

RainbowWarrior

Bauanleitung



Benötigte Materialien	3
Eine kurze Einführung ins Terminal	4
Getting Started	4
Abhängigkeiten	7
FreeRTOS	8
USB Port ermitteln	8
Compiler	11
Interpreter	12
Bauteile verbinden	14
LED Strip an ESP anschließen	14
Auf den ESP zugreifen	14
Micropython	15
Neopixel	15
Farben	17
Animationen Teil 1: Blinken	19
Schleifen	19
For-Schleife	19
While-Schleife	21
Links	24
Funktionen mit optionalen Parametern	24
Aufgabe - Blinken in zwei Farben	25
Ein erstes kleines Skript	26
Skripte auf dem ESP ausführen	29
Aufgabe: Schleifen erstellen	29
Dateisystem auf dem ESP	30
Terminalbefehle: Linux	36
Terminalbefehle Windows	37
Terminalbefehle MacOS	38

1. getting started

- terminal
- dependencies & getting the framework
- freeRTOS
- how to run a skript (repl and local)
- introduction micropython (let there be light)
- introduction MicroPython (make it blink)
- --- " --- (let it run)
- introduction in colors
- exercise

Seite 2 2. input / output

- sensor explanation (touch, piezo, ldr)
- control light by sensor (s) (turn off/on, toggle between ani/non-ani)
- add some noise

3. make it extraordinary!

-
-

X. attachments

Benötigte Materialien

Für einfache Pixel-Animationen:

- ESP32 Development Board
- LED Strip SK6812 RGBW, ca 170 mm lang (oder nach Belieben)
- Breadboard
- Jumperwires

Für das gesamte Projekt zusätzlich:

Neben der gezeigten Variante gibt es sehr viele verschiedene Möglichkeiten, LED auf Plexiglas in Szene zu setzen, siehe im Kapitel "Plexiglas" . Hierbei darf man auch gerne kreativ sein! Jedoch sollte bei abweichendem Design im Vorfeld überlegt werden, wie groß das Objekt insgesamt, wie die Plexiglasplatte und wie lang der Strip sein sollte.

- Plexiglasplatte - 170 x 170 x 10 mm (oder nach Belieben)
- Holzsockel, passend zur Platte. Hier ca 50 x 70 x 190 mm
- Metallknopf als Touch-Sensor (gerne auch Ersatzknöpfe von Hosen, schöne Nägel o.ä.)
- Piezo Speaker (passiv)
- Photoresistor
- Widerstand, 1k Ω (braun – schwarz – rot – gold)
- Litzen in 3 Farben (Schwarz, rot, gelb oder andere)

Desweiteren werden folgende Werkzeuge benötigt:

- Lötkolben, Lötzinn, Löttauglitze
- Holzsäge, Kreissäge oder die Möglichkeit, Holz zu verarbeiten
- Bohrmaschine incl. Bohrer in 5mm, ggf. auch 35mm
- Schleifpapier (am besten ein gröberes und ein recht feines)
- Heißklebepistole
- ggf. Öl, Lasur, oder Lack um das Holz zu versiegeln
- Computer inklusive USB Kabel.

Die Anleitung bezieht sich auf Rechner mit installiertem Linux Mint System. Soweit es mir möglich ist, werde ich versuchen, auch die nötigsten Schritte für Windows und MacOS bereitzustellen. Für Vollständigkeit kann ich leider keine Garantie geben, jedoch hilft hier oft eine kurze Internetrecherche zur betreffenden Fehlermeldung.

Getting Started

Eine kurze Einführung ins Terminal

Mit dem Terminal könnt ihr über Shellbefehle euren kompletten PC steuern. Hier könnt ihr Dateien anlegen, suchen und packen sowie System-, Netzwerk- oder Hardware-Befehle ausführen oder Benutzer verwalten.

Es sieht zwar sehr simpel aus, ist aber umso mächtiger im Vergleich zu den Optionen auf der grafischen Oberfläche. So können viele Aufgaben schneller und sauberer erledigt werden.

Manche Anwendungen bieten darüber hinaus auch bestimmte Kommandozeilen-Dienstprogramme, welche Funktionen bieten, die über das normale Programm nicht zu erreichen sind.

Terminal öffnen:

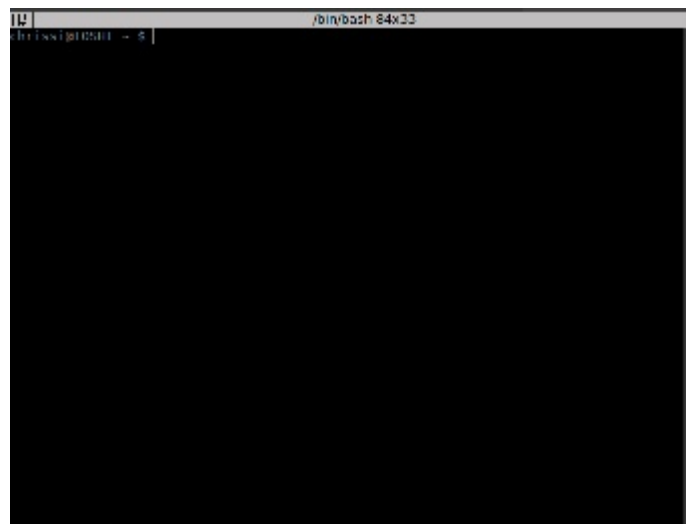
- Unter Linux: Super - bzw. Starttaste drücken und "Terminal" eingeben. Alternativ kann mit STRG+ALT+T per Tastatur das Terminal geöffnet werden
- Windows: Windows-Taste drücken, "cmd" eingeben, mit Enter bestätigen
- MacOS: CMD Leertaste drücken um Spotlight zu öffnen, "Terminal" eingeben.

Die hier vorgestellten Befehle beziehen sich hauptsächlich auf Linux. Viele der Befehle sind auch auf MacOS zu finden. Einige auch auf Windows - manche davon aber leicht abgewandelt.

Eine kurze Auflistung der wichtigsten Befehle pro Betriebssystem befindet sich im Anhang!

Auf den ersten Blick sieht das Terminal überhaupt nicht spektakulär aus.

Jede Zeile beginnt mit dem Nutzernamen, gefolgt von dem Computernamen (getrennt durch ein @) Anschließend wird der Ordnername angezeigt, in dem man sich aktuell befindet. Hier sieht man nur ein unscheinbares „~“, welches aussagt, dass man sich gerade im Home-Verzeichnis befindet. Abgeschlossen wird die Zeile mit einem „\$“, das signalisiert, dass man gerade als



Aktuelles Arbeitsverzeichnis:

pwd (=print working directory)

Falls du dir je nicht sicher sein solltest, ob du Dich im richtigen Ordner befindest, kannst du einfach **pwd** eingeben, was nichts weiter bedeutet als „gib aktuelles Arbeitsverzeichnis aus“

Ordnerinhalt auflisten

ls (=list)

Mit dem Befehl **ls** gibt man den Inhalt eines Ordners aus. Nur **ls**, ohne Optionen, listet den Inhalt des aktuellen Arbeitsverzeichnisses, mit

```
$ ls /home/$USER/Bilder
```

kann man, unabhängig vom derzeitigen Arbeitsverzeichnis, zum Beispiel den Inhalt des Ordners "Bilder" im home-Verzeichnis des aktuellen Benutzers (= \$USER) ausgeben.

Verzeichnis wechseln

cd (=change directory)

Um in ein Unterverzeichnis zu wechseln, gibst du einfach den Befehl **cd** ein, gefolgt durch den Namen des Unterverzeichnisses. Leerzeichen zwischen diesen beiden nicht vergessen!

Tipp für Schreibfaule: Du kannst auch einfach die ersten Buchstaben des Verzeichnisses eingeben, und mit einem Klick auf die TAB taste erledigt die Autovervollständigung den Rest der Schreibarbeit – es sei denn es gibt mehrere Verzeichnisse mit dieser Zeichenfolge am Anfang, oder es befindet sich ein Schreibfehler darin.

Möchtest Du in ein ganz anderes Verzeichnis wechseln, gibst du einfach **cd** ein, gefolgt vom kompletten Verzeichnispfad ein. Dieses Mal muss allerdings ein **/** vor den Pfad, um zu signalisieren, dass es sich nicht um ein Unterverzeichnis handelt

Um wieder ins Übergeordnete Verzeichnis zu wechseln, einfach **cd ..** eingeben.

Dateien kopieren

cp (=copy)

Um eine Datei zu kopieren, benutzt man den Befehl **cp** (= copy)
Hierzu wechselst Du mit **cd** in den Ordner, in dem sich die Datei befindet.
Anschließend gibst du ein:

```
$ cp dateiname.html /home/$USER/htmlFiles
```

Dieser Befehl besteht aus drei Teilen, getrennt durch ein Leerzeichen:

1. der Befehl **cp**, gibt an, dass eine Datei kopiert werden soll.
2. die Datei, die kopiert werden soll (hier als Beispiel: "dateiname.html")
3. der komplette Pfad zum Zielordner (/home/\$USER/htmlFiles).

Wenn du allerdings Ordner anstelle von Dateien kopieren möchtest, muss noch die Option **-r**, was für Rekursiv steht, hinter den Befehl **cd** gehängt werden. So wird der Ordner inklusive allen darin enthaltenen Dateien kopiert.

Das ganze kann dann so aussehen:

```
$ cp -r htmlFiles /home/chrisi
```

Achtung! Ein wichtiger Unterschied zwischen Terminal und der grafischen Benutzeroberfläche: Das Terminal meckert nur, wenn etwas nicht stimmt!

Es ist also völlig normal, dass nach Eingabe des Befehls und Bestätigung mit der Enter-Taste einfach wieder Name/Computername + Arbeitsverzeichnis angezeigt werden! Das signalisiert, dass alles geklappt hat.

Zudem gibt es **keine Sicherheitsabfrage** und **keinen Papierkorb!** gelöscht ist gelöscht. Verschieben ist verschieben. Solange du Lese- und Schreibrechte in den betreffenden Ordnern und den Dateien hast, wird es keine Sicherheitswarnung geben. Daher ist hier **große Vorsicht** geboten!

Dies waren erst einmal die wichtigsten Befehle, damit wir erst einmal im Terminal zurecht finden. Weitere werden an gegebener Stelle folgen und nochmals erklärt.

Abhängigkeiten

Bevor es mit der Installation des Betriebssystems und des Micropython Frameworks losgehen kann sind noch einige Pakete von Nöten, damit wir mit Python arbeiten können, der PC auch eine Verbindung zum Board aufbauen kann und so weiter. Außerdem müssen wir das Framework herunterladen.

Unter Linux hierzu einfach das Terminal öffnen und folgende Befehle nacheinander eingeben und jeweils mit Enter bestätigen (Achtung: der erste Befehl geht über zwei Zeilen!):

```
$ sudo apt-get install git wget make libncurses-dev flex bison  
gperf python python-serial python-pip rsync  
$ sudo pip install --upgrade pip  
$ sudo pip install esptool --upgrade
```

Unter Mac OS sind folgende Befehle nötig (ohne Gewähr! Falls etwas fehlen sollte → Google fragen):

```
$ sudo easy_install pip rsync  
$ sudo pip install pyserial
```

Um das MicroPython Framework auf das Board zu kriegen, hast Du zwei Möglichkeiten.
Entweder Du nutzt das Terminal und holst Dir eine Kopie davon auf deinen PC:

```
$ git clone https://github.com/loboris/  
MicroPython_ESP32_psRAM_LoBo.git
```

Oder aber Du gehst auf die Seite
„https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo“,
klickst auf den Button „clone or download“ und lädst dir manuell die Zip-Datei herunter.
Diese musst du allerdings erst noch entpacken.

freeRTOS

FreeRTOS (free Realtime Operating System) ist nichts weiter als ein Echtzeitbetriebssystem, zugeschnitten für Mikrocontroller. Es basiert auf einer Mikrokernarchitektur und wurde auf verschiedene Mikrocontroller portiert. Nur, was ist ein Kernel?

Ein Kernel, oder auch Betriebssystemkern ist - wie der Name schon sagt, der Kern, das Herzstück eines jeden Betriebssystems. Hier wird die Prozess- und Datenorganisation festgelegt, außerdem hat er direkten Zugriff auf die Hardware. Das bedeutet, der Kernel steuert, wann welcher Prozess den Prozessor belegen darf, welchen Speicherbereich dieser benutzen darf, wie die Hardware mit den Prozessen kommuniziert und so weiter.

Falls zusätzlich Lesebedarf besteht, hier einige zusätzliche Informationen:

<https://de.wikipedia.org/wiki/FreeRTOS>

<https://www.searchdatacenter.de/definition/Kernel>

USB Port ermitteln

Bevor wir mit dem Framework loslegen können, ist es wichtig zu wissen, auf welchem USB Port der Controller angesprochen wird. Jedes USB-Gerät wird vom PC an einen speziellen Port geknüpft, mit dem es vom System angesprochen wird.

Unter Linux, sehr wahrscheinlich auch unter MacOS, ist dies für den ESP32 meist ttyUSB0 oder ttyUSB1.

Sobald der Controller eingesteckt ist, kannst du dies auch einfach prüfen:

```
$ ls dev/ttyUSB*
```

ls bedeutet **list**, sprich es listet nun alle angeschlossenen Geräte, die mit ttyUSB anfangen. Der * signalisiert, dass nach dieser Zeichenfolge noch andere Zeichen folgen können.

Falls mit dem Befehl ls keine Ausgabe erfolgen sollte, kann man auch die **Kernelmeldungen auslesen**. Hierfür nutzt man - leider weder nur unter Linux und MacOS - den Befehl

dmesg (=display messages)

Um die Arbeit vom Kernel zu demonstrieren, kann man den Befehl einmal ausführen bevor die Hardware eingesteckt wurde,

```
[33423.787884] wlp4s0: associate with 5c:40:79:da:5c:35 (try 1/3)
[33423.793640] wlp4s0: RX AssocResp from 5c:40:79:da:5c:35 (capab=0x431 status=0
aid=3)
[33423.793840] wlp4s0: associated
[33423.794151] ath: EEPROM regdomain: 0x8114
[33423.794152] ath: EEPROM indicates we should expect a country code
[33423.794153] ath: doing EEPROM country->regdmn map search
[33423.794154] ath: country maps to regdmn code: 0x37
[33423.794155] ath: Country alpha2 being used: DE
[33423.794155] ath: Regpair used: 0x37
[33423.794157] ath: regdomain 0x8114 dynamically updated by country IE
[33423.800988] IPv6: ADDRCONF(NETDEV_CHANGE): wlp4s0: link becomes ready
[33494.695520] input: B8:69:C2:CD:92:4C as /devices/virtual/input/input20
chrissi@TOSH1 ~ $
```



```

[33423.888988] IPv6: ADDRCONF(NETDEV_CHANGE): wlan0: link becomes ready
[33494.695520] input: 08:69:C2:CD:92:4C as /devices/virtual/input/input20
[33744.871253] toshiba acpi: Unknown key e00
[34138.011877] input: 08:69:C2:CD:92:4C as /devices/virtual/input/input21
[34140.342886] Bluetooth: hci0: last event is not cmd complete (0x0ff)
[35463.627158] usb 3-1: new full-speed USB device number 2 using xhci_hcd
[35463.825010] usb 3-1: New USB device found, idVendor=10c4, idProduct=ea60
[35463.825018] usb 3-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[35463.825023] usb 3-1: Product: CP2102 USB to UART Bridge Controller
[35463.825028] usb 3-1: Manufacturer: Silicon Labs
[35463.825032] usb 3-1: SerialNumber: 0001
[35464.897575] usbcore: registered new interface driver usbserial_generic
[35464.897584] usbserial: USB Serial support registered for generic
[35464.900039] usbcore: registered new interface driver cp210x
[35464.900102] usbserial: USB Serial support registered for cp210x
[35464.900155] cp210x 3-1:1.0: cp210x converter detected
[35464.913310] usb 3-1: cp210x converter now attached to ttyUSB0
chrissi@TOSHIBA ~ %

```

und einmal, nachdem der Controller eingesteckt wurde. Der markierte Bereich zeigt alle Meldungen bezüglich dem zuletzt eingesteckten USB-Geräts. Zum Beispiel idVendor (= Hersteller-ID) und idProduct, verwendeter

Treiber (Product: CP2102 USB to UART Bridge Controller).

Für uns ist lediglich die letzte Zeile interessant. Hier steht "cp210x converter now attached to ttyUSB0", was bedeutet, dass unser Gerät am Port ttyUSB0 angeschlossen wurde.

Für Windows gibt es leider kein Terminal-Äquivalent zu dmesg. Jedoch kann man hier im Hardwaremanager (einfach in die Suche eingeben und mit Enter bestätigen) unter "USB-Controller" oder "Anschlüsse (COM & LTP)" bei eingestecktem Gerät sehen, unter welchem Port das Gerät zu erreichen ist. Unter Windows sind USB-Geräte an den sogenannten "COM-Port" gebunden.

Betriebssystem / Framework installieren

Wichtig! Die folgenden Informationen funktionieren auf Linux gleichermaßen wie auf MacOS.

Für Windows-Nutzer sind weitere, teils andere Schritte nötig, welche unter folgendem Link abrufbar sind:

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/winsetup

Um das Framework zu installieren kannst du entweder mit deinem FileBrowser zuerst in das heruntergeladene Verzeichnis, dann ins Unterverzeichnis „MicroPython_BUILD“ wechseln, von dort aus mittels Rechtsklick das Terminal öffnen.

Oder, falls du auf im Terminal geblieben bist, mit

```
$ cd MicroPythonESP32_psRAM_LoBo/MicroPython_BUILD
```

ins entsprechende Unterverzeichnis wechseln. Anschließend geht's an die Kernel-Settings.

Eigentlich ist in diesem Schritt nicht viel zu tun, jedoch müssen wir kurz die menuconfig aufrufen, um die sdkconfig zu erstellen.

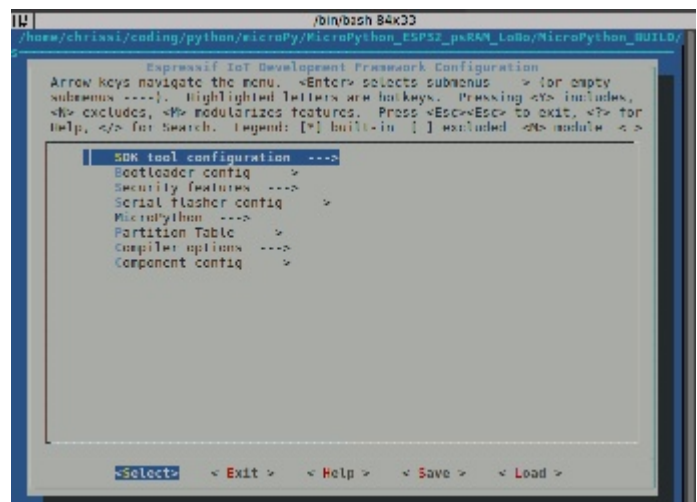
```
$ ./BUILD.sh menuconfig
```

Mit diesem Befehl öffnest du die menuconfig.

Hier navigiert man ausschließlich mit den Cursor-Tasten. Oben/Unten um die verschiedenen Menüpunkte auszuwählen, Rechts/Links um in der Menüliste unten auszuwählen.

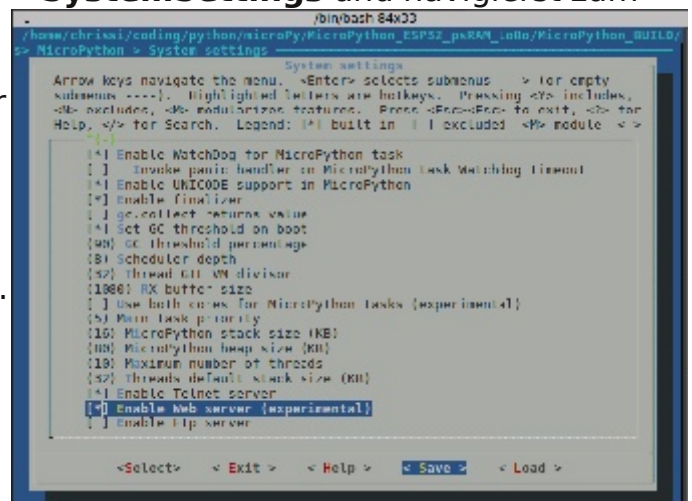
Gerne kannst Du dich etwas umschauen und dir Gedanken machen, aber, auch ich weiß bei vielem hier noch nicht, was es zu bedeuten hat, bzw. welche Auswirkungen das hat.

Mit „select“ kannst du das Untermenü, bzw. in die Einstellungen des ausgewählten Menüpunktes gelangen. Wenn du einmal nach rechts drückst, sodass „exit“ markiert ist, anschließend auf Enter, gelangst Du wieder ins übergeordnete Menü.



Wir müssen hier allerdings die Webserver-Funktion aktivieren. Dazu wechselst Du in **Micropython** → **SystemSettings** und navigierst zum Menüpunkt „**Enable WebServer**“.

Wenn dieser noch nicht mit einem Stern markiert ist, einmal auf „y“ für „yes“ drücken um diesen zu aktivieren. Falls Du versehentlich etwas aktiviert haben solltest, das Du nicht wolltest, kannst Du dieses mit „n“ für „no“ wieder deaktivieren. Zum Beispiel kann man hier auf den Telnet- sowie den FTP-Server verzichten. Es sei denn, Du spielst mit dem Gedanken, einen Fileserver zu erstellen oder etwa Kommandos über das Internet, sprich via WLAN an den Controller zu schicken. Dann könnte Telnet interessant werden.



Das Webserver-Modul benötigt das Modul namens „**Websockets**“. Dieses befindet sich unter **Micropython** → **Modules**.

Also mit dem Cursor einmal nach rechts, sodass „Exit“ markiert wird, mit Enter bestätigen, einmal nach unten um auf Modules zu gelangen und ebenso mit Enter bestätigen. Hier einmal „**use Websockets**“ mit **y** auswählen, falls dieses noch nicht aktiviert war.

Außerdem müssen wir dafür sorgen, dass wir den vollen Speicherbereich nutzen können. Dies kann man unter **MicroPython** → **System settings** → **MicroPython heap size (KB)** einstellen. Hier sollte der Wert **96** stehen.

Anschließend in der unteren Menüleiste mit Pfeil-Rechts Taste zu **save** wechseln und mit Enter bestätigen. Du wirst noch einmal gefragt, unter welchem Dateinamen die sdkconfig gespeichert werden soll, hier bitte auch Seite 10

nur mit Enter bestätigen Anschließend 3x **exit** wählen um die menuconfig zu beenden.

Falls je etwas bei der Ausführung schief gehen sollte oder Du Dir nicht sicher sein solltest, kannst Du ebenso die Datei „sdkconfig“ aus dem Ordner „instruction“ nehmen und in „MicroPython_ESP32_psRAM_LoBoNeu/MicroPython_BUILD“ einfügen.

Anschließend ebenso die Menuconfig öffnen, mit dem Cursor nach rechts auf **load**, mit Enter bestätigen und die Datei „sdkconfig“ auswählen. Nun – wie oben beschrieben – speichern und beenden.

Jetzt geht's an die binary!

Hört sich aber schlimmer an als es ist :)

```
$ .BUILD.sh clean
$ ./BUILD.sh
```

... eingeben ... warten ... erledigt!

Betriebssystem / Framework installieren

Du möchtest wissen, was hier passiert?

Wichtig! Die folgenden Informationen funktionieren auf Linux gleichermaßen wie auf MacOS.

Ein Build ist der Vorgang in der Softwareentwicklung, bei dem eine Anwendung, bestehend aus Binärcode (binary) aus dem Quellcode erzeugt wird. Diesen Prozess nennt man "kompilieren". Kompiliert wird mit einem sogenannten Compiler.

Compiler

Compiler sind spezielle Übersetzer, die Programmcode aus problemorientierten Programmiersprachen, sogenannten Hochsprachen, in ausführbaren Maschinencode, also die Sprache, die die Prozessoren verstehen, übersetzen.

Wenn du dich einmal in der Ordnerstruktur des Micropython Frameworks umsiehst, findest du zum Beispiel im Ordner

MicroPython_ESP32_psRAM_LoBoNeu/MicroPython_BUILD/components/micropython/esp32 einige Dateien mit der Endung *.c oder *.h. Dies sind Module, geschrieben in der Programmiersprache C, und die zugehörigen Header-Files.

Diese sind für uns Menschen noch einigermaßen lesbar, der Prozessor, der dies allerdings ausführen muss, kann noch gar nichts damit anfangen. Dieser "versteht" nämlich nur den sogenannten Maschinencode, bestehend aus einer Folge von Bytes, die sowohl Befehle als auch Daten repräsentieren. Da dieser Code für den Menschen schwer lesbar ist, werden in der Assemblersprache (die Sprache, die der Maschiensprache noch am nächsten kommt) die Befehle durch besser verständliche Abkürzungen, sogenannte Mnemonics, dargestellt.

Der Compiler scannt also während des kompiliervorgangs den Code und übersetzt ihn - so ressourcenschonend es geht - in Maschinensprache. Die dadurch entstandene binary lässt sich dann vom Computer ausführen.

Nachfolgend ein kleines Beispiel, wie es für den Prozessor aussehen kann, wenn man der Variablen "a" einen Wert zuweist oder wenn man zwei Werte, die in Variablen gespeichert sind, miteinander addiert.

Wichtig zu wissen ist hierbei, die Variablen werden in sogenannten Registern gespeichert. Dies sind Speicherbereiche innerhalb des Prozessors, die direkt mit dem Rechenkern verbunden sind.

Maschinencode (Hexadezimal)	Assemblercode	C-Code	Erklärung
C7 45 FC 02	mov DWORD PTR [rbp-4], 2	int a = 2;	Setze Variable a, die durch Register RBP adressiert wird, auf den Wert 2.
8B 45 F8	mov eax, DWORD PTR [rbp-8]	int c = a + b;	Setze Register EAX auf den Wert von Variable b.
8B 55 FC	mov edx, DWORD PTR [rbp-4]		Setze Register EDX auf den Wert von Variable a.
01 D0	add eax, edx		Addiere den Wert von EDX zum Wert von EAX.
89 45 F4	mov DWORD PTR [rbp-12], eax		Setze Variable c, die durch RBP adressiert wird, auf den Wert von EAX.

Falls du mehr darüber erfahren möchtest, hier einige Links:

<https://de.wikipedia.org/wiki/Compiler>

<https://de.wikipedia.org/wiki/Maschinensprache>

[https://de.wikipedia.org/wiki/Register_\(Computer\)](https://de.wikipedia.org/wiki/Register_(Computer))

Interpreter

Im Gegensatz zu einem Compiler, der vor Ausführung des Programms den vorhandenen Code in Maschinensprache übersetzt, macht dies ein sogenannter Interpreter erst während der Laufzeit.

Vorteil hiervon ist, dass ein zum Beispiel in Python geschriebenes Programm

auf verschiedenen Systemen mit verschiedenen Prozessorarchitekturen ausführbar ist.

Framework "flashen"

Genug des Ausflugs in die Theorie, nun wird es Zeit, den ESP auszupacken und anzuschließen!

Also. Raus aus der Verpackung, ran ans USB-Kabel und einstecken!

Jedes USB-Gerät wird vom PC an einen speziellen Port geknüpft, mit dem es vom System angesprochen wird.

Unter Linux ist dies meist ttyUSB0 oder ttyUSB1.

Sobald der Controller eingesteckt ist, kannst du dies auch einfach prüfen:

```
$ ls dev/ttyUSB*
```

ls bedeutet list, sprich es listet nun alle angeschlossenen Geräte, die mit ttyUSB anfangen. Der * signalisiert, dass nach dieser Zeichenfolge noch andere Zeichen folgen

Bevor wir die Binary drauf spielen können, löschen wir vorsichtshalber alles:

```
$ esptool.py --port /dev/ttyUSB0 erase_flash
```

oder:

```
$ ./BUILD.sh erase
```

Und spielen dann die binary auf den Controller:

```
$ ./BUILD.sh flash
```

Bauteile verbinden

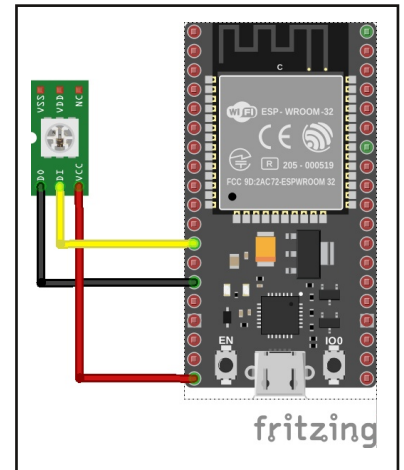
Neugierig? Ersten Test??

LED Strip an ESP32 anschließen

Also.

Du hast den LED-Strip, dieser hat an jedem Ende drei Anschlüsse, einen für 5V (rot), einen Data-Pin (meistens gelb) und einen für Ground, oder GND (schwarz oder weiß). Schließe hier das rote und das schwarze (oder weiße) Kabel an eine externe Stromquelle (falls es nur wenige LED sind, kannst du auch den 5V Ausgang und GND vom ESP nutzen. Allerdings kann dies bei Überspannung deinen USB-Port am Rechner zumindest vorübergehend außer Gefecht setzen). Wichtig: um den Stromkreis zu schließen muss der GND der externen Stromquelle mit dem GND des ESP verbunden sein. Anschließend verbindest Du den Data-Pin des LED-Strips (DIN oder DI) mit Pin 14 am ESP.

Der Controller wird - falls noch nicht geschehen - per USB-Kabel mit dem Computer verbunden.



Auf den ESP zugreifen

Hierfür verwenden wir das Tool "**rshell**" (= remote shell), das es uns ermöglicht, eine Verbindung zum Controller herzustellen um Dateien und Ordner hochzuladen, auszulesen, zu verschieben oder löschen, und um REPL zu starten.

Rshell benötigt einige Angaben, wie zum Beispiel die Puffergröße (--buffer-size=30) und den Port (-p /dev/ttyUSB0), wobei hier der vorher ermittelte Port (/dev/ttyUSB0) anzugeben ist. Bitte hier den richtigen Port für dein System verwenden!

repl (= Read-eval-print loop) wiederum ist dazu da, um per Kommandozeile Befehle auf dem Board auszuführen, kleine Funktionen zu erstellen und diese, oder andere vorhandene Skripte auszuführen.

```
$ rshell --buffer-size=30 -p /dev/ttyUSB0 -a -e nano
$ repl
```



```
/bin/bash
/bin/bash 80x24
chrissi@TOSH1 ~ $ rshell --buffer-size=30 -p /dev/ttyUSB0 -a -e nano
Connecting to /dev/ttyUSB0 ...
Welcome to rshell. Use Control-D to exit.
/home/chrissi> repl
Entering REPL. Use Control-X to exit.
MicroPython ESP32 LoBo v3.2.21 - 2018-08-27 on ESP32 board with ESP32
Type "help()" for more information.
>>>
>>> []
```

Ab hier verändert sich die Darstellung der Zeilen! im Python-REPL werden die Zeilenanfänge mit den Zeichen ">>>" signalisiert. So weiß man sofort, dass man sich nicht in der systemeigenen Shell,

sondern in der Python-Shell befindet, hier also alle Befehle in Python geschrieben sind.

Micropython

Module

Modulare Programmierung ist eine Software-Design-Technik, die auf dem allgemeinen Prinzip des modularen Designs beruht. Unter modularem Design versteht man, dass man ein komplexes System in kleinere selbständige Einheiten oder Komponenten zerlegt. Diese Komponenten bezeichnet man üblicherweise als Module. Ein Modul kann unabhängig vom Gesamtsystem erzeugt und separat getestet werden. In den meisten Fällen kann man ein Modul auch in anderen Systemen verwenden.

Modul: machine

Um in Micropython unter anderem die Pins anzusteuern benötigt man das Modul "machine". Eigentlich handelt es sich bei "machine" um ein Modul, das sogar weitere Module, beziehungsweise sogenannte "Klassen" beinhaltet.

mit dem machine-Modul kann man zum Beispiel die Prozessorgeschwindigkeit in MHz auslesen oder einstellen (`machine.freq([speed in MHz])`) oder den Controller neu starten (`machine.reset()`).

Außerdem beinhaltet es das für uns wichtige Modul, bzw. die Klasse "Neopixel", mit dem wir den Strip konfigurieren und ansteuern, also zum leuchten bringen können.

Neopixel

Bevor wir den Strip leuchten lassen können, müssen wir also zuerst das Modul namens machine einbinden und damit den Pin definieren, an den wir den LED-Strip angeschlossen haben. Anschließend muss ein Objekt von "Neopixel" erzeugt werden. Im Fachjargon nennt man das "eine Instanz einer Klasse" erstellen. Aber da das Prinzip der Klassen für Anfänger oft schwer nachvollziehbar ist, lassen wir dieses Detail einmal aus. Gerne darf aber bei

Interesse nachgelesen werden. Das deutschsprachige Python-Tutorial bietet dazu eine einfach geschriebene und detaillierte Erklärung:
https://www.python-kurs.eu/python3_klassen.php.

mit dem **"import"** - Befehl sind wir in der Lage, Module einzubinden.

machine.Pin([pin_number], [input / output]) definiert den Pin, den wir in [pin_number] angeben, sowie ob hier Signale empfangen (input) oder gesendet (output) werden. **machine.Pin.OUT** gibt zum Beispiel an, dass hier Signale vom Controller aus (an den LED-Strip) gesendet werden. Gib nun folgende Befehle nacheinander in REPL ein und bestätige sie jeweils mit Enter:

```
>>> import machine
>>> led_pin = machine.Pin(14, machine.Pin.OUT)
```

Um ein Objekt "Neopixel" zu erstellen, müssen ebenso einige Angaben gemacht werden. So muss der Pin angegeben werden, an dem der Strip angeschlossen ist, sowie die Anzahl der Pixel auf den Strip (in meinem Fall sind das 14 Pixel). Auch der verwendete LED-Typ sollte angegeben werden, sofern du auch den Strip vom Typ "SK6812". Wenn man dies auslässt ist als Standard-Typ TYPE_RGB angegeben, also LEDs mit jeweils drei Farben: rot, blau grün. Der SK6812 besteht sogenannten "RGBW" LED, sprich ein Pixel besteht aus vier Farben. Neben RGB auch weiß, so sind die LEDs auch einzeln dimmbar. So setzt sich dieser Schritt nun zusammen:

machine.Neopixel(pin, num_pixels[, type])

Den Pin haben wir ja bereits im letzten Schritt in einer Variablen namens "led_pin" gespeichert. Es bietet sich an, auch die Pixelanzahl (hier: 14) in einer Variablen zu speichern. Warum dies sinnvoll ist, wirst du in einem späteren Kapitel erfahren.

Gib nun folgende Befehle ein, um die Anzahl der Pixel in der Variablen "num_pixels" zu speichern und eine Instanz der Klasse Neopixel in der Variablen "strip" zu speichern:

```
>>> num_pixels = 14
>>> strip = machine.Neopixel(led_pin, num_pixels,
machine.Neopixel.TYPE_RGBW)
```

Selbstverständlich kann man dies auch auf zwei Zeilen, bzw. zwei Befehle komprimieren, allerdings wirst du gleich sehen, dass dies vergleichsweise sehr unübersichtlich ist:

```
>>> import machine
>>> strip = machine.Neopixel(machine.Pin(14, machine.Pin.OUT),
11, machine.Neopixel.TYPE_RGBW)
```


LEDs leuchten lassen

`np.set(pos, color [, white, num, update])` ist dafür zuständig, die Pixels leuchten zu lassen. Auch hier sind zwei Angaben zwingend erforderlich, alle anderen (in der eckigen Klammer) optional.

- **pos** gibt an, welcher Pixel genau gesetzt werden soll
 - **color** gibt - wie der Name schon sagt - an, in welcher Farbe der Pixel leuchten soll. Hier kann man entweder vordefinierte Farben verwenden (wie im folgenden Beispiel), oder Farbwerte als Hexadezimalwert (wird später noch erklärt)
 - **white** (optional) gibt - bei RGBW-Strips - an, wie hell die Farbe weiß in diesem Pixel leuchten soll. Wird hier nichts angegeben, wird als Standardwert 0 gesetzt. Hier sind Werte von 0-255 möglich. Je höher, desto mehr weiß wird in die Farbe gemischt.
 - **num** (optional) definiert, wieviele Pixel in diesem Schritt gesetzt werden sollen, ausgehend vom Pixel, der in pos angegeben wurde. Standardwert ist hier 1, also wird standardgemäß nur der eine Pixel gesteuert, der in num angegeben wurde.
 - **update** (optional) bestimmt, ob der oder die Pixel sofort in der jeweiligen Farbe angezeigt werden soll, bzw sollen (was auch Standardwert ist), oder ob mit der Anzeige gewartet werden soll. Falls abgewartet werden sollte, ist `update=False` anzugeben.
- Um dann allerdings die Pixel in der vorher gesetzten Farbe leuchten zu lassen, ist zusätzlich noch der Befehl `strip.show()` nötig.

```
>>> strip.set(0, strip.NAVY)
```

lässt also den ersten Pixel des Strips in der Farbe "Navy" leuchten, während

```
>>> strip.set(0, strip.NAVY, num=num_pixels)
```

alle Pixel in der Farbe "Navy" leuchten lässt

Farben

Wie bereits erwähnt kann man die Farbe aus vordefinierten Farben, oder als Hexadezimalwert angeben.

Um herauszufinden, welche Farben verfügbar sind, gibt es einen kleinen Trick:

Gib einfach "strip." in REPL ein und drücke einmal auf TAB.

So erscheint eine Auflistung aller vordefinierten Werte und sogar auch der vorhandenen Funktionen. Die vordefinierten Werte werden in Großbuchstaben angezeigt. So sind folgende Farbwerte bereits definiert: BLACK, WHITE, RED, LIME, BLUE, YELLOW, CYAN, MAGENTA, SILVER, GRAY, MAROON, OLIVE, GREEN, PURPLE, TEAL, NAVY.

```

>>>
>>> strip.
__class__      BLACK          BLUE          CYAN
GRAY           GREEN          HSBtoRGB      HSBtoRGBint
LIME           MAGENTA        MAROON        NAVY
OLIVE          PURPLE         RED           RGBtoHSB
SILVER         TEAL           TYPE_RGB      TYPE_RGBW
WHITE          YELLOW        brightness    clear
color_order    deinit         get           info
rainbow        set           setHSB        setHSBint
setWhite       show          timings
>>> strip.

```

Alternativ hat man die Möglichkeit, die Farbe anhand des Hexadezimalwerts anzugeben.

RGB-Farben werden angegeben durch einen Wert für rot (=r), einen für grün (=g) und einen für blau (=b). Dieser Wert darf zwischen 0 und 255 liegen. Für die Farbe rot muss also r=255, sprich höchster Wert, g=0, b=0. Anders geschrieben: (255, 0, 0).

Dies ist der sogenannte RGB-Wert als Tupel dargestellt.

Der Hexadezimalwert wäre für rot #FF0000, bzw. 0xFF0000, wobei die Zeichen "#" oder "0x" nur vorangestellt werden, um zu zeigen, dass es sich hierbei um einen Hexadezimalwert handelt.

Wandeln wir die Dezimalzahl "255" in eine Hexadezimalzahl erhalten wir "FF". Der Hexadezimalwert einer Farbe ist also nichts weiter, als die jeweiligen Farbwerte für r, g und b, konvertiert in eine Hexadezimalzahl, aneinandergereiht.

Die Farbe "aquamarin" hat zum Beispiel folgenden RGB-Wert: (127,255,212). Konvertieren wir die einzelnen Werte in Hexadezimalwerte erhalten wir 0x7FFFD4 (also 127 = 0x7F, 255 = 0xFF, 212 = 0xD4. Konvertieren wir wiederum den kompletten Hexadezimalwert in einen Dezimalwert erhalten wir 8388564. Beide Werte sind hier als Farbangabe zulässig.

Möchten wir jedoch ein RGB-Tupel, also hier (127, 255, 212) als Angabe verwenden, müssen wir dieses erst in den entsprechenden Hexadezimalwert konvertieren.

Wollen wir die Farbe Schwarz, was im additiven Farbsystem, also dem Farbsystem unseres sichtbaren Lichts soviel wie "kein Licht" bedeutet, müssen wir einfach alle Farben auf 0 setzen. Dies geht indem man für color den Wert "0x00" angibt. also 0.

Animationen Teil 1: Blinken

Bevor wir die LEDs blinken lassen können, müssen wir uns überlegen, was hier eigentlich genau geschieht:

Die LEDs werden alle zur gleichen Zeit angeschalten, leuchten eine Weile, anschließend werden sie ausgeschalten. Nach einer Weile gehen sie wieder an, und so weiter.

Wenn ich die LEDs also zweimal blinken lassen möchte, könnte ich einfach schreiben:

mach an – warte – mach aus – warte – mach an – warte – mach aus.

Nur. Das wäre ja ganz schön doof. Wollten wir die LEDs auf diese Weise 1000x blinken lassen, hätten wir mittels copy + paste einiges zu tun! Außerdem würde das Programm unübersichtlich werden!

Viel besser wäre es doch wenn wir sagen könnten:

mach bitte 5x:
mach an – warte – mach aus – warte.

Schleifen

Aus diesem Grund gibt es Schleifen.

Mit Schleifen kann man zum Beispiel festlegen, dass bestimmte Codezeilen genau 5, 10, 1000 Mal ausgeführt werden sollen (**for-Schleife**). Außerdem kann man auch sagen, dass ein Code-Block, wie bei uns das Blinken, nur ausgeführt werden soll, während es draußen dunkel ist (**while-Schleife**).

Zum Beispiel: während es draußen dunkel ist: blinke, checke obs immernoch dunkel ist. wenn dunkel: nochmal das ganze! Wenn hell, abbrechen.

Auch könnte man mit einer while-Schleife sagen: mache das einfach immer wieder! Eine sogenannte Endlos-Schleife.

For-Schleife

Um zu sagen: führe diesen Codeblock 5x aus schreibt man in Python:

```
>>> for i in range(5):
```

range(5) ist eine Funktion, die nacheinander alle ganzen Zahlen von 0 bis 4(!!!) ausgibt. Genauer sollte man eigentlich schreiben: range(0, 5). Warum 4 und nicht 5? 5 gibt hier das obere Limit der Zahlenfolge, sowie die Anzahl der Zahlen an. Also bedeutet range(5) (bzw range(0,5)), dass eine Zahlenreihe, bestehend aus 5 ganzen Zahlen, beginnend bei 0, endend bei < 5 - also 4, zurückgegeben wird.

Bedeutet wiederum: dieser Codeblock wird 5x ausgeführt. Für jedes i von 0 bis

< 5.

Aber - was befindet sich in diesem Codeblock - was außerhalb?
Dies wird in Python durch Einrückungen definiert.
Sprich alles, was hier 5x ausgeführt werden soll, muss nun im Vergleich Zeilenanfang der for-Schleife einmal mit Tab eingerückt sein.

REPL erledigt das für uns, sobald man die Zeile "for i in range(5): " mit einem Doppelpunkt beendet und einmal Enter gedrückt hat. Der Cursor ist nun eingerückt, zudem hat sich wieder der Zeilenanfang verändert. Hier stehen nun "...":

```
>>>
>>>
>>> for i in range(5):
...     □
```

Auch wenn wir mit Enter in die nächste Zeile wechseln, bleibt dieser noch auf selber Position.

Um die For-Schleife mit der Range-Funktion mal zu demonstrieren, sagen wir: für jedes i von 0 bis <5 : gib einmal den Wert von i aus.

Um Werte (oder Texte) im Terminal auszugeben nutzt man die Funktion print([variable / Text (in Anführungszeichen)])

```
>>> for i in range(5):
...     print(i)
```

Zweimaliges Drücken der Enter-Taste lässt den Cursor wieder an den Anfang der Zeile stellen, sprich, falls hier noch etwas gemacht werden sollte, dann wird das erst ausgeführt, sobald die Schleife verlassen, also 5x ausgeführt wurde.

```
>>>
>>> for i in range(5):
...     print(i)
...
... □
```

Da wir nach der Schleife nichts weiter ausführen lassen wollen, können wir nochmals mit Enter bestätigen, sodass die Schleife ausgeführt werden kann.

```
>>>
>>> for i in range(5):
...     print(i)
...
...
...
0
1
2
3
4
>>>
```

Das ging nun allerdings sehr sehr schnell! Wenn wir in diesem Tempo unsere LEDs blinken lassen würden, würden wir das Blinken sehr wahrscheinlich gar nicht mehr wirklich wahrnehmen!
Um diesen Prozess zu verlangsamen, müssen wir sogenannte delays einbauen. Dies macht man mit dem Modul "**utime**", bzw. dessen Funktion **sleep_ms()**

utime.sleep_ms(200) sorgt dafür, dass die nächste Codezeile (oder das nächste Mal, dass die Schleifenbedingung geprüft wird) erst 200 ms später stattfindet. Gib nun folgende Zeilen ins Terminal und bestätige - wie eben schonmal - 3x mit Enter.

```
>>> import utime
>>> for i in range(5):
...     print(i)
...     utime.sleep_ms(200)
```

Wie du siehst erscheinen nun die Ergebnisse der einzelnen Schleifendurchläufe in zeitlichen Abständen auf dem Terminal. Selbes Verhalten wollen wir für unsere blinkenden LEDs!

While-Schleife

Die einfachste While-Schleife ist die Endlosschleife. Einmal gestartet soll sie endlos den beinhalteten Codeblock ausführen. Hierzu sind gerade mal zwei Worte notwendig:

while True:

Wollen wir also unsere LEDs endlos blinken lassen müsste die Anweisungsfolge lauten:

mach immer: schalte ein – pause – schalte aus – pause.

Dies würde dann folgendermaßen aussehen:

```
>>> import utime
>>> while True:
...     strip.set(0, strip.TEAL, num=num_pixels)
...     utime.sleep_ms(200)
...     strip.set(0, 0x00, num=num_pixels)
...     utime.sleep_ms(200)
```

Wir können allerdings ebenso mit einer while-Schleife ausdrücken, dass der Codeblock nur 5x ausgeführt werden soll.
Nehmen wir an, eine Variable i hat vor Schleifenaufruf den Wert 0. Dann kann

man sagen `while (i<5)`, also während der Wert von `i` kleiner ist als 5, wird die Schleife ausgeführt. Dazu muss aber auch die Variable `i` innerhalb des nachfolgenden Codeblocks um 1 erhöht werden, ansonsten kann man sich auch so eine Endlosschleife generieren!

```
>>> while (i < 5):  
...     i += 1  
...
```

Die Bedingung einer `while`-Schleife darf auch von Funktionen abhängen!

Nehmen wir einmal an, wir haben bereits eine Funktion geschrieben, die prüft, ob es draußen dunkel ist. Wenn es dunkel ist, gibt sie den Wert `"True"` zurück, ansonsten `"False"`. Nennen wir sie mal `"is_dark()"`.

Schleifen funktionieren so, dass sie prüfen, ob die Bedingung nach dem Codewort `"while"` wahr ist. Wenn dem so ist, darf die Schleife ausgeführt, bzw. fortgeführt werden, ansonsten wird die Ausführung des Codes nach der Schleife fortgesetzt.

```
>>> while is_dark():
```

wäre also genauso möglich.

Die Funktion `"is_dark()"` werden wir jetzt allerdings noch nicht implementieren. Jedoch wird es Zeit, einen Blick auf Funktionen zu werfen. Selbstverständlich kannst du auch ohne Funktionen ein wenig mit den Werten beim Blink-Algorithmus spielen.

Allerdings wird dich sehr schnell die viele Schreibarbeit nerven. Dabei möchtest du doch nur die Farbe verändern, oder auch die Zeiten, wie lange die LEDs ein oder aus sein sollte.

Funktionen

Zu diesem Zweck gibt es Funktionen. Bevor man eine Funktion implementieren möchte, muss man sich darüber im Klaren sein

1. Welche Anweisungen in dieser Funktion gemacht werden - Funktionsname daher möglichst beschreibend wählen
2. ob oder welche Werte dazu nötig sind, die man ggf. verändern kann
3. ob die Funktion irgend etwas zurückgeben muss.
wie `True/False` bei `is_dark()`

Funktionen werden mit dem Schlüsselwort `"def"` (=define) eingeleitet. Der Aufbau sieht folgendermaßen aus:

```
def funktions-name(Parameterliste):  
    Anweisung(en)  
    (return)
```

das **return** Statement ermöglicht es einem, Ergebnisse von Berechnungen, oder eben Werte eines ausgelesenen Sensors, wieder zurück zu geben. Diesen Wert kann man also beim Funktionsaufruf in eine Variable speichern. Wollten wir jetzt, als sehr einfaches Beispiel, eine Funktion schreiben, in der zwei Werte (meist Parameter genannt) miteinander addiert werden sollen, würden wir wie folgt vorgehen:

- Zu 1. Wir wollen eine Funktion, in der zwei Zahlen miteinander addiert werden. Nennen wir sie also einfach "addieren".
- Zu 2. zwei Werte sind dazu nötig.
- Zu 3. das Ergebnis der Addition soll zurückgegeben werden.

```
def addieren(a, b):  
    e = a + b  
    return e  
  
ergebnis = addieren(a, b)
```

Übertragen auf unsere Blink-Funktion würde es also folgendermaßen aussehen:

- Zu 1. Wir wollen den LED-Strip blinken lassen. Daher nennen wir die Funktion `strip_blink()`
- Zu 2. Wir möchten die Farbe und die Zeiten verändern. Brauchen dazu also drei Werte.
- Zu 3. Da wir lediglich eine Ausgabe an einen Pin machen, brauchen wir also keinen Wert zurückgeben

In unserer bisherigen Blink-Funktion haben wir mit festen Werten für die Farbe und die Zeiten gearbeitet. Diese müssen wir nun durch Variablen ersetzen, dessen Wert erst bei Aufruf der Funktion bestimmt, bzw übergeben wird.

Auch bei Variablen gilt stets darauf zu achten, treffende Namen zu wählen, um die Lesbarkeit zu gewährleisten und - ganz wichtig - um auch nach einigen Wochen oder Monaten noch nachvollziehen zu können, was hier genau geschieht.

Für die Farbe bietet es sich daher an, die Variable "color" zu nennen.

Für die Zeit, die die LEDs leuchten sollen, nehmen wir "time_on"

Für die Zeit, die die LEDs nicht leuchten sollen, nehmen wir "time_off"

```
>>> def blink(color, time_on, time_off):  
...     strip.set(0, color, num=num_pixels  
...     utime.sleep_ms(time_on)
```

```
...     strip.set(0, 0x00, num=num_pixels)
...     utime.sleep_ms(time_off)
```

Nun kannst du ganz einfach mit Farbe und Blink-Intervallszeiten herumspielen.

Gebe dazu einfach folgendes im Terminal ein und ersetze die Werte nach Belieben:

```
>>> blink(strip.RED, 200, 10)
```

Gerne kannst du anstatt den vordefinierten Werten auch selbst mit Farbwerten herumspielen. Im Netz findet sich unter dem englischen Suchbegriff "hex color picker" (= hexadezimaler Farbwähler) eine einfache Möglichkeit, die gewünschte Farbe als Hexadezimalwert zu erhalten.

Funktionen mit optionalen Parametern

Sicherlich hat es dich bereits verwundert, dass bei der Funktion `strip.set()`, im Falle, dass alle LEDs gleichzeitig geschaltet werden sollen, die Anzahl als Parameter `"num= .."` übergeben wird.

Hierbei handelt es sich um einen optionalen Parameter, auch default-Parameter genannt.

Dies ist sehr praktisch, da wir hierfür einen Standardwert (default-Wert) nutzen können, der bei Bedarf verändert werden kann.

Wenn du nun schon etwas mit den Zeiten der Blink-Funktion herumgespielt hast und für dich optimale Zeit-Werte herausgefunden hast, kannst du diese ebenso als optionale Parameter angeben:

```
>>> def blink(color, time_on = 200, time_off = 100):
...     strip.set(0, color, num=num_pixels)
...     utime.sleep_ms(time_on)
...     strip.set(0, 0x00, num=num_pixels)
...     utime.sleep_ms(time_off)
```

Zu beachten ist hier lediglich, dass die default-Parameter grundsätzlich zuletzt genannt werden, da alle Variablen davor, hier lediglich "color" sogenannte Positionsparameter sind, sprich anhand ihrer Position identifiziert werden.

Die ursprüngliche Fassung der Blink-Funktion bestand nur aus Positionsparametern. `color`, `time_on`, `time_off` wurden beim Funktionsaufruf in dieser Reihenfolge eingegeben und auch so zugewiesen.

Da wir nun `time_on` und `time_off` als optionale Parameter definiert haben, dürfen diese genau aus diesem Grund nicht am Anfang stehen. Ansonsten würde die Gefahr bestehen, dass die Werte falsch zugeordnet werden!

Würden wir nun alle drei Werte nennen, müssten wir die Schlüsselwörter (time_on / time_off) nicht zwingend mit angeben.
Falls wir aber nur den Wert für time_off ändern wollten, den für time_on aber auf default lassen würden, können wir time_on also einfach auslassen und die Funktion mit

```
>>> blink(0x3F03F, time_off=50)
```

aufrufen

<https://www.python-kurs.eu/kurs.php>

<https://www.python-kurs.eu/klassen.php>

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/machine

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki/neopixel

Jetzt kommt der entscheidende Vorteil der MicroPython implementierung zum Einsatz!

Während man bei Arduino-Umgebung oft mühsam durch Recherchen erst herausfinden muss, welche Funktionen in einem Modul zur Verfügung stehen, und erst nach dem Upload auf den Controller herausfindet, ob das so überhaupt funktioniert, kann man das hier mittels REPL (Read Eval Print Loop) ganz schnell.

Aufgabe: Blinken in zwei Farben

Schaffst du es auch, den Strip anstatt ein- und auszuschalten zwischen zwei Farben wechseln zu lassen?

Oder abwechselnd in zwei Farben blinken zu lassen?

Ein Beispiel hierfür findest du im Anhang.

Ein erstes kleines Skript

Sicherlich hast auch du dich beim Funktionen schreiben im Terminal bereits einige Male vertippt und musstest alles neu eingeben? Dies ist auf Dauer sehr mühsam.

Da unsere Funktionen nun nach und nach immer komplexer werden und es nun vorteilhaft ist, Codepassagen einfach überarbeiten zu können, steigen wir nun auf einen Texteditor um.

Editor

Hierzu könntest du für den Anfang den normalen Texteditor nutzen, der auf jedem System vorinstalliert ist. Allerdings ist es sehr von Vorteil, einen Editor zu nutzen, der auch fürs coden gedacht ist. Ich nutze für meine MicroPython Projekte den Editor "Atom.io", der kostenlos für alle Systeme verfügbar ist. Dieser kann auf <https://atom.io/> heruntergeladen werden.

Falls du noch keinen Ablageort für deine Skripte haben solltest, empfiehlt es sich, gleich einen anzulegen.

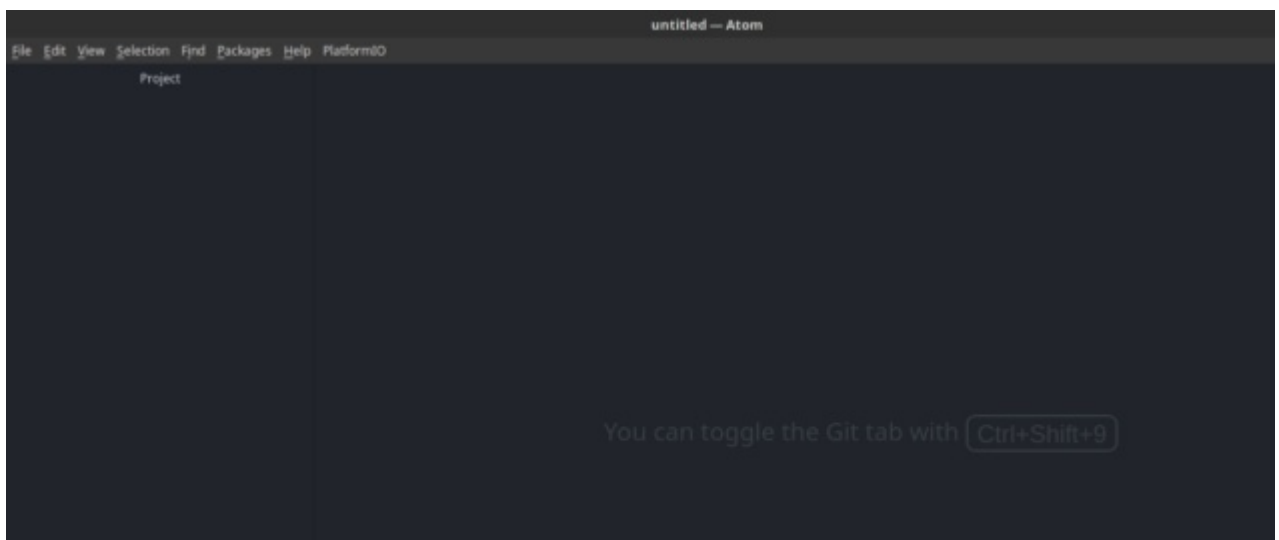
Ich habe zum Beispiel in meinem Home-Verzeichnis ein Verzeichnis namens "coding" in dem ich meine Skripte - sortiert nach Programmiersprache - ablege.

Atom besteht im Wesentlichen aus zwei Bereichen:

Im linken Bereich, "Project" genannt, hast du die Möglichkeit, Projektordner hinzuzufügen. Im rechten Teil werden dann die Textdateien angezeigt.

Um einen Projektordner zu erstellen, bzw. auszuwählen, klickst du einfach mit der rechten Maustaste in das "Project"-Feld und gehst auf "Add Project Folder".

Am besten ist es, du legst im selben Schritt innerhalb des Ordners für deine coding-Projekte einen eigenen Ordner für das Projekt an, das hier entsteht. Nennen wir es "rainbowWarrior"



Wenn du nun auf den neu erstellten Projektordner wieder mit der rechten Taste klickst, kannst du mit "new File" eine neue Datei anlegen. Alternativ geht das - bei markiertem Projektordner - mit der Taste A.
Die neue Datei muss gleich benannt werden (Dateiendung nicht vergessen!):
Nennen wir sie "animations.py"

Im rechten Bereich befindet sich dann der eigentliche Texteditor, hier werden deine geöffneten Dateien angezeigt. Und hier werden wir nun nach und nach, beginnend mit dem import-Statement, alles bisher gemachte übertragen.

```
1  import machine
2  import utime
3
4  led_pin = 14
5  num_pixels = 14
6
7  strip = machine.Neopixel(led_pin, num_pixels, machine.Neopixel.TYPE_RGBW)
8
9  def blink(color, time_on = 200, time_off = 100):
10     strip.set(0, color, num=num_pixels)
11     utime.sleep_ms(time_on)
12     strip.set(0, 0x00, num=num_pixels)
13     utime.sleep_ms(time_off)
14
```

Sehr schnell erkennt man hier die Vorteile:

Durch farbige Hervorhebungen wirkt der Text auf Anhieb viel strukturierter. So werden zum Beispiel alle Schlüsselwörter (import, def, ...) lila gekennzeichnet. Alle Funktionen, Konstruktoren sind cyan, sowie Werte orange gekennzeichnet sind.

Auch die Zeilenzahlen am linken Rand werden noch von Vorteil sein, wenn es zu den ersten Syntaxfehlern (oder manchmal einfach Tippfehlern) kommt.

Nachdem nun alles in den Editor übertragen und gespeichert wurde, müssen wir aber doch wieder ins Terminal switchen.

Hier wechseln wir mit cd zu dem Ordner, in dem unser Skript gespeichert wurde (in meinem Fall chrissi/coding/microPy/RainbowWarrior) und stellen eine Verbindung zum ESP her.

Auch ich muss mir den Befehl hierfür immer wieder rauskramen. Copy & Paste geht hier einfach schneller :)

```
$ rshell --buffer-size=30 -p /dev/ttyUSB0 -a -e nano
```

bevor wir aber REPL nutzen, kopieren wir das erstellte Skript auf den ESP:

```
$ cp animations.py /flash
```

Anschließend müssen wir den ESP neu starten. Hierzu einfach den Reset-Button (rst) auf dem Controller drücken.

Nun können wir den REPL-Mode starten. Also wieder einfach "repl" im Terminal eingeben und mit Enter bestätigen.

Um nun die Funktion blink zu starten, müssen wir das Skript zuerst importieren.

```
>>> import animations
```

Wie du sicherlich erkannt hast, funktioniert das auf die selbe Weise wie vorhandene Module eingefügt werden. Das bedeutet ja - du hast im Prinzip schon dein eigenes Mini-Modul geschrieben! Besteht zwar momentan nur aus einer Funktion, aber die wird nicht lange alleine bleiben!

Oooops! Wem ist's aufgefallen?

```
>>> import animationen
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "animationen.py", line 11
SyntaxError: invalid syntax
>>> █
```

Da hat sich der Fehlerteufel eingeschlichen! Aber so können wir gleich mal sehen, wozu die Zeilenanzeige im Editor gut ist!

Bei dieser Fehlermeldung sind für uns erst einmal die letzten beiden Zeilen interessant. In der letzten steht die Art des Fehlers, der gefunden wurde - also ein Syntax-Fehler. Die Zeile darüber sagt mir, welche Datei betroffen ist und in welcher Zeile es zu einem Fehler gekommen ist - also Zeile 11.

`utime.sleep_ms(time_on)` sieht allerdings richtig aus. Ich kann hier keinen Syntax-Fehler erkennen! In dem Fall sollte man sich auch die darüberliegende Zeile genauer ansehen.

`strip.set(0, color, num=num_pixels` - hier stimmt definitiv etwas nicht! Eine Klammer wurde geöffnet, aber nicht wieder geschlossen! So hat der Interpreter verstanden, dass die darauf folgende Zeile - Zeile 11 - also noch Bestandteil dessen ist, was in die Klammer gehört. Da dies aber so keinen Sinn macht, gab es nen Syntax-Fehler.

Also. Im Editor Klammer einfügen. Skript speichern. Im Terminal mit STRG+X REPL beenden. Skript erneut kopieren und REPL wieder starten.

Tipp für "faule": Du kannst diese Befehle auch auf eine Zeile komprimieren! In Linux ist die Systemsprache eigentlich C. Sprich Linux wurde komplett in C geschrieben. In fast allen Programmiersprachen (außer Python) werden die Zeilenenden mit einem ";" versehen, um dem Compiler oder dem Interpreter zu signalisieren, dass der Befehl hier nun zuende ist. Selbes können wir uns im

Seite 28

Terminal zunutze machen, indem wir den Befehl zum kopieren und REPL starten auf eine Zeile bringen und mit einem ";" trennen:

```
$ cp animations.py; repl
```

Auch wenn in Python keine Semikolons nötig sind um die Zeile (oder den Befehl) zu beenden, funktioniert selber Trick auch im REPL-Mode!
Da wir wieder den Controller neu starten müssen können wir - anstatt den Button zu drücken - hier einfach eingeben:

```
>>> import machine; machine.reset()
```

Nun sollte das Importieren problemlos funktionieren.

Skript auf dem ESP ausführen

Eine Funktion in einem Modul wird aufgerufen, indem man zuerst den Modulnamen schreibt, anschließend kommt, durch einen Punkt getrennt, der Funktionsnamen inkl. Parameter in der Klammer:

```
animationen.blink(0x00ff00)
```

Nun haben wir zwar eine Blink-Funktion, diese lässt die LED allerdings wieder nur 1x blinken!

Aufgabe: Schleifen erstellen

Erstelle nun anhand der vorherigen Beispiele eine Blink-Funktion, die

1. 10 x blinkt (einmal mit for- und einmal mit while-Schleife)
2. für immer blinken wird.
3. (Special) erweitere die Funktion (Parameterliste und Funktionsrumpf) so, dass beim Funktionsaufruf definiert werden kann, wie oft die Schleife durchlaufen werden soll.

Dateisystem auf dem ESP

Schauen wir uns nun mal die vorhandenen Dateien auf dem ESP an. Dazu ist es am einfachsten, wenn wir REPL wieder verlassen und uns den Inhalt mittels `ls` ausgeben lassen.

Um REPL zu verlassen, muss nun wieder STRG+X gedrückt werden.

```
ls /flash
```

gibt uns nun den aktuellen Inhalt des ESP aus.

Neben unserer "animationen.py" befindet sich nur eine Datei auf dem Controller: "boot.py".

Dies ist die erste Datei die beim Start des ESP ausgeführt wird. Hier sollte lediglich Code stehen, der finale Einstellungen vornimmt um den Boot-Prozess abzuschließen. Sehr viel passiert hier nicht - und wir sollten die Datei auch nicht verändern, jedoch können wir gerne mal einen Blick hinein werfen.

```
cp /flash/boot.py /home/chrisi/coding/microPy
```

kopiert die Datei vom Controller in dein lokales Filesystem, sodass du sie mit jedem beliebigen Editor öffnen kannst.

```
# This file is executed on every boot (including wake-boot from deepsleep)
import sys
sys.path[1] = '/flash/lib'
```

Hier wird das Modul "sys" (Abkürzung für System) eingebunden, und die Funktion `sys.path()` aufgerufen.

Um herauszufinden, was dies bewirkt, lohnt es sich, einen Blick in die Dokumentation der Micropython-Implementierung zu werfen:

<https://docs.micropython.org/en/latest/pyboard/library/sys.html>

`sys.path`: A mutable list of directories to search for imported modules.

Bedeutet, dass hier eine Liste an Ordnern in der Variablen gespeichert wird, in denen nach eingefügten Modulen gesucht wird. Sprich falls wir je weitere (externe) Module benötigen, die nicht in der Micropython-Implementierung vorhanden sind, müssen diese im Ordner "flash/lib" gespeichert werden, andererseits werden sie nicht gefunden.

Nachdem `boot.py` ausgeführt wurde, wird standardgemäß nach einer File namens "main.py" gesucht. Falls diese vorhanden ist, wird sie ausgeführt.

Nur, was macht die main.py?

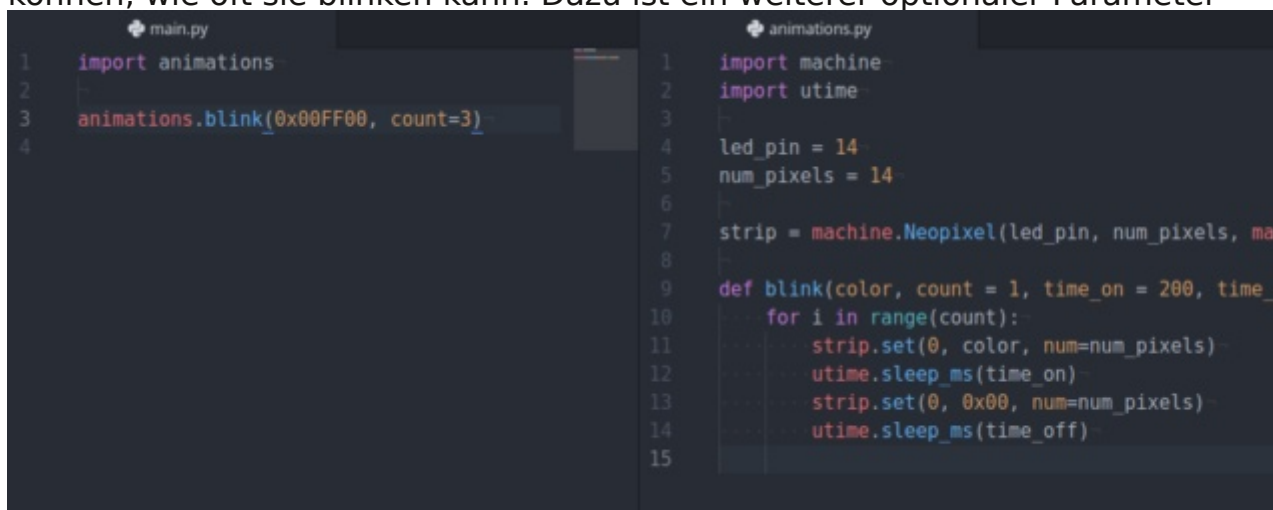
Dies kommt stark darauf an, wofür der ESP eingesetzt werden soll! So können hier weitere Module eingebunden und ausgeführt werden, wie unsere Blink-Funktion.

In unserem Fall können wir eigentlich alles übertragen, was wir zuletzt nach dem Neustart in REPL eingegeben haben, sodass die Funktion "blink" automatisch nach dem Boot-Prozess startet.

Im Editor erstellen wir also eine neue Datei namens "main.py" in unserem Projektordner. und fügen die zuletzt in REPL ausgeführten Zeilen ein.

In Atom können nicht nur mehrere Dateien gleichzeitig geöffnet sein (Zugriff über die Reiter), es gibt auch einen sogenannten Split-Mode, mit dem wir zwei Dateien nebeneinander setzen können. Dazu klicken wir auf den Reiter "animations.py", halten die Taste gedrückt und ziehen sie in den rechten Bereich des Fensters. So kann man beide Dateien nebeneinander betrachten, was gerade beim Schreiben von Funktionsaufrufen praktisch sein kann.

In der Zwischenzeit hast du sicherlich bereits die Aufgaben erledigt! Für den weiteren Verlauf des Projekts ist es gut, wenn wir der Blink-Funktion mitteilen können, wie oft sie blinken kann. Dazu ist ein weiterer optionaler Parameter



The screenshot shows the Atom code editor with two files open side-by-side. The left pane shows 'main.py' with the following code:

```
1 import animations
2
3 animations.blink(0x00FF00, count=3)
4
```

The right pane shows 'animations.py' with the following code:

```
1 import machine
2 import utime
3
4 led_pin = 14
5 num_pixels = 14
6
7 strip = machine.Neopixel(led_pin, num_pixels, ma
8
9 def blink(color, count = 1, time_on = 200, time_
10     for i in range(count):
11         strip.set(0, color, num=num_pixels)
12         utime.sleep_ms(time_on)
13         strip.set(0, 0x00, num=num_pixels)
14         utime.sleep_ms(time_off)
15
```

notwendig.

Vorsicht bei der Namensgebung für diesen Parameter! Hier würde sich "range" als treffender Name anbieten. Jedoch kann dies zu Konflikten führen. Ebenso darf man keine Schlüsselwörter als Variablennamen einsetzen. Daher nennen wir diese Variable lieber "count"

Animationen Teil 2: running dots

Eine Möglichkeit, mehr Bewegung in die Animation einzubringen, ist es, wenn man zum Beispiel nur eine LED zur gleichen Zeit - jedoch nach und nach um eine Position verschoben - leuchten lässt.

Nur - wie erreichen wir das?

schalte LED an Pos 0 ein - warte - schalte LED an Pos 0 aus - schalte LED an Pos 1 ein - warte - schalte LED an Pos 1 aus. schalte LED an letzter Pos ein - warte - schalte LED an letzter Pos aus.

Kommt dir bekannt vor?

Hierfür brauchen wir eine for-schleife, wobei die obere Grenze der range-Funktion gleich der Anzahl der LED (also `num_led`) ist.

Wir brauchen nun keine weitere Datei mehr anzulegen. Diese Funktion können wir einfach in `animations.py` einfügen!

Wichtigste Linux Befehle

VERZEICHNISSE, DATEIN:

cd	Wechselt in ein beliebiges Verzeichnis	cd /media/disk
cd ..	Wechselt ein Verzeichnis zurück	(/media/disk -> /media)
cd /	Wechselt in das tiefste Verzeichnis	cd /
cd -	Wechselt in das zuletzt besuchte Verzeichnis	cd -
cp	Kopiert eine Datei in angegebenes Verzeichnis	cp /tmp/test.txt /
media/disk		
mv	Verschiebt eine Datei und löscht die Quelldatei	mv /tmp/bla.txt /
media/disk		
mv	Benennt auch Dateien um	mv /tmp/x1.txt /tmp/x3.txt
rm	Löscht eine Datei	rm /tmp/bla.txt
rm -rf	Löscht alles in dem Verzeichnis	rm -rf /tmp/
mkdir	Erstellt ein Verzeichnis	mkdir /media/disk/bla
rmdir	Löscht ein Verzeichnis	rmdir /media/disk/bla
ls	Zeigt alle Dateien in einem Ordner an	ls /home/ubuntu
ls -l	Zeigt eine ausführliche Liste, mit ausführlichen Rechten an	ls -l /home/
ubuntu		
ls -la	Zeigt auch versteckte Dateien an	ls -la /home/ubuntu
pwd	Zeigt den Pfad zum aktuellen Verzeichnis	pwd
cat	Zeigt Inhalt einer Textdatei an	cat /home/test.txt
more	Zeigt Inhalt einer Datei seitenweise an	more test.txt
touch	Erstellt eine leere Datei in einem beliebigen Ordner	touch /ubuntu/
123.txt		

SYSTEM:

top	Gibt eine Übersicht über alle laufenden Prozesse und Systemauslastung
free	Zeigt an wie stark der RAM ausgenutzt wird
uptime	Dieser Befehl zeigt an wie lange das System schon online ist
uname	Zeigt mit der Option -a einige Systeminformationen an z.B. die Kernelversion uname -a
shutdown -h now	Führt den Computer runter
whoami	Zeigt den eingeloggten Benutzer ein whoami

Windows: Wichtigste Terminalbefehle:

Befehl	Was macht das
dir	Listet den Inhalt des aktuell ausgewählten Verzeichnisses auf
dir /p	Zeigt den Inhalt seitenweise an
dir /w	Lässt die ausführlichen Informationen weg
dir /s	Listet zusätzlich die Unterverzeichnisse mit auf
cd	Wechstelt das Verzeichnis
cd..	Wechselt ins übergeordnete Verzeichnis
cd\	Wechselt ins Root-Verzeichnis
md Verzeichnisname	Legt ein neues Verzeichnis an (auch mkdir Verzeichnis)
del Datei	Löscht die angegebene Datei
del Datei /s	Löscht zusätzlich zur angegebenen Datei auch alle Unterordner
rd Verzeichnis	Löscht das angegebene Verzeichnis (muss leer sein)
re Verzeichnis /s	Löscht das Verzeichnis (muss nicht leer sein)
copy Quelle Ziel an	Kopiert von Quelle nach Ziel und legt eine neue Datei an
move Quelle Ziel	Verschiebt von Quelle nach Ziel
rename NameAlt NmeNeu	Benennt eine Datei um
attrib	Ändert die Windows-Dateiattribute
attrib +R	Schaltet Schreibschutz an
attrib -R	Schaltet Schreibschutz aus
attrib s	Datei ist eine Systemdatei
attrib +H	Datei ist versteckt
attrib -H	Datei ist sichtbar
attrib A	Datei ist geändert/ archiviert
type	Zeigt den Inhalt einer Textdatei an

macOS: Wichtigste Terminal Befehle

Befehl	Bedeutung	Funktion
cd	change directory	Welchset in das angegebene Verzeichnis
ls	list	Auflistung von Verzeichnissen und Inhalten
cp	copy	Kopiert Dateien und Verzeichnisse
mv	move	Verschiebt Dateien und Verzeichnisse
rm	remove	Löscht Dateien oder Verzeichnisse
mkdir	make directory	Verzeichnis/Ordner erstellen
rmdir	remove directory	Verzeichnis/Ordner löschen
open		Öffnet die angegebene Datei
sudo	substitute user do	Führt den Befehl als Superuser (root) aus
pbcopy	pasteboard copy	Kopiert die Inhalte in die Zwischenablage
pbpaste	pasteboard past	Fügt die Inhalte aus der Zwischenablage ein
kill		Beendet den angegebenen Prozess
killall		Beendet alle Prozesse, die den angegebenen
Befehl ausführen		
chmod	change mode	Zugriffsrechte von Dateien und Ordner ändern
zip		Verpackt Dateien und Verzeichnisse
unzip		Entpackt Dateien und Verzeichnisse
clear		Leert das aktuelle Terminal-Fenster
screencapture		Erstellt ein Screenshot des aktuellen
Bildschirms		
find		Suche nach Datei
mdfind		Spotlight-Suche
ps		Listet alle aktuell aktiven Prozesse auf
top		Listet eine detaillierte Prozessliste auf
history		Listet die zuletzt benutzten Befehle auf
reboot		Das System neustarten
shutdown		Das System herunterfahren

