Einführung in die Programmierung mit (Micro)Python

Mikrocontroller-spezifisches Programmieren in Micropython auf dem Pi Pico

© G. Meng-Wacker, Velbert 2025, Version 1.0.1

I Vorwort

Es gibt unzählige Einführungen in die Programmierung und in einzelne Programmiersprachen, warum also noch eine? Zum einen macht es mir Spaß, meine langjährigen Erfahrungen einzubringen. Und zum anderen habe ich noch kein Buch zu Micropython gefunden, das eine einfache, praxisbezogene Hinführung zur Programmierung in Python und zu mikrocontroller-gerechter Programmierung in deutsch liefert.

Wie schon mein Buch „Eine kleine Einführung in die Elektronik“ entstand auch dieses aus dem Wechselspiel zwischen Vorbereitung und Durchführung eines Workshops in den Räumlichkeiten der Stadtbibliothek Velbert.

Ich beschränke mich hardwareseitig auf den Pi Pico[1] und verwende bewusst eine simple IDE, den Thonny-Editor[2]. Dennoch behandle ich so anspruchsvolle Themen wie Objekt Orientierte Programmierung (OOP) und Multitasking (die kooperative Variante mit AsyncIO).

Fast alle Module für die Unterstützung der Mikrocontroller-Hardware sind als Klassen implementiert; wir müssen also so oder so Instanzen erzeugen, mit Methoden arbeiten usw. Das Verständnis für zumindest die Grundlagen der OOP hilft daher sehr. Im gesamten Buch verwende ich „Funktion“ und „Methode“ synonym; nur da, wo es darauf ankommt, differenziere ich die Begriffe.

Während die asynchrone Verarbeitung mit Interrupts in C (oder Arduino) Projekten üblich ist, favorisiere ich echtes Multitasking für Micropython. Interrupts, die durchaus vorhanden sind, machen den Programmcode schlecht lesbar, leiden unter Race Condition Problemen und in Micropython schlechter Latenz[3]. Die AsyncIO Implementierung ist so schnell, dass Reaktionen im Millisekunden Bereich möglich sind. Zudem stellt die Programmierung mit AsyncIO m.E. deutlich geringere Anforderungen an den Programmierer als das hantieren mit Interrupts oder auch die Nutzung von Threading.

Wem einige der genannten Begriffe unbekannt sind: Keine Sorge, alles wird im Buch erklärt!

---

1. Der Pi Pico stammt von der Raspberry Pi Foundation: https://www.raspberrypi.org. Leider auf englisch. Die Seite https://tutorials-raspberrypi.de/raspberry-pi-pico-mikrocontroller-programmieren/ liefert Informationen auf deutsch.
2. Im Elektronik Workshop habe ich zunächst den Mu Editor verwendet, bin aber aus technischen Gründen (Mu läuft nicht auf dem Raspberry Pi) und einer gewissen Absturzfreudigkeit zu Thonny gewechselt.
3. Latenz nennt man die zeitliche Verzögerung zwischen Auslöser und Reaktion. Mit der Weiterentwicklung von Micropython verbessert sich dieser Aspekt aber: Meine letzten Messungen zeigen bessere Werte als frühere Versionen. Mit ca. 90 µs (90 millionestel Sekunden) sind sie dennoch wesentlich schlechter als native Interrupts.

Inhaltsverzeichnis

I Vorwort....2
II Erste Schritte....6
1 Mikrocontroller Hardware....6
2 Micropython installieren....9
3 Entwicklungsumgebungen....9
3.1.1 Mu....10
3.1.2 Thonny....11
3.1.3 Vscode....11
3.1.4 Was fehlt....11
4 Ein „Anfänger“ ist einfach jemand, der anfängt!....11
5 Grundelemente von Programmiersprachen....14
5.1 Was ist eigentlich Programmieren?....14
5.2 Wie Python intern arbeitet....16
5.3 Von Variablen, Bedingungen und Schleifen....17
5.3.1 Variablen....17
5.3.2 # Kein Kommentar....18
5.3.3 Komplexere Datentypen....19
5.3.3.1 Listen....19
5.3.3.2 Schlüssel – Wert Paare: Dictionaries....20
5.3.3.3 Sogar komplex....21
5.3.3.4 Noch komplizierter geht auch....21
5.3.3.5 Un-dynamisch....22
5.3.4 Mehrfachzuweisungen und Slices....22
5.3.5 Lauter Klammern....24
5.3.6 Schleifen und Entscheidungen....25
5.3.7 Operatoren....27
5.3.8 Änderbar oder unveränderbar....27
5.4 „Prozeduren“, Funktionen und Methoden....28
5.4.1 Funktions- und Methodenparameter....29
5.4.2 Variable Anzahl von Argumenten....30
5.4.3 Schlüsselwort-Parameter....31
5.4.4 Von Werten, Referenzen und Seiteneffekten....32
5.4.5 (Zurück)geben ist seliger denn nehmen....36
5.5 Schönere Ausgaben....37
5.6 Von Räumen und Sichtbarkeit....38
5.7 Feinheiten....39
5.7.1 Anfang und Ende....39
5.7.2 Trennen und Zusammenfügen....39
5.7.3 Der Walross-Operator....40
5.7.4 Das Wörtchen „with“: der Kontextmanager....41
5.7.5 Iteratoren – die Unendlichkeit handhaben....41
5.8 Funktionale Programmierung....42
5.9 Callbacks und anonyme Funktionen....43
5.10 Dateioperationen – aber auf dem Pi Pico?....45
5.11 (Micro)pythons Helferlein....47
5.11.1 Die Funktion type....47

5.11.2 Die Funktion dir....48
5.11.3 Hilfe! Help!....49
5.12 Module und Pakete....49
5.12.1 Mal so – mal so....50
5.13 Dynamischer Speicher und der Müllsammler....51
5.14 Logisch?....52
5.15 Das muss hübscher werden: Decoratoren....53
III OOP – Objekt Orientierte Programmierung ganz pragmatisch....57
1 Eigenschaften von OOP....57
2 OOP – Vererbung....59
3 Klassenvariablen und -Methoden....61
4 Vererbung anwenden....61
5 Eine andere Anwendung von Vererbung....63
6 Eine besondere Art des Methodenaufrufs....64
7 Einfach magisch!....65
8 Ein Iterator im Eigenbau....66
9 Geheime Punkt-Sache....67
IV Typisch Mikrocontroller!....68
1 Das Tor zur Welt – Sensoren....68
1.1 Digital....68
1.2 Analog....70
1.2.1 Was ist ein Potentiometer?....71
1.2.2 Eine Benutzersteuerung....71
2 Aktoren....73
2.1 Ein Beispiel für einen Aktor: der Servo-Motor....74
2.1.1 Ein Exkurs in die Elektronik....75
2.1.2 Eine Servo-Klasse....76
2.2 Bis an die Grenze: ein PWM Experiment....78
3 Kommunikation ist alles....81
4 Von Mensch zu Mikrocontroller....83
4.1 Ein Taster, viele Möglichkeiten....83
4.2 Verprellt....84
4.3 Probieren geht über studieren....84
4.4 Der Drehgeber....85
4.5 Text und Bild....86
5 Wie viele GPIOs braucht ein Mikrocontroller?....87
6 Im Takt....88
7 Wenn es mal hängt …....89
V Netzwerke....90
1 Was ist ein Netzwerk?....90
2 Standarddienste....93
2.1 DHCP....93
2.2 DNS....94
2.3 NTP....94
2.4 Weitere Dienste....95
3 Die Säulen des Internet....95
4 Der Heim-Router....97
5 Ein Exkurs: IPv6....98
6 Das „W“ hinter Pi Pico....98
7 Gibt es es hier WLAN?....99

8 Eine WLAN Verbindung aufbauen....99
9 Steckdosen....102
10 Licht an – Licht aus!....103
11 Was ist eigentlich IoT?....105
11.1 MQTT....106
11.2 MQTT Spezial....108
11.3 Öffentliche MQTT Dienste....109
VI Warten ist doof – von Interrupts und Multitasking....110
1 Interrupts – asynchrone Unterbrechungen....110
2 Multitasking....112
2.1 AsyncIO....113
2.2 Entwickeln mit AsyncIO....115
VII Fehler, Fehler, Fehler!....116
VIII Taster - zum Zweiten....118
1 Zustände oder Flanken....118
2 Stopp-Uhr....119
3 Status Ermittlung....121
4 Eine Taster-Auswertungsklasse....122
5 Was sie können soll....123
6 Zurück in die Gegenwart....123
7 Und Prellen?....124
8 Ja wie denn nun?....125
9 Das zweite Mal schreiben....127
IX Debuggen....128
X Der Neue....129
XI Alternative Sprachen und Entwicklungsumgebungen....130
XII Anhang....131
1 Automatisiertes Cross-Kompilieren und Übertragen auf den Mikrocontroller....131
1.1 Für Windows als Batch Datei (Endung „bat“)....131
1.2 Für Linux....131
XIII Glossar....134

II Erste Schritte

Wer noch nie mit einem Mikrocontroller zu tun hatte oder noch nie programmiert hat, wird nach dem Vorwort vielleicht denken, dass das Alles zu schwer sei. Bevor die Flinte gleich im Korn landet, möchte ich erst mal Mut machen: Gerade eine Sprache wie Python macht erste Schritte leichter. Warum? Weil man ganz viel direkt auf der sogenannten Kommandozeile ausprobieren kann. Denn im Gegensatz zur Entwicklung mit dem Arduino, einer sehr populären Entwicklungsplattform, mit der man in einer C / C++ ähnlichen Sprache arbeitet, brauche ich ein Programm nicht erst kompilieren, dann auf den Mikrocontroller übertragen und dann zu sehen was es macht, sondern kann interaktiv mit dem Controller „sprechen“ und seine Antworten sofort sehen. Beim Lesen des Buches empfiehlt es sich, den Pi Pico an den Rechner anzuschließen, was über den USB Anschluss ganz leicht geht, und eine IDE zu starten. Das wird gleich noch näher erläutert. Dann kann man die Beispiele parallel selbst ausprobieren und ein Gefühl für die Arbeit mit dem Mikrocontroller entwickeln. Aber der Reihe nach.

1 Mikrocontroller Hardware

Bevor wir gleich in die Programmierung in Python einsteigen, möchte ich noch darauf eingehen, was den Unterschied zwischen einem PC und einem Mikrocontroller ausmacht.

Ein Mikrocontroller ist klein, aber das alleine ist noch nicht ausschlaggebend. Ein heutiges Smartphone ist auch klein und hat doch mehr Power als ein Großrechner aus den 80er Jahren. Ein Mikrocontroller wurde aber ganz anders und für andere Zwecke entwickelt: Er soll „embedded“, eingebettet in andere Hardware wie Autos, einen Thermomix[4], die Stereoanlage, die Waschmaschine usw. laufen. Dafür braucht er bestimmte Eigenschaften, andere nicht oder weniger. In einem PC werkelt eine CPU (Central Processing Unit); die gibt es in einem Mikrocontroller auch. Daneben existieren aber Hardware-Teile, die, wenn überhaupt, in einem PC auf z.B. Steckkarten ausgelagert sind: serielle Schnittstellen, Zeitgeber, ADC, DAC (Umwandler von analog zu digital und anders herum), I2C, ISP (Schnittstellen zur Kommunikation mit anderen Hardware-Modulen) und vieles mehr. Ziel des Designs eines Mikrocontrollers ist es, ihn so autonom und leistungsfähig wie nötig (und so preiswert wie möglich!) für seine Ziel-Anwendung zu machen. Wenn das eine elektrische Zahnbürste ist, reicht eine 4 Bit CPU und ein Akku-Management.

Unser Pi Pico erscheint da schon überdimensioniert. Eigentlich handelt es dabei aber auch um eine Entwicklungsplattform. Wir profitieren davon,weil er so praktisch für jedes Projekt ausreicht.[5]

Auf einem PC läuft ein Betriebssystem, ob Windows, MacOS, Linux. Ein Mikrocontroller braucht so etwas nicht; hier läuft quasi direkt ein Programm. Durch Micropython kommt aber der Python Interpreter hinzu, der nun wie ein Mini-Betriebssystem arbeitet und im REPL[6] sogar interaktives Arbeiten ermöglicht.

Auch kein Alleinstellungsmerkmal eines Mikrocontrollers sind Interrupts[7], Timer etc. Sie werden aber hier nicht vom Betriebssystem verwendet, sondern fallen direkt in die Verantwortung des

---

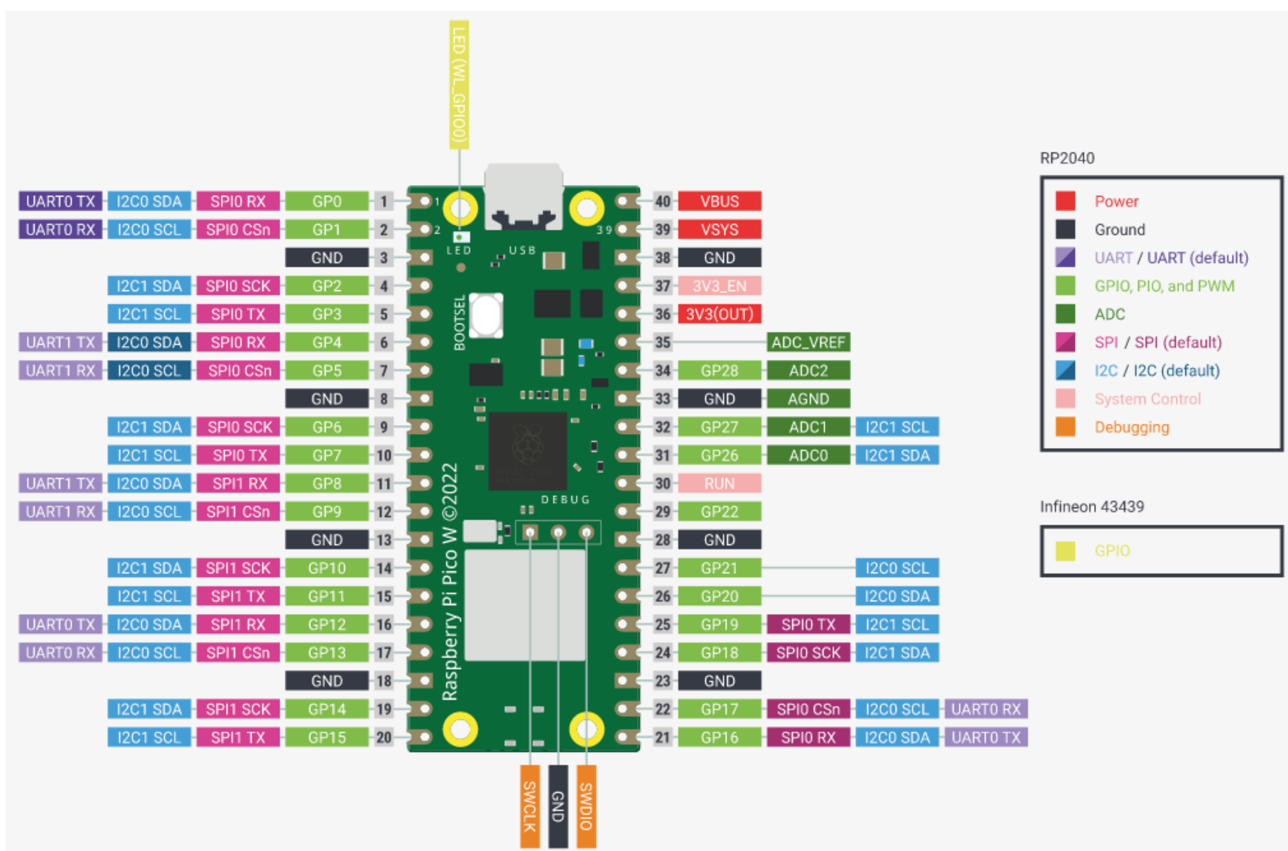
4 Siehe Kapitel „Was ist eigentlich Programmieren?“

5 Bei Batterie betriebenen Anwendungen, wo es auf jedes Zehntel mA ankommt, würde man sich gegebenenfalls für einen anderen Mikrocontroller entscheiden.

6 REPL steht für Read-Eval-Print Loop, Lesen-Ausführen-Ausgeben-Schleife.

Programmierers. Denn anders als innerhalb eines Betriebssystems ist das gleichzeitige Ausführen von mehreren Programmen oder Programmteilen nur durch bestimmte Programmiertechniken erzielbar. Über „Multitasking“, wie der Oberbegriff für solche Techniken heißt, wird weiter unten noch ausführlich gesprochen. Man muss aber nicht immer gleich Multitasking verwenden. Ein Mikrocontroller verfügt oft über Hardwareeinheiten, die spezielle, aber oft benötigte autonome Funktionen bereitstellen. Hier sei die PWM genannt; PWM steht für Puls-Weiten-Modulation und kann z.B. dazu benutzt werden, eine LED zu Dimmen. Die LED wird, schneller als das Auge es wahrnehmen kann, ein- und ausgeschaltet. Bei 50% an hat die LED die halbe Helligkeit, bei 100% an die maximale. Anstatt dies nun einem Python Programm zu überlassen, können wir die PWM Hardwareeinheit[8] des Pi Pico einmalig so programmieren, dass das gewünschte Tastverhältnis an einem Pin ausgeben wird. Danach schaltet der Pin automatisch an und aus, während unser Programm weiterarbeitet. Nur bei Änderungen am Tastverhältnis oder der Schaltfrequenz müssen wir wieder eingreifen.

Schauen wir uns die Hardware des Pi Pico einmal näher an.



In der offiziellen Dokumentation der Raspberry Pi Foundation (https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html) finden wir folgende Aufzählung, die ich hier frei übersetzt habe:

- RP2040 Microcontroller Chip, von der Raspberry Pi Foundaton entwickelt in England
- Dual-Core Arm Cortex M0+ Processor, 32 Bit, flexibler CPU-Takt Generator bis 133 MHz

7 Ein Interrupt wird durch Hardware, z.B. einer bestimmten Spannung an einem GPIO Pin, ausgelöst; er unterbricht das laufende Programm an beliebiger Stelle und führt dann ein – kleines – Programmteil aus.

8 Die Hardware-PWM hat (für den Pi Pico) eine Einschränkung: Die Frequenz aller PWM Kanäle ist gleich. Beim ESP32 zum Beispiel ist das nicht der Fall.

- 264kB SRAM (flüchtiger Arbeitsspeicher) und 2MB Flash Speicher
- USB 1.1 mit sowohl Geräte- als auch Host-Modus
- Verschiedene Stromspar-Modi
- Im Host-Modus verhält sich der Pi Pico wie ein USB-Speicher und kann darüber programmiert werden. So kommt das Micropython auf den Pi Pico.
- 26 × GPIO Pins, das sind die elektrischen Ein- und Ausgänge des Mikrocontrollers
- 2 × SPI, 2 × I2C, 2 × UART, 3 × 12-bit ADC, 16 × hardwaremäßige PWM Kanäle
- Präzise Zeitmessfunktionen
- Temperatur Sensor
- Auf dem Chip befindliche Fließkomma- Bibliotheken zur Beschleunigung
- 8 × Programmierbare I/O (PIO) State-Maschinen für schnelle Peripherieunterstützung

Ganz schön beeindruckend finde ich. Andererseits: Mein Laptop hat 8 Gigabyte Hauptspeicher statt 264 Kilo Byte ;-)
Da wir mit dem Mikrocontroller hardwaremäßig experimentieren wollen, möchte ich noch auf einige grundlegende Dinge, die die Zerstörung des Mikrocontrollers (oder des PC) nach sich ziehen können, wenn sie nicht beachtet werden, eingehen.

Merke:

- Kein Ausgang ist kurzschlussfest!
- Obwohl das Mikrocontroller-Board mit 5 V versorgt wird, arbeitet der Controller intern mit 3,3 V. Diese Spannung liefert der Mikrocontroller an den GPIOs als Ausgang und diese Spannung verträgt er auch nur an als Eingang geschalteten GPIOs! Das gilt auch für Eingänge, die für ADC Verwendung finden!
- Der maximale Ausgangsstrom beträgt ca. 4 mA[9]. Schließe ich eine rote LED an, sieht die Rechnung so aus: 3,3 V – 1,7 V / 0,004 A = 400 Ω, d.h., der Vorwiderstand sollte um die 400 Ω haben.[10] Ein gut erhältlicher Wert ist 390 Ω. Bei Praxistests ergab sich aber, dass 270 Ω auch noch einwandfrei funktionierten. Werden viele LEDs angeschlossen, sollte man doch eher 390 Ω verwenden.
- Brauche ich mehr Strom, sollte ein externer Transistor, vorzugsweise ein MOSFET verwendet werden.
- Das Board hat einen 3,3 Volt Anschluss (mit „3V3“ beschriftet). Davor sitzt ein Spannungsregler, der aus den 5 V diese 3,3 V erzeugt. Man kann hier ohne Gefahr ca. 50 – 100 mA abgreifen.
- Der 5 V Anschluss (mit „VBUS“ beschriftet) ist bei USB Versorgung praktisch die USB Spannung. Ungefährlich sind auch hier ca. 50 – 100 mA (dann aber bei 5V!).
- Versorgen wir damit unser Steckbrett, empfiehlt sich ein Widerstand von ca. 15 Ω von „VBUS“ zur Plusschiene. Damit begrenze ich bei einem Kurzschluss (kommt in den besten Familien vor) den maximalen Strom auf 5 V / 15 Ω = 0,33 A. Gleichzeitig ist der Widerstand aber so klein, dass er sich nicht störend auf meine Schaltung auswirkt. Auch hier ein Beispiel: Wenn meine Schaltung auf dem Steckbrett 20 mA benötigt, sinkt die Versorgungsspannung an der Plusschiene auf ca. 4,7 V, was für die meisten Schaltungen keine Rolle spielt.

---

9 In Micropython scheint es zur Zeit nicht möglich zu sein, diesen Wert zu ändern, obwohl laut Datenblatt zwischen 2 und 12 mA möglich sind. Die Einstellung auf 4 mA bedeutet sowieso nicht, dass der Strom auf diese Größe begrenzt wird! Bei einem – unfreiwilligen – Experiment habe ich als maximalen Strom 30 mA gemessen. NICHT TUN!!!

10 Wer jetzt ausgestiegen ist: Mein Buch „Eine kleine Einführung in Elektronik“ kann man von „bibliothek.velbert.de/angebote/elektronik-workshop“ herunterladen.

- Bei vielen Experimenten ist es sinnvoll, die 3,3 V an die Versorgungsspannungsschienen des Steckbretts, auch über einen Widerstand, anzuschließen. Da das Steckbrett solche Schienen oben und unten hat, geht auch beides parallel. Dann aber Vorsicht bei der Auswahl. Wie gesagt morden 5 V an einem GPIO mindestens diesen Anschluss, wenn man Pech hat, den ganzen Cotroller.
- Auf dem Board befindet sich schon eine grüne LED. Um sie zu benutzen, muss man „interne_led = machine.Pin(„LED“)“ beim Pi Pico W und „interne_led = machine.Pin(25, machine.Pin.OUT)“ beim Pi Pico (ohne WLAN) eingeben.
- Die Unterscheidung zwischen GPIO Pin Nummer und physikalischem Pin am Board ist wichtig! Im Buch werden **immer** die GPIO Pin Nummern verwendet.
- Ein Kurzschluss **ohne** Vorwiderstand kann auch den PC / Laptop zerstören! Am Besten den Mikrocontroller an einem USB-Hub mit eigener Stromversorgung betreiben.

2 Micropython installieren

Ein ganz kurzes Kapitel, wenn wir den Pi Pico verwenden. Für andere Mikrocontroller stellt es aber auch kein Hexenwerk dar[11].

Stecken wir einen „frischen“ Pi Pico über USB an einen PC, egal welches Betriebssystem, öffnet sich der Explorer oder wie diese Art von Programmen jeweils heißen und zeigt uns so etwas wie einen USB Stick. Wenn nichts passiert, den Datei-Manager öffnen und nach einem Laufwerk schauen. Wenn kein passendes auftaucht, den Pi Pico wieder abziehen, den BOOTSEL Knopf auf der Oberseite gedrückt halten und wieder einstecken. Spätestens jetzt erscheint das Laufwerk.

Vorher oder jetzt sucht man sich das sogenannte Image für Micropython auf der Seite: https://micropython.org/download/?mcu=rp2040

Dann muss man sich noch für die passende Variante Pi Pico oder Pi Pico W (letzterer mit WLAN) oder die inzwischen verfügbaren neuen Varianten Pi Pico 2 (W) entscheiden. Die mit Releases und „latest“ bezeichnete Datei ist die richtige. Die heruntergeladene Datei hat die Endung uf2 und wird nun in das „Laufwerk“ des Pi Pico kopiert. Nach kurzer Zeit verschwindet das Laufwerk. Fertig!

Zur Zeit ist die Version v1.25 aktuell. Alle Programmbeispiele wurden damit getestet.

Nun kann der Mikrocontroller über USB als serielles Gerät[12] angesprochen werden. Am besten macht man das mit Thonny oder ähnlichen Programmen.

3 Entwicklungsumgebungen

Man kann in Python auf der Kommandozeile, die hier REPL heißt, direkt Programmtext eingeben und hat auch rudimentäre Editiermöglichkeiten, wie Cursor bewegen, Eingabehistorie usw. Erst muss man aber auf diese Kommandozeile kommen! Es braucht in jedem Fall ein Programm, das es mir erlaubt, vom PC aus mit dem Mikrocontroller zu kommunizieren. Dann kann ich aber auch direkt eine sogenannte IDE (Integrated Development Environment, zu deutsch integrierte Entwicklungsumgebung) nehmen. Denn außer für ganz kleine Programme macht es gar keinen Spaß, im REPL zu arbeiten!

Ich möchte daher hier einige IDEs vorstellen, die ich für Python verwende. Eine IDE kombiniert einen Editor zur Eingabe und Änderung von Programmtext mit Hilfsmitteln, die den Programmierer

11 Wie genau man den jeweiligen Mikrocontroller „flasht“ kann man im Internet finden.
12 Unter Windows heißen die COMx, unter Linux ttyACMx.

bei der Arbeit unterstützen. Solche Unterstützungen sind Hinweise auf Syntaxfehler, Auto-Vervollständigung von Namen, automatische Formatierung des Programmtextes und vieles mehr.

Im Falle der Mikrocontroller-Entwicklung ist die wichtigste Funktion jedoch die Kommunikation mit dem Controller. Ich möchte aus der IDE heraus mein Programm auf ihm ausführen können, es in seinem Flash-Dateisystem speichern, kurzum, ich möchte wie mit einem Python auf dem PC arbeiten können.

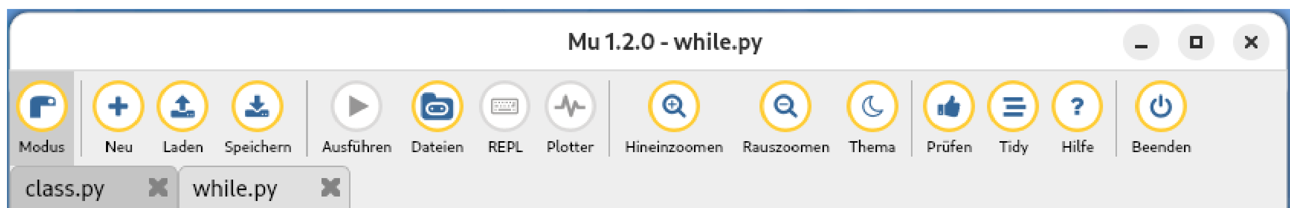
Hier nun drei Programme, die so etwas leisten; alle sind kostenlos und lassen sich unter Windows, MacOS und Linux nutzen.

3.1.1 Mu

Ich fange mal mit Mu an, nein, ich imitiere keine Kuh, sondern so heißt die IDE: Mu Editor.

Sie wurde speziell für die (Micro)Python Entwicklung entwickelt. Mu ist sehr einfach gehalten und erschien daher als der ideale Einstieg. Im Laufe inzwischen längerer Beschäftigung mit ihr stolperte ich aber über häufigere Abstürze mit der 1.2.0 Version, teilweise unter Verlust des eingegebenen Programms :-((

Vielleicht beheben die Entwickler solche ärgerlichen Fehler in der Zukunft.



So sieht die Bedienleiste aus. Die IDE besticht durch die Funktionen „Prüfen“ und „Tidy“ (hübschen). Letzteres formatiert den Programmtext einheitlich (Leerzeichen nach Komma oder Gleichheitszeichen usw.). „Prüfen“ führt eine Syntaxüberprüfung durch und weist auf Fehler hin: vergessene schließende Klammern, falsche Einrückung usw. Ein Workshop-Teilnehmer wies mich darauf hin, dass ein fehlender Import des Moduls „machine“ als Fehler angezeigt wird. Das ist für Anfänger verwirrend. Die IDE kann aber nicht wissen, welche Module auf dem Mikrocontroller schon importiert sind. Sie könnte es herausfinden, indem sie das „dir“ Kommando ausführt, aber wann und wie oft soll sie es ausführen?

Neben Python auf dem PC unterstützt Mu einige Mikrocontroller, u.a. Lego, das Pyboard und den Pi Pico. Programmtext im Editor kann auf Knopfdruck sofort dort ausgeführt werden. Tritt nun ein Fehler auf, ändert man den Programmtext im Editor und kann sofort wieder starten. Eine, wenn auch sehr rudimentäre, Unterstützung für das Flash-Dateisystem ist auch vorhanden (Button „Dateien“). Auch kann man direkt im REPL arbeiten.

Wählt man Python (auf dem PC, muss es dafür natürlich installiert sein) statt Micropython auf einem Controller aus, wird auch das Debuggen erlaubt, eine Technik, die es erlaubt, das Programm an einem gewünschten Punkt zu unterbrechen (ein sogenannter Breakpoint) und Zeile für Zeile auszuführen, dabei den Inhalt von Variablen zu inspizieren usw.

3.1.2 Thonny

Der Editor Thonny ist ähnlich einfach zu bedienen, verfügt aber über vielfältige Konfigurationsmöglichkeiten. Leider fehlt ihm auch die Fehleranzeige auf Knopfdruck von Mu und dessen Tidy Option[13]. Dafür unterstützt Thonny wesentlich mehr Mikrocontroller. Ein weiterer Vorteil besteht darin, dass Thonny das Laden und Bearbeiten des Flash-Dateisystem vollständig beherrscht. Wegen technischer Gründe[14] werde ich Thonny im Workshop einsetzten.

3.1.3 Vscode

In der, ich bin geneigt zu sagen, Königsklasse ist die IDE Vscode angesiedelt. Das Programm stammt von Microsoft und man kann damit nicht nur in Python, sondern Programme in fast jeder Programmiersprache entwickeln. Durch nach-installierbare Erweiterungen stehen dem Programmierer unzählige Hilfsmittel zur Verfügung. Dieser Komfort hat seinen Preis. Man muss sich länger einarbeiten und die Unterstützung von Micropython ist nicht eingebaut, steht aber in Form nicht einer, sondern gleich mehrerer solcher Erweiterungen zur Verfügung. Dadurch hat man die Qual der Wahl.

3.1.4 Was fehlt

Was allen IDEs fehlt, liefert ein Programm auf der Kommandozeile: mpy-cross. Als Python Programm auf dem PC wird es mit „pip“[15] installiert (pip install mpy-cross). Damit lassen sich Micropython-Programme vor-kompilieren[16]. Speichert man ein solcherart kompiliertes Programm, dass die Endung „mpy“ hat, im Flash-Dateisystem, spart man nicht nur dort viel Speicherplatz, sondern auch im RAM. Warum das so ist, erläutere ich später.

Wenn ich an einem Micropython-Projekt arbeite, möchte ich z.B. alle Module vorkompilieren, auf den Mikrocontroller übertragen und die entsprechen „py“-Dateien löschen. Um das automatisieren zu können, benötige ich ein skriptbares Kommandozeilen-Programm, das mit dem Mikrocontroller kommunizieren kann. Solch ein Programm gibt es; es heißt „mpremote“ und lässt sich auch über „pip“ installieren. Im Anhang findet sich eine Windows „bat“-Datei und ein Unix-Shell Skript, das die gewünschte Automatisierung bereitstellt.

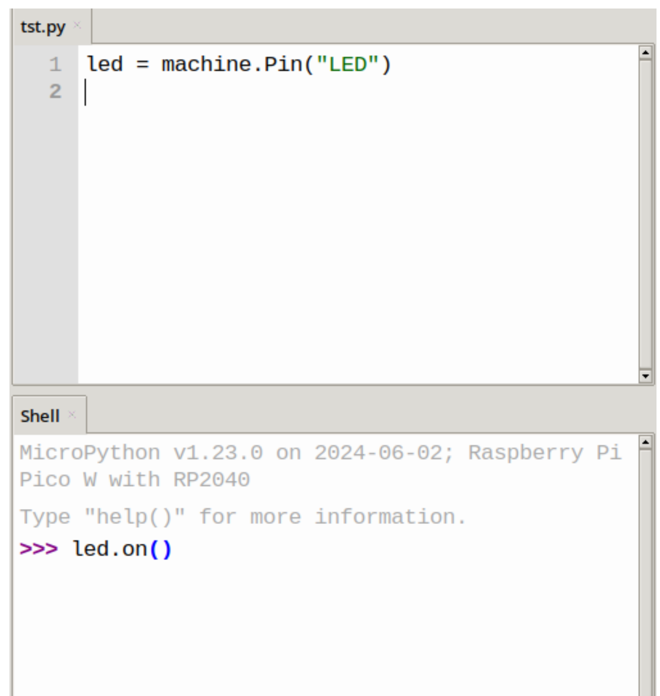
4 Ein „Anfänger“ ist einfach jemand, der anfängt!

Die Lernkurve für Micropython ist ganz unterschiedlich steil, je nachdem, ob man sich z.B. schon mit Elektronik-Hardware beschäftigt hat, eine andere Programmiersprache kennt usw.

Für jemanden ohne Vorkenntnisse möchte ich hier einige Tips geben. Mit Hilfe der IDE habe ich einen Bereich, den Editor, zur Verfügung, in dem ich ein Programm eintippen, es speichern, ändern und auch wieder laden kann[17] und einen Bereich, in dem ich Python Befehle direkt dem Interpreter zur Ausführung übergebe. Da sieht dann z.B. in Thonny so aus:

13 Thonny kennt sogenannte Plug-Ins, die neue Funktionen bereitstellen. Ein Plug-In soll die Formattierung nachrüsten: „https://pypi.org/project/thonny-black-formatter/“. Zuletzt hat das bei mir aber nicht funktioniert.
14 Auf dem Raspberry Pi Server im Workshop läuft Mu nicht.
15 Siehe Kapitel „Module und Pakete“
16 Siehe Kapitel „Wie Python intern arbeitet“
17 Der Editor verfügt über noch viel mehr Möglichkeiten. In Thonny kann ich einen Absatz markieren und mit Kommentarzeichen versehen, in beiden IDEs können mit Strg-Z Eingaben rückgängig gemacht werden usw.

Oben habe ich eine erste Zeile meines Programms geschrieben, unten kann ich direkt ausprobieren, ob es klappt, was ich mir vorgestellt habe. Um unten „led“ benützen zu können, muss ich aber erst das Programm oben laufen lassen. In Thonny und Mu geschieht das durch die Funktionstaste F5 oder durch drücken der „Ausführen“ Button.



Warum ist das so? Der Editor oben verwaltet nur einen Text. Erst, wenn ich diesen Text explizit an den Mikrocontroller sende, bekommt der etwas davon mit.

Gebe ich unten die Zeile ein und drücke ENTER bevor ich das Programm ausgeführt habe, erhalte ich eine Fehlermeldung, die mit „NameError: name 'led' isn't defined“ endet. Das sollte eigentlich weiterhelfen, oder? Fehlermeldungen wirklich lesen, ist ganz wichtig. Allerdings gibt es die nur auf englisch -(([18]

Starte ich nun das Programm oben und wiederhole unten die Eingabe (was ganz einfach dadurch geht, dass ich im Bereich unten den Cursor hinter die >>> setze und die „nach oben“ Taste drücke), geht die auf dem Mikrocontroller-Board befindliche LED an.

Auf diese Art kann man Schritt für Schritt entwickeln. Auch Daten kann ich mir im unteren Bereich einfach ansehen, in dem ich einen Variablennamen dort eintippe. Wir wissen hier noch nicht, was eine Variable oder eine Funktion ist, aber beide haben einen Namen. Ändere ich nun oben etwas, ohne diesen Namen zu verändern, überschreibe ich die alte Version unten. Sie ist dann unwiederbringlich weg.

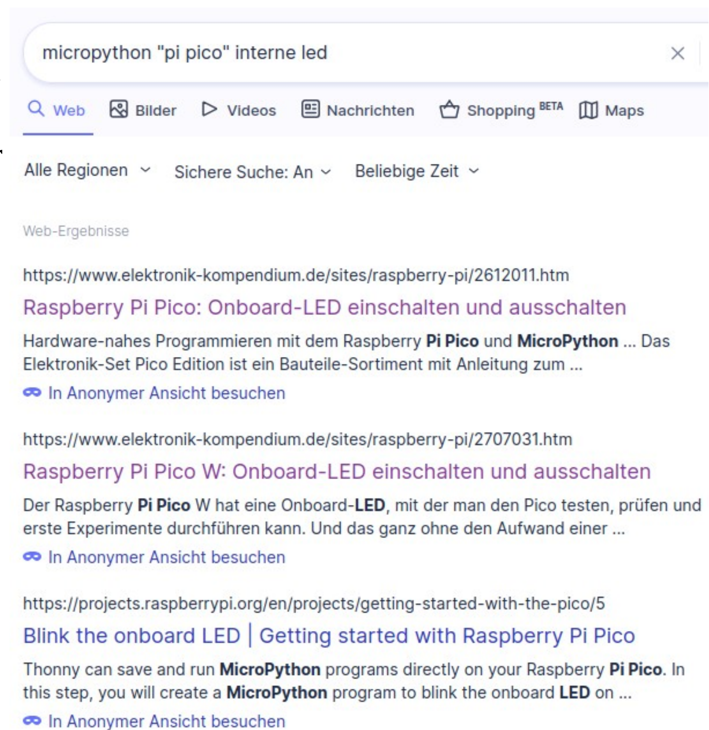
Komplizierte, verschachtelte Ausdrücke (wie „str(int(ipl[3])+ICH_BIN)“ aus dem Beispiel-programm „wconn.py“) teste ich bis heute gerne direkt im unteren Bereich. Wenn dann das Ergebnis stimmt, kopiere ich den Ausdruck (mit der Maus markieren und Strg-C drücken) und gebe ihn oben in mein Programm ein (Strg-V) ein. So vermeide ich Schreibfehler.

Eine mögliche Stolperfalle ergibt sich daraus, dass ich manchmal Fehlermeldungen erhalte - dann funktioniert es gar nicht - oder aber keine, und es läuft dennoch nicht. Gebe ich z.B. „led.on“ ein und erhalte *keine* Fehlermeldung, aber die LED geht auch nicht an, liegt das hier daran, dass Funktionsaufrufe immer mit runden Klammern abgeschlossen werden müssen: „led.on()“. Für Python ist es aber kein Fehler, nur „led.on“ zu schreiben, nur der Sinn ist ein anderer[19].

Fast immer kann ich ein laufendes Programm auf dem Mikrocontroller mit der Tastenkombination „Strg-C“ stoppen. Wenn das nicht hilft, bietet Thonny den „Stop“ Button. Brutal aber wirksam: Den USB-Stecker ziehen (am Besten nicht auf der Seite des Mikrocontrollers, sondern am Computer; der Stecker dort ist haltbarer).

18 Ich empfehle ein Übersetzungsprogramm wie deepl.com, um Fehlermeldungen zu übersetzen.
19 Es wird eine Referenz (in C würde man Zeiger sagen) auf die Funktion zurückgeliefert.

Ein weiterer Tip: das Internet. So banal das klingt, aber mit einem Satz, der das Problem kurz beschreibt, erhalte ich über die Suchmaschine oft den Hinweis, der mir über die Hürde hinweg hilft. Die Suche sollte immer die Wörter „Micropython“ und „Pi Pico“ enthalten, z.B., wenn man Probleme mit der eingebauten LED hat: „micropython "pi pico" interne led“.



Die Anführungszeichen um Pi Pico helfen der Suchmaschine, weil sie sie zu einem Suchbegriff macht. Die Fundstellen mit ihren kurzen Texten zeigen schon, dass der zweite Eintrag wahrscheinlich helfen könnte.

5 Grundelemente von Programmiersprachen

5.1 Was ist eigentlich Programmieren?

Praktisch alle modernen Programmiersprachen teilen sich wesentliche Grundideen, wie man Programme schreiben kann und zu schreiben hat. Ein Programm stellt ja eine Anleitung für den Computer dar, was er wann und wie zu machen hat. Damit ähnelt programmieren der Erstellung eines Kochrezepts:

```
Man nehme 5 Eier, verrühre sie mit 30 Millilitern Milch, dann eine Prise Salz
und Pfeffer hinzugeben. Man erhitze ein Stück Butter in der Pfanne, bis es
beginnt zu brutzeln und gebe die Eimasse unter rühren mit einem Holzlöffel
hinzu. Die Hitze herunter drehen. Wenn das Ei stockt, ist das Rührei fertig.
```

Die Zutaten könnte man als die Daten eines Programms betrachten. Es gibt verschiedene Typen: Eier werden z.B. in Stück gezählt, während Milch in Millilitern gemessen wird. Im Rezept finden sich auch Angaben, die nicht genau quantifiziert werden: ein Stück Butter oder eine Prise Salz. In einem Computerprogramm ginge das nicht, da Computer sehr pingelig sind. Man könnte aber eine Konstante hinterlegen, die Stück und Prise in Gramm definiert.

Wir finden auch Bedingungen und zeitliche Abhängigkeiten im Rezept: Erst, wenn die Butter brutzelt, das Ei hinzugeben. Dazu muss beobachtet werden, wann das Brutzeln beginnt und dann eine Aktion eingeleitet werden. In einem Programm wäre das eine sogenannte Schleife, also die Wiederholung eines Programmabschnitts, bis eine bestimmte Bedingung eintritt (hier eine bestimmte Temperatur in der Pfanne). Dazu muss eine Entscheidung getroffen werden: gemessene Temperatur (über einen angeschlossenen Temperatursensor) größer oder gleich der (Brutzel-)Temperatur. Das ist ein Wert, den man eventuell empirisch ermitteln muss und der dann auch als Konstante im Programm zur Verfügung steht.

Wer schon einmal einen Thermomix in Aktion gesehen hat, der weiß, dass das Kochen per (Computer-)Programm tatsächlich so geht.

Übrigens Thermomix: Darin steckt ein Mikrocontroller, der die Tastatur und den Bildschirm ansteuert, diverse Sensoren für Temperatur, Motordrehzahl usw. ausließt und den Motor, die Heizung und die Zeiten steuert.

Halten wir fest: Programme arbeiten mit Daten, die verschiedene Typen haben, mit Bedingungen, Schleifen und Prozeduren, die Details eines „Rezepts“, wie rühren mit einem Holzlöffel, beinhalten.

Ich versuche jetzt einmal, einen Teil des Rühreirezepts in ein Programm in der Sprache Python umzusetzen. In einer anderen Programmiersprache, C, sähe es ähnlich, aber doch anders aus. Das fängt damit an, dass in Python die sichtbare Form des Programmtextes eine Bedeutung hat (im Gegensatz zur Sprache C).

```
def ruehre(geschwindigkeit: int):
    motorgeschwindigkeit_setzen(geschwindigkeit)
    absenken_ruehrloeffel()
    motor_an()
```

Die Einrückung um genau 4 Leerzeichen macht die 2. bis 4. Zeile zugehörig zur Definition der Prozedur[20].

In C:

```
void ruehre(int geschwindigkeit)
{ motorgeschwindigkeit_setzen(geschwindigkeit);
absenken_ruehrloeffel(); motor_an() }
```

Mit Absicht sieht das nicht schön aus, soll aber demonstrieren, das in C die Textform egal ist. Ich kann das auch so darstellen:

```
void ruehre(int geschwindigkeit)
{
  motorgeschwindigkeit_setzen(geschwindigkeit);
  absenken_ruehrloeffel();
  motor_an()
}
```

Dann bin ich fast bei der in Python zwingenden Schreibweise (bis auf die geschweiften Klammern).

In Python ist der Zusatz hinter dem Parameter „geschwindigkeit“ , das „: int“, optional[21]. Gewöhnt man sich an, diese Typisierung genannten Angaben zu machen, erhält man viel besser lesbare Programme. Und irgendwann, wenn man Fehler beheben muss oder Verbesserungen einfügen will, muss man sein Programm ja auch wieder lesen …

Was bedeutet „int“? In Python eine beliebig lange Ganzzahl, in C ist die Antwort nicht so einfach. Sicherlich auch hier eine Ganzzahl, aber welche Größe sie haben darf, hängt von vielen Faktoren ab, die nicht immer einfach zu ermitteln sind.

Warum ist das wichtig? Ein „int“ in C ist so groß, wie es die CPU vorgibt, also z.B. 32 Bit oder 64 Bit. Was geschieht dann, wenn ich zu dieser maximalen Größe eine 1 hinzu addiere? Zunächst muss ich noch hinzufügen, dass die maximale Größe bei 32 Bit nicht etwa 4294967295 (2 hoch 32 - 1) beträgt, sondern nur die Hälfte (2147483647), weil für die negativen Zahlen das höchste Bit als Vorzeichenindikator verwendet wird. Die Antwort ist vielleicht verblüffend, aber das Ergebnis ist -2147483648.

Das war aus mathematischer Sicht nicht zu erwarten. Aus technischer Sicht, der, wie eine CPU intern arbeitet, schon eher. Denn für Zahlen stellt jede CPU eine feste Anzahl an Bits zur Verfügung; reichen die nicht aus, erfolgt ein Überlauf.

Für einen erfahrenen C Programmierer mag das alles kein Problem sein. Wobei, wenn er plötzlich gezwungen ist, auf einer anderen Plattform zu arbeiten, macht er doch vielleicht einen Fehler. Und Programme sollten eigentlich auf verschiedensten Plattformen gleich laufen, vom Mikrocontroller bis zum PC. Und natürlich keine Fehler enthalten.

Wenn ich jetzt noch erzähle, dass die meisten Betriebssysteme, Windows, Linux, MacOS, weitgehend in C geschrieben sind, wird das Ausmaß der Misere deutlich. Die vielen Meldungen über erfolgreiche Hackerangriffe, die ständigen Patchorgien usw. beruhen auf diesem Umstand[22].

20 Die begriffliche Unterscheidung zwischen Prozedur, Funktion, Methode usw. folgt später.

21 Im Gegensatz zu C, wo es allerdings vor den Variablennamen geschrieben wird; daher muss auch vor die Funktion „ruehre“ das Wort „void“ geschrieben werden, da sie nichts (void) zurückliefert.

22 Neben dem Problem bei der Verarbeitung von Zahlen gibt es noch viele weitere Untiefen in C. Für die Sicherheit im Internet spielt hier vor allem die Speicherverwaltung von C eine Rolle. Darauf wird später noch eingegangen.

Python kennt dieses und andere Probleme von C nicht. Könnte man die Betriebssysteme dann nicht einfach in dieser Sprache schreiben?

Leider nein! Python ist eine sogenannte interpretierende Sprache, während C und viele andere Sprachen kompiliert werden. Das bedeutet vereinfacht Folgendes: Bei letzteren wird der von Menschen geschriebene Quelltext von einem Kompiler genannten Programm in Maschinensprache, etwas, was die CPU direkt versteht, übersetzt; diese Übersetzung ist direkt lauffähig, aber nur auf der Plattform, für die sie übersetzt wurde. Plattform meint dabei das Betriebssystem, aber auch die Hardware wie z.B. 32 oder 64 Bit.

Ein Python Quelltext wird dagegen erst beim Start in eine Form umgewandelt, die ein Interpreter genanntes Programm ausführen kann. Damit kann das Python Programm, wenn es dafür einen Interpreter auf dem Computer gibt, (fast) Plattform unabhängig laufen.

Der Zwischenschritt über den Interpreter macht den Programmablauf aber viel, viel langsamer!

5.2 Wie Python intern arbeitet

Jetzt haben wir schon von kompilieren und interpretieren gehört. Aber was heißt das für Python genau? Ein Interpreter, wie Python, muss auch kompilieren, denn dieses Verb bedeutet einen menschenlesbaren Text so umzuwandeln, dass etwas maschinen-ausführbares herauskommt. Python, als interpretierende Sprache, optimiert dies aber, indem es einen sogenannten Byte-Code[23] kompiliert, der vom Interpreter dann schneller ausgeführt werden kann. Ein Programm, das im REPL eingegeben wird, wird zunächst in Byte-Code umgewandelt (kompiliert) und dann dem Interpreter zur Ausführung übergeben. Genauso verfährt Python bei einer Datei, die ich auf der Kommandozeile aufrufe (python mein_python_programm.py).

Importiere ich jedoch ein Modul, wird der Byte-Code mit der Dateiendung „pyc“ im Dateisystem zwischengespeichert. Beim nächsten Import wird auf diese Datei zugegriffen. Bei großen Projekten mit vielen Importen bedeutet das eine gewaltige Geschwindigkeitssteigerung.

Was macht Micropython? Leider bietet es einen solchen Mechanismus, den man im übrigen „caching“ (Zwischenspeicherung) nennt, nicht an. Aber es gibt ein Programm, „mpy-cross“, das diese Vorkompilierung auf dem PC übernimmt. Gerade für Micropython ergibt dies besonders viel Sinn, da ja die Ressourcen auf dem Mikrocontroller beschränkt sind. Insbesondere der Kompilierungs-Vorgang verbraucht RAM, das nicht vollständig zurückgegeben werden kann. Es entstehen nämlich Lücken im Speicher (siehe Kapitel „Dynamischer Speicher und der Müllsammler“).

Durch die Verwendung von „mpy-cross“ wird beim Import einer „mpy“ Datei nur der vom vorkompilierten Modul benötigte Speicher alloziert[24]. Daher können bei wenig freiem Speicher noch Module geladen werden, deren Import sonst scheitern würde.

Das große Python bemerkt automatisch (anhand der Zeitstempel der Datei), wann es aus dem Cache laden kann und wann es neu kompilieren muss (und die „pyc“ Datei ersetzten muss).

23 Python verfügt über einen Disassembler, das ist ein Programm, dass den Byte-Code in menschenlesbarer Form darstellt. Wer neugierig ist, kann „import dis“ gefolgt von „dis.dis(<Funktion>)“ aufrufen.

24 Manchmal auch „allokiert“. So nennt man den Vorgang, Speicherbereiche zu reservieren.

Bei Micropython ist das so gelöst, dass, wenn zwei gleichnamige Dateien mit Endung „py“ und „mpy“ im Flash-Dateisystem existieren, immer die mit „py“ verwendet wird. Um also in den Genuss des schnellen Ladens und Speicherplatz-Sparens zu kommen, muss ich diese löschen[25].

5.3 Von Variablen, Bedingungen und Schleifen

5.3.1 Variablen

In Python kann man Daten[26] speichern wie in jeder mir bekannten Programmiersprache. Dies geschieht in der Form

```
<name> = <wert>[27]
```

<name> wird zu einer Variablen, wobei <name> auf den Wert verweist. Der Interpreter wird immer auf den Wert zugreifen, wenn er <name> findet, außer <name> steht auf der linken Seite einer Zuweisung, also links von einem Gleichheitszeichen: Dann wird der alte Wert überschrieben.

```
anzahl = 0 # Zuweisung
print[28](anzahl) # Zugriff, ergibt 0
```

Der Wert einer Variablen kann nachträglich geändert werden.

```
anzahl = 2
```

In Python ist es möglich, einer Variablen nacheinander verschiedene Datentypen zuzuweisen.

```
anzahl = 1
anzahl = "Hallo"
```

Der Umstand, dass das „geht“, heißt aber nicht, das man das auch machen sollte. Man erzeugt eher sehr unleserlichen Programmcode. Es gibt aber Fälle, in denen diese Eigenschaft sehr nützlich ist.

Um für den „Leser“ die Sache einfacher und eindeutiger zu machen, kann man schreiben:

```
anzahl: int = 1[29]
```

Bei entsprechender Konfiguration der Entwicklungsumgebung, in der man seine Programme schreibt, erhält man danach eine Warnung, wenn man „Hallo“ zuweisen will.

Mit „Hallo“ haben wir einen neuen Datentyp, genannt String (Zeichenkette), eingeführt. Der wird mit `str` abgekürzt:

```
s: str = "Hallo"
```

In Python[30] besteht ein String aus Unicode-Zeichen. Die erlauben es, die Buchstaben auch von Sprachen mit völlig anderen Zeichensystemen (arabisch, chinesisch usw.) darzustellen. Der Nachteil: Die Länge eines Zeichens kann von einem bis vier Byte (mit 8 Bit pro Byte) reichen. Im normalen Programmieralltag braucht man sich darum nicht zu kümmern; Python macht das schon.

---

25 Und händisch dafür Sorge tragen, bei Änderungen die „mpy“ Datei neu zu erzeugen und auf den Mikocontroller zu laden.

26 Ohne geht es auch nicht, nicht umsonst spricht man von „Datenverarbeitung“.

27 Die spitzen Klammern in Programmbeispielen stellen Platzhalter für Bezeichner dar: Für <name> = <wert> kann ich z.B. pi = 3.14 schreiben.

28 „print“ ist eine Funktion; sie gibt auf den Bildschirm aus, was ihr als Argumente übergeben wird.

29 Python kennt keine Typisierung, d.h. eine strenge Bindung eines Datentyps an eine Variable. Mit der gezeigten Schreibweise erhält aber der Leser einen wichtigen Hinweis und Zusatzprogramme, sogenannte Linter, haben die Chance, fehlerhafte Typänderungen anzuzeigen.

30 Das gilt für die moderne Version 3.x. Auch der Programmtext selbst ist Unicode.

Wenn ich die Länge eines String abfrage, erhalte ich die Anzahl der Zeichen, nicht die Anzahl der Bytes.

Micropython, als kleiner Abkömmling des großen Python, tut sich da schon schwerer. Wie gesagt, Speicherplatz ist Mangelware und die Entscheidung für Unicode kostet Speicherplatz. Zur Zeit dieser Niederschrift ist die Version 1.25.0 von Micropython aktuell. Erste Tests ergaben keine Probleme mit Unicode im Gegensatz zu älteren Versionen. Trotzdem ist es keine gute Idee, eine Variable äöü zu nennen.[31]

Übrigens Namen: Variablennamen müssen mit einem oder mehreren Buchstaben oder Unterstrichen anfangen und dürfen ziemlich[32] lang sein. Groß- und Kleinbuchstaben werden unterschieden! „**A**lter = 42 und print(**a**lter)“ geht nicht. Auch bestimmte Sonderzeichen sind nicht erlaubt: +, -, *, / usw., da Python sie als Operatoren[33] verstehen würde.

Eine dritter Datentyp nennt sich `float`. Damit werden sogenannte Fließkommazahlen gespeichert.

```
f = 1.0 / 3.0 # man beachte den Punkt als Dezimaltrenner
print(f) # gibt 0.3333333 aus
```

In Python wird auf die hardwareseitige Arithmetikeinheit des Rechners und in Micropython auf die des Mikrocontrollers[34] zugegriffen. Sie arbeiten intern mit Brüchen, allerdings nicht dezimal, sondern binär. Eine Fließkommazahl 0.28125 wird durch 1/4 (0.25) + 1/32 (0.03125) gebildet.

In den allermeisten Fällen braucht man sich nicht darum zu kümmern. Teilt man nämlich 1 / 3 erhält man automatisch eine Fließkomma-Zahl. Will man das nicht, gibt es einen speziellen Ganzzahl-Teilungsoperator: „//“. 5 // 3 ergibt damit 1.

Im Bereich der Mikrocontroller-Programmierung braucht man float“ meist wenig, höchstens bei der Ausgabe. Es gibt aber Funktionen, die ein „float“ als Argument[35] brauchen.

Ein Typ sei noch genannt: bool. Im deutschen gern Wahrheitswert genannt, kennt er nur zwei Zustände, wahr und falsch. In Python gibt es zwei Bezeichner, „True“ und „False“ dafür. Das Ergebnis eines Vergleichs, wie man ihn aus der Mathematik kennt, liefert solch einen Typ:

```
5 < 6 # ergibt True
5 > 6 # ergibt False
```

Auch Variablenwerte haben einen Wahrheitswert. Die Regeln: Alle Zahlen außer 0 sind True. Nur ein leerer String, eine leere Liste, ein leeres Dictionary usw. ist False. Wozu das gut ist, erfahren wir im Kapitel „Schleifen und Entscheidungen“.

5.3.2 ~~# Kein~~ Kommentar

Stillschweigend habe ich schon Kommentare eingefügt. Das geschieht mit dem Zeichen „#“, der Raute. Einfache Regel: Hinter der Raute geht der Kommentar los bis zum Zeilenende. Die Raute darf also irgendwo in der Zeile stehen, ganz am Anfang oder am Ende.

---

31 Im REPL weigerte sich Micropython bis vor kurzem, ein ä oder ö als Variablennamen anzunehmen. Mit Version 1.25 geht es!

32 Ich weiß nicht genau, wie lang, aber ein paar hundert Zeichen sind kein Problem.

33 Operatoren werden Symbole genannt, die z.B. für Addition, Subtraktion usw. stehen. Neben den mathematischen Operatoren kennt Python auch solche für z.B. Strings, die oft gleich aussehen. „AB“ + „ab“ ergibt „ABab“.

34 Wenn vorhanden. Der Pi Pico hat „nur“ Fließkommaroutinen in Hardware.

35 Siehe dazu Kapitel „Funktions- und Methodenparameter“.

Es gibt keine mehrzeiligen Kommentare in Python, also wie in C ein /* … */, das über beliebig viele Zeilen gehen kann. Möchte man z.B. 20 Zeilen auskommentieren, muss man 20 Mal eine Raute an den Zeilenanfang setzten.

Hier helfen gegebenenfalls die IDEs. Thonny kennt mit Alt-3 und Alt-4 Tastenkombinationen, mit denen man einen markierten Block kommentieren oder entkommentieren kann. Mu scheint das nicht zu haben. Bei Vscode geht es mit Shift-Strg-7.

Hinweis: Wenn ich in allen folgenden Programmbeispielen Ausgaben zeige, setzte ich eine Raute davor, damit man leicht mit Ausschneiden und Einfügen (Cut & Paste) den Code in Python einfügen und ausführen kann, ohne über Fehlermeldungen wie „not defined“ zu stolpern.

5.3.3 Komplexere Datentypen

5.3.3.1 Listen

Wir können jetzt Zahlen und Texte speichern. Wenn wir aber mit dem Mikrocontroller Daten von außen einlesen und speichern wollen, wie es bei einem Temperatursensor der Fall wäre, kämen wir in Schwierigkeiten. Bei nur einer Variablen würde der alte Wert ja überschrieben; das ist nicht das, was wir erreichen wollen.

Python hat natürlich eine Lösung dafür: Listen. Sie haben, wie eine Variable, einen Namen, kennen aber eine Methode[36] „append“:

```
l = [] # erzeugt eine Listenvariable
l.append(27.3)
l.append(28.2)
l.append(19.0)
print(l) # gibt [27.3, 28.2, 19.0] aus
print(l[1]) # Zugriff auf 2. Element, gibt 28.2 aus
```

Eine Liste kann beliebig viele Elemente speichern, die nicht alle den gleichen Typ haben müssen. Vom ersten Element mit dem Index 0 bis zum letzten mit dem Index Länge der Liste -1 kann ich mit den eckigen Klammern und der Index Nummer zugreifen.

Eine besonders pfiffige Art des Zugriffs auf eine Liste nennt sich Slice (zu deutsch Scheibe). Mit

```
print(l[0:2])
# Ausgabe:
# [27.3, 28.2]
```

hole ich mir alle Elemente vom ersten (Index 0) bis zum, aber nicht einschließlich des zweiten.

Python kennt viele Bedürfnisse von Programmierern und hat die Lösungen schon eingebaut. Ich möchte auf das letzte Element der Liste zugreifen. Nun könnte ich schreiben

```
l[len(l)-1] # len() gibt die Anzahl der Elemente zurück, der
            # letzte Index ist aber 1 weniger, weil Indexe mit 0 beginnen
```

Eleganter geht es so:

```
l[-1] # negative Indexe beginnen von hinten!
```

36 Weil Funktionen in Klassen Methoden genannt werden und Listen als Klasse implementiert sind, nenne ich es hier so. Die genaue Bedeutung und der Unterschied zu Funktionen wird im Kapitel OOP behandelt.

Was mag wohl „Hallo[1]“ bedeuteten? Es wird „a“ ausgegeben, also der zweite Buchstabe in „Hallo“. Python fasst bei der Nutzung eines Indexes einen String als Liste von Buchstaben auf.

Gibt eine Funktion eine Liste zurück, kann direkt ein Index an den Funktionsaufruf angehängt werden: „range(10)“ liefert eine Liste[37] mit den Elementen 0 bis 9, „range(10)[2]“ ergibt deshalb: 2 und „range(10)[11]“ eine Fehlermeldung!

5.3.3.2 Schlüssel – Wert Paare: Dictionaries

In Python gibt es einen sehr eleganten und effizienten Datentyp, um Schlüssel, die eineindeutig sind, und Werte, die einen beliebigen Datentyp haben können, zu verwalten. Dank eines intern optimierten Verfahrens gelingt der Zugriff über den Schlüssel auch bei großen Datenmengen sehr schnell.

```
telefon = {} #  ein Dictionary anlegen
telefon["Gert"] = "0205188221" # die Telefonnummer von Gert
telefon["Susi"] = "05021723816" # die Telefonnummer von Susi
telefon["Willi"] = "03204776129" # die Telefonnummer von Willi
print(telefon["Susi"])
# Ausgabe:[38]
# "05021723816"
```

Weise ich telefon[„Susi“] einen neuen Wert zu, wird der alte überschrieben. Es kann nur einen Schlüssel „Susi“ geben! Da der Typ der Werte beliebig ist, geht aber folgendes:

```
sensor = {}
sensor["Temperatur"] = [] # Wert ist eine Liste!
sensor["Luftfeuchte"] = []
sensor["Temperatur"].append(13.9)
print(sensor["Temperatur"])
# Ausgabe:
# [13.9]
```

Eine wichtige Sache muss ich hier nochmals wiederholen (siehe Kapitel „Schlüsselwort-Parameter“) . Was passiert, wenn ich auf einen nicht vorhandenen Schlüssel zugreife? Das Programm bricht mit einer Fehlermeldung ab! Im Kapitel „Fehler! Fehler! Fehler!“ werden wir lernen, wie man mit solchen sogenannten Laufzeitfehlern umgeht. Aber es geht auch anders und einfacher. Der Datentyp „dict“ verfügt über eine Zugriffsmethode „get“, die erstens keine Fehlermeldung produziert und zweitens einen Default-Wert, der zurückgegeben wird, wenn der Schlüssel nicht existiert, festlegen kann. Der Default-Wert dieses Default-Wertes ist „None“.

```
print(sensor.get("Luftdruck", "unbekannt")
print(sensor.get("Luftdruck")
# Ausgabe:
# unbekannt
# None
```

Es gibt noch eine Möglichkeit zu prüfen, ob es einen Schlüssel gibt:

```
if "Luftdruck" in sensor:
    ....
```

Das Wörtchen „in“ sorgt dafür, dass True oder False zurück geliefert wird.

37 Genauer: „range“ liefert einen Iterator.
38 Immer, wenn ich eine Programmausgabe zeige, schreibe ich eine Raute und ein Leerzeichen davor.

5.3.3.3 Sogar komplex

Python kennt schon als Basistyp komplexe Zahlen. Der ein oder andere wird sich aus dem Mathematikunterricht daran erinnern. Da sie zum Sprachumfang gehören, seien sie hier zumindest erwähnt:

```
c = 1 + 10j #  das j macht die 10 zum sogenannten Imaginärteil
print(v)
print(v.real)
print(v.imag)
# Ausgabe:
# (1+10j)
# 1.0
# 10.0
```

Vereinfacht gesagt ist der Realteil die Länge eines Vektors in einen Koordinatensystem und der Imaginärteil der Winkel des Vektors, angegeben als Länge der Gegenkathete ;-)

5.3.3.4 Noch komplizierter geht auch

Dass eine Liste wieder aus Listen bestehen kann, macht mehrdimensionale Matrizen möglich.

```
matrix = [[[1], [2], [3]], [[10], [20], [30]], [[100], [200], [300]]]
# Hervorhebung der inneren Elemente durch kursive Schrift
matrix[1][2] # 2. Zeile, 3. Spalte
# Ausgabe:
# [30]
```

Der Wert eines Dictionary-Elements kann wieder ein Dictionary sein usw. Eigentlich gibt es kaum eine Beschränkung, wie man die Objekte verschachtelt. Wer jetzt denkt, dass das unnötig oder zu kompliziert sei, dem sei entgegengehalten, dass ein von mir sehr geschätzter Informatiker, Herr Niklas Wirth, schon 1975 in seinem Buch „Algorithmen und Datenstrukturen" nachwies, dass die richtige Datenstruktur wesentlich für schnelle und fehlerfreie Programme ist.

Wie greift man auf solche geschalteten Strukturen zu? Für den Fall von Listen habe ich das oben mit „matrix[1][2]" schon gezeigt. Wie sieht das bei Dictionaries aus? Hier ein geschachteltes Beispiel:

```
d = {1: "a", 2: {"B": "b"}}
print(d[2]["B"]) # abhängig vom Typ mit oder ohne Anführungszeichen
# Ausgabe:
# b
```

Man sieht, die Syntax ist dieselbe wie für Listen.

Ein Personendatensatz, der Name und Vorname, Geburtsjahr, Adresse und Telefonnummer beinhalten soll, könnte so definiert werden:

```
pers = {"Müller, Klaus": [1983, "Velbert, Waldweg 17", 803471], "Maier, Anton": [2002, "Velbert, Schlossstrasse 4", 759012]}
```

Müsste ich jetzt einen „Klaus Müller", der in Mülheim wohnt, einfügen, hätte ich schon ein Problem. Das würde nämlich meinen Datensatz für den „Klaus Müller" in Velbert überschreiben. Auch eine Suche nach Personen, die „Maier" heißen, geht nicht (einfach), da ich den Vornamen zum Schlüsselbestandteil gemacht habe.

Als schöne Übung schlage ich vor, sich einmal ein Struktur für den Personendatensatz auszudenken, der die dargelegten Probleme vermeidet.

5.3.3.5 Un-dynamisch

Pythons Datentypen zeichnen sich fast alle durch eine dynamische Verwaltung aus, d.h. eine Liste kann beliebig wachsen oder schrumpfen. Was allgemein von Vorteil ist, wirft aber Probleme in besonderen Fällen auf, besonders dann, wenn hohe Geschwindigkeit und / oder Speicherverbrauch eine Rolle spielen. Denn das dynamische Verhalten wird dadurch erkauft, dass die Verwaltung aufwendiger, der Verbrauch von Speicher höher ist. Weiter unten, im Kapitel „Änderbar oder unveränderbar“, wird eine Lösung, die Python dafür hat, vorgestellt. Wenn das aber auch nicht reicht, kann man statische Datenstrukturen[39] verwenden. Die ähneln dann z.B. einem Array der Sprache C: Jedes Element hat eine feste Größe, die Anzahl der Elemente muss zu Beginn festgelegt werden. Python ist hier aber nicht so streng wie C, es lässt eine Vergrößerung zu.

```
import array
ar = array.array('I', [0, 1, 2, 3, 4]) # 'I' bezeichnet den Type uint[40]

ar[4] = 40
print(ar)
print(ar[2])
# Ausgabe:
# array('I', [0, 1, 2, 3, 40])
# 2
```

Ebenfalls speichereffizient ist der Typ „bytearray“, der, wie der Name schon sagt, einzelne Bytes verwaltet.

```
b = bytearray(10) # bytearray mit 10 Elementen, alle mit 0 initialisiert
b[5] = 255
print(b)
# Ausgabe:
# bytearray(b'\x00\x00\x00\x00\x00\xff\x00\x00\x00\x00')
b[6] = 256 # führt zu einer Fehlermeldung
```

Mit diesem Datentyp lassen sich ganz hervorragend Netzwerkdaten handhaben! Das Besondere beider Typen besteht darin, dass die Werte immer an die gleichen Stellen im RAM geschrieben werden. Eine Neuzuweisung zu einem Element einer Liste alloziert dagegen neues RAM.

5.3.4 Mehrfachzuweisungen und Slices

In Python ist es möglich, mehrere Werte gleichzeitig an Variablen auf der linken Seite eines Gleichheits-, also Zuweisungszeichens durchzuführen.

```
var1, var2 = 1, 2
```

Nach dieser Anweisung enthält var1 eine 1 und var2 eine 2. Statt literalen Werten auf der rechten Seite sind natürlich auch Variablen erlaubt. Von "literal" sprechen wir, wenn ein Wert als Zeichenfolge direkt im Programmtext steht, also

```
x = 1234 # oder
s = "Text"
```

39 So werden sie offiziell in Python nicht genannt, ich will aber damit verdeutlichen, dass die Speicheradresse bei einer neuen Zuweisung gleich bleibt. Das geht nur bei festen Längen, wie sie z.B. der Typ Byte darstellt.

40 „uint“ oder vorzeichenloser int ist nur einer der Typen, der mit verschiedenen Buchstaben als erstem Argument ausgewählt werden kann.

Nützlich ist dieses Verhalten auch im Zusammenhang mit der for-Schleife, die etwas später ausführlicher behandelt wird. Habe ich z.B. Daten in der Form folgender Liste [[1, 2], [3, 4], [5, 6]] vorliegen, kann ich schreiben:

```
for var1, var2 in [[1, 2], [3, 4], [5, 6]]: # Geht über jedes Element der Liste
    print(var1, var2)
# Ausgabe:
# 1 2
# 3 4
# 5 6
```

Brauche man einen der Werte aus der Liste nicht, verwendet man eine spezielle Variable mit dem Namen "_" (Unterstrich). Ich kann schreiben:

```
for var1, _ in [[1, 2], [3, 4], [5, 6]]:
```

und ignoriere damit den jeweils zweiten Wert.

Möchte ich den Schlüssel und Wert eines Dictionary in einer for-Schleife verarbeiten, sieht das so aus:

```
for k, v in {"a": 1, "b": 2, "c": 3}.items():
    print(k, v)
# Ausgabe:
# c 3
# a 1
# b 2
```

Die Methode „items“ der Klasse „dict“ zerlegt jeden Eintrag in Schlüssel und Wert. Nur der Vollständigkeit halber: Die Methoden „keys“ und „values“ geben entweder den Schlüssel oder den Wert zurück.

Über Slices habe ich schon kurz gesprochen. Hier geht es darum, dass man sie auch auf der linken Seite einer Zuweisung verwenden kann:

```
t = ['H', 'a', 'l', 'l', 'o']
t[2:4] = "X", "Y"
print(t)
# Ausgabe:
# ['H', 'a', 'X', 'Y', 'o']
```

In den Beispielen wurden immer zwei Elemente verwendet. Natürlich geht jede beliebige Anzahl. Auch sei noch einmal daran erinnert, dass bei Slices die erste Zahl den Startindex, der ab Null gezählt wird, bildet, während die zweite Zahl den nicht mehr eingeschlossenen Endwert darstellt. Anders ausgedrückt: Bei der ersten Zahl geht es los bis *vor* die zweite Zahl.

Auch negative Zahlen sind wieder erlaubt: [-1:-3] wären die letzten beiden Einträge. Die Schreibweisen „[2:]“ und „[:10]“ bedeuten von 2 an bis Ende und vom Anfang an bis 9.

In Python gibt es noch einen dritten Parameter für Slices, dessen default Wert 1 ist und daher meist ausgelassen werden kann. Er nennt sich Step (Schritt) und legt die Schrittweite fest:

```
l = ['H', 'a', 'l', 'l', 'o']

print(l[::2])
print(l[::-1])
# Ausgabe:
# ['H', 'l', 'o']
# ['o', 'l', 'l', 'a', 'H']
```

Im ersten „print“ geben wir Alles ([:] entspricht [0:<Länge der Liste>]) des Bereichs aus, aber im Abstand von 2 Schritten. Im zweiten „print“ kehren wir die Reihenfolge mit negativen Schritten um.

Die gewöhnungsbedürftige Festlegung des zweiten Parameters als „bis, aber nicht einschließlich“, hängt m.E. einmal mit der Null-basierten Indexierung zusammen, wird aber als Konvention in Python für viele Funktionen eingehalten, z.B. bei dem schon erwähnten „range“.

```
l = [1, 2, 3, 4]
l[0:0]
l[0:1]
l[0:4]
# Ausgabe:
# []
# [1]
# [1, 2, 3, 4]
```

Hier entspricht 4 der Länge der Liste, da der erste Index 0 ist, werden alle Elemente ausgegeben. Dass „l[0:0]“ die leere Liste zurück gibt, erscheint logisch, „l[3:3]“ würde dasselbe ausgeben.

Wenn wir „l[10] = 10“ eingäben, erhielten wir eine Fehlermeldung „IndexError: list index out of range“. Machen wir Ähnliches mit einem Slice auf der linken Seite, erhalten wir wieder ein unerwartetes Ergebnis:

```
l[9:11] = 10,11
print(l)
# Ausgabe:
# [1, 2, 3, 4, 10, 11]
```

Wer jetzt Lücken erwartet hat, liegt falsch. Mir persönlich gefällt dieses Verhalten nicht. Die implizite Umwandlung in ein „append“, solange weniger Elemente in der Liste sind, als der Slice umfasst, ist intransparent.

Die Beispiele sehen nicht nur sehr konstruiert aus, sie sind es auch! In echten Programmen wird man fast immer Variablen oder Rechenausdrücke anstelle der literalen Zahlen finden.

5.3.5 Lauter Klammern

In den vorigen Kapitel konnten wir sehen, dass Python gern und ausgiebig Gebrauch von Klammern macht. Da droht man den Überblick zu verlieren. Aber eigentlich ist es ganz einfach und logisch.

Runde Klammern	()	Tupel (unveränderbare Liste) / unveränderbare Ausdrücke
Geschweifte Klammern	{}	Dictionary
Eckige Klammern	[]	Listen und Zugriff auf Schlüssel (Dictionary) oder Indexe (Tupel oder Listen)

Mit „unveränderbaren Ausdrücken“ meine ich z.B. den mathematische Vorrang erzwingen, wie bei „(3 + 5) * 2“ (ohne Klammern ein anderes Ergebnis). Selbst wenn wir hier Variablen verwendeten, wäre der Ausdruck nicht veränderbar; es wird nur lesend zugegriffen.

Den Typ „Tupel“ behandle ich gleich in Kapitel „Änderbar oder unveränderbar“.

Die Bedeutung der eckigen Klammern geht aus dem Zusammenhang hervor: „[1, 2, 3, „HALLO“]“ ist eine Liste, bei „[1, 2, 3, ‚HALLO‘][3]“ wähle ich Index 3 aus.

Ein Doppelpunk innerhalb der eckigen Klammern macht ihn zum Slice, einem eigenen Typ in Python, der die drei Werte „Start“, „Ende“ und „Schrittweite“ zusammenfasst.

Zur Erzeugung eines Dictionary benutze ich die geschweiften Klammern. Innerhalb werden die Schlüssel und die Werte durch einen Doppelpunkt getrennt und die Elemente durch ein Komma. Auf die Werte greife ich mit dem Schlüssel in den eckigen Klammern zu oder weise unter dem Schlüssel einen Wert zu, je nachdem ob nach den Klammern ein Gleichheitszeichen folgt oder nicht.

Manchmal verwirren Ausdrücke wie „[[1, 2], [3, 4], [5, 6]]“. Es hilft oft, gedanklich die äußersten Klammern zu entfernen: „[1, 2], [3, 4], [5, 6]“ lässt die Elemente „[1, 2]“ usw. deutlicher hervortreten. Beim indexierten Zugriff habe ich nun die Wahl zwischen „[[1, 2], [3, 4], [5, 6]][1]“ (wählt das zweite Element „[3, 4]“ aus) oder „[[1, 2], [3, 4], [5, 6]][1][0]“ (wählt vom zweiten Element das Nullte aus: „3“).

5.3.6 Schleifen und Entscheidungen

Stellen wir uns einmal vor, wir schreiben ein Programm, um die Temperaturwerte von einem Sensor einzulesen. Oben haben wir gesehen, dass es dafür in Python eine geeignete Datenstruktur, die Liste, gibt. In dem Beispiel haben wir im Programmtext aber die Zeilen l.append() wiederholt untereinander geschrieben. Für viele Temperaturwerte, oder noch wahrscheinlicher, beliebig viele Werte, kann das nicht das geeignete Vorgehen sein. In Python (und in jeder anderen Programmiersprache auch) benutzt man dazu eine Schleife:

```
while True:
    l.append(lese_sensor()) # lese_sensor() holt die Temperatur vom Sensor
```

Das Wort „while“ kann man mit „solange“ übersetzten. Das Wort „True“ bedeutet hier, das die Schleife nie beendet wird (außer ich schalte den Strom ab). Man beachte den Doppelpunkt am Ende der Zeile: Python erfordert ihn immer, bevor ein eingerückter Block kommen muss. Für den Editor wird es dadurch einfach zu „wissen“, wann er nach Eingabe von ENTER vier Leerzeichen einrücken muss.

Was, wenn ich nach 100 gelesenen Temperaturwerten die Schleife verlassen und mal was anderes machen möchte? Nichts einfacher als das:

```
n = 0
while n < 100:
    l.append(lese_sensor())
    n += 1

# nun was anderes
```

Der Operator += ist auch etwas ganz Feines. Er fasst die Addition von der Zahl hinter dem Operator mit der Zuweisung zu der Variablen vor dem Operator zusammen. Ich könnte dafür auch schreiben

```
n = n + 1 # nehme den Wert von n, addiere 1 und weise es an n zu
```

Werte von Variablen werden im Kontext einer Entscheidung, also hinter „while“ oder „if“, auch auf ihren Wahrheitswert überprüft. Die schon erwähnte Regel, dass alle Zahlenwerte außer 0 zu „True“ werden, kann ich ausnutzen, um obiges Programm so umzuschreiben:

```
n = 100
while n: # Schleife stoppt, wenn „n“ 0 wird
    l.append(lese_sensor())
    n -= 1 # herunterzählen
# nun was anderes
```

Auch den Inhalt einer Liste kann ich mir mit while ausgeben lassen. Dazu benötige ich wieder eine Variable, die von 0 bis zur Länge der Liste – 1 ihren Wert ändert:

```
n = 0
while n < len(l): # von 0 bis 99
    print(l[n])
```

Weil so etwas so häufig vorkommt, kennt Python eine Abkürzung:

```
for i in l:
    print(i)
```

Hier ermittelt Python selbständig die Länge der Liste und weist sie Element für Element der Variablen, hier i wie Index, zu.

Da Strings aus Buchstaben bestehen, erlaubt Python die gleiche for-Schleife auch dafür:

```
txt = „Hallo“
for i in txt:
    print(i)
# Ausgabe:
# H
# a
# l
# l
# o
```

Programmierer lieben so etwas, weil es sie vom langatmigen Schreiben von Programmtext befreit.

Die while-Schleife brauchte eine Entscheidung, wann sie enden sollte; die for-Schleife macht das implizit am Ende der Liste.

Sehr oft braucht man aber Entscheidungen über die Weiterführung des Programmflusses. Dazu gibt es in Python die Konstruktion:

```
if <Bedingung>:
    # wenn Bedingung wahr, diesen Block ausführen
elif <neue Bedingung>: # optional, d.h. es muss nicht da sein
    # wenn diese Bedingung wahr ist, den Block ausführen
else: # else ist optional, d.h. es muss nicht da sein
    # wenn die Bedingungen falsch sind, diesen Block ausführen
# weiter geht es
```

Während der Block der Schleifen immer und immer wieder durchlaufen wird, wird der Block nach „if“, „elif“ und nach „else“ nur einmal ausgeführt.

Die Bedingung kann auch hier wieder eine einzelne Variable sein:

```
s = "Hallo" # oder "", d.h. leer
if s:
```

```
    # Variable nicht leer
else:
    # Variable leer
```

Mit Bedingungen kann ich meinem Programm beibringen, auf unterschiedliche Zustände zu reagieren. Eine Programmiersprache ohne Bedingungen könnte nur vom Start an immer wieder exakt das gleiche tun.

5.3.7 Operatoren

Vielleicht ist der Begriff des Operators aus der Mathematik bekannt. In Python addieren wir mit +, subtrahieren mit -, multiplizieren mit * und dividieren mit / oder //[41]. Aus der Schule kennen wir vielleicht noch den Modulo-Operator %, der den Rest einer Division ergibt. Mit == prüfen wir, ob zwei Werte gleich sind und mit != ob sie ungleich sind.

Python kennt zahlreiche Operatoren, also Symbole für Operationen auf Daten. Im Unterschied zur Mathematik gibt es die Operatoren auch für nicht-numerische Daten wie Strings. Hier haben sie naturgemäß eine andere Bedeutung, wie man an diesem Beispiel sehen kann:

```
s = "Hallo"
n = s + " Gert"
print(n)
# Ausgabe:
# 'Hallo Gert'
```

Die Addition von zwei oder mehr Strings hängt sie aneinander. Was macht eine String „Multiplikation“?

```
"abc" * 10
'abcabcabcabcabcabcabcabcabcabc'
```

Das sieht fast wie eine Multiplikation aus, oder nicht?

Ich kann und will hier nicht alle Operatoren vorstellen, aber eine Sache ist noch wichtig zu erwähnen: Vorrang (englisch Precedence). Aus der Mathematik kennen wir die Aussage: Punktrechnung geht vor Strichrechnung, was bedeutet, 10 + 5 * 3 = 25 ist nicht dasselbe wie (10 + 5) * 3 = 45. In Python darf ich immer runde Klammern benützen, um es eindeutig zu machen; ansonsten richtet sich der Vorrang für Zahlen nach der Mathematik.

5.3.8 Änderbar oder unveränderbar

Python kennt eine Reihe von Datentypen, die zur der Erhöhung der Verarbeitungsgeschwindigkeit unveränderbar sind. Der Typ String gehört z.B. dazu. Nanu, wir haben doch oben Strings aneinander gehängt und einer String-Variablen zugewiesen! Ja, aber intern hat Python dabei aus den aneinander gehängten Strings einen neuen erzeugt und dann den alten Wert in der Variablen ersetzt. Und das soll „schneller“ sein? In vielen Fällen durchaus, da Strings sehr oft nur lesend verwendet werden, z.B. wenn ich einen auf dem Bildschirm ausgebe.[42]

41 Damit wird eine Division mit einem Ganzzahlenergebnis erzwungen.

42 Ein String belegt im Speicher eine Anzahl Bytes. Wäre er änderbar, müsste für einen längeren String ein wenig Speicher hinzugefügt werden. Das geht aber meist nicht direkt am Ende des Strings, also müsste er im Speicher gestückelt werden …. Das alles spart man sich, wenn er unveränderbar ist.

Das gleich gilt auch für „Tupel“. So werden in Python unveränderliche Listen genannt. Wenn Python weiß, das sich eine Daten-Liste nicht verändern kann, kann es sie intern viel effektiver verwalten. Ein Tuple sieht so aus:

```
(1, 2, 3, „A“)
```

Es sind die runden Klammern, die Python darauf hinweisen, dass ein Tuple gemeint ist. Weil Python aber, wie schon angemerkt, weiß, was sich Programmierer wünschen, erlaubt es trotzdem

```
t = (1, 2 ,3, „A“)
t += (4, 5)
print(t)
# Ausgabe:
# (1, 2 ,3, „A“, 4, 5)
```

Wieder wird intern ein komplett neuer Tuple aus den beiden gebildet und „t“ zugewiesen.

5.4 „Prozeduren“, Funktionen und Methoden

Um es gleich vorweg zu sagen: In Python gibt es keine Prozeduren. Warum führe ich sie auf? Weil es sie in anderen Sprachen, insbesondere in *Lehr*sprachen wie Pascal gibt. Der Name Prozedur erinnert an unser Kochrezept. Der Computer macht etwas, prozessiert etwas anhand einer Anleitung.

Prozeduren liefern, im Gegensatz zu Funktionen, nichts zurück. In Python hat man es sich einfach gemacht. Eine Funktion kann, muss aber nichts zurückgeben, heißt aber immer Funktion.

Wozu dient nun eine Funktion? Sie fasst unter einem Namen eine Reihe von Programmzeilen zusammen. Ihre Syntax sieht so aus:

```
def <Name>([<Parameter>],…):
    # Block
```

Das mit den Parametern sieht kompliziert aus, bedeutet aber nur, das eine Funktion keinen, einen oder mehrere Parameter haben kann, so wie ich das als Programmierer gerne hätte.

Im Programm kann ich die Funktion wiederum nie, ein oder mehrmals aufrufen. Ihr Programmtext wird dann abgearbeitet, als wenn der Block an dieser Stelle im Text stünde. Das ist sehr praktisch, wenn ich an unterschiedlichen Orten meines Programms gleiche Dinge tun muss[43] und mit Hilfe der Parameter bin ich zugleich flexibel genug, nicht immer nur exakt dasselbe zu tun.

Genug der Vorrede, ein Beispiel:

```
# blitzen.py
# LEDs an GPIO 13 und 14 müssen angeschlossen sein (mit Vorwiderstand!)
def blitzen(led: machine.Pin, dauer: float=0.05):
    # Lässt eine LED kurz aufblitzen
    led.on()
    time.sleep(dauer)
    led.off()
```

Wir **def**inieren eine Funktion mit Namen „blitzen“. Für Funktionsnamen nimmt man gerne Verben. Sie hat zwei Parameter. Der erste ist vom Typ machine.Pin (Klasse Pin aus Modul machine). Der zweite hat den Namen „dauer“ und den Typ float. Außerdem besitzt dieser Parameter einen

43 Die eingebauten Funktionen in Python haben wir ja auch schon so benutzt.

sogenannten Default-Wert. Beim Aufruf der Funktion muss man für all die Parameter Argumente[44] übergeben, die *keinen* Default-Wert haben. Da die Parameter der Reihe nach angeordnet sind[45], kann man aber nach Parameter mit Default-Wert keinen mehr ohne einen solchen festlegen.

Die Funktion „blitzen" rufe ich nun so auf:

```
# blitzen.py, Fortsetzung
led1 = machine.Pin(13, machine.Pin.OUT)
led2 = machine.Pin(14, machine.Pin.OUT)
blitzen(led1) # led1 blitzt für 0.05 Sekunden
# irgendetwas anderes …
blitzen(led2, 0.1) # led2 blitzt für 0.1 Sekunden
blitzen(led2) # led2 blitzt für 0.05 Sekunden
```

Mit Funktionen strukturiert man sein Programm; man kapselt darin wiederkehrende Aufgaben. Die Daten in der Funktion sind aber nur temporär. Bei jedem Aufruf werden sie neu erstellt. Anders formuliert: Eine Funktion kann sich nichts „merken".

Nehmen wir an, wir wollten „blitzen" so ändern, dass der jeweils letzte übergebene Wert für „dauer" gespeichert wird, wenn ich also einmal

```
blitzen(led1, 0.7)
```

schreibe, soll beim nächsten Aufruf

```
blitzen(led1)
```

wieder 0,7 Sekunden die LED leuchten. In Python ist das nicht möglich; andere Programmiersprachen wie C, Perl usw. haben dafür sogenannte statische Variablen. Natürlich gibt es in Python doch eine Lösung, sie führt aber über die OOP. Wir müssten aus der Funktion „blitzen" die Methode „blitzen" machen.

```
# Nicht lauffähig!
def blitzen(self, led: machine.Pin, dauer: float=None):
    if dauer: # True, wenn ich eine Zahl übergeben habe
        self.dauer = dauer
    led.on()
    time.sleep(self.dauer)
    led.off()
```

Das ist noch nicht wirklich lauffähig, aber so sähe es in etwa aus. „self" macht die Methode als zu einer Klasse zugehörig erkennbar, „None" ist übrigens ein Nicht-Wert, ein absichtlich nicht festgelegter Wert. Hier hilft er zu erkennen, ob ein Argument übergeben wurde oder nicht. Mehr über OOP und Klassen weiter unten.

5.4.1 Funktions- und Methodenparameter

Bisher haben wir nur sogenannte positionelle Parameter verwendet. Sie sind auch die erste Wahl bei wenigen Parametern. Was aber, wenn ich 10 oder 20 Parameter brauche? Als derjenige, der die Funktion[46] programmiert, habe ich damit keinerlei Probleme: Jeder positionelle Parameter hat einen Namen, auf den ich in der Funktion, auch in unterschiedlicher Reihenfolge, zugreifen kann.

---

44 Parameter nennt man das, was die Funktion festlegt, Argument das, was beim Aufruf übergeben wird.

45 Das nennt man positionelle Parameter, sie haben eine feste Position in der Parameterleiste. Daneben gibt es in Python noch andere Formen von Parametern, die wir gleich ausführlich behandeln.

46 Für Methoden gilt alles hier geschriebene 1 : 1, bis auf das „self" als immer erstem Parameter (nicht als Argument!).

Will ich aber später die Funktion aufrufen, muss ich mich daran erinnern, in welche Reihenfolge sie gehören. Besonders bei Parametern mit Default-Werten stört das sehr, da ich alle Argumente bis zu dem, dessen Default-Wert ich überschreiben möchte, übergeben muss.[47]

So etwas mutet Python uns natürlich nicht zu! Die Lösung besteht darin, dass ich den als Parameter vergebenen Namen auch beim Aufruf verwenden kann:

```
def parm(a, b=1, c=2, d=3):
 print(a, b, c, d)

parm(0) # b bis d behalten ihren Default-Wert
parm(0, 1, 2, 5) # Umständlich!
parm(0, d=5) # Die Lösung
parm(a=42) # auch der erste Parameter kann per Namen aufgerufen werden
```

5.4.2 Variable Anzahl von Argumenten

Manchmal weiß ich bei der Definition einer Funktion nicht, wie viele Argumente später übergeben werden sollen. In manchen Programmiersprachen schon eine große Hürde, meistert das Python mit links:

```
def varparm(*args):
    for i in args:
        print(i)
    print(args[0])

varparm(1, 2, 3, 4, 5)
# ergibt
1
2
3
4
5
1
```

Aufgerufen mit 5 Argumenten stehen diese in der Funktion als Elemente der Liste „args“ zur Verfügung. Ein „args" ohne das Sternchen davor würde Python missverstehen; es sieht dann nur den Namen eines Parameters. Der Name „args“ stellt eine Konvention dar: Ich kann es nennen, wie ich will. Das Sternchen bewirkt diese besondere Funktionalität.

Nehmen wir an, ich habe eine Liste „liste“ mit 5 Elementen. Wie kann ich die meiner Funktion „varparm“ übergeben? Ein vielleicht naheliegender Gedanke:

```
liste = [10, 20, 30, 40]
varparm(liste)
```

Leider funktioniert das nicht, die Ausgabe zeigt das:

```
[10, 20, 30, 40]
[10, 20, 30, 40]
```

Die for-Schleife gibt [10, 20, 30, 40] aus und print(args[0]) gibt auch [10, 20, 30, 40] aus.

Versetzen wir uns in die Lage von Python. Es „sieht“ ein Argument. Das ist absolut OK, denn von gar keinem bis zu unendlich vielen Argumenten ist alles erlaubt. Die for-Schleife ist auch mit einem

47 Woher soll Python wissen, dass ich "eigentlich" den 10ten Parameter meine?

Element zufrieden, auch wenn es aus einer Liste besteht. Und „args[0]“, also das erste Element, passt auch auf eine Liste. Wir erinnern uns, die Elemente einer Liste können wieder Listen sein.

Wir müssen Python hier unter die Arme greifen:

```
varparm(*liste)
```

löst das Problem. Wieder verwenden wir das Sternchen, aber beim Aufruf; diesmal sagt es Python: Hallo, ich übergebe hier eine Liste, aber verarbeite sie bitte Element für Element.

5.4.3 Schlüsselwort-Parameter

Was passiert eigentlich, wenn ich beim Aufruf der Funktion „parm“

```
parm(10, e=25)
```

also einen nicht definierten Parameter eingebe. Python zeigt mit die rote Karte:

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unexpected keyword argument 'e'
```

Aber schön wäre es schon, wenn so etwas ginge. Und – es geht:

```
# parm_tst.py
def parm(a, b=1, c=2, d=3, **kwargs):
    print(a, b, c, d)
    print(kwargs)

parm(10, 20, e=25)

# Ausgabe:
# 10 20 2 3
# {'e': 25}

# auch das geht:
parm(11, e="E", f="F", b=12)
# Ausgabe:
# 11 12 2 3
# {'e': 'E', 'f': 'F'}
```

Ich kann positionelle, benannte und sogenannte Keyword-Argumente mischen[48]. Und auch hier sind es die Doppelsternchen, die das bewirken; der Name kwargs ist nur Konvention.[49]

Wenn wir an die komplexeren Datentypen zurückdenken, sind Keyword- oder Schlüsselwort-Argumente Elemente eines Dictionaries (dict Typ). Und wieder stellen wir uns die Frage, was ich mit einer Dictionary-Variable machen muss, um sie an meine Funktion zu übergeben.

```
d = {"e": "E", "f": "F"}
parm(10, **d)
```

Richtig: Ich muss Python syntaktisch auf die Sprünge helfen, denn es kann einfach nicht wissen, wie es damit umgehen soll. Die Variable „d" wird durch das Doppelsternchen in seine Elemente zerlegt.

48 Nicht aber mit beliebig mit *args Argumenten. Auch hier steht Python eventuell vor der unlösbaren Aufgabe, wissen zu müssen, ab mein Argument ein variables ist oder nicht. Sie können daher nur am Ende, aber gegebenenfalls vor einem Schlüsselwort-Argument stehen.

49 **kwargs funktioniert wie ein Sammler für alle Schlüsselwort-Wert Argumente, die **nicht** in der Parameterliste stehen.

Ein wichtiger Hinweis darf hier nicht fehlen: Wie gehe ich mit **kwargs in der Funktion um? Ich weiß, dass es sich um ein Dictionary handelt, weshalb ich es, wie im Beispiel, einfach der print-Funktion übergeben kann oder mit „for“ über seine Elemente iterieren darf. Wie greife ich aber auf ein einzelnes Element zu, von dem ich nicht wissen kann, ob es vorhanden ist?

Im Kapitel „Schlüssel – Wert Paare: Dictionaries“ haben wir neben der Schreibweise

```
d["key"]
```

auch „d.get“ kennen gelernt. Gibt es das Schlüsselwort „key“ nicht in „d“, würde das Programm mit einer Fehlermeldung abgebrochen. Mit der Schreibweise

```
d.get("key"[, <Default>])
```

verhindere ich das. Ist „key“ vorhanden, wird damit sein Wert, wie in der Schreibweise mit den eckigen Klammern, zurückgegebenen. Ist „key“ nicht vorhanden und kein Default-Wert angegeben, wird nun „None“, der Nicht-Wert geliefert. Ist ein Default definiert, wird der zurückgegeben. Problem gelöst!

Ein letzter, wichtiger Punkt betrifft eine Schreibweise bei Parametern, die man immer mal wieder in anderer Leute Quelltext findet und die auf den ersten Blick verwirrend aussieht:

```
def tst(pos1, pos2, *, kw1, kw2):
    ...
```

Gemeint ist das Sternchen zwischen „pos2“ und „kw1“. Mit meiner Benamung habe ich schon einen ersten Hinweis gegeben. „pos1“ und „pos2“ sind positionelle Parameter, wegen des Sternchens **müssen** „kw1“ und „kw2“ als Schlüsselwort Argumente eingegeben werden.

```
# tst("a", "b", "c", "d") # FALSCH
tst("a", "b", kw1="c", kw2="d") # RICHTIG
```

Wozu braucht man das? Schreibe ich eine Funktion, die von anderen Leuten genutzt werden soll, zwinge ich sie, Argumente als Keyword zu übergeben. Das kann der entscheidende Hinweis darauf sein, dass cih nun das Verhalten der Funktion grundlegend verändere.

5.4.4 Von Werten, Referenzen und Seiteneffekten

Was ein Wert ist, haben wir im Kapitel „Variablen“ kennengelernt. Ein Wert kann jedes Datum sein, dass der Programmierer speichern will: eine Zahl, ein Text, eine Liste usw. Und der Name einer Variablen dient dem Programmierer zur Unterscheidung der Daten.

Kompilierende Programmiersprachen wie C oder C++ übersetzen eine Variable direkt in die Speicheradresse, an der ihr Wert abgelegt ist. Die Namen werden – intern - gar nicht mehr verwendet, nur noch die Adresse.

Als interpretierende und vor allem nicht typisierte[50] Sprache muss Python das anders machen. Da es eine objektorientierte Sprache ist, setzt es das konsequenterweise so um, dass alles zum Objekt wird. An dieser Stelle muss ich nun unter Vorgriff auf die spätere ausführliche Erklärung von OOP erläutern, was ein Objekt darstellt. Vereinfacht gesagt handelt es sich – im Falle von Variablen – um den Wert **plus** zusätzlicher Verwaltungsinformationen, als da sind: Der augenblickliche Typ, die Speicheradresse des Objekts, anwendbare Methoden auf den Wert und noch einiges mehr,

50 Die Größe (in Bytes) eines Wertes kann gänzlich unterschiedlich je nach Typ sein.

zusammengepackt in eben ein Objekt. Man kann sich leicht vorstellen, dass das ein Vielfaches der Größe des eigentlichen Werts einnehmen kann. Pythons Lösung besteht darin, dass es ein Objekt nicht kopiert, also verdoppelt, wenn es z.B.

```
a = 42
b = a
```

ausführt. Sondern es erzeugt nur einen „Referenz“ genannten Verweis vom Namen der Variablen auf das Objekt. Man kann das prüfen, indem man mit der Funktion „id“ die Speicheradresse, die auch ID genannt wird, abfragt:

```
print(id(a), id(b))
# Ausgabe:
# 139872779958848 139872779958848 #  diese Zahlen sind nur ein Beispiel
```

Sind die Zahlen gleich, haben natürlich auch die Variablen den gleichen Wert.

Was geschieht nun, wenn ich der Variablen „b“ einen neuen Wert zuweise? Es wird ein neues Objekt angelegt und der Verweis von der Variablen darauf geändert.

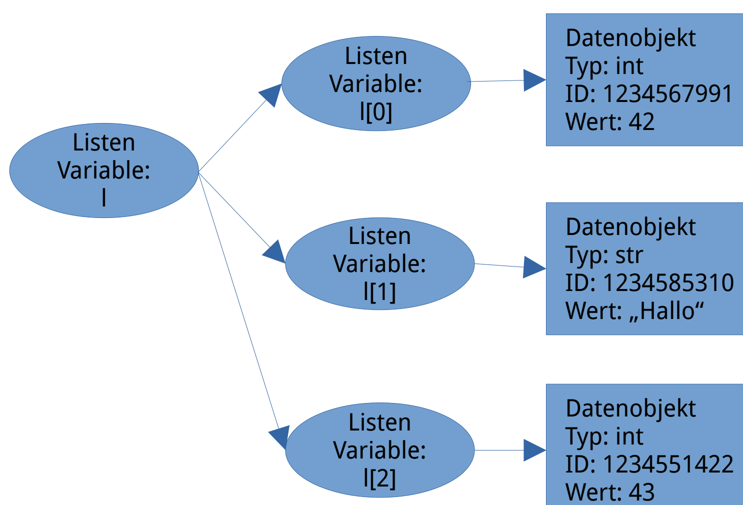
```
b = 2
print(id(b))
# Ausgabe:
# 139872772375872 # nur ein Beispiel, die Adresse hat sich aber geändert
```

Um zu erkennen, wie schlau und effektiv Python mit dem Speicher umgeht, hier noch ein Beispiel:

```
a = "Hallo"
b = a
print(id(a), id(b))
b = "Hallo"
print(id(b))
# Ausgabe:
# 139873020235216 139873020235216
# 139873020235216
```

Obwohl ich „b“ einen neuen Wert zuweise, bleibt die Adresse gleich, denn Python hat gemerkt, dass der Wert der gleiche ist!

Etwas komplizierter wird es, wenn wir es mit strukturierten Daten wie Listen oder Dictionaries zu tun haben. Vielleicht hilft die Vorstellung, dass auch jedes Element einer Liste nur eine Referenz auf das eigentliche Datenobjekt ist.



Die Bedeutung wird im folgendem Programm sichtbar:

```
l1 = [42, "Hallo", 43]
l2 = l1
l1[1] = "Kuckuck"
print(l1, l2)
# Ausgabe:
# [42, 'Kuckuck', 43] [42, 'Kuckuck', 43]
```

Beide Variablen enthalten den neuen Wert! Eine Überprüfung mit „id“ zeigt, dass die IDs von „l1“ und „l2“ gleich bleiben, die von „l1[1]“ und „l2[1]“ sich geändert haben.

Ein solches Verhalten wird „Seiteneffekt“ genannt und wird normalerweise gar nicht geschätzt, da es zu schwer auffindbaren Fehlern führen kann.

Aber kommt so etwas denn normalerweise vor? Warum sollte eine Programmierer „l2“ „l1“ zuweisen? Er kann doch ohne irgendwelche Einschränken auch nur mit „l1“ arbeiten.

Jetzt kommen Funktionen und Methoden mit ihren Parametern ins Spiel! Jeder Parameter hat einen (Variablen-)Namen. In der Funktion wird auf diesen Namen zugegriffen. Übergebe ich ein Argument an eine Funktion, wird intern unter dem Parameternamen darauf eine Referenz gesetzt.

```
def tst(a):
    print(id(a))

x = 42
print(id(x)
tst(x)
```

Die beiden Adressen sind identisch. Weise ich „a“ in der Funktion, sagen wir einmal 55 zu, würde wieder das oben beschriebene geschehen: Die Variable „a“ erhält eine Referenz auf das neue Objekt mit dem Wert 55 .

```
def tst(a):
    print(1, a, id(a))
    a = 55
    print(2, a, id(a))

x = 42
print(3, x, id(x))
tst(x)
print(4, x, id(x))

# Ausgabe:
# 3 42 123456785
# 1 42 123456785
# 2 55 123456111
# 4 42 123456785
```

Schauen wir uns die Ergebnisse an, sehen wir, wie erwartet, dass die Variable „a“ die gleiche ID hat wie „x“. Nach der Zuweisung hat sie eine neue ID. Die Adresse von „x“ bleibt unverändert.

Würden wir „x“ in „a“ umbenennen, erlebten wir vielleicht eine Überraschung: Am Ende des Programms würde „a“ wieder den Wert 42 und die ursprüngliche ID haben. Das zeigt aber nur, dass die Aussage, dass innerhalb der Funktion die Parametervariable eine Referenz auf die Argumentenvariable (quasi „$a_p = a_a$“) erhält, richtig ist.[51]

51 Im Kapitel „Von Räumen und Sichtbarkeit“ werden wir hierzu mehr erfahren.

Übergebe ich der Funktion als Argument eine Liste (oder ein Dictionary oder eine andere komplexe Datenstruktur), würde sich nichts ändern, da ich ja mit der Zuweisung „a = 55“ nur die Referenz auf dieses neue Objekt ändere. Anders sähe es aus, wenn meine Funktion ein Element der Liste auf einen neuen Wert setzt; nun bleibt die Referenz dieselbe, mit dem schon bekannten Seiteneffekt!

```
def tst(a):
    print(a, id(a)) # 1
    a[1] = 55
    print(a, id(a)) # 2

x = [1, 2, 3]
tst(x)

print(x) # 3
# Ausgabe:
# [1, 2, 3] 140044577700416 # 1
# [1, 55, 3] 140044577700416 # 2
# [1, 55, 3] # 3
```

Nun fragt man sich vielleicht, wie man denn diesen Seiteneffekt verhindern kann, wenn er nicht explizit gewünscht ist.

Dazu muss ich eine Kopie des Objekts vor oder in der Funktion erzeugen. Nach dem oben Erläuterten geht das aber gar nicht so einfach. Python stellt als Lösung das Modul „copy“ zur Verfügung. Damit reduziert sich der Aufwand auf einen Methodenaufruf; Micropython kennt dieses Modul standardmäßig nicht[52]. Hier muss man selbst ein wenig kreativ werden.

Es gibt verschiedene Methoden für Listen. Eine haben wir schon kennengelernt: Slicing. Denn die Slicing-Syntax erzeugt eine Kopie. Das sieht dann so aus:

```
l = [1, 2, 3]
tst(l[:]) # hier wird vom ersten bis letzten Element ein Slice gebildet
```

Eine andere Methode lässt sich auf Listen und Dictionaries anwenden. Hier wird ausgenutzt, dass die Klassen „list“ und „dict“ neue Instanzen erzeugen und sie mit den Daten, die ihnen übergeben werden, füllen.

```
l = [1, 2, 3]
d = {1: "a", 2: "b"}
dkopie = dict(d)
dkopie[3] = "c"

lkopie = list(l)
lkopie.append(4)
print(d, dkopie, id(d), id(dkopie))
print(l, lkopie, id(l), id(lkopie))

# Ausgabe:
# {2: 'b', 1: 'a'} {1: 'a', 2: 'b', 3: 'c'} 536949280 536944464[53]
# [1, 2, 3] [1, 2, 3, 4] 140312695339584 140312687940288
```

Die Welt scheint wieder in Ordnung, Problem gelöst. Dass es nicht ganz so einfach ist und die Existenz des Moduls „copy“ einen Sinn ergibt, zeigt folgender Code:

52 Forscht man im Internet nach, finden sich dann aber doch Module, die diesen Zweck erfüllen, für Micropython.

53 Das die Reihenfolge der Elemente anders ist, darf uns bei einem Dictionary nicht stören. Nur die Schlüssel - Wert Paare sind von Bedeutung.

```
l = [1, [11, 22], 3]
lkopie = l[:] # Kopie erzeugen
l[1][0] = 33
l[0] = 11
print(l, lkopie)
# Ausgabe:
# [11, [33, 22], 3] [1, [33, 22], 3]
```

Ups - das scheint nicht richtig zu sein. Aber Python hat schon alles richtig gemacht.

In der Informatik nennt man dieses Phänomen tiefe oder flache Kopie[54]. Während letztere wie in unseren Beispielen einfach zu erzeugen ist, muss ich für die tiefe Kopie eine Menge Code schreiben. Das liegt daran, dass ein Dictionary- oder ein Listeneintrag nicht nur wieder einen solchen beinhalten kann, sondern dass auch der wieder einen Dictionary- oder einen Listeneintrag enthalten kann, der wieder …

Für eine tiefe Kopie muss ich also nicht nur jedes Element untersuchen, sondern auch die Elemente in den Elementen usw. Dabei gibt es keine vorher bekannte Grenze und mein Programm muss damit umgehen können.

Auch hierfür haben die Informatiker einen Fachbegriff: Rekursion. Das bezeichnet auch gleichzeitig einen Algorithmus, der sich solange selbst aufruft, bis eine Abbruchbedingung – z.B. die tiefste Verschachtelungsebene wurde erreicht – eintritt.

Eine Frage wäre noch zu stellen: Wozu braucht man eigentlich Kopien? Dass Python mit Referenzen arbeitet, hat ja einen guten Grund: Es spart Platz und erhöht die Geschwindigkeit.

Dennoch kann es sein, dass ich mir einen Datenzustand merken möchte, z.B. um ihn später mit einem aktuellen Zustand vergleichen zu können.

In meiner Praxis kam das bislang sehr selten vor, aber wenn, musste ich mir genau anschauen, ob meine Daten tiefer geschachtelt waren und daher eine tiefe Kopie benötigten. Wie gesagt: Die Fehler, die dabei entstehen können, sind schwer zu finden.

5.4.5 (Zurück)geben ist seliger denn nehmen

Funktionen, dass wurde eingangs schon erwähnt, können auch etwas zurückgeben. Bisher haben wir uns aber noch nicht angeschaut, wie das im Detail geht.

Machen wir nichts, macht Python etwas für uns: Der implizite Rückgabewert einer Funktion ist „None“. Wollen wir bewusst etwas an den Aufrufer der Funktion liefern, müssen wir das reservierte Wort „return“ verwenden. Schreiben wir nur „return“, ergänzt Python das „None“ und beendet die Funktion. Das ist an jeder beliebigen Stelle erlaubt; auch ein Mehrmaliges „return“ in einer Funktion ist möglich, ergibt aber nur im Zusammenhang mit „if“ und „else“ einen Sinn, weil immer das zuerst erreichte „return“ ja zum Aufrufer zurückführt.

Hinter dem „return“ kann ich jeden Datentyp verwenden, also nicht nur einfache, wie Zahlen oder Strings, sondern auch Listen, Dictionaries usw., die ich an den Aufrufer zurückgebe.[55]

---

54 In der meist anglisierten Literatur als deep oder swallow copy bezeichnet.

55 Mein „Lieblingsfehler“ bei der Programmierung in C oder C++ bestand darin, Daten aus einer Funktion zurückzugeben, die erst in ihr erzeugt wurden. Das geht nicht! Wer also nebenbei auch Arduino Programme schreiben möchte, sollte dringend auf diesen Unterschied zwischen Python und kompilierenden Sprachen achten.

Wir hatten schon die Typ-Angaben für Parameter erörtert. Wie aber kann ich einen Typ-Hinweis für den Rückgabewert festlegen? Python benutzt dazu die Schreibweise „-> <Typ>“ zwischen schließender Klammer der Parameterleiste und dem Doppelpunk:

def tst(a: int, s: str="Hallo") **->** **str**:

5.5 Schönere Ausgaben

Bei unseren kleinen Programmen haben wir nun schon ganz oft „print“ gebraucht. Zum einen ist „print“ ein Beispiel für eine Funktion, die beliebig viele Argumente entgegennehmen kann. Zum anderen kennt sie einige Schlüsselwort Parameter. Mit „end=<Endezeichen>“ kann man den Default <neue Zeile> ändern, um z.B. alle Ausgaben hintereinander zu schreiben:

```
for i in range(10):
    print("-", end="")
# Ausgabe:
# ----------
```

Daneben gibt es noch u.a. „flush=True|False“ und „file=<Dateikennung>. Näher möchte ich hier nicht darauf eingehen.

Die einzelnen Argumente werden von „print“ mit einem Leerzeichen getrennt ausgegeben. Möchte man es schöner haben, kann man auf die sogenannten f-Strings zurückgreifen. Die sehen so aus:

```
name = "Anton"
print(f"Hallo {name}, wie geht es Dir?") # einfach ein „f“ vor dem String
# Ausgabe:
# Hallo Anton, wie geht es Dir?
```

Alles in geschweiften Klammern wird von Python verarbeitet. Eine Variable, wie unser „name“ wird durch ihren Wert ersetzt, eine Berechnung wie „12 * 12“ durch ihr Ergebnis. Auch Formatierungen sind möglich; als Beispiel will ich hier nur auf Zahlen eingehen. So sieht die Ausgabe von

```
print(27.12345678901234567890)
27.123456789012344
```

auf meinem Computer aus. Mit print(f“{27.12345678901234567890**:0.2f**}“) verändert sie sich zu:

```
27.12
```

Dafür verantwortlich ist der von mir fett gedruckte Teil. Der Doppelpunkt leitet den Formatierung-Text ein. Die Zahl vor dem Punkt sind die minimalen Stellen für die gesamte ausgegebene Zahl[56], die Zahl hinter dem Punkt gibt die Nachkommastellen an. Python rundet die Ausgabe kaufmännisch auf.

Nicht unerwähnt soll eine andere Art der Formatierung einer Ausgabe bleiben, da sie in einigen Fällen Vorteile[57] gegenüber f-Strings hat. Sie sieht sehr ähnlich aus:

```
"Hallo {}, wie {} es Dir {}".format("Anton", "ging", "gestern")
# Ausgabe:
# Hallo Anton, wie ging es Dir gestern
```

56 Eine 6 statt der 0 rückt die Ausgabe um eine Stelle ein: 27.12 hat 5 Stellen.

57 Ich habe eine Mehrsprachenunterstützung programmiert; da z.B. im Englischen der Satzbau anders ist, kann ich, ohne die Reihenfolge der Argumente zu ändern die Ausgabe syntaktisch richtig gestalten.

In die geschweiften Klammern dürfen Zahlen beginnend bei 0 geschrieben werden; sie ändern die Zuordnung der Argumente:

```
x="Hallo {1}, wie {0} es Dir {2}".format("ging", "Anton", "gestern")
#                                         andere Reihenfolge!
# Ausgabe:
# Hallo Anton, wie ging es Dir gestern
```

Auch die oben erwähnte Formatierung mit z.B. „:0.2f“ funktioniert hier.

5.6 Von Räumen und Sichtbarkeit

Wenn wir uns ein Programm als Haus vorstellen, dann strukturieren die Räume es. Sie schränken aber auch die Sichtbarkeit ein: Von der Küche kann ich nicht direkt ins Schlafzimmer schauen.

In einem Programm möchte ich nicht alles wie in einem einzigen, großen Raum haben, sozusagen ein Haus ohne Wände. Was bei einem sehr kurzen Programm noch vertretbar erscheint, wird mit zunehmender Länge immer unübersichtlicher. Mit einer Funktion erschaffe ich quasi ein neues Zimmer; was sich in diesem Zimmer befindet, ist versteckt. Eine Variable in einer Funktion ist auch nur während ihres Aufrufs gültig. Endet die Funktion, erlischt auch die Variable.

Damit sind zwei Merkmale von Variablen, Funktionen und Methoden (zu denen kommen wir später) angesprochen: Gültigkeit und Sichtbarkeit. Was meint das genau?

Die Variable in der definierten Funktion ist außerhalb weder sichtbar noch gültig. Wird die Funktion aufgerufen, wird ihre Variable gültig, bleibt von außen aber unsichtbar.

Gibt es eine gleichnamige Variable außerhalb der Funktion, wird sie in der Funktion ungültig[58]; die lokale Variable überschreibt sie aber nicht, sonder verdeckt sie nur. Endet die Funktion, kann auf die äußere Variable wieder zu gegriffen werden.

```
def verdecken(a):
    x = a
    print("In verdecken:",x, a)

x = 1
verdecken(10)
print("Nach verdecken:", x)
# Ausgabe:
# In verdecken: 10 10
# Nach verdecken: 1
```

Auch der Fall, dass ich die äußere Variable in der Funktion benutze, unterliegt klaren Regeln: Lesender Zugriff ist erlaubt, um auch schreibenden Zugriff zu erlangen, muss ich explizit die äußere Variable mit dem - reservierten - Wort „global“ versehen.

```
def nutzen(a):
    global x
    x = a
    print("In nutzen:", x, a)

x = 1
nutzen(10)
print("Nach nutzen:", x)
# Ausgabe:
```

58 Nur beim Überschreiben. Lesend ist die äußere Variable gültig.

```
# In nutzen: 10 10
# Nach nutzen: 10
```

Eine goldene Regel bei der Programmierung besagt: möglichst wenig „globale“ Variablen! Das wären Variablen, die überall im ganzen Haus gültig und sichtbar wären. Man braucht sie zwar manchmal, aber hier ist Sparsamkeit angesagt.

5.7 Feinheiten

Etwas außer der Reihe möchte ich hier auf einige Python Besonderheiten eingehen.

5.7.1 Anfang und Ende

Als wir über Strings gesprochen haben, wurde in den Beispielen stillschweigend z.B. „Text“[59] geschrieben. Ein Anführungszeichen am Anfang und eines am Ende. Daran erkennt der Interpreter einen literalen String. Die Anführungszeichen um „Text“ sind eigentlich doppelte, denn es gibt auch ‚Text‘, also einfache Anführungszeichen. Relevant wird das, wenn wir zwischen den Anführungszeichen wieder Anführungszeichen verwenden wollen: „Wie geht’s“. Hier sind nur doppelte Anführungszeichen möglich, weil bei einfachen der String hinter „geht“ zu Ende wäre[60]. Braucht man in einem String sowohl doppelte als auch einfache Anführungszeichen kann man auf die Schreibweise

```
"""Hallo, wie geht's im "Text" weiter?"""
```

ausweichen. Auch drei einfache Anführungszeichen sind erlaubt. Die dreifachen Anführungszeichen werden auch für die Quelltextdokumentation verwendet:

```
def tst(p1: str, p2):
    """Funktion macht dies oder das"""
    pass

help(tst)
# ergibt in Python:
# Help on function tst in module __main__:
# tst(p1: str, p2)
#     Funktion macht dies oder das
```

Die Ausgabe der Dokumentation wird aber nicht von Micropython unterstützt :-((

5.7.2 Trennen und Zusammenfügen

Um Listenoperationen anwenden zu können, braucht man eine Liste oder ein Tuple. Oft stehen die Daten aber nur als String zur Verfügung. Um ihn in seine einzelnen Buchstaben zu zerlegen, reicht die Funktion[61] „list“: „list(‚Hallo‘)“ erzeugt die Liste „['H', 'a', 'l', 'l', 'o']“.

In CSV Dateien[62] werden die einzelnen Elemente durch z.B. ein Komma getrennt. Eine Zeile könnte so aussehen: „Meng-Wacker,Gert,0205188356“, also die Elemente Name, Vorname und

---

59 Dass das erste Anführungszeichen unten steht, liegt an der Textverarbeitungssoftware!

60 Es gibt noch die Möglichkeit der sogenannten Quotierung, bei der man einen „\“ vor das Anführungszeichen setzt und es dann auch hier verwenden kann. Mit Quotierungen kann man auch Zeilenendezeichen in einen String einfügen: „\n“.

61 Genau genommen ist das keine Funktion, sondern ich instantiiere die Klasse „list“ mit dem String als Argument.

62 CSV (Comma Separated Values) ist ein weit verbreitetes Datenformat, das den Austausch zwischen unterschiedlichen Programmen ermöglicht.

Telefonnummer enthalten. Um daraus eine Liste zu machen verwendet man die String-Methode „split“ (deutsch aufteilen):

```
s = "Meng-Wacker,Gert,0xxx188356"
l = s.split(",") # aufteilen des Strings am Komma
print(l)
# Ausgabe:
# ['Meng-Wacker', 'Gert', '0xxx188356']
```

Jetzt kann ich z.B. eine neue Telefonnummer erfassen. Aber wie bekomme ich die Liste wieder zurück ins CSV Format? Mit der String-Methode „join“. Sie stellt die Umkehrung von „split“ bereit:

```
l[2] = '02051471148' # Änderung der Telefonnummer
j = ",".join(l) # das Trennzeichen wird vorangestellt
print(j)
# Ausgabe:
# 'Meng-Wacker,Gert,02051471148'
```

5.7.3 Der Walross-Operator

Der lustige Name leitet sich aus seiner Form ab: „**:=**“. Das soll wie die Nase und der Bart eines Walrosses aussehen. Naja.

Das, was dieser Operator bietet, ist schon erwähnenswert. Das wird an einem Beispiel am Besten deutlich.

```
import time
start = time.time() # gibt einen Zeitstempel zurück
while start + 10 < time.time(): # solange wahr, bis Startzeit + 10 Sekunden
    # nicht mehr größer als gerade ermittelt
    pass
print("Jetzt erreicht:", time.time())
```

Das kleine Programm kreist in der Schleife, bis die Startzeit plus 10 Sekunden erreicht sind. Das „pass“ ist ein Füllwort; wann immer syntaktisch Python Code verlangt wird, aber ich *nichts* tun will, schreibe ich „pass“ hin. Hier wird es gebraucht, weil der Körper der Schleife irgendetwas enthalten muss.

Wer genau hinschaut, kann feststellen, dass ich hier mogele. Ich schreibe „print("Jetzt erreicht:", time.time())“, aber „time.time()“ wird hier später aufgerufen, als der Aufruf, der die Schleife sich beenden ließ. Und jetzt komm das Walross ins Spiel. Es erlaubt nämlich in einem Ausdruck, hier "while ..." eine Zuweisung unterzubringen:

```
start = time.time() # gibt einen Zeitstempel zurück
print(start)
while start + 10 > (stop := time.time()): # solange wahr, bis Startzeit + 10
    # Sekunden nicht mehr kleiner als jetzt ist
    pass
print("Jetzt", stop, s := time.time(), s - stop)
```

Ich habe hier jetzt zwei mal davon Gebrauch gemacht, beide fett hervorgehoben. Beim ersten Mal wird der Zeitstempel[63] der Variablen „stop“ zugewiesen *und* mit „start + 10“ verglichen, beim zweiten Mal der Variablen „s“ um dann die Differenz zu „stop“ anzuzeigen.

63 Als Zeitstempel bezeichnet man eine Zeitangabe, die von einem Startzeitpunkt an immer weiter ansteigt. Damit lassen sich Zeiten messen: Stoppzeit - Startzeit = Differenz.

Auf meinem PC sieht die Ausgabe so aus: (1.9073486328125e-06 bedeutet ~1,9 µs)

```
1719837721.0057955
Jetzt 1719837724.0057955 1719837724.0057974 1.9073486328125e-06
```

Auf dem Pi Pico muss ich statt „time.time()“ „time.ticks_us()“ verwenden, um einen genauen Zeitstempel zu erhalten. Das Ergebnis: (130 bedeutet 130 µs)

```
181107340
Jetzt 181107793 181107923 130
```

In vielen Fällen hilft der Walross Operator, mehrfache, unnötige Funktionsaufrufe zu vermeiden.

5.7.4 Das Wörtchen „with“: der Kontextmanager

Habe ich z.B. eine Datei geöffnet, worauf wir später noch ausführlicher eingehen werden, sollte sie nach ihrer Nutzung auch wieder geschlossen werden.

Mit einem sogenannten Kontextmanager gibt es dafür eine elegante Möglichkeit. Dazu schreibt man:

```
with open("config.txt", "r") as f:
    while l := f.readline():
        print(l)
```

Nun wird die Datei in jedem Fall geschlossen, sobald der Block nach der Zeile „with ...“ endet.

Kontextmanager sind beliebt, weil der Code dadurch sehr übersichtlich bleibt. Natürlich darf man „with“ auch schachteln, also z.B. eine Datei öffnen und mit einer neuen „with“ Zeile eine weitere Datei.

Kontextmanager gibt es nicht nur für Dateien, sondern auch für andere Ressourcen, die verwaltet, d.h. geöffnet – geschlossen, angefordert – freigegeben werden müssen. Bei paralleler Programmierung mit „asyncio“ kann ich eine sogenannte Sperre (englisch lock) anfordern:

```
lock = asyncio.Lock()
async with lock:
    ...
```

Hier muss das "async" vorangestellt werden, damit Micropython weiß, dass es sich um etwas handelt, das parallel bearbeitet werden kann.

5.7.5 Iteratoren – die Unendlichkeit handhaben

Als wir Listen behandelt haben, konnten wir sehen, wie einfach es ist, mit einer for-Schleife die einzelnen Elemente abzuklappern[64]. Außerdem begegnete uns die „range“ Funktion, mit der man Zahlenreihen erzeugen kann. „range(10)“ in einer for-Schleife, gibt die Zahlen von 0 bis 9 aus.

Was macht wohl „range(1000000000000000000000000000000)“[65]? Platzt jetzt mein Computer? Die Zahl ist offensichtlich zu groß, um als Liste in den Speicher zu passen. Und dennoch, klaglos wird ein „for i in range(...“ akzeptiert[66].

---

64 Der Fachausdruck dafür ist „iterierbar“. Ganz viele Python Objekte lassen sich so, Element für Element, verarbeiten.

65 Hier muss ich ein Geständnis machen: In Micropython sind so große Ganzzahlen nicht immer möglich. Bei 999999999 war bei der range Funktion Schluss (auf dem Pi Pico mit Micropython v1.24) .

66 Ich muss allerdings eine Abbruchbedingung formulieren, sonst würde die for-Schleife fast ewig laufen.

Python greift hier zu einem Trick, der sich Iterator nennt. Dabei erzeugt „range“ gar keine Liste, sondern berechnet immer nur das nächste auszugebende Element. Deshalb verbraucht „range(100000)“ auch nicht mehr Speicherplatz als „range(10)“.

Man kann Iteratoren auch selber schreiben. Eine Möglichkeit dafür ist eine Klasse. Auf sie werden wir im Kapitel OOP noch einmal zurückkommen.

Die andere Möglichkeit besteht in der Nutzung eines (Iterator-)Generators. Am Beispiel einer Funktion „myrange“ zeige ich, wie einfach das geht:

```
# myrange.py
def myrange(von: int, bis: int=0, schrittweite: int=1):
    if bis == 0:
        bis = von
        von = 0
    while True:
        if von < bis:
            yield von
            von += schrittweite
        else:
            return

i = myrange(5)
for n in i:
    print(n)
print()
i = myrange(5, 10)
for n in i:
    print(n)
# Ausgabe:
# 0
# 1
# 2
# 3
# 4
#
# 5
# 6
# 7
# 8
# 9
```

Damit unser „myrange“ sich so verhält wie „range“ prüfe ich die übergebenen Argumente. Wird nur eines angegeben, wird es als „bis“ gewertet und bei Null begonnen. Das einzige Neue hier besteht im reservierten Wort „yield“. Es macht das Gleiche wie „return“ (Rückgabe des „von“ Wertes) mit einem Unterschied: Die Funktion wird nicht beendet, sondern bei der nächsten Iteration der Code nach „yield“ ausgeführt. In unserem Beispiel also „von += schrittweite“, danach zu „while True“ usw.

5.8 Funktionale Programmierung

Mit dem Begriff Programmierparadigma meint man z.B. die OOP. Funktionale Programmierung ist auch ein Paradigma. Es beschreibt eine Art zu programmieren, als wenn man mathematische Funktionsgleichungen erstellen wollte.

Python hat sich einige sehr praktische „funktionale“ Methoden von anderen Sprachen geklaut. Ich möchte hier nur die List-Comprehension (zu deutsch Listen-Abstraktion, was auch nicht verständlicher ist) vorstellen. Sie ist sehr nützlich und – schnell.

Nehmen wir an, wir haben eine Liste mit Temperatur- und Luftfeuchtigkeitswerten und einem Zeitstempel: [[22.3, 47, 1234567.29], [22.2, 48, 1234568.39], [22.1, 49, 1234569.29]].

Jetzt möchte ich nur die Temperatur zusammen mit dem Zeitstempel ausgeben:
[[22.3, 1234567.29], [22.2, 1234568.39], [22.1, 1234569.29]].

Mit der for-Schleife geht das prima. Aber es geht noch besser und, wie gesagt, schneller.

```
[[x[0], x[2]] for x in [[22.3, 47, 1234567.29], [22.2, 48, 1234568.39], [22.1,
49, 1234569.29]]]
```

Vor lauter Klammern sieht man den Wald nicht mehr; daher habe ich sie eingefärbt. Die Grünen sind nur ein Listen-Index, also nulltes und zweites Element von x. Die beiden Roten umschließen die Ergebnisliste:

```
[[22.3, 1234567.29], [22.2, 1234568.39], [22.1, 1234569.29]]
```

Und die sieht nun genau so aus, wie gewünscht. Das „for x in“ gleicht ja unserer for-Schleife. Es nimmt ein Element aus der Liste, die drei Unterelemente hat, und packt sie „nach vorn“. Dort wähle ich das nullte und zweite davon aus.

Die List-Comprehension verfügt noch über ein optionales „if“.

```
[[x[0], x[2]] for x in [[22.3, 47, 1234567.29], [22.2, 48, 1234568.39], [22.1,
49, 1234569.29]] if x[0] > 22.1]
```

Jetzt erhalte ich nur zwei Ergebnisse, da das letzte Element das Kriterium nicht erfüllt.

Auch hier wieder die Aussage: Alle Details, insbesondere die Abwandlungen wie dict-Comprehension, muss ich hier schuldig bleiben. Bitte im Internet selber nachschlagen.

Ein nettes Beispiel für die Caesar-Chiffre, ein simples, auf Julius Caesar zurückgehendes Kodierverfahren, möchte ich aber zum Abschluss noch zeigen. Zuerst die Funktion dieses Verschlüsselungsverfahrens. Caesar hat seine Briefe Buchstabe für Buchstabe so verschlüsselt, dass jedem Buchstaben eine Zahl zugewiesen wurde, A = 1, B = 2 usw. Dann hat er z.B. jeweils 3 dazu addiert und den dazu passenden Buchstaben geschrieben. So wird aus „CAESAR“ „FDHVDU“.

Das ganze „Programm“ sieht so aus:

```
[chr(ord(x)+3) for x in "CAESAR"] # Der String „CAESAR“ wird durch das „for“
                                  # in die einzelnen Buchstaben zerlegt
```

„ord“ ergibt die einem Buchstaben zugeordnete Zahl, „chr“ macht wieder einen Buchstaben aus einer Zahl. Wenn Cäsar das schon zur Verfügung gehabt hätte!

5.9 Callbacks und anonyme Funktionen

Für Python ist alles ein Objekt, auch eine Funktion (oder Methode). Objekte kann man als Argumente an Funktionen übergeben, also geht das auch mit Funktionen. Damit Python unterscheiden kann, wann man ein Funktionsergebnis (z.B. „len(„Hallo“) gibt 5 zurück) als

Argument übergeben möchte oder die Funktion selbst, schreibt man für Letzteres nur den Funktionsnamen ohne die sonst obligatorischen runden Klammern dahinter:

```
def add2(x: int) -> int:
    x = x + 2
    return x

def cb_tst(cb: callable[67], y: int):
    print(cb(y))

cb_tst(add2, 5) # die Funktion "add2" ohne Klammern!
# Ausgabe:
# 7
```

Hätte ich eine Funktion „sub2“, die 2 subtrahiert, könnte ich „cb_tst(sub2, 5)“ aufrufen und erhielte 3.

Am Beispiel Sortieren zeige ich nun, warum Callbacks *wirklich* nützlich sind. Gegeben sei folgende Liste:

```
l = ["cde","b", "ab", "CDE"]
```

Die Funktion „sorted“ sortiert eine Liste und gibt sie auch wieder als Liste zurück:

```
sorted(l)
# Ausgabe:
# ['CDE', 'ab', 'b', 'cde']
```

OK, die Funktion sortiert anscheinend alphabetisch. Wer Lust hat, kann ausprobieren was sie mit Zahlen macht ;-)

Jetzt möchte ich aber eine Sortierung auf Basis der Länge der Strings. Um das zu erreichen, kann ich einen weiteren Parameter[68], einen Callback angeben:

```
sorted(l, key=len)
# Ausgabe:
# ['b', 'ab', 'cde', 'CDE']
```

Die Funktion „len“ ermittelt die Anzahl der Elemente dessen, was man ihr übergibt: bei Strings die Anzahl der Zeichen.

Intern arbeitet „sorted“ so, dass es für je zwei aufeinander folgende Elemente den Callback aufruft und die Ergebnisse vergleicht.

Mir gefällt die Ausgabe immer noch nicht. Bei gleicher Länge wünsche ich mir dann eine alphabetische Ordnung (also „CDE“ vor „cde“). Jetzt habe ich zwei Möglichkeiten: Ich schreibe eine passende Funktion[69] und übergebe sie „sorted“ oder – ich verwende eine sogenannte Lambda oder anonyme Funktion:

```
sorted(l, key=lambda x: (len(x), x))
# Ausgabe:
# ['b', 'ab', 'CDE', 'cde']
```

67 Ich verwende den Typ „callable“ als Hinweis an den Leser, dass es sich um eine Funktion oder Methode handelt; Micropython kennt das Modul „typing“ des großen Pythons nicht, indem dieser Typ enthalten ist.

68 Die Funktion erwartet einen sogenannten Schlüsselwort-Parameter. Siehe dazu das gleichnamige Kapitel.

69 Passend bedeutet hier, dass zuerst nach Länge sortiert wird und dann, bei gleichen Elementen, nach Alphabet. Python und die „sorted“ Funktion machen es uns wieder einfach, da man auch Listen direkt vergleichen kann. Eine Liste [5, „CDE“] ist kleiner als [5, „cde“].

Dabei liefert „lambda“ ein Funktionsobjekt ohne Namen, daher anonym, zurück. In „x“ wird das jeweilige Element übergeben; der Rückgabewert besteht aus einer Liste mit der Länge des Strings und dem String selbst. Diese Liste wird mit der Ergebnisliste für das nächste Element verglichen. Ist das erste Element gleich, wird das Zweite berücksichtigt: „cde“ und „CDE“ haben die gleiche Länge, daher werden die Zweiten verglichen und „CDE“ vor „cde“ angeordnet.

Lambdas in Python haben noch eine Eigenschaft, die wichtig ist. Sie „frieren“ nämlich ihren Programm-Kontext ein. Das hört sich komplizierter an, als es ist:

```
def lb(s: str) -> int:
    return lambda x=s: print(len(x), x)

z = lb("hugo")
print(z)
z()
z("willibald")
# Ausgabe:
# <function xx.<locals>.<lambda> at 0x7fe4c10516c0>
# 4 hugo
# 9 willibald
```

Der Aufruf der Funktion „lb“ mit dem Argument „hugo“ gibt ein Lambda-Objekt zurück, das wir in „z“ speichern. Mit „print(z)“ zeigen wir, dass in „z“ diese Lambda-Funktion gespeichert wurde.

Die Aufrufe „z()“ und „z(„willibald“) demonstrieren, dass sich die Funktion an den Parameter „s“ erinnert und bei direkter Übergabe eines anderen Arguments dies überschreibt.

Informatiker nennen das ein „Closure“.

5.10 Dateioperationen – aber auf dem Pi Pico?

Python unterstützt als große Programmiersprache natürlich das Arbeiten mit Dateien. Aber was sind eigentlich Dateien? Klar, wir schreiben einen Text und speichern ihn ab oder wir öffnen ein Programm, das die notwendigen Daten aus einer Datei ließt. Aber was geschieht da genau?

Wenn wir von der Hardware-Ebene abstrahieren, also Festplatte, USB-Speicher, CD-ROM usw., dann bleibt eine große, les- und beschreibbare „Fläche“ übrig. Üblicherweise wird diese dann in Blöcke mit fester Größe eingeteilt, oft auch Sektoren genannt. Jeder Block erhält eine Nummer, die dann wieder bei Null anfängt und geht weiter, bis die ganze Fläche durchnummeriert ist. Übliche Größen solcher Blöcke sind 512, 1024 oder 2048 Byte. Informatiker arbeiten gerne mit Binärzahlen.

Der Einfachheit halber nehmen wir die Blöcke zu 1000 Bytes an, das lässt sich gut rechnen. Unser Datenspeicher sei 10 Millionen Byte groß. Dividiert durch 1000 ergibt 10000 Blöcke. Wenn ich jetzt ein Python Programm darauf speichern möchte, suche ich mir einen Block aus, sagen wir den mit der Nummer 100 und schreibe meinen Text in diesen Block. Ist der einzelne Block zu klein dafür, mein Text wäre z.B. 1100 Byte lang, so schreibe ich den Rest in Block 101.

Schreiben heißt hier jeweils, dass die Hardware meine Daten auf den Datenträger bringt. Lesen bedeutet dann umgekehrt, dass die Hardware die Daten vom Datenträger wieder in meinen Arbeitsspeicher kopiert. Dann brauche ich noch einen Mechanismus, der den Block auswählt. Da das aber eine reine Rechenoperation ist, Blocknummer * 1000, dann den Schreib- oder Lesebeginn

auf das errechnete Byte setzten, stellt das auch kein Problem dar. Innerhalb des Blocks muss ich mich dann auch um die exakte Schreibposition kümmern: Das erste Zeichen auf Platz 0, das zweite auf Platz 1 usw.

Das funktioniert so gut, dass es Sprachen wie Forth[70] gibt, die genau dieses Verfahren benutzen.

Als Grundlage dient dieses Verfahren *jedem* Dateisystem. Aber, wie an dem Beispiel ersichtlich, muss sich hier der Anwender sehr viel merken. Welche Blocknummer hat mein Text? Gibt es Folgenummern? Welche? Und was geschieht, wenn ich meinen Text ändere, wenn er länger oder kürzer wird?

Fragen über Fragen! Die Antworten liegen in dem darüber liegenden Dateimanagement. Das hat in der Regel Namen wie NTFS, exFAT, FAT32 (Windows) oder Ext3, Ext4, Btrfs (Linux). Sie kümmern sich ums Vergrößern, Verkleinern von Dateien, erlauben Namen dafür anstatt einer Blocknummer usw.

In Micropython und Python benötigt eine Datei einen Namen, gegebenenfalls einen Verzeichnisnamen. In Micropython schreibt man am Besten alles (und das ist nicht so viel) ins Wurzelverzeichnis. Man muss eine Datei explizit öffnen und schließen, damit die Verwaltung optimal arbeiten kann. Innerhalb einer Datei wird von den zugrunde liegenden Blöcken abstrahiert. Die Datei fängt bei Byte Null an und geht linear bis zu ihrem Ende. Es gibt ein Kommando, um den Lese- oder Schreibzugriff an eine beliebige Byte-Position zu setzen. Beide Zugriffe setzen automatisch diese Byte-Position weiter. Also lesen ab Position 100 von 10 Bytes ergibt einen Dateizeiger auf Position 110.

Eine Binär-Datei hat keine Struktur, sie ist nur eine Bytefolge. Ich kann aber selber eine Struktur festlegen, indem ich Datensätze einer festen Länge definiere. Auf sie kann ich dann wieder durch eine Nummer, den n-ten Datensatz, zugreifen. Oft handelt es sich aber um Text. Der unterscheidet sich von binären Daten zum Einen dadurch, das Buchstaben in Unicode kodiert sind, ein Zeichen also aus mehr als einem Byte bestehen kann. Eine Textzeile kann beliebig lang sein und endet mit einem Zeilenendezeichen. Weil das so häufig vorkommt, gibt es in Python spezielle Kommandos, um Zeilen zu lesen und zu schreiben. Es gibt aber keine Möglichkeit, gezielt z.B. die 18. Zeile zu lesen. Da die Zeilen unterschiedlich lang sind, muss man sich von der ersten bis zur 18. durchhangeln.

Die Frage, wozu ich Dateioperationen auf dem Pi Pico brauche, habe ich aber immer noch nicht beantwortet. Ich gebe zwei Beispiele. Ich messe mit dem DHT22 die Temperatur und schicke die Messdaten über WLAN an einen Server. Fällt das WLAN aus, gehen meine Messdaten verloren – es sei denn, ich schreibe sie jetzt ins Dateisystem! Da ich sie mit einem Zeitstempel versehe, kann ich sie später an den Server schicken und die Datei wieder löschen.

Meine Pi Picos laufen eigentlich Tag und Nacht, über Wochen und Monate. Dabei tun sie klaglos ihren Dienst. Was aber, wenn nicht? Würde ich einen nun an meinen Computer anschließen, um zu sehen, warum er nicht läuft, müsste ich ihn ja zuvor ausschalten. Die Ursache wäre wahrscheinlich nicht mehr herauszufinden. Was aber, wenn mein Pi Pico Probleme ins Dateisystem schreibt, wo ich sie später auslesen kann?

70 Forth ist eine Programmiersprache und auch sehr gut für Mikrocontroller geeignet. Es verfügt ebenfalls über eine interaktive Eingabe und kompiliert ad hoc.

Es gibt also gute Gründe, sich mit Dateioperationen zu befassen.Und es ist auch gar nicht schwer.

```
# datei_tst.py
f = open("tst.txt", "w") # eine Datei zum Schreiben ("w" wie write) öffnen
f.write("Hallo!\n")  # eine Zeile schreiben. Endezeichen ist "\n" ("newline")
f.write("Wie\n")     # noch eine
f.write("geht es\n") # und noch eine
f.close()            # Datei schließen

f = open("tst.txt", "r") # Datei lesend öffnen ("r" wie read)
r = f.readline()         # eine Zeile lesen, bis einschließlich Zeilenende
print(r)                 # ausgeben
r = f.readlines()        # alle Zeilen lesen, aber ab Dateizeiger!
print(r)                 # ausgeben
r = f.readlines()        # keine Daten mehr vorhanden
print(r)                 # leere Liste
f.seek(0)                # Dateizeiger auf Anfang
r = f.readlines()        # alle Zeilen lesen
print(r)                 # ausgeben
f.close()                # Datei schließen
# Ausgabe:
# Hallo!

# ['Wie\n', 'geht es\n']
# []
# ['Hallo!\n', 'Wie\n', 'geht es\n']
```

Die Leerzeile hinter dem ersten"Hallo!" wird durch das Zeilenendezeichen „\n“ erzeugt.

5.11 (Micro)pythons Helferlein

Ich möchte hier einige wichtige, eingebaute Funktionen vorstellen, die bei der Arbeit mit (Micro)Python hilfreich sind. Als interpretierende Sprache muss es sogenannte Meta-Informationen über seine Laufzeit vorhalten, weil alles dynamisch verändert werden kann. Die hier vorgestellten Funktionen greifen auf auf diese Meta-Informationen, die oft in Form von internen Dictionaries vorliegen, zurück. Das kann man auch von Hand tun, nur muss man dann die Namen dieser – versteckten – Datenstrukturen kennen.

5.11.1 Die Funktion type

Da wäre die type() Funktion, die eigentlich eine Methode ist, aber ich will es nicht zu kompliziert machen. Die Eingabe von

```
type(1.0)
```

liefert: <class 'float'>. Aha, float, also Fließkommazahlen, sind eine Klasse. Was liefert

```
type(„Hallo“)
```

zurück? <class 'str'>! Ich erfahre also, mit welchem Daten-Typ ich es zu tun habe. Ziemlich oft wird mir dabei „class“ <irgendetwas> genannt; Python macht intensiven Gebrauch von Klassen, gerade für interne Funktionalitäten. Wenn ich

```
type(print)
```

eingebe, ist die Anwort <class 'builtin_function_or_method'> in Python oder <class 'function'> in Micropython. Eine „Funktion“ ist auch eine Klasse! Der Versuch

```
type(while)
```

abzufragen, scheitert mit einer Fehlermeldung:

```
Traceback (most recent call last):
File "<stdin>", line 1
SyntaxError: invalid syntax
```

Python verfügt über - nicht allzu viele - fest eingebaute Befehle; ihre Namen sind reservierte Worte[71], d.h., der Versuch, eine Funktion „while“ zu nennen, scheitert.

So ein Scheitern kann in Python recht lang aussehen, die Fehlermeldung oben ist schon die ziemlich kürzeste. Eingeleitet werden sie mit „Traceback“ und dann kommt der Versuch, die Fehlerursache und den Ort ihrer Entstehung möglichst genau zu beschreiben. Der Hinweis „most recent call last“ macht deutlich, dass die jüngste Fehlermeldung am Ende kommt:

```
def x():                                   # 1
    def y():                               # 2
        print(1/0)                         # 3
    y()                                    # 4
x()                                        # 5
Traceback (most recent call last):
File "<stdin>", line 5, in <module>
File "<stdin>", line 4, in x
File "<stdin>", line 3, in y
ZeroDivisionError: divide by zero
```

Hier habe ich zwei Funktionen ineinander geschachtelt (ja, das geht), um zu demonstrieren, was das meint. Eine Division durch Null geht nie, also tritt ein (Laufzeit)-Fehler[72] auf. Diese Meldung steht ganz unten; dann erfahren wir, nach oben aufsteigend, dass der Fehler in Zeile 3 in y, dann in Zeile 4 in x und dann in Zeile 5 in <module> (das ist der aktuelle Programmtext) auftrat.

5.11.2 Die Funktion dir

Mit Hilfe von dir() können wir schnell ermitteln, was in einer Klasse oder einem Modul (beides wird weiter unten näher erläutert) alles enthalten ist. Auf dem Pi Pico konnten wir bei Eingabe von „dir()“ z.B. folgendes sehen:

```
['machine', 'x', '__name__', 'rp2']
```

Das „x“ ist der Name der Funktion x(), die ich oben zur Fehlerdemonstration benutzt habe. Wo ist das „y“? Das es in “x“ enthalten ist, wird es nicht angezeigt. Die Ausgabe von dir() betrifft die oberste Ebene. Wenn ich dir(machine) eingebe, erhalte ich diese lange Liste:

```
['__class__', '__name__', 'ADC', 'I2C', 'I2S', 'PWM', 'PWRON_RESET', 'Pin',
'RTC', 'SPI', 'Signal', 'SoftI2C', 'SoftSPI', 'Timer', 'UART', 'WDT',
'WDT_RESET', '__dict__', 'bitstream', 'bootloader', 'deepsleep',
'dht_readinto', 'disable_irq', 'enable_irq', 'freq', 'idle', 'lightsleep',
'mem16', 'mem32', 'mem8', 'reset', 'reset_cause', 'soft_reset',
'time_pulse_us', 'unique_id']
```

Mit dir() und type() kann ich mich durchhangeln und mir die Informationen zur Nutzung besorgen. Manchmal geht es aber schneller, die Internetseiten von Micropython aufzurufen …

71 Das sind vor allem die sog. Kontrollstrukturen, while, if, else aber auch def, class usw.
72 Ein sogenannter Syntaxfehler verhindert, dass das Programm überhaupt startet.

5.11.3 Hilfe! Help!

Python hat eine eingebaute Hilfe-Funktion: help(). Bei ihrem Aufruf dokumentiert sie selbst, welche Hilfen sie anbietet. Auch Micropython verfügt über diese Funktion. Hier dient sie vor allem dazu, die im Flash-Speicher vorhandene Module aufzulisten. Dies geschieht durch Eingabe von help(„modules“), hier im Editor Mu:

```
>>> help('modules')
__main__            array               framebuf            random
_asyncio            asyncio/__init__    gc                  re
_boot               asyncio/core        hashlib             requests/__init__
_boot_fat           asyncio/event       heapq               rp2
_onewire            asyncio/funcs       io                  select
_rp2                asyncio/lock        json                socket
_thread             asyncio/stream      lwip                ssl
_webrepl            binascii            machine             struct
aioble/__init__     bluetooth           math                sys
aioble/central      builtins            micropython         time
aioble/client       cmath               mip/__init__        uasyncio
aioble/core         collections         neopixel            uctypes
aioble/device       cryptolib           network             urequests
aioble/l2cap        deflate             ntptime             webrepl
aioble/peripheral   dht                 onewire             webrepl_setup
aioble/security     ds18x20             os                  websocket
aioble/server       errno               platform
Plus any modules on the filesystem
>>> 
```

Wir kennen schon die Module „machine“ und „time“.

Es kann eine Klasse oder Funktion / Methode angegeben werden. In Python wird dann eine sehr übersichtliche Auflistung der Elemente geliefert. In Micropython geschieht das nur sehr rudimentär. In jedem Fall werden für einen Klasse, auch eine eingebaute, alle Methoden gelistet. Hier rechts sehen wir die Micropython Ausgabe von „help“ für die Klasse „str“ (String). Ganz schön viele Methoden!

```
>>> help(str)
  object <class 'str'> is of type type
    find -- <function>
    rfind -- <function>
    index -- <function>
    rindex -- <function>
    join -- <function>
    split -- <function>
    splitlines -- <function>
    rsplit -- <function>
    startswith -- <function>
    endswith -- <function>
    strip -- <function>
    lstrip -- <function>
    rstrip -- <function>
    format -- <function>
    replace -- <function>
    count -- <function>
    partition -- <function>
    rpartition -- <function>
    center -- <function>
    lower -- <function>
    upper -- <function>
    isspace -- <function>
    isalpha -- <function>
    isdigit -- <function>
    isupper -- <function>
    islower -- <function>
    encode -- <function>
```

Einige davon werde ich später noch behandeln, aber um alle zu besprechen, müsste ich noch mindestens zwei Bücher schreiben.

Vielleicht fällt jemandem auf, dass es eine Reihe von Methoden gibt, die mit „is“ anfangen. Mit dieser Konvention macht Python deutlich, dass die Methode einen boolschen Wert zurück gibt. „isupper“ liefert True, wenn alle Zeichen des Strings Großbuchstaben sind.

5.12 Module und Pakete

Wenn wir mit Python arbeiten, kommen wir immer wieder an einen Punkt, wo das, was wir gerade brauchen, nicht da ist. Klar, wir können uns das selber schreiben ;-)) Aber vielleicht hat das ja schon mal jemand …? Genau, meistens hat. Wie komme ich daran? Über PyPi, den Python Package Index. Das ist ein wirklich riesiges Archiv von Python Programmen für alle möglichen Zwecke. Wie aber finde ich das, was ich brauche? Hier hilft das Internet. Ich beschreibe in der Suchmaschine möglichst genau, was ich haben möchte und erhalte im Idealfall einen Paketnamen. Für den PC muss ich nun nur noch

```
# pip install <Paket Name>
```

in der Konsole eingeben. Ohne zu sehr ins Detail gehen zu wollen: Pakete enthalten ein oder mehrere Module und lassen sich importieren. Danach steht das importierte Programm zur Verfügung. Enthält das Paket nur ein Modul, dann sieht ein Import so aus:

```
import mod
mod.Class1 # Klasse Class1 in Modul / Paket „mod“
```

Wenn Paket pac die Module mod1 und mod2 enthält, dann bewirkt:

```
import pac
pac.mod1.Class1 # Klasse Class1 aus Modul mod1
```

den Zugriff auf die Klasse Class1 in Modul mod1.

Für Micropython kommt noch die Besonderheit dazu, dass ganz viele Module zwar schon im Mikrocontroller vorhanden, d.h. in das Micropython-Image hineingepackt, aber nicht sichtbar sind. Um zu sehen, welche Module eingebaut sind, kann ich, wie oben beschrieben, im REPL help(„modules“) eingeben. Brauche ich ein auch dort nicht gefundenes Modul, suche ich im Internet und erhalte hoffentlich eine Seite, die den Quelltext des Moduls enthält. Den speichere ich erst auf dem PC[73] und dann, z.B. mit Mu, auf dem Mikrocontroller in dessen Flash-Dateisystem.

Noch habe ich gar nicht verraten, was beim Importieren alles geschieht. Der Interpreter ließt die zum Paket gehörigen Dateien, dann kompiliert er sie. Stößt er dabei auf Fehler, bricht er ab. Alle enthaltenen globalen Variablen werden initialisiert, alle Klassen, Funktionen usw. bereitgestellt. Das alles dauert und verbraucht Speicher!

Es gibt verschiedene Methoden, um zu importieren. Die einfachste, z.B.

```
import machine [as <Alias>]
```

kennen wir schon. Optional kann nach dem Modulnamen ein Alias erzeugt werden; würden wir „as m“ schreiben, könnten wir auf „Pin“ durch „m.Pin“ zugreifen. Enthält ein Paket mehrere Module oder ein Modul mehrere Klassen oder Funktionen, bietet sich an, nur das zu importieren, was man wirklich braucht:

```
from machine import Pin
```

Damit ändert sich auch die Schreibweise unseres Beispiels:

```
pin13 = Pin(13, Pin.OUT)
```

Ein mehrfacher Import, sei es in der Variante mit Alias oder durch „from“ verbraucht übrigens keinen[74] zusätzlichen Speicher.

Wird in einem Modul ein Import codiert, z.B. „import socket“, steht er dem importierenden Programm *nicht* zur Verfügung. Das Modul muss erneut importiert werden, wenn man es braucht.

Ein Löschen eines Imports ist nicht vorgesehen. Daher sollte man Module nicht auf Verdacht importieren, nach dem Motto, vielleicht benötige ich es irgendwann mal. Python meldet sich, wenn man vergessen hat, ein Modul zu importieren.

Wenn wir später - auch kleine - Programme schreiben, werden wir feststellen, dass man selbst dann selten ohne Importe auskommt. Das liegt daran, dass Python bemüht ist, nicht so viel fest Eingebautes gegebenenfalls unnötig mit sich herumzuschleppen. Das kommt auch den Entwicklern von Python zugute, die dadurch viel modularer an Verbesserungen arbeiten können.

5.12.1 Mal so – mal so

Die meisten Importe scheinen aber, geben wir den Befehl z.B. im REPL ein, nichts Sichtbares auszulösen. Ein „import time“ stellt zwar Klassen und Methoden bereit, aber die Wirkung ist allenfalls durch einen „dir()“ Befehl zu bemerken. Später werden wir selbst ein Modul erstellen,

73 Ich kann ihn natürlich auch direkt auf dem Pi Pico speichern. Zum ausprobieren goldrichtig, aber für die Zukunft doch lieber in einem Arbeitsordner für das Projekt.

74 Je nachdem doch ein wenig, aber eben nicht doppelt so viel.

und es wird, sobald wir „import wconn“ eingegeben haben, eine WLAN Verbindung herstellen. Woran liegt das, was bewirkt solche Verhaltensunterschiede?

Kommen wir noch einmal auf die Funktionsdefinition zurück. Sie wird vom Interpreter „kompiliert“, aber nicht ausgeführt. Bevor wir sie nicht explizit aufrufen, geschieht nichts.

Befinden sich in einem Modul nur Definitionen, passiert dasselbe. Steht aber in unserem Modul ein Funktionsaufruf, dann wird der beim Import auch ausgeführt.

Als Programmierer muss ich mir genau überlegen, ob ich das will! Die meisten Module vermeiden es zwar, aber es gibt eben auch gute Gründe dafür.

Schreibe ich an einem Modul, möchte ich es auch testen. Anstatt nun eine Hilfsdatei anzulegen, von der aus ich mein Modul importiere und die darin befindlichen Funktionen aufrufe, kann ich auch im Modul[75] schreiben:

```
if __name__ == "__main__":
    # teste dies
    # teste das
```

Die Variable „__name__“ enthält nur dann den Wert „__main__“, wenn ich das Modul als Programm starte, nicht aber beim Import.

5.13 Dynamischer Speicher und der Müllsammler

Der Python Interpreter lässt von Zeit zu Zeit und unter bestimmten Bedingungen ein Programm laufen, das sich auf englisch Garbage Collector nennt. Zu deutsch heißt das Müllsammler. Aber welcher „Müll“ wird denn da gesammelt?

Jetzt muss ich etwas ausholen …

Wenn ich in einem Programm eine Variable x deklariere und ihr den Inhalt 42 zuweise, weiß ein C-Kompiler ganz genau, wie viel Speicherplatz im RAM er dafür benötigt. Selbst wenn der Wert der Variablen später auf 1000 gesetzt wird, ändert sich am Speicherplatzbedarf nichts. Warum? Weil der C-Kompiler dafür einige Byte im RAM reserviert (z.B. so viel, wie der Architektur des Mikroprozessors entspricht, also für den Pi Pico 32 Bit oder 4 Bytes). Im Rahmen der mit 32 Bit darstellbaren Zahlen kann also jeder Wert gespeichert werden.[76]

Python erlaubt aber, zuerst der Variablen x 42 zuzuweisen und dann den String „Hallo, guten Morgen“ in x zu speichern, der ersichtlich mehr als 4 Bytes benötigt. Oder es erlaubt eine Liste beliebig zu erweitern. Dafür benötigt man eine sogenannte dynamische Speicherverwaltung. Wenn man sich den Speicher als fortlaufende Ein-Byte Folge vorstellt, dann hat jedes Byte eine eindeutige Adresse, nämlich eine bei Null beginnende und für jedes Byte um eins steigende Zahl. Benötige ich nun 19 Bytes für den String, sucht die Speicherverwaltung einen mindestens 19 Byte[77] freien Platz im Speicher und merkt sich Anfang und Ende. Dieser Vorgang des Anforderns von Speicherplatz wiederholt sich nun ständig. Wird eine Variable nicht mehr gebraucht, kann ihr Speicherplatz auch

75 So etwas steht meist ganz am Ende in einem Modul.

76 Auch in C gibt es dynamischen Speicher. Die „Verwaltung“ erfolgt aber von Hand durch den Programmierer.

77 Die tatsächliche Größe eines Strings ermittelt mit „sys.getsizeof(<Variable>)“ beträgt in Version 3.13.0 von Python 41 Bytes plus die Anzahl der Bytes, die ein Text in Unicode verbraucht. Eine entsprechende Funktion existiert in Micropython nicht; die Größe dürfte aber ähnlich sein.

wieder freigegeben werden. Dadurch entstehen aber Lücken im Speicher. Und irgendwann kann die Speicherverwaltung eine Anforderung nicht mehr erfüllen, nicht unbedingt, weil der Speicher komplett voll ist, sondern weil die angeforderte Größe in keine der Lücken mehr hineinpasst.

Spätestens jetzt, lieber etwas früher, ist die Arbeit der Müllabfuhr gefragt. Die geht jetzt durch den Speicher und versucht, aus den vielen kleinen Lücken wieder eine Große zu machen!

Dazu muss sie Daten im Speicher verschieben, was bedeutet, sie zu kopieren und dann das Original zu löschen. Dafür braucht die Speicherverwaltung aber Speicher. Bei ganz vielen, ganz kleinen Lücken kann das scheitern und wir erhalten eine Laufzeitfehlermeldung.

Was passiert nun, während die Müllabfuhr werkelt? Genau: nichts! Mein Programm „steht", bis die Müllabfuhr endlich fertig ist. Da das zu unvorhersehbaren Zeiten geschehen kann, ist auch mein Python Programm zeitlich nicht exakt vorhersehbar. Stellen wir uns vor, wir steuern die Drehgeschwindigkeit eines Motors, indem wir Pulse in einer bestimmten Geschwindigkeit ausgeben. Arbeitet die Müllabfuhr, wird der Motor langsamer, bleibt vielleicht sogar stehen. Eine kritische Situation!

Eine Umgebung, die mit solchen kritischen Situationen wohldefiniert umgehen kann, nennt man Echtzeit-Umgebung. Es gibt sie für Computer als Betriebssystem oder für Mikrocontroller als Laufzeitsystem, aber nicht für Micropython.

In Micro(-Python) können wir aber Maßnahmen ergreifen, um die Situation weniger kritisch zu machen. Eine ist die, dass wir die Müllabfuhr selbst aufrufen[78] können. Damit bestimmen *wir* den Zeitpunkt. Zum anderen können wir auf Hardware-Module zurückgreifen wie die schon erwähnte PWM. Deren Geschwindigkeit ist unabhängig von der Müllabfuhr. Und wir können Speicheranforderung in kritischen Programmteilen vermeiden, indem wir nur Datenstrukturen mit fester Speichergröße[79] verwenden.

5.14 Logisch?

Python kennt gleich zwei Arten von Logik. Wenn wir ein „if" oder ein „while" verwenden, muss der Ausdruck dahinter sich zu True oder False auswerten lassen. Python macht es einem einfach, weil letztlich alles dazu wird: eine 0 zu False, alles andere zu True usw. Das habe ich schon im Kapitel „Schleifen und Entscheidungen" beschrieben.

Es gibt aber noch eine komplexere Form, bei der Vergleiche kombiniert werden mit UND und ODER. In Python heißt das dann „and" und „or":

```
if x > 10 or y < 5:
    ...
```

Wenn also die Variable „x" größer als 10 wird ODER „y" kleiner 5, dann ist die Gesamtbedingung True.[80]

78 Dazu importiert man das Modul "gc" und ruft "gc.collect()" auf.
79 Die Klassen „array" und „bytearray" stellen z.B. so etwas zur Verfügung.
80 Für logische Operatoren gibt es sogenannte Wahrheitstabellen.
Für „**or**" sieht die so aus:
False or False = False
True or False = True
False or True = True

Ein Mikrocontroller näheres Beispiel:

```
pin13 = machine.Pin(13, machine.Pin.IN, machine.Pin.PULL_UP)
n = 0
while n < 100 or pin13.value() == 1:
    # warte auf Drücken des Taster, aber nur solange n < 100
    time.sleep(0.3)
    n += 1
```

Ohne die Abfrage „n < 100“ würde die Schleife ohne Tastendruck endlos laufen; durch den Zähler „n“ wird nach 0,3 * 100 Sekunden die Schleife verlassen.

Passt die Logik nicht, gibt es noch das Wort „not“ zur Negierung. „not True“ ist falsch und „not False“ ist richtig.

Die zweite Gruppe von logischen Operatoren arbeitet auf Bit-Ebene, also nur für ganze Zahlen. Wieder gibt es UND und ODER und NICHT, aber die Operatoren sind Zeichen: „&“ (bitweises UND), „|“ (der senkrechte Strich, bitweises ODER), „~“ (bitweises NICHT), „^“ (bitweises EXKLUSIV ODER[81]).

Es gibt auch zwei sogenannte Schiebebefehle. Wenn wir uns ein CPU Register[82] mit einer 1 darin vorstellen, sieht das bei 8 Bit Breite[83] so aus: 0000 0001. Das Leerzeichen dient nur der besseren Lesbarkeit. Ein links-schiebe Befehl „1 << 3“ schiebt die 1 drei mal nach links: 0000 1000. Das entspricht einer dezimalen 8. „8 >> 3“ (rechts-schiebe Befehl) ergibt dann wieder 1. Mathematisch entspricht das übrigens Multiplikationen bzw. Divisionen mit 2 hoch Anzahl der Verschiebungen.

Solche bitweisen Operationen braucht man häufig, wenn hardwarenah gearbeitet werden muss. Dann werden nämlich oft einzelne Bits gesetzt (auf 1) oder gelöscht (auf 0), um bestimme Status zu signalisieren.

5.15 Das muss hübscher werden: Decoratoren

Wenn man, was sehr zu empfehlen ist, Python Quelltexte von anderen Leuten ließt, fällt einem vielleicht auf, dass manchmal über einer Funktion oder Methode ein „@<Name>“ steht. Ein Beispiel findet sich auch schon hier im Text, etwas versteckt in einer Anmerkung.

Die Verwendung schon vorhandener Decoratoren, wie sie genannt werden, ist sehr einfach: Man schreibt sie direkt über die zu dekorierende Funktion.

Wie sie funktionieren und wie man eigene Decoratoren erstellt, mutet dagegen wie Quantenphysik an. Aber auch die kann man, wenn richtig erklärt, verstehen. Versuchen wir es!

True or True = True
Das gleiche für „**and**“:
False and False = False
True and False = False
False and True = False
True and True = True

81 Während es bei ODER egal ist, ob nur eine oder beiden Seiten True sind, darf beim exklusiven ODER nur eine Seite True sein um True zu ergeben.

82 Die CPU merkt sich Zahlen und arbeitet mit ihnen in sogenannten Registern. Die sind in der Regel so breit wie die CPU Architektur: 16 Bit, 32 Bit, 64 Bit usw.

83 Auch bei Binärzahlen ist die niederwertigste Stelle rechts.

Im Kapitel „Callbacks und anonyme Funktionen“ tauchte der Name „Callback“ erstmals auf. Ein „Callback“ ist eine Funktion, die als Argument an eine andere Funktion übergeben wird. Nicht der Rückgabewert, also ein Argument wie in „print(pin13.value())“, sondern die Referenz auf die Funktion: „pin13.value“, also *ohne* die runden Klammern dahinter. Was bei „print“ überhaupt keinen Sinn ergibt, wird vielleicht verständlicher in folgendem Beispiel:

```
import time

def warten(cb: callable, w: float=1.0):
    time.sleep(w) # warten für „w“ Sekunden
    cb(f"{w} Sekunden vorbei")

def cb(txt: str):
    print(txt)

warten(cb, 5)
print("Wieder Hauptprogramm ...")
# ergibt nach 5 Sekunden die Ausgabe:
# 5 Sekunden vorbei
# Wieder Hauptprogramm
```

Eine Funktion, genauer die Referenz auf eine Funktion, kann also wie ein beliebiges anderes Argument an eine Funktion übergeben werden. Eine Funktions-Referenz kann aber auch von einer Funktion zurückgegeben werden. Dazu ein anscheinend sinnfreies Beispiel:

```
def rahmen(cb):
    var_local = "gemerkte_Variable" # zeigt, dass sich die Funktion etwas merkt

    def innen(*args):
        print("innen, Variable von rahmen:", var_local, ", Args:", args)
        ret = cb(*args)
        print("Return von innen", ret)
    return innen # Referenz auf die gerade definierte Funktion zurück

def tst(*args):
    l = []
    for i in args:
        print("Arg:", i)
        l.append(i**2)
    return l

tst(1,2,3)
r = rahmen(tst)
r(2,3,4)
# Ausgabe:
# Arg: 1
# Arg: 2
# Arg: 3
# innen, Variable von rahmen: gemerkte_Variable , Args: (2, 3, 4)
# Arg: 2
# Arg: 3
# Arg: 4
# Return von innen [4, 9, 16]
```

Der Aufruf von „rahmen“ mit dem Argument „tst“ liefert eine Funktion, die ich hier in „r“ speichere, zurück. Wenn ich diese Funktion wiederum aufrufe, „erinnert“ sie sich an die Variable in „rahmen“ und sie ruft die ebenfalls gemerkte Funktion „tst“ auf.

Jetzt mal ernsthaft: Was soll das? Ist das nur für studierte Informatiker? Meiner Meinung nach nein! Denn wir haben gerade unseren ersten Decorator entwickelt:

```
@rahmen
def tst(*args):
    ...
```

diese Schreibweise befreit uns von der Notwendigkeit, die schwer lesbaren Varianten:

```
r = rahmen(tst)
r(2,3,4)
# kann man zusammenfassen zu:
rahmen(tst)(2, 3, 4)
```

benutzen zu müssen.

So sinnfrei war unser Beispiel gar nicht. Man kann es zum debuggen einer Funktion verwenden. Ohne die Funktion zu ändern, gebe ich nun die Argumente und das Ergebnis bei jedem Aufruf aus!

Eine weitere Anwendungsmöglichkeit für Decoratoren, die gerne auch in der Literatur angeführt wird, besteht darin, die Argumente vor Aufruf der Funktion einem Test auf Einhaltung von Bedingungen (nur Ganzzahl, nicht negativ, Typ String usw.) zu unterwerfen.

Mir gefällt besonders noch das Caching[84] mittels Decorator. Hier ein simples Beispiel dafür:

```
# decorator_cache.py
def cache(cb): #  Ein Zwischenspeicher
    _cache = {}
    def _in(*args):
        if args in _cache: # Ergebnis liegt schon vor!
            print("Habe ich schon berechnet!")
            return  _cache[args]
        ret = cb(*args)
        _cache[args] = ret
        return ret
    return _in

def timer(cb): # Ein Zeitmesser
    def _time(*args):
        start = ticks_us()
        cb()
        print("Dauer:", ticks_diff(ticks_us(), start, "µs")
    return _time

@timer
@cache
def calc(*args):
    r = 3
    for i in args:
        r **= i # Aufwendige Berechnung
    return r

x = calc(6, 2, 4)
print(">",x)
x = calc(3, 7, 2)
print(">",x)
x = calc(6, 2, 4)
print(">",x)
```

84 Mit Caching wird das Zwischenspeichern von Ergebnissen bezeichnet.

```
# Ausgabe:
# Dauer: 541 µs
# > 79766443076872509863361
# Dauer: 629 µs
# > 109418989131512359209
# Habe ich schon berechnet!
# Dauer: 315 µs
# > 79766443076872509863361
```

Der Decorator „timer“, den ich hier zusätzlich verwende, misst die Dauer des Aufrufs von „calc“. Schön zu sehen ist hier, dass ich Decoratoren auch staffeln kann. Im Beispiel wird u.a. noch ausgenutzt, dass auch eine Liste als Schlüssel eines Dictionary verwendet werden kann. Der dritte Aufruf von „calc“ wird als schon ausgeführt erkannt und das Ergebnis aus dem Speicher genommen (was oft um Größenordnungen schneller ist).

III OOP – Objekt Orientierte Programmierung ganz pragmatisch

Eine Einführung in Python bzw. Micropython und dann gleich ein Kapitel über OOP, ist das nicht viel zu schwer? Nein, denn OOP stellt gerade bei der Mikrocontroller Programmierung den Zusammenhang zwischen Hardware und Software besonders schön dar. Ein Hardware-Baustein, z.B. die PWM, ist ja auch ein Objekt, dass ich durch die Klasse PWM ansteuern kann. Zunächst ein kleines Beispiel:

```
pin13 = machine.Pin(13, machine.Pin.OUT)
```

Das Modul[85] „machine“ brauchen wir nicht zu importieren, da es, zusammen mit „rp2“, dem Modul für die spezielle Unterstützung des Pi Pico, automatisch geladen wird. Das Modul enthält die Klasse „Pin“, von der wir eine Instanz erstellen und in der Variablen pin13 speichern. Eine Klasse stellt die Schablone für die daraus erstellbaren Objekte dar. Man kann sich das wie einen Plätzchenausstecher vorstellen, mit dem man beliebig viele Objekte in der Form des Stanzers herstellen kann. In der OOP Terminologie heißen die Objekte auch Instanzen.

1 Eigenschaften von OOP

Das Besondere an diesen Instanzen ist, dass sie die Daten und die darauf angewendeten Funktionen, die in der OOP Methoden heißen, zu einer Einheit zusammenfassen. Jede Instanz enthält ihre eigenen Daten, die sie nicht mit anderen Instanzen der gleichen Klasse teilt. Daher kann ich mit der Zeile:

```
pin14 = machine.Pin(14, machine.Pin.IN)
```

eine andere Instanz von Pin erzeugen. Während pin13 den Anschluss 13 des Pi Pico als Ausgang ansteuert, kann pin14 von Anschluss 14 den Zustand lesen.

Beide Instanzen haben den gleichen Vorrat an Methoden zur Verfügung. Für einen Ausgangs-Pin dienen die Methoden „on“ und „off“ zum Schalten der Ausgangsspannung:

```
pin13.on()
```

würde eine mit diesem Anschluss verbundene LED einschalten.

```
pin14.value()
```

gibt den digitalen Wert, also 0 oder 1, zurück, je nachdem, ob wir eine Spannung an Anschluss 14 legen oder nicht.[86]

Die Eingabe von „pin13“ ergibt übrigens:

```
Pin(GPIO13, mode=OUT)
```

Die Klasse Pin ist hier so nett, und gibt einige Informationen über diese Instanz aus. Wir werden bald sehen, wie man das in Python macht.

Die Klasse Pin schreibe ich nun beispielhaft nach.

85 Siehe dazu Kapitel „Module und Pakete“. Ein Import schadet aber auch nicht.
86 Siehe dazu Kapitel „Mikrocontroller Hardware“.

```
class Pin:
    def __init__(self, pin: int, mode:int = 1):
        self.pin = pin
        self.mode = mode
        self._pin_status = 0

    def on(self):
        # Code, um den Pin einzuschalten
        self._pin_status = 1
        print("Pin“, self.pin, „an")

    def off(self):
        # Code, um den Pin auszuschalten
        self._pin_status = 0
        print("Pin“, self.pin, „aus")

    def value(self) -> int:
        # Code, um den Pin abzufragen
        print(f"Pin“, self.pin, „steht auf“, self._pin_status)
        return self._pin_status

    def __repr__(self):
        return f"Ich bin eine Pin Instanz für Anschluss {self.pin}"
```

Klassen werden in Python mit dem Schlüsselwort „class“ eingeleitet. Durch die Einrückung des folgenden Textes mache ich die Zugehörigkeit zu dieser Klassendefinition deutlich. Als Namen für die Klasse habe ich „Pin“ gewählt.

Die spezielle Methode „__init__“ erzeugt[87] eine Instanz der Klasse und legt fest, welche Parameter ich übergeben kann und muss. Das folgt der bei Funktionen schon gezeigten Syntax. Sie wird nicht direkt aufgerufen, sondern implizit, wenn ich den Klassennamen gefolgt von den Argumenten aufrufe.

Auffällig ist der Bezeichner „self“. Jede Variable, die in der Instanz überall gelten soll, wird mit „self“ eingeleitet; alle Methoden haben als ersten Parameter „self“.

Python macht dadurch klar, dass es sich um Instanz-Variablen und -Methoden handelt.

Wir haben die Methoden on, off, value und eine mit dem Namen „__repr__“. Wenn ich im REPL z.B. eingebe:

```
pin = Pin(21) # nur ein Argument, da das zweite optional ist
pin
```

erhalte ich die Ausgabe

```
Ich bin eine Pin Instanz für Anschluss 21
```

Weitere Besonderheiten sind der Unterstrich am Anfang der Variablen self._pin_status. Er stellt eine Konvention in Python dar, die besagt, dass diese Variable nur intern in der Klasse genutzt werden soll. Dasselbe kann man auch für Methoden anwenden. Für den Python Interpreter hat er keine Auswirkung, sondern dient nur als Hinweis für den Leser des Programmtextes.

Dieses „privat machen“ von Variablen und Methoden dient der Wartbarkeit des Programms. Ein Programm ist NIE fertig. Also muss ich, zu einem späteren Zeitpunkt, meinen Text ändern; dazu

87 Eigentlich erzeugt sie nicht wirklich die Instanz, aber da sie intern von Python als erstes aufgerufen wird, sieht es so aus. Ich kann eine Klasse formal auch ohne „__init__“ Methode schreiben.

muss ich aber verstehen, wie das Programm funktioniert. Der Unterstrich am Anfang eines Bezeichners sagt mir nun, dass er nicht von „außen“ verwendet wird. Ich kann den Bezeichner daher gefahrlos umbenennen, einen anderen Typ dafür benutzten usw.

2 OOP – Vererbung

Wenn die Kapselung von Daten und Methoden das einzige wäre, was OOP leisten kann, würde man sie nicht wirklich brauchen. Wie schon gesagt, verfügen andere Programmiersprachen über statische Variablen, um es einer Funktion zu ermöglichen, sich etwas zu „merken“.

Wenn man sich Sprachen mit festen Typen anschaut, gibt es ein Problem beim Aufruf, wenn das Argument nicht hundertprozentig dem des Parameters entspricht. So kann ich in C keine Funktion schreiben, die wahlweise einen int oder einen float annimmt. Mit der objektorientierten Erweiterung C++ geht das, indem man eine Methode gleichen Namens mit unterschiedlichen Parametern[88] definiert.

In Python braucht man dafür keine OOP, da ein Parameter ja prinzipiell jedes Argument mit jedem Typ entgegennehmen kann.

Doch es gibt da noch die „Vererbung“. Wir gleiten hier nicht zu tief in die Biologie ab, denn auch auf einer abstrakteren Ebene kann ich nämlich von einer Klasse erben. Hier ein "klassisches" Beispiel, wie man es gerne in OOP Lehrbüchern findet:

```
# vererbung1_tst.py
class Tier:
    def __init__(self, gewicht, alter):
        self.gewicht = gewicht
        self.alter = alter

    def alter_aendern(self, alter=None):
        if alter == None:
            return self.alter
        else:
            self.alter = alter

    def beschreibe(self):  # Beschreibung wird ausgegeben
        print(f"Tier mit Alter {self.alter} und dem Gewicht {self.gewicht} kg")

class Hund(Tier):  # "Hund" erbt von "Tier"
    def __init__(self, name, rasse, gewicht, alter):
        self.name = name
        self.rasse = rasse
        super().__init__(gewicht, alter) # hier wird die Elternklasse initiert

    def rasse_aendern(self, rasse):
        if rasse == None:
            return self.rasse
        else:
            self.rasse = rasse

    def beschreibe(self):  # Name wird ausgegeben
        super().beschreibe()
        print(f"Hund {self.name} der Rasse {self.rasse}, dem Alter {self.alter}
Jahre und dem Gewicht {self.gewicht} kg")
```

88 Der Fachausdruck dafür ist Polymorphie.

```
tier = Tier(10, 2)
hund = Hund("Fido", "Dackel", 7, 3)
tier.beschreibe()
hund.beschreibe()
hund.alter_aendern(5) # ändert das Hundealter
hund.beschreibe()

# Ergebnis:
#Tier mit dem Alter 2 Jahre und dem Gewicht 10 kg
#Tier mit dem Alter 3 Jahre und dem Gewicht 7 kg
#Hund Fido der Rasse Dackel, dem Alter 3 Jahre und dem Gewicht 7 kg
#Tier mit dem Alter 5 Jahre und dem Gewicht 7 kg
#Hund Fido der Rasse Dackel, dem Alter 5 Jahre und dem Gewicht 7 kg
```

Obwohl man hier alle formalen Merkmale der Vererbung aufzeigen kann, worauf ich gleich noch näher eingehe, ist das Beispiel ziemlich praxisfern. Mich hat so etwas früher eher abgeschreckt.

Doch bleiben wir zunächst bei diesem Beispiel und sehen es uns der Reihe nach an. Die Klasse "Tier" enthält mit „gewicht" und „alter" zwei Instanzvariablen und die Methoden „beschreibe" und „alter_abfragen_oder_aendern“. Soweit nichts Neues. Nun definieren wir die Klasse „Hund“, die durch „Tier“ als Parameter zum Klassennamen von dieser Klasse erbt. In der Methode „__init__“ werden wieder die Parameter festgelegt. Die Parameter „name“ und „rasse“ werden Instanzvariablen zugewiesen. Die weiteren, „gewicht“ und „alter“, wurden schon in „Tier“ definiert. Die Zeile beginnend mit „super“ sorgt nun dafür, dass die Klasse „Tier“ mit diesen Parametern versorgt wird. Hier findet sozusagen die Instantiierung der übergeordneten („super“) Klasse statt. Dann wird eine Methode „rasse_abfragen_oder_aendern“ definiert.

Die nächste Methode, „beschreibe“ hat den gleichen Namen wie in ihrer Eltern-Klasse „Tier“. Damit überschreibt sie sie; aber wiederum durch „super“ kann ich auf die Eltern-Methode zugreifen. Hier benütze ich das, um beide Ausgaben zu erhalten.

Beim benützen der Klassen wird deutlich, dass die Kind-Klasse auf alle Methoden der Eltern-Klasse und ihre eigenen Methoden zugreifen kann. Auch die Instanzvariablen stehen zur Verfügung.

In Python kann man auch von mehreren Klassen gleichzeitig erben. Eine Klasse „Pflanze“ könnte zusammen mit „Tier“ als Elternklasse für alle halb-halb Lebewesen wie Seeanemonen, Seesterne usw. dienen:

```
class Seestern(Tier, Pflanze):
```

In der Informatik wird das Thema Mehrfachvererbung durchaus kontrovers diskutiert. So wird z.B. argumentiert, dass Mehrfachvererbung bestimmten OOP Prinzipien widersprechen würde. Das Hauptproblem besteht aber darin, dass Uneindeutigkeiten entstehen könnten. Wenn ich z.B. von zwei Elternklassen erbe, die beide eine Methode xyz definieren, welche wird dann aufgerufen? In Python hat man sich für eine klare links nach rechts Auswertung entschieden. D.h., dass die erste in der Parameterleiste aufgeführte Klasse gewinnt. Ihre Methode „xyz“ würde aufgerufen.

3 Klassenvariablen und -Methoden

Ich habe zu Anfang des Kapitels eine „Klasse“ als eine Schablone für die Objekte, die man daraus erzeugen kann, bezeichnet. Wie so oft war das noch nicht die ganze Wahrheit. Eine Klasse an sich kann nämlich auch Werte speichern und Klassenmethoden, wie sie dann genannt werden, enthalten.

Wozu soll das gut sein? Stellen wir uns vor, wir wollten zählen, wie viele Tiere oder Hunde wir erzeugen. In einer Instanzvariablen geht das offensichtlich nicht; sie wird ja pro Instanz neu erzeugt. Außerhalb der Klasse, z.B. in einer globalen Variable, wollen wir aber auch ganz sicher so etwas nicht speichern (siehe Kapitel „Von Räumen und Sichtbarkeit“).

Wenn wir nach der „class Tier“ Zeile

```
    anzahl_tiere = 0
```

schreiben und in __init__ die Zeile

```
    Tier.anzahl_tiere += 1
```

einfügen, erhalten wir genau so einen gewünschten Zähler. Da „anzahl_tiere = 0“ nicht in einer Methode definiert wird, wird es von Python als Klassenvariable verstanden. Greifen wir innerhalb einer Instanzmethode darauf zu, muss der Klassenname vorangestellt werden.

Eine Klassenmethode erzeugt man, indem man eine Methode ohne das „self“ definiert[89]:

```
   def bellen():
        print("WAU!")

# Aufruf:
Hund.bellen()
WAU!
```

Eine Klassenmethode verhält sich wie eine Funktion, aber sie gehört zur Klasse.

4 Vererbung anwenden

Klassen haben wir jetzt schon häufig benutzt, um z.B. ein „PWM“ Objekt zu erzeugen. Wenn wir nun eine „Led“ Klasse erstellen wollen, haben wir zwei Möglichkeiten: Wir können die Klasse „PWM“ darin verwenden (engl. composition) oder wir können von „PWM“ erben. Schauen wir uns beide Varianten einmal an.

```
# vererbung2_tst.py
class LedErbt(machine.PWM): # erbt von PWM
    def __init__(self, id: int):
        super().__init__(id)
        self.freq(200) # Frequenz 200 Hz
        self.duty_u16[90](0) # LED aus

    def duty(self, percent: int):
        # u16 meint eine 16 Bit Zahl ohne Vorzeichen, also 0 - 65535
        self.duty_u16(int(65536/100*percent)) # Helligkeit in %,
        # das int() ist nötig, weil duty_u16 keine Fließkommazahl erlaubt
```

89 Es gibt einen Decorator „@classmethod“ (siehe Kapitel „Das muss hübscher werden – Decoratoren), der es erlaubt, eine Klassenmethode auch vom Instanznamen aus aufzurufen, also z.B. hund.bellen(). Dieser Decorator verlangt als ersten Parameter ein per Konvention „cls“ genanntes Element.

90 Im englischen heißt Tastverhältnis „duty cycle“, oft nur als „duty“ abgekürzt.

```
class LedNutzt: # verwendet PWM
    def __init__(self, id: int):
        self.led = machine.PWM(id)
        self.led.freq(200) # Frequenz 200 Hz
        self.led.duty_u16(0) # LED aus

    def duty(self, percent: int):
        self.led.duty_u16(int(65536/100*percent)) # Helligkeit in %

ledgeerbt = LedErbt(13)
ledgenutzt = LedNutzt(14)
ledgeerbt.duty(10)
ledgenutzt.duty(20)
```

Auf den ersten Blick sieht das sehr ähnlich aus. Jetzt wollen wir die PWM Frequenz[91] ändern. Für „ledgeerbt“ geht das so:

```
ledgeerbt.freq(8)
```

Für „ledgenutzt“ muss das so gemacht werden:

```
ledgenutzt.led.freq(8)
```

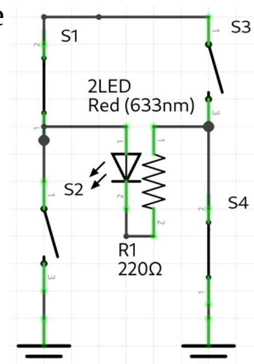
Ich benötige also zusätzliches Wissen darüber, dass es in LedNutzt eine Variable „led“ gibt, die auf eine Instanz von PWM verweist :-(

Wenn ich das vermeiden möchte, kann ich die Methode „freq“ mit einer Methodendefinition

```
def freq(self, freq: int):
    self.led.freq(freq)
```

nach Außen direkt als „ledgenutzt.freq(300)“ zur Verfügung stellen.

Ein weiteres Beispiel zeigt nun eine Klasse, die eine Zweifarben-LED[92] ansteuert. Zweifarben-LEDs zeigen z.B. Rot oder Grün, je nach Polarität der Spannung. Um die Polarität umkehren zu können, brauche ich eine sogenannte H-Brücke (siehe Bild rechts). Sind, wie im Bild, die Schalter S_1 und S_4 geschlossen, fließt der Strom von links Oben nach rechts Unten. Öffne ich die beiden Schalter und schließe S_3 und S_2, so fließt der Strom von rechts Oben nach links Unten und die Stromrichtung durch die LED kehrt sich um.



Zufälligerweise ;-)) entsprechen die Ausgänge des Mikrocontrollers genau solchen Schaltern: Jeder GPIO bildet eine Hälfte der H-Brücke nach; zwei dieser Halbbrücken bilden dann die gesamte H-Brücke.[93]

```
# dualled_tst.py
import time

class DualLedErbt(machine.PWM):
    def __init__(self, id1: int, id2: int):
        super().__init__(id1)  # Halbbrücke mit PWM, geerbt
        self.hb = machine.Pin(id2, machine.Pin.OUT)  # Halbbrücke ohne PWM
        self.freq(200)  # Frequenz 200 Hz
```

91 Alle PWM-Kanäle haben die gleiche Frequenz. Es widerspricht den OOP Prinzipien, wenn man für eine Instanz die Frequenz ändert und alle anderen Instanzen ihre Frequenz ebenfalls ändern. Das ist zwar hier der Hardware geschuldet, aber dennoch nicht „pythonisch“. Das Tastverhältnis ist aber für jede Instanz unterschiedlich ;-))

92 Es gibt verschiedene Bauformen mit 2 oder 3 Beinchen. Hier wird eine zweibeinige verwendet.

93 Natürlich sind im Mikrocontroller keine Schalter, sondern es werden elektronische Schaltelemente verwendet.

```
        self.duty_u16(0)
        self.hb.off()  # beide Ausgänge auf 0 = LED aus

    def duty(self, percent: int, color: bool = True):
        if color:
            self.duty_u16(int(65536 / 100 * percent)) # Helligkeit in %
            self.hb.off()
        else:
            # Wenn die H-Brücke andersherum arbeitet, muss der "percent" Wert
            # invertiert werden
            self.duty_u16(int(65536 / 100 * (100 - percent))) # Helligkeit in %
            self.hb.on()

dled = DualLedErbt(13, 14) #LED an Pin 13 und 14. Widerstand nicht vergessen!!!
while True:
    for c in range(2):
        for n in range(0, 101, 5):
            #print(c, n)
            dled.duty(n, c)
            time.sleep(0.1)
```

Das Programm wiederholt nun unendlich ein Dimmen der LED von 0 – 100% mit wechselnder Farbe. Die Methode „range“ liefert eine Liste von Zahlen zurück, hier in n von 0 – 100 in 5er Schritten, in c von 0 – 1.

Was fällt noch auf? Unsere Klasse erbt zwar von PWM, *nutzt* aber auch noch einen GPIO (self.hb). Für das Dimmen einer LED benötigt man keine zwei PWM Ausgänge – im Gegenteil, das wäre kontraproduktiv, da sich die Phase der Pulse überlagern könnte und zu unvorhersehbaren Ergebnissen führen könnte.

Da das Tastverhältnis in unserer Kalkulation einen positiven Puls erzeugt, muss er nach der Umkehr der H-Brücke in einen Negativen umgewandelt werden (100 – percent), sonst würde die LED in der dann gewählten Farbe mit kleinen Prozentwerten am hellsten leuchten.

Egal, ob Vererbung oder Nutzung: Die Erstellung einer Klasse für unsere Zweifarben-LED macht den Code übersichtlich und gut lesbar. Eine Umsetzung mit Funktionen wäre dagegen auf globale Variablen angewiesen und damit deutlich fehleranfälliger.

5 Eine andere Anwendung von Vererbung

Wenn ich eine, insbesondere nicht selbst geschriebene Klasse verwende, deren interne Funktion ich nicht verstehe und auch nicht verstehen muss, kann es aber doch vorkommen, dass ich Kleinigkeiten verändern möchte. Ich möchte z.B. ein Argument vorher auf korrekte Daten prüfen oder ein Ergebnis umformen, was ich notfalls auch außerhalb der Klasse tun kann, aber dann bei jedem Aufruf in meinem Programm mitberücksichtigen muss.

Das Eingreifen und Umschreiben einer Klasse, die ich mir über „pip“ oder durch Download eines Quelltextes besorgt habe, stellt keine gute Idee dar. Erstens: Was geschieht, wenn es eine neue, fehlerbereinigte oder neue Fähigkeiten bereitstellende Version gibt? Zweitens: Wenn nach meinen Änderungen Fehler auftauchen, woher stammen sie?

Mit Erben ist man da auf der sicheren Seite. Neue Version: Im Idealfall, wenn der Programmierer bestimmte Konventionen eingehalten hat, kann meine erbende Klasse direkt damit klar kommen. Bei Fehlern kann ich die Eltern-Klasse und meine erbende Klasse unabhängig testen.

Auch zum Debuggen kann ich erben: Ich schreibe eine erbende Klasse, die die Aufrufargumente und die Rückgabewerte per „print“ ausgibt.

```
# vererbung2_tst.py
class DebugLed(LedErbt):
    def __init__(self, id):
        self.id = id
        super().__init__(id)

    def duty(self, percent):
        if percent < 0: # Argument „percent“ negativ?
            percent = 0
        if percent > 100: # Argument „percent“ über 100?
            percent = 100 # Beispiel für Argumentenvalidierung[94]
        self.percent = percent
        super().duty(percent)
        print(f"Pin {self.id} mit Frequenz {self.freq()} und Duty 
{self.percent}")

p = DebugLed(13)
p.duty(120)
# ergibt
Pin mit Frequenz 200 und Duty 100
```

6 Eine besondere Art des Methodenaufrufs

Auf eine oft genutzte Möglichkeit, wie ich eine Methode erstellen kann, die sowohl etwas schreibt (verändert) als auch etwas ließt (keine Veränderung), möchte ich noch eingehen. Weiter oben, bei der Klasse "Tier", habe ich sie schon einaml stillschweigend verwendet.

Nichts hindert mich zu schreiben:

```
class Motor:
    ....

def drehen(self,lesen: bool, drehzahl: int):
    if lesen:
        return self.drehzahl # gibt die aktuelle Drehzahl zurück
    else:
        self.drehzahl = drehzahl # ändert die aktuelle Drehzahl

m = Motor()
m.drehen(True, 0) # gibt die aktuelle Drehzahl zurück, ich muss 
                  # drehzahl einen Wert mitgeben, auch wenn er 
                  # nicht verwendet wird
m.drehen(False, 1000) # ändert die aktuelle Drehzahl
```

Elegant ist das nicht gerade. Als wir über Funktionen gesprochen haben, war schon die Rede von Default-Werten. Wie kann ich das für meine Aufgabenstellung nutzen? Ein erster Ansatz könnte so aussehen:

```
def drehen(self, drehzahl=0):
    if drehzahl:
```

94 Von Validierung spricht man, wenn ein Wert auf bestimmte, einzuhaltende Bedingungen überprüft wird.

```
        self.drehzahl = drehzahl # ändert die aktuelle Drehzahl
    else:
        return self.drehzahl # gibt die aktuelle Drehzahl zurück
```

```
m.drehen()  # gibt die aktuelle Drehzahl zurück
m.drehen(1000) # ändert die aktuelle Drehzahl
```

Was passiert hier? Weil "drehzahl" nun einen Default-Wert hat, *muss* ich beim Aufruf keinen mitgeben. Im Kapitel "Variablen" haben wir schon erfahren, dass Variablen-Werte auch in einem boolschen Kontext ausgewertet werden können. "if drehzahl:" ergibt "True", wenn „drehzahl“ nicht Null ist.

Leider haben wir jetzt ein Problem, wenn wir die Drehzahl zu Null machen wollen! Die pythonische Lösung: Wir nehmen als Default-Wert den Nicht-Wert „None“:

```
def drehen(self, drehzahl=None):
    if drehzahl != None:
    ...
```

"None" wird zwar im booleschen Kontext auch zu „False“ ausgewertet, aber um zu vermeiden, dass eine Übergabe von Null nicht falsch verstanden wird, müssen wir explizit auf Gleichheit (oder Ungleichheit) zu „None“ abfragen.

Dieses Entwurfsmuster ist in Python enorm verbreitet. Wir werden später eine Menge Beispiele aus Modulen wie „machine“, „network“ usw. dafür sehen.

Noch eine Bemerkung: Natürlich kann man das auch in Funktionen anwenden. Weil aber Funktionen sich nichts merken können, wird man es hier seltener finden.

7 Einfach magisch!

Python verfügt über „magische" Methoden. Formal sind sie daran zu erkennen, dass ihr Name vorn und hinten mit zwei Unterstrichen eingerahmt wird.

Sie können und dürfen[95] nicht direkt aufgerufen werden, sondern Python ruft sie in bestimmten Situationen selbst auf. Mit den Methoden: „__getitem__“, „__setitem__“ und „__delitem__“[96] kann ich meine eigene Dictionary-Klasse schreiben.

```
# vererbung_dict.py
class MyDict(dict): # erbt von dict Klasse

    def __init__(self, d: dict={}):
        super().__init__(d)

    def __setitem__(self, key, value): # Aufruf bei Zuweisung
        if isinstance(key, str): # isinstance prüft „key“ gegen „str“
            super().__setitem__(key, value)
        else:
            raise ValueError("Key muss vom Typ String sein")

    def __getitem__(self, key): # Aufruf bei Auslesen
        return f"Mein Wert für Schlüssel {key}: {super().__getitem__(key)}"
```

95 Im Quelltext sehen wir die Ausnahme: Wenn super() vorangestellt wird, können diese Methoden auch direkt aufgerufen werden.
96 Auf deutsch bedeutet das „erhalte Eintrag“, „setzte Eintrag“, „lösche Eintrag“.

```
    def __delitem__(self, key): # Aufruf bei Löschen
        return f" Ich mag Schlüssel {key} nicht löschen"

d = MyDict()
d["key1"] = "Wert_key1"
print(d["key1"])
# ergibt
Mein Wert für Schlüssel key1: Wert_key1
d[0] = 42
# ergibt
Traceback (most recent call last):
...
ValueError: Key muss vom Typ String sein
```

Hier wird nur beispielhaft eine Validierung des Schlüssels beim Setzten eines Wertes vorgenommen und beim Zugriff auf einen Schlüssel ein Text ausgegeben.

Es gibt zahlreiche weitere magische Funktionen. Sie heißen z.B. „__add__", „__mul__" oder „__sub__". So kann ich mir meine eigene Subtraktionsfunktion für Strings bauen. Zuerst muss ich mir überlegen, was wohl sinnvoll dabei herauskommen könnte. „Ottolein" – „lein" soll „Otto" ergeben. Was soll „Ottolein" – „xxx" ergeben? Erst, wenn ich das genau definiert habe, kann ich das als Programm codieren.

8 Ein Iterator im Eigenbau

Der formale Aufbau einer Iterator-Klasse fällt sehr einfach aus. In der Klasse müssen, neben „__init__", zwei Methoden definiert werden: „__iter__" und „__next__". Wir erkennen sie als „magische" Methoden. „__iter__" gibt in der Regel nur die Instanz zurück. Die wichtigere Methode „__next__" liefert jeweils den nächsten Wert: In einer for-Schleife implizit, durch Aufruf der Funktion „next" explizit. Unsere Beispielklasse liefert bei jeder Iteration eine Bitbreite und die maximale Länge einer damit darstellbaren Zahl.

```
# iterator_tst.py
class BitbreitenIterator:
    def __init__(self):
        self._index = 2

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < 2048:
            item = 2**self._index
            breite = self._index
            self._index *= 2
            return (breite, item)
        else:
            raise StopIteration

i = BitbreitenIterator()
for el in i:
    print(f"Bits {el[0]}: Anzahl Digits {len(str(el[1]))}")
# Ausgabe:
# Bits 2: Anzahl Digits 1
# Bits 4: Anzahl Digits 2
# Bits 8: Anzahl Digits 3
```

```
# Bits 16: Anzahl Digits 5
# Bits 32: Anzahl Digits 10
# Bits 64: Anzahl Digits 20
# Bits 128: Anzahl Digits 39
# Bits 256: Anzahl Digits 78
# Bits 512: Anzahl Digits 155
# Bits 1024: Anzahl Digits 309
```

Zugegeben, wieder ein etwas sinnfreies Beispiel. Es zeigt aber die grundlegende Syntax und die Tatsache, dass – obwohl iteriert wird – das Resultat jeweils berechnet wird.

Der Ausdruck „len(str(el[1]))“ ermittelt von innen nach außen den Index 1 des Tupels „el“ (in der Methode „__next__“ den Inhalt von „item“), wandelt diese Zahl in einen String um und gibt dessen Länge zurück.

9 Geheime Punkt-Sache

Mit Punkten haben wir uns schon bei den Modulen beschäftigt. Ein Modul wie „machine“ enthielt Klassen wie PWM und Pin, auf die man durch die Schreibweise <Modul-Name>**.**<Klassen-Name> zugreifen konnte. Es gibt noch eine andere Form der Verwendung von Punkten.

Wenn wir ein Objekt haben, haben wir auch eine Klasse, von dem es abstammt bzw. aus dem es instantiiert wurde. In Python ist nun eigentlich alles ein Objekt: 123 ist ein Objekt, „Hallo“ ist ein Objekt, [1, 2, 3] – eine Liste – ebenfalls. Also wäre es doch schön, wenn ich von einem Objekt aus auch auf die Methoden seiner Klasse zugreifen könnte. Und natürlich, man kann:

```
"Hallo".upper() # die Methode „upper"  aus der str Klasse wandelt
                # in Großbuchstaben
# Ausgabe:
# 'HALLO'
```

Die Methode „upper“ liefert einen String in Großbuchstaben; der ist auch ein Objekt. Was hindert uns zu schreiben:

```
"Hallo".upper().count("L") # "count" zählt, wie oft ein "L" vorkommt
# Ausgabe:
# 2
```

Zu lesen ist das von links nach rechts: Erst wird „Hallo“ in „HALLO“ gewandelt, dann werden alle großen „L“ gezählt, die im String vorkommen.

Es gibt keine Beschränkung, wie oft ich das wiederhole.

IV Typisch Mikrocontroller!

In den folgenden Kapiteln möchte ich auf Aspekte der Programmierung in Python eingehen, die typisch oder speziell für Mikrocontroller sind. Auf einem PC mit Python bin ich meist nicht in der Lage, bestimmte Hardwarebausteine, z.B. GPIOs, anzusprechen. Das sieht auf einem Raspberry Pi[97] Minicomputer, der mit Linux läuft, anders aus: Hier gibt es Bibliotheken wie RPi, die direkt in Python (dem „normalen" Python) den Zugriff auf die GPIO Pins erlauben. Mit Micropython auf Mikrocontrollern besteht diese Möglichkeit natürlich immer.

Um kurz auf den Raspberry Pi zurückzukommen: Mit ihm können zwei Welten verheiratet werden! Einerseits ein vollwertiges Betriebssystem und andererseits hardwarenahe Programmierung. Die im folgenden beschriebenen Sensoren und Aktoren können direkt auch an ihn angeschlossen[98] werden!

1 Das Tor zur Welt – Sensoren

Mikrocontroller und Sensoren gehören zusammen. Sensoren sind quasi die Augen und Ohren für unseren Mikrocontroller. Sie lassen ihn die reale Welt erfahren. Wir können Sensoren nach verschiedenen Kategorien unterscheiden, sei es z.B. danach, was sie erfassen oder danach, wie sie die Daten für den Mikrocontroller bereitstellen.

Da wir den Mikrocontroller ja mit einem Programm befähigen wollen, die Sensordaten auszulesen, steht für uns zunächst das „wie" der Datenübermittlung im Vordergrund. Zwei grundlegende Arten existieren: digital und analog. Im Folgenden werden wir erfahren, was es damit auf sich hat. Beginnen wir mit:

1.1 Digital

Es hat sich eingebürgert, die Worte „digital", „Digitalisierung" usw. ständig zu verwenden. Was dabei häufig untergeht, ist die exakte Definition dessen, was das ausmacht[99]. „Digital" kommt aus dem Lateinischen, dem Wort „digitus", und bedeutet Finger. Mit denen kann man zählen und daher stammt dann auch die Bedeutung. Die Digitalisierung erfasst etwas Analoges und setzt es in eine Zahl um. Ich rede jetzt z.B. nicht mehr von der Farbe Violett, sondern von 9400D3 im RGB Farb-Schema. Je zwei Ziffern stehen für Rot-Grün-Blau; der Rot-Anteil in Violett ist 94, der Grün-Anteil 0 und der Blau-Anteil D3, diese Zahlen in hexadezimaler[100] Schreibweise.

Obwohl wir immerhin 256 verschiedene Werte für die Grundfarben haben, leuchtet ein, dass das eine extreme Reduktion darstellt. Andere Farb-Schemata erweitern daher den Zahlenbereich auf 1024, 2048 usw. Was bleibt, ist die Reduktion, denn eine Farbe kann sich sozusagen aus unendlich vielen Werten zusammensetzen. Halten wir fest: Digitalisierung reduziert das Reale auf mit Zahlen darstellbare Werte.

---

97 Den Raspberry Pi (Himbeere) gibt es inzwischen in der fünften Generation. Die dritte bis fünfte sind aktuell neben einer Sonderversion, dem „Zero", erhältlich. Jede Version war mit einer großen Leistungssteigerung verbunden.

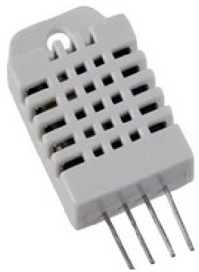
98 Auch hier die Warnung: Die GPIOs arbeiten mit 3,3 Volt und sind NICHT kurzschlussfest!

99 Wenn übrigens Politiker von Digitalisierung sprechen, meinen sie eher die Vernetzung der elektronischen Kommunikation.

100 Die Basis für unser dezimales Zahlensystem ist 10, die Basis für hexadezimal Zahlen 16 und für Binärzahlen 2. Hexadezimalzahlen werden mit den Ziffern 0-9 und A-F geschrieben.

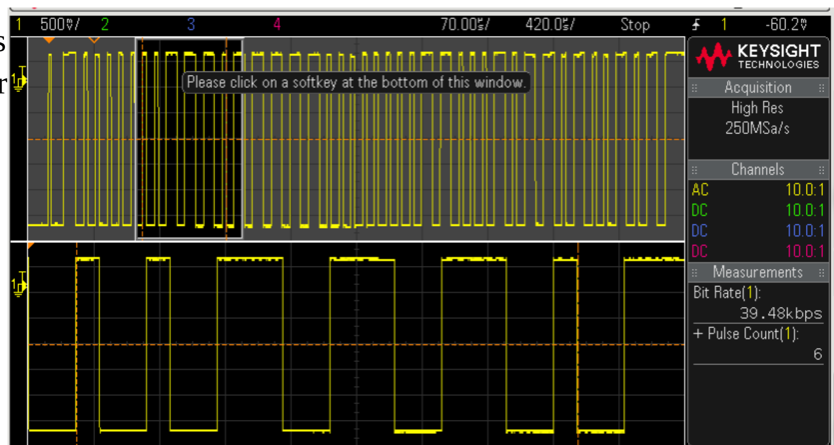
Ein riesiger Vorteil der digitalisierten Daten: Sie sind ohne weitere Verluste kopierbar. Die erste Kopie ist genauso gut wie die 100.000ste. Der Verlust entsteht nur beim ersten Schritt von Analog zu Digital.[101] Ein weiterer, gerne unterschätzter Vorteil: Auf Zahlen kann ich mathematische Operationen ausführen!

Die Extremform von „digital“ ist 0 – 1, an – aus, also genau zwei Zustände. Viele Sensoren liefern genau so ein Signal, z.B. Fensterkontakt geschlossen, Taster gedrückt. Müssen sie mehr Information liefern, codieren die Sensoren sie direkt in eine Zahl (im Binärformat) oder liefern ein Muster verschieden langer Pulse[102], die sich wieder in eine Zahl umwandeln lassen. Letzteres Verfahren erfreut sich großer Beliebtheit, weil man dann nur eine Leitung benötigt[103]. Vereinfacht wird bei ihr ein Puls einer bestimmten Länge als 1 und einer anderen Länge als 0 interpretiert. Das ähnelt morsen.



Ein Beispiel liefert der DHT22, ein weit verbreiterter Sensor für Temperatur und Luftfeuchtigkeit. Obwohl er 4 Beinchen hat, braucht er nur drei Anschlüsse zum Mikrocontroller hin. Das 3. Beinchen von links hat keine Funktion. Anschluss 1 erhält 3,3 V, Anschluss 4 ist die Masse und Anschluss 2 liefert das puls-codierte Signal[104]. Auf dem Oszilloskop-Bildschirm sieht es sehr kompliziert aus: Um es zu verstehen, muss man ja wissen, was eine 1 und was eine 0 ist, in welcher Reihenfolge sie gesendet werden, niederwertigstes oder höchstwertigstes Bit zuerst, wie die Daten geschachtelt sind, denn der Sensor liefert ja zwei Werte, wie überhaupt eine Datenauslesung initiiert wird …

So sieht das Signal aus:



Oben ist das gesamte Signal, unten ein Ausschnitt abgebildet. Dort sieht man sehr schön die unterschiedlichen Pulslängen.

Der Micropython-Programmierer hat es gut. Das Paket „dht“ findet sich in der Liste der schon im Image befindlichen Module. Ein

```
# dht22_tst.py
import dht
# Instantiierung für DHT Sensor an Pin 4, mit Pull-Up
```

101 Wenn ich aus meinem digitalen Wert wieder einen analogen mache, kommt als weitere Verfälschung die Linearität des Wandlers ins Spiel. Mit einem besseren Wandler kann ich aber später mit den gleichen Daten bessere Ergebnisse erzielen.

102 Als Puls wird ein Signal bezeichnet, dass z.B. von 0 auf 1 (Spannung aus – an) sehr schnell ansteigt, dort eine definierte Zeit verbleibt, um dann wieder auf 0 zurückzufallen.

103 Eigentlich sind es zwei: Man braucht ja eine Masse, einen Gegenpol zum Signal. Da alle Sensoren sich die Masse teilen können, fällt sie nur einmal an.

104 Laut Datenblatt braucht der DHT22 einen Widerstand von der Datenleitung nach Plus (3,3 V) in der Größe von ca. 5 kΩ. Ohne geht es, mal ja, mal nein, abhängig vom Pin. Schaltet man den internen Pull-Up-Widerstand an, lief es bei mir immer.

```
d = dht.DHT22(machine.Pin(4, machine.Pin.IN, machine.Pin.PULL_UP))
d.measure() # Messung auslösen
print(d.temperature()) # Temperatur in °C ausgeben
print(d.humidity()) # Luftfeuchte in % ausgeben
```

reicht aus.

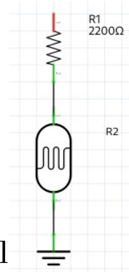
Um einen Taster (ein Knopf, der gedrückt einen Kontakt schließt) anzuschließen, benötige ich eigentlich nur die Klasse „Pin“ aus dem Modul „machine“. Dann konfiguriere ich mir einen Pin als Eingang (Pin.IN) mit einem Widerstand nach Plus ("PULL_UP"). Mit

```
pin = machine.Pin(5, machine.Pin.IN, machine.Pin.PULL_UP)
print(pin.value())
```

gebe ich den Zustand aus. Dass das Verarbeiten eines simplen Tasters doch nicht so einfach ist, zeige ich im Kapitel „Warten ist doof – von Interrupts und Multitasking“.

1.2 Analog

Ein Sensor, bestehend aus einem normalen Widerstand R_1 und einem lichtempfindlichen Widerstand R_2 in Serie geschaltet, liefert eine analoge Spannung.



Der Widerstand des LDR genannten Fotowiderstands kann nicht beliebig klein und auch nicht beliebig groß werden, daher schwankt die Spannung, je nach Lichtintensität, zwischen ca. 0.3 V und 3.0 V. Die Spannungsschwankungen können beliebig klein sein.

Was mache ich nun mit einer solchen Spannung? An einen normalen GPIO kann ich sie nicht anschließen, weil diese Pins nur 0 oder 1 erkennen. Unser Pi Pico hat aber mehrere sogenannte ADC Wandler. ADC steht für analog to digital conversion, analog nach digital Umwandlung. Wir digitalisieren also einen analogen Wert. Wie oben schon erwähnt, kann das nur mit einer begrenzten Genauigkeit erfolgen. Beim Pi Pico sind das immerhin 12 Bit, also der Zahlenbereich von 0 bis 4095. Jetzt schauen wir uns mal an, welche Auswirkung die 12 Bit Auflösung hat. Der Eingangsspannungsbereich des ADC geht von 0 bis 3,3 V. Negative oder höhere Spannungen gefährden den Mikrocontroller.

Teilen wir 3,3 V durch 4095 erhalten wir 0,000805861 V, also ungefähr 0,8 mV Auflösung. Gängige ADC in Mikrocontrollern haben oft nur 8 oder 10 Bit. 3,3 durch 1023 (10 Bit) ergibt ~3 mV. Ich muss also die Spannung um mehr als 3 mV erhöhen, um ein um eins höheres Ergebnis zu erhalten. Ohne zu sehr in Details eingehen zu wollen, kommen bei ADC noch Faktoren wie Linearität, Wandlungsfehler usw. hinzu. Und, nicht zu vergessen, die Wandlungsgeschwindigkeit, also die Zeit, die der ADC benötigt, um eine Spannung zu messen und die Zeit, bis er bereit ist, die nächste Messung durchzuführen[105].

Für ein Experiment mit einem LDR, um zu registrieren ob es hell oder dunkel ist, reicht der ADC im Pi Pico allemal.

```
# adc_tst.py
from machine import Pin, ADC
import time
```

105 Ein schönes Beispiel für ADC sind digitale Oszilloskope. Die erreichen heute schon im Preissegment um die 600€ 2 Milliarden Messungen pro Sekunde bei 16 Bit Auflösung. Der Pi Pico schafft immerhin eine Messung alle 2 µs, was 500000 Hz entspricht.

```
while True:
    adc = ADC(Pin(26)) # erzeuge eine ADC Instanz für Pin 26[106]
    print(adc.read_u16() / 65535 * 3.3, "V") # lese die Spannung am ADC
    time.sleep(1)
```

Die zurückgegebenen Werte muss ich mit den vorhin errechneten 3,3 V multiplizieren und, wegen der Spreizung auf 16 Bit durch die Methode read_u16[107], durch 65535 teilen, um auf eine Spannung in V zu kommen[108]. Die hilft im Falle eines LDR aber nicht, denn eigentlich wollen wir ja Helligkeit messen, z.B. in Lumen. Wie kompliziert das ist, kann man in einem Wikipedia Artikel erfahren: https://de.wikipedia.org/wiki/Helligkeit.

Das Einlesen ist also sehr einfach; die Umrechnung in eine aussagekräftige Einheit stellt dagegen vor teils große Herausforderungen.

Es gibt eine ganze Reihe weiterer analoger Sensoren: temperaturabhängige Widerstände, Gas-Sensoren oder auch ein Potentiometer.

1.2.1 Was ist ein Potentiometer?

Im Bild rechts befindet sich das Foto eines Potentiometers. Es stammt aus den 1950er Jahren. Es befindet sich nicht in einem Gehäuse und so kann man sehen, wie es funktioniert. Die Achse, an der man es einstellt, geht im Bild nach hinten und ist nicht sichtbar. Es gibt eine kreisförmige Widerstandsbahn, die aus einer Kohleschicht besteht. Ein Schleifer genannter Abgriff (hier steht er gerade oben) gleitet auf der kohlebeschichteten Bahn hin- und her. Diese Bahn bildet den Widerstand; die Anschlüsse links und rechts unten stellen quasi die Draht-Enden des Widerstands dar. Drehe ich den Schleifer ganz nach links, ist der Widerstandswert zwischen dem linken Anschluss und dem Schleiferanschluss in der Mitte klein, fast Null. In der gezeigten Mittelstellung hat der Wert genau die Hälfte des gesamten Widerstands und ganz rechts hat er den vollen Wert.

Benutzt man in einer Schaltung nur den Schleiferanschluss und einen der beiden Endanschlüsse, spricht man von einem regelbaren Widerstand. Werden alle drei Anschlüsse verwendet, spricht man von einem Potentiometer. In einem Verstärker dient ein Potentiometer z.B. der Lautstärkeregelung. Man spricht gerne auch von einem variablen Spannungsteiler, weil es damit möglich ist, die Ausgangsspannung von Null bis zum Maximum am Schleifer abzugreifen.

1.2.2 Eine Benutzersteuerung

Nehmen wir einmal an, wir möchten als Benutzer die Helligkeit einer LED regeln. Im OOP Kapitel wurde das schon angesprochen. Wie aber kann mein Mikrocontroller erfahren, welche Helligkeit ich gerade wünsche? Richtig: Hier hilft das Potentiometer.

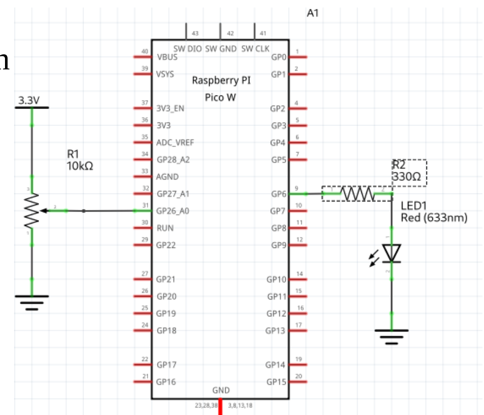
Im Bild weiter unten sieht man das (amerikanische) Schaltplansymbol eines Potentiometers. Hier steht die Widerstandsbahn senkrecht und der Schleifer in der Mitte.

106 Beim Pi Pico sind 3 ADC frei verfügbar. Einer ist intern mit der Versorgungsspannung verbunden. Die Anschlüsse liegen auf Pin 26, 27 und 28.

107 Leider finde ich keine näheren Informationen zur genauen Ausführung und den Gründen dieser Spreizung. Die 4096 unterschiedlichen Werte des ADC werden aber auf den Zahlenbereich 0 – 65535 abgebildet.

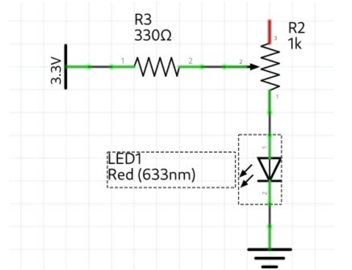
108 Meine Tests zu Hause zeigen, dass der ADC recht ordentliche Ergebnisse liefert.

Wie im Bild zu sehen, wird der Schleifer des Potentiometer an den ADC Eingang, hier A0 (GPIO 26), angeschlossen. Die beiden Enden liegen an 3,3 V und Masse (0 Volt). Bewege ich den Schleifer, ändert sich die Spannung an ihm von 0 bis 3,3 V. Der ADC digitalisiert diese Spannung. Das kleine Programm steuert die Helligkeit der LED dann durch drehen am Potentiometer (10 kΩ) von Aus bis maximale Helligkeit.




```
# led_adc.py
from machine import ADC, PWM
import time

led =  PWM(6, freq=100, duty_u16=0)
adc = ADC(Pin(26)) # erzeuge eine ADC Instanz für Pin 26
while True:
    v = adc.read_u16() # lese die Spannung am ADC
    print(v) # Kontrolle
    if v < 500: # um auszugleichen, dass das Poti nicht ganz 0 V liefert
        v = 0
    led.duty_u16(v) # setzt das An-Aus-Verhältnis des PWM Signals
    time.sleep(0.1) # 0.1 Sekunde warten
```

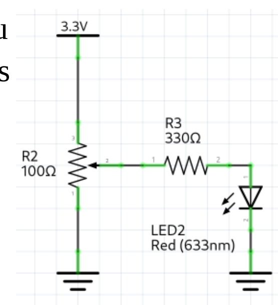
Je kleiner die „sleep“ Zeit, desto ruckelfreier gelingt die Helligkeits-Einstellung. Die gewählten 0,1 Sekunden stellten bei meinen Tests die obere Grenze dar. Außerdem ließ sich die LED ohne die Zeilen „if v < 500: v = 0“ nicht zuverlässig ausschalten, da der ADC auch beim Erreichen des Anschlags noch eine kleine Spannung maß. Da „sleep“ das gesamte Programm blockiert, sollte man in einem „echten“ Programm andere Verfahren wählen (siehe Kapitel „Warten ist doof – von Interrupts und Multitasking“).



Man könnte sich fragen, warum ich die LED-Helligkeit nicht direkt mit dem Potentiometer regele. Darauf gibt es gleich mehrere Antworten.

Machte man den Vorwiderstand regelbar (im Bild ist R2 so ein regelbarer Widerstand), würde man damit kämpfen, entweder - bei kleinem Widerstand - die LED nicht sehr dunkel zu bekommen oder - bei großem Widerstand - die LED nur auf den letzten paar Graden Drehung erkennbar zu beeinflussen. Das hängt mit dem Ohmschen Gesetz (U = R * I) und der Kennlinie der LED zusammen.[109]

Die Idee, die Lösung mit einer Spannungsteiler-Schaltung wie im Bild[110] zu machen, führt auch nicht zu einer befriedigenden Lösung. Wieder spielt uns das Ohmsche Gesetz und die sehr krumme Kennlinie der LED einen Streich; hinzu kommt, dass, um überhaupt eine sichtbare Regelung zu erhalten, der Widerstand des Potentiometers klein sein muss. Mit der Dimensionierung, wie in der Schaltung, fließt allein schon durch das



109 Da das hier zu weit führen würde, schlage ich vor, dass selbst auszuprobieren.

110 Man nennt das einen belasteten Spannungsteiler. Steht der Abgriff ganz oben, liegen die 3,3 V über den Schutzwiderstand R3 an der LED. Bewege ich den Abgriff nach unten, sinkt die Spannung an ihm und die LED wird dunkler. Wem das alles vorkommt wie böhmische Dörfer, könnte mein Buch „Eine kleine Einführung in die Elektronik“ (steht unter „bibliothek.velbert.de/angebote/elektronik-workshop“ zum Download bereit) lesen.

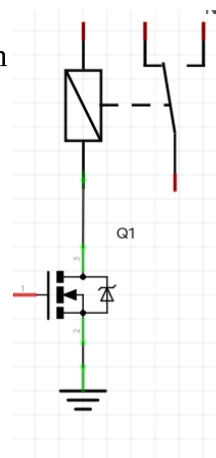
Potentiometer ein Strom von 33 mA (3,3 V / 100 Ω) - mehr als der Mikrocontroller (ohne WLAN) aufnimmt.

Eine PWM stellt dagegen die Art von Helligkeits- bzw. allgemeiner Leistungssteuerung dar, die energetisch am günstigsten ist. Während bei der Widerstandsregelung ein großer Teil der Leistung (im Vorwiderstand) in Wärme umgesetzt wird, hat die PWM durch ihre An-Aus-Arbeitsweise einen hohen Wirkungsgrad. Elektrische Leistung berechnet sich zu U * I. Wird einer der beiden Werte sehr klein, wird auch das Produkt klein. Ein (elektronischer) Schalter lässt keinen Strom fließen, dann ist die Spannung an ihm hoch, oder einen großen Strom fließen, dann ist die Spannung an ihm klein.

Der dritte Grund: Die PWM kann ich in meinem Programm auch noch durch andere Ereignisse steuern. So könnte ich zusätzlich die LED bei Bedarf hell aufleuchten oder blinken lassen, um die Aufmerksamkeit des Anwenders zu erregen.

Zum Abschluss dieses Kapitels noch eine Bemerkung zu anderen Mikrocontrollern als dem Pi Pico, die eventuell über keinen ADC verfügen: Es gibt diesen auch als externen Baustein, der dann, meist über SPI oder I2C[111], an ihn angeschlossen werden kann.

2 Aktoren



Aktoren sind das Gegenstück zu Sensoren. Wenn Sensoren die Augen und Ohren sind, bilden die Aktoren die Hände. Die einfachsten Aktoren sind die GPIO Pins im OUT-Modus. Sie liefern eine Spannung oder keine, können damit eine LED zum leuchten bringen oder ein Relais[112] schalten. Aber schon dies kann nicht direkt von den GPIOs angesteuert werden, weil es zu viel Strom verbraucht.

Im Bild rechts wird ein Relais von einem MOSFET[113] angesteuert. Ein MOSFET braucht je nach Typ nur die 3,3 V, um komplett durchzuschalten und sperrt vollständig bei 0 V. Das Rechteck mit der diagonalen Linie ist die Spule, die elektromechanisch einen Schalter betätigt. Die gestrichelte Linie deutet an, dass es eine galvanische Trennung zwischen dem linken und dem rechten Stromkreis gibt. Galvanische Trennung bedeutet, das es **keinen** Stromfluss zwischen dem Spulenkreis und den Schaltkontakten geben kann, auch nicht über Erde. Das bedeutet, dass ich mit den rechten Kontakten hohe Spannungen und Ströme, sogar Wechselspannungen, z.B. Netzspannung schalten könnte. Das Relais muss dazu natürlich geeignet sein. Die Parameter wie Spulenspannung und maximale Schaltleistung, meist für Gleich- und Wechselspannung getrennt[114], sind oft aufgedruckt. Sobald Netzspannung ins Spiel kommt, weil ich z.B. ein Gerät oder eine Lampe einschalten will, heißt es: VORSICHT. Sofern man nicht genau weiß, was man tut, lieber Finger weg!

111 Siehe Kapitel „Kommunikation ist alles“.
112 So nennt man einen elektromagnetischen Schalter.
113 Ein MOSFET ist ein Transistor nach dem Feldeffekt-Prinzip. Leider führt auch das hier zu weit. Er ist ein fast idealer Schalter.
114 Die viel niedrigere Gleichspannungs-Schaltleitung bei mechanischen Schaltern hängt mit der Funkenbildung beim an- und abschalten zusammen. Bei Wechselspannung erlischt der Funke in den Nulldurchgängen.

Üblicherweise schaltet man noch eine Diode in Sperrrichtung parallel zur Spule, um die beim Abschalten der Spule induzierte Spannungsspitze kurzzuschließen. Weil dieser MOSFET schon eine solche Diode eingebaut hat, kann ich hier darauf verzichten.

So, wie es spezielle Sensoren gibt, gibt es auch spezielle Aktoren für verschiedene Aufgaben. Einen (Schritt-)Motor antreiben (Drehbewegung), ein Magnetventil (Flüssigkeit) schalten, jedes Mal benötige ich besondere Hardware, um das umzusetzen.

In vielen Fällen reicht ein externer Transistor, um den geringen GPIO-Ausgangsstrom so zu verstärken, dass er für die Anwendung reicht.

2.1 Ein Beispiel für einen Aktor: der Servo-Motor

Von Modellbauern werden gerne sogenannte Servo-Motore benutzt, die eine Motorachse in einem Winkel von ca. 270° drehen können. Damit lassen sich Schiffsruder, Fahrmodelllenkungen usw. einfach realisieren.

So ein Servo-Motor besitzt seine eigene Elektronik, die den Motor ansteuert (und auf der gewählten Postion hält) und Positionierungsbefehle annimmt. Deshalb hat er drei Anschlüsse: Minus und Plus für die Stromversorgung und einen Steuereingang. Der Motor des Servos muss mit einer, meist separaten Spannungsquelle, die die benötigte Spannung und den Strom liefern kann, versorgt werden. Viele Servos lassen sich mit 3,3 V am Signaleingang ansteuern, also direkt mit unserem Mikrocontroller. Das muss im Einzelfall dem jeweiligen Datenblatt entnommen werden.

Die Ansteuerung des Servos erfolgt digital durch ein PWM Signal; die Frequenz beträgt 50 Hz (20 ms Periodendauer). Bei einem meiner Servos (MicroServo 9G) messe ich, dass bei ca. 0,7 ms die Motorachse am einen Ende steht, bei ca. 2,6 ms am anderen und bei ca. 1,5 ms in der Mitte. Eine unangenehme Eigenschaft meines Servos ist, dass er bei Pulsen, die kürzer als 0,7 ms aber größer als 0 sind, dauerhaft eine enorme Stromaufnahme von mehr als 700 mA entwickelt, ohne sich zu bewegen. Ein Fehler in der Elektronik? Normalerweise nimmt der Servo nur einen größeren Strom auf, wenn er seine Position verändert. Bei meinem Servo sind das in der Spitze mehr als 750 mA. Das kann meine USB Versorgung für den Mikrocontroller nicht liefern!

Bei fehlendem Steuersignal (0 Volt oder 3,3 Volt am Steuereingang) macht der Servo übrigens nichts, d.h. er bleibt an der Stelle stehen, die zuletzt angefahren wurde.

Mit Hilfe der PWM Klasse aus „machine“ können wir mit wenigen Zeilen eine Steuerung schreiben. Aber zunächst: Wie sieht das elektrisch aus?

2.1.1 Ein Exkurs in die Elektronik

Da dieses Thema einen Workshopteilnehmer sehr interessierte und es auch von allgemeinem Interesse ist, will ich hier darauf etwas ausführlicher eingehen.

Der Zusammenhang zwischen Spannung, Strom und Widerstand wird durch das sogenannte Ohmsche Gesetz definiert: U = R * I. Darin sind U die Spannung, R der Widerstand und I der Strom. Durch Gleichungsumformung wird daraus I = U / R und R = U / I.

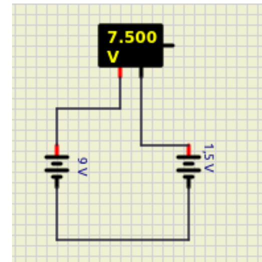
Ohne eine Spannung und einen Widerstand kann kein Strom fließen. Das demonstriert die Steckdose: Wenn ich keinen Stecker eines Gerätes hineinstecke, passiert im wahrsten Sinne nichts. Scherzhaft kann man sagen: Der Strom rieselt nicht aus der Steckdose.

In der Steckdose befinden sich zwei Anschlüsse. Den Schutzleiter lasse ich hier außen vor. Zwischen diesen Anschlüssen muss sich ein Widerstand befinden, damit Strom fließt. Dieser „Widerstand" kann eine Lampe, ein Heizofen, ein Kühlschrank sein.

Halte ich ein Spannungsmessgerät (das auch einen – wenn auch sehr hohen – Widerstand darstellt) an die beiden Anschlüsse, zeigt es mir die Wechsel-Spannung[115] von 230 V an.

Jetzt nehme ich zwei Batterien zur Hand: eine AA-Zelle mit 1,5 V und einen 9 V Block. Halte ich mein Spannungsmessgerät an die Pole der AA-Zelle, zeigt es die Spannung von ca. 1,5 V an; dasselbe passiert bei der Blockbatterie, hier werden ca. 9 V angezeigt. Wenn ich die Pole vertausche, ändert sich die Stromrichtung und das Messinstrument zeigt gegebenenfalls ein Minus vor dem Wert.

Was aber geschieht, wenn ich mein Spannungsmessgerät mit dem einen Anschluss an einen Pol der AA-Zelle und mit dem anderen Anschluss an einen Pol der 9 V Blockbatterie halte?



Es passiert – nichts! Das Spannungsmessgerät zeigt nichts an. Warum ist das so? Weil die beiden Batterien jede für sich eine Spannung erzeugen, aber ein Strom nur jeweils vom einen Pol der Batterie zum anderen fließen kann. Der Fachbegriff lautet: Die beiden Batterien sind galvanisch getrennt. Heben wir die Trennung auf, indem wir den Minuspol der AA-Zelle mit dem Minuspol der Blockbatterie verbinden, können wir sogar zwischen den Pluspolen eine Spannung (hier von 7,5 V, entspricht 9 V – 1,5 V) messen!

Der Motor des Servos wird mit einer Spannungsquelle betrieben, die die passende Spannung und ausreichend Strom für ihn bereitstellt. Um die Servo-Elektronik mit dem Mikrocontroller verbinden zu können, müssen wir – wie wir gerade gelernt haben – die GND Leitung von Mikrocontroller und Servo-Motor verbinden und den GPIO des Mikrocontrollers mit dem Steuereingang des Servos.

Wir können noch eine Überlegung anstellen: Wenn wir mehrere AA-Zellen in Serie schalten, erhalten wir eine Spannungsquelle mit <Anzahl der Batterien> * 1,5 V. Jeder kennt das aus batteriebetriebenen Geräten. Wir könnten auch mehrere AA-Zellen mit einer 9 V Blockbatterie in Serie schalten und erhielten dann eine um 9 V höhere Spannung. Das macht man aber normalerweise nicht, weil AA-Zellen und 9 V Blockbatterien unterschiedliche Innenwiderstände besitzen. Der Innenwiderstand, der durch den inneren Aufbau und die Chemie der Batterie bestimmt wird, ist aber für den maximalen Strom, den die Batterie liefern kann, zuständig. Man kann sich den Innenwiderstand wie einen in Serie geschalteten Widerstand zu einer idealen Spannungsquelle vorstellen.

115 Bei einer Wechsel-Spannung ändert sich die Spannung über die Zeit. Sie kann dabei auch negativ werden. Unsere Hausversorgung ist immer eine Wechsel-Spannung von 230 V.

Ist nun der Innenwiderstand einer Zelle deutlich größer als die der anderen, wirkt das wie ein Defekt bei einer größeren Last (die einem kleineren Widerstand entspricht).

Kann man Batterien (oder Spannungsquellen wie Netzteile) auch parallel schalten? Im Prinzip ja, aber wenn die Spannungen leicht unterschiedlich sind, was bei Batterien praktisch immer der Fall ist, fließen Ströme in Höhe der Differenzspannung dividiert durch den Innenwiderstand. Da Batterien chemisch funktionieren, entspricht das dann aber einer Entladung.

Alles Gesagte gilt natürlich auch für Akkus.

Bei z.B. zwei Netzteilen im Parallelbetrieb muss man berücksichtigen, dass sie meist nur Strom liefern, aber nicht aufnehmen können. Ansonsten kann ich aber aus zwei Netzteilen mit, sagen wir 12 V und 1 A, eine Spannungsquelle mit 12 V und **2** A bilden.

2.1.2 Eine Servo-Klasse

Warum schreiben wir eine Klasse und nicht ein oder zwei Funktionen? Auch hier gilt wieder, dass eine Klasse den Vorteil besitzt, dass sie sich Werte merken kann. Dazu kommt, dass wir nicht nur einen Servo, sondern durch mehrfaches Instantiieren auch viele Servos anschließen können.

```
# servo.py
from machine import PWM
from time import sleep

class Servo:
    """Eine ganz einfache Servo-Steuerungsklasse"""

    # Bei der Instantiierung muss nur der GPIO Pin angegeben werden,
    # die übrigen Parameter setzten die kürzeste und die längste Zeit
    # (z.B. 0,7 ms und 2,6 ms) sowie die Zeit für die (mechanische)
    # Mitte (z.B. 1,6 ms).
    # Mit der duty Methode wird in Prozent die gewünschte Position angegeben.
    # Hier wird 0 % als das eine Ende, 100 % als das andere Ende und 50%
    # als Mitte definiert.
    # Die Methode num_to_range bildet einen Zahlenbereich gleichmäßig auf
    # einen anderen ab.
    # Um die gewünschte Mitte zu erhalten (und nicht die arithmetische
    # Mitte zwischen 0,7 und 2,6), teilen wir programmtechnisch die Bereiche
    # auf: 0 - 50 wird auf 0,7 bis 1,6 und > 50 - 100 auf 1,6 bis 2,6
    # abgebildet.
    # Die PWM Klasse verfügt über die Methode duty_ns, die, wie der Name sagt,
    # den Duty in ns (Nanosekunden) erwartet. Weil unsere Links, Rechts, Mitte
    # Angaben in ms (Millisekunden) erfolgen, müssen wir diese Werte noch mit
    # 1_000_000 multiplizieren.

    def __init__(
        self,
        pin: int,
        mitte_ms: float = 1.6,
        links_min_ms: float = 0.7,
        rechts_max_ms: float = 2.6,
    ):
        self.pwm = PWM(pin, 50) # 50 Hz entsprechen 20 ms Periode
        self.links_min_ms = links_min_ms # in Prozent
        self.rechts_max_ms = rechts_max_ms # in Prozent
        self.mitte_ms = mitte_ms
        self.pos = -1
```

```
    def num_to_range(self, num: float, inMin: float, inMax: float, outMin:
        float, outMax: loat):
        return outMin + (float(num - inMin) / float(inMax - inMin) * (outMax –
outMin))

    def duty(self, val: float):
        """Duty in Prozent, 0 % enspricht dem einen Drehanschlag, 100 % dem
        anderen"""
        if val >= 0.0 and val <= 50:
            self.pwm.duty_ns(
                int(
                    self.num_to_range(val, 0.0, 50.0, self.links_min_ms,
                    self.mitte_ms)
                    * 1_000_000
                )
            )
        elif val > 50 and val <= 100:
            self.pwm.duty_ns(
                int(
                    self.num_to_range(val, 50.0, 100.0, self.mitte_ms,
                    self.rechts_max_ms)
                    * 1_000_000
                )
            )
        else:
           print("Ungültiger Wert", val)

    def position(self):
        return self.pos
```

Um dem Anwender der Klasse es so bequem wie möglich zu machen – was immer eine gute Idee ist, auch wenn man selbst der Anwender ist – werden alle Steueranweisungen an den Aktor in Prozent vorgenommen. Hier habe ich festgelegt, dass die Mittelposition 50 % entspricht. Denkbar wäre aber auch, die Mitte mit 0 % und die beiden Endpositionen mit +100 % und -100 % festzulegen. Solche Entscheidungen sind nicht trivial und beeinflussen später massiv die Anwendbarkeit der Klasse.

Für die Endpositionen einer angeschlossenen Mechanik kann man die entsprechenden Positionen vorgeben; bei einem Schiffsruder könnte das z.B. nur je 60 ° nach links und rechts bedeuten. In diesem Beispiel soll die Mitteleinstellung eine möglichst gerade Fahrtrichtung erreichen.

Die Nutzung der Klasse sieht dann wie folgt aus:

```
# servo_tst.py

servo1 = servo.Servo(11) # Servor an GPIO 11, Defaultwerte
servo2 = servo.Servo(12, 1.5, 0,9, 2.1) # GPIO 12, Mitte 1.5 usw.

pos1 = 0
pos2 = 100
while True:
    servo1.duty(pos1)
    servo2.duty(pos2)
    pos1 += 1
    pos2 -= 1
    if pos1 > 100:
        pos1 = 0
```

```
    if pos2 < 1.5:
        pos2 = 100
    sleep(0.1)
```

In der while-Schleife wird die Position der Motoren geändert. Dann wird „pos1“ vergrößert und „pos2“ verkleinert. Über die beiden if-Abfragen stelle ich sicher, dass bei Erreichen eines Endwertes die Positions-Variablen wieder auf einen passenden Wert gesetzt werden.

Mit angeschlossenen Servos dreht der eine in kleinen Schritten nach rechts, der andere nach links. Am Ende angekommen wechseln sie mit maximaler Geschwindigkeit wieder auf die entgegengesetzte Position.

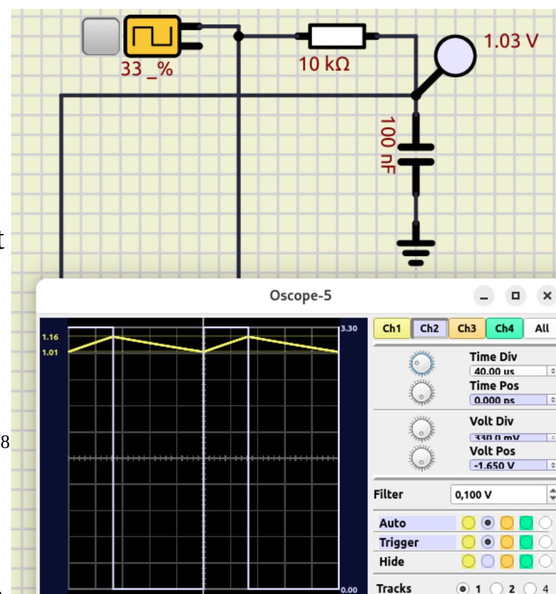
Das „sleep“ in der Schleife ist ganz wichtig. Fehlt es, werden die Duty-Werte so schnell geändert, dass die Mechanik des Servos nicht mehr hinterher kommt. Das Ergebnis ist ein wüstes hin und her der Drehachse.

2.2 Bis an die Grenze: ein PWM Experiment

Ein DAC, ein Wandler von digital zu analog, gehört auch zu den Aktoren. Meine Haus-Lüftungsanlage wird z.B. durch bestimmte Spannungen in verschiedene Betriebsarten umgeschaltet.

Der Pi Pico verfügt nicht über einen eingebauten, hardwarebasierten DAC[116]. Wenn ich einen Zahlenwert in eine Spannung verwandeln möchte und es nicht eilig habe, kann ich aber mit wenigen elektronischen Bauteilen und einem PWM Ausgang eine Digital-Analog-Konvertierung selbst erstellen.

Bei der Nutzung der PWM für eine LED-Helligkeitsregelung haben wir gesagt, dass die Trägheit des Auges die kurzen Lichtblitze in einen Helligkeitseindruck umwandelt. Für die Funktion als DAC müssen wir nun eine elektrische Trägheit erreichen, um aus der rechteckförmigen Spannung eine Gleichspannung zu erzeugen. In einem elektrischen Schaltkreis erledigt das die Serienschaltung eines Widerstandes mit einem Kondensator, ein sogenannter Tiefpass, da er vorzugsweise tiefe Frequenzen durchlässt und hohe unterdrückt.[117] Ein Tiefpass besitzt eine Grenzfrequenz genannte Schwelle; unterhalb dieser Frequenz bleibt das Eingangssignal unverändert, oberhalb wird es immer mehr abgeschwächt. Um zu zeigen, wie das wirkt, habe ich im Schaltungssimulator[118] die folgende Schaltung „aufgebaut“.



In der Realität käme das Rechteck-Signal vom Pi Pico. Hier wird im Funktionsgenerator mit einer Frequenz von 5 kHz und einem Tastverhältnis von 33 % der Pi Pico simuliert. Über den 10 kΩ Wiederstand und

116 In einem CD-Spieler wandelt z.B. ein DAC das digitalisierte Tonsignal in eine analoge Wechselspannung, die dann über den Verstärker aus den Lautsprechern wiedergegeben werden kann.

117 Ich darf hier nochmals auf mein Buch „Eine kleine Einführung in die Elektronik hinweisen“, wo die Bauteile ausführlich erläutert werden.

118 Ich verwende hier „SimulIDE“, eine Open Source Software für Windows und Linux. Die Datei heißt „DAC.sim1“.

den 100 nF Kondensator wird das Signal gefiltert[119]. Ich habe bewusst eine niedrige PWM Frequenz gewählt, damit man die Rippel genannte Welligkeit des hier gelb angezeigten Signals, das zwischen Widerstand und Kondensator entsteht, gut erkennen kann. Immer dann, wenn das Rechtecksignal hoch geht, wird der Kondensator geladen; die gelbe Kurve steigt relativ steil an. Geht das Signal wieder auf 0 V zurück, wird der Kondensator wieder entladen. Der runde Prüfpunkt in der Schaltung misst die Gleichspannung, hier 1,03 V. Rechnet man nach, sieht man, dass das Tastverhältnis von 33 % ziemlich genau einem Drittel der 3,3 V der Eingangsspannung entspricht. Bei 50 % erhalte ich ca. 1,65 V, bei 10 % ca. 0,33 V usw. Je höher die PWM Frequenz desto kleiner der Rippel und desto genauer stimmen die theoretischen Werte mit den gemessenen überein.

Kurz gesagt: Die analoge Spannung folgt dem Tastverhältnis direkt proportional. Wie sieht es nun mit der Änderungsgeschwindigkeit des Tastverhältnisses aus? Erst einmal stelle ich die grundsätzliche Überlegung an, dass die Änderungsgeschwindigkeit nicht höher sein kann als die PWM Frequenz[120]. Die zweite Überlegung: Aus den Messwerten des Oszilloskops sehe ich, dass innerhalb von etwas mehr als 60 µs Spannungspuls die Spannung am Kondensator um nur 60 mV steigt. Hochgerechnet brauche ich 3,3 V / 0,06 V = 55 mal diese Zeit, also 60 µs mal 55 = 3300 µs, um von 0 V auf 3,3 V zu kommen[121], wenn ich also das Tastverhältnis von 0 % auf 100 % erhöhe. Rechne ich 3300 µs in eine Frequenz um (1 / 3300 µs = 1 / 0,0033 s = ~330 Hz), erhalte ich den Grenzwert für die Änderungsgeschwindigkeit.

Ein anderer Weg, der aber m.E. nicht sehr leicht nachvollziehbar ist, führt über die physikalische Formel $f_g = 1 / (2 * PI * R * C)$, wobei f_g für Grenzfrequenz steht. Mit unseren Bauteilen:

1 / (2 * PI * 10000 Ω * 0,0000001 F) = 1 / (6,28 * 0,001) = ~159 Hz.[122]

Man sieht, die Größenordnung stimmt, wenn auch unsere Vergröberung fast zum doppelten Wert geführt hat.

Aber hält das auch einem Praxistest stand? Wir gehen ans Programmieren! Ziel ist eine Dreieckskurve, die gleich schnell ansteigt und abfällt.

```
# dreiecksgenerator.py
import time
from machine import PWM
pwm=PWM(15) #  PWM an Pin 15
pwm.freq(5000)
pwm.duty_u16(0) # Tastverhältnis als Wert zwischen 0 und 65535
n = 0 # Tastverhältnis
w = 100 # Wartezeit in µs
min=700 # untere Grenze für PWM
max=65535 # obere Grenze für PWM
schritte = 655 # 100 Schritte 
richtung = 1 # Auf oder ab
while True:
    n += schritte * richtung # neuer Wert für Tastverhältnis
    pwm.duty_u16(n) # setze Tastverhältnis
    # Richtungsentscheidungen
    if n > max:
```

119 Andere Bauteilwerte sind natürlich auch möglich. Ein größerer Widerstand ebenso wie ein größerer Kondensator erniedrigen die Grenzfrequenz.

120 Warum das so ist, wäre eine kleine Rechercheherausforderung im Internet.

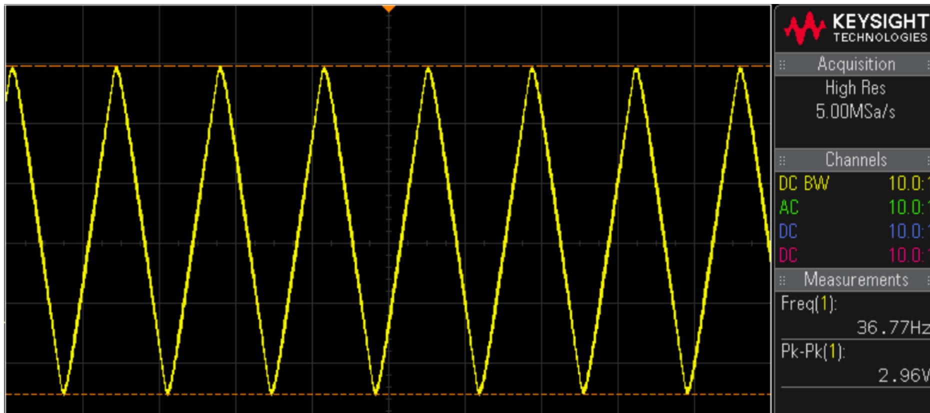
121 Das ist eine sehr grobe Vereinfachung, da die Kurve in Wirklichkeit einer exponentiellen Linie folgt.

122 Der Wert eines Kondensators wird in „F“ für Farad, nach dem Physiker Michael Faraday angegeben.

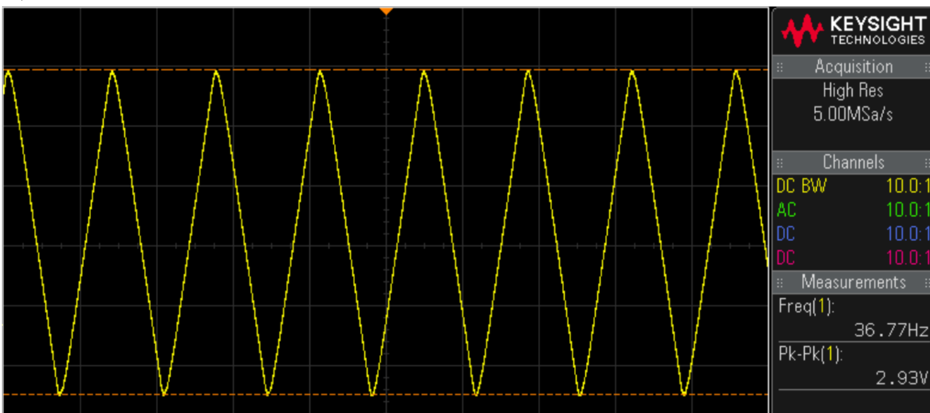
```
        richtung = -1
    if n < min:
        richtung = 1
    #print(n) # Nur Kontrolle
    time.sleep_us(w) # warten
```

Und das kommt dabei heraus:

a) mit 5 kHz PWM

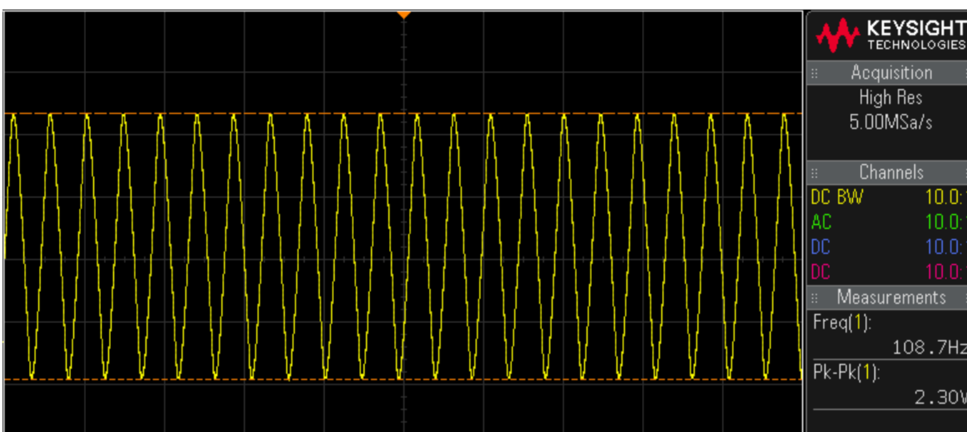


b) mit 50 kHz PWM



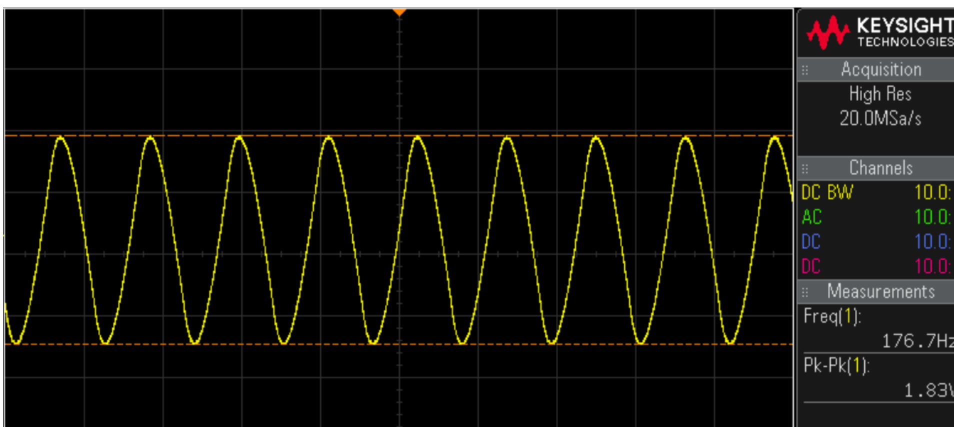
Der bei 5 kHz noch deutlich sichtbare Rippel verschwindet bei 50 kHz!

Die Frequenz des Dreiecksignals ist mit 36 Hz mehr als vier Mal von der Grenzfrequenz von 159 Hz entfernt. Jetzt ändere ich im Programm den Wert von „w“ von 100 auf 10:



Die Frequenz steigt nur auf 108 Hz und nicht auf 360 Hz! Der Spitze-Spitze Wert der Spannung sinkt zugleich von 2,93 V auf 2,30 V, d.h. unser Filter beginnt schon zu wirken.

Wir sehen hier auch die erwähnte „langsame“ Ausführungsgeschwindigkeit von Python. Die Berechnungen und Zuweisungen innerhalb der Schleife dauern ja auch einige Mikrosekunden. Selbst mit der Auskommentierung der „time.sleep_us“ Zeile, also keiner Wartezeit mehr, erreiche ich nur 176 Hz und das Ergebnis sieht jetzt so aus:



Das ähnelt einer Sinuskurve. Die Amplitude ist auf 1,83 V gesunken. Und in der Tat, da ein Tiefpass-Filter ja nur die tiefen Frequenzen durchlässt, werden die Spitzen des Dreiecks, die seine höchsten Frequenzen darstellen, abgeschliffen.[123] Wir befinden uns nun ja auch schon deutlich oberhalb der errechneten Grenzfrequenz.

Ich möchte noch besonders auf den Einfluss der Größe „step“ hinweisen. Er wurde im Beispiel so gewählt, dass sich die theoretisch möglichen 65536 Schritte auf 100 reduzierten. Man kann sehen, dass bei mehr Schritten sich die Häufigkeit der Schleifendurchläufe erniedrigt, was mit einer Senkung der Frequenz der Dreieckspannug einhergeht. Verringern wir dagegen die Zahl der Schritte, um höhere Frequenzen zu erreichen, weichen wir immer mehr von der gewünschten Kurvenform ab :-((

Mit Micropython auf dem Pi Pico haben wir jetzt eine Leistungs-Grenze erreicht. Unser Mikrocontroller ist zu 100 % ausgelastet!

Und noch eine wichtige Anmerkung zum Programm. Die beiden „if“-Abfragen verändern je nach Ergebnis die Geschwindigkeit des Schleifendurchlaufs durch die zusätzlich auszuführende Zuweisung an die Variable „richtung“. Das bewirkt eine kurzfristige Frequenzänderung; man nennt das Jitter (zittern). In einem realen Programm müsste man das kompensieren, indem man z.B. einen „else“-Zweig mit gleich lang laufendem Code, der nichts bewirken muss, einfügt.

3 Kommunikation ist alles

Unser Mikrocontroller verfügt über jede Menge Kommunikations-Schnittstellen. Im Kapitel „Mikrocontroller Hardware“ wurde u.a. aufgeführt, dass der Pi Pico

- 2 × SPI
- 2 × I2C
- 2 × UART

123 Der Mathematiker Jean Baptiste Joseph Fourier fand 1822 heraus, dass man jede periodische Kurvenform aus Sinuskurven verschiedener Frequenzen zusammensetzen kann. Die Spitzen der Dreiecksspannung stellen besonders hohe Frequenzen dar, weil „Frequenz“ auch als Spannungsänderung pro Zeiteinheit gedeutet werden kann. Theoretisch hat ein „Knick“ eine unendlich hohe Frequenz.

- 5 × 12-bit ADC[124]

enthält. Den ADC Bausteinen haben wir uns schon in einem anderen Kapitel gewidmet. Auch bei ihnen handelt es sich um Kommunikation im weiteren Sinne; die ADC „hören“ quasi auf ein analoges Signal von außen.

Die anderen genannten Schnittstellen kommunizieren auf direktere Weise. Eine benutzen wir schon die ganze Zeit, wenn wir im REPL etwas eingeben und die Antwort des Mikrocontrollers auf dem Bildschirm sehen. Die Schnittstelle nennt sich UART[125], was für Universal Asynchronous Receiver-Transmitter, zu deutsch universeller, asynchroner Empfänger und Sender, steht. Die USB Schnittstelle vom PC zum Mikrocontroller arbeitet (zum Mikrocontroller hin) mit folgendem Codierverfahren: Es werden gleichlange Signale, die mit Synchronisationspulsen eingerahmt werden, gesendet. Dadurch ist es möglich, ohne eine gemeinsame Taktleitung, wie sie z.B. I2C benutzt, zu jeder beliebigen Zeit zu beginnen zu senden. Der Empfänger reagiert auf die erste Flanke. Innerhalb der Start- und Stopbit genannten Pulse enthält das Signal zeitlich hintereinander Nullen oder Einsen. Man muss genau wissen a) wie viele Bits übertragen werden (6, 7, 8, 9 sind möglich), wieviele Stopbits (1 oder 2 sind erlaubt) und man muss b) die Geschwindigkeit kennen, sonst lässt es sich nicht decodieren. In der Mikrocontrollertechnik stellt der UART bis heute ein absolutes Muss dar. Unser Pi Pico hat zwei, einer davon mit der USB Schnittstelle verbunden, der andere frei verfügbar. Es handelt sich um eine bidirektionale, zudem *gleichzeitig* sende- und empfangsfähige Hardware. Die Daten gehen nur über je eine Leitung pro Richtung und werden byteweise verarbeitet. Der USB Anschluss befindet sich sozusagen oben drüber; früher benutzte man eine spezielle Hardware, den RS232 Anschluss, auf beiden Seiten. Der ist heute obsolet und so hat man das UART Protokoll einfach in das USB Protokoll und seine Hardware eingebettet.

Für die Kommunikation gibt es wichtige Funktionsparameter. Die Geschwindigkeit, meist in Bit pro Sekunde ausgedrückt, ist die Wichtigste. Unsere Kommunikation mit dem Pi Pico erfolgt mit 115200 Bit/s. Die krumme Zahl rührt zum einen aus der Geschichte her, als Datenübermittlung noch z.B. mit 75 Bit/s erfolgte. In der Folge verdoppelte man anfangs diese Geschwindigkeiten: 150, 300, 600, 1200, 4800, 9600, 19200 um dann später, nicht mehr mit Verdoppelungen, bis 115200 (und höher) zu kommen. Jetzt waren es die Einschränkungen, die die Takterzeugung[126] auferlegte, die krumme Zahlen erforderten.[127]

UARTs verfügen neben den beiden Datenleitungen über optionale Steuerleitungen, die ein sogenanntes Hardwarehandshaking[128] möglich machen. Die empfangende Seite kann der Sendenden asynchron über eine separate Leitung mitteilen, dass sie keine Daten mehr annehmen kann. Das verhindert Datenverluste. Auch eine Softwarelösung dafür, Xon / Xoff genannt, steht optional zur Verfügung. Hier wird ausgenutzt, dass ein Empfänger auch während des Empfangs Daten senden kann. Ist sein (fast) Puffer voll, sendet er das Byte Xoff, kann er wieder Daten verarbeiten, wird

---

124 Davon sind nur drei frei verfügbar. Zwei sind intern mit einem Temperaturfühler und der Spannungsversorgung verbunden. Man kann damit die Temperatur (des Chips) und die Höhe der Versorgungsspannung messen.

125 Der UART braucht zwei Signalleitungen plus Masse. Er kann hardwareseitig gleichzeitig senden und empfangen.

126 Der Takterzeugung ist ein eigenes Kapitel („Im Takt“) gewidmet.

127 Um aus dem Timertakt eines Mikrocontrollers den für den UART erforderlichen Takt zu generieren, muss er ganzzahlig geteilt werden. Dazu benutzt man üblicherweise zwei Teilerbausteine. Damit ist aber nicht jede beliebige Baud-Rate möglich.

128 Das englische "hand shake", Hände schütteln, beschreibt ziemlich gut, worum es geht.

Xon gesendet.[129] Beispiele für die Kommunikation über UARTs sind nicht nur unsere Mikrocontroller, sondern auch Messgeräte. Mein Digitalvoltmeter kann mit meinem PC über die serielle Schnittstelle genauso reden wie mein Labornetzteil.

SPI und I2C sind neueren Datums[130]. Sie dienen nicht dem Datenaustausch mit einem externen Gerät, sondern erlauben es Bausteinen auf einer Platine miteinander zu kommunizieren. Wenn ich z.B. ein Display an meinen Mikrocontroller anschließen möchte, fließen die Daten höchstwahrscheinlich über eine von beiden Schnittstellen.

4 Von Mensch zu Mikrocontroller

Bislang haben wir praktisch immer über die USB-Serielle Schnittstelle mit dem Mikrocontroller kommuniziert. Aber wie gibt man etwas ein, wenn keine Tastatur angeschlossen ist? Und das ist, nach der Entwicklung eines Projekts, der Normalzustand.

Die Königsdisziplin hier wäre ein Touchpad. Bei ausreichend großem Bildschirm die ideale Möglichkeit, Eingaben zu tätigen. Da der Bildschirm den Kontext zeigt, sind so auch komplexe Eingaben wie Konfigurationen (IP-Adresse, Namen usw.) denkbar. Zugleich ist es die teuerste Variante. Und die programmiertechnisch aufwendigste.

Häufig braucht man aber gar nicht so viele verschiedene Eingabemöglichkeiten. Die Einleitung eines Neustarts, einen Zahlen-Parameter vergrößern oder verkleinern: Oft reichen zwei bis drei Taster. In Kombination mit Leuchtdioden kann so eine zuverlässige Interaktion mit dem Anwender aufgebaut werden.

4.1 Ein Taster, viele Möglichkeiten

Was kann man mit *einem* Taster alles mitteilen? Wenn man nun denkt, damit geht nur eine Information, dann denkt man zu kurz. Denn was ist, wenn ich auch die *Dauer*, wie lange ich den Taster gedrückt halte oder – wie bei einer Maustaste – auch Mehrfachbetätigungen erfasse?

Es hat sich eingebürgert, zwischen kurz und lang drücken, manchmal noch *sehr* lang drücken und ein- oder zweifach Druck, dann kurz hintereinander, zu unterscheiden. Damit haben wir schon mindestens drei Möglichkeiten für nur einen Taster. Unterstützen wir dies durch eine LED-Ausgabe, die für den Anwender seine Eingabe bestätigt und / oder den jeweiligen Modus anzeigt, verfügen wir mit wenigen, kostengünstigen Bauteilen über eine variantenreiche Eingabe-möglichkeit.

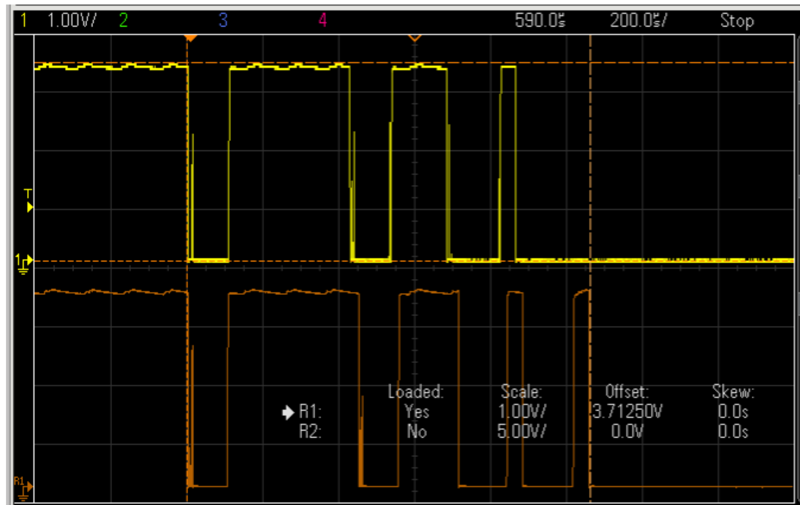
Bei zwei Tastern kommt Druck auf beide Taster zugleich hinzu. Außerdem kann ich nun einfach ein verkleinern-vergrößern realisieren.

Noch eine kurze Vorüberlegung zur programmiertechnischen Umsetzung. Wenn ich die Dauer des Drückens erfassen will, bin ich gezwungen, nicht mehr das Schließen allein, sondern auch das Loslassen zu erfassen. Außerdem muss ich die Zeiten messen können. Das erfordert Programmierkenntnisse, die wir erst später behandeln.

129 Es müssen jeweils beide Seiten die Verfahren unterstützen.
130 I2C ist erst 42 Jahre alt ;-)

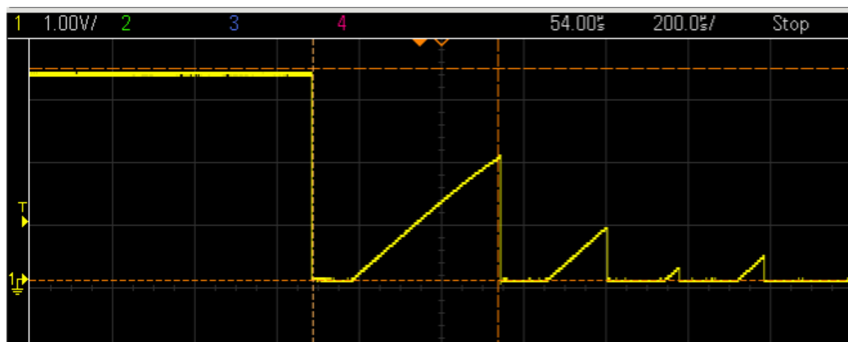
4.2 Verprellt

Vielleicht hat der ein oder andere schon davon gehört: Taster prellen. Das Bild weiter unten zeigt zwei Oszillogramme einer Tastenbetätigung. Unten eine aufgezeichnete, oben die aktuelle. Deutlich ist die mehrfache Umschaltung zu sehen und die zwar ähnlichen, aber doch deutlich unterschiedlichen Kurvenzüge. Die untere Kurve dauert ca. 1,05 und die obere 0,86 ms. Wie lange es genau dauert, bis diese Schwingung vorbei ist, hängt vom Taster, aber auch davon, wie man ihn drückt, ob kräftig oder zaghaft, ob schnell oder langsam, ab. Auch Alterung[131] spielt eine Rolle.



Wenn ich in meinem Programm nur einen einmaligen Druck auf den Taster erfassen möchte, kann ich eine lange Sperrzeit[132] vorsehen, in der weitere Signale vom Taster ignoriert werden. Damit wird aber die Reaktion träger.

Eine „Lösung“ ist ein Kondensator parallel zu den Schaltkontakten. Das sieht dann so wie im nächsten Bild aus. Die ersten beiden ansteigenden Pulse könnten den GPIO noch beinflussen; sie enden aber schon nach 0,5 ms. Wenn ich einen größeren Kondensator verwende, vergrößert sich aber auch die Anstiegszeit des Signals, wenn ich den Taster wieder loslasse. Und beim Loslassen prellen Schalter auch! Ohne das Messmittel Oszilloskop ist da kaum eine optimale Anpassung an den jeweiligen Schalter möglich. Und, wie gesagt, jeder Schalter ist anders!



4.3 Probieren geht über studieren ...

Wenn wir aber schon mit einem Mikrocontroller arbeiten, könnten wir da nicht per Software entprellen? Im Prinzip ja, könnte man sagen. Aber alle mir bekannten Algorithmen lindern das Problem nur und daher braucht man immer eine kleine Verzögerung.

Möchte ich die Tastendrücke zählen, komme ich um eine großzügig dimensionierte Sperrzeit nicht herum. Vor Jahren habe ich mal für das Kinderspiel, bei dem man eine kleinen Öse an einem Stiel um einen gewundenen Draht herumführen muss, ohne ihn zu berühren, eine Fehlerzahl-Anzeige entwickelt. Hier tritt das gleiche Problem auf, weil eine Drahtberührung in Wirklichkeit aus zig

131 Ich besitze Geräte mit Drehgebern, die im Laufe der Jahre immer schlechter auf Drehen reagieren. Wahrscheinlich hat sich durch Materialermüdung die Prellzeit wesentlich geändert.

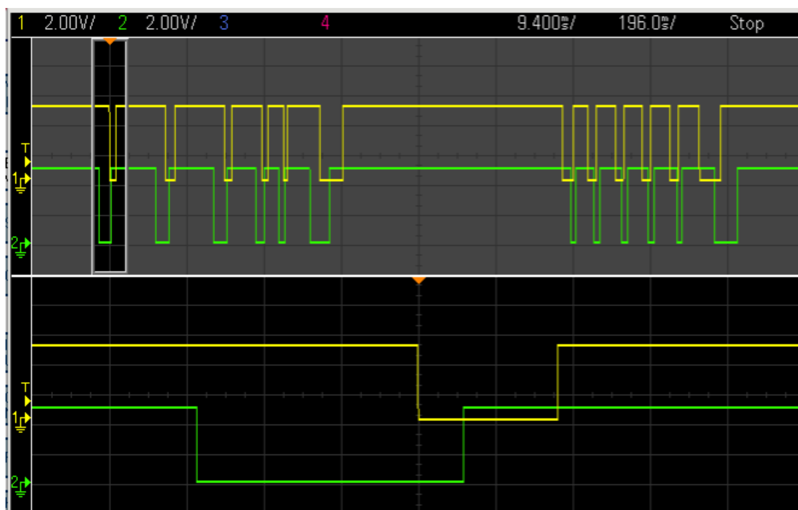
132 Lang bedeutet hier mehr als 15 ms!

Kontaktierungen besteht. Ich musste mich entscheiden, wie ich das werte. Mein Wahl fiel auf eine Sperrzeit von 0,5 Sekunden, bevor erneut eine Berührung den Zähler hoch setzte. Für viele Kinder war das zu kurz. Beim Weitermachen nach einem Fehler erzeugten sie gleich wieder einen Fehler. Sehr frustrierend! Warum ich das erzähle? Um auf den ganz wichtigen Aspekt des **Praxistests** hinzuweisen!

4.4 Der Drehgeber

So schön einfach und billig Taster auch sind, für manche Aufgaben sind sie nicht so gut geeignet. Stellen wir uns vor, wir wollen die Helligkeit einer LED mit PWM Steuerung einstellen. Bei 1% Auflösung müsste ich dazu einen Taster gegebenenfalls hundert Mal drücken – puh.

Drehgeber[133] (auch Drehimpulsgeber, Drehregler, Drehencoder, Inkrementalgeber oder Rotary Encoder genannt) sehen aus wie ein Potentiometer[134], haben aber keinen Anschlag. Mit ihnen werden zwei phasenverschobene Pulse A und B erzeugt; die werden gezählt und durch die Phase die Drehrichtung ermittelt. Drehe ich nach links, wird z.B. ein Zähler vermindert, drehe ich nach rechts, erhöht. Je nach Bauart gibt so ein Drehgeber bei jeder vollständigen Drehung ca. 15 Impulse ab, mit etwas mehr als sechs Umdrehungen kann ich meine LED von 0% auf 100% regeln. Durch geschickte Programmierung kann ich auch die Drehgeschwindigkeit auswerten und bei hoher Geschwindigkeit z.B. in Zehnerschritten zählen.



Drehgeber prellen leider auch. Im Falle eines Drehgebers kann ich die Sperrzeit auch nicht so einfach hoch setzten. Er verliert dann nämlich Impulse, was die Benutzererfahrung sehr beeinträchtigt. Wer schon einmal ein billiges Gerät mit Drehregler in Händen hatte, kennt das.

Optische Drehgeber, die kein Prellen kennen, kosten leider leicht 20 Mal so viel wie mechanische.

Hier sehen wir ein Oszillogramm eines Drehgebers. Zuerst wurde er 12 Mal[135] nach rechts, dann 12 Mal nach links bewegt.[136] Im unteren Bildteil, dem gezoomten Bereich, sieht man, dass, wenn das gelbe Signal auf Low[137] geht, das Grüne schon Low ist. Oben rechts erkennt man, dass beim Übergang auf Low das Grüne noch High ist. Betrachtet man die steigenden Flanken, ist es genau umgekehrt. Damit erfolgt die Drehrichtungserkennung. Das hier kein Prellen erkennbar ist, liegt an der langsamen Horizontalablenkzeit während der Messung, also der groben zeitlichen Auflösung.

133 Ein Drehregler gibt richtungsabhängige Impulse aus, so dass z.B. eine Variable erniedrigt oder erhöht werden kann. Ein sehr informativer Artikel findet sich unter „https://www.mikrocontroller.net/articles/Drehgeber“.
134 Siehe dazu Kapitel „Was ist ein Potentiometer“.
135 Beim meinem verwendeten Drehgeber existiert eine spürbare mechanische Rastung pro Flanke. Das muss nicht bei jedem Drehgeber sein.
136 Die Auswertung des Signals ist recht kompliziert, weshalb ich erst später darauf eingehe.
137 Mit „Low“ und „High“ werden die logischen Zustände Spannung aus oder an bezeichnet.

Wie müssen wir nun vorgehen, um den Drehgeber auszulesen? Eine vereinfachte Beschreibung: Die erste, hier negative Flanke[138] des Signals A (gelb) startet die Auswertung. Zuerst müssen wir prüfen, wie viel Zeit seit der letzten negativen Flanke vergangen ist. War diese Zeit länger als die von uns definierte maximale Prellzeit, lesen wir den Pegel des Signals B (grün) und inkrementieren oder dekrementieren davon abhängig einen Zähler. Danach setzten wir den Timer wieder zurück. War die Zeit dagegen kürzer als die Prellzeit, ignorieren wir das Signal. Wie das mit Micropython umgesetzt werden kann, kann in den zwei Beispielprogrammen „isr_encoder.py“ und „isr_encoder2.py“ angeschaut werden. Es im Detail zu erklären sprengt hier den Rahmen.

Diese Form der Codierung eines Signals nennt man übrigens nach seinem Erfinder Gray-Code.

4.5 Text und Bild

Auch Anzeigen gehören grob gerechnet zu den Aktoren. Sinnvollerweise unterscheidet man zwischen reinen Textanzeigen und grafischen Displays. Früher waren - schon aus Kostengründen - Textanzeigen sehr beliebt. Sie hatten zwischen ein und vier Zeilen mit meist sechzehn Zeichen. Ihre Vorzug liegt darin, relativ einfach angesteuert werden zu können. Inzwischen sind die Preise für grafische Anzeigen soweit gefallen, dass auch deren Nutzung nichts mehr im Wege steht.

Was macht eine grafische Anzeige aus? Während man bei den Textanzeigen nur Buchstaben senden und darstellen kann, erlaubt ein grafisches Display die Ansteuerung jedes einzelnen Pixels[139] auf der Fläche der Anzeige. Man muss sich aber vor Augen führen, dass bei gängigen Displays mit z.B. 128 mal 160 Pixeln bei ca. 65000 Farben pro Pixel 2 oder 3 Bytes benötigt werden (entspricht maximal 61440 Bytes). Die müssen (und können) aber nicht vom Mikrocontroller bereitgestellt werden, sondern befinden sich in den sogenannten Grafik-Controllern.[140] Diese verwalten das eigentliche Display[141], nehmen die Daten per SPI oder I2C entgegen, speichern sie intern und stellen für jedes Pixel die gewünschten Farb- und Helligkeitswerte ein.

Über wie viel „Intelligenz“ ein Grafik-Controller verfügt, entscheidet über den Aufwand, den mein Mikrocontroller aufwenden muss, um z.B. eine schräge Linie in blau auf das Display zu zaubern. Der ST7735S kann das Display rotieren und scrollen – immerhin. Die Linie müssten wir aber alleine zeichnen. Daher gibt es auch noch Treiber-Software in Micropython, die einem diese und weitere komplizierte Berechnungen abnehmen. Da das nicht spezifisch für ein bestimmtes Display ist, gibt es im Modul „framebuf“ eine Klasse „FrameBuffer“, die all diese netten Funktionen bereitstellt, hier die Ausgabe von „help“:

```
object <class 'FrameBuffer'> is of type type
 fill -- <function>
 fill_rect -- <function>
 pixel -- <function>
 hline -- <function>
 vline -- <function>
 rect -- <function>
 line -- <function>
```

138 Sie entsteht beim Schließen des Schalters im Drehgeber.

139 Als „Pixel“ wird ein Bildpunkt bezeichnet. Da er farbig sein kann, besteht er ggf. aus drei Subpixeln für die Farben Rot-Grün-Blau.

140 Das im Workshop schon gezeigte Display Pico LCD 1.8 verwendet einen ST7735S.

141 Mit *eigentlichem* Display ist die Hardware gemeint, die die einzelnen Pixel darstellt, also LCD oder Oled, im Gegensatz zur gesamten Hardware bestehend aus Board, Grafik-Controller, Display usw.

```
ellipse -- <function>
poly -- <function>
blit -- <function> # "Block Image Transfer" rechteckigen Bereich verschieben
scroll -- <function>
text -- <function>
```

Die Namen der Methoden sprechen für sich. Der Grafik-Treiber für ein spezielles Display erbt von dieser Klasse und braucht *nur* noch Anpassungen an den Grafik-Controller vorzunehmen ;-))

5 Wie viele GPIOs braucht ein Mikrocontroller?

Die lustige Antwort: Er hat immer einen weniger als das eigene Programm benötigt! Viele Mikrocontroller-Hersteller haben daher Familien von Controllern im Programm, d.h. es gibt den fast gleichen Controller mit 8, 16, 32 … Beinen und damit mit immer mehr GPIOs.

Der Pi Pico hat 26 GPIOs. Dazu kommen noch spezialisierte Anschlüsse für ADC usw. Das hört sich viel an, aber …

Stellen wir uns vor, wir möchten Sieben-Segment-Anzeigen[142] mit dem Mikrocontroller ansteuern. Wir benötigen acht GPIOs für die sieben Segmente einer Ziffer und den Dezimalpunkt plus die Versorgungsspannung. Bei sechs Ziffern wären das dann schon 6 x 8 = 48 :-((.

Oder ich möchte Taster in 5 x 5 Spalten und Reihen benützen: Schon bleibt mir nur noch ein GPIO Pin übrig.

So geht das nicht. Hier kommt „multiplexen“ ins Spiel. So nennt man Verfahren, bei denen Signale „umgeschaltet“ werden. Machen wir es lieber konkret: Anstatt für jede Sieben-Segment-Anzeige 8 GPIOs zu verwenden, schaltet man alle gleichen Segmente der Ziffern parallel an je einen GPIO und macht dafür die Versorgungsspannung schaltbar. Jetzt schaltet man die GPIOs für die Segmente der ersten Ziffer wie gewünscht ein und die Versorgungsspannung kurz an, dann macht man das für die zweite Ziffer usw. Geschieht das schnell genug, ergibt sich, wie bei der PWM für eine LED, eine nicht flackernde, sechsstellige Anzeige. Diese Art der Ansteuerung braucht nur noch einmalig die acht Segmentanschlüsse und so viele GPIOs, wie es Ziffern gibt, also hier 8 + 6 = 14. Einen Nachteil hat das Ganze doch: Die Stromaufnahme der Sieben-Segment-Anzeige kann ja acht Mal einer LED (wenn alle Segmente an sind) entsprechen und daher nicht von unserem Pi Pico direkt geliefert werden. Wir brauchen also noch sechs Schalttransistoren. Für große, sehr lichtstarke Sieben-Segment-Anzeigen übersteigt aber auch der Segmentstrom die Spezifikation des Pi Pico. Das wäre zwar eventuell auch ohne Multiplexen der Fall, aber wir müssen ja noch ausgleichen, dass jede Ziffer nur in einem Sechstel der Zeit an ist, also einen sechs Mal höheren Strom für gleiche Helligkeit fließen lassen.

Positiv bezogen auf den Hardwareaufwand ist dagegen, dass wir nur noch 8 statt 6 x 8 Vorwiderstände benötigen. Wir erinnern uns: LED **nie** ohne Vorwiderstand!

In der Datei „segment_7.py“ zeige ich ein funktionsfähiges Beispiel. Es nutzt AsyncIO wegen des Multiplexens. An seiner Länge kann man erahnen, dass der Teufel im Detail steckt. So gibt es z.B. Anzeigen mit gemeinsamer Anode und mit gemeinsamer Kathode. Für die Ansteuerung bedeutet dies, dass man zum Einschalten eines Segments entweder eine 0 oder eine 1 am GPIO ausgeben

142 Das sind die – meist roten – LED Anzeigen, die Zahlen mit sieben Balken darstellen. Sie haben je einen Anschluss für jedes der sieben Segmente, einen für den Dezimalpunkt und einen für die gemeinsame Anode oder Kathode.

muss. Schaltet man noch Transistoren dazwischen kann sich das wieder umkehren. Auch die Multiplex-Schalter müssen beide Logiken berücksichtigen. Die Invertierung einer 1 in eine 0 und einer 0 in eine 1 kann man mit einer „if .., else ...“ Konstruktion umsetzten. Eleganter ist die Nutzung des XOR Operators „^“. XOR (exklusives Oder) funktioniert so, dass 1 ^ 1 eine 0 ergibt, 1 ^ 0 dagegen 1. Mit dem einem der Argumente kann man also eine Invertierung erreichen. Hilfreich ist hier zudem, dass True und False als Wert auch funktionieren; True entspricht der 1, False der 0. Zum einfachen ausprobieren besitzt das Programm eine „test()“ Methode, die die darstellbaren Zeichen eines 7-Segment-Displays der Reihe nach zeigt. Mit „show()“ kann ohne AsyncIO eine einzelne 7-Segment-Anzeige angesteuert werden.

Es sei noch kurz erwähnt, dass die Lösung für unsere 25 Taster ebenfalls im Multiplexen besteht. Hier wird noch ein wenig elektronische Verschaltung gebraucht. Im Internet gibt es eine Fülle von spannenden Seiten dazu.[143]

6 Im Takt

Ein Mikrocontroller braucht, um zu funktionieren, einen Takt, d.h. ein periodisches, elektrisches Signal. Das hängt mit der internen Funktionsweise herkömmlicher CPUs zusammen: Der Takt, oder besser das Taktsignal, dient dazu, die einzelnen Funktionsschritte zu steuern. Vereinfacht löst ein Takt das Holen des nächsten Befehls aus dem Speicher, der nächste Takt die Ausführung des Befehls aus usw.

Je schneller der Takt ist, desto schneller arbeitet unser Mikrocontroller. Leider steigt mit höherem Takt auch die Stromaufnahme und die Erwärmung.

Der Takt wird oft mit einem (Quarz-)stabilen Oszillator erzeugt. Nehmen wir an, dessen Frequenz betrüge 0,5 MHz, also eine halbe Million Schwingungen pro Sekunde, dann wäre ein Takt 2 µs lang.

Nutze ich nun den Funktionsaufruf „time.sleep_ms(1)“ muss Micropython 500 Takte warten. Bei der Funktion „time.sleep_us“ hätte Micropython nun aber ein Problem: Selbst, wenn es nur einen Takt wartet, wäre es schon zu spät :-((

Als wir über den UART sprachen, wurden die krummen Geschwindigkeiten und die Verdopplung thematisiert. Bei einer Baud- oder Bitrate von 150 dauert ein Bit ~0,006666667[144] ms (1 / 150), bei 115200 Baud sind es nur noch ~8,680555556 µs. Ich muss also sehr präzise alle 6,6 ms bzw. 8,68 µs den Ausgang meines GPIO auf den neuen Wert setzen. Dazu habe ich 3300 Takte bei 150 Baud und 4,34 Takte bei 115200 Baud Zeit. Ich bräuchte einen Zähler, der bei jedem Takt hoch gesetzt und dann mit einem Wert verglichen werden müsste. In Software, gar in Python, wäre das geschwindigkeitsmäßig nicht machbar. Und selbst wenn: Das Problem mit den Nachkommastellen hätte ich aber auch dann noch nicht gelöst.

Aus diesen Gründen verwendet man für die Generierung von Taktzeiten sogenannte Hardware-Teiler und Timer. Die Teiler können in ihrer einfachsten Form zwar nur durch zwei und vielfache davon teilen, doch in Verbindung mit den oft 8- oder 16-Bit Timern lassen sich auch „krumme“

143 Z.B. hier: https://www-user.tu-chemnitz.de/~heha/Mikrocontroller/Tastenmatrix.htm oder hier https://de.wikipedia.org/wiki/Multiplexverfahren

144 Dieser Wert ist schon gerundet, daher setzte ich die Tilde davor.

Werte erzielen. Man realisiert das, indem man mit dem Teiler in die Größenordnung der Zeiten herunter teilt, in denen man messen will und dann mit dem Timer die genaue Zeit festlegt.

Ein Beispiel: Mit einem 8-Bit Timer erfolgt bei 500000 Takten pro Sekunde alle ~0,512 ms ein Überlauf. Das ist viel zu schnell, um auf 6,6 ms zu kommen. Deshalb teile ich zunächst die 500000 Takte durch 256 (= 2^4) und erhalte einen Takt von ~31250 Hz, etwas mehr als 208 mal so schnell wie die gewünschten 150 Hz. Der Hardware-Timer wird nun so eingestellt, dass er nach 208 Takten (das ist kleiner als der Überlauf von 256!) ein Signal ausgibt. Damit erhalte ich recht genau 150 Hz!

Um 115200 Baud zu erzeugen, reichen 0,5 MHz nicht aus. Zum Glück beträgt die tatsächliche Taktrate des Pi Pico 125 MHz. Damit lassen sich Auflösungen bis in den Mikrosekunden-Bereich realisieren. Die Funktion „time.ticks_us“ von Micropython macht davon Gebrauch.

Man sollte aber immer berücksichtigen, dass die Zeiten nur Näherungen darstellen. Je näher sie sich der Taktrate nähern, desto ungenauer werden sie.

7 Wenn es mal hängt …

Programme können sich „aufhängen“. So nennt man den Zustand, wenn von außen betrachtet ein Programm scheinbar nichts mehr macht. In Wirklichkeit ackert das Programm z.B. wie wild in einer Schleife. Nicht immer haben wir die Möglichkeit, das zu verhindern, aber wir haben eine Möglichkeit, sozusagen per Programm aus diesem ungewollten Zustand auszubrechen. Im englischen nennt sich das Watch-Dog: Wachhund.

1. Die Idee ist ganz einfach die der Totmann-Schaltung in einer Lokomotive: Der Zugführer muss von Zeit zu Zeit einen Knopf drücken; unterlässt es das, bremst der Zug automatisch ab.

Mit „wdt = machine.WDT(timeout=5000)“ aktiviere ich den Mechanismus. Danach muss ich mindestens alle 5 Sekunden[145] „wdt.feed()“ aufrufen, sonst erfolgt ein Reset, also ein Kaltstart.

Nach dem Start kann ich mit der Methode „machine.reset_cause()“ abfragen, warum der Mikrocontroller (neu)gestartet ist. Nach einem Watch-Dog Reset zeigt die Methode „3“ an. Ein „Power-On“ ergibt eine „1“.

145 Der „timeout“ wird in Millisekunden angegeben. Der höchste Wert ist 8388 ms.

V Netzwerke

1 Was ist ein Netzwerk?

Zunächst einmal: Es gibt nicht nur eine Art von Netzwerk. Netzwerke unterscheiden sich einmal darin, über welches Medium sie Daten übertragen. Das kann ein spezielles Kabel wie Koax oder Cat5 sein oder eben auch durch die Luft. Da gibt es dann auch wieder viele Unterschiede: die Frequenz der Funkverbindung, die Art der Modulation (das ist die Methode, wie die Daten in die Trägerfrequenz hineingemischt werden).

Hat man dies, noch sehr hardwarenahe gemeistert, geht es mit den Möglichkeiten weiter. Wie kommt meine Information von A nach B und nicht zu C. Muss ich den vollständigen Weg von A nach B wissen oder reicht es, dass ich meine Information mit der Zieladresse losschicke und darauf hoffen kann, dass es unterwegs schon etwas gibt, das den Weg weiß[146]?

Damit man ein bisschen Ordnung in all diese unterschiedlichen Möglichkeiten bringt und das Zusammenspiel von verschiedenen Techniken leichter gewährleisten kann, hat man vor vielen Jahren zwei Modelle entwickelt, von denen sich eins, das US-amerikanische, gegenüber dem europäischen[147] durchgesetzt hat. Es nennt sich TCP/IP. Beide Modelle sind gar nicht so unähnlich, beide verwenden Schichten, denen bestimmte Aufgaben übertragen werden. Das ist nun sehr abstrakt, deshalb ein Beispiel, das sehr vereinfacht das Modell erläutert. Zunächst die Schichten im TCP/IP Modell:

Anwendung	Interpretation der Rohdaten
Transport	Gegenseitige Ende-zu-Ende Verbindung
Vermittlung	Wie erreiche ich meinen Kommunikationspartner
Netzzugang	Die physikalische Schicht, Kabel, Luft usw.

Von unten nach oben:

Netzzugang: Da jede Schicht – zumindest logisch - getrennt ist, spielt es für die darüber liegenden Schichten keine Rolle, ob ich über Funk, Kabel oder Glasfaser ins Netzwerk / Internet gehe.

Die Vermittlungsschicht ist die Telefonzentrale. Ich möchte mit Herrn Müller in Argentinien telefonieren. Ich weiß seine dortige Rufnummer. Ich rufe bei der Telefonvermittlung in Deutschland an und die stellen eine Verbindung zur Vermittlung in Argentinien her. Die wiederum stellt die Verbindung mit Herrn Müller bereit.

Dort klingelt es. Auf der Transport Schicht: Herr Müller hebt ab und ich kann mit ihm direkt sprechen und er mit mir. Ich habe die Verbindung aufgebaut, aber die Kommunikation ist bidirektional. In der Fachsprache ist Herr Müller der Server und ich bin der Client[148]. Herr Müller muss meine Telefonnummer nicht kennen!

146 So funktioniert nämlich unser Internet.
147 Das nennt sich ISO/OSI Modell.
148 Nach meinem Wissen gibt es keine einheitliche deutsche Übersetzung dafür. Dem Sinn nach ist der Client der Kunde und der Server der Mensch, der an der Ladentheke bedient.

Die Anwendungsschicht ist beim Telefon die Sprache. Im Internet kann es HTTP, Telnet, SIP[149] usw. sein.

Halten wir das Wichtigste für uns fest: Der Client initiiert die Verbindung; er muss die Adresse des Servers wissen. Einmal hergestellt, kann die Verbindung Daten in beide Richtungen übertragen. Vielleicht ist es ja offensichtlich, aber es ist ganz bedeutsam, dass der Server von Amazon nicht meine (IP-)Adresse wissen muss, damit ich dort einkaufen kann. Er erfährt sie aber, wenn ich mich mit ihm verbinde, denn das TCP/IP Protokoll sieht vor, dass der Absender seine IP-Adresse mitsendet. Nur so kann der Server antworten, denn der Weg der Daten geht, wie im Telefonbeispiel angedeutet, über viele Stationen[150].

In den Anfängen der Datenübertragungstechnik gab es lange einen Streit über die Frage, ob Leitungs- oder Paketvermittlung besser sei. Das klassische Telefon ist ein Beispiel für Leitungsvermittlung: Es wird eine elektrische Punkt zu Punkt Verbindung zwischen den Teilnehmern erstellt. Da das nur zu Beginn einmalig erforderlich ist, ist die Effizienz der Datenübertragung sehr hoch. Bei Paketvermittlung werden die Daten in kleine Blöcke zerteilt, die jeder für sich weitergeleitet werden. So fällt bei TCP in jedem Paket ein mindestens 20 Byte langer Steuerdatenanteil an. Sende ich 20 Byte Nutzdaten, ist der Wirkungsgrad nur 50% :-((

Damit wäre der Streit doch entschieden? Nein, denn die Leitungsvermittlung erlaubt eben nur Punkt zu Punkt und *eine* Anwendung! Bei Paketvermittlung sind dagegen nicht nur 1 zu „viele" Verbindungen, sondern auch verschiedene Anwendungen quasi gleichzeitig möglich.

Die Kommunikation in Blöcken bestimmter Länge erzwingt, dass, will ich mehr Daten übertragen, als in einen Block passen, ein weiteres Paket gesendet werden muss. Dabei besteht *jedes* Paket aus den Nutzdaten und dem Steuerdatenanteil, also der Zieladresse, Quelladresse usw. und einer fortlaufenden Nummer, die die Reihenfolge der Datenpakete widerspiegelt. Diese Nummer hilft beim Empfänger, die Pakete wieder in die richtige Reihenfolge zu bringen, denn es ist durchaus möglich, dass ein später gesendetes Paket früher eintrifft.

Zur Zeit gibt es parallel zwei Adress-Systeme: IPv4 und IPv6. Letzteres ist neuer und leistungsfähiger. Wir müssen aber meist davon gar nichts wissen, da es – auf der Anwendungsschicht – einen Telefonbuch-Service gibt. Der nennt sich DNS (dynamisches Namensauflösungs System) und funktioniert wirklich wie ein Telefonbuch: Ich schlage unter A wie Amazon nach und erhalte die IP-Adresse. Wenn ich im Browser „www.amazon.de" eingebe, passiert genau das im Hintergrund.

Im Workshop brauchen wir manchmal doch IP-Adressen, z.B., weil für unseren Server auf Raspberry Pi Basis kein DNS zur Verfügung steht. Dann muss ich etwas wie „10.42.0.1" eingeben, um ihn zu erreichen. Das ist eine IPv4 Adresse. Sie besteht aus 4 Zahlen, durch Punkt getrennt, die jeweils von 0 bis 255 reichen können. Eine IP-Adresse besteht immer aus einem Netzwerk- und einem Serveranteil[151]. Dazu werden die vier Zahlenwerte in einen vorderen und einen hinteren Teil getrennt. Im vorderen Teil sind dann einige Netzwerk-Konfigurationsparameter versteckt.[152] Sie bestimmen, welche IP-Adressen innerhalb des lokalen Netzes gültig sind und legen fest, was mit

149 HTTP für Webseiten, Telnet für Kommandozeilenzugang und SIP für Internet-Telefonie.
150 Die heißen Router.
151 Der Serveranteil wird auch „Host"-Anteil genannt, weil Rechner als Host (Gastgeber) bezeichnet werden.
152 Ohne zu sehr ins Detail gehen zu können, legen bestimmte Zahlenbereiche den Netzwerktyp fest.

Adressen passiert, die nicht zu diesem lokalen Netzwerk gehören. Die werden nämlich „geroutet“ (das englische to route kann man am besten mit lotsen übersetzen), d.h. an den Router geschickt. Wie die Telefonvermittlung weiß der Router zwar nicht, wie sie ihr Ziel direkt erreichen können, aber er weiß, wohin er sie als nächstes schicken muss. Mein Heimrouter bekommt nämlich vom Provider eine IP-Adresse, an die er alle Pakete, für die er nicht selbst zuständig ist, weiterleitet. Das funktioniert nach dem Motto: Schau mal, was du damit machen kannst. Der Router beim Provider kennt nun wiederum eine Router-Adresse, an die er weiterleitet usw. bis endlich, nach 5, 10, 20 oder mehr „Hops“ das Ziel erreicht wird. Hier ein Beispiel für eine Internet-Seite mit dem Kommando „traceroute“[153]:

```
traceroute www.wd.de
traceroute to www.wd.de (80.92.65.144), 30 hops max, 60 byte packets
 1  fritz.box (192.168.1.1)  0.622 ms  0.677 ms  0.717 ms
 2  37.99.201.1 (37.99.201.1)  18.742 ms  18.697 ms  18.619 ms
 3  ae12-472.fra20.core-backbone.com (81.95.2.1)  18.475 ms  18.404 ms  18.342 ms
 4  ae2-2025.fra30.core-backbone.com (5.56.17.26)  18.414 ms  18.338 ms  18.204 ms
 5  ipv4.de-cix.fra.de.as35280.volterra.io (80.81.193.10)  18.380 ms  18.326 ms 18.262 ms
 6  100.100.1.37 (100.100.1.37)  18.197 ms  18.960 ms  16.795 ms
 7  107.162.79.3 (107.162.79.3)  16.870 ms  8.739 ms  8.829 ms
 8  107.162.24.198 (107.162.24.198)  23.168 ms  22.624 ms  22.279 ms
 9  irb-99.wd1250-ex4600-b01.as24611.net (80.92.83.224)  83.775 ms  34.781 ms 48.392 ms
10  urlfwd-1.eurodns.com (80.92.65.144)  18.376 ms  18.006 ms  18.994 ms
```

Zunächst wird die IP-Adresse ermittelt, der Name www.wd.de also aufgelöst (80.92.65.144). Nach 0.717 ms erhält mein Router das erste spezielle Paket[154]. Nach 18.697 ms erhält der Router mit der IP-Adresse 37.99.201.1 (2. Hop) das zweite spezielle Paket. Das geht so weiter bis zur letzten Zeile: Hier wird das Zielsystem nach 18.994 ms und 10 Hops erreicht. Die Zeiten werden jeweils vom Beginn der Messung bis zu einem „Hop“ aufgezeichnet. Die Gesamtzeit, um „www.wd.de“ zu erreichen, beträgt also 18.994 ms. Die längste Zeit wird auf dem Weg von meinem Router zum Provider verbraucht (und das trotz Glasfaseranschluss). Die Zeiten sind für jede Messung[155] ein wenig unterschiedlich, so dass der 10. Hop hier im Beispiel scheinbar etwas früher erreicht wird als der 8. und 9. Hop.

Ein anderes Kommando, „ping“, erlaubt die Überprüfung der Erreichbarkeit eines Ziels:

```
ping www.wd.de
PING www.wd.de (80.92.65.144) 56(84) Bytes an Daten.
64 Bytes von urlfwd-1.eurodns.com (80.92.65.144): icmp_seq=1 ttl=58 Zeit=18.8 ms
64 Bytes von urlfwd-1.eurodns.com (80.92.65.144): icmp_seq=2 ttl=58 Zeit=17.6 ms
64 Bytes von urlfwd-1.eurodns.com (80.92.65.144): icmp_seq=3 ttl=58 Zeit=18.0 ms
^C
--- www.wd.de ping-Statistik ---
3 Pakete übertragen, 3 empfangen, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 17.628/18.152/18.807/0.490 ms
```

Wenn man es nicht abbricht, misst es kontinuierlich die Laufzeit eines Paketes bis zum Ziel. Am Ende liefert es noch eine Statistik. „traceroute“ macht zwar fast das Gleiche, benötigt aber insgesamt viel mehr Zeit, die es für die Ermittlung der einzelnen Weiterleitungstationen benötigt.

Wenn das Ziel nicht erreicht werden kann, erhalte ich eine Fehlermeldung:

153 Das ist ein Linux Kommando. Unter Windows gibt es ein funktional gleiches mit dem Namen „tracert“, das auch auf der Kommandozeile (Programm „cmd“) gestartet wird.
154 Auf Netzwerkdienste wie ICMP will ich hier nicht eingehen. Sie dienen der Prüfung, ob und wie Netzwerkverbindungen zustande kommen oder eben auch scheitern.
155 Es werden bei „traceroute“ jeweils drei Messwerte angezeigt: der langsamste, der schnellste und ein Mittelwert.

```
PING mint21 (2a04:6ec0:227:6ab0:b52d:e030:71a9:c827) 56 Datenbytes
Von fedora.fritz.box (2a04:6ec0:227:6ab0:f7b5:ec6:afd:ae88) icmp_seq=53 Ziel 
nicht erreichbar: Adresse nicht erreichbar
```

Hier ist auch einmal eine IPv6 Adresse zu sehen, von mir fett hervorgehoben. Mein Linux Rechner „mint21“ ist ausgeschaltet. Die Namensauflösung klappt, da sie vom Router (per DNS) erledigt wird. Aber der Computer antwortet – verständlicherweise – nicht. Während IPv4 aus einer 32 Bit Zahl besteht, umfasst eine IPv6 Adresse 128 Bit. Das war auch der Grund für die Entwicklung. Die IPv4 Adressen gingen nämlich zur Neige; heute sind nur noch ganz wenige frei.

Nanu, zu Hause habe ich doch jede Menge IPv4 Adressen! Ja, aber das sind „private“, nicht routebare Adressen.[156] Der Heimrouter selbst hat nach außen vom Provider eine „öffentliche“ IP-Adresse erhalten. Alle Geräte in meinem Heimnetz sind im Internet mit dieser Adresse unterwegs. Kommen Datenpakete zurück, erkennt der Router, für welches Gerät sie bestimmt sind und setzt die passende „private“ IP-Adresse ein. Dieses Verfahren, aus der Not der knappen Adressen geboren, nennt sich NAT (Network-Address-Translation).

2 Standarddienste

Ganz kurz will ich auf Standarddienste für das Internet und mein Heimnetz eingehen.

Ein Dienst oder Service ist nichts anderes als ein Programm auf der Serverseite, das für eine spezifische Aufgabenstellung gemacht ist. Weiter unten, im Kapitel „Steckdosen“, wird der technische Hintergrund erläutert. Hier geht es mehr um die logische Ebene. Eine sogenannte Port- oder Service-Nummer steuert das: Will ich eine WEB-Seite aufrufen, muss ich den richtigen Dienst auf dem Server ansprechen. Das mache ich, indem ich die Port-Nummer 80 für HTTP oder, wenn von der Programmiersprache unterstützt, den Servicenamen „http“ wähle. Diese Nummern werden in RFCs[157] definiert und bilden die allgemein bekannten Services. Schreibe ich meinen eigenen Service, benutzte ich eine freie Port-Nummer.

Über eine Netzwerk-Verbindung gehen immer nur Roh-Daten, Byte für Byte. Die Interpretation dieser Daten obliegt der Anwendungschicht. Daher ist es so wichtig, dass der Client den *richtigen* Service anspricht.

2.1 DHCP

Zu Hause brauche ich DHCP (dynamisches Host Konfigurations Protokoll). Praktisch jeder Router im Heimnetz, der die Verbindung zum Internet bildet, stellt es bereit. Jedes Gerät, ob Laptop, PC, Tablett, Smartphone, das sich im Heimnetz befindet, erhält über diesen Dienst eine IP-Adresse und meist auch einen Namen. Daneben werden noch Informationen über den Router, das Netzwerk usw., die für den Betrieb erforderlich sind, übermittelt.

Geräte, die über WLAN ins Heimnetz wollen, müssen noch eine vorgeschaltete Hürde überwinden. Bei einer Kabelverbindung ist klar, wer am anderen Ende sitzt. Bei Funk könnte ich mich leicht unabsichtlich beim Nachbarn anmelden. Um das zu verhindern, hat jedes WLAN einen Namen (SSID, Service Set Identifier) und ein Passwort. Bis auf Ausnahmefälle, wie öffentliche Hotspots, wird der WLAN Datenverkehr verschlüsselt und der Zugang auf Leute beschränkt, die das Passwort

156 Beginnt eine IP-Adresse z.B. mit 192.168 weist sie sich als „privat“ aus.
157 Request for Comment (Bitte um Kommentare) wird die Internet Dokumentation genannt.

des Access-Points[158] wissen. Habe ich die Verbindung zum WLAN erfolgreich aufgebaut, wird wieder mit DHCP die Konfiguration automatisch durchgeführt.

Unser kleiner Mikrocontroller hat einen DHCP Client im Netzwerk Modul eingebaut. Auch er erhält also eine passende IP-Adresse usw. nach dem Aufbau der Netzwerkverbindung.

2.2 DNS

Die Namen- zu IP-Adressenauflösung wurde schon angesprochen. Sie ist uralt und – unverschlüsselt! Das schafft nette Anwendungsgebiete für Hacker.

Es gibt, wenn man so will leider, gleich mehrere verschlüsselnde Ersatzdienste. Die Frage ist zum einen, was der Router unterstützt. Daneben gibt es aber auch Verfahren, bei denen der Browser die verschlüsselte DNS Anfrage abwickelt. Die Details würden den Rahmen hier sprengen.

Auch hier: Der Pi Pico bzw. Micropython verfügt über einen DNS Client, kann also Namen in IP-Adressen auflösen.

2.3 NTP

Über diesen Dienst haben wir noch gar nicht gesprochen. NTP steht für Network Time Protocol und stellt die Uhrzeit und das Datum bereit.

Wenn wir unseren Pi Pico einschalten, kann er von da ab die verflossenen Zeit messen. Die aber ist eben relativ zu seinem Startzeitpunkt. Außerdem gibt es zwei Arten, Zeiten zu erfassen. Den Zeitstempel, eine fortlaufend größer werdende Zahl, meist in Sekunden mit Nachkommastellen für hundertstel oder noch feinere Auflösungen, und die uns geläufigere Uhrzeit, mit einem Datums- und (Tages-)Zeitanteil. Für letzteres benützt Micropython eine Datenstruktur[159], die Jahr, Monat, Tag, Stunde, Minute, Sekunde, Tag der Woche, Tag des Jahres umfasst. Um darauf zuzugreifen gibt es die Methode „time.gmtime“.

Beispiel:

```
import wconn # wir verbinden uns mit dem WLAN, siehe unten
import time
print(time.gmtime(), time.time())
import ntptime
ntptime.settime()
print(time.gmtime(), time.time())
# Ausgabe:
# (2021, 1, 1, 0, 0, 27, 4, 1) 1609459227
# (2024, 7, 12, 6, 55, 36, 4, 194) 1720767336 # Zeitpunkt des Aufrufs
```

Wir erhalten die exakte Uhrzeit und das Datum, aber wir müssen „ntptime.settime“ nicht ständig aufrufen, weil nach dem Setzen der Uhrzeit Micropython intern diese weiter aktualisiert. Da wie bei einer Quarzuhr die Abweichung von der genauen Zeit nur wenige Sekunden pro Tag beträgt, reicht es, alle ca. 24 Stunden die Uhr aus dem Netz nachzustellen.

Wozu braucht ein Mikrocontroller die genaue Uhrzeit? Das hängt von der Anwendung ab. Einige Möglichkeiten wurden schon aufgeführt: Daten können mit einem absoluten Zeitstempel versehen

158 Der Access-Point ist in vielen Fällen im Router mit eingebaut.

159 Ein Hardwarebaustein genannt RTC (Real Time Clock) kann das je nach Mikrocontroller unterstützen. Oft ist er mit einer Batterie gekoppelt, so dass die Uhr auch im ausgeschalteten Zustand weiterlaufen kann.

werden, um nach einem Netzwerkausfall die gespeicherten Daten extern verarbeiten zu können, auch Log-Daten[160] profitieren davon.

Ein weiterer zu berücksichtigender Aspekt ist die Zeitzone. Micropython[161] liefert immer eine sogenannte UTC (Universal Time Coordinated), ältere Semester kennen das noch als Greenwich Mean Time. Unsere Zeitzone ist 1 Stunde voraus, im Sommer 2 Stunden. Die „6, 55, 36“ von oben entsprechen daher 8:55 Uhr.

Wer sich wundert, dass bei Benutzung von Thonny der Mikrocontroller ohne expliziten Aufruf von „ntptime.settime“ die genaue Uhrzeit hat, dem sei verraten, dass Thonny die Uhr stellt, wenn es die Verbindung zum Mikrocontroller herstellt.

2.4 Weitere Dienste

Ein paar Services will ich hier noch kurz anreißen. Auf Telnet und VPN wird weiter unten noch eingegangen. Zu erwähnen wäre noch „Tor“, ein IP-Adressen-Verschleierungsdienst. Wozu braucht man denn so etwas? Nun, das Internet gibt es weltweit, aber Regierungen und Konzerne schränken den freien Zugriff auf bestimmte Inhalte schon mal ein. Ob berechtigt zur Verhinderung von Straftaten oder aus politischen Gründen soll hier nicht diskutiert werden. Um die Rückverfolgbarkeit anhand der IP-Adresse zu verhindern und Zugriff auf durch Netzsperren normal nicht erreichbare WEB-Seiten oder Dienste zu erlauben, geht man mit dem Tor-Dienst in das Tor-Netzwerk, das aus tausenden Servern weltweit besteht, die jeder die IP-Adresse meines Datenpaketes verändern. Irgendwann verlässt mein Paket dann dieses Netzwerk zum gewünschten Ziel.

Um sich sicher auf anderen Rechnern einloggen zu können, gibt es „SSH“ (Secure Shell). Im Linux-Umfeld ist es inzwischen der Standard. Aber auch Windows verfügt über einen SSH-Client. SSH verschlüsselt nicht nur die Daten, sondern erlaubt einen Login mit einem asymmetrischen Schlüssel anstatt eines Passworts. Das gilt als sehr sicher.

Mit dem RDP Protokoll (Remote Desktop) von Microsoft sind grafische Benutzersessions möglich. Leider ist der Sicherheitsstandard unzureichend. Im Workshop nutze ich RDP, um von jedem Teilnehmer Laptop oder den bibliothekseigenen Mac Books eine Anmeldung auf dem Raspberry Pi Rechner, an dem die Pi Picos über USB angeschlossen sind, zu erlauben.

3 Die Säulen des Internet

Jetzt haben wir viele technische Details kennengelernt. Was fehlt, ist der Blick von oben, um zu verstehen, wie das Internet funktioniert. Wenn wir uns das gesamte Internet wie das Dach eines griechischen Tempels vorstellen, dann wird dieses Dach durch zahlreiche Säulen getragen. Einige Säulen sind unverzichtbar, damit das Dach nicht einstürzt, andere sind eher zur Zierde da.

Eine der Hauptsäulen besteht aus einer Festlegung: IP-Adressen sind weltweit einmalig; man nennt das auch Eineindeutigkeit. Sie ähneln damit Telefonnummern inklusive der Länder- und Ortsvorwahl. Die Ausnahme der „privaten“ IP-Adressen habe ich schon erwähnt. Damit wird ein

160 Als Log-Daten werden solche bezeichnet, die Informationen über das Laufzeitverhalten eines Programms liefern, z.B. über Fehler. Da hilft eine genaue zeitliche Einordnung sehr.

161 Das große Python kann lokale Zeiten korrekt umrechnen.

Trick angewendet, um den Mangel an Adressen zu umschiffen. Theoretisch sind bei IPv4 über 4 Milliarden Adressen möglich; aber heute, wo jedes Smartphone und fast jeder Kühlschrank eine Adresse braucht, reicht das bei weitem nicht mehr.[162]

Die Provider, also die Firmen, die Endkunden ans Internet anbinden, bedienen sich aus einem offiziellen Pool an Adressen[163], was die Eineindeutigkeit garantiert. Sie weisen einem Kunden bzw. seinem Router eine davon zu.[164] Diese Adresse ändert sich meist nach einer bestimmten Zeit. Das hilft dem Provider, die knappe Ressource effektiver nutzen zu können, da eine Adresse, die gerade nicht verwendet wird, anderweitig vergeben werden kann.

Für den Kunden, der sein Heimnetzwerk auch von außen erreichen möchte, stellt das eine große Hürde da. Wie so oft, gibt es aber auch dafür eine Lösung: Dyn-DNS. Anbieter stellen damit einen DNS Namen für die wechselnden IP-Adressen der Heimnetzwerke bereit.

Während die eine Säule des Internet also die Eineindeutigkeit der Adressen ist, sind die Router bei den Providern und Firmen, die sich auf das Routing spezialisiert haben, die nächste Säule. Sie kennen sich untereinander in bestimmten Regionen, das können Länder oder Kontinente sein, und wissen, wohin sie ein Paket weiterleiteten müssen, entweder direkt zum Ziel oder in eine andere Region, wo dann wiederum das gleiche Spiel abläuft.[165]

Die Router verfügen über eigene Software, Router-Protokolle genannt, mit deren Hilfe sie die notwendigen Daten austauschen. Geht das schief, was in der Vergangenheit schon einige Male vorkam, steht das Internet, zumindest in einigen Regionen, still.

Diese Router-Protokolle optimieren ständig die Wege, die ein Paket von A nach B zu durchlaufen hat. Dabei spielt nicht die räumliche Entfernung eine Rolle, sondern nur die Geschwindigkeit der Daten. Daher kann es vorkommen, das ein Paket aus Velbert nach Düsseldorf über Frankfurt geht. Sogar länderübergreifende Wege sind möglich.

Als Anwender merke ich von alledem gar nichts. Das liegt auch am erwähnten DNS, womit die Zieladresse für einen Namen ermittelt wird. Die Namen sind strukturiert von hinten nach vorne, getrennt durch Punkte, aufgebaut. Eine Adresse wie „www.amazon.de“ wird aufgelöst als „de“, Länderkennung für Deutschland, „amazon“, der Name der Firma und „www“, das „Word Wide Web“, das für das HTTP bzw. heute fast immer das HTTPS[166] Protokoll steht. Ein „ftp.uni-hagen.de“ beträfe die Universität Hagen in Deutschland mit dem Dienst FTP, einem Datei-Übertragungsdienst.

Das DNS bildet wieder eine Säule. Zusammen mit den Routern und ihrer Verbindung untereinander und ohne die Eineindeutigkeit der Adressen würde das Internet nicht funktionieren. Neben diesen Hauptsäulen gibt es noch zahlreiche andere Säulen und Stützpfeiler, die zum reibungslosen Betrieb dienen. Sie alle aufführen zu wollen, bräuchte es ein eigenes Buch.

---

162 Die Verbreitung von IPv6, das diese Einschränkung nicht hat, ist laut einer Google Statistik weltweit noch unter 50%.

163 „Bedienen“ müsste eigentlich in dicke Anführungszeichen, denn sie zahlen dafür sehr viel Geld.

164 Das stammt noch aus der Zeit, als Boris Becker „ich bin drin“ jubelte und ein Rechner zu Hause das höchste der Gefühle war.

165 Anders als bei Telefonnummern bzw. den Vorwahlen sind die IP-Adressbereiche nicht fest Staaten zugeordnet, sondern die Verteilung der Adressbereiche erfolgte historisch gewachsen dem Zufall.

166 Das „S“ darin steht für Sicherheit, weil die Verbindung zwischen Client und Server verschlüsselt wird.

4 Der Heim-Router

Ein kleiner Exkurs zum Begriff Router. Umgangssprachlich wird damit das Gerät bezeichnet, das für den Endkunden den Zugang zum Internet einerseits und das lokale Netzwerk über Kabel und WLAN andererseits bereitstellt. Ein Router bezeichnet aber auch ein Gerät, dass einzig und allein Datenpakete auf den richtigen Weg lotst. Sie stehen z.B. beim Provider.

Unser Heimrouter vereinigt ein Modem[167], das die Hardwareschnittstelle zum Internet über DSL, Kabel oder Glasfaser ist, mit einem WLAN Access-Point, einem Switch, Routingfunktion, NAT, DNS, DHCP, NTP usw. Ein kleiner Tausendsassa!

Die wichtigste Funktion Aus *technischer* Sicht stellt das NAT (Network Address Translation) dar; es konvertiert die privaten internen IP-Adressen in die eine öffentliche Adresse[168], mit der dann die Verbindung zu Servern im Internet aufgebaut wird. Die zurückkommenden Pakete werden dann wieder den internen IP-Adressen zugeordnet. NAT ist keine ideale Lösung; insbesondere der Zugriff von außen ist nur über „Sonderlocken“ (Port-Forwarding[169]) möglich.

Aus Kundensicht ist die Firewall vielleicht der wichtigste Aspekt: Eine Firewall lässt nur bestimmten Regeln entsprechende Pakete durch. Vom Internet kommende Verbindungsanfragen werden (fast) komplett geblockt. Um aber auf mein Heimnetz selbst zugreifen zu können, kann ich kleine „Löcher“ hinein bohren, die mir z.B. einen VPN Zugang ermöglichen. VPN steht für virtuelles, privates Netzwerk. Technisch wird ein verschlüsselter Kanal erstellt, durch den alle Pakete hindurch müssen. Damit stellt VPN eine Punkt zu Punkt Verbindung zwischen Rechnern oder Netzwerken her. Befinde ich mich mit meinem Smartphone in einem öffentlichen, unverschlüsselten WLAN, könnte ohne VPN jeder in diesem WLAN z.B. ein Passwort, das ich eingebe, mitlesen :-((

Innerhalb des internen, privaten Netzwerks findet **kein** Routing statt. Geräte, die zu einem Netzwerk gehören, finden sich anhand des ARP-Protokolls auf der Basis von MAC-Adressen. Das sei hier aber nur kurz erwähnt.

Ganz wichtig: Das WLAN und der Kabelanschluss eines Routers gehören zum gleichen Netz! Das wird intern durch entsprechende Softwarefunktionen bereitgestellt. Moderne Router haben einen Switch eingebaut, an den mehrere kabelgebundene Geräte angeschlossen werden können. Bei mir reichen die vier Anschlüsse meines Routers nicht aus; deshalb habe ich noch mehrere weitere Switche kaskadiert. Auch ein Switch ändert nichts am Netzwerk, er leitet nur die Datenpakete physisch weiter.

Viele Router stellen auch ein internes DNS bereit, so dass ich auch die Geräte in meinem Heim-Netzwerk mit Namen ansprechen kann. Die Geräte wiederum können beim Anmelden ans Netzwerk ihren gewünschte Namen mitgeben. Das kann auch der Pi Pico!

167 Modem ist ein Kunstwort, das sich aus Modulator-Demodulator zusammensetzt.
168 Wenn mein Provider das unterstützt, habe ich heutzutage oft noch eine zweite IPv6 Adresse. Eigentlich brauchte ich dann kein NAT mehr, denn diese Adresse ist schon weltweit eindeutig.
169 Dazu wird ein festgelegter Port der öffentlichen IP-Adresse, zu dem von außen ein Verbindungsaufbau erfolgt, auf eine interne IP-Adresse und einen wählbaren Port weitergeleitet.

5 Ein Exkurs: IPv6

Wer einen aktuellen Heim-Router und technisch modernen Provider hat, kann heutzutage auch IPv6 aktivieren. Wirklich notwendig ist das zwar noch nicht, denn IPv4 wird voraussichtlich erst in vielen, vielen Jahren abgeschaltet, aber es schadet nicht, sich schon mal daran zu gewöhnen.

Im Gegensatz zum „tunneln“ von IPv6 durch IPv4 zeigt der Parallelbetrieb nach meiner Kenntnis keine Probleme. Öffne ich im Browser eine Adresse, die auch etwas vereinfacht Domain oder URL genannt wird, kann sie, ohne dass ich davon etwas merke, in eine IPv6 IP-Adresse aufgelöst werden.

Weiter oben habe ich schon erwähnt, dass IPv6 Adressen wesentlich länger als IPv4 Adressen sind. Würde man sie in ähnlicher Notation wie IPv4 angeben, müsste man 16 Zahlengruppen angeben. Daher hat man sich darauf geeinigt, IPv6 Adressen *hexadezimal* in 8 Grüppchen zu schreiben. Dabei umfasst die Zahl einer Gruppe den Zahlenbereich von 0 bis 65535, der sich hexadezimal mit maximal 4 Zeichen darstellen lässt: Die Ziffern 0 – 9 sind identisch zur dezimalen Schreibweise, dann geht es weiter mit „A“ – „F“, was der 10 - 15 entspricht. „FFFF“ ist die höchste Zahl (65535). Da ich keine führenden Nullen schreiben muss und eine Folge von Nullen einmalig durch zwei aufeinander folgende Doppelpunkte (::) abgekürzt werden darf, sind IPv6 Adressen trotz ihrer Länge noch einigermaßen übersichtlich.

IPv6 Adressen sind aber nicht nur einfach länger, sondern auch viele Eigenschaften und Verhaltensweisen sind neu. Gab es bei IPv4 nur zwei Typen, öffentlich und privat, gibt es nun fünf Gültigkeitsbereiche. Die Zuweisung von IPv6 Adressen in einem Netzwerk kann automatisch, ohne DHCP, erfolgen. Und ganz wichtig: Eine Netzwerkschnittstelle kann nun mehr als eine Adresse haben.

Die Micropython Unterstützung für IPv6 ist – noch – rudimentär und stark vom jeweiligen unterstützten Mikrocontroller abhängig. Zu unserem Pi Pico habe ich keine Dokumentation gefunden. Eigene Experimente führten nur zu spaßigen Fehlermeldungen (OSError: -6). Um IPv6 als Zieladresse auf dem Pi Pico zu nutzen, muss er selbst eine IPv6 Adresse haben. Also erscheint es mir sinnvoll zu sein, erst mal bei IPv4 zu bleiben.[170]

6 Das „W“ hinter Pi Pico

Den Pi Pico gibt es in zwei Ausführungen, die sich im Preis nicht wesentlich unterscheiden. Der Eine hat noch ein „W“ hinter der Bezeichnung. Das steht für „hat WLAN“, d.h. der Mikrocontroller kann sich darüber in ein bestehendes WLAN einklinken, der sogenannte Stations-Modus, oder er kann selbst ein WLAN aufspannen, der Access-Point-Modus[171], abgekürzt AP-Modus. Je nachdem, was für mein Programm günstiger ist, kann ich mich für eine der beiden Möglichkeiten entscheiden.

Wenn ich mich mit einem bestehenden WLAN verbunden habe, kann ich danach eine TCP/IP Verbindungen aufbauen. Mit dem Wissen um IP-Adresse oder Namen (ein DNS Client ist in Micropython schon eingebaut) ist eine Verbindung zu einem PC, Laptop oder – in vielen Fällen – zu einem anderen Pi Pico möglich.

170 Mit der 2025 erschienenen Version 1.25 scheinen jetzt auch IPv6 Adressen für den Pi Pico möglich zu sein. Ausprobiert habe ich das noch nicht.
171 Ich habe keine deutsche Bezeichnung dafür gefunden.

Was, wenn ich einen Access-Point aufgezogen habe? Erstens steht nun kein DNS zur Verfügung, für eine Verbindung bräuchte ich eine IP-Adresse. Aber mit wem soll ich mich verbinden? Da ist ja niemand![172] Erst, wenn ein anderes Gerät sich an diesem WLAN Access-Point anmeldet, d.h. ein „Luftkabel“ gelegt wird, habe ich einen Kommunikationspartner. Obwohl möglich, wird normalerweise aber dieser Nutzer des Access-Point die Verbindung initiieren und nicht andersherum. Doch auch der braucht eine IP-Adresse dazu. Die legt der Programmierer fest, wenn er auf dem Pi Pico den Access-Point-Modus konfiguriert. Jetzt taucht die nächste Schwierigkeit auf: Auf dem Access-Point muss „jemand“ den Verbindungswunsch erkennen. Dazu ist es erforderlich, dass eine Listener (Zuhörer) genannte Funktion in einem Programm aufgerufen wird.

Obwohl das alles kompliziert klingt und zum Teil auch ist, lässt es sich in Micropython mit wenigen Codezeilen realisieren. Das werden wir gleich sehen. Doch zuerst noch: Wann ergibt es Sinn, auf einem Pi Pico einen Access-Point bereitzustellen? Die Antwort ist einfach: Immer wenn ich, egal wo, auf ihn zugreifen möchte. Denn unterwegs steht ja mein Heimnetzwerk nicht zur Verfügung.

7 Gibt es es hier WLAN?

Unser Pi Pico bzw. Micropython erlauben uns auch festzustellen, ob und welche Access-Points verfügbar sind. Dafür gibt es die Methode „scan“. Mit folgenden wenigen Zeile kann ich alle zur Zeit verfügbaren WLAN anzeigen:

```
nic = network.WLAN(network.STA_IF)
nic.active(True) # schaltet das WLAN Hardware-Modul ein
nic.scan()
# ergibt bei mir:
[(b'biblio', b'\xdc\xa62r\n\x82', 11, -38, 3, 8), (b'devolo-388',
b'\xb6\xbe\xf4\x56\x92\x1d', 1, -70, 5, 2), (b'GAST', b'\xde9oUZ\xcc', 8, -52,
5, 2), (b'GAST', b'\xfa\xdfp\xc3\x04l', 8, -86, 5, 2), (b'MW', b'\xec9oUZ\xcc',
8, -52, 5, 3), (b'MW', b'\xe7\xdfp\xc3\x04l', 8, -85, 5, 2), (b'FamFiber',
b'<7\x11\x0c\x16\xc7', 11, -72, 5, 3), (b'Home2018',
b'\x02\xb4\xde\x94\x0e\x95', 9, -82, 5, 2)]
```

Scan liefert eine Liste mit Tupeln[173] zurück. Das jeweils erste Element stellt den SSID Namen dar, es folgen die Hardware Kennung (BSSID), der Kanal (Funkfrequenz), die Signalstärke. Was die letzten beiden Werte besagen, habe ich nicht herausgefunden[174].

Mit diesen Informationen, besonders der Signalstärke, kann ich Probleme mit dem WLAN Zugang erkennen. Die Signalstärke wird als Zahl angegeben, die hier immer negativ ist. Der Wert steht für dBm, eine in der Funktechnik gebräuchliche logarithmischer Einheit; je kleiner der Wert, desto schlechter der Empfang. Also -70 ist besser als -82. Gibt es mehrere WLAN, mit denen ich mich verbinden könnte, suche ich mir das mit der größten Signalstärke heraus.

8 Eine WLAN Verbindung aufbauen

Der erste Schritt beim Netzwerken ist immer das Verlegen eines Kabels, auch wenn es in unserem Fall ein „Luftkabel“ ist.

---

172 Ein Netzwerkgerät kann immer eine Verbindung zu sich selbst aufbauen, der Name ist „localhost“ und die IP-Adresse 127.0.0.1.

173 Ein Tupel ist eine Liste, deren Elemente unveränderbar sind.

174 In der Dokumentation zu Micropython wird gesagt, dass sie für die Sicherheitsstufe und das „Hidden“-Kennzeichen stünden. Allerdings kann ich die Werte nicht sinnvoll zuordnen.

Auf dem Pi Pico mit Micropython sieht das so aus:

```
import network
import time

nic = network.WLAN(network.STA_IF)
nic.active(True)
nic.disconnect() # Wichtig! Siehe Text unten
ssid = "GAST"
pw = "123456"
nic.connect(ssid, pw)
while nic.status() == network.STAT_CONNECTING:
    print(".", end="")
    time.sleep(0.1)
print()
print(nic.ifconfig())
# Ausgabe:
# ................................
# ('192.168.179.2', '255.255.255.0', '192.168.179.1', '192.168.179.1')
```

Bei einer Wartezeit von 0,1 Sekunde und 32 Punkten dauerte bei mir der Verbindungsaufbau ca. 3 Sekunden. Die vier Elemente in der Ausgabe der Methode "ifconfig" bedeuten der Reihe nach:

- Die via DHCP zugewiesene IP-Adresse, hier die 192.168.179.2.
- Die Netzwerkmaske[175] 255.255.255.0 legt den Bereich gültiger IP-Adressen auf 192.168.179.1 - 192.168.179.255 fest.
- Die IP-Adresse 192.168.179.1 sagt, wer der Router ist.
- Die letzte IP-Adresse, hier ebenfalls 192.168.179.1 ist die Adresse, die DNS Abfragen entgegennimmt.

Die einzelnen Schritte erkläre ich nun. Zuerst wird das Modul „network“ importiert. Es enthält alle notwendigen Klassen. Das Modul "time" brauche ich wieder für "sleep".

Jetzt wird es spannender: Ich speichere eine Instanz der Klasse WLAN mit dem Parameter „STA_IF“, dem Kennzeichen für den Stations-Modus. in der Variablen „nic“. Wenn ich nun mit

```
nic.active(True)
```

die WLAN Hardware einschalte, erhöht sich die Stromaufnahme des Pi Pico von 17 auf 51 mA.

Mit „nic.disconnect“ stelle ich sicher, dass eine eventuell noch bestehende Verbindung getrennt wird. Intern merkt sich nämlich Micropython eine Verbindung und versucht sie wiederherzustellen. Habe ich nun eine andere SSID gewählt, würde sie nicht berücksichtigt.

Ich lege die SSID und das Passwort fest, beides muss mit dem übereinstimmen, was auf dem Access-Point / Router konfiguriert wurde.

Die Methode „connect“ initiiert den Verbindungsaufbau. In der Schleife warte ich, bis die Methode „status“ nicht mehr „STAT_CONNECTING“, eine Konstante im Modul „network“, liefert. Nach dem Verlassen der Schleife ist der Pi Pico mit dem WLAN verbunden (oder ein Fehler aufgetreten). Ich kann ihn nun von meinem PC aus anpingen.

Das funktioniert wunderbar, aber nach meiner Erfahrung meist dann nicht, wenn man vorher ein anderes WLAN erreicht hat. Eigentlich nicht schlimm, aber – es gibt keine Fehlermeldung. Daher

175 Von einer Maske sprechen wir immer dann, wenn die Bits der Zahl durch bitweises UND mit einer anderen Zahl verglichen werden. Siehe Kapitel "Logisch?"

und damit man nun nicht wieder den obigen Programmtext eintippen muss, machen wir daraus ein Modul:

```
# wconn.py
import network
import time

def status(status: int):
    """"Hilfsfunktion Status Textausgabe"""
    return {
        0: "STAT_IDLE",
        1: "STAT_CONNECTING",
        -3: "STAT_WRONG_PASSWORD",
        -2: "STAT_NO_AP_FOUND",
        -1: "STAT_CONNECT_FAIL",
        3: "STAT_GOT_IP",
        2: "STAT_NOIP"}[status]
    }

def connect(ssid: str, pw: str, hostname: str):
 n = 0
    network.country("DE")
    network.hostname(hostname) #  der DNS Name des Pi Pico
    nic = network.WLAN(network.STA_IF)
    nic.active(True)
    nic.disconnect()
    nic.connect(ssid, pw)
    while nic.status() == network.STAT_CONNECTING:
        print(".", end="")
        #print(n) # zum debuggen
        time.sleep(0.1)
        n += 1
        if n > 150:
            break #  Schleife nach 15 Sekunden verlassen
    # Lösung für Workshop, weil DHCP nicht funktioniert
    if stat == 2: # das Beziehen der IP-Adresse hat nicht geklappt
        print("\nStatische Netzwerk Konfiguration")
        from boot import ICH_BIN
        # Statische Basis Konfiguration; in den folgenden Zeilen wird die
        # IP-Adresse um die in ICH_BIN gespeicherte Zahl erhöht
        static = ["10.42.0.40", "255.255.255.0", "10.42.0.1", "10.42.0.1"]
        ipl = static[0].split(".")
        ipl[3] = str(int(ipl[3]) + ICH_BIN)
        static[0] = ".".join(ipl)
        nic.ifconfig(static) # statische IP und Netzwerk Konfiguration
    print("\nStatus:", status(nic.status()), nic.ifconfig())
    if nic.isconnected():
       return nic # wird zu True ausgewertet
    else:
       nic.deinit()
       return None


n = 0
nic = None
ssid = "GAST" # anpassen!!!
pw = "123456" # anpassen!!!
while not (nic := connect(ssid, pw, "pico1")):
    n += 1
```

```
    if n > 5:
        raise Exception(f"Kann mich nicht mit WLAN {ssid} verbinden!")
```

Das Ganze ist ein bisschen aufgehübscht. Die Logik dahinter ist die, dass ich solange in der Schleife warte, bis die Methode „nic.status“ nicht mehr STAT_CONNECTING liefert oder der Zähler „n“ einen höheren Wert hat, als vorgegeben (hier 150, was bei Zehntelsekunden warten 15 Sekunden bedeutet). Dann überprüfe ich mit der Abfrage „if nic.isconnected():“, ob die Anmeldung erfolgreich war. Ich gebe in der Variablen „nic“ die Instanz der WLAN-Verbindung zurück. Im „else“ Zweig ist hier neu „nic.deinit()“, was die WLAN Hardware wieder abschaltet. Dass das geklappt hat, sieht man[176] an der Verminderung des Stromverbrauchs auf die 17 mA vom Anfang. Die Hilfsfunktion „status“ gibt statt kryptischer Zahlen den WLAN-Status als Text aus. Bei mir trat im Test sporadisch ein Return-Code „2“ auf, der auf der Micropython Web-Seite nicht dokumentiert ist. Auf einer anderen Seite[177] fand ich eine einleuchtende Erklärung: Die 2 bedeutet, dass die WLAN Verbindung erfolgreich war, aber keine IP-Adresse erhalten wurde. Für diese Nummer wird „STAT_NOIP“ ausgegeben. In diesem Fall hilft es, eine IP-Adresse statisch[178] vorzugeben. Damit funktionierte es, aber feste IP-Adressen sind nicht schön :-((

In den Zeilen ganz unten wird der Verbindungsaufbau nun 5 mal versucht.

Brauchen wir nun WLAN, tippen wir einfach „import wconn“ ein oder packen diese Zeile in unser Programm – fertig!

9 Steckdosen

Bisher habe ich recht allgemein von Verbindungsaufbau und vom „Listener“ gesprochen. Bevor wir ans Programmieren gehen können, muss ich hier noch einige Details liefern.

Das TCP/IP Modell sieht zwei unterschiedliche Arten[179] der Kommunikation vor: Eines, bei dem eine quasi feste Verbindung zwischen zwei Teilnehmern hergestellt wird (TCP, Transmission Control Protocol) und eines, bei dem die Datenpakete an einen oder mehrere Partner herausgeschickt werden, ohne sich darum zu kümmern, ob sie auch ankommen (UDP, User Datagram Protocol).

Obwohl Letzteres mit Micropython und unserem Pi Pico auch geht, werde ich hier nicht näher darauf eingehen. Wir konzentrieren uns auf TCP. Mit diesem Protokoll arbeiten auch die meisten der Services wie HTTP oder Telnet.

Aber wie genau geht das? Was eine IP-Adresse darstellt, ist, denke ich, schon ziemlich klar: Ein Gerät, das über ein Netzwerk erreichbar ist. Gäbe es nur diese Adresse, könnte zu einem Gerät nur exakt eine Verbindung aufgebaut werden. Daher gibt es zusätzlich Port-Nummern, genau 65536, eine 16 Bit Zahl. Telnet hat die Port-Nummer 23, HTTP die 80. Will ich von meinem Pi Pico also eine Verbindung zu einem Telnet-Server[180] herstellen, muss ich die IP-Adresse *und* die Port-Nummer angeben.

---

176 Wenn man ein USB-Messgerät hat oder mit einem Strommessgerät in der Versorgungsleitung misst.
177 https://www.elektronik-kompendium.de/sites/raspberry-pi/2708121.htm
178 Das geht ganz einfach, indem man „ifconfig(["10.42.0.40", "255.255.255.0", "10.42.0.1", "10.42.0.1"])“ aufruft.
179 Eigentlich sogar noch mehr, aber Broadcast, Multicast usw. lasse ich hier aus.
180 Wir erinnern uns: Der Server nimmt die Verbindungsanfrage entgegen; er muss die IP-Adresse des Client nicht wissen, erfährt sie aber beim Verbindungsaufbau.

Programmiertechnisch nennt man so etwas „Socket“ (englisch Steckdose). Ein Socket umfasst beim TCP Verbindungsaufbau die Ziel-IP-Adresse und die Ziel-Port-Nummer. Beim Verbindungsaufbau wird außerdem die Quell-IP-Adresse und eine zufällig generierte, lokale Port-Nummer übergeben.

		Quelle	Ziel
Erste Verbindung	IP-Adresse	192.168.0.27	10.42.0.1
Erste Verbindung	Port-Nummer	49643	23
Zweite Verbindung	IP-Adresse	192.168.0.27	10.42.0.1
Zweite Verbindung	Port-Nummer	4964**9**	23

Jetzt habe ich eine eineindeutige Verbindung zwischen zwei Geräten und einem Service. Die gesamte Kommunikation erfolgt anhand dieser 4 Angaben. Jede weitere Verbindung erzeugt wieder eine eineindeutige Zahlenkonstellation. Auch wenn ich mich vom gleichen Gerät aus an den gleichen Service auf dem gleichen Ziel-Gerät anmelde, erhalte ich eine neu Quell-Port-Nummer. Damit wissen beide Gegenstellen genau, wer mit wem Datenpakete austauschen darf.

Wie viele eingehende Verbindungen der Service annimmt, hat sein Programmierer festgelegt. Der Socket-Mechanismus ist jedenfalls so beschaffen, dass ein Service nach Annahme einer Verbindung wieder in die Lage versetzt werden kann, eine weitere Verbindung aufzubauen.

10 Licht an – Licht aus!

Nach all dem theoretischen Vorgeplänkel geht es nun zur Sache! Wir entwickeln ein Programm, mit dem wir die LED des Pi Pico per Funk ein- und ausschalten.

Eigentlich könnten wir das mit zwei Pi Picos machen, aber ich zeige erst mal die Variante, dass ein PC Programm – natürlich in Python – mit einem Programm auf dem Mikrocontroller spricht.

Die Rahmenbedingungen: Wir haben einen WLAN Access-Point, der vom PC und vom Pi Pico erreicht werden kann. Wenn mein PC kein WLAN hat: Ist das schlimm? Nein, denn, wie wir jetzt über Netzwerke wissen, werden die Datenpakete automatisch weitergeleitet. In einem Heimnetzwerk, das sowohl über ein WLAN als auch über Netzwerkkabel verfügt, braucht es dazu nicht mal ein Routing[181]. Auch wenn die WLAN-Geräte und die, die mit Kabel angebunden sind, physisch voneinander getrennt sind, teilen sie sich ein logisches Netzwerk[182].

Um gleich schneller experimentieren zu können, verwenden wir die oben gezeigte Funktion. Dazu speichern wir Code in einer Datei des Flash-Dateisystem mit dem Namen wconn.py[183].

Wenn wir jetzt auf dem Pi Pico WLAN benötigen, tippen wir „import wconn“ ein. Fertig!

Nun zwei Programme mit „socket“. Das Erste muss dem Pi Pico laufen, da wir hier eine LED leuchten lassen wollen. Es muss zuerst gestartet werden, da sonst das zweite Programm sofort mit

181 Aber schon den Heim-Router, da dieses Gerät eine Reihe von Hard- und Softwarefunktionen bereitstellt: WLAN Access-point, Modem (Anbindung an DSL, Glasfaser usw.), Routingsoftware, DHCP ...

182 Die IP-Adressen der über WLAN angebundenen Geräte gehören in ein, durch die Netzmaske definiertes Netzwerk. Die Pakete werden nur zwischen unterschiedlichen Medien (kabelgebunden, Funk) ausgetauscht. Der Access-Point hat einen Kabel- und einen Funkanschluss.

183 Mit Thonny brauchen wir nur die Datei auf dem lokalen Computer rechts anzuklicken und den Menüpunkt „Upload to /“ zu wählen.

einer Fehlermeldung aussteigt. Warum? Der „Server“ muss schon laufen, bevor sich ein „Client“ anmelden kann.

```
# serversocket_tst.py
import wconn
import socket
import time

# SERVER Programm
# Adresse aufbereiten, 0.0.0.0 steht für alle IP-Adressen des Hosts,
# AF_INET schränkt auf IPv4 ein
addr = socket.getaddrinfo("0.0.0.0", 8004, socket.AF_INET)[0][-1]
# Den Server instantiieren aus der Klasse socket
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(addr) # den erzeugten socket an die Adresse binden
serversocket.listen(1) # auf – genau eine - Kontaktanfrage von außen lauschen
                       # und warten (unendlich lange!)
(clientsocket, address) = serversocket.accept() # jemand hat angeklopft
print(clientsocket,address) # in address steht wer
led = machine.Pin("LED") # die interne Led des Pi Pico W

while True:
    r = clientsocket.recv(1) # genau 1 Zeichen lesen
    print(r)
    if r == b"0": # r ist im Byte Format
        led.off()
    elif r == b"1":
        led.on()
    elif r == b"2":
        clientsocket.close()
        serversocket.close()
        break
    time.sleep(1)
```

Das zweite Programm kann auf dem PC oder dem Pi Pico laufen. Wenn man also nur einen Pi Pico sein Eigen nennt, kann man das trotzdem ausprobieren.

```
# clientsocket_tst.py
import socket
# import wconn # wenn es auf einem Pi Pico läuft entkommentieren
# CLIENT Programm
# Port Nummer muss übereinstimmen, hier z.B. 8004
addr = socket.getaddrinfo("<IP des Pi Pico>", 8004, socket.AF_INET)[0][-1]
clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect((addr))

while True:
    r = input("Bitte 0 oder 1 eingeben, 2 zum Abbruch:") # Es wird von der
                                                         # PC Tastatur gelesen
    r = r.encode('ascii') # siehe Erklärung unten
    print(r)
    clientsocket.send(r)
    if r == b"2":
        break
clientsocket.close()
```

Ich will ja niemanden frustrieren, aber in dieser Form werden wir wohl niemals Programme schreiben. Das hier nennt sich „low-level“ Programmierung, also auf unterster Ebene. Aber als Anschauungsobjekt ist es geeignet, weil es Dinge sichtbar macht, die bei der „high-level“ Programmierung unsichtbar werden.

Die Schreibweise b"0" gibt an, dass der Inhalt zwischen den Anführungszeichen kein String, sondern Bytes sind. Wir erinnern uns vielleicht, das Strings immer Unicode Text sind. Ein „Ö“ sind mehrere Bytes. Über Sockets können nur Bytes verschickt werden. Trotzdem können wir auch ein „Ö“ senden, müssen es vorher aber in Bytes umwandeln:

```
"Ö".encode("ascii")
```

Ich fange im ersten Programm mal von hinten an. Dort finden wir mehrfach ein „close()“. Wenn das fehlt oder, weil wir Strg-C drücken, das Programm diese Methoden nicht ausführt, scheitert höchstwahrscheinlich der nächste Versuch, das Programm laufen zu lassen. Wir sind ja hier nicht nur zu zweit, der Pi Pico und das Python Programm auf dem PC, sondern wir befinden uns in einem Netzwerk und in einem PC-Betriebssystem. Und da gelten für TCP Verbindungen bestimmte Regeln und Zeitabläufe! Ohne ein korrektes Schließen der Verbindung wird sie offen gehalten; es könnte ja auch gerade eine temporäre Netzwerkstörung vorliegen, bei deren Ende die Verbindung weiter genutzt werden soll.

Wer die Verbindung zuerst schließt, der Server oder der Client, ist eigentlich egal. In den Beispielprogrammen ist aber ein server-seitiges Schließen nicht einprogrammiert. Hier muss der Client, durch die Eingabe der Ziffer 2, die Programme beenden!

Geben wir im Client-Programm eine 1 ein und drücken Enter, geht auf dem Pi Pico die LED an. Bei 0 geht sie wieder aus. Alle anderen Zeichen, bis auf die 2, haben keine Auswirkungen. Das wird durch das „elif“ bewirkt.

Die „input“ Funktion, ließt von der Tastatur, bis die Enter-Taste gedrückt wird und speichert das Ergebnis hier in der Variablen „r“. Das oder die Zeichen werden gesendet, aber der Empfänger ließt immer nur ein Zeichen „recv(**1**)“ und wertet es aus. Das „time.sleep(1)“ macht das sichtbar, denn sonst ginge das viel zu schnell. Gebe ich „10101010101010101010“ ein, geht die LED im 2 - Sekundentakt zehnmal an-aus.

Die kryptischen Bezeichnungen „socket.AF_INET“ und „socket.SOCK_STREAM“ legen eine sogenannte Adress- und Protokollfamilie fest. Ersteres ist z.B. IPv4 oder IPv6, Letzteres unser TCP, also verbindungsorientiert[184]. In der verwendeten Python Version 3.11.2 und in Micropython ist das der Default, kann also weggelassen werden.

Kann ich mit diesen Programmen auch über das Internet die LED an- und ausschalten? Im Prinzip ja, aber wegen der privaten IP-Adresse muss ich im Router ein Port-Forwarding[185] einrichten. Dann muss ich noch die öffentliche IP-Adresse wissen, aber dann geht es![186]

Aber es gibt bessere Methoden.

11 Was ist eigentlich IoT?

Seit einiger Zeit geistert ein Begriff durch die Elektronikszene: IoT, das Internet of Things, das Internet der Dinge. Er hängt mit der immer stärkeren Verbreitung von netzwerkfähigen

184 Eine quasi feste Verbindung mit 2 Endpunkten. Die Verbindung wird auch aufrecht erhalten, wenn keine Daten fließen (in einem gewissen zeitlichen Rahmen).

185 Das stellt immer ein Sicherheitsrisiko dar. Wenn man nicht genau weiß, was man tut, sollte man die Finger davon lassen!

186 In der Stadtbibliothek ist der Raspberry Pi Server nicht mit dem Internet verbunden; hier geht das also nicht.

Mikrocontrollern zusammen. Kleine Sensoren können ohne großen Aufwand ihre Messdaten weitergeben, aus der Ferne können Motoren an-, Heizungen ausgeschaltet, Fenster geschlossen werden usw.

Wie wir schon gesehen haben, spielt das Client- Serverprinzip dabei eine große Rolle. Denn wenn Sensoren und Aktoren sich „unterhalten“ wollen, müsste jeder von jedem die IP-Adresse kennen. Was in einem Haushalt mit 10 Geräten geht, wird aussichtslos bei hunderten von Geräten. Zumal sich IP-Adressen gern auch mal ändern. Neuer Heim-Router? Schon passt nichts mehr. Und nicht zu vergessen: Die kleinen Gerätchen müssen an einen Computer gehängt werden, um z.B. die Adressen zu ändern.

Als ein Beispiel für ein gerne als Basis für IoT verwendetes Protokoll werde ich gleich auf MQTT näher eingehen. Höhere Schichten, die z.B. Hausautomatisierungslösungen anbieten, bauen oft darauf auf.

11.1 MQTT

Das „Message Queuing Telemetry Transport“ Protokoll stellt die Lösung dar. Die Grundidee ist ganz einfach. Ein Server, dessen IP-Adresse bzw. seinen Namen (wegen DNS) alle kennen müssen, vermittelt die bidirektionale Kommunikation zwischen den Geräten. Diese Geräte müssen aber keine IP-Adressen kennen, um mit einem anderen zu reden, sondern jede Nachricht wird von einem String angeführt, dem „Topic“, gefolgt von der eigentlichen Nachricht. Der Topic ist so etwas wie ein gegliederter Pfad, ähnlich dem in einem Datei-System. Er wird durch „/“ untergliedert, wodurch unterschiedliche Level entstehen. Prinzipiell kann ich da hineinschreiben, was ich will. Sinnvoll wird das ganze aber durch eine „Semantik“, d.h. ich gebe den einzelnen Elementen eine Bedeutung. Bevor es jetzt zu theoretisch wird, lieber ein Praxisbeispiel:

```
# alle Geräte befinden sich in „home“, das ist meine produktive Umgebung
# homepilot ist ein sog. Device (Gerät)
# Ein Device kann ein oder mehrere Nodes (das sind Sensoren / Aktoren) 
enthalten
home/homepilot/L-Terassenleuchte 1 # Die Terassenleuchte ist an („1“)
home/homepilot/R-WZ 0 # Rolladen WZ ist gerade oben (Position 0%)
home/homepilot/temperature 17.2 # Die Temperatur von Device Homepilot ist 17,2°
home/system/mqtt_ping_ms 2 # Interne MQTT Geschwindigkeitsmessung: 2 ms
home/power/temperature 15.9 # Der Node "temperature" misst die Temperatur 
                            # im Gerät "power"
```

Ich habe mich für eine Dreigliederung entschieden, um das Schema nicht zu kompliziert zu machen. Ein – physisches – Device gibt auch den Ort der Messung vor. Der „homepilot“ misst die Temperatur an der südlichen Außenwand. Ein anderer Temperatursensor sitzt in Device „power“ und misst die Kellertemperatur.

Um damit arbeiten zu können, muss man noch Folgendes wissen:
a) Wie bei Dateien gibt es auch sogenannte „Wildcards“, die Suchzeichen „+“ und „#“. Das Pluszeichen sucht innerhalb eines Levels, das Lattenkreuz über mehrere Level. Eine Suche nach „#“ findet alles, eine Suche nach „home/#“ findet alle produktiven Nachrichten[187], eine Suche nach „home/+/temperature“ findet alle produktiven Temperaturmessungen.

187 Ein „test/#“ würde alle Nachrichten aus meiner Testumgebung finden.

b) Jedes Device, in unserem Falle der Pi Pico, kann Nachrichten senden (mit einem sinnvollen Topic) und empfangen; in der MQTT Terminologie nennt man das abonnieren. Hier gebe ich ein Suchmuster vor. Wann immer ein Sensor Daten an den MQTT Server schickt, schickt der an Geräte, die ein entsprechendes Suchmuster abonniert haben, diese Nachricht weiter.

Überhaupt macht der Server[188] praktisch die ganze Arbeit. Das ist auch gut so, denn dann braucht der Client, also unser Mikrocontroller, nur wenige Zeilen Code um zu funktionieren.

Für Micropython gibt es mehrere Module, die einen solchen Client implementieren. Der einfachste, den ich hier auch vorstelle, heißt „umqttsimple.py".

```
# mqtt_tst.py
import wconn # WLAN bereitstellen
import umqttsimple as mqtt

def cb(t, m): # wird asynchron aufgerufen, wenn eine 
              # abonnierte Nachricht ankommt
    print(t, m) # t enthält den Topic, m die nachricht
    
m = mqtt.MQTTClient("<Name des Client>", "<Server>", keepalive=30)
m.connect()
m.publish("hallo", "test")
m.set_callback(cb)
m.subscribe("hallo/#")
while True:
    m.check_msg()
```

Mit „import wconn" stellen wir die WLAN Verbindung her. Wir importieren umqttsimple mit einem Alias, weil wir schreibfaul sind. Dann folgt eine Funktion "cb", ein Callback. Sie gibt die übergebenen Argumente **t**opic und **m**essage aus.

Dann instantiieren wir die Klasse MQTTClient. Der erste Parameter muss eineindeutig sein! Jeder Client braucht einen eigenen Namen. Als 2. Parameter kann ich hier unseren Raspberry Pi Server „biblio" angeben. Auf ihm muss ein MQTT Broker laufen. Ich habe „mosquitto", eine Open Source Implementierung von MQTT, darauf installiert. Zur Zeit ist der Parameter "keepalive=30" zwingend nötig, da sonst eine Fehlermeldung "2" erscheint. Das liegt an der aktuellen Implementierung der Klasse.

Mit "m.connect()" stellen wir nun die (TCP) Verbindung zum Broker her. Kommt keine Fehlermeldung, hat die Anmeldung funktioniert.

Nun verschicken wir eine Nachricht. In der Terminologie von MQTT nennt man das publizieren (englisch publish). Mit einem Programm auf dem Server, „mosquitto_sub"[189], können wir diese Nachricht sehen:

```
# -t = Topic Suchmuster, -v = zeigt neben der Nachricht auch den Topic an
pi@biblio:~ $ mosquitto_sub -t '#' -v
hallo test
```

Die nächste Zeile in unserem Programm, „m.set_callback", teilt dem MQTTClient mit, welche Funktion bei einer eingehenden Nachricht aufgerufen werden soll, hier der Callback „cb".

188 In der MQTT Terminologie wird er Broker (Vermittler) genannt. Es gibt die Open Source Implementierung „mosquitto" für Windows, MacOS und Linux.

189 Die Hilfsprogramme „mosquitto_sub" (wie subscribe) und „mosquitto_pub" (wie publish) gehören zum mosquitto MQTT Broker und erlauben es, auf der Kommandozeile Nachrichten zu senden und zu empfangen.

Dann abonnieren wir mit „m.subscribe“ den Topic „hallo/#“. Durch den Wildcard „#“ werden alle Nachrichten, die mit „hallo“ beginnen, empfangen.

Dann treten wir in eine Endlosschleife ein, in der die Hintergrundverarbeitung von eingehenden Nachrichten durch die Methode „check_msg“ angestoßen wird. Jede Nachricht erscheint jetzt durch die Callback Funktion auf dem Bildschirm.

Tauschen wir die Reihenfolge

```
m.set_callback(cb)
m.subscribe("hallo/#")
m.publish("hallo", "test")
```

erscheint unsere eigene Nachricht „test“ sofort. Man kann also auch sich selbst etwas senden!

Das ständige Aufrufen von „check_msg“ stört natürlich. Außerdem erhalten wir nach einer Weile eine Fehlermeldung und das Programm bricht ab :-((.

Schuld ist der Broker. Wir haben eine Zeitspanne von 30 Sekunden für das „am Leben erhalten“ (keepalive) gesetzt. Nach dieser Zeit, wenn keine Kommunikation mit dem Broker stattfindet, geht der von einer nicht mehr gewollten Verbindung aus und schließt den Netzwerk-Socket.

Senden wir nun ab und zu eine Nachricht, bleibt die „Leitung“ offen:

```
while True:
    m.check_msg()
    time.sleep(1)
    m.publish("test", "ping")
```

Da wir „test“ nicht abonniert haben, sehen wir davon nichts; dafür überschwemmen wir jetzt unseren „mosquitto_sub -t ‚#‘“.

Laufen mehrere Pi Picos mit diesem Programm, muss jeder einen eigenen Namen vergeben und der abonnierte Topic sollte angepasst werden. Dann kann jetzt der eine Pi Pico einem oder mehreren anderen sowohl Nachrichten senden, als auch von ihnen empfangen.

Ich brauche nicht zu betonen, dass innerhalb von umqttsimple mit Sockets programmiert wird.

11.2 MQTT Spezial

Normalerweise funktioniert MQTT wie das klassische Radio: Einer oder mehrere senden, keiner, einer oder mehrere empfangen. Wenn jemand eine Nachricht sendet, kommt sie sofort bei einem Abonnenten an. Nachrichten, die vor dem Abonnieren gesendet wurden, sieht er nicht. Hier agiert der Broker wie ein Vermittler.

In einer anderen Betriebsart speichert der Broker bestimmte Nachrichten und schickt sie automatisch an jeden neuen Abonnenten. Das ergibt dann Sinn, wenn ich dem Abonnenten bestimmte Eigenschaften, z.B. eines Sensors, mitteilen möchte. Nehmen wir einen Topic wie „home/homepilot/temperature“. Was bedeutet die Zahl, die er als Nachricht liefert? Wo befindet sich der Sensor? Das alles kann ich mit sogenannten persistenten Nachrichten mitteilen:

```
home/hompilot/temperature/unit     °C
home/hompilot/temperature/range    -20, 40
home/hompilot/temperature/location Aussen, Süden
```

Eine weitere Besonderheit stellt der „letzte Wille“ dar. Ich kann dem Broker als mein „Testament“ eine Nachricht mitgeben, die bei einer nicht mehr funktionierenden Netzwerkverbindung ausgegeben werden soll.

Es ist auch möglich, einen Benutzer mit Passwort einzurichten oder die Verbindung zu verschlüsseln. Das führt aber hier zu weit.

11.3 Öffentliche MQTT Dienste

Wenn man MQTT ausprobieren möchte, aber keinen Broker hat, kann man auf öffentliche MQTT Dienste[190] zurückgreifen:

```
mosquitto_sub -h broker.emqx.io -t 'brokertest/#' -v
# beziehungsweise
mosquitto_pub -h broker.emqx.io -m "Hallo" -t 'brokertest/test'
```

Das geht natürlich auch vom Pi Pico aus.

190 Auf dieser Web-Seite wird ein öffentlicher Broker vorgestellt:https://www.emqx.com/en/mqtt/public-mqtt5-broker

VI Warten ist doof – von Interrupts und Multitasking

Wenn wir eine LED mit einer bestimmten Geschwindigkeit blinken lassen wollen, hilft uns die PWM Klasse. Sie baut auf einem Hardwarebaustein im Mikrocontroller auf, der unabhängig von unserem Python Programm arbeitet und einen Pin an- und ausschaltet. Leider können wir die LED nur recht schnell blinken lassen: Bei Frequenzen unter 8 Hz erhalten wir eine Fehlermeldung. Dafür gehen aber über 40 MHz[191]!

Wir können jetzt eine einfache Funktion schreiben, die die LED z.B. mit 1 Hz blinken lässt:

```
def blink(led: machine.Pin, geschwindigkeit: int):
    while True:
        led.on()
        # geschwindigkeit in Hz, daher 1/geschwindigkeit und dann noch durch 2
        # wegen der zwei Wartezeiten
        time.sleep(1 / geschwindigkeit / 2)
        led.off()
        time.sleep(1 / geschwindigkeit / 2)
```

Der Nachteil wird sofort offensichtlich, wenn wir die Funktion aufrufen, und danach noch etwas anderes tun wollen: Das geht nämlich nicht! Die while-Schleife endet ja nie.

Überhaupt, wenn ich eine Zeit lang warten muss, stoppt „time.sleep“ mein Programm vollständig.

1 Interrupts – asynchrone Unterbrechungen

In C-ähnlichen Sprachen (u.a. Arduino[192]) kämen nun Interrupts zum Einsatz. Was ist das? Jeder Mikrocontroller beinhaltet einen oder mehrere Interrupt-Bausteine, die auf interne oder externe Ereignisse hin asynchron eine Funktion[193] aufrufen und dazu das Programm zufällig irgendwo unterbrechen. Interne Quellen können Timer[194], der UART, ADC[195] usw. sein, externe Quellen sind Signale an den GPIOs.

Natürlich geht das auch in Micropython. Aber es gibt sprachspezifische und allgemein interrupt-spezifische Restriktionen. Die wichtigste und schwerwiegendste: Man muss Speicherplatz anfordernde Operationen vermeiden. Leider heißt das u.a. auch, kein „append“ für eine Liste.

---

191 Ca. 41 MHz scheinen bei 50% Tastverhältnis das Maximum zu sein.

192 Arduino ist eine IDE, eine Hardwareplattform und vieles mehr. Man kann damit für Arduino-kompatible Mikrocontroller Programme entwickeln.

193 Asynchron bedeutet, dass das laufende Programm irgendwo, da wo es sich beim Eintreten des Interrupts gerade befindet, unterbrochen wird und die Funktion, die man Callback (Rückruf) nennt, aufgerufen wird. Beim Festlegen des Interrupts wird die Funktion angegeben, aber ohne sie aufzurufen. In Python geschieht das durch den Eintrag des Funktionsnamens ohne die runden Klammern dahinter.

194 Jeder Mikrocontroller verfügt über Bausteine, die Zeiteinheiten zählen und bei Erreichen eines bestimmten Wertes ein Ereignis auslösen können.
Die Syntax bei Verwendung z.B. eines Timers sieht ganz anders aus als für GPIOs:
from machine import Timer
t = Timer(-1)
t.init(mode=Timer.PERIODIC, freq=0.1, callback=p)
# Ruft alle 10 Sekunden die Funktion „p“ auf.

195 In Micropython werden viele Interruptquellen, wie ADC, UART für den Programmierer nicht zur Verfügung gestellt.

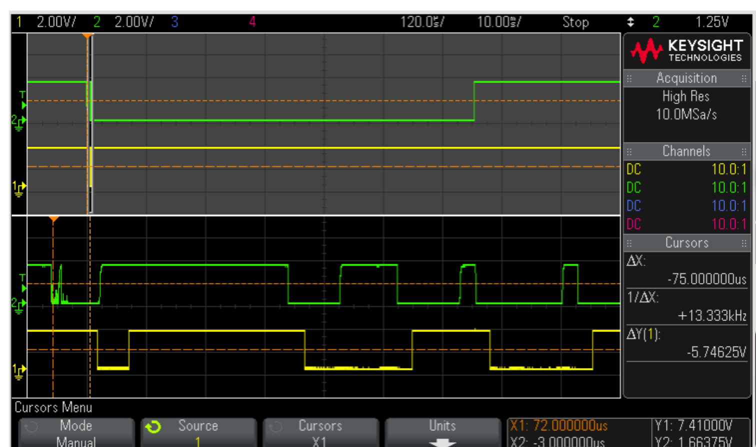
Das Beispiel verwendet der Einfachheit halber einen Interrupt mit einem GPIO als Quelle, um das Prinzip zu verdeutlichen:

```
# isr_tst.py
import time
pin = machine.Pin(14, machine.Pin.IN, pull=machine.Pin.PULL_UP)
led = machine.Pin(13, machine.Pin.OUT)
led.off()
toggle = True

def isr(pin): # Interrupt Service Routine (ISR)
    # Bei jeder fallenden Flanke an Pin 14 wird die LED an oder ausgeschaltet
    global toggle
    if toggle:
        led.on()
    else:
        led.off()
    toggle = not toggle # kehrt den Zustand von toggle um

# Die Methode „irq“ setzt die Bedingung für den Interrupt, hinter „handler“
# folgt der Funktionsname der Callback-Funktion. Sie wird beim Eintreten des
# Interrupt-Ereignisses aufgerufen.
pin.irq(trigger=machine.Pin.IRQ_FALLING, handler=isr) # reagiert nur auf
                                                      # fallende Flanken
while True:
    print(".", end="") # end=““ unterdrückt die Zeilenschaltung
    time.sleep(0.1)
```

Das läuft, aber wir werden vom Ergebnis nicht begeistert sein. Anfangs ist die LED an und auf dem Bildschirm werden Punkte ausgegeben. Nehme ich nun ein Drähtchen oder einen Taster und verbinde Pin 14 mit Masse, geht die LED aus, aber eventuell gleich wieder an usw. Ein eindeutiges Verhalten, also beim ersten Drücken aus, beim zweiten an usw., so wie das Programm das nahelegt, wird nicht erreicht.

Woran liegt das? Das dem zugrunde liegende Phänomen nennt sich „prellen“; drücke ich den Taster, wird eine negative Flanke erzeugt, aber der Schalter schwingt mechanisch und so folgen noch mehrere weitere Flanken. Ob die LED dann an oder aus ist, ist vom Zufall abhängig.

Das Oszillogramm zeigt das recht anschaulich: Die grüne Linie misst am Taster, die gelbe Linie ist das Signal am Ausgang Pin 13. Im oberen Bereich sieht man das Drücken des Tasters, vermisst aber das Unten-bleiben der gelben Linie. Der gezoomte Ausschnitt unten erklärt das. Die Latenz, so nennt man die Verzögerung zwischen Ursprung und Ergebnis eines Signals, verschiebt zeitlich die Reaktion des Ausgangssignals nach rechts. Die erste fallende Flanke (grün) schaltet den Ausgang auf 0, die nächste, unmittelbar folgende (die hier vergrößert dargestellt wird), wieder auf 1. Es folgen noch vier prell-bedingte abfallende Flanken; am Ende ist das Ausgangssignal wieder 1 (LED aus) bevor wir den Taster loslassen :-((



Eine Bemerkung noch zu Interrupts. Anscheinend reagiert Micropython sehr schnell auf die

Schaltflanken. Es vergehen bis zur ersten Reaktion ca. 93 µs, also 93 millionstel Sekunden. Setzt man das aber ins Verhältnis zur Taktfrequenz des Mikrocontrollers von 125 MHz, dann ist das doch ganz schön lang. Auch hier wieder sehen wir den Overhead, den der Micropython Interpreter erzeugt.

Was passiert, wenn während der Ausführung der ISR-Routine ein weiterer Interrupt auftritt? Nach meinen Messungen[196] wird er ignoriert. Das hat zur Konsequenz, dass, wenn ich z.B. einen Ausgang bei jedem – erkannten – Interrupt hin- und herschalte, der Ausgang nicht immer in dem Zustand ist, den ich erwarte. Also nach zwei Flanken erwarte ich den Ausgang auf Low, aber wegen des Ignorierens ist er mal Low, mal High. Wie dem beizukommen ist, werden wir im Kapitel „Taster – zum Zweiten“ behandeln.

2 Multitasking

Gehört hat diesen Begriff, Multitasking, wahrscheinlich jeder schon. Das hat irgendwas mit Mehreres gleichzeitig machen zu tun ...

Wie immer bei Computern kommt es aber auf die Details an. Heute gibt es kein Betriebssystem mehr ohne Multitasking. Ob Windows, MacOS, Linux, alle erlauben es, viele Programme gleichzeitig auszuführen. Aber halt: stimmt „gleichzeitig“ wirklich? Ein Prozessor mit einer CPU kann, vereinfacht gesagt, nur ein Programm ausführen. Deshalb gibt es schon viele Jahre Prozessoren mit mehr als einer CPU: 2, oft 4 oder 8. Ganze neue Prozessoren haben schon mal hunderte CPUs. Nur so viele Programme können gleichzeitig laufen, wie ich CPUen in meinem Rechner habe. Schaut man in die Programm- / Taskliste eines PC, werden einem aber Dutzende aufgelistet. Damit kommen wir zum Begriff „Quasi-Parallelität“, den ich kurz erklären möchte.

In einem Betriebssystem arbeitet ein Scheduler (engl. schedule, planen) genanntes Programm, das andere Programme immer wieder zwangs-unterbricht und ein anderes laufen lässt. Das geschieht – im Idealfall – so schnell, dass ich die kleinen Verzögerungen gar nicht wahrnehme. Moderne Scheduler kombinieren dieses Verhalten mit dem Verteilen der Programme auf alle vorhandenen CPUen. Der Scheduler nutzt dazu natürlich intern auch Interrupts!

Unser Pi Pico besitzt 2 Kerne, wie die CPUen in einem Prozessor auch genannt werden. Eigentlich hat er sogar noch einen 3. Kern[197]; der ist aber ganz anders aufgebaut als die beiden anderen.

Micropython nutzt davon nur einen Kern. Wenn ich das möchte, kann ich auf dem 2. Kern des Pi Pico[198] aber auch (genau) eine Task laufen lassen.

Die Benutzung ist recht einfach:

196 Das gilt nur für den Pi Pico in Kombination mit Micropython. Die Messungen erfolgten mit einem Impulsgenerator und dem Oszilloskop.
197 Das ist die PIO-State-Machine. Die wäre mindestens ein eigenes Kapitel, wenn nicht gar ein ganzes Buch wert ;-).
198 Die sogenannte Multithreading Implementierung in Micropython ist noch experimentell und zudem sehr, sehr hardwareabhängig. Weiterführendes in https://www.elektronik-kompendium.de/sites/raspberry-pi/2712081.htm.

```
import _thread
_thread.start_new_thread(<Funktion>, ())
```

Wenn in der Funktion z.B. ein „time.sleep()" auftaucht, wird davon mein übriges Programm nicht ausgebremst! Die Funktion läuft ja auf einem zweiten CPU Core. Wirklich sinnvoll ist das vor allem dann, wenn sehr lange dauernde, CPU-lastige Aufgaben zu erledigen sind.

Dennoch: nur etwas für sehr erfahrene Programmierer.

Warum? Das Problem besteht in der Zwangs-Unterbrechung. Dieses Verfahren nennt man auch preemptives Multitasking. Der Scheduler, der für das Umschalten von einem Programm(teil) auf das nächste zuständig ist, weiß nichts darüber, was das Programmteil, das er gerade unterbricht, in diesem Moment macht. Diese Programmteile nennt man Task (englisch Aufgabe).

Ein Gedankenspiel: Ich will zu einer Variablen mit dem Wert 5 eine 1 addieren. Dazu muss Python (und jede andere Programmiersprache auch) zunächst den Wert der Variablen lesen, die 1 hinzufügen und dann diesen neuen Wert zurückschreiben: 6.

Wenn der Scheduler das Programm nach dem Lesen unterbricht und das dann laufende Programmteil die gleiche Variable um 2 erhöhen will, dann kann, je nachdem, wann der Scheduler nun wieder unterbricht, Folgendes herauskommen:

a) Der Scheduler unterbricht nach der Addition mit 2 und dem Zurückschreiben; der erste Programmteil addiert 1 zum gelesenen Wert 5. Endergebnis 6.

b) Der Scheduler unterbricht vor der Addition mit 2; der erste Programmteil addiert 5 + 1, schreibt 6 zurück. Nun läuft der zweite Programmteil weiter und addiert 5 + 2. Endergebnis 7.

Es sind noch mehr zeitliche Abläufe denkbar, aber es wird klar, dass das so nicht funktionieren kann. Ein Programmablauf darf nicht von zeitlichen Zufälligkeiten abhängen. Der Fachbegriff dafür heißt Race Condition.

Natürlich gibt es Mittel und Wege, um diese (und andere, hier nicht im Einzelnen aufgeführte) Probleme zu umschiffen. Aber dennoch bleibt die Komplexität der Programmierung hoch.

Es gibt auch noch das sogenannte kooperative Multitasking, dass solche Fallstricke vermeidet. Vereinfacht gesagt, wird hier eine Task nicht zwangsweise vom Scheduler unterbrochen, sondern meldet dem Scheduler, dass sie bereit ist, zu pausieren (deswegen kooperativ). Der Scheduler sucht dann aus der Liste der Tasks, die gerne wieder laufen würden, eine Task heraus und übergibt ihr die Kontrolle über die CPU, bis dann diese Task von sich aus meldet, wieder unterbrochen werden zu können.

Wer jetzt an Windows 3.1 denkt (falls er dieses Uralt-Betriebssystem kennt), der liegt gar nicht so falsch. Ein Betriebssystem mit kooperativem Multitasking zu betreiben ist aber ganz etwas anderes, als das innerhalb *eines* Programms zu tun. Hier hat, im Gegensatz zu einem Betriebssystem, der Programmierer den Ablauf selbst in der Hand.

2.1 AsyncIO

Das Zauberwort für Python und Micropython heißt AsyncIO. Das ist ein kooperativer Scheduler für Python. Schauen wir uns an einem kleinen Beispiel an, wie der grundsätzlich arbeitet.

```
# async_tst.py
import asyncio # Import des Schedulers und anderer notwendiger Programme

led = machine.Pin(13, machine.Pin.OUT)

async def blink(): # das Wort „async“ macht die Funktion multitaskingfähig
    while True:
        led.on()
        # vor einer multitaskingfähigen Funktion muss „await“ stehen
        await asyncio.sleep(0.5) # diese Task für 0.5 Sekunden pausieren
        led.off()
        await asyncio.sleep(0.5) # diese Task für 0.5 Sekunden pausieren

async def haupt():
    # Eine Task erzeugen, in der die Funktion "blink" läuft
    asyncio.create_task(blink())
    while True:
        print(".", end="")
        await asyncio.sleep(1) # warten ohne zu blockieren für 1 Sekunde

asyncio.run(haupt()) # Den Scheduler und die Funktion "haupt" starten
```

Zugegeben, das ist komplizierter als unser Blink-Programm weiter oben. Dafür blinkt jetzt die LED **und** die Ausgabe von Punkten erfolgt.

Von unten angefangen: Zunächst muss immer der Scheduler gestartet werden. Das erledigt die Funktion „asyncio.run“. Dem Scheduler muss außerdem noch eine Funktion genannt werden, die er als erstes aufrufen kann. Diese Funktion muss mit „async def“ definiert worden sein.

Der Scheduler führt „haupt“ aus. Wir können auch Parameter übergeben.

In „haupt“ wird mit der Zeile „asyncio.create_task“ eine neue Task erzeugt. Auch hier muss die angegebene Funktion, hier „blink“, mit dem Vorsatz „async“ definiert worden sein. Dann folgt in „haupt“ eine Endlosschleife, die einen Punkt ausgibt. Vergäßen wir die nächste Zeile „await asyncio.sleep“, verletzten wir eine Grundregel kooperativen Multitaskings: Wir gäben nie die Kontrolle an den Scheduler ab; unsere Task „blink“ käme nie zum Zuge. Das „await“ teilt dem Scheduler mit, dass er jetzt am Zug ist.

In „blink“ haben wir auch zwei Wartezeiten; das eben Gesagte gilt natürlich auch hier.

Wenn ich eine Task erzeuge, wird sie vom Scheduler irgendwann ausgeführt (also nicht unbedingt direkt nach ihrer Erzeugung!). Wenn diese Task endet, liefert sie gegebenenfalls (über „return“) einen Rückgabewert, der ebenfalls irgendwann, aber nach ihrem Ende, abgefragt werden kann. Und wenn sie nicht endet? Dann haben wir einen unabhängig vom übrigen Programm laufenden Programmteil, wie unsere Funktion „blink“.

Warum heißt AsyncIO eigentlich so? Weil das Warten mit „asyncio.sleep“ nur ein, wenn auch gern und häufig genutzter Sonderfall ist. Viel wichtiger ist, dass der Scheduler auf IO[199], also Daten warten kann! Ich kann z.B. auf Daten der Seriellen Schnittstelle (UART) oder Netzwerkdaten warten[200]. Und während ich warte, dürfen andere Tasks laufen. Ich kann auch Daten senden und wiederum warten, bis alle abgeschickt sind. Auch hier laufen andere Tasks weiter. Für unseren

199 IO steht für Input/Output.

200 Dazu brauche ich spezielle Klassen, die asynchrone Methoden bereitstellen. Für den UART heißt so eine Klasse z.B. asyncio.StreamReader. Bei der Instantiierung wird als Argument eine UART Instanz übergeben.

Mikrocontroller spielt es eine besondere Rolle, dass ich auf eine Veränderung des Wert an einem Pin[201] warten kann. Ich muss wohl nicht betonen, dass auch hier andere Tasks weiterlaufen.

2.2 Entwickeln mit AsyncIO

Mit AsyncIO kann ich nun viel einfacher Mikrocontroller-Programme schreiben. Der Nachteil: Die Entwicklung und vor allem das Testen sind viel schwieriger. Eine „async“ Funktion kann ich z.B. nicht im REPL starten, es läuft ja kein Scheduler. Fehler in Tasks führen zu deren Abbruch oder deren „hängen“ (wenn sie dem Scheduler keine Chance geben), ohne dass man immer die Ursache sehen kann.

Python und auch Micropython haben dafür eine Lösung parat. Fangen wir mit Micropython an. Hier nennt sich das Modul „aiorepl“ und, wir ahnen es, es stellt einen REPL mit AsynIO Scheduler bereit. Dieses Modul und ein kleines Hilfsprogramm, das dann „aiorepl“ startet, speichere ich im Dateisystem des Mikrocontrollers[202]. Mit „import arepl“ lade und starte ich es.

Dann ändert sich der Prompt von „>>>“ auf „-->“. Nun kann ich auf der Kommandozeile z.B. „asyncio.create_task(blink())“ eingeben und mit „await asyncio.sleep(10)“ 10 Sekunden warten, ohne dass das Blinken unterbrochen wird.

Das „große“ Python kennt eine Aufrufoption (auch Kommandozeilen-Parameter genannt), mit deren Hilfe ebenfalls der AsyncIO-Scheduler gestartet wird:

```
python -m asyncio
```

Danach befindet man sich in einem asynchronen REPL wie oben für Micropython beschrieben.

Jetzt ist es möglich, asynchrone Funktionen einzeln zu testen.

201 Dazu ist zwar ein „Treiber“ notwendig, aber der kurze Code dafür findet sich als adriver.py unter https://bibliothek.velbert.de/angebote/elektronik-workshop.

202 Auch die Dateien aiorepl.py und arepl.py sind unter https://bibliothek.velbert.de/angebote/elektronik-workshop zu finden.

VII Fehler, Fehler, Fehler!

Manchmal sind Fehler nicht zu vermeiden. Eine erfahrene Programmiererin hat mir einmal gesagt, das Elend finge an, wenn Programme auf Daten treffen ...

Der Programmierer denkt, aber der Benutzer des Programms denkt anders. „Geben Sie eine Zahl von 0 bis 10 ein“, könnte ein Programm den Benutzer auffordern. Der Benutzer gibt 10 ein, der Programmierer aber hatte gedacht wie Python, für das nur bis aber nicht einschließlich der 10 die Werte gültig sind (siehe die „range“ Funktion).

Ein Sensor kann wegen einer technischen Kommunikationsstörung ungültige Daten liefern oder man kann schlicht über unterschiedliche Kodierungen von Zeichen (Unicode, ASCII, Codepage 1252 o.ä.) stolpern.

Bei alledem handelt es sich um Laufzeitfehler. Sie unterscheiden sich von Syntaxfehlern, die schon vor dem Programmstart einen Abbruch herbeiführen.

Laufzeitfehler werden in Python durch die Kapselung in einem Fehlerbehandlungsblock abgefangen. Das heißt, das Programm kann trotz des Fehlers weiterlaufen. Das ergibt nicht immer einen Sinn und so kann man auch selbst das Handtuch werfen. Ein Beispiel:

```
inp = input(„Bitte eine Zahl eingeben“)
# float(inp) wandelt den String in inp in eine Fließkommazahl um
print(„Ich dividiere 100 durch diese Zahl“, 100 / float(inp))
```

Die „input“ Funktion ließt etwas von der Tastatur und gibt einen String zurück. Nun gebe ich verschiedene Zahlen ein: 10.2, 42, 431234504378572577525874235 – klappt alles wunderbar!

Python ist nicht so pingelig mit Integer und Fließkommazahlen. Sie werden stillschweigend umgewandelt. Jetzt gebe ich aber 0 ein! Das hatten wir schon mal:

```
Traceback (most recent call last):
File "/home/gert/mu_code/try_tst.py", line 3, in <module>
print("Ich dividiere 100 durch diese Zahl:", 100 / float(inp))
                                             ~~~~^~~~~~~~~~~~
ZeroDivisionError: float division by zero
```

Noch ein Versuch, jetzt gebe ich „abc“ ein:

```
Traceback (most recent call last):
File "/home/gert/mu_code/try_tst.py", line 3, in <module>
print("Ich dividiere 100 durch diese Zahl:", 100 / float(inp))
                                                   ^^^^^^^^^^
ValueError: could not convert string to float: 'abc'
```

Wenn ich verhindern will, das mein Programm nun unwiderruflich abgebrochen wird, schreibe ich:

```
try:
    # Code, der den Laufzeitfehler hervorruft
except [<Fehlertyp>] [as <Fehlervariable>]:
    # Code nach Auftreten eines Fehlers, der zu „Fehlertyp“ passt
[else:]
    # wird ausgeführt, wenn kein Fehler auftrat
[finally:]
    # wird in jedem Fall noch ausgeführt
# weiter geht es
```

Wie immer sagen die eckigen Klammern, das es sich um optionale Teile handelt, in den spitzen Klammern befinden sich Platzhalter für Namen.

Zwei Fehlertypen habe wir gerade kennengelernt: ZeroDivisionError und ValueError. Wenn ich hinter „except“ einen Fehlertyp setzte, wird nur der abgefangen; steht dort „Exception“, das ist sozusagen der Urvater aller Fehlertypen, wird jeder Fehler behandelt. Diese Behandlung erfolgt im Block unter dem „except“. Wieder bewundern wir die Logik von Programmiersprachen: Der Doppelpunkt als klarer Hinweis an Python, das jetzt ein dazugehöriger Bereich folgt. Hier kann ich nun einfach eine Meldung ausgeben oder, für unser Beispiel besser geeignet, die Eingabe einer Zahl erneut anfordern.

Hier das Beispiel am Stück:

```
# try_tst.py
def eingabe():
    inp = input("Bitte eine Zahl eingeben: ")
    print("Ich dividiere 100 durch diese Zahl:", 100 / float(inp))

while True:
    try:
        eingabe()
    except ValueError as err: # in "err" wird der Fehler gespeichert
        print("Ein ValueError Fehler:", err)
    except ZeroDivisionError as err:
        print("Ein ZeroDivisionError Fehler:", err)
    except Exception as err:
        print("Ein anderer Fehler:", err) # In Windows stünde hier „unbekannter
Fehler“ ;-)
```

VIII Taster - zum Zweiten

Die meisten Programme bisher waren – zugegebenermaßen – eher Beispiele oder Tests. Wir befinden uns aber nun in der Lage, mit OOP und Multitasking eine sehr praktische Anwendung zu schreiben: Eine Klasse, die mit Hilfe eines Tasters kurze, lange und wiederholte Tastendrücke auswerten kann!

1 Zustände oder Flanken

Es gibt zwei grundsätzliche Verfahren, wie man einen Taster abfragen kann. Beginnen wir mit der Zustandserkennung. Auch sie dient dazu Flanken, also Wechsel von einem Zustand zu einem anderen Zustand zu erkennen. Danach werden wir uns mit der - hardwarebasierten - Flankenerkennung beschäftigen.

Wir beginnen mit den möglichen Zuständen eines Tasters. Davon gibt es vier:

Jetzt	**Vorher**	**Status**	**Bemerkung**
Nicht gedrückt	Nicht gedrückt	0	Stabil
Gedrückt	Nicht gedrückt	1	Flanke
Gedrückt	Gedrückt	2	Stabil
Losgelassen / nicht gedrückt	Gedrückt	3	Flanke

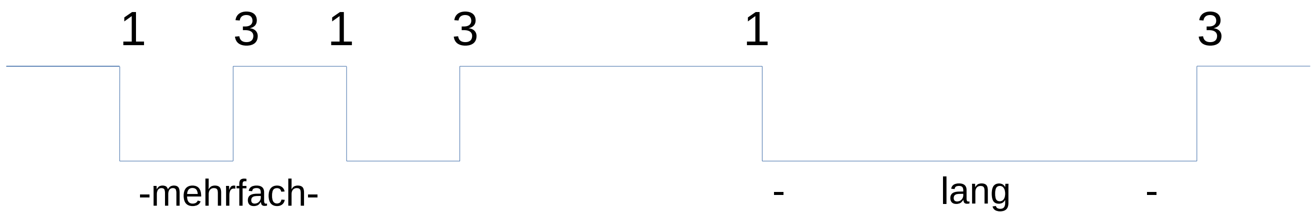
Wenn wir diese Zustände durchnummerieren, erhalten wir die Zahlen 0 – 3. Ermittle ich einen Zustand, indem ich den GPIO Pin z.B. in einer Schleife asynchron[203] abfrage, und der neue Zustand weicht vom alten, gemerkten Zustand, ab, habe ich es mit einem Zustandswechsel, einer „Flanke“ zu tun. Jetzt kann man Timer setzen oder auswerten usw. Der Zustand danach[204], wenn der gemerkte Wert und der gerade ermittelte wieder gleich sind, dient der Entprellung. Die Timer benötige ich, um zwischen kurzem und langem Drücken oder mehrfachem Drücken zu unterscheiden. Hier liegt eine Stolperfalle vergraben: Die Länge eines Tastendrucks kann ich erst dann ermitteln, wenn ich wieder loslasse. Ich kann also Tastendrücke immer erst bei Status 3 endgültig[205] auswerten.

Ein Bild sagt mehr als tausend Worte. So sieht das Signal am GPIO aus, wenn ich von links nach rechts den Taster zweimal kurz hintereinander drücke, dann etwas warte, dann lang drücke. Den normalen Einfachdruck habe ich weggelassen. Die Zahlen sind die Zustände aus unserer Tabelle; die 0 und 2 habe ich der Übersichtlichkeit halber nicht mit angezeigt.

203 In einer Schleife frage ich den Zustand des Tasters zu irgendeinem Zeitpunkt, daher „asynchron“ ab.

204 Ob ich im Status 1 oder 2 den Tastendruck auswerte, hängt davon ab, was ich genau vorhabe. Um z.B. sehr kurze Tastendrücke oder Prellen zu erkennen, würde ich in Status 1 eine Stoppuhr erst abfragen und dann starten. Ist der Zeit beim Abfragen zu klein, ignoriere ich die Tastenbetätigung und erkenne auf Prellen.

205 Warum bei „3“? Lasse ich den Taster los und erhalte im Programm mit „pin.value()“ eine „1“, ist es mir egal, ob das nur ein Preller war oder nicht. Wichtig ist: der Taster wurde losgelassen. Auf „0“ abfragen geht hier nicht, da das auch der Anfangs- oder Ruhezustand ist.



Zu sehen ist, dass ich, um einen Mehrfachdruck zu erkennen, von der fallenden bis zur nächsten fallenden Flanke die Zeit messen muss. Um einen langen Tastendruck zu detektieren, muss ich dagegen von der fallenden bis zur nächsten steigenden Flanke messen. Diese Überlegungen sind wichtig für die Planung der benötigten Funktionen.

Zustandserkennung lässt sich immer anwenden, wenn es sinnvoll oder möglich ist, in ziemlich regelmäßigen, nicht zu langen Abständen den Taster abzufragen. Der Elektroniker nennt das „Polling“ (dt. abfragen, oft auch aktives Warten genannt). Klassische, nicht Multitasking nutzende Programmierung, die eine Hauptschleife benutzt, in der dann die verschiedensten Aktionen angestoßen werden, würde dieses Verfahren nutzen. Aber auch mit AsyncIO und einer Task, die periodisch den Taster ließt, kann man dies implementieren.

Verwende ich dagegen eine Flankenerkennung, die durch Hardware Interrupts oder im Falle von AsyncIO durch einen Treiber als Software Interrupt bereitgestellt wird, wird eine Callback-Funktion in dem Moment, wo die Flanke erscheint, aufgerufen. Das ist quasi die Stufe 1 und die 3 in unserem Statusmodell. Im Callback kann ich die erforderlichen Schritte, wie z.B. eine Zeit zu messen oder einen Zähler zu erhöhen, direkt gehen.

Während in Micropython ein Hardware Interrupt zwar auf fallende und steigende Flanken reagieren kann, aber nur eine Callback-Funktion aufrufen kann, erlaubt mir der AsyncIO Treiber[206] zwei unterschiedliche Callbacks für jeden Flankentyp.

2 Stopp-Uhr

Wir haben im vorigen gesehen, dass wir ermitteln können müssen, wie lange etwas dauert. Beginnen wir daher unser Programm mit einer Stopp-Uhr. Das Grundgerüst haben wir im Kapitel „Der Walross-Operator“ schon kennengelernt. Der Stopp-Uhr teilen wir mit, ab wann die Messung beginnen soll (Start Knopf) und wie lange etwas gedauert hat (Stopp Knopf). Dazu wird zuerst die aktuelle Zeit gespeichert. Wenn wir später die Frage stellen, wie viel Zeit vergangen ist, beziehen wir uns auf diese Referenz; die Zeitmessung soll aber weiterlaufen.

Diese Aufgabenstellung drängt sich in Python geradezu auf für den Entwurf als Klasse. Damit können wir dann so viele Stopp-Uhren erzeugen, wie wir brauchen.

Die Methode time.time() liefert in Micropython nur Sekunden zurück und ist daher für die Zeitmessung an einem Taster ungeeignet. Micropython kennt zwei[207] Zeitmessmethoden mit

206 Der ist im Anhang abgedruckt.
207 Es gibt noch eine Dritte, "ticks_cpu", die eine noch höhere Auflösung hat.

höherer Auflösung: „time.ticks_ms“ und „time.tick_us“[208], die Millisekunden- oder Mikrosekundenauflösung bieten.

Einen Fallstrick bei der Nutzung bildet der Überlauf der Zahlen. Aus Geschwindigkeitsgründen werden diese Werte nämlich nicht als – theoretisch unbegrenzt große – Integer-Zahlen, sondern als 32 Bit[209] Zahl verarbeitet. Die maximale Größe einer vorzeichenlosen 32 Bit Ganzzahl ist 4294967295[210]. Bei Mikrosekundenauflösung müssen wir durch 1000000 teilen; das ergibt ~4294 Sekunden. 3600 Sekunden dauert eine Stunde, also läuft dieser Timer nach etwas mehr als einer Stunde und zehn Minuten über. Bei Millisekundenauflösung findet der Überlauf erst nach über 1000 Stunden statt.[211] Messen wir also kurze Zeiträume, sollte eigentlich alles OK sein. Aber **nein**:

Stellen wir uns vor, der µs Timer stände kurz vor dem Überlauf. Wir merken uns z.B. 4294967000. Nach 300 µs prüfen wir, wieviel Zeit vergangen ist. Die Funktion „ticks_us“ gibt 5 zurück. Ermitteln wir die vergangene Zeit durch die Subtraktion 5 - 4294967000, erhielten wir einen negativen Wert.

Um solches Fehlverhalten zu vermeiden, gibt es für die Differenzbildung eine eigene Funktion „ticks_diff(<aktuelle Zeit>, <gespeicherte Zeit>)“[212]. Es versteht sich von selbst, dass man nur ms mit ms und µs mit µs vergleichen darf.

Achtung: das Überlaufproblem ist damit nicht behoben, sondern nur das Problem, kurz vor dem Überlauf die Startzeit gesetzt zu haben. Anders ausgedrückt: Wir können nur genau *einen* Überlauf detektieren.

```
# hilfsprogramme.py
# Auf Pi Pico
import time

class StopWatch:
    def __init__(self, resolution: str= "us"):
        self.resolution = resolution
        self.start()

    def start(self):
        if self.resolution == "us":
            self._start = time.ticks_us()
        else:
           self._start = time.ticks_ms()

    def stop(self):
        if self.resolution == "us":
            return time.ticks_diff(time.ticks_us(), self._start)
        else:
            return time.ticks_diff(time.ticks_ms(), self._start)
```

208 Wie bei der PWM Klasse, wird ein Timer intern hardwaremäßig hochgezählt. Außer, wenn ich mit „time.tick_us“ den Wert abfrage, ist der Python Interpreter **nicht** involviert.
209 Auf einem 32 Bit Controller wie dem Pi Pico.
210 Das errechnet sich aus 2^32-1. Die Zahl 2^32 wäre eine 1 mit 32 Nullen dahinter im Binärformat, also genau 1 zu groß.
211 Auf 8 Bit Controllern (wie auf dem Arduino Uno) sind die Timer-Zähler oft 16 Bit breit. Dort laufen Millisekunden Timer schon nach 18 Sekunden über!
212 In der Micropython Dokumentation wird das lang und breit erklärt. (https://docs.micropython.org/en/latest/library/time.html). Ganz vereinfacht: Sobald die Differenz negativ wird, subtrahiert man den Startwert vom der maximalen Größe (hier 2^32) und addiert den aktuellen Wert dazu.

```
    def print(self):
        if self.resolution == "us":
            print(f"{self.diff()} us")
        else:
            print(f"{self.diff()} ms")

sw = StopWatch() # Auflösung in µs ist der Default
sw.print()
sw.print()
t1 = sw.stop()
t2 = sw.stop()
print(t2 - t1)
# 183 us
# 1059 us # Differenz 1059 – 183 = 876
# 61
```

Man kann hier sehen, dass zwischen je zwei Aufrufen von „sw.print“ und „sw.stop“ unterschiedlich viel Zeit vergeht, hauptsächlich wegen des im Verhältnis sehr langsamen „print“[213].

Wir nennen die Klasse „StopWatch“. Intern speichert sie für jede Instanz die gewünschte Auflösung (englisch „resolution“) und die Startzeit, also die Zeit, die man bei einer Sport-Stopp-Uhr mit der „Start“ Taste festlegen würde.

Die Methode „start“ gestattet es, die Startzeit neu festzulegen. Die Methode „stop“ ermittelt die verstrichene Zeit und gibt sie als Zahlenwert zurück. Die Methode „print“ benutzte ich hier nur zur Bequemlichkeit.

Was können wir aus diesen wenigen Zeilen lernen? In Zeile 6 des Programms rufe ich „self.start“ auf. Diese Methode wird erst weiter unten definiert. Das zeigt, dass der Python Interpreter, wie ja schon ausgeführt, das gesamte Programm kompiliert, bevor er es startet. Daher „weiß“ er, schon bevor die Methode im Quelltext auftaucht, dass es die Methode „self.start“ weiter unten gibt.

Ich kann eine Methode auch „print“ nennen, obwohl es diese Funktion schon gibt. Da der Aufruf aber mit „sw.print“ erfolgt, gibt es keine Verwechslungsgefahr.

Jetzt können wir eine verstrichene Zeit genau erfassen!

3 Status Ermittlung

Wie wir in der obigen Tabelle gesehen haben, brauchen wir jeweils die Übergänge vom alten Zustand in den neuen. Wieder können wir eine Klasse programmieren, um beliebig viele Zustandserkennungen, z.B. für viele Taster, zu realisieren.

Die Klasse ist etwas komplizierter.

```
# hilfsprogramme.py

class State:
    def __init__(self):
        self.transition_rules = ((1, 1), (0, 1), (0, 0), (1, 0))
        self.rule_state = 0 # der Index der aktuellen Regel
```

213 Die Zeichen werden mit 115200 Baud nach und nach ausgegeben; grob gerechnet heißt das 115200 / 8 (Bit) ergibt 14400 und 1 / 14400 ungefähr 70 µs pro Zeichen.

```
        self.state = self.transition_rules[0][0] # der aktuelle Status für 
Regel 0
        self.last_state = self.transition_rules[0][1] # der vorherige Status 
für Regel

    def evaluate(self, input):
        cur, prev = self.transition_rules[self.rule_state] # die Werte aus der 
                                                           # Regel extrahieren
        # wenn input und die gespeicherten Werte passen
        input_match = (input == cur) and (self.last_state == prev)
        if input_match:
            ret = self.rule_state # Index der Regel zurückgeben
            self.rule_state += 1  # Nächste Regel auswählen
            # wenn Index der letzten Regel überschritten
            if self.rule_state >= len(self.transition_rules):
                self.rule_state = 0 # erste Regel auswählen
                # wieder initialisieren
                self.state, self.last_state = self.transition_rules[0]
            else:
                self.last_state = input # aktuelle input wird letzter Status
            return ret
        else:
            return -1 # keine Änderung oder Fehler
```

Mit "sm.evaluate(button.value())" wird der Taster ausgewertet. "evaluate" gibt -1 für alle unveränderten, aber auch ungültigen Werte zurück. Dadurch wird Prellen sehr stark unterdrückt[214]. Gebe ich die Rückgabewerte aus, z.B. indem ich in einer Schleife den Zustand des GPIO Pins abfrage, sieht das so aus:

```
-1 # keine Änderung
-1
-1
1  # Taster wird gedrückt (Flanke)
2  # Taster weiterhin gedrückt
-1 # keine Änderung, Taster weiterhin gedrückt
-1
...
-1
3  # Taster losgelassen (Flanke)
0  # Taster nicht gedrückt, wieder auf Start 
-1
-1
```

Wenn bei „1“ die Stopp-Uhr gestartet wird, kann bei „3“ gemessen werden, wie lange der Taster gedrückt wurde. Für das Erkennen von Mehrfachbetätigungen müssen wir überlegen, was das bedeutet. Innerhalb einer Zeitspanne, die vom Status „1“ eingeleitet wird, messe ich die Zeit bis zum nächsten Status „1“. Wenn sie kürzer als die definierte Spanne ist, handelt es sich um einen Mehrfachdruck, sonst um einen Einfachdruck.

4 Eine Taster-Auswertungsklasse

Um die Klasse zum Auswerten von entprellten Tastendrücken zu realisieren, die wir ButtonHandler taufen, müssen wir jetzt die vorher erstellten Teile kombinieren. Wir haben die Wahl, entweder von

214 Ob Prellen damit *verhindert* wird, ist Gegenstand umfangreicher Informatiktheorien. M.E. wird mit dieser Methode Prellen tatsächlich eliminiert, wenn die Abfragegeschwindigkeit langsam genug ist.

den Klassen StopWatch und State zu erben oder diese Klassen in der Klasse ButtonHandler zu benutzen.

Ich möchte die Gelegenheit ergreifen und nochmals auf den - wichtigen - Unterschied zwischen erben von einer Klasse und verwenden der Instanz einer Klasse hinweisen. Letzteres machen wir ständig, wenn wir einen String, eine Zahl, eine PWM usw. in unserem Programm verwenden, denn das alles sind Instanzen ihrer Klassen. Hier würde es überhaupt keinen Sinn ergeben, von ihnen zu erben: Wenn wir mehr als eine Instanz einer Klasse innerhalb einer anderen benötigen, verbietet sich zudem meist [215]Vererbung! Von einer Klasse erben erweitert (oder ändert) den Funktionsumfang der bestehenden Klasse, also der Blaupause, oder wie ich es genannt habe, die Form des Plätzchenaustechers. Damit haben alle Instanzen, die davon gebildet werden, diese neuen Eigenschaften. Sie kapseln aber auch ihre Daten gegen andere Instanzen, auch von der gleichen Klasse, ab. Nutze ich hingegen Instanzen innerhalb einer Klasse teilen sie sich ihre Daten!

5 Was sie können soll

Analysieren wir die Aufgaben von ButtonHandler: Wir werden eine State-Instanz benötigen, da wir einen Taster verwenden, aber wie viele StopWatch-Instanzen? Wir benötigen eine, um die Mehrfachdrücke zu ermitteln und eine, um kurzes von langem Drücken zu unterscheiden, wie oben beschrieben. Aber Halt: Beide Stopp-Uhren werden in Status "1" gestartet. Die eine, die das Mehrfachdrücken detektieren soll, wird vor dem Start in Status "1" noch ausgewertet. Die andere zur Erkennung von langem drücken werten wir erst in Status "3" aus. Schaut man sich das genau an, wird ersichtlich, dass wir ein- und dieselbe Stopp-Uhr zweimal verwenden können. Das liegt daran, dass wir bei der Entwicklung entschieden haben, die Stopp-Uhr beim Kommando "stop" weiterlaufen zu lassen. Ein Problem haben wir aber immer noch. Wenn wir mehrmals kurz drücken, erscheint die Ausgabe[216]

```
# print(self.button_cnt, self.button_lang)
1 False
2 False
3 False
```

Wir erhalten also nicht das fertige Ergebnis "dreimal gedrückt", sondern bei jedem Drücken erscheint eine Zeile. Können wir das beseitigen?

6 Zurück in die Gegenwart

Dazu müssen wir zunächst analysieren, woher das kommt. Durch Auswerten der Stopp-Uhr im Status "1" können wir feststellen, dass der vorherige Tastendruck **kurz** vorher stattgefunden hat. Diese Feststellung erfolgt also erst beim zweiten Tastendruck. Aber zudem wissen wir nicht - und können das auch nicht wissen, weil wir keine Hellseher sind - ob der nächste Tastendruck auch ein Mehrfachdruck, also innerhalb der kurzen, festgelegten Zeit, erfolgen wird.

Sofern wir nicht irgendwo eine Glaskugel auftreiben, ein anscheinend unlösbares Problem! Aber wie jeder „Magier“ lösen wir es doch: durch einen Trick! Der Bühnenzauberer lenkt von dem, was er macht, geschickt ab. Das tun wir auch, indem wir die Zukunft durch Verzögerung der Gegenwart

215 Wie immer gibt es Ausnahmen, die aber sehr spezielle Nebenbedingungen haben.
216 Das bezieht sich auf Variante 1 unserer Klasse.

ins Jetzt verschieben. In weniger magischem Deutsch: Wir warten vor der Ausgabe des Ergebnisses des Tastendrucks immer eine kleine Weile. Wir brauchen dazu noch eine Stopp-Uhr, die wir diesmal in Status „3“ starten und in der „while“-Schleife, außerhalb der „if“-Kette für die Statusabfragen, abfragen. Dort warten wir länger als *ein* Mehrfachdruck sein darf; bei jedem weiteren als Mehrfachbetätigungen erkannten Druck wird diese Stopp-Uhr wieder neu gestartet[217]. Erst dann geben wir aus, was wir erkannt haben: einen normalen-, einen Mehrfachdruck oder einen langen Druck. Der Zuschauer unseres Bühnentricks bekommt die Verzögerung erfahrungsgemäß nicht mit.

7 Und Prellen?

Brauchen wir auch noch eine Stopp-Uhr, um sehr kurze Flankenänderungen zur Vermeidung von Prellen zu detektieren? **Ja** - denn unsere State-Klasse kennt keine Zeiten. Es würde auf Änderungen im Millisekundenbereich genauso reagieren wie auf Änderungen im Sekundenbereich. **Nein** - wenn die Abfrage von "sm.evaluate(button.value())" nur z.B. alle 20 ms erfolgen würde. Fazit: es hängt von unserer Implementierung ab, also davon, wann und wie oft oder schnell wir Flanken feststellen.

Welche Möglichkeiten haben wir, Flanken zu detektieren? Eine Schleife, die eventuell sogar mit „time.sleep“ wartet, kommt nicht in Frage. Unser Programm, so lautet die simple Antwort, könnte dann nichts anderes mehr tun.

Weiter oben habe ich Interrupts und AsyncIO erläutert. Bei Interrupts wird jedes mal, wenn ein GPIO Eingang seinen Zustand ändert, eine Funktion aufgerufen, ein sogenannter Callback[218]. Damit kann ich sehr schnell auf Flanken reagieren. Das wäre also die erste Möglichkeit.

Wenn wir aber schon ein asynchrones Programm mit AsyncIO schreiben, bietet es sich an, die Flankenerkennung auch mit den Mitteln von AsyncIO zu implementieren. Wir könnten eine Task erzeugen, in ihr den Status des GPIO abfragen und bei einer Änderung z.B. einen Zähler verändern.

```
# Beispielprogramm
from machine import Pin
import asyncio
pin = Pin(15, Pin.IN, Pin.PULL_UP)

async def button_pressed():
    v = pin.value() # Anfangswert setzten
    cnt = 0
    while True:
        if pin.value() != v:
            print("Taster Status geändert!", cnt)
            v ^= 1 # kehrt Wert um
            cnt += 1 # Zähler Variable inkrementieren
        await asyncio.sleep_ms(50) # andere Tasks zum Zuge kommen lassen

asyncio.run(button_pressed()) # Start des Schedulers
```

Auf keinen Fall dürfen wir eine Wartezeit, natürlich asynchron, in dieser Schleife vergessen, sonst kommen die anderen Tasks nicht zum Zuge. Aber es geht noch besser: Man kann den AsyncIO

217 Das ist die Variante 2 der Klasse.
218 Im Kontext von Interrupts wird diese Funktion gerne ISR (Interrupt Service Routine) genannt.

Scheduler veranlassen, diese GPIO Abfragen direkt zu übernehmen. Das Programm „adriver.py“[219], das sich auf der WEB-Seite der Stadtbibliothek findet, realisiert das. Es ist deutlich performanter.

8 Ja wie denn nun?

Jetzt haben wir so viele Alternativen kennengelernt, dass es schwer fällt, sich zu entscheiden. Auch das gilt es zu verstehen: Diese Situation, zig Möglichkeiten zu haben, weil es eben nicht den einen Weg, die eine Lösung gibt, werden wir beim Programmieren immer wieder vorfinden. Unter Programmierern gibt es den Spruch, dass man jedes Programm dreimal schreibt.

Noch ein Aspekt soll hier hervorgehoben werden. Es gibt die Begriffe des „Top Down“ und des „Bottom Up“ Programmieransatzes. Gemeint ist damit, dass man „von Oben“, also aus der Vogelperspektive, ein Projekt entwerfen kann; zwangsläufig konzentriert man sich dann mehr auf die Architektur. Die Schwierigkeiten fangen dann oft erst bei der Umsetzung der Details an. So, wie ein Bauingenieur häufig feststellen muss, dass der wunderschöne Entwurf des Architekten an der Statik scheitert. Fange ich hingegen mit dem Klein-Klein technischer Einzellösungen an, also „von Unten“, verliere ich gegebenenfalls das „Gebäude“, das es zu errichten gilt, aus dem Auge. Wünschenswert wäre eine Kombination aus „von Unten“ und „von Oben“, ein hin- und herwechseln zwischen den Ansätzen.

Es gibt daneben aber auch solche Einflussgrößen wie eigene (Vor-)Kenntnisse, Vorlieben und Gewohnheiten. Und das unterliegt dann auch noch dem Faktor Zeit, weil durch die Tätigkeit des Programmierens, das ständige Hinzulernen dabei, sich die Voraussetzungen ändern. Womit wir wieder bei dem dreimal Schreiben eines Programms wären.

Hier entscheide ich mich nun für die folgende Implementierung, pragmatisch, um die schon entwickelte[220] „State“ Klasse nutzen zu können:

Die Klasse „ButtonHandler“ nutzt (anstatt zu erben) die Klassen „StopWatch“ und „Stage“. „ButtonHandler“ wird als AsyncIO Task implementiert, um sie später in einem größeren Mikrocontroller-Programm wiederverwenden zu können.

Die Funktionsbeschreibung, man nennt das auch Pflichtenheft, sieht so aus:

Druck	Zeit / Status	Anzahl	Ergebnis	Erläuterung
Kurz	< 800 ms / 1-3	1	normal	Normaler Einfachdruck
Kurz	< 300 ms / 1-3	2-n	Mehrfachdruck	2 – n maliges kurzes Drücken (wird gezählt)
Lang	> 800 ms / 1-3	1	lang	Langer Druck
Verzög.	> 300 ms / 3,3		Mehrf. als 1 Wert	Nach Loslassen wird um 300 ms verzögert

Schreiten wir zur Umsetzung. Da wir entschieden haben, nicht zu erben, sondern zu Nutzen, sieht die Klassendefinition so aus:

219 Der dort abgedruckte Code läuft einwandfrei auf meinen Pi Picos mit WLAN, nicht aber auf denen ohne! Ich habe dann die WLAN Firmware auf den Pi Pico ohne WLAN geflasht; dann lief zwar der Code aus dem Modul „adriver“, aber ob ansonsten alles korrekt ist, kann ich nicht sagen. :-((

220 Das wäre dann der „Bottom Up“ Ansatz, der oft aus der Situation heraus entsteht, schon eine Detail-Lösung zu haben, die man dann auch verwenden will.

```
# buttonhandler.py
class ButtonHandler: # keine Elternklasse
```

Wir wollen bei der Instantiierung angeben können, an welchem GPIO der Taster angeschlossen werden kann:

```
    def __init__(self, pin: int):
        self.pin = Pin(pin, Pin.IN, Pin.PULL_UP)
        self.sw_multi = StopWatch("ms")  # Nutzung von Klasse StopWatch
        self.sw_delay = StopWatch("ms")  # Nutzung von Klasse StopWatch
        self.state = State()  # Nutzung von Klasse State
        self.LANG = 800
        self.MEHRFACH = 300
        self.button_cnt = 0
        self.button_lang = False
        # Erzeugt die Task "button_pressed_task"; läuft nun unbahängig
        asyncio.create_task(self.button_pressed_task())
```

Jetzt passen wir die Funktion „button_pressed_task“ so an, dass sie „State“ verwendet:

```
    async def button_pressed_task(self):
        cnt = 1
        fl = False # fl für Flagge
        while True:
            e = self.state.evaluate(self.pin.value())
            if e == 1:
                if self.sw_multi.stop() < self.MEHRFACH:
                    cnt += 1
                else:
                    self.button_cnt = 1
                self.sw_multi.start()
            elif e == 3:
                fl = True # Status "3" erreicht
                self.sw_delay.start()
                if self.sw_multi.stop() > self.LANG:
                    self.button_lang = True
                    self.button_cnt = 1
                else:
                    self.button_lang = False
                # Variante 1
                # print("Button", self.button_cnt, self.button_lang)
            await asyncio.sleep_ms(20)
            # ausserhalb der Statusabfrage, Variante 2
            if self.sw_delay.stop() > self.MEHRFACH and fl:
                 # Inline "if": wenn cnt > 1 dann cnt sonst 1
                 self.button_cnt = cnt if cnt > 1 else 1
                 print("Button", self.button_cnt, self.button_lang)
                 fl = False # rücksetzen
                 cnt = 1 # rücksetzten

    def get_button(self):
        return self.button_cnt, self.button_lang
```

Um das Ganze auszuprobieren, ergänzen wir

```
if __name__ == "__main__": # Wird nicht ausgeführt, wenn importiert
    b = ButtonHandler(15) # Pin 15
    async def main():
        while True:
            await asyncio.sleep(1.5)
```

```
            #print(b.button_cnt)
    asyncio.run(main())
```

Wollen wir später in einem eigenen Programm diese Klasse beützen, schreiben wir

```
from bottonhandler import ButtonHandler
…
b = ButtonHandler(15)
# irgendwo im Programm
cnt, lang = b.get_button() # liefert die jeweils letzte Tastenbetätigung zurück
```

9 Das zweite Mal schreiben ...

Unser Programm, das die Klasse „ButtonHandler“ einbindet, ist jetzt schon einige Wochen im Einsatz und auch andere Personen benutzen es. Der lange Druck öffnet ein Menü. Der Mehrfachdruck dient dazu, einen bestimmten Menüpunkt (den ersten, zweiten oder dritten) auszuwählen. Auf dem Display sieht der Anwender, welcher Menüpunkt gerade aktiv ist.

Die ersten Klagen kommen: Für die eine Gruppe ist die Eingabe „zu träge“, sie würden dreimal drücken aber das Menü würde nur zum zweiten Eintrag wechseln, die andere Gruppe behauptet, so schnell könnten sie gar nicht drücken, es würde immer nur ein Einfachdruck erkannt.

Anscheinend widersprechen sich diese Aussagen. Wir analysieren, vielleicht mit einem Oszilloskop, wie lang die Drücke der einen Gruppe und wie lang die der anderen Gruppe sind. Unsere Annahme, dass ein Tastendruck ca. 100 ms dauert, wird widerlegt: es gibt etliche Personen, die nur etwa 30 ms lang drücken, andere aber brauchen eher 250 ms.

Der Abstand zwischen den einzelnen Abfragen des Status beträgt nur 20 ms. Das sollte doch schnell genug sein! Aber halt - es gibt ja 4 Stufen, ein kompletter Tastendruck zerfällt in 4 x 20 ms. Wenn ich also sehr schnell hintereinander den Taster betätige, kann es sein, dass unser Algorithmus das schon als Prellen interpretiert. Oder, und das ist bei kooperativem Multitasking ja leicht möglich, andere Tasks hindern die „button_pressed_task“ daran, wirklich alle 20 ms den Taster auszuwerten.

Zeit, die Entwicklungsumgebung wieder bereit zu machen und nachzubessern. Vielleicht stoßen wir dabei auf ein ganz anderes Problem, z.B. dass durch einen Fehler wirklich eine Task manchmal viel länger braucht. Oder wir merken, dass unser Bedienkonzept mit dem Mehrfachdruck doch nicht so toll ist.

Was auch immer uns einfällt, um eine Lösung zu finden, wir schreiben unser Programm nun das zweite mal ...

IX Debuggen

Am Anfang meiner Programmiererfahrung, vor vielen, vielen Jahren, schrieb ich ein Programm in Fortran, das mir eine ausgedruckte Liste von Impedanzen für verschiedene Widerstände und Kondensatoren erzeugen sollte, da ich keinen Taschenrechner besaß und die komplizierte Formel zu berechnen sehr mühselig war. Leider hatte ich keinen Erfolg (außer, dass ich nun ein wenig Fortran konnte), da mein Programm nicht das machte, was ich wollte. Später, in Basic, hatte ich wieder oft das Problem, das zwischen dem, was ich mir vorgestellt hatte, und dem, was sich die Programmiersprache darunter vorstellte, Welten lagen.

Mein Erstkontakt zu einem Debugger ergab sich erst mit dem Aufkommen von Linux Anfang der 90er Jahre. Plötzlich[221] gab es C mit einer kompletten Toolchain vom Kompiler, Linker bis hin zum Debugger. Der konnte ein Programm an einer wählbaren Stelle unterbrechen, man konnte das Programm in Einzelschritten fortführen und – man konnte sich die Daten zu diesem Zeitpunkt anschauen!

Überraschung! Die Daten sahen oft ganz anders aus, als gedacht. Damit war klar, wo das Problem gelegen hatte und die Behebung nur noch einen Schritt entfernt.

In Python gehört ein Debugger standardmäßig dazu. Er heißt „pdb“ (Python Debugger) und in der Konsole sieht das so aus:

```
python -m pdb <Programm-Name>
```
[222]

Bis heute lese ich im Internet oder in Büchern, dass man Programme ja auch durch Ausgabeanweisungen, in Python also „print“, debuggen könne. Das ist richtig und wird von mir auch genutzt. In einem 100 Zeilen Programm kommt man damit auch ganz gut zurecht; was aber, wenn das Programm 1000 oder 10000 Zeilen umfasst? Wie viele „print“ will ich da einbauen, wie kann ich die Ausgaben den (Fehler-)Stellen zuordnen? Was mache ich mit den ganzen print-Anweisungen, wenn mein Programm fertig ist?

Mit geeigneter IDE, mit Mu geht es eingeschränkt auch, brauche ich nur an den linken Rand klicken, ein roter Punkt erscheint und mein Programm hält dort an und ich kann mit der Fehlersuche loslegen.

Jetzt kommt der Wermutstropfen: In Micropython steht KEIN Debugger zur Verfügung. Jetzt macht „Micro“python seinem Namen keine Ehre. Verständlich ist das schon, um Micropython auf auch kleinen Mikrocontrollern laufen lassen zu können, müssen die Ressourcen schon sehr schonend eingesetzt werden. Und ein Debugger ist kein kleines Programm(teil)!

Meine Methode besteht darin, komplizierte Datenverarbeitungen in Python zu entwickeln, zu debuggen und erst dann gegebenenfalls an Micropython anzupassen.

221 Natürlich gab es das auch schon früher, aber nur für sehr, sehr viel Geld!
222 In Mu muss „Python“ als Modus gewählt werden. Dann steht der Button „Debuggen“ bereit. Leider sind die Möglichkeiten, Variablen detailliert zu betrachten, sehr eingeschränkt. Andere IDEs, wie Thonny, Vscode usw. leiden nicht unter solchen Begrenzungen.

X Der Neue

Ende 2024 kam ein neuer Pi Pico unter dem Namen Pi Pico2 auf den Markt. Seit Ende Januar 2025 habe ich auch einen!

Die Highlights sind eine höhere Taktfrequenz, mehr PWM Kanäle, leistungsfähigere ARM Kerne und – wichtig für Micropython – eine Verdoppelung des RAM.

Das Programm:

```
pin = machine.Pin(15, machine.Pin.OUT)
while True:
    pin.toggle()
```

schaltet den GPIO 15 mit der höchsten Geschwindigkeit an und aus, die Micropython ohne Hardwareunterstützung liefern kann. Während der „alte“ Pi Pico ein 79 kHz Rechtecksignal ausgab, liefert der Pi Pico2 145 kHz.

Meine Messungen ergaben auch eine Erhöhung der maximalen PWM Frequenz auf 75 MHz (gegenüber 62,5 MHz).

Interessant ist auch die Stromaufnahme. Der Pi Pico verbraucht 17 mA im interaktiven Modus. Läuft die Schleife, werden es 19 mA. Der Pi Pico2 verbraucht 14 mA, bei laufender Schleife erhöht sich auf meinem Messgerät[223] die Stromaufnahme nicht.

Der Preis ist mit 7,90 € (in der WLAN Ausführung) auch nicht viel teurer als für den Vorgänger.

223 Die kleinste Auflösung meines USB Messgeräts ist 1 mA; Änderungen des Stroms werden also erst bei mehr als einem Milliampere sichtbar.

XI Alternative Sprachen und Entwicklungsumgebungen

Ich finde Python und natürlich Micropython sehr gut geeignet, um damit Programme zu schreiben. Nach meiner Erfahrung reicht die Geschwindigkeit auf dem PC oder Laptop auch für anspruchsvolle Applikationen aus. Meine Sensoren für die Heimautomation sind alle mit Micropython entwickelt worden.

Es gibt aber Situationen, wo man das letzte Quäntchen an Geschwindigkeit herausholen muss. Wer bis hierher im Buch gekommen ist, weiß, dass das nicht ohne Sprachen wie C, C++, Rust oder eben auch „Arduino“ geht. Letzteres stellt keine Sprache dar, aber man schreibt C oder C++ ähnlichen Code in einem Arduino Dialekt.

Arduino macht es dem Programmierer leichter, weil es über die IDE alle Schritte, vom Editor über kompilieren und flashen des Mikrocontrollers steuert. Ein weiterer Vorzug besteht in der riesigen Anzahl fertiger Bibliotheken, die man einfach in seine eigenen Programme integrieren kann. Als Beispiel sei nur MQTT genannt, für das es gleich mehrere Bibliotheken gibt.

Auch für Rust gibt es ein gut funktionierendes Ökosystem, um Mikrocontroller-Programme zu schreiben und sogar auf dem Controller zu debuggen. Die Einstiegshürde für Rust liegt aber recht hoch.

Leider sprengt auch das den Rahmen dieses Buches.

XII Anhang

1 Automatisiertes Cross-Kompilieren und Übertragen auf den Mikrocontroller

Für beide Varianten müssen mit Hilfe von „pip“ die Module „mpremote“ und „mpy-cross“ installiert werden. Beide Programme (cross.bat und cross.sh) stehen auch auf der Stadtbibliothek Seite[224] zum Download bereit.

1.1 Für Windows als Batch Datei (Endung „bat“)

```
REM cross.bat
REM Cross-Kompilierung aller *.py Dateien im aktuellen Verzeichnis
REM und Uebertragung auf den Pi Pico oder andere Mikrocontroller.
REM Die entsprechenden *.py Dateien auf dem Mikrocontroller werden gelöscht!!!

@echo  off

REM Default Anschluss
set device=com8

if "%1" == "-d" set device=%2
echo Using device %device%
forfiles /M *.py /C "cmd /C mpy-cross @FILE"
forfiles /M *.py /C "cmd /C mpremote connect %device% rm :@FILE"
forfiles /M *.mpy /C "cmd /C mpremote connect %device% cp @FILE :@FILE"
```

1.2 Für Linux

```
#!/usr/bin/bash
# cross.sh
# Cross-Kompilierung aller *.py Dateien im aktuellen Verzeichnis
# und Uebertragung auf den Pi Pico oder andere Mikrocontroller.
# Die entsprechenden *.py Dateien auf dem Mikrocontroller werden gelöscht!!!
# Mit -C kann die Cross-Kompilierung abgeschaltet werden; dann werden
# die *.py Dateien auf den Mikrocontroller übertragen.

device="a0" # Abgekürzt für /dev/ttyACM0

NOCROSS=0
mppath=""

usage() {
    echo "Benutzung: [-C|?|h|p:] [-d <Geräte Pfad> oder Abkürzung wie a0, a1
usw.] <Dateien>"
    echo "             -C ohne Cross-Compilierung"
    echo "             -p Pfad zu mpremote und mpy-cross"
    echo "             -?|h Hilfe"
}

while getopts ":d:C?hp:" opt
do
    #echo "OPT" $opt $OPTIND
    # Optionales Argument in Variable $OPTARG
    echo "OPT $opt"
```

224 https://bibliothek.velbert.de/angebote/elektronik-workshop

```
    case ${opt} in
        d) #
            device=($OPTARG)
            ;;
        C) # keine Cross-Kompilierung
            NOCROSS=1
            ;;
        p) # keine Cross-Kompilierung
            mppath=($OPTARG)
            ;;
        ?|h) #
            if [[ $OPTARG != "" ]]
            then
                echo "Falsche Option \"$OPTARG\""
            fi
            usage
            exit 2
            ;;
    esac
done
shift $(($OPTIND – 1))

mpremote="${mppath}/mpremote"
mpycross="${mppath}/mpy-cross"
files="*.py"
processedfiles=""
mpyfiles=""
let total=0
echo "Benutze Gerät" $device
if [[ "$*" == "" ]]
then
    txt="alle .py Dateien im Verzeichnis"
else
    txt="Datei(en): $*"
    files="$*"
fi
echo "Verarbeite $txt"
echo

if [[ $NOCROSS -eq 0 ]]
then
    for i in $files
    do

        ls -1 $i
        if [[ $? -gt 0 ]]
        then
            echo "No such file $i"
            continue
        else
            processedfiles="$processedfiles $i"
        fi
        echo "Compile" $i
        $mpycross -O3 $i
        if [[ $? -gt 0 ]]
        then
            echo "Cross-Kompiler Fehler"
            exit 1
        fi
        mpyfiles="$mpyfiles ${i%.*}.mpy"
    done
```

```
    #echo "PRO $processedfiles"
    for i in $processedfiles
    do
        if [[ $(basename "$i") != "main.py" ]] # ignoriere main.py
        then
            echo "Entferne Datei $i"
            $mpremote $device rm :$i > /dev/null
            echo "RC $?"
        fi
    done

    for i in $mpyfiles
    do
        echo "Kopiere Datei $i zu Gerät $device"
        $mpremote $device cp $i :$i
        rc=$?
        let total=$total+$rc
        echo "RC $rc"
    done
else
    for i in $files
    do
        ls -1 $i
        if [[ $? -gt 0 ]]
        then
            echo "No such file $i"
            continue
        fi
        echo "Kopiere *.py Datei $i zu Gerät $device"
        $mpremote $device cp $i :$i
        rc=$?

        let total=$total+$rc
        echo "RC $rc"
    done
fi

echo "Gesamt Fehler:" $total
```

XIII Glossar

Begriff	Erläuterung
ADC	Analog Digital Konverter: Umwandlung eines analogen Spannungssignals in einen digitalen Wert
asynchron	das Eintreten eines Ereignisses unabhängig vom normalen Programmlauf
AsyncIO	Name des Kooperativen Multitaskings für Python
Closure	Eine anonyme Funktion, die sich den Kontext zur Zeit ihrer Erstellung merkt
CPU	Die Recheneinheit eines Mikrocontrollers
DAC	Digital zu analog Konverter: Umwandlung eines digitalen Werts in eine analoge Spannung. Auf dem Pi Pico nur mit PWM und Filter möglich.
DHT22	Ein Temperatur und Feuchte Sensor
I2C	Ein Schnittstellenprotokoll für die Interaktion von einem Mikrocontroller mit anderen Geräten wie Displays, Sensoren usw., die im Gegensatz zu SPI mehrere Geräte parallel an den gleichen Leitungen ansprechen kann.
IDE	Integrierte Entwicklungs-Umgebung (Integrated Developtment Environment)
Interrupt	Eine asynchrone Unterbrechung des laufenden Programms durch ein Ereignis: Signal an GPIO, Timer usw.
IoT	Internet of Things, die gezielte Nutzung des Internet zur Vernetzung von Geräten
Iterator	Anstatt eine Liste von Daten zu berechnen, wird jeweils nur das nächste benötigte Datum ermittel
Lambda	Eine Namenlose Funktion. Kann überall im Programmcode auftauchen, wo auch ein Wert hingeschrieben werden könnte
Latenz	Allgemein eine unerwünschte Verzögerung zwischen dem Signal, das als Auslöser dient, und der Reaktion darauf
LED	Eine Diode (Strom fließt nur in eine Richtung), die leuchtet, wenn Strom fließt
literal	Ein Informatik-Begriff, der bedeutet, dass ein Wert direkt im Programmtext steht, z.B. weise ich einer Variablen den Wert 4711 zu, indem ich die Zeichen „4711“ in den Programmtext schreibe
Micropython	Eine von Python abgeleitete Programmiersprache für Mikrocontroller
Mikrocontroller	Bezeichnet eine spezielle CPU für eingebettete System oder eine solche CPU auf einem kleinen Board mit den notwendigen zusätzlichen Bauelementen
Minicomputer	Ein sehr kleiner Computer, z.B. wie der Raspberry Pi
MQTT	Ein Internet Protokoll zur Vernetzung kleiner Geräte
Multiplex	Die Übertragung von Informationen durch zeitlich gestaffelte Daten über die gleichen Leitungen
Multitasking	Zeitlich paralleles oder quasi-paralleles abarbeiten von Programmcode in einem System
Paradigma	Bezeichnet eine grundsätzliche Denk- oder Verfahrensweise
Pi Pico	Ein Mikrocontroller-Entwicklungsboard der Raspberry Pi Foundation
Potentiometer	Ein regelbarer Widerstand mit meist drei Anschlüssen; zwei Anschlüsse bilden die Enden eines Widerstands, der Dritte lässt sich elektrisch stufenlos vom einen zum anderen verändern
PWM	Puls-Weiten-Modulation. Der Pi Pico verfügt über 16 PWM-Kanäle. Achtung: alle haben die gleiche Frequenz, auch wenn man für jeden Kanal die Frequenz einstellen kann.
Python	Eine Programmiersprache, die direkt Quelltext und direkte Eingaben (interaktiv) verarbeiten kann. In der heute gängigen Version 3 unterstützt sie Objektorientierte Programmierung und Funktionale Programmierung
Race Condition	In der Informatik wird damit ausgedrückt, dass die zufällige Reihenfolge der Verarbeitung in mehreren zeitlich parallelen Programmen zu ungewünscht zufälligen Ergebnissen führt
Referenz	In anderen Programmiersprachen oft auch als Zeiger (Pointer) bezeichnet, in Python ein Verweis auf Daten
RS232	Eine auf das UART Protokoll aufsetzende Hardwareschnittstelle, In PCs als serielle Schnittstelle bezeichnet. Heute veraltet
SPI	Ein Schnittstellenprotokoll für die Interaktion von einem Mikrocontroller mit anderen Geräten wie Displays, Sensoren usw.
Thonny Editor	Eine IDE für Mikrocontroller
Typ / Datentyp	Die Art des Wertes: Ganzzahl, Fließkommazahl, Liste, String usw.
Typisiert	Festlegung in bestimmten Programmiersprachen, dass eine Variable nur einen Typ beinhalten kann
UART	Ein serielles, bidirektionales Schnittstellenprotokoll
Variable	Der Name für irgendeinen im Programm zu speichernden Wert