuft.



Aufgaben:

30.1) Erstelle einen Programmablaufplan.

30.2) Verwende den unten abgebildeten Aufbau.

30.3) Erstelle das Programm in MicroPython.

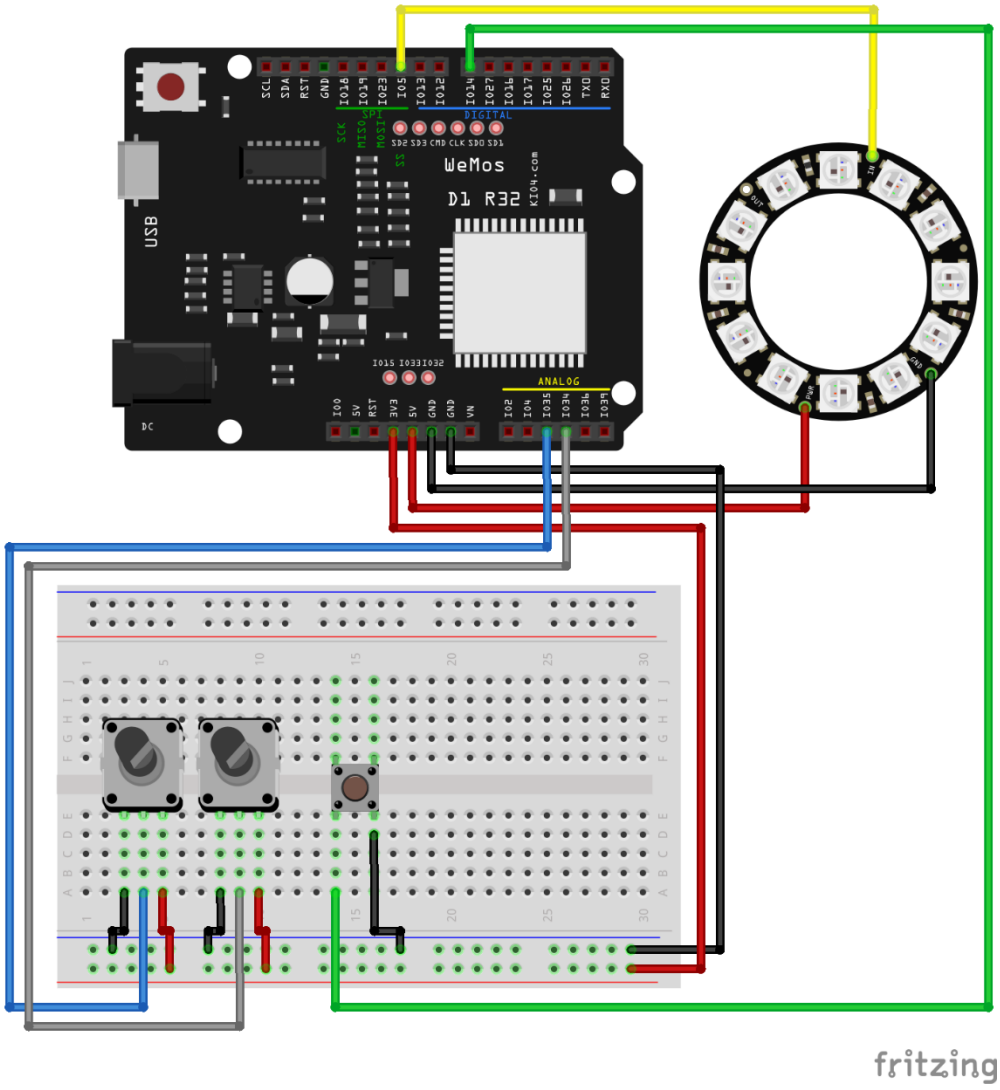
04.2026 (Sch)

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-31: NeoPixel LED-Ring

Info:

Die **NeoPixel Module** bestehen aus einer Reihe von **RGB-LEDs**, mit denen sich schöne **Effekte** realisieren lassen. Es gibt sie als **Streifen, Matrizen oder Ringe**. Für unser Beispiel nehmen wir einen **Ring mit 12 LEDs**, da man damit praktische Anwendungen bauen kann. Die Bibliothek für die NeoPixel-Module ist in MicroPython bereits enthalten.



Aufgaben:

31.1) Erstelle den obigen Aufbau.

31.2) Probiere das Demo-Programm auf der Folgeseite aus und analysiere es.

31.3) Erstelle ein Programm, das mit dem einem Potentiometer die Anzahl der leuchtenden LEDs steuert und mit dem anderen Potentiometer deren Helligkeit. Der Taster soll die verschiedenen Grundfarben durchschalten können.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-31: NeoPixel LED-Ring

Demo-Programm:

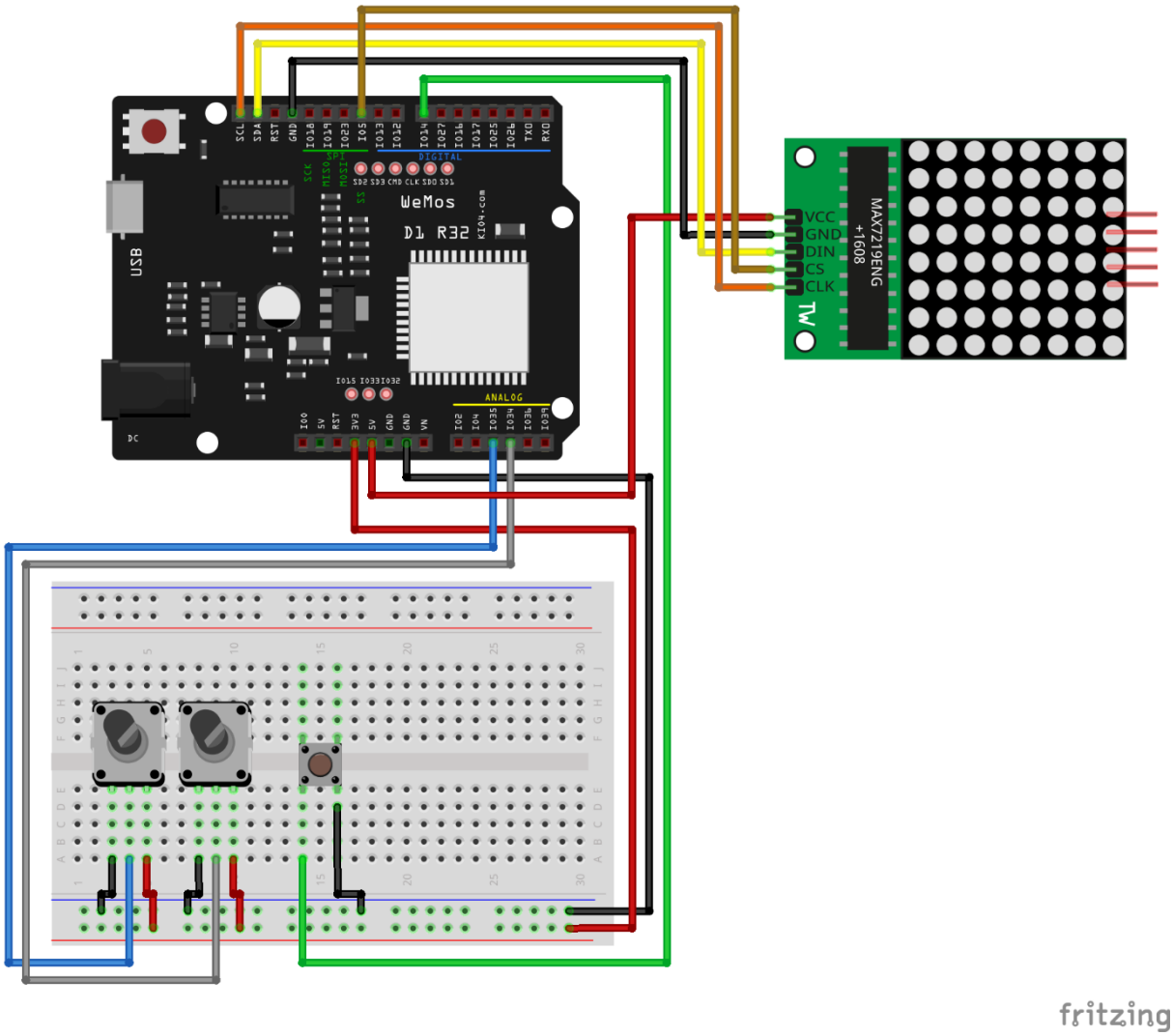
```
from machine import Pin
from neopixel import NeoPixel
import time

LED_PIN = 5
NUM_LEDS = 12
np = NeoPixel(Pin(LED_PIN), NUM_LEDS)

RED    = (255, 0, 0)
GREEN  = (0, 255, 0)
BLUE   = (0, 0, 255)
YELLOW = (255, 255, 0)
WHITE  = (255, 255, 255)
OFF    = (0, 0, 0)

def clear():
    for i in range(NUM_LEDS):
        np[i] = OFF
    np.write()

def fill_ring(color):
    for i in range(NUM_LEDS):
        np[i] = color
    np.write()

def running_light(color, wait):
    for i in range(NUM_LEDS):
        clear()
        np[i] = color
        np.write()
        time.sleep(wait)

def blink(color, count, wait):
    for i in range(count):
        fill_ring(color)
        time.sleep(wait)
        clear()
        time.sleep(wait)


while True:
    running_light(RED, 0.1)
    running_light(GREEN, 0.1)
    running_light(BLUE, 0.1)
    fill_ring(RED)
    time.sleep(1)
    fill_ring(GREEN)
    time.sleep(1)
    fill_ring(BLUE)
    time.sleep(1)
    fill_ring(YELLOW)
    time.sleep(1)
    blink(WHITE, 5, 0.2)
    clear()
    time.sleep(1)
```

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-32: Max7219 8x8-Pixel Matrix

Info:

Das **Max7219-LED-Matrix Modul** hat **8x8 LEDs** und eignet sich somit ideal für die Anzeige von **Laufschriften**. Noch viel besser geht das mit **4 dieser Displays** in Reihe. Dafür ist nur eine minimale Anpassung erforderlich. Für dieses Modul wird eine **Bibliothek** benötigt.



Aufgaben:

32.1) Erstelle den obigen Aufbau.

32.2) Probiere das Demo-Programm auf der Folgeseite aus und analysiere es.

32.3) Erstelle ein Programm, das mit dem einem Potentiometer die Geschwindigkeit der Laufschrift steuert und mit dem anderen Potentiometer deren Helligkeit. Der Taster soll die Laufschrift anhalten können.

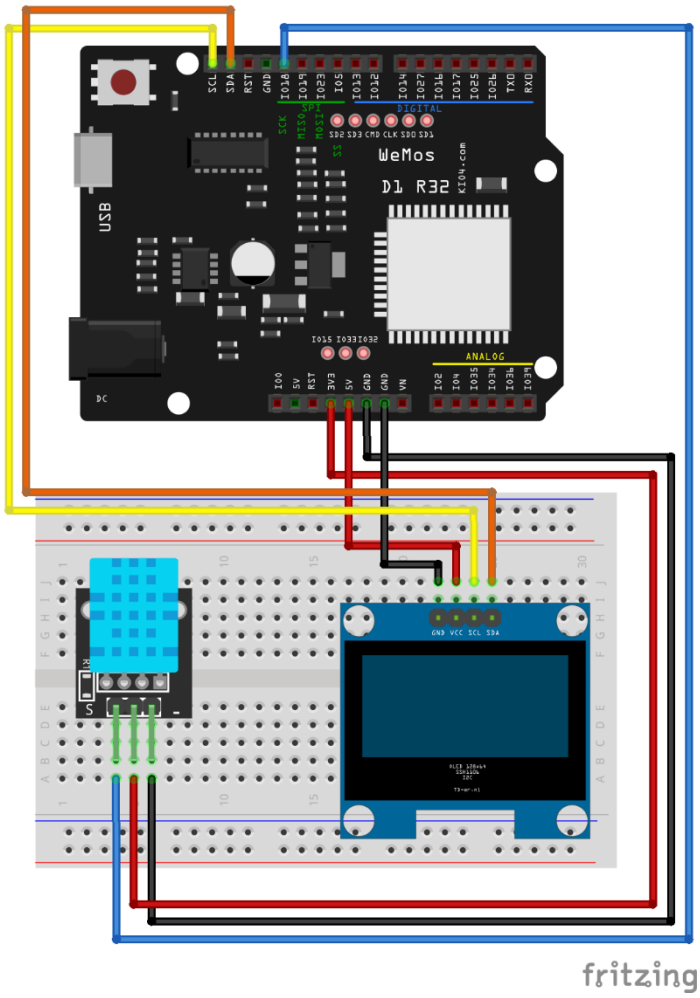
AB-32: Max7219 8x8-Pixel Matrix

Demo-Programm:

```
from machine import Pin, SPI
import max7219
import time

spi = SPI(
    1,
    baudrate=10000000,
    polarity=0,
    phase=0,
    sck=Pin(22),
    mosi=Pin(21) #sda
)

cs = Pin(5, Pin.OUT)

# Anzahl Module = 1
display = max7219.Matrix8x8(spi, cs, 1)

# Display aktivieren
display.brightness(5)
display.fill(0)
display.show()

def zeige_text(text, wartezeit=0.1):

    laenge = len(text) * 8
    for x in range(32, -laenge, -1):
        display.fill(0)
        display.text(text, x, 0, 1)
        display.show()
        time.sleep(wartezeit)

while True:
    zeige_text("HALLO WELT!!!")
    time.sleep(1)
```

Info:

Der **DHT11** ist ein häufig verwendeter **Temperatur- und Luftfeuchtigkeitssensor**. Er ist zwar **nicht besonders genau** (+-2 °C) und (+- 5%), aber dafür **günstig** und leicht zu verwenden. Es gilt zu beachten, dass es **zwei Bauformen** gibt, einmal mit PCB (Trägerplatine) und einmal ohne PCB. Bei der Version mit PCB ist der **nötige Widerstand von 10KΩ** schon integriert, bei der Version ohne PCB muss er **zwischen der Datenleitung und der Stromversorgung** (3,3V) ergänzt werden. Für die Funktion des DHT11 ist keine zusätzliche Bibliothek notwendig.




```python
import machine
import dht
import time

sensor = dht.DHT11(machine.Pin(18))

while True:
    try:
        sensor.measure()
        temp = sensor.temperature()
        hum = sensor.humidity()
        print('Temperature:'+str(temp)+'°C')
        print('Humidity:' +str(hum)+'%')
        print()

    except OSError as e:
        print('Sensor Error!')

    time.sleep(2)
```

try-except:

Mit der **try-except Struktur** kann ein möglicher **Fehler abgefangen** werden. Das **Programm stürzt dann nicht komplett ab**. So ist es auch möglich dem Nutzer zu melden, wenn beispielweise eine falsche Eingabe erfolgt ist.

Aufgaben:

33.1) Bringe den Sensor mit dem entsprechenden Skript in Funktion.

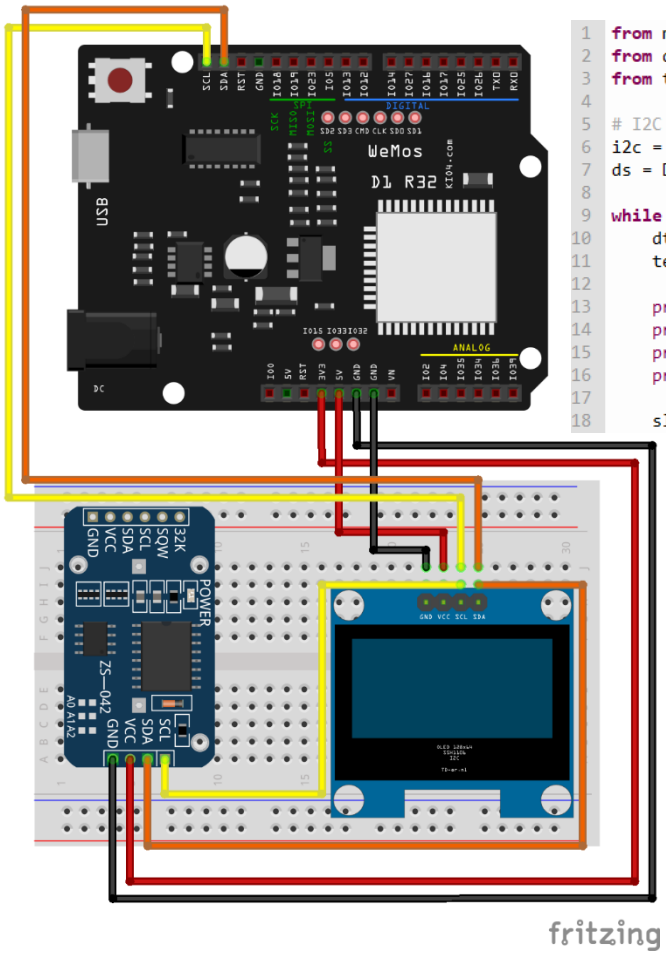
33.2) Ergänze ein SH1106-OLED-Display und lasse die Werte dort anzeigen.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-34: Echtzeituhr (DS3231)

Info:

Für Anwendungen ohne dauernde Verbindung zum PC oder WLAN kann es nötig sein ein **Echtzeit-Uhr-Modul** (RTC-Real-Time-Clock) zu verwenden. Dieses Modul enthält eine **Batterie** und behält auch ohne externe Stromversorgung die Zeit und das Datum. Das **DS3231-Modul** enthält **zusätzlich noch einen Temperaturmesser**. Um das Modul richtig einzustellen muss Zeile 12 im Skript **einmal ausgeführt werden** und **danach** mit dem Kommentarzeichen (#) **abgeschaltet** werden. Das Modul benötigt eine zusätzliche **externe Bibliothek**.




```
1  from machine import I2C, Pin
2  from ds3231 import DS3231
3  from time import sleep
4
5  # I2C + RTC Setup
6  i2c = I2C(0, scl=Pin(22), sda=Pin(21))
7  ds = DS3231(i2c)
8
9  while True:
10     dt = ds.get_time()
11     temp = ds.temperature()
12
13     print("Date: {:02d}/{:02d}/{:04d}".format(dt[1], dt[2], dt[0]))
14     print("Time: {:02d}:{:02d}:{:02d}".format(dt[3], dt[4], dt[5]))
15     print("Temp: {:.1f} °C".format(temp))
16     print()
17
18     sleep(1)
```

Aufgaben:

34.1) Bringe das Modul mit dem entsprechenden Skript in Funktion.

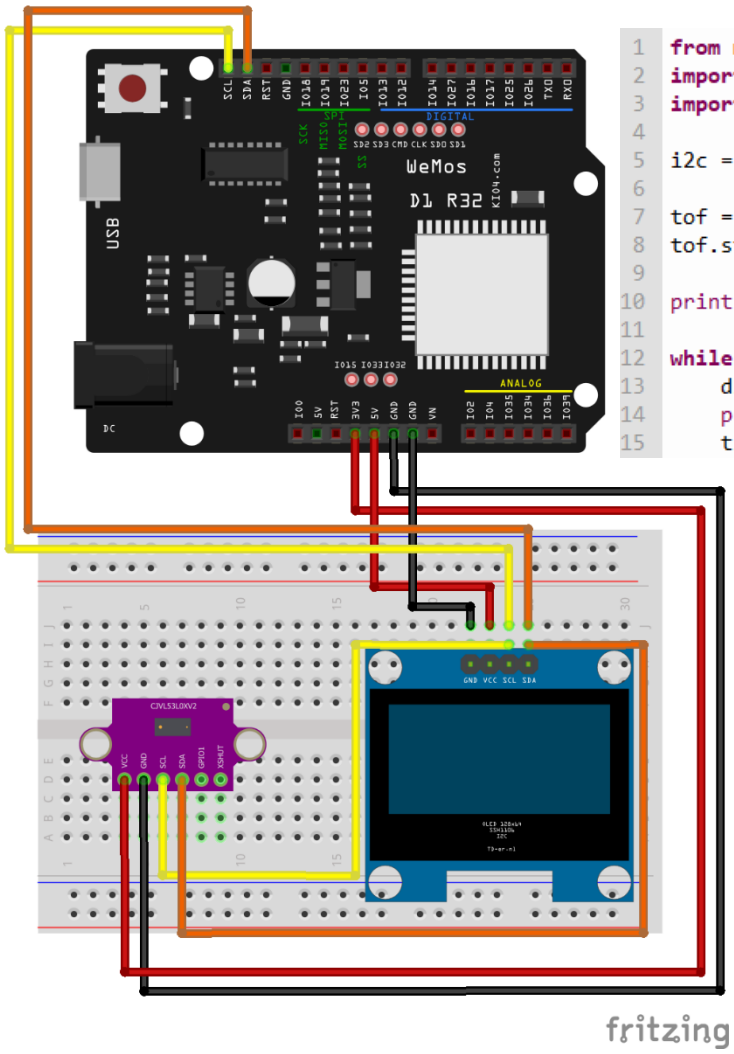
33.2) Ergänze ein SH1106-OLED-Display und lasse die Werte dort anzeigen.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-35: Entfernungsmessung mit Licht (VL5310X)

Info:

Für die Messung von Entfernungen stehen verschiedene Möglichkeiten zur Verfügung. Mit Ultraschallsensoren ist dies möglich, aber auch mit **Licht**. Der **VL5310X-Sensor** arbeitet nach dem Prinzip der **Lichtlaufzeit** (**TOF**=Time-of-Flight). Dieser Sensor **ist klein und leicht zu integrieren**. Er arbeitet bis zu **mittleren Entfernungen bis 2m** relativ genau. Für Licht- oder Schallmessungen sind **unterschiedliche Oberflächen** besser geeignet. Der VL5310X brauche eine zusätzliche externe Bibliothek.




```
from machine import I2C, Pin
import time
import vl53l0x

i2c = I2C(0, scl=Pin(22), sda=Pin(21), freq=400000)

tof = vl53l0x.VL53L0X(i2c)
tof.start()

print("Starting VL53L0X...")

while True:
    distance = tof.read()
    print("Distance:", distance, "mm")
    time.sleep(0.5)
```

Aufgaben:

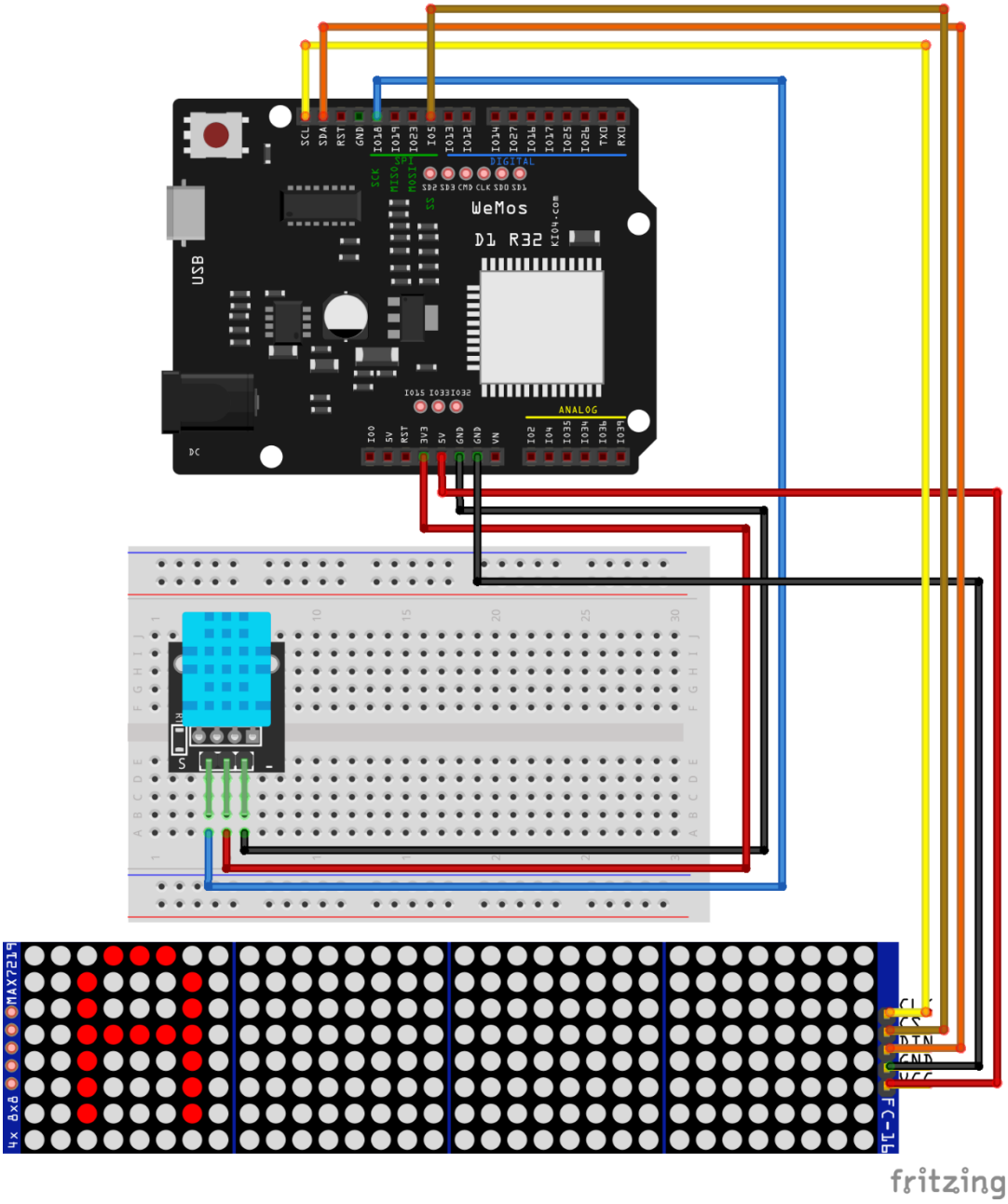
35.1) Bringe das Modul mit dem entsprechenden Skript in Funktion.
35.2) Ergänze ein SH1106-OLED-Display und lasse die Werte dort anzeigen.
35.3) Ergänze eine RBG-LED, die je nach Abstand unterschiedlich farbig leuchtet.

Microcontroller-Programmierung mit MicroPython (ESP32)

AB-36: Zeit- und Datumsanzeige mit animiertem Text

Projekt:

Wir wollen eine Laufschriftanzeige mit Zeit, Datum, Temperatur und Luftfeuchtigkeit bauen. Dafür nehmen wir ein MAX7219-Modul mit das aus 4 Einzelmodulen besteht. Es ist auch möglich diesen Aufbau noch zu erweitern, da sich diese Module in Reihe schalten lassen. Hier ist dann aber meiste die Stromversorgung nicht mehr ausreichend.



Aufgaben:

36.1) Erstelle einen Programmablaufplan.
36.2) Verwende den abgebildeten Aufbau.
36.3) Erstelle das Programm in MicroPython.

04.2026 (Sch)