

**Bachelorarbeit**

Vorgelegt von

Ngahan nyonse Francois Brice

am. 24.Mai.2020

Studiengang Biomedical Engineering Bachelor of Ingenieur

**Kommunikationsprotokoll zwischen einem Computer und einer**

**Mikrocontroller Des Typs Pyboard**

Fachbereich 9 – Medizintechnik & Technomathematik

Fachhochschule Aachen, Campus Jülich

Diese Arbeit ist von mir selbständig angefertigt und verfasst. Es sind keine anderen als

die angegebenen Quellen und Hilfsmittel benutzt worden.

Unterschrift…...……….…………………….

1. **Prüfer:** Prof. Dr. -Ing Torsten Wagner
2. **Prüfer:** Prof. Dr.

# Danksagung

An dieser Stelle bedanke ich mich bei allen, die mich während der Bearbeitung der Bachelorarbeit unterstützt haben.

Insbesondere bedanke ich mich bei meinem Betreuer **Herrn Prof. Dr. Ing- Torsten Wagner** für sein Hilfreiche Unterstützung und seine Anregungen bei der Bearbeitung. Ferner möchte ich Herr \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ danken für seinen Zuspruch und Rat.

Zusätzlich möchte ich mich ganz herzlich bei meinem Freund **Donald Kapyouo** bedanken für seine stetige Anteilnahme an meiner Arbeit.

Darüber hinaus bedanke ich mich bei meiner Gast Familie **Tayoumo** und meiner ganzen Familie, die immer für mich da sind, bei meinen Eltern für ihre Unterstützung und ihr guter Rat.

# Inhaltsverzeichnis

[Danksagung III](#_Toc46845776)

[Inhaltsverzeichnis IV](#_Toc46845777)

[Abbildungsverzeichnis VI](#_Toc46845778)

[Tabellenverzeichnis VI](#_Toc46845779)

[Abkürzungsverzeichnis VII](#_Toc46845780)

[1 Einleitung 1](#_Toc46845781)

[2 Grundlagen und Definition 2](#_Toc46845782)

[2.1 Mikrocontroller 2](#_Toc46845783)

[2.1.1 Eigenschaften eines Mikrocontrollers 2](#_Toc46845784)

[2.1.2 Einsetzung von Mikrocontroller 2](#_Toc46845785)

[2.1.3 Funktionsweise Mikrocontroller 3](#_Toc46845786)

[2.2 Mikroprozessor 5](#_Toc46845787)

[2.3 Mikropython (µC) 6](#_Toc46845788)

[2.3.1 Mikropython Devices 7](#_Toc46845789)

[2.4 Der Pyboard 10](#_Toc46845790)

[2.4.1 Definition des Pyboard & Hardware. 10](#_Toc46845791)

[(siehe 2.3.1\_\_i) 10](#_Toc46845792)

[3 Module und Klassen 13](#_Toc46845793)

[3.1 Das Module Pyserial & Class Serial 13](#_Toc46845794)

[3.2 Das Klasse UART 15](#_Toc46845795)

[3.3 Module „Pyb“ 17](#_Toc46845796)

[3.4 Das Klasse USB\_VCP - USB Virtual Comm Port 18](#_Toc46845797)

[3.5 Das Modul uasyncio 20](#_Toc46845798)

[3.5.1 Wichtige Klassen &Funktionen von Uasyncio 24](#_Toc46845799)

[3.6 Klasse Pins 28](#_Toc46845800)

[3.7 Das Module Uarray 29](#_Toc46845801)

[ Das Module Array 29](#_Toc46845802)

[4 Framework und Umgebung 32](#_Toc46845803)

[4.1 Python 32](#_Toc46845804)

[4.2 Benutze Software 33](#_Toc46845805)

[4.2.1 Visual Studio Code 33](#_Toc46845806)

[5 Benutzte Materiellen 35](#_Toc46845807)

[5.1 Pyboard 35](#_Toc46845808)

[5.2 Breadboard (Steckplatine) 35](#_Toc46845809)

[5.3 Potentiometer 35](#_Toc46845810)

[6 37](#_Toc46845812)

[7 38](#_Toc46845813)

[8 40](#_Toc46845814)

[9 Hauptteil 2: Grundlagen und Methodik 41](#_Toc46845815)

[9.1 Theoretischer Bezugsrahmen 41](#_Toc46845816)

[9.2 Methodische Vorgehensweise 41](#_Toc46845817)

[10 Hauptteil 3: Empirische Untersuchung 42](#_Toc46845818)

[10.1 Analysen 42](#_Toc46845819)

[10.2 Interpretation der Ergebnisse 42](#_Toc46845820)

[11 Zusammenfassung 43](#_Toc46845821)

[Literaturverzeichnis VII](#_Toc46845822)

[Anhänge IX](#_Toc46845823)

[Anhang 1 IX](#_Toc46845824)

[Anhang 2 IX](#_Toc46845825)

[Eidesstattliche Versicherung X](#_Toc46845826)

[Stichwortverzeichnis XI](#_Toc46845827)

# Abbildungsverzeichnis

Abbildung 1 Aufbau eines Mikrocontroller 4

Abbildung 2 Erläuterung der Begriffe Mikroprozessor, Mikrorechner-/System 6

Abbildung 3 Das erste und Original Pyboard 7

Abbildung 4 BBC Micro: Bit 9

Abbildung 5 Die Adafruit-Feder HUZZAH-Tafel (Bildnachweis: Adafruit). 9

Abbildung 6: Das Espressif ESP32 Development Board (Bildnachweis: Adafruit). 10

Abbildung 7 Pin-belegung für PYBv1.1 11

Abbildung 8 Pins on a Micro: bit 28

Abbildung 9 Python Programm mit Visual Studio 33

Abbildung 10 Breadboard 35

# Tabellenverzeichnis

Tabelle 1: Typ Code 30

# Verzeichnis und benutzender Abkürzungen und Symbole

**µ** Mikro-

**µPY**  Mikropython.

**µC**  Mikrocontroller.

**ADC**  Analog Digital Converter.

**BBC** British Broadcasting Corporation

**BIT**  Maßeinheit für den Informationsgehalt.

**BUF**  Puffer sind in der Informatik und Telekommunikation Speicher für die Zwischen-

Lagerung von Daten

**BSD**  Die Berkeley-Standard Distribution (BSD)ist die Implementierung von UNIX,

die an der Universität von Kalifornien, Berkeley, entwickelt wurde.

**BYTE** Maßeinheit der Digitaltechnik und der Informatik, welches meist für eine Folge

aus 8 Bit steht.

**CHAR** Datentyp für Datenbereiche/Felder, deren Elemente jeweils ein Zeichen repräs-

entieren

**CHIP**  Ein Platinen Computer

**CPU**  Central Processing Unit

**CTS**  Clear to send

**DAC**  Digital To Analogic Converter

**DFU**  Device Firmware Update

**EPROM** Erasable Programmable Read-Only Memory

**EEPROM** Electrically erasable programmable read-only memory

**GND**  Ground

**GPIO**  General Purpose Input Output

**HZ**  Herz

**ID**  Identification

**IDE**  Integrated Development Environment

**INT** Interger

**IO**  Input Output

**IPC** Inter-processing Communication

**LED**  Light-emmiting Diode

**LINUX**  unix-like multi-user operating systems

**MAX** Maximum

**MHZ**  Megahertz

**MIN**  Minimum

**OSX**  Betriebssystem der Firma Hard- und Software-Unternehmens Apple.

**PHP** Hypertext Preprocessor

**PIN**  Personal Identification Number

**PY** Python

**PYB** Pyboard

**RAM**  Random Access Memory

**REPL**  Read-evaluation-print Loop

**ROM**  Read only Memory

**RTS**  Request to send

**RX**  Receiver

**SHELL** Mittel, durch die ein Benutzer mit einem Betriebssystem interagiert

**SRAM** Static random-access memory

**TX** Transmitter

**VCP**  Virtual Com Port

**V**  Volt

**UART**  Universal Asynchronous Receiver Transmitter

**USB**  Universal serial Bus

**XON**  Signal to turn on a Transmitter

**XOFF**  Signal to turn off a Transmitter

# Einleitung

Damit mein Mobiltelefon funktioniert, muss es mit einem Mikrocontroller ausgestattet sein, der das gesamte Gerät steuern kann. Und dies gilt für fast alle elektronischen Geräte, die wir in unserem Alltag benutzen. Seit dem Aufkommen neuer Technologien in der Welt, vieles hat sich geändert. Heutzutage sind elektronische Geräte, die früher groß waren, sehr klein geworden, und der Mikrocontroller, der zu ihrer Herstellung verwendet wurde, wird Tag für Tag extrem klein wie zum Beispiel das Pyboard. Apropos Pyboard, dies wird das wichtigste Element meiner Arbeit sein. Das Pyboard ist ein Mikrochip, der zur Familie der Mikrocontroller gehört und mit Mikropython-Programmierung arbeitet.

(1) Diese Arbeit beschäftigt sich mit der asynchronen Kommunikation, und Gegenstand dieser Untersuchung ist das Kommunikationsprotokoll zwischen einem µC vom Typ "Pyboard" und einem Computer. In dieser Arbeit wird ein µPy-basiertes Board programmiert, damit die Analogen Ein-Ausgang angesteuert werden können.

(2) Dann muss eine Kommunikation zwischen PC und Mikrocontroller über USB aufgebaut werden.

Ansatzpunkt für diese Arbeit ist, dass die ersten und zweiten oben genannten Punkte voneinander entkoppelt werden können, so dass sie asynchron arbeiten können. Das bedeutet, dass µC auch während der Messung weiterhin Befehle empfangen und Daten senden muss, und solange ein Befehl empfangen wird, darf er die Messungen nicht verändern.

Ziel dieser Arbeit ist es, eine Ausarbeitung eines Kommunikationsprotokoll, also eine Reihe von Befehlen die an den Mikrocontroller verschickt werden können, die diesen verstehen und abarbeiten, ohne dass irgendwelche Befehle gestört oder unterbrochen wird.

Damit dieses Projekt erfolgreich durchführen zu können, muss ich das Module "Serial" für die Kommunikation zwischen meinem Computer und meinem Pyboard verwenden. Und um mehrere Befehle parallel verarbeiten zu können, ohne sich gegenseitig zu unterbrechen, muss ich das Uasyncio-Modul verwenden.

# Grundlagen und Definition

## Mikrocontroller

Mikrocontroller sind spezielle Mikrocomputer auf einem Chip, die an bestimmte Anwendungen angepasst sind. Dabei handelt es sich in der Regel um Steuer- oder Kommunikationsaufgaben, die einmal programmiert und dann während der Lebensdauer des Mikrocontrollers auf diesem ausgeführt werden. Die Anwendungsbereiche sind sehr breit gefächert. Mikrocontroller arbeiten normalerweise unsichtbar in einer Vielzahl von Geräten und Systemen, die uns in unserem Alltag umgeben.

### Eigenschaften eines Mikrocontrollers

Ein Mikrocontroller hat die folgenden typischen Merkmale:

* Besitz einen Programmspeicher (früher ROM, EPROM, EEPROM, heute FLASCH)
* Ein Datenspeicher (RAM)
* Eine Verarbeitungseinheit (CPU)
* digitale Eingabe- / Ausgabe- Ports (GPIO), oft analoge Eingabegeräte (ADC)
* einen oder mehrere Zeitgeber
* Kommunikationsbausteine (COM, UART, etc…)

### Einsetzung von Mikrocontroller

Mikrocontroller sind häufig in eingebetteten Systemen zu finden. Eingebettete Systeme sind in einen technischen Kontext integrierte Computer/elektronische Rechner (embedded = eingebettet). Die Aufgabe des Bordcomputers ist es, das System zu steuern, zu regeln oder zu überwachen. Es kann auch zur Daten- und Signalverarbeitung verwendet werden und verantwortlich sein. Zum Beispiel Ver.-/Entschlüsselung, Decodierung oder Filterung.[[1]](#footnote-1)

Obwohl Mikrocontroller in fast allen Bereichen unserer Gesellschaft vorkommen, werde ich die wichtigsten Bereiche auflisten, in denen wir mit Mikrocontrollern zu tun haben

Hier sind einige Beispiele für Aufgaben, wo die Mikrocontroller eingesetzt werden können:

* In Geräten der Medizintechnik
* Steuerung von Infusionsgeräten,
* Kardiovaskulären Monitoren,
* Atemschutzgeräte,
* Dialysegeräte,
* Im Haushalt
* Fernsehers,
* Staubsaugers,
* Waschmaschine,
* Kaffeemaschine,
* Etc…
* In Kraftfahrzeugs Technik
* Das Motorsteuergerät,
* Das Stabilitätsprogramm,
* Verschiedene Assistenten, z.B. beim Bremsen,
* Antriebskontrolle,
* In der Automatisierungstechnik
* Prozesssteuerung und -regelung,
* Materialflusssteuerung,
* Produktionsanlagen,
* Das Überwachen von Prozessen

### Funktionsweise Mikrocontroller

Die Funktionalitäten der Mikrocontroller sind vielfältig, aber ich werde mich auf die wichtigsten konzentrieren.

Der Mikrocontroller besteht aus:

* Einer Zentraleinheit:

ist eine zentrale Verarbeitungseinheit (CPU) der Teil eines Computersystems, der die anderen Teile des Systems steuert und verwaltet.

* Einem SRAM:

einen Informationsspeicher, der beschreibbar, löschbar und wieder lesbar ist

* Einem EEPROM:

können jederzeit beliebig oft beschrieben und entfernt werden und bewahren ihren Zustand auch nach Ausschalten der Versorgungsspannung.

* Einem Flash:

sind sehr performante EEPROMs, mit denen der Controller in das Zielsystem programmierbar ist.

* Seriellen Schnittstelle:

ist eine Beschreibung für eine Datenübertragungsstrecke von zwei Geräten, bei der die Einzelbits nacheinander transportiert werden.

* Zeitgeber.[[2]](#footnote-2)

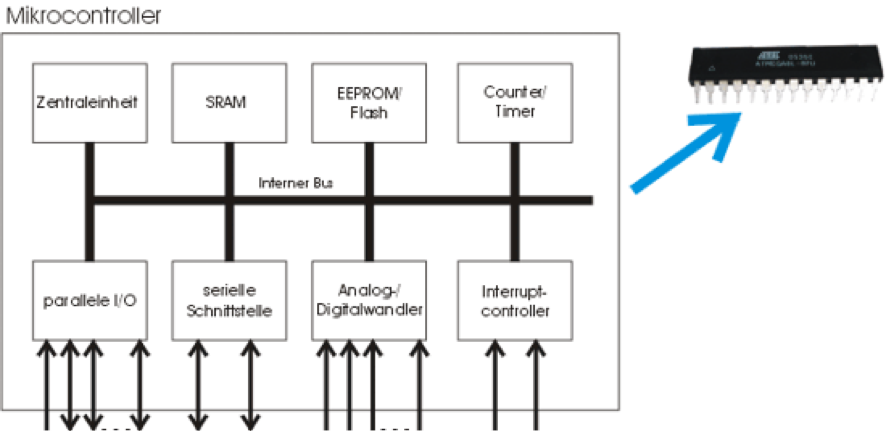


Abbildung 1 Aufbau eines Mikrocontroller

Aus Peripheriebaustein können z.B. sein:

* Digitale Ein- und Ausgabebausteine:

Die werden auch Ports genannt und dienen zum Lesen oder Erzeugen von Ein-/Aus-Zuständen, Schaltvorgängen, Impulse oder Frequenzen.

* Komparator:

Die können " irgendwelche" Spannungen miteinander vergleichen.

* A/D-Wandler, D/A-Wandler:

kann einen "beliebigen" analogen Spannungswert in einen Zahlenwert umwandeln und als ganzzahlige Zahl für die weitere Verarbeitung bereitstellen.

* Zähler(„Timer“) -Baustein:

für zeitabhängige Steuerungen, die notwendig sind, wenn man beispielsweise die Uhrzeit ausgeben möchte

* Sender- / Empfänger-Baustein:

man kann größere Datenmengen übertragen, typische Schnittstellenbausteine sind UART, SPI, IÂ²C-Bus oder CAN-Bus.

* Interrupt Controller-Baustein:

dienen dazu, kurze Reaktionszeiten des Mikrocontrollers zu erhalten oder laufende Programme durch ein Interrupt zu unterbrechen.[[3]](#footnote-3)

## Mikroprozessor

heutzutage kennen viele Leute nicht den Unterschied zwischen einem Mikrocontroller und einem Mikroprozessor, deshalb wird es für mich wichtig sein, zu definieren, was ein Mikroprozessor ist.

Ein Mikroprozessor ist die zentrale Steuereinheit (CPU) eines Datenverarbeitungssystems, die heute in der Regel zusammen mit anderen Bauelementen auf einem einzigen Mikrochip eingebaut ist. Sie besteht in der Regel aus einer Bedieneinheit und einer arithmetischen Einheit, auch Verfahrenskern genannt, sowie einer Schnittstelle zur Außenwelt. Je nach Bedarf und Leistung können weitere Verarbeitungskomponenten wie Cache-Speicher und virtuelle Speicherverwaltung hinzugefügt werden. Die Tätigkeit eines Mikroprozessors ist die Bearbeitung eines Programms, das aus einer Folge von Befehlen zur Bearbeitung einer Anwendung besteht. Zu diesem Zweck muss der Mikroprozessor auch alle anderen Bestandteile der Datenverarbeitungseinrichtung steuern, wie z.B. Speicher und Ein-/Ausgabeschnittstellen4.[[4]](#footnote-4)4

* Ein Mikroprozessorsystem ist ein technisches System, das einen Mikroprozessor enthält. Dabei muss es sich nicht um einen Rechner oder eine Kaffeemaschine handeln, die von einem Mikroprozessor geregelt wird, sondern um ein Mikroprozessorsystem.
* Ein Mikrocomputer ist ein Computer oder eine Maschine, deren zentrale Verarbeitungseinheit aus einem oder mehreren Mikroprozessoren besteht. Zusätzlich zu dem/den Mikroprozessor(en) enthält ein Mikrocomputer einen Speicher, Ein-/Ausgabeschnittstellen und ein Anschlußsystem.
* Ein Mikrocomputersystem ist ein Mikrocomputer mit angeschlossener Peripherie an den Ein-/Ausgabeschnittstellen, z.B. einem Keyboard, einer Maus, einem Computerbildschirm, einem Printer oder ähnliches.

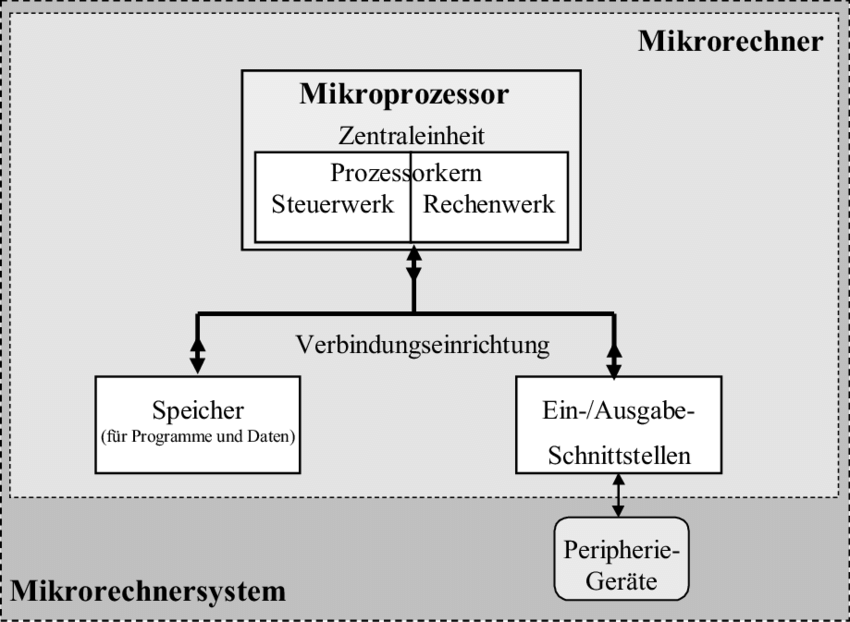
x

Abbildung 2 Erläuterung der Begriffe Mikroprozessor, Mikrorechner-/System[[5]](#footnote-5)6

### 2.3 Mikropython (µPy)

µPy ist eine Neuimplementierung von Python 3 für Mikrocontroller und eingebettete Systeme. Es ist auch eine erstaunliche Ingenieursleistung, um eine Neuimplementierung von Python zu haben, die unter solchen eingeschränkten Bedingungen funktioniert. es bringt eine der schönsten, leicht erlernbarsten und ausdrucksstärksten Programmiersprachen in die Welt der eingebetteten Entwicklung.[[6]](#footnote-6)5

Mikropython ist vollgepackt mit fortschrittlichen Funktionen wie interaktive Eingabeaufforderungen, Ganzzahlen beliebiger Genauigkeit, Schließungen, Listenverständnis, Generatoren, Ausnahmebehandlung und mehr. Aber er ist kompakt genug, um in nur 256k Code-Space und 16k RAM zu passen und zu laufen.

Mikropython versucht, so kompatibel wie möglich zu regulärem Python zu sein, um Ihnen die einfache Übertragung von Code vom Desktop auf einen Mikrocontroller oder ein eingebettetes System zu ermöglichen.[[7]](#footnote-7)7

### 2.3.1 Mikropython Devices

Ich kann nicht über Mikropython sprechen, ohne die 4 Hauptmikrocontroller zu erwähnen, mit denen er arbeiten könnte.

Ich würde natürlich mit dem Hauptmikrocontroller beginnen, der unser Arbeitswerkzeug ist.

1. **Der Pyboard**

Das Pyboard ist ein Gerät, das für Mikropython entwickelt und gebaut wurde (siehe Abbildung 3).

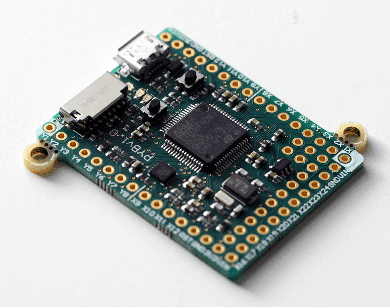


Abbildung 3 Das erste und Original Pyboard

Das Board wird über ein Mikro-USB-Kabel mit dem PC verbunden. Diese Verbindung bietet zwei Möglichkeiten, mit den Geräten zu interagieren: als USB-Flash-Laufwerk und als seriell-basiertes Python REPL[[8]](#footnote-8)8.

Die Platine enthält ein kleines Dateisystem als Teil ihres Flash-Speichers (on Chip). Wenn wir auf das Pyboard als Flash-Speicher zugreifen, können wir Dateien auf und vom Dateisystem des Boards kopieren. Wenn wir ein Python-Skript namens main.py auf dem Dateisystem des Boards erstellen, führt Mikropython dieses Skript beim Start aus. Folglich kann das Skript auf dem Board ausgeführt werden, ohne dass es mit einem PC verbunden sein muss.

Genau wie normales Python können wir die anderen Pyboard-Dateien kopieren, sie in main.py importieren und Ihren Code in entsprechende Module organisieren.

Einmal physikalisch verbunden, ist es auch möglich, jedes serielle Programm zu verwenden, um sich mit dem Pyboard zu verbinden und Mikropython REPL-Prompt zu erhalten. Und um sie sofort auszuführen und auszuwerten, müssen wir Befehle in das REPL eingeben.

* **Der Hardware**

Das Pyboard ist in vielerlei Hinsicht das leistungsfähigste Board, das in dieser Arbeit verwendet wird. Es verfügt über einen STM32F405RG Mikrocontroller, eine 168 MHz Cortex M4 CPU mit Hardware-Fließkomma, 1.024 KB Flash-ROM und 192 KB RAM.

Die Verbindung mit Peripheriegeräten und anderen Komponenten erfolgt über die GPIO-Pins, die um drei Kanten der Platine herum verlaufen.

Zusätzlich zum Mikro-USB-Steckplatz kann das Pyboard mit Batterien oder einer anderen Stromquelle betrieben werden. Schließen Sie die positive Leitung der Stromversorgung an den mit VIN gekennzeichneten Port an und erden Sie sie an GND. Die Eingangsspannung muss zwischen 3,6V und 10V liegen.[[9]](#footnote-9)9

Zu den weiteren Peripheriegeräten gehören ein Bluetooth-Modul, Prototyp-Skins (auf denen Testschaltungen aufgebaut werden können) und verschiedene Arten von Servomotoren zur Herstellung von Dingen, die sich bewegen.

Es war für mich ein konkretes Problem, zu erklären, dass Pyboard unser Hauptmaterial ist. Ich werde einfach versuchen, die anderen Geräte aufzulisten, ohne viele Details zu nennen.

1. **BBC Micro: Bit**

Das BBC Micro: Bit ist ein einfaches und dennoch leistungsstarkes Rechengerät für Programmieranfänger. Das Gerät wurde von der BBC (British Broadcasting Corporation) entwickelt, um die digitale Kreativität zu fördern. Es ist klein, billig und einfach zu bedienen.



Abbildung 4 BBC Micro: Bit[[10]](#footnote-10)10

1. **ESP8266**

ESP8266 ist ein beliebtes WiFi-fähiges System-on-Chip (SoC) von Expressif Systems.

Es gibt eine Vielzahl von Modulen und Platinen aus verschiedenen Quellen, die den ESP8266-Chip tragen.

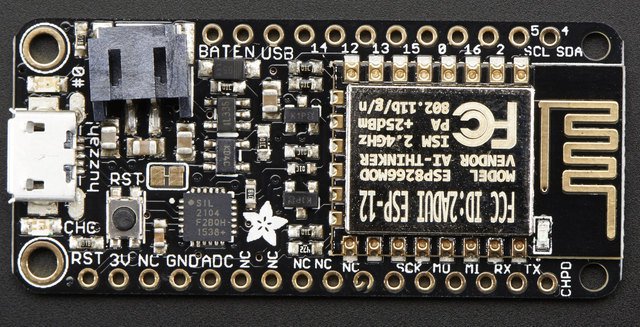


Abbildung 5 Die Adafruit-Feder HUZZAH-Tafel (Bildnachweis: Adafruit).

1. **ESP32**

Wie das ESP8266, das ESP32 ist ein beliebtes WiFi- und Bluetooth-fähiges System-on-Chip (SoC) von Expressiv Systems.

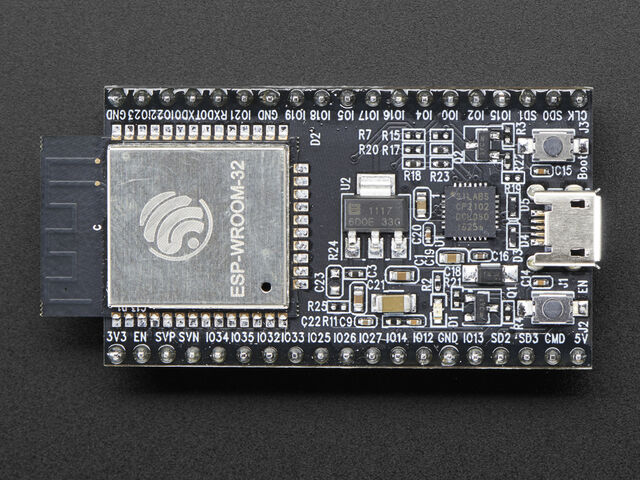


Abbildung 6: Das Espressif ESP32 Development Board (Bildnachweis: Adafruit).

## 2.4 Der Pyboard

## 

## 2.4.1 Definition des Pyboard & Hardware.

## (siehe 2.3.1\_\_i)

**Entwickler-Setup**

Im Pyboard-paket[[11]](#footnote-11)11 ist Mikropython bereits auf das Gerät geflasht. Es kann jedoch sein, dass dies nicht die neueste Version ist, die für das Board verfügbar ist, daher ist es immer eine gute Idee, das Gerät neu zu flashen. Es ist kein schwieriger Prozess, und es stellt sicher, dass Sie Zugang zu den neuesten Fehlerbehebungen und Updates erhalten.

Laden Sie zunächst die neueste Firmware von der Mikropython-Website (https://micropython.org/download/) herunter. Zweitens müssen Sie alles von Ihrem Pyboard trennen und den DFU-Pin (Device Firmware Update) mit dem 3,3-V-Pin verbinden. Sie liegen direkt nebeneinander, und wenn Sie eine Lupe haben, sind sie auch auf der Rückseite der Platine beschriftet. Sie sind auf dem Foto auf der Vorderseite der Platine hervorgehoben.

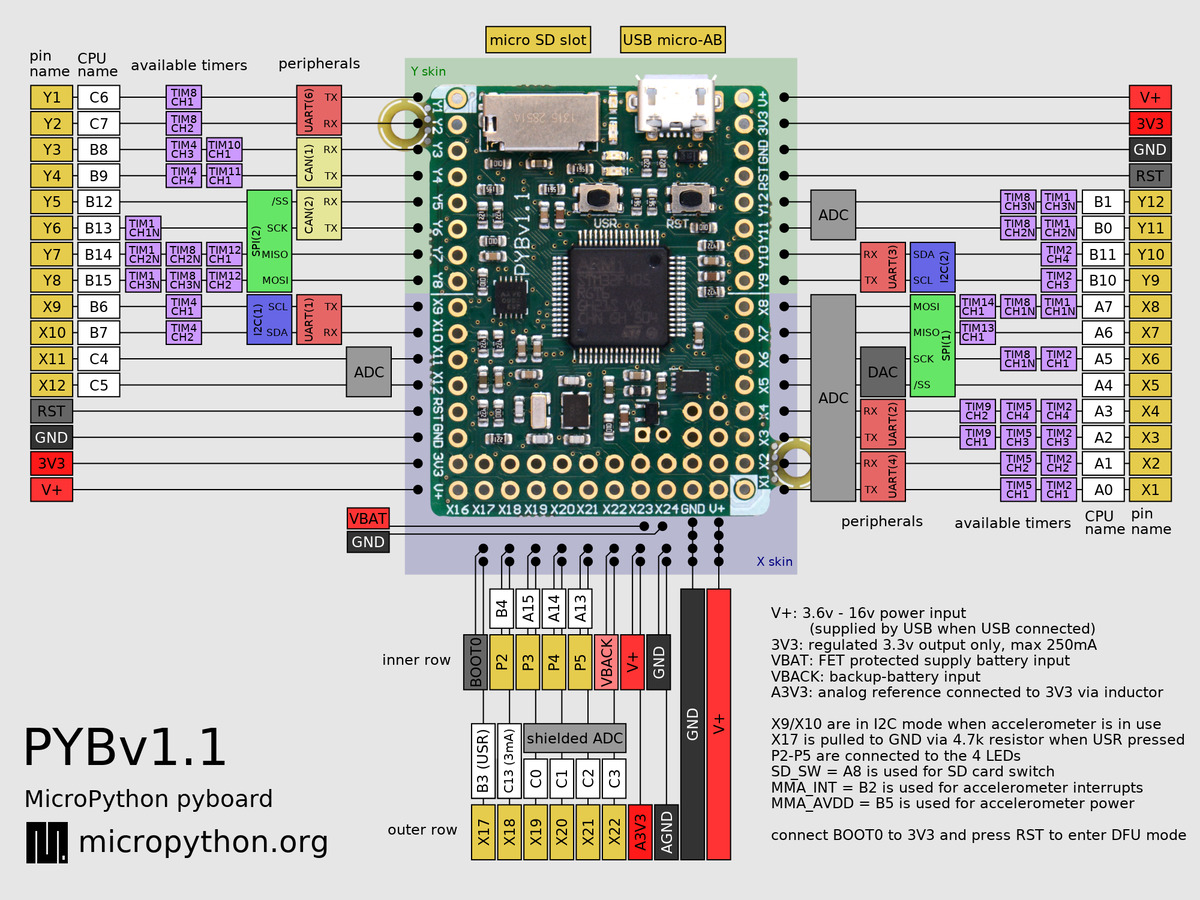


Abbildung 7 Pin-belegung für PYBv1.1

Die Verbindung der Pins lässt sich am besten mit einem Stecker-zu-Stecker-Brückendraht herstellen.

Für die Aktualisierung der Firmware auf dem Pyboard ist ein DFU-Dienstprogramm erforderlich. Es gibt zwei Optionen: dfu-util oder pydfu. Das pydfu-Skript kann von Mikropython‘s Github-Repository (https://github.com/micropython/micropython/blob/master/tools/pydfu.py) heruntergeladen werden.

Verbinden Sie das Pyboard über das USB-Kabel mit Ihrem Computer. Geben Sie für dfu-util den folgenden Befehl ein:

" sudo dfu - util --alt 0 -D firmware.dfu".

Wir müssen die Datei „Firmware.dfu“ durch die neueste Datei ersetzen, die Sie von der Mikropython-Website heruntergeladen haben.

um pydfu zum Aktualisieren der Firmware zu verwenden, verwenden Sie den folgenden Befehl:

"s sudo python pydfu.py -u firmware.dfu"

Nachdem die Firmware aktualisiert wurde, trennen Sie das Pyboard von Ihrem Computer und entfernen Sie das Überbrückungskabel zwischen den DFU- und 3,3-V-Pins.[[12]](#footnote-12)12

Wenn das Gerät als abnehmbares USB-Flash-Laufwerk erscheint, wird es zunächst drei Dateien enthalten:

**boot.py**

Wird beim Start des Geräts ausgeführt und richtet verschiedene Konfigurationsoptionen ein.

**main.py**

Das Hauptskript, das Ihren Code enthält. Es wird unmittelbar nach boot.py ausführt.

**README.txt**

Enthält grundlegende Informationen über das Pyboard.

Öffnen Sie die Datei main.py in Ihrem Texteditor. Ersetzen Sie den Code-Kommentar, den Sie darin finden, durch den folgenden Codeschnipsel:

"import pyb"

"Pub.LED(4). on()"

Das Pyb-Modul enthält alle Funktionen und Klassen, die für die Arbeit mit der Hardware des Pyboards erforderlich sind. Das obige Skript schaltet einfach LED4 (die blaue LED) ein.

# Module und Klassen

Für die gute Durchführung unserer Arbeit benötigen wir eine Menge Module und Klassen, die die Ausführung der Aufgaben ermöglichen. Es wäre mir wichtig, zunächst jedes Modul und jede Klasse zu definieren und zu erklären, um ein gutes Verständnis der Arbeit zu ermöglichen.

## Das Module Pyserial

Dieses Modul enthält den Zugang für die serielle Schnittstelle. Es stellt Hintergrundsysteme für Python zur Verfügung, die unter Windows, OSX, Linux, BSD (möglicherweise jedes POSIX-kompatible System) und IronPython laufen. Das Modul mit dem Namen "Serial" wählt automatisch das entsprechende Hintergrundsystem aus.

* **Eigenschaften**

Gleiche klassenbasierte Schnittstelle auf allen unterstützten Plattformen.

Zugriff auf die Port-Einstellungen über Python-Eigenschaften.

Unterstützung für verschiedene Bytegrößen, Stoppbits, Parität und Flusskontrolle mit RTS/CTS und/oder Xon/Xoff.

Arbeiten mit oder ohne Empfangs-Timeout.

Datei-ähnliche API mit "Lesen" und "Schreiben" ("readline" etc. ebenfalls unterstützt).

Die Dateien in diesem Paket sind 100% reine Python-Dateien.

Der Port ist für binäre Übertragung eingerichtet. Kein NULL-Byte-Stripping, CR-LF-Übersetzung usw. (die oft für POSIX aktiviert sind.) Dies macht dieses Modul universell einsetzbar.

Kompatibel mit der IO-Bibliothek.

RFC 2217-Client (experimentell), Server in den Beispielen angegeben.

* **Erfordernisse**

Python 2.7 oder Python 3.4 und neuer.

* **Installation**

Dadurch wird ein Paket installiert, das von Python verwendet werden kann („import serial“).

PySerial kann von „PyPI“ direkt installiert werden: („ python -m pip install pyserial „).

* **Class Serial**

Parameter:

1. **Port:**

Der Port wird sofort bei der Objekt Erstellung geöffnet, wenn ein Port angegeben wird. Er wird nicht geöffnet, wenn Port None ist und ein nachfolgender Aufruf von open () erforderlich ist.

Port ist ein Gerätename: abhängig vom Betriebssystem. z.B. /dev/ttyUSB0 unter GNU/Linux oder COM3 unter Windows.

1. **Baudrate(int):** Einheit der Kommunikationstechnik, die die Gehgeschwindigkeit darstellt.

Der Parameter Baudrate kann einer der Standardwerte sein: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200. Diese werden auf allen Plattformen gut unterstützt.

1. **Bytesize:** Anzahl der Datenbits.

Mögliche Werte: „FIVEBITS, SIXBITS, SEVENBITS, EIGHTBITS“

1. **Parity**: Aktivieren der Paritätskontrolle.

Mögliche Werte: PARITY\_NONE, PARITY\_EVEN, PARITY\_ODD.

1. **Stopbits:** Anzahl der Stoppbits.

Mögliche Werte: STOPBITS\_ONE, STOPBITS\_ONE\_POINT\_FIVE.

1. **Timeout(float):** Legt einen Lese-Timeout-Wert fest.
2. **Write\_timeout(float):** Legt einen Schreib-Timeout-Wert fest.
3. **rtscts(bool):** Hardware (RTS/CTS) -Flusskontrolle aktivieren.
4. **Open ():** Port öffnen.

Der Zustand von rts und dtr wird angewendet.

1. **Close ():** Port schließen.
2. **Read (Size=?):**

Size: Anzahl von Bytes zu lesen.

Liest Size Bytes von der seriellen Schnittstelle (Port). Wenn ein Timeout eingestellt ist, kann es vorkommen, dass weniger Zeichen als angefordert zurückgegeben werden. Ohne Timeout blockiert es, bis die angeforderte Anzahl von Bytes gelesen wurde.

1. **Write(data):** Data zu senden, und gibt die Anzahl der geschriebenen Bytes zurück

Schreibt der Byte-Daten an den Port.

1. **inwaiting:**  Gibt die Anzahl der Bytes im Empfangsbereich zurück.[[13]](#footnote-13)13

## Das Klasse UART

Wenn man ein Board über das USB-Kabel an einem Computer anschließt, ist es möglich, über das REPL mit dem Gerät zu kommunizieren. Was dies ermöglicht, ist der universelle asynchrone Empfänger/Sender (UART), ein Teil des Mikrocontrollers, der zwischen serieller und paralleler Kommunikation vermittelt. Serielle Nachrichten kommen bitweise (ein Hoch/Tief-Signal), und die UART-Hardware setzt das Signal in Bytes zusammen (eine parallele Darstellung, die normalerweise aus 8 Bits besteht), die über den internen Bus zur weiteren Verarbeitung durch den Mikroprozessor gesendet werden. Umgekehrt nimmt der UART zum Senden einer Nachricht ein Byte und wandelt es in eine Reihe von High/Low-Signalen um, die die einzelnen Bits repräsentieren.

Für diese Arbeit müssen mehrere Arrangements getroffen werden. Zunächst muss der Sendeanschluss (normalerweise als TX bezeichnet) von Gerät A mit dem Empfangsanschluss (normalerweise als RX bezeichnet) von Gerät B verbunden werden und umgekehrt.

Zweitens muss auch Übereinstimmung über das Timing der seriellen Kommunikation bestehen, damit der UART die einzelnen High/Low-Signale erkennen kann. Dies ist die Geschwindigkeit der Kommunikation und wird als eine von mehreren Standard-Baudraten ausgedrückt: 9600, 14400, 19200, 28800, 28800, 38400, 57600 und 115200 Bit pro Sekunde.

Drittens müssen Sie manchmal die Anzahl der Bits pro Byte angeben (obwohl der Standard normalerweise 8 ist). Möglicherweise müssen Sie auch angeben, ob das Paritätsbit (dessen Funktion darin besteht, Übertragungsfehler zu erkennen) und die Anzahl der Stoppbits, die das Ende einer Übertragungseinheit signalisieren, verwendet werden soll.

Die UART hat auch ein ""first in/first out"". (FIFO)-Warteschlange, so dass Bytes gepuffert werden können, wenn sie nicht sofort nach dem Empfang gelesen werden.

Standardmäßig ist der UART auf Mikropython-Boards mit den internen USB-UART TX/RX-Pins verbunden, die an einen USB-Seriell-Wandler angeschlossen sind, so dass der UART über den USB-Anschluss mit Ihrem PC verbunden ist. Auf der PC-Seite öffnet eine Bibliothek wie PySerial oder ein Tool wie Picocom eine serielle Verbindung über einen USB-Port eines PCs und ermöglicht uns so das Senden und Empfangen von Daten von und zum Python REPL. Die Standard-Baudrate für eine solche Verbindung mit Mikropython ist 115200.

UART-Interaktionen in Mikropython erfordern, dass die Verbindung konfiguriert ist (unter Angabe der Pins, der Baudrate und anderer bereits besprochener Attribute). Jede Karte hat eine leicht unterschiedliche Art und Weise, den UART zu initialisieren und zu konfigurieren, obwohl sie auf konzeptioneller Ebene alle auf die gleiche Weise funktionieren. Nach der Konfiguration arbeiten Sie mit einem Byte-Stream mit bekannten Methoden wie „Read“, „Readline“ und „Write“. Dies ist auf allen Plattformen identisch.[[14]](#footnote-14)14

die UART-Methode basiert zunächst auf der Initialisierung:

„UART.init(*baudrate, bits=8, parity=none,stop=1,|\*timeout=0,flow=0,timeout\_char=0,read\_buf\_len=64)*

* **Baudrate:** ist die Taktfrequenz.
* **Stop:** ist die Anzahl der Stoppbits, 1 oder 2.
* **Flow:** h setzt die Flusskontrollart. Kann 0, UART.RTS, UART.CTS oder UART.RTS | UART.CTS sein.
* **Timeout:** ist das Zeitintervall in Millisekunden für das Warten auf das Schreiben/Lesen des ersten Zeichens.
* **Timeout\_char:** b ist das Zeitintervall in Millisekunden, die beim Schreiben oder Lesen zwischen den Zeichen gewartet werden muss.
* **Read\_buf\_len:** ist die Zeichenlänge des Lesepuffers (0 zum Deaktivieren).

Diese Methode erzeugt eine Ausnahme, wenn die Baudrate nicht innerhalb von 5% des gewünschten Wertes eingestellt werden konnte. Die minimale Baudrate wird durch die Frequenz des Busses bestimmt, auf dem der UART liegt; UART (1) und UART (6) sind APB2, der Rest liegt auf APB1. Die Standardbusfrequenzen ergeben eine Mindestbaudrate von 1300 für UART (1) und UART (6) und 650 für die anderen. Man verwendet „pyb.freq“, um die Busfrequenzen zu reduzieren, um niedrigere Baudraten zu erhalten.

Einige wichtige Funktionen des pyb-Moduls:

* **UART.deinit():** Schaltet den UART-Bus aus.
* **UART.any():** Gibt die Anzahl der wartenden Bytes zurück (kann 0 sein).
* **UART.read([*nbytes*]):** Zeichen lesen. Wenn *nbytes* angegeben ist, dann lesen Sie höchstens so viele Bytes. Wenn *nbytes* im Puffer verfügbar sind, kehrt er sofort zurück, andernfalls kehrt er zurück, wenn genügend Zeichen eintreffen oder die Zeitüberschreitung verstrichen ist.

Wenn keine nbytes angegeben sind, dann liest die Methode so viele Daten wie möglich. Sie kehrt nach Ablauf der Zeitüberschreitung zurück.

* **UART.readchar():** Empfangt ein einzelner Charakter auf dem Bus.

Rückgabewert: Das gelesene Zeichen, als Ganzzahl. Gibt -1 bei Zeitüberschreitung zurück.

* **UART.readinto(*buf* [, *nbytes*]**)**:** Liest Bytes in den *Buf* . Wenn *nbytes* angegeben ist, dann liest höchstens so viele Bytes ein. Andernfalls lesen Sie höchstens *len(buf)-*Bytes ein.[[15]](#footnote-15)15

Rückgabewert: Anzahl der gelesenen und in *buf* gespeicherten Bytes oder *None* on timeout.

* **UART.readline():** Liest eine Zeile, die mit einem Zeilenumbruchzeichen endet. Wenn eine solche Zeile existiert, erfolgt die Rückkehr sofort. Wenn die Zeitüberschreitung verstrichen ist, werden alle verfügbaren Daten zurückgegeben, unabhängig davon, ob ein Zeilenumbruch vorhanden ist.

Rückgabewert: die gelesene Zeile oder *None* bei Zeitüberschreitung, wenn keine Daten verfügbar sind.

* **UART.write(*buf*):** Schreibt den Puffer von Bytes auf den Bus. Wenn die Zeichen 7 oder 8 Bit breit sind, ist jedes Byte ein Zeichen. Wenn Zeichen 9 Bits breit sind, werden zwei Bytes für jedes Zeichen verwendet (Little Endian), und *buf* muss eine gerade Anzahl von Bytes enthalten.

Rückgabewert: Anzahl der geschriebenen Bytes. Wenn eine Zeitüberschreitung auftritt und keine Bytes geschrieben wurden, wird *none* zurückgegeben.

* **UART.write\_char(*char*):** Schreibt ein einzelnes Zeichen auf den Bus. *Char* ist eine Ganzzahl zum Schreiben.
* **UART.sendbreak():** bSendet eine Abbruchbedingung auf den Bus. Dies bewirkt, dass der Bus für eine Dauer von 13 Bit auf Low geschaltet wird.

Rückgabewert: *none*.[[16]](#footnote-16)15

## Module „Pyb“

Das „Pyb“-Modul enthält spezifische Funktionen im Zusammenhang mit dem Board.

Einige wichtige Funktionen des Pyb-Moduls:

* **Zeitliche Funktionen**
* Pyb.delay(ms): Wartezeit für die angegebene Anzahl an Millisekunden.
* Pyb.udelay(µs): Wartezeit für die angegebene Anzahl an Mikrosekunden.
* Pyb.millis(): Gibt die Anzahl der Millisekunden zurück, seit dem Board zuletzt zurückgesetzt wurde.
* **Funktionen zur Zurücksetzung**
* Pyb.hard\_reset(): Setzt das Pyboard auf ähnliche Weise zurück wie das Klicken auf den externen RESET-Knopf.
* Pyb.bootloader(): Aktiviert den Bootloader ohne BOOT\*-Pins.
* Pyb.fault\_debug(value): Aktivieren bzw. Deaktivieren der Fehlersuche bei schweren Fehlern. Ein Hard-fehler liegt vor, wenn ein fataler Fehler im zugrunde liegenden System vorliegt, wie z.B. ein ungültiger Speicherzugriff.

Wenn das Wertargument Falsch ist, wird der Board automatisch zurückgesetzt, wenn ein Hard-fehler auftritt.

Wenn der Wert True ist, wird das Board bei einem Hard-fehler die Register und den Stack-Trace ausgeben und dann die LEDs auf unbestimmte Zeit zyklisch schalten.

* **Leistungsabhängige Funktionen**
* Pyb.stop(): Bringt das Pyboard in einen "schlafenden" Zustand.

Dies reduziert den Stromverbrauch auf weniger als 500 uA. Um aus diesem Ruhezustand aufzuwachen, ist ein externes Interrupt oder ein Echtzeit-Uhr-Ereignis erforderlich. Beim Aufwachen wird die Ausführung dort fortgesetzt, wo sie unterbrochen wurde.

* Pyb.standby(): Bringt das Pyboard in einen "Tiefschlaf"-Zustand.

Dies reduziert den Stromverbrauch auf weniger als 50 uA. Um aus diesem Ruhezustand aufzuwachen, ist ein Echtzeituhr-Ereignis oder ein externes Interrupt auf X1 (PA0=WKUP) oder X18 (PC13=TAMP1) erforderlich. Beim Aufwachen wird das System einem Hard-Reset unterworfen.[[17]](#footnote-17)16

## Das Klasse USB\_VCP - USB Virtual Comm Port

Die meisten verwendeten Klassen gehören zum Pyb-Modul.

Die Klasse USB\_VCP ermöglicht die Erstellung eines Stream-ähnlichen Objekts, das den virtuellen USB-Kommunikationsanschluss darstellt. Sie kann zum Lesen und Schreiben von Daten über USB an den angeschlossenen Host verwendet werden.

* **Constructors:** *class* pyb.USB\_VCP(*id=0*)

Erstellt ein neues USB\_VCP-Objekt. Das *id*-Argument gibt an, welcher USB-VCP-Port verwendet werden soll.

* **Methoden**
* USB\_VCP.init(*\\*,flow=-1*):

Konfiguriert den USB-VCP-Anschluss. Wenn das Flow-Argument nicht -1 ist, dann setzt der Wert die Flusssteuerung, die bitweise oder aus USB\_VCP.RTS und USB\_VCP.CTS bestehen kann. RTS wird zur Steuerung des Leseverhaltens und CTS zur Steuerung des Schreibverhaltens verwendet.

* USB\_VCP.setinterrupt(*chr*):

Legt das Zeichen fest, das den laufenden Python-Code unterbricht. Dies ist standardmäßig auf 3 (CTRL-C) eingestellt, und wenn ein CTRL-C-Zeichen über den USB-VCP-Port empfangen wird, wird eine KeyboardInterrupt-Ausnahme erzeugt.

Setzt diesen Wert auf -1, um diese Unterbrechungsfunktion zu deaktivieren. Dies ist nützlich, wenn Sie Raw-Bytes über den USB-VCP-Port senden möchten.

* USB\_VCP.isconnected():

Gibt *True* zurück, wenn USB als serielles Gerät angeschlossen ist, andernfalls *False*.

* USB\_VCP.any():

Gibt *True* zurück, wenn irgendwelche Zeichen warten, sonst *False*.

* USB\_VCP.close():

Diese Methode macht nichts. Sie existiert, damit das USB\_VCP-Objekt als Datei fungieren kann.

* USB\_VCP.read([*nbytes*]):

Liest höchstens *nbytes* vom seriellen Gerät und gibt sie als Byte-Objekt zurück. Wenn *nbytes* nicht angegeben ist, liest die Methode alle verfügbaren Bytes vom seriellen Gerät. Der USB\_VCP-*Stream* arbeitet implizit im nichtblockierenden Modus. Wenn also keine ausstehenden Daten verfügbar sind, kehrt diese Methode sofort mit dem Wert *None* zurück.

* USB\_VCP.readinto(*buf* [*, maxlen*]):

Liest Bytes aus dem seriellen Gerät und speichert sie in *buf*, das ein pufferähnliches Objekt sein sollte. Es werden höchstens *len(buf)-*Bytes gelesen. Wenn *maxlen* angegeben wird und dann werden höchstens *min (maxlen, len(buf))* Bytes gelesen.

Gibt die Anzahl der gelesenen und in *buf* gespeicherten Bytes zurück oder *None*, wenn keine ausstehenden Daten verfügbar sind.

* USB\_VCP.readline():

Liest eine ganze Zeile vom seriellen Gerät.

Gibt ein Bytes-Objekt zurück, das die Daten enthält, einschließlich des abschließenden Zeilenumbruchzeichens oder *None*, wenn keine ausstehenden Daten verfügbar sind.

* USB\_VCP.readlines():

Liest so viele Daten wie möglich von dem seriellen Gerät und unterteilt sie in Zeilen.

Gibt eine Liste von Bytes-Objekten zurück, wobei jedes Objekt eine der Zeilen ist. Jede Zeile enthält das Zeilenvorschubzeichen.

* USB\_VCP.write(*buf*):

Schreibt die Bytes von buf auf das serielle Gerät.

Gibt die Anzahl der geschriebenen Bytes zurück.

* USB\_VCP.recv(*data, \\*, timeout=5000*):

Empfängt Daten auf dem Bus:

* *Data* kann eine ganze Zahl sein, die die Anzahl der zu empfangenden Bytes angibt, oder ein veränderbarer Puffer, der mit empfangenen Bytes gefüllt wird.
* *Timeout* ist die Zeit in Millisekunden, die auf den Empfang wartet.

Rückgabewert: wenn *Data* eine ganze Zahl ist, dann ein neuer Puffer der empfangenen Bytes, andernfalls wird die Anzahl der in *Data* eingelesenen Bytes zurückgegeben.

* USB\_VCP.send(*data, \\*, timeout=5000*):

Sendet *Data* über den USB-VCP:

* *Data* ist die zu sendenden Data (eine zu sendende Ganzzahl oder ein Pufferobjekt).
* *Timeout* ist das Zeitintervall in Millisekunden, das auf das Senden wartet.

Rückgabewert: Anzahl der gesendeten Bytes.[[18]](#footnote-18)17

## Das Modul uasyncio

Uasyncio ist der wichtigste Teil unserer Arbeit, denn es ist das Modul, das es uns ermöglicht, Befehle konkurrenzfähig zu machen, ohne dass sie sich gegenseitig beeinflussen.

Es ist wichtig, dass wir zunächst den Asyncio definieren und verstehen.

Asyncio ist eine Bibliothek zum Schreiben von Konkurrent Code unter Verwendung der *async/await*-Syntax. Asyncio wird als Grundlage für mehrere asynchrone Python-Frameworks verwendet, die hochleistungsfähige Netzwerk- und Web-Server, Bibliotheken für Datenbankverbindungen, verteilte Task-Warteschlangen usw. bereitstellen.

Asyncio passt oft perfekt zu strukturiertem Netzwerkcode, der an IO gebunden ist und auf hoher Ebene strukturiert ist.

Asyncio bietet eine Reihe von High-Level-APIs (Application programming Interfaces) an:

* führt gleichzeitig Python-Koroutinen aus und hat die volle Kontrolle über deren Ausführung;
* Netzwerk IO und IPC ausführen;
* Steuerung von Unterprozessen;
* Verteilung von Aufgaben über Warteschlangen;
* gleichzeitigen Code synchronisieren;
* Erstellen und Verwalten von Ereignisschleifen, die asynchrone APIs für die Vernetzung, die Ausführung von Unterprozessen, die Behandlung von Betriebssystemsignalen usw. bereitstellen;
* Implementieren effiziente Protokolle mit Hilfe von Transporten;[[19]](#footnote-19)18

Bevor ich die Funktionen von Uasyncio erkläre, ist es für mich wichtig, zunächst zu verstehen, was Koroutines und Awaitables und Task sind:

* **Koroutines**

Koroutinen, die mit der async/await-Syntax deklariert werden, sind die bevorzugte Art, asynchrone Anwendungen zu schreiben. Beispielsweise gibt das folgende Codestück "Hallo" aus, wartet 1 Sekunde und gibt dann " Welt" aus:

**>>> import** **asyncio**

**>>> async** **def** main():

**...**  print('hello')

**...**  **await** asyncio.sleep(1)

**...**  print('world')

**>>>** asyncio.run(main())

hello

world

Um eine Koroutine tatsächlich zu betreiben, bietet asyncio drei Hauptmechanismen an:

* Die Funktion *asyncio.run()* zum Ausführen der Funktion "main()" des obersten Einstiegspunktes (siehe obiges Beispiel).
* Warten auf einer Koroutine. Der folgende Codeabschnitt druckt nach einer Wartezeit von 1 Sekunde "Hallo" und nach weiteren 2 Sekunden "Welt":

**import** **asyncio**

**import** **time**

**async** **def** say\_after(delay, what):

**await** asyncio.sleep(delay)

print(what)

**async** **def** main():

print(f"started at *{*time.strftime('*%X*')*}*")

**await** say\_after(1, 'hello')

**await** say\_after(2, 'world')

print(f"finished at *{*time.strftime('*%X*')}")

asyncio.run(main())

Erwartete Output:

started at 17:13:52

hello

world

finished at 17:13:55

* Die *asyncio.create\_task()-*Funktion zum gleichzeitigen Ausführen von Koroutines als Asyncio-Tasks.

Verändern wir das obige Beispiel und führen zwei *say\_after-Koroutinen* gleichzeitig aus:

**async** **def** main():

task1 = asyncio.create\_task(

say\_after(1, 'hello'))

task2 = asyncio.create\_task(

say\_after(2, 'world'))

print(f"started at *{*time.strftime('*%X*')*}*")

*# Wait until both tasks are completed (should take*

*# around 2 seconds.)*

**await** task1

**await** task2

print(f"finished at *{*time.strftime('*%X*')*}*")

Erwartete Output:

started at 17:14:32

hello

world

finished at 17:14:34

* **Awaitable**

Ein Objekt ist ein erwartbares Objekt, wenn es in einem Warte-Ausdruck verwendet werden kann. Viele asynchrone APIs sind so gestaltet, dass sie *awaitables* akzeptieren.

Es gibt drei Haupttypen von erwartbaren Objekten: Koroutinen, Aufgaben und Futures.

* + - Koroutines: Python-Koroutinen sind abzuwarten und können daher auch von anderen Coroutinen erwartet werden:

**import** **asyncio**

**async** **def** nested():

**return** 42

**async** **def** main():

*# Nothing happens if we just call "nested()".*

*# A coroutine object is created but not awaited,*

*# so it \*won't run at all\*.*

nested()

*# Let's do it differently now and await it:*

print(**await** nested()) *# will print "42".*

asyncio.run(main())

Wichtig: Der Begriff "Koroutine" kann für zwei miteinander verwandte Begriffe verwendet werden:

* eine Koroutine-Funktion: eine *async def* Funktion;
* ein Koroutine-Objekt: ein Objekt, das durch den Aufruf einer Koroutine-Funktion zurückgegeben wird
* **Task**

Tasks werden zur gleichzeitigen Planung von Koroutinen verwendet.

Wenn eine Koroutine in einen Task mit Funktionen wie *asyncio.create\_task()* eingepackt wird, wird die Koroutine automatisch so geplant, dass sie bald ausgeführt wird:

Beispiel Eines Tasks:[[20]](#footnote-20)19

**import** **asyncio**

**async** **def** nested():

**return** 42

**async** **def** main():

*# Schedule nested() to run soon concurrently*

*# with "main()".*

task = asyncio.create\_task(nested())

*# "task" can now be used to cancel "nested()", or*

*# can simply be awaited to wait until it is complete:*

**await** task

asyncio.run(main())

### Wichtige Klassen &Funktionen von Uasyncio

* **Ausführen eines Asyncio-Programms**

asyncio.run(coro, \*, debug=False)

Führt die Coroutine Coro aus und gibt das Ergebnis zurück. Diese Funktion führt die übergebene Koroutine aus, kümmert sich um die Verarbeitung der Asynchron-Ereignisschleife und schließt die Ausführung der Asynchrongeneratoren ab.

Diese Funktion kann nicht aufgerufen werden, wenn eine andere asynchrone Ereignisschleife im selben Thread läuft.

Wenn Debug *True* ist, wird die Ereignisschleife im Debug-Modus ausgeführt.

Diese Funktion erzeugt immer eine neue Ereignisschleife und schließt sie am Ende. Sie sollte als Haupteinstiegspunkt für asynchrone Programme verwendet werden und sollte idealerweise nur einmal aufgerufen werden.

Bsp:

**async** **def** main():

**await** asyncio.sleep(1)

print('hello')

asyncio.run(main())

* **Tasks erstellen**

asyncio.create\_task(coro, \*, Name=none) Packt die Coro-Koroutine in eine Aufgabe ein und plant ihre Ausführung. Gibt das Objekt Task zurück.

Wenn der Name nicht None ist, wird er mit Task.set\_name() als Name der Aufgabe festgelegt.

Die Aufgabe wird in der von get\_running\_loop() zurückgegebenen Schleife ausgeführt, RuntimeError wird ausgeführt, wenn im aktuellen Thread keine laufende Schleife vorhanden ist.

* **Sleeping**

Sperrung für Verzögerungssekunden. Wenn ein Ergebnis geliefert wird, wird es an den Anrufer zurückgegeben, wenn die Koroutine beendet ist.

sleep() bewirkt immer eine Unterbrechung der aktuellen Aufgabe, so dass andere Aufgaben ausgeführt werden können.[[21]](#footnote-21)20

Beispiel einer Koroutine, die 5 Sekunden lang jede Sekunde das aktuelle Datum anzeigt:

**import** **asyncio**

**import** **datetime**

**async** **def** display\_date():

loop = asyncio.get\_running\_loop()

end\_time = loop.time() + 5.0

**while** **True**:

print(datetime.datetime.now())

**if** (loop.time() + 1.0) >= end\_time:

**break**

**await** asyncio.sleep(1)

asyncio.run(display\_date())[[22]](#footnote-22)20

* **Konkurrenten laufende Aufgaben**

Awaitable *asyncio.gather*(\*aws, loop=None, return\_exceptions=False)

Führt erwartete Ereignisse in der aws-Sequenz gleichzeitig aus.

Wenn es sich bei einem in der aws-Sequenz erwarteten Ereignis um eine Coroutine handelt, wird es automatisch als Task geplant.

Wenn alle erwartbaren Objekte erfolgreich abgeschlossen werden, ist das Ergebnis eine gesammelte Liste der zurückgegebenen Werte. Die Reihenfolge der Ergebniswerte entspricht der Reihenfolge der awaitables in aws.

Wenn *return\_exceptions* *False* (Standard) ist, wird die erste ausgelöste Ausnahme sofort an die Aufgabe propagiert, die bei *gather()* wartet. Andere erwartete Variablen in der aws-Sequenz werden nicht abgebrochen und laufen weiter.

Wenn return\_exceptions *True* ist, werden Ausnahmen wie erfolgreiche Ergebnisse behandelt und in der Ergebnisliste gesammelt.

Wenn *gather()* abgebrochen wird, werden alle eingereichten awaitables (die noch nicht abgeschlossen sind) ebenfalls abgebrochen.

Wenn eine Task oder ein Future aus der aws-Sequenz abgebrochen wird, wird sie so behandelt, als hätte sie *CancelledError* verursacht - der Aufruf von *gather()* wird in diesem Fall nicht abgebrochen. Damit soll verhindert werden, dass die Stornierung eines eingereichten Task/Future dazu führt, dass andere Tasks/Futures storniert werden.[[23]](#footnote-23)21

Beispiel:

**import** **asyncio**

**async** **def** factorial(name, number):

f = 1

**for** i **in** range(2, number + 1):

print(f"Task *{*name*}*: Compute factorial(*{*i*}*)...")

**await** asyncio.sleep(1)

f \*= i

print(f"Task *{*name*}*: factorial(*{*number*}*) = *{*f*}*")

**async** **def** main():

*# Schedule three calls \*concurrently\*:*

**await** asyncio.gather(

factorial("A", 2),

factorial("B", 3),

factorial("C", 4),

)

asyncio.run(main())

*# Expected output:*

*#*

*# Task A: Compute factorial(2)...*

*# Task B: Compute factorial(2)...*

*# Task C: Compute factorial(2)...*

*# Task A: factorial(2) = 2*

*# Task B: Compute factorial(3)...*

*# Task C: Compute factorial(3)...*

*# Task B: factorial(3) = 6*

*# Task C: Compute factorial(4)...*

*# Task C: factorial(4) = 24*

* **Zusatzfunktionen**

*Uasyncio.wait\_for* (awaitable, timeout):

Wartet bis das Abwarten abgeschlossen ist, aber bricht es ab, wenn es länger als ein paar Sekunden dauert. Wenn "awaitable" kein Task ist, wird daraus ein Task erstellt. Wenn eine Zeitüberschreitung auftritt, bricht sie den Task ab und löst asyncio.timeoutError aus: dies sollte vom Aufrufer abgefangen werden.

Gibt den Rückgabewert von awaitable zurück.[[24]](#footnote-24)22

*Uasyncio.wait\_for\_ms* (awaitable, timeout):

Ähnlich wie wait\_for, aber das Timeout ist eine ganze Zahl in Millisekunden. Dies sind eine Coroutine und eine Mikropython-Erweiterung.[[25]](#footnote-25)22

* **Klasse Event**
* *Class* uasyncio.Event

Erstellt ein neues Ereignis, das zur Synchronisierung von Aufgaben verwendet werden kann. Ereignisse starten im gelöschten Zustand.

* Event .is\_set()

Gibt True zurück, wenn das Ereignis gesetzt ist, andernfalls False.

* Event.set()

Festlegen des Ereignisses. Alle auf das Ereignis wartenden Vorgänge werden zur Ausführung geplant.

* Event.clear()

Löschen der Aktion.

* Event.wait()

Abwarten, bis das Ereignis eingestellt ist. Wenn das Ereignis bereits gesetzt ist, kehrt es sofort zurück.

* **Klasse Lock**
* Class uasyncio. Lock

Erstellt eine neue Sperre, die zur Koordinierung von Tasks verwendet werden kann. Lock beginnt im entsperrten Zustand. Zusätzlich zu den unten aufgeführten Methoden können Locks in einer async with-Anweisung verwendet werden.

* Lock. locked()

Gibt True zurück, wenn die Lock gesperrt ist, andernfalls False.

* Lock. acquire()

Wartet, bis sich das Schloss im entsperrten Zustand befindet, und verriegeln Sie es dann auf atomare Weise. Es kann immer nur einen Task die Sperre zu einem Zeitpunkt erfassen.

* Lock. release()

Öffnung der Sperre. Wenn Tasks auf die Lock warten, wird die nächste Task in der Warteschlange zur Ausführung eingeplant, und die Lock bleibt gesperrt. Andernfalls warten keine Tasks und die Lock wird entsperrt.[[26]](#footnote-26)23

## Klasse Pins

„Pins“ ist ein genialer Begriff für etwas, das historisch gesehen wie Nadeln aussah, heutzutage aber oft nicht mehr. Ein Pin ist ein mit dem Mikrocontroller verbundener leitender Bereich, über den die Kommunikation mit externen Peripheriegeräten stattfinden kann.

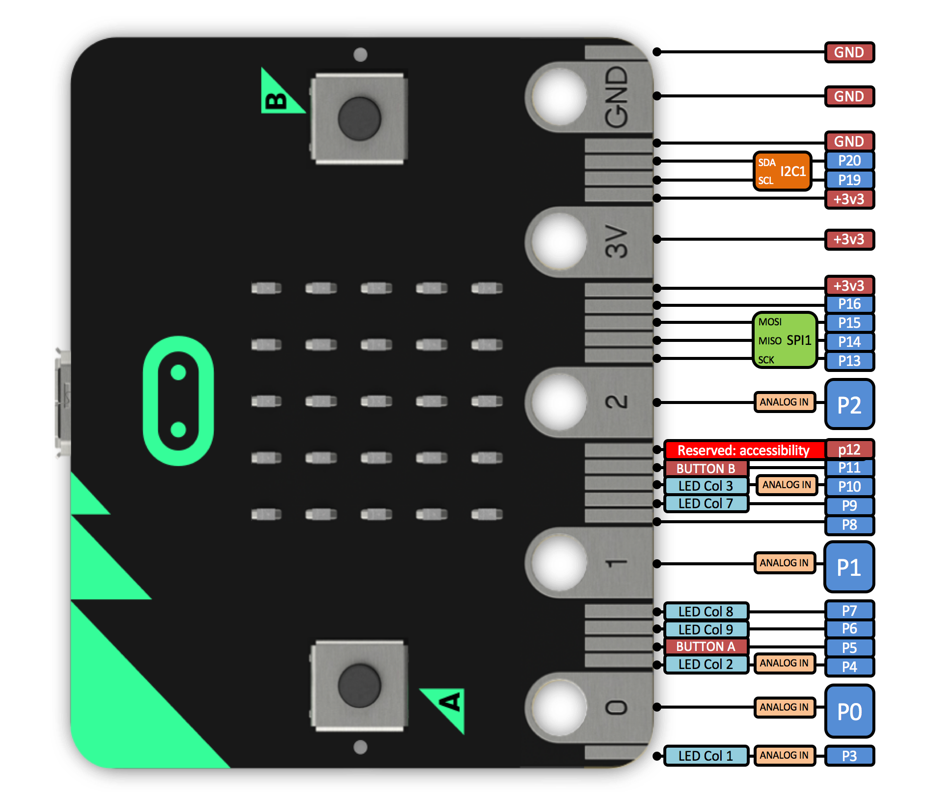


Abbildung 8 Pins on a Micro: bit[[27]](#footnote-27)24

Abbildung 8 zeigt eine genaue Nahaufnahme der "Pins" auf dem Micro: Bit: Sie sehen überhaupt nicht wie Pins aus. Die Pins bilden den rechten Rand des Boards. Und um die kleinen Pins verbinden zu können, verwendet man einen Randstecker, in den man Überbrückungskabel steckt, die mit externen Peripheriegeräten verbunden sind, oder eine Lochrasterplatine, auf der man externe Komponenten platzieren kann.

Die Pyboard gibt es in zwei Konfigurationen: ohne jegliche Stiftverbindungen (es gibt nur Löcher in der Leiterplatte, in die man solche Konnektoren einlötet) oder vorgelötet mit weiblichen Stiften, in die man Sprungkabel steckt, an die man die externen Peripheriegeräte anschließt. Beispielweise: Pyboard des Typs PYBv1.1, oder ESP8266/32.

Die Pins sind so benannt, dass man sie in unserem Code referenzieren kann. Referenzen auf Pins finden sich an verschiedenen Stellen, je nachdem, welche Version der Mikropython-Version man auf dem Gerät laufen hat. Wenn man das Pyboard verwendet, findet man sie im Pyboard-Modul. Das Original-Pyboard und die ESP8266/32-Ports von Mikropython haben eine Pin-Klasse, die Sie mit dem Namen des Pins und einer Vorstellung seiner Eigenschaften instanziieren. [[28]](#footnote-28)25

Die Namen sind in der Regel auf die Platine aufgedruckt, so dass es möglich ist, den Pin anzuschauen und herauszufinden, wie er heißt. Verschiedene Pins können für verschiedene Arten von Dingen verwendet werden. Einige Pins liefern einfach elektrischen Strom mit einer angegebenen Spannung, um ein externes Peripheriegerät zu versorgen.

Andere Pins dienen als Masse (oft als GND bezeichnet), was dem negativen Ende einer Batterie entspricht. Die Pins, die nur Strom liefern, und die mit Masse gekennzeichneten Pins stehen nicht unter Ihrer Kontrolle, da sie nur das tun, was ihr Name vermuten lässt.

Es sind die anderen Pins, die für uns interessanter sind, und sie sind möglicherweise in der Lage, verschiedene Dinge zu tun. Zum Beispiel können sie alle als digitale Pins dienen. man steuert sie entweder auf niedrig (0 V) oder hoch (sie erzeugen Strom bei der Versorgungsspannung des Boards, oft 3,3 V). Einige andere sind in der Lage, als analoge Pins zu arbeiten, die in der Lage sind, Signale zu senden oder zu empfangen, die nicht hoch oder niedrig sind, aber irgendwo zwischen den beiden Extremen liegen können. Gewöhnlich zeigt sich eine solche Wertverschiebung in Form von Spannungsunterschieden, die von einem Analog-Digital-Wandler (ADC) gelesen und in eine Zahl innerhalb eines bestimmten Bereichs umgewandelt werden. Die Analogausgabe wird von einem Digital-Analog-Wandler (DAC) erzeugt, der eine Zahl nimmt und sie in eine Spannungsdarstellung des Analogwerts umwandelt.[[29]](#footnote-29)25

Anwendungsmodell Bsp: Alle Board Pins sind als *pyb.Pin.board.name* vordefiniert

x1\_**pin** = pyb.**Pin**.board.X1

g = pyb.**Pin**(pyb.**Pin**.board.X1, pyb.**Pin**.IN)[[30]](#footnote-30)26

## Das Module Uarray

Dieses Modul implementiert eine Teilmenge des entsprechenden Python-Moduls. zum Verständnis ist es zuerst wichtig, auf die Python-Dokumentation zu referenzieren: array.

### Das Module Array

Dieses Modul definiert einen Objekt-Typ, der eine Reihe von Basis-Werten wie Zeichen, Ganzzahlen und Fließkommazahlen kompakt darstellen kann. Arrays sind Folgetypen und verhalten sich sehr ähnlich wie Listen, außer dass der in diesen Arrays gespeicherte Objekttyp eingeschränkt ist. Der Datentyp wird bei der Erstellung des Objekts mit Hilfe eines Typencodes angegeben, der aus einem einzigen Zeichen besteht. Die folgenden Typ-Codes sind definiert:[[31]](#footnote-31)27

| **Type code** | **C Type** | **Python Type** | **Minimum size in bytes** | **Notes** |
| --- | --- | --- | --- | --- |
| 'b' | signed char | int | 1 |  |
| 'B' | unsigned char | int | 1 |  |
| 'u' | Py\_UNICODE | Unicode character | 2 | (1) |
| 'h' | signed short | int | 2 |  |
| 'H' | unsigned short | int | 2 |  |
| 'i' | signed int | int | 2 |  |
| 'I' | unsigned int | int | 2 |  |
| 'l' | signed long | int | 4 |  |
| 'L' | unsigned long | int | 4 |  |
| 'q' | signed long long | int | 8 | (2) |
| 'Q' | unsigned long long | int | 8 | (2) |
| 'f' | float | float | 4 |  |
| 'd' | double | float | 8 |  |

Tabelle 1: Typ Code[[32]](#footnote-32)27

* Der Code vom Typ 'u' entspricht dem überflüssigen Unicode-Zeichen von Python (Py\_UNICODE, d.h. wchar\_t). Abhängig von der Plattform kann es 16 Bit oder 32 Bit sein.
* Die '*q*'- und *'Q'*-Typ Codes sind nur verfügbar, wenn der Plattform-C-Compiler, der zur Erstellung von Python verwendet wird, C, oder unter Windows \_\_int64 unterstützt.

Das Modul definiert den folgenden Typ:

Class *array*.array(typecode[, initializer])

Ein neues Array, dessen Elemente durch den Typ Code eingeschränkt sind und vom optionalen Initialisierungswert initialisiert werden, die eine Liste, ein Byte-ähnliches Objekt oder iterierbar über Elemente des entsprechenden Typs sein muss.

Wenn eine Liste oder Zeichenfolge angegeben wird, wird der Initialisierter an die Methode fromlist(), frombytes() oder fromunicode() des neuen Arrays übergeben, um dem Array die ersten Elemente hinzuzufügen. Andernfalls wird der iterierbare Initialisierter an die extend()-Methode übergeben.[[33]](#footnote-33)28

Class *uarray*.array(typecode [, iterable])

Erstellt ein Array mit Elementen eines bestimmten Typs. Der initiale Inhalt des Arrays wird durch Iteration angegeben. Wenn er nicht angegeben wird, wird ein leeres Array erstellt.[[34]](#footnote-34)29

# Framework und Umgebung

## Python

Python ist eine leistungsfähige und leicht zu erlernende Programmiersprache. Es verfügt über Datenstrukturen auf hoher Ebene und ermöglicht einen einfachen, aber effektiven Ansatz für die objektorientierte Programmierung. Aufgrund seiner eleganten Syntax, seiner dynamischen Schreibweise und seiner Interpretation ist Python eine ideale Sprache für die Skripterstellung und schnelle Anwendungsentwicklung in vielen Bereichen und auf den meisten Plattformen. [[35]](#footnote-35)31

Python ist bekannt dafür, eine reine und unkomplizierte Programmiersprache mit einer eindeutigen strukturierten Umgebung zu bieten. Die Programmierung ist nicht nur intuitiv, sondern auch verständlich und lesbar. Obwohl Python relativ unkompliziert ist, verfügt es über eine gute Kombinierbarkeit und eignet sich für komplexe Softwareprojekte. Aufgrund seiner expressiven und reduzierten Syntax können Programme mit nur ein paar Zeilen Code geschrieben werden und weisen keine Programmierfehler auf. Um der Klarheit und Eindeutigkeit zu dienen, verwendet Python nur sehr wenige Schlüsselwörter und verwendet eine Einrückung als strukturierendem Element.

Dadurch, dass Python eine angeblich vielschichtige Sprache ist, sind Programmierer nicht an einen spezifischen Programmiermodus beschränkt. Für unterschiedliche Tätigkeiten, kann der optimale Programmierstil gewählt werden. Python bietet die Möglichkeit, Python-Programme als individuelle Module in andere Sprachen zu implementieren.[[36]](#footnote-36)30

Der Python-Interpreter verfügt über eine umfangreiche Standardbibliothek und ist als Quellcode oder Binärdateien für alle wichtigen Plattformen frei verfügbar.

Der Python-Interpreter kann leicht um neue Funktionen und Datentypen erweitert werden, die in C oder C++ (oder einer anderen Sprache, die von C aus aufgerufen werden kann) implementiert sind. Python eignet sich auch als Erweiterungssprache für das Customizing von Anwendungen.

Die Standardbibliothek von Python ist sehr umfangreich und bietet eine Vielzahl von Möglichkeiten. Die Bibliothek enthält eingebaute Module (in C geschrieben), die den Zugriff auf Systemfunktionen wie Datei-E/A ermöglichen, die sonst für Python-Programmierer unzugänglich wären, sowie in Python geschriebene Module, die standardisierte Lösungen für viele Probleme bieten, die bei der täglichen Programmierung auftreten. Einige dieser Module sind explizit darauf ausgelegt, die Portierbarkeit von Python-Programmen zu fördern und zu verbessern, indem plattformspezifische Elemente in plattformneutrale APIs abstrahiert werden.[[37]](#footnote-37)31 In der Regel, enthalten die Installationsprogramme von Python für Windows und Mac die gesamte Standardbibliothek und fügen oft weitere Komponenten hinzu. Für Unix-Betriebssysteme wird Python typischerweise als eine Sammlung von Paketen zur Verfügung gestellt, so dass es notwendig sein kann, den vom Betriebssystem bereitgestellten Paketmanager zu verwenden, um einige optionale Komponenten zu erhalten.[[38]](#footnote-38)31

* **Entwicklungsumgebungen für Python**

Im Prinzip ist für Python keine Entwicklungsumgebung notwendig, da Python-Code nicht kompiliert werden muss und Skripte mit jedem Texteditor geschrieben werden können. Die Interpreter sind interaktiv und erlauben, mit den Möglichkeiten der Sprache zu spielen. Allgemeine Editoren, die von Programmierern wie Emacs oder Vim benutzt werden, sind für Python benutzbar. IDLE wird oft zur gleichen Zeit wie Python installiert. IDLE besteht aus einer Shell, einer Textumgebung und Debugging-Funktionen, ist aber keine Integrierte Entwicklungsumgebung (IDE). Beispiele für komplette Entwicklungsumgebungen ist PyCharm. Für große IDEs wie NetBeans, Eclipse oder Visual Studio gibt es Plug-Ins für Python. Grafische Benutzeroberflächen sind mit Hilfe verschiedener grafischer Toolkits relativ einfach zu erstellen.[[39]](#footnote-39)30

## Benutze Software

### Visual Studio Code

Visual Studio Code ist ein einfacher, aber leistungsstarker Quellcode-Editor, der auf unserem Desktop läuft und für Windows, MacOS und Linux verfügbar ist. Er wird mit integrierter Unterstützung für *JavaScript*, *TypeScript* und *Node.js* geliefert und verfügt über ein reichhaltiges Ökosystem von Erweiterungen für andere Sprachen (wie C++, C#, Java, Python, PHP, Go) und Laufzeiten (wie .NET und Unity).[[40]](#footnote-40)32

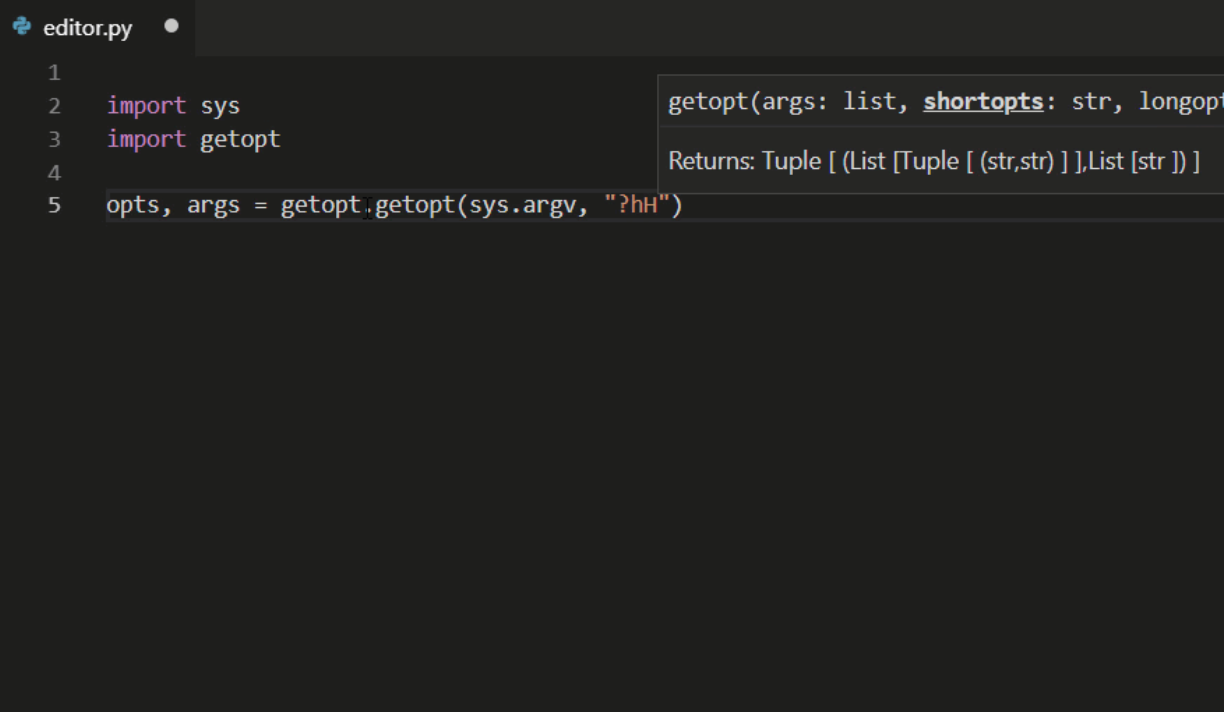


Abbildung 9 Python Programm mit Visual Studio[[41]](#footnote-41)33

VS Code ermöglicht zusätzliche programmierte Sprachen, Themen, Debugger, Befehle und mehr. Die wachsende Gemeinschaft von VS Code teilt ihre Geheimnisse, um den Workflow zu verbessern. [[42]](#footnote-42)32

Ich habe Visual Studio Code verwendet, um den Python-Code zu erstellen und die Kommunikation zwischen dem Computer und dem Pyboard über sein integriertes Terminal zu simulieren.

Visual Studio Code verfügt über ein integriertes Terminal, das uns den Zugriff auf das Terminal des Betriebssystems ermöglicht, das zur Erleichterung der Programmierung und Simulation verwendet wird.

Dank des Terminals können wir auf die versteckten Dateien und Ordner des Computers zugreifen, bzw. den Ordner zugreifen, der den Mikrocontroller enthält.

# Benutzte Materiellen

## Pyboard

(siehe 2.3.1\_\_i)

## Breadboard (Steckplatine)

Ein Breadboard/Steckplatine dient zur mechanischen Befestigung und elektrischen Verbindung von elektronischen Komponenten für Testschaltungen und Experimente.[[43]](#footnote-43)35

Um die verschiedenen Daten, die das Pyboard an den Computer senden muss, messen zu können, müssen wir eine Schaltung herstellen, die das Pyboard als Spannungsgenerator und einen Sensor als Empfänger enthält. Beide sind durch das Breadboard verbunden. Der Sensor sendet dann analoge Daten an das Pyboard, das diese in digitale Daten umwandelt. Das Breadboard kann dann als die zentrale Kommunikationseinheit zwischen dem Pyboard und dem Sensor betrachtet werden.

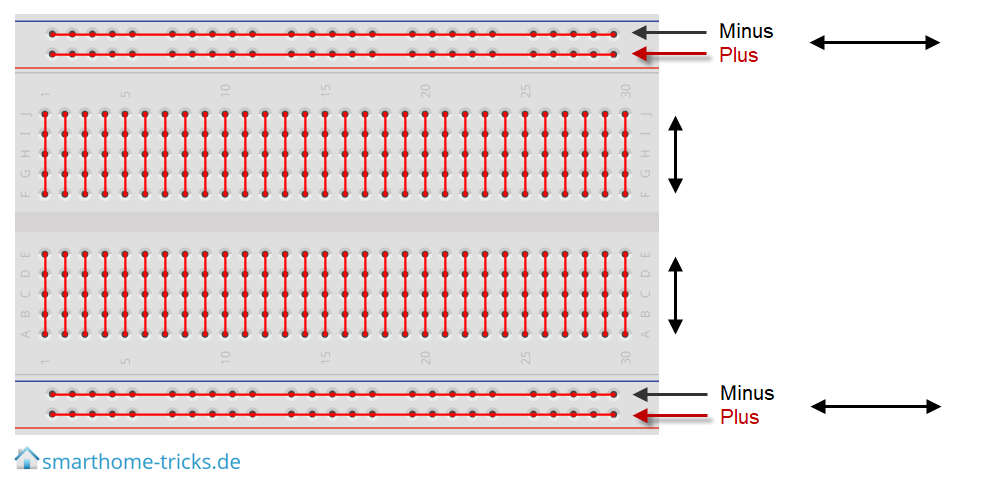


Abbildung 10 Breadboard[[44]](#footnote-44)34

## Potentiometer

## Ein Potentiometer[[45]](#footnote-45)36 ist ein elektrisches Widerstandsbauteil, dessen Widerstandswerte mechanisch (durch Drehen oder Bewegen) verändert werden können. Es hat mindestens drei Anschlüsse und wird hauptsächlich als kontinuierlich einstellbarer Spannungsteiler verwendet.

Seine verschiedenen Werte halfen uns, Tausende von Messungen an das Pyboard zu senden, dass sie dann an den Computer weiterleitete.

Diese Daten wurden durch den Computer in menschenlesbare Werte umgewandelt und modifiziert.

# Programmierung und Realisierung

In diesem Teil der Arbeit, geht es darum, die verschiedenen Schritte zu erläutern, die unternommen wurden, um das gewünschte Ergebnis zu erreichen.

Es wurde eine Kommunikation zwischen dem Computer und dem Mikrocontroller erstellt. Der Mikrocontroller bzw. der Pyboard empfangt die Daten vom Sensor und leitet die dem Computer weiter.

Dabei kann auch der Computer dem Mikrocontroller Befehle senden, ohne dass die Messungen gestört werden.

## Aufbau

Damit Tausende oder mehr Daten gesendet werden können, ist es in erster Linie wichtig, eine elektronische Montage mit den oben genannten Komponenten (**siehe 5**) herzustellen.

Aufbau der Schaltung:

Wenn das Kabel mit Pin 3v3 an den positiven Anschluss des Breadboard angeschlossen wird, ist es wichtig, Pin 3v3 zu aktivieren, bevor das Programm mit der Funktion pyb.Pin("EN\_3V3").on() starten wird. Verbindet man dann die Masse mit dem Minuspol des Breadboard und verbindet das Potentiometer mit dem Breadboard. darauf muss geachtet werden, dass die Anschlüsse parallel und nicht in Reihe geschaltet werden, da sonst die Gefahr eines Kurzschlusses besteht. Verbinden Sie die Anschlüsse des Potentiometers mit den Anschlüssen des Pyboard.

Um die Variation der Potentiometer-Widerstandswerte besser sichtbar zu machen, wurde auf der Schaltung eine LED in Reihe hinzugefügt.

Nachdem die Montage abgeschlossen ist, ist es an der Zeit, das Pyboard an den Computer und das Programm computer.py im Terminal anzuschließen.

Es ist für uns wichtig, das richtige Boot-Modus zu wählen, sonst wird die Shell die Ausführung der Datei main.py verhindern, und wir werden immer eine Fehlermeldung erhalten, wenn der Computer einen Befehl senden wird.

## Asynchrone Serielle Kommunikation zwischen Pyboard und Computer

* Auf dem Computer

Zuerst sollte ich das Serial-Modul der Pyserial-Bibliothek importieren, damit der Computer den Pyboard erkennen kann. Als Parameters haben wir den Port „dev/tty.\*“; die Baudrate „115200“ und das Timeout „3“ gewählt. Die „115200“ ist die meinem Pyboard entsprechende Frequenz, und „3“ entspricht der Wartezeit in Sekunde. Der Port „dev/tty.\*“ ist unser Gerät bzw. der Pyboard.

* Auf dem Pyboard

Dieser Schritt besteht darin, die USB\_VCP-Klasse aus der Pyb-Bibliothek zu importieren, die eine virtuelle und asynchrone Kommunikation mit dem Computer ermöglicht.

Diese virtuelle Kommunikation wird die Aktion des vorinstallierten REPL verhindern. Als Folge wird die Mikropython-Shell nicht mehr verfügbar sein, da die vom REPL ermöglich ist.

## Dateien einlesen und bearbeiten

Die ADC-Klasse des Pyb-Moduls ist erforderlich, damit das Pyboard die Informationen vom Sensor oder Potentiometer einlesen kann. Diese Informationen bzw. Daten werden mit der Funktion adc.read() gelesen. Anschließend müssen diese Daten mit Hilfe von Verarbeitungsfunktionen (Funktion der Konvertierung der Messwerte in Volt), die auf das Pyboard geschrieben werden, in ihre jeweiligen Einheiten (z.B. Volt) umgewandelt werden. Es ist möglich, Informationen von den verschiedenen an das Pyboard angeschlossenen Sensoren einzulesen und zu bearbeiten.

## Befehle senden und empfangen

Damit das Programm kontinuierlich läuft, ist es wichtig, auf beiden Seiten bzw. der Computer und das Pyboard eine unendliche Schleife zu erzeugen. Und diese Schleifen können mit der Break-Funktion gestoppt werden. Wenn das Pyboard die Informationen vom Potentiometer oder Sensor erhalten hat, wandelt es die Informationen in eine Zeichenfolge um und fügt Zeilen-schub hinzu. Diese Verarbeitung wird das Senden und Empfangen von Daten erleichtern. Das Senden der vom Pyboard verarbeiteten Daten erfolgt mit der Funktion "awrite()". Der Computer empfängt diese Daten mit der "readline()"- Funktion, wandelt sie in eine Zeichenkette um und entfernt die Zeilen-schub, die das Senden erleichtert haben. Während der Datenübertragung, wird der Computer auch in der Lage sein, Befehle an das Pyboard zu senden, indem er die "write"-Funktion benutzt, um die Art und Weise zu steuern, wie die Informationen gesendet werden, und um anzuzeigen, wann die Übertragung beendet ist.

PS: Wenn der Computer einen Befehl an das Pyboard sendet, ist es wichtig, die Daten in "byte" mit dem Symbol "b" den Anführungszeichen zu kodieren oder mit der Funktion Encode("utf-8") in "utf-8" zu kodieren, da sonst der Befehl nicht gesendet werden kann und eine Fehlermeldung angezeigt wird. Um die vom Computer empfangenen Daten zu speichern, wurde ein Array vom Typ "int" erstellt.

Um sicherzustellen, dass die Daten in geordneter Reihenfolge gesendet werden und dass die vom Computer an das Pyboard gesendeten Befehle die Datenübertragung nicht stören, kommt die Synchronisation im Einsatz. Dies geschieht über die Bibliothek "Uasyncio". Vor der Synchronisierung, muss eine Variable erstellt werden, die den Ablauf jeder Aufgabe steuert. Diese Variable ist die "Lock ()"-Klasse der "Uasyncio"-Bibliothek. Jedem unter "Uasyncio" geschriebenen Befehl folgt immer das "await"-Argument. Das Starten eines Tasks, der alle anderen Tasks blockiert, erfolgt unter anderem mit "lock.acquire()", wobei der Task den Schlüssel erwirbt und als einziger Zugriff auf den Prozessor hat. Wenn die Aufgabe beendet ist, setzt er den Schlüssel mit der Funktion "lock.release" zurück und er erlaubt einer anderen Task, den Schlüssel zu erwerben, und pausiert für eine beliebige, vom Programmierer festgelegte Zeit mit der Funktion "Uasyncio.sleep(time)". Dieser Prozess wird fortgesetzt, bis jeder Befehl abgeschlossen ist.

Damit eine Aufgabe die Fakultät hat, einen Schlüssel zu erwerben oder freizugeben, wird die Funktion "Uasyncio.create\_task()" in der "main"-Funktion verwendet.

Hinweis: Bei der Definition der "main"-Funktion muss "async" vorangestellt werden. Um das Programm zu starten, verwendet man die Funktion "Uasyncio.run(main)".

Nach dem Schreiben und Speichern des Programms, trennt und verbindet man das Programm wieder und führt die Datei "computer.py" im Terminal mit dem Befehl "python3 computer.py" aus.

# 

# 

# Hauptteil 2: Grundlagen und Methodik

## Theoretischer Bezugsrahmen

## Methodische Vorgehensweise

# Hauptteil 3: Empirische Untersuchung

## Analysen

## Interpretation der Ergebnisse

# Zusammenfassung

Hier werden die wesentlichen Ergebnisse der Arbeit zusammengefasst. Insbesondere wird hier der Bogen zu der in der Einleitung formulierten Forschungshypothese geschlagen. Hier können auch die Ergebnisse kritisch betrachtet und in den Zusammenhang der Forschungsentwicklung eingeordnet werden (Ausblick/Desiderata).

Textbausteine/Formulierungsvorschläge für das Fazit können Sie hier herunterladen:  
<http://lektorat-korrekturlesen.de/wp-content/uploads/2017/02/Textbausteine-wiss.-Arbeit-1.docx>

# Literaturverzeichnis

Überschrift nicht nummeriert.

Das Literaturverzeichnis folgt in der Hausarbeit nach dem Textteil. Es enthält sämtliche in der Hausarbeit benutzte Literatur, d. h. die Texte, die zur Stützung der Argumentation entweder wörtlich oder sinngemäß zitiert werden. Alibi-Nennungen von nicht herangezogener Literatur, um Gelehrsamkeit zu demonstrieren, gelten als verpönt.

Die im Beispielverzeichnis angeführten Titel beziehen sich auf aktuelle Literatur zum wissenschaftlichen Arbeiten. Zitierstil: Citavi-Basis-Stil.

**Brink**, Alfred (2013): Anfertigung wissenschaftlicher Arbeiten. Ein prozessorientierter Leitfaden zur Erstellung von Bachelor-, Master- und Diplomarbeiten. 5., überarbeitete und aktualisierte Aufl. 2013. Dordrecht: Springer.   
Online verfügbar unter http://gbv.eblib.com/patron/FullRecord.aspx?p=1317754.

**Eco**, Umberto; Schick, Walter (2010): Wie man eine wissenschaftliche Abschlußarbeit schreibt. Doktor-, Diplom- und Magisterarbeit in den Geistes- und Sozialwissenschaften. 13., unveränd. Aufl. der dt. Ausg. Wien: Facultas Univ.-Verl. (UTB Schlüsselkompetenzen, 1512).   
Online verfügbar unter http://www.gbv.de/dms/faz-rez/FD120010510877043.pdf.

**Esselborn-Krumbiegel**, Helga (2014): Richtig wissenschaftlich schreiben. Wissenschaftssprache in Regeln und Übungen. 3., durchges. Aufl. Paderborn: Schöningh (Uni-Tipps, 3429).   
Online verfügbar unter http://www.utb-studi-e-book.de/9783838541570.

**Hirsch-Weber**, Andreas; **Scherer**, Stefan (2015): Wissenschaftliches Schreiben und Abschlussarbeit in Naturwissenschaften und Ingenieurwissenschaften. Grundlagen – Praxisbeispiele – Übungen. 1. Aufl. Stuttgart, Stuttgart: UTB.

**Karmasin**, Matthias; **Ribing**, Rainer (2014): Die Gestaltung wissenschaftlicher Arbeiten. Ein Leitfaden für Seminararbeiten, Bachelor-, Master- und Magisterarbeiten sowie Dissertationen. 8., aktualisierte Aufl. Wien: facultas.wuv (UTB Schlüsselkompetenzen, 2774).   
Online verfügbar unter http://www.utb-studi-e-book.de/9783838542591.

**Kornmeier**, Martin (2013): Wissenschaftlich schreiben leicht gemacht. Für Bachelor, Master und Dissertation. 6., aktualisierte Aufl. Bern, Stuttgart: Haupt; UTB (utb-studi-e-book, 3154).   
Online verfügbar unter http://www.utb-studi-e-book.de/9783838540733.

**Manschwetus, Uwe (2016):** Ratgeber wissenschaftliches Arbeiten. Leicht verständliche Anleitung für das Schreiben wissenschaftlicher Texte im Studium. Lüneburg: Thurm Wissenschaftsverlag.

**Oehlrich**, Marcus (2015): Wissenschaftliches Arbeiten und Schreiben. Schritt für Schritt zur Bachelor- und Master-Thesis in den Wirtschaftswissenschaften. Berlin: Springer Gabler. Online verfügbar unter http://dx.doi.org/10.1007/978-3-662-44099-5.

**Oertner**, Monika; **St. John**, Ilona; **Thelen**, Gabriele (2014): Wissenschaftlich schreiben. Ein Praxisbuch für Schreibtrainer und Studierende. Paderborn, Stuttgart: Fink; UTB (UTB Schlüsselkompetenzen, 8569).   
Online verfügbar unter http://www.utb-studi-e-book.de/9783838585697.

**Sesink**, Werner (2012): Einführung in das wissenschaftliche Arbeiten. Inklusive E-Learning, Web-Recherche, digitale Präsentation u. a. 9., aktualisierte Aufl. München: Oldenbourg.

**Weber**, Daniela (2014): Die erfolgreiche Abschlussarbeit für Dummies. 2., aktualisierte Aufl. Weinheim: Wiley-VCH

**Weber**, Daniela (2015): Wissenschaftliches Arbeiten für Wirtschaftswissenschaftler. Hoboken: Wiley.   
Online verfügbar unter http://gbv.eblib.com/patron/FullRecord.aspx?p=2059125.

**Wolfsberger**, Judith (2010): Frei geschrieben. Mut, Freiheit & Strategie für wissenschaftliche Abschlussarbeiten. 3. Aufl. Wien: Böhlau (UTB Schlüsselkompetenzen, 3218).

# Anhänge

Überschrift nicht nummeriert

## Anhang 1

## Anhang 2

# Eidesstattliche Versicherung

Überschrift nicht nummeriert.

Wissenschaftliche Arbeiten müssen selbstständig erbracht und verfasst werden.

Die meisten Prüfungsordnungen schreiben einen Mustertext vor, der am Ende der Arbeit beigeheftet wird. Er ist zu datieren und handschriftlich zu unterschreiben.

Kann man nach Abgabe einer wissenschaftlichen Arbeit mit eidesstattlicher Erklärung einen Verstoß nachweisen, droht nicht nur die Aberkennung der Prüfungsleistung, sondern auch ein Verfahren wegen Täuschung bzw. Betrug (§ 156 StGB).

Beispiel für eine eidesstattliche Versicherung:

*„Hiermit erkläre ich, (Name), an Eides statt, dass ich die vorliegende Bachelorarbeit bzw. Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.“*

Ort, Datum, Unterschrift

# Stichwortverzeichnis

Überschrift nicht nummeriert.

Das Stichwortverzeichnis ist bei der Abgabe von Abschlussarbeiten eher nicht üblich. Es wird meist bei der Druckfassung ergänzt.

**Brink**, Alfred VII

**Eco**, Umberto VII

**Esselborn-Krumbiegel**, Helga VII

**Hirsch-Weber**, Andreas VII

**Karmasin**, Matthias VII

**Kornmeier**, Martin VII

**Manschwetus, Uwe** VIII

**Oehlrich**, Marcus VIII

**Oertner**, Monika VIII

**Ribing**, Rainer VII

**Scherer**, Stefan VII

**Sesink**, Werner VIII

**St. John**, Ilona VIII

**Thelen**, Gabriele VIII

**Weber**, Daniela (2014) VIII

**Weber**, Daniela (2015) VIII

**Wolfsberger**, Judith VIII

1. (Laser & Co. Solutions GmbH, 2020) [↑](#footnote-ref-1)
2. (Laser & Co. Solutions GmbH, 2020) [↑](#footnote-ref-2)
3. (Laser & Co. Solutions GmbH, 2020) [↑](#footnote-ref-3)
4. 4 (Uwe Brinkschulte, 2010) [↑](#footnote-ref-4)
5. 6 (George, 2014) [↑](#footnote-ref-5)
6. 5 (Tollervey, Programming with Micropython , 2017) [↑](#footnote-ref-6)
7. 7 (George, 2014) [↑](#footnote-ref-7)
8. 8 REPL steht für Read Evaluate Print Loop (Print, Lesen und Auswerten) und ist der Name der interaktiven Mikropython-Eingabeaufforderung, auf die Sie über WiPy zugreifen können. [↑](#footnote-ref-8)
9. 9 (Tollervey, Programming with Micropython , 2017) [↑](#footnote-ref-9)
10. 10 (CO, 2020) [↑](#footnote-ref-10)
11. 11 (Tollervey, Programming with Micropython, 2017. Seite17-19) [↑](#footnote-ref-11)
12. 12 (Tollervey, Programming with Micropython, 2017, S. 20) (Liechti, 2020) [↑](#footnote-ref-12)
13. 13 (Liechti, 2020) [↑](#footnote-ref-13)
14. 14 (Tollervey, Programming with Micropython, 2017, S. 104 & 105) [↑](#footnote-ref-14)
15. 15 (Damien P.George, 2020) [↑](#footnote-ref-15)
16. 15 (Damien P.George, 2020) [↑](#footnote-ref-16)
17. 16 (Damien P. George, 2020) [↑](#footnote-ref-17)
18. 17 (Damien P.George, 2020) [↑](#footnote-ref-18)
19. 18 (Foundation, Python , 2020) [↑](#footnote-ref-19)
20. 19 (Foundation, Python, 2020) [↑](#footnote-ref-20)
21. 20 (Foundation, Python , 2020) [↑](#footnote-ref-21)
22. [↑](#footnote-ref-22)
23. 21 (Foundation, Python , 2020) [↑](#footnote-ref-23)
24. 22 (Foundation, Python , 2020) [↑](#footnote-ref-24)
25. [↑](#footnote-ref-25)
26. 23 (Paul Sokolovsky, 2020) [↑](#footnote-ref-26)
27. 24 (affiliates), 2020) [↑](#footnote-ref-27)
28. 25 (Tollervey, Programming with Micropython, 2017, S. 101-103) [↑](#footnote-ref-28)
29. 25 [↑](#footnote-ref-29)
30. 26 (Damien P. George P. S., 2020) [↑](#footnote-ref-30)
31. 27 (Foundation, Python » 3.5.9 Documentation , 2020) [↑](#footnote-ref-31)
32. 27 (Foundation, Python » 3.5.9 Documentation, 2020) [↑](#footnote-ref-32)
33. 28 (Foundation, Python » 3.5.9 Documentation, 2020) [↑](#footnote-ref-33)
34. 29 (Damien P. George P. S., MicroPython, 2020) [↑](#footnote-ref-34)
35. 31 (Foundation, Python , 2020) [↑](#footnote-ref-35)
36. 30 (Group, 2020) [↑](#footnote-ref-36)
37. 31 (Foundation, Python , 2020) [↑](#footnote-ref-37)
38. 31 (Foundation, Python , 2020) [↑](#footnote-ref-38)
39. 30 (Group, 2020) [↑](#footnote-ref-39)
40. 32 (Microsoft, Visual Studio Code , 2020 ) [↑](#footnote-ref-40)
41. 33 (Microsoft, 2020) [↑](#footnote-ref-41)
42. 32 (Microsoft, Visual Studio Code , 2020 ) [↑](#footnote-ref-42)
43. 35 (Wikipedia, 2020) [↑](#footnote-ref-43)
44. 34 (smarthome-tricks.de, 2020) [↑](#footnote-ref-44)
45. 36 (Wikipedia, Wikipedia, 2020) [↑](#footnote-ref-45)