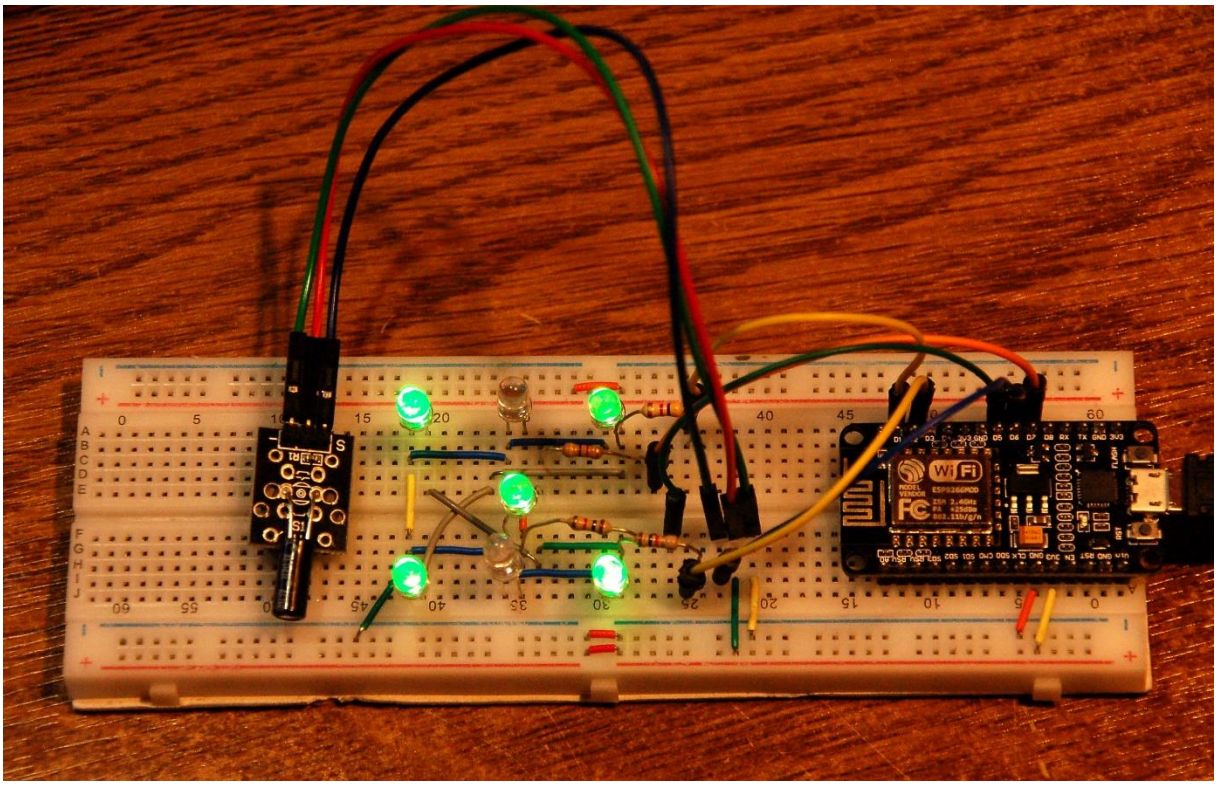

Schüttelwürfel mit einzelnen LEDs

Diese Anleitung ist auch als PDF-Dokument verfügbar. Schaltung und Programm sind besonders für Anfänger geeignet.

Würfelschaltungen gibt es in der Elektronik eine ganze Menge, diskret mit ICs der 74-er-Serie oder CMOS-Chips, mit AVR-Controllern, PICs etc. Die Darstellung der Augenzahl passiert üblicherweise mit sieben LEDs. Bei diskreten Schaltungen zählt ein Ringzähler sechs Zustände so schnell durch, dass Auge und Gehirn der Anzeige nicht folgen können. Die sechs binären Codes von 0x00 bis 0x05 müssen für die Anzeige mit den sieben LEDs umcodiert werden. Das erfordert eine Wahrheitstabelle und einen Haufen logischer Gatter. Ich gehe heute mit Ihnen den Weg zu einer Komfortlösung über drei Stationen unter Mitwirkung eines ESP32 oder eines ESP8266. Die Codierung wird in der Programmiersprache MicroPython erfolgen. Damit willkommen zu einer neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der Schüttelwürfel

Wir beginnen mit einer ganz einfachen Schaltung, die noch einen Ringzähler benutzt, natürlich in ein MicroPython-Programm übersetzt. Der 6-zu-7-Decoder wird teils

schaltungstechnisch, teils programmtechnisch gelöst. Werfen wir einen Blick auf die Schaltpläne für ESP32 und ESP8266.

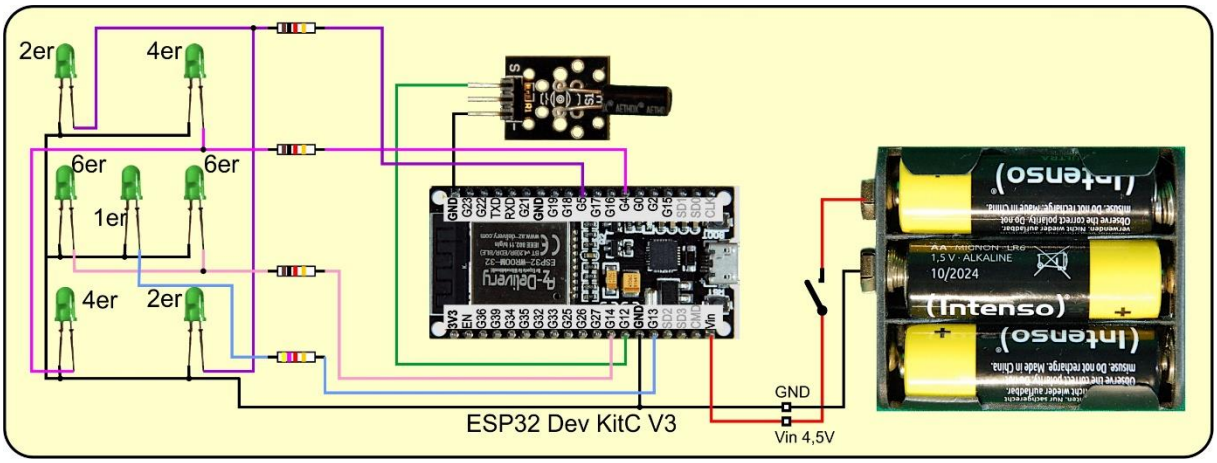


Abbildung 1: Würfel mit LEDs uns ESP32

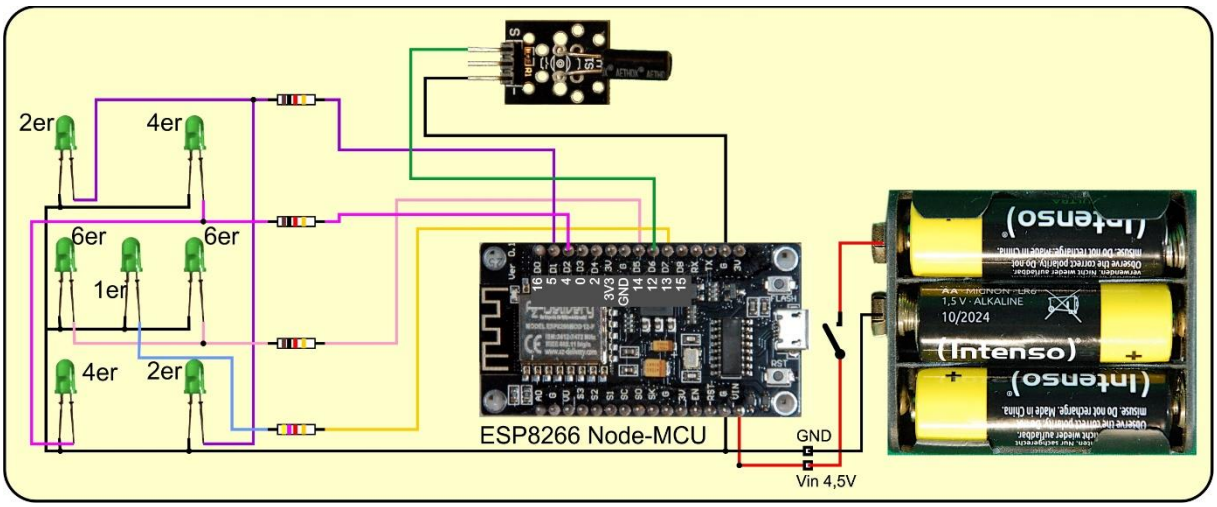


Abbildung 2: Würfel mit LEDs und ESP8266

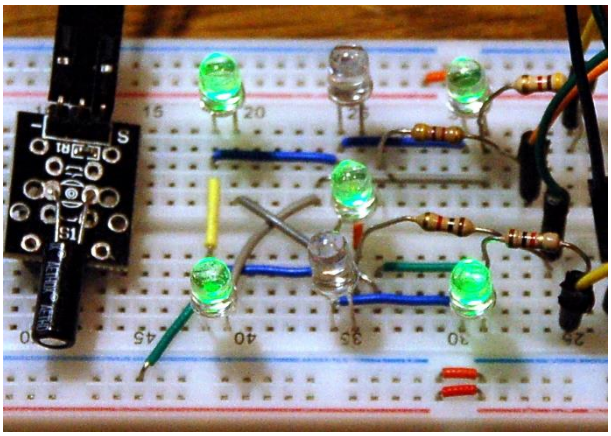


Abbildung 3: LED-Verdrahtung

Drei Zweiergruppen und eine einzelne LED bilden die Anzeige. Durch die geschickte Verbindung der diagonalliegenden LEDs lassen sich mit zwei GPIO-Pins (5 und 4) die Werte 2 und 4 darstellen. Wenn wir die mittlere LED an GPIO13 mit dazunehmen, sind wir zusätzlich schon bei 1, 3 und 5. Die beiden mittleren LEDs in

den Außenreihen ergeben mit den Ecken die 6. Wir brauchen also nur vier Leitungen für sechs Zustände.

Schüttelwürfel habe ich das Projekt deshalb genannt, weil ein Rüttelkontakt oder Neigungsschalter den schon angesprochenen Ringzähler aktiviert, sobald er geschlossen wird.

Während der Entwicklung wird die Schaltung über das USB-Kabel mit Energie versorgt. Im Produktionsbetrieb sorgt ein 4,5V-Block oder ein Batteriehalter mit drei AA-Zellen für die nötige Betriebsspannung.

Hardware

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F oder ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	MAX7219 8x8 1 Dot Matrix MCU LED Anzeigemodul
1	4,5V - Batterie
diverse	Jumperkabel
1	Minibreadboard oder Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins
1	KY-002 Erschütterung Schalter Schock Sensor oder KY-020 Neigung Schalter

ESP32 oder ESP8266 sind beide mit einer Ausnahme gleichermaßen für dieses Projekt geeignet. Die Ausnahme ist der ESP8266-01, weil der einfach zu wenig herausgeführte GPIO-Leitungen hat.

Für Anfänger, die mit dem Löten nicht so vertraut sind, empfehle ich die verlöteten Controller-Boards. Tipps zum Umgang mit dem Lötkolben gibt es hier.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP8266/ESP32:

MicropythonFirmware
Bitte eine Stable-Version aussuchen
ESP8266 mit 1MB Version 1.18 Stand: 25.03.2022 oder
ESP32 mit 4MB Version 1.18 Stand 25.03.2022

Die MicroPython-Programme zum Projekt:

matrix8x8.py Treibermodul für den MAX7219
spibus.py Portabfrage und SPI-Bus Instanzierung
wuerfel.py
wuerfel1.py
wuerfel2.py
wuerfel3.py

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 05.02.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Kontaktaufnahme mit (Micro)-Python

Wenn sie bereits mit der Arduino-IDE gearbeitet haben, hatten sie vielleicht hin und wieder das Verlangen einen Befehl auszuprobieren, um dessen Verhalten zu untersuchen. Aber dort ist eine interaktive Arbeit mit dem System nicht vorgesehen, ich habe das weiter oben schon erwähnt.

MicroPython als Abkömmling der Interpretersprache C-Python verhält sich da anders. Sie arbeiten in einer IDE, einer integrierten Entwicklungs- Umgebung, mit direktem Draht zu einem Python-Interpreter. In Python nennt sich dieses Tool **REPL**. Das ist ein Akronym für Read – Evaluate - Print – Loop. Sie machen am Terminal eine Eingabe, der MicroPython-Interpreter wertet die Eingabe aus, gibt eine Antwort zurück und wartet auf die nächste Eingabe. Die Eingabeaufforderung von Windows, früher MS-DOS, arbeitet nach dem gleichen Schema.

Es gibt eine ganze Reihe von IDEs, Thonny, µPyCraft sind zwei davon, Idle und MU sind zwei weitere. Ich arbeite gerne mit Thonny, weil es außer dem Terminal (REPL) und einem komfortablen Editor zum Erstellen und Testen von Programmen auch noch über 12 Assistenten, einen Plotter und ein Tool zum Flashen der Firmware verfügt. Mit REPL ist es ein Leichtes, jede Anweisung einzeln zu testen. Wir werden das an verschiedenen Stellen in dieser Anleitung ausprobieren.

Noch etwas ist bei Python anders als bei C oder anderen Compiler-Sprachen. Es gibt keine Typfestlegung beim Deklarieren von Variablen oder Funktionen. MicroPython findet den Typ selbst heraus. Das testen wir doch gleich einmal. Starten Sie schon mal Thonny. **Eingaben** am REPL-Prompt >>> formatiere ich im Folgenden **fett**, die *Antworten* vom System *kursiv*.

Bezeichner für Variablen, Konstanten, Funktionen und andere Objekte beginnen stets mit einem Buchstaben oder mit einem Unterstrich, nie mit einer Ziffer. MicroPython ist case sensitive, es unterscheidet also Groß- Kleinschreibung. Hier weise ich der Variable x den Wert 129 zu, und der Interpreter erkennt den Typ Ganzzahl (integer oder int), der Typ von y ist float, also Fließkommazahl.

>>> x = 129
>>> type(x)
<class 'int'>

>>> y=23.7
>>> type(y)
<class 'float'>

Alles in MicroPython ist ein Objekt. x ist ein Objekt der Klasse int, und y eines der Klasse float. Genau genommen sind die Bezeichner x und y nur Referenzen auf die eigentlichen Daten, also die Zahlenwerte im Speicher. Es gibt eine ganze Reihe von "Geheimnissen" zum Thema Variablen, aber das soll fürs Erste an Infos zu den Hintergründen genügen.

Natürlich gibt es auch Zeichenketten oder Strings.

>>> t="Hallo Freunde!"
>>> type(t)
<class 'str'>

Ein ganz besonderer Datentyp ist **None**. Ein Objekt, auch Instanz genannt, davon verweist auf nichts. Man verwendet None immer dann, wenn man eine Variable deklarieren, aber deren Wert noch nicht festlegen möchte.

>>> r=None
>>> type(r)
<class 'NoneType'>

Nach den ersten Gehversuchen erstellen wir ein Programm, das mit einem ESP8266 oder ESP32, einem Neigungsschalter und 7 LEDs einen Würfel simuliert.

Das Programm wuerfel.py

Die verwendeten Anschlüsse am ESP32 und ESP8266 sind, von MicroPython aus gesehen, identisch, daher kann das Programm wuerfel.py auf den Controllern beider Familien ohne Änderung betrieben werden. Allerdings sind die Pins auf den ESP8266-boards anders benannt. Hier ist die Übersetzung, die Sie auch in den Programmen finden.

```
# Pintranslator fuer ESP8266-Boards
# LUA+Arduino-Pins D0 D1 D2 D3 D4 D5 D6 D7 D8
# ESP8266 Pins     16  5  4  0  2 14 12 13 15
```

Genötigt werden fünf GPIO-Pins. Der Rüttelkontakt liegt an Pin 12, der als Eingang programmiert wird. Damit das funktioniert, muss natürlich zuvor die Klasse **Pin** aus dem Modul **machine** importiert werden. Vier weitere GPIOs dienen als Ausgänge zu den LEDs. Module enthalten MicroPython-Code im Klartext und dienen unter anderem der Anbindung von externer Hardware. Ihr Gegenstück in der Arduino-IDE sind die Libraries oder Bibliotheken.

Objekt/Anschluss	ESP	Bemerkung
LED einser	GPIO13	Ausgang
LED zweier	GPIO5	Ausgang
LED vierer	GPIO4	Ausgang
LED sechser	GPIO14	Ausgang
+ Batterie	Vin	
- Batterie	GND	
kontakt	GPIO12	Eingang
Schalter Mitte	3,3V	
Schalter -	GND	

Die Platine des Neigungsschalters enthält einen Widerstand R1 von 10kOhm, den wir aber nicht benutzen. Der andere Schaltkontakt liegt an Masse oder GND. Der +Vcc-Anschluss in der Mitte bleibt unbeschaltet. Abbildung 3 illustriert den Sachverhalt.

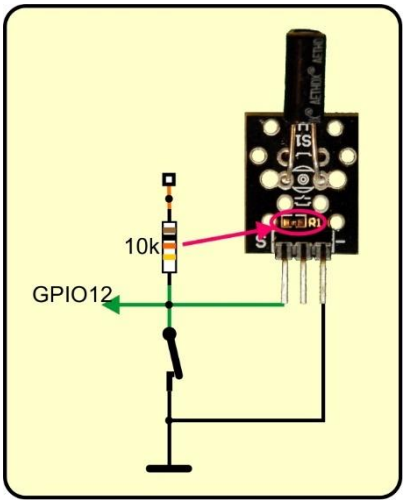


Abbildung 4: Funktion des Neigungsschalters KY-002

In der dargestellten Lage des Moduls ist der Schalter geschlossen. GPIO12 liegt daher auf GND-Potenzial. Das entspricht einer logischen 0. In diesem Zustand läuft unser Zähler. Stellt man den Neigungsschalter auf den Kopf, wird der Kontakt geöffnet, was einer logischen 1 entspricht. Den 10kOhm-Widerstand auf dem Platinchen können wir leider nicht verwenden, denn dadurch würde GPIO12 bereits beim Booten des Controllers fest gegen +3,3V gezogen. Die Folge davon wäre, dass der Controller nicht starten kann. Andererseits muss GPIO12 aber an +3.3V liegen, wenn der Schalter geöffnet ist, weil wir sonst nicht zuverlässig eine 1 einlesen.

Die Lösung des Problems ist einfach. Die ESPs haben für jeden Eingang einen eigenen, internen Pullup-Widerstand, den man ein- und ausschalten kann. Und den schalte ich für GPIO12 erst zu, wenn der Bootvorgang schon abgeschlossen ist, nämlich im Programm.

Das Programm sieht bislang folgendermaßen aus. Alle Ausgänge werden nach der Deklaration sofort auf Null gesetzt, die LEDs sind alle aus.

```
from machine import Pin

kontakt=Pin(12,Pin.IN, Pin.PULL_UP)
einser=Pin(13,Pin.OUT,value=0)
zweier=Pin(5,Pin.OUT,value=0)
vierer=Pin(4,Pin.OUT,value=0)
sechser=Pin(14,Pin.OUT,value=0)
```

Als nächstes definiere ich eine Liste. Das kommt einem Array in der Arduino-IDE gleich. In MicroPython kann eine Liste beliebige Elemente enthalten, ohne dass man den Typ festlegen muss.

>>> from machine import Pin
>>> L=[1, 2.34, "ein String", Pin(12)]
>>> L
[1, 2.34, 'ein String', Pin(12)]

Meine Liste **led** enthält die vier Pin-Objekte für die LED-Steuerung, die ich grade definiert habe. Dadurch kann ich über den Index, die Platznummer, der Elemente in einer for-Schleife auf sie zugreifen. Die Indizierung von Listen beginnt in MicroPython mit der Platznummer 0. **vierer** enthält das Pin-Objekt **Pin(4)** und steht an Position 2.

```
led=[einser,zweier,vierer,sechser]
```

>>> led=[einser,zweier,vierer,sechser]
>>> led[2]
Pin(4)

Damit ich die beiden Zeilen der for-Schleife nicht sechsmal ins Programm schreiben muss, definiere ich vorab eine Funktion mit dem Namen switch. Als Eingabe dient der Parameter **state**, den ich in die runden Klammern hinter dem Namen schreibe. Eingeleitet wird die Funktionsdefinition durch das reservierte Wort def. Den Abschluss der Kopfzeile bildet ein Doppelpunkt.

Alle Zeilen des Funktionskörpers, der nach der Kopfzeile folgt, werden um zwei (bei uPyCraft) oder vier Zeilen (bei Thonny) eingerückt. Anhand dieser Indentation (Einrückung) kann MicroPython erkennen, wann die Definition des Funktionskörpers zu Ende ist. Die nächste Zeile, die im umgebenden Programm folgt, ist nämlich wieder um 2 oder 4 Zeichenpositionen ausgerückt. Die Indentation in MicroPython entspricht dem Paar geschweifter Klammern in der Arduino-IDE.

```
def switch(state):
    for i in range(4):
        led[i].value(state[i])
```

Eine Schleife ist programmtechnisch gesehen, ein Block von Anweisungen, die mehrfach durchlaufen werden können. Das erhöht die Flexibilität und spart Programmspeicherplatz, der bei Micro-Controllern nicht gerade üppig ist.

Die Zählschleife mit for verwendet die Variable i als Laufindex, der automatisch mit 0 beim ersten Durchlauf startet und mit jedem weiteren Durchlauf um jeweils 1 erhöht

wird. Das regelt das reservierte Wort **range**. Die 4 begrenzt den Bereich **ausschließend**. Das heißt, der letzte Wert, den i annimmt, ist 3. Die Schleife wird also genau 4-mal durchlaufen. Die Kopfzeile aller Befehlsstrukturen in MicroPython enden mit einem Doppelpunkt, so auch die Kopfzeile der for-Schleife. Danach wird stets eingerückt.

Was macht jetzt diese Zeile?

```
led[i].value(state[i])
```

Nun, **led**[i] pickt sich das i-te Pin-Objekt aus der Liste **led**, **led**[0] ist Pin(13) bis **led**[3] das ist Pin(14). Mit der Methode **value**() können wir den Zustand eines Pin-Objekts abfragen oder wie hier mit **value(state[i])** setzen. Und zwar wird der Zustand dem sogenannten Tuple **state** an der Position i entnommen.

Ein Tuple ist eine ähnliche Datenstruktur wie eine Liste. Während die Werte von **Elementen in Listen verändert** werden können (Listen sind mutable), sind nach der Deklaration die **Elemente von einem Tuple unveränderlich** (Tuples sind immutable). Auch Tuples können aber, wie Listen, Elemente von unterschiedlichem Typ aufnehmen. Und wie eine Liste, ist ein Tuple ein sequenzieller, geordneter Datentyp. Das bedeutet, dass die Elemente stets in der Reihenfolge gelistet werden, in der sie deklariert wurden. Zum besseren Verständnis kommen hier ein paar experimentelle Zeilen, die mit unserem Programm nichts zu tun haben. Dessen weitere Besprechung greife ich gleich danach wieder auf.

Beispiel Liste:

```
>>> a=4
>>> L=[5.689,23,a,"test",a+16]
>>> L
[5.689, 23, 4, 'test', 20]    # Wert von a wird übernommen nicht der Bezug (Referenz)
>>> a=9    # Veränderung hat keinen Einfluss auf die Werte in der Liste
>>> L
[5.689, 23, 4, 'test', 20]    #
>>> L[3]
'test'
>>> L[2]=a    # Listenelement 2 erhält einen neuen Wert
>>> L
[5.689, 23, 9, 'test', 20]
```

Beispiel Tuple

```
>>> a=4
>>> t=(2.34,5,a,a+3)    # Wert von a wird übernommen nicht der Bezug (Referenz)
>>> t
(2.34, 5, 4, 7)
>>> a=9    # Veränderung hat keinen Einfluss auf die Werte in der Liste
>>> t
(2.34, 5, 4, 7)
```

>>> t[3]
7
>>> t[1]=12.7 # Elemente im Tuple können nicht verändert werden
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item assignment

Zurück zum Programm. Es fehlt noch der Ringzähler, der letztlich die Ausgänge für die LEDs steuern soll.

```
n=1
while True:

    if kontakt.value()==0:
        n= n+1 if n<6 else 1
        if n == 1:
            switch((1,0,0,0))
        if n == 2:
            switch((0,1,0,0))
        if n == 3:
            switch((1,1,0,0))
        if n == 4:
            switch((0,1,1,0))
        if n == 5:
            switch((1,1,1,0))
        if n == 6:
            switch((0,1,1,1))
```

In n lege ich die Augenzahl des Würfels ab, sie startet mit 1. Dann geht es in die while-Schleife.

Nach dem reservierten Wort oder auch Schlüsselwort **while** steht als Bedingung ein Ausdruck, der bei der Auswertung **True** (wahr) oder **False** falsch) ergibt. True und False sind Wahrheitswerte oder auch **boolsche Werte**. Dem Datentyp **bool** stehen nur diese beiden Werte zur Verfügung. Die while-Schleife wird solange durchlaufen, bis die Bedingung False liefert. Das wird aber nie passieren, denn True ist und bleibt True. Wir haben es hier mit einer Endlosschleife zu tun. Weil der Schleifenkörper den Hauptteil des Programms enthält, nennt man diese Schleife **Hauptschleife** oder **Mainloop**.

if oder wenn leitet eine Programmstruktur ein, die dazu dient, einen Anweisungsblock einmalig auszuführen, falls der Ausdruck nach dem Schlüsselwort **if True** liefert. Der Ausdruck ist hier ein Vergleich. Ich lese den Status des Neigungsschalters ein und prüfe, ob der Wert gleich 0 ist. Der Vergleichsoperator besteht aus zwei "="-Zeichen. Das wird oft übersehen und führt dann zur Meldung eines Syntaxfehlers, dessen Ursache und Lokalität, gerade am Anfang, nicht sofort gesehen wird. Weitere Vergleichsoperatoren sind <, <=, >, >= und !=. Das letzte Symbol steht für ungleich. Auch die if-Zeile wird mit einem Doppelpunkt abgeschlossen. Zur höchsten Ausbaustufe des if-Konstrukts komme ich bei Besprechung des übernächsten Programms wuerfel3.py.

Wenn der Schalter nicht den Wert 0 sondern eine 1 liefert, dann ist der Wert des Ausdrucks False, und es passiert gar nichts, es startet sofort der nächste Durchlauf. War der Schalter im Moment der Abfrage geschlossen, dann landen wir bei der ersten Anweisung des if-Blocks, die natürlich – ganz recht - eingerückt ist.

```
    n= n+1 if n<6 else 1
```

Die Zeile enthält einen bedingten Ausdruck (conditional expression) und ist syntaktischer Zucker (syntactic sugar, Peter J. Landin um 1960), das heißt wir haben hier eine Schreibweise, die mehrere Codezeilen in einer Zeile zusammenfasst. Im Langtext würde das so ausschauen:

```
if n<6:
   n = n +1
else:
   n = 1
```

Auch in der else-Zeile steht der Doppelpunkt, der nachfolgende Block wird eingerückt. Beide Schreibweisen sind eine Erweiterung des einfachen if-Konstrukts und in ihrer Wirkung identisch. Die Sprechweise gibt genau die Verhaltensweise wieder.

Erhöhe n um 1, **wenn** n kleiner als 6 ist, **sonst** setzte n auf 1

```
>>> n=5
>>> n=n+1 if n<6 else 1
>>> n
6
>>> n=n+1 if n<6 else 1
>>> n
1
```

Im bedingten Ausdruck fallen übrigens die Doppelpunkte weg!

n hat also jetzt einen Wert zwischen 1 und 6 inklusive der Intervallgrenzen. Je nach Wert müssen die LEDs ein- oder ausgeschaltet werden. Das erledigt die Funktion **switch**(), der wir ein Tuple mit den Schaltzuständen der Pin-Objekte als Argument übergeben. Die Reihenfolge entspricht genau derjenigen der Elemente der Liste **led**. Tuples werden stets mit runden Klammern geschrieben. Auch beim Aufruf von Funktionen stehen die Argumente in runden Klammern. Das führt hier dazu, dass die Zahlenfolge der Schaltzustände in zwei Paaren runder Klammern eingefasst ist, das innere Paar für das Tuple, das äußere für die Argument-Liste. Die Vierer-Tuples werden beim Aufruf von **switch**() dem Parameter **state** übergeben. Was die Funktion damit macht, habe ich ja weiter oben schon beschrieben.

Haben Sie die Schaltung zusammengesteckt und am PC angeschlossen? Dann geben Sie jetzt den Programmtext in einem Editorfenster von Thonny ein oder laden Sie die Datei wuerfel.py herunter und öffnen Sie sie in einem Editorfenster. Mit F5 starten Sie das Programm.

Solange Sie den Neigungsschalter aufrecht halten, leuchten alle sieben LEDs. Das Ein- und Ausschalten bei den verschiedenen Augenzahlen geht so schnell, dass Auge und Gehirn dem nicht folgen können. Wir nehmen ein leicht flackerndes, allgemeines Leuchten wahr. Stellen wir den Schalter auf den Kopf, bleibt der letzte Zählzustand erhalten und die LEDs zeigen das übliche Punktemuster einer Würfelseite.

Der Zufall und das chaotische Verhalten des Würfels, auf die es ja beim Würfeln gerade ankommt, schlagen sich bei diesem ersten Programm lediglich in der Zeitdauer nieder, die wir den Neigungsschalter aufrecht halten. Der Zählvorgang ist alles andere als chaotisch. Das will ich in einem zweiten Ansatz ändern.

Dem Zufall auf die Sprünge geholfen

Das zweite Programm hat, genau wie das erste, 39 Codezeilen, Kommentare und Leerzeilen inbegriffen. Das kommt nicht von ungefähr, denn ich habe lediglich eine Zeile durch eine andere ersetzt, eine Importzeile hinzugefügt und die Zeile mit der Vorbelegung von n weggelassen.

Aus

n= n+1 if n<6 else 1

wird

n=random.getrandbits(8) % 6 + 1

MicroPython bietet ein Modul zur Erzeugung von Zufallszahlen (random numbers). In der Importabteilung nehme ich dieses Modul mit ins Programm.

```
from machine import Pin
import random
```

Den Teil bis zur while-Schleife kennen wir schon, die Zeile mit n = 1 habe ich, wie schon erwähnt, eingestampft. Die while-Schleife bleibt erhalten, bis auf diese Zeile:

n=random.getrandbits(8) % 6 + 1

Sobald und solange der Neigungsschalter geschlossen ist, wird jetzt tatsächlich gewürfelt und zwar eine zufällig ermittelte Zahl mit 8 Bits, also ein Byte. Aus diesem Wert berechne ich den Teilungsrest zum Teiler 6. Also wird zum Beispiel aus 0b01101110 = 0x6E = 110 der Wert 2. Denn 110 : 6 = 18 Rest 2. 0b01101110 ist die Darstellung der Zahl 110 im Binärsystem, 0x6E ist das Äquivalent im Hexadezimalsystem. Die Teilungsreste modulo 6 bewegen sich im Bereich 0..5. Ich brauche aber den Bereich von 1 bis 6, also wird kurzerhand eine 1 addiert. Mit diesem Wert für n gehen wir weiter zur Anzeige.

Für einen neuen Test starten Sie das Programm wuerfel1.py in einem neuen Editorfenster.

Aus 1 mach 2

Wenn wir schon die seriellen Datentypen Liste und Tuple behandelt haben, dann nutzen wir doch gleich noch deren Vorteile, um bei gleicher Performanz das Programm wuerfel1.py um 10 Zeilen zu kürzen. Hier ist das Listing.

```
# wuerfel2.py
#
# Pintranslator fuer ESP8266-Boards
# LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
# ESP8266 Pins 16  5  4  0  2 14 12 13 15
#                 SC SD

from machine import Pin
import random

kontakt=Pin(12,Pin.IN)
einser=Pin(13,Pin.OUT,value=0)
zweier=Pin(5,Pin.OUT,value=0)
vierer=Pin(4,Pin.OUT,value=0)
sechser=Pin(14,Pin.OUT,value=0)
#
led=[einser,zweier,vierer,sechser]
image=((1,0,0,0),(0,1,0,0),(1,1,0,0),(0,1,1,0),(1,1,1,0),(0,1,
1,1))

def switch(state):
    for i in range(4):
        led[i].value(state[i])

while 1:
    if kontakt.value()==0:
        n=random.getrandbits(8) % 6
        switch(image[n])
```

Was liegt näher, als die Schaltinformationen für die LEDs in einem Tuple zu sammeln. Der Datentyp **tuple** arbeitet resourcenschonender als der Typ **list**. Weil ich nur die Fähigkeit eines geordneten, seriellen Zugriffs auf die Statuselemente benötige, genügt der Einsatz eines Tuples. Aus demselben Grund habe ich für die Angabe der Schaltzustände der LED-Gruppen auch den Typ **tuple** verwendet. Übrigens, wiederum aus demselben Grund hätte ich für **led** statt des Typs **Liste** auch den Typ **tuple** nehmen können, aber dann hätte ich keinen Grund gehabt, Ihnen den Datentyp Liste vorzustellen.

Der Würfel vom Feinsten

Sicher haben Sie eine gewisse Steigerung in den Anforderungen festgestellt. Was jetzt kommt greift noch einmal tiefer in die Fundus-Kiste von MicroPython.

Schaltungstechnisch wird es einfacher und optisch schöner, programmtechnisch wird es ein bisschen anspruchsvoller, aber keine Sorge, Schritt für Schritt kriegen wir das hin. Beginnen wir wieder mit den Schaltplänen für ESP32 und ESP8266. Für den

Matrixwürfel habe ich als Controller einen ESP8266 D1 mini V3 gewählt. Er ist klein, lässt sich zusammen mit den anderen beiden Modulen gut auf einem Mini-Breadboard unterbringen aber auch platzsparend in ein kleines Gehäuse einbauen.

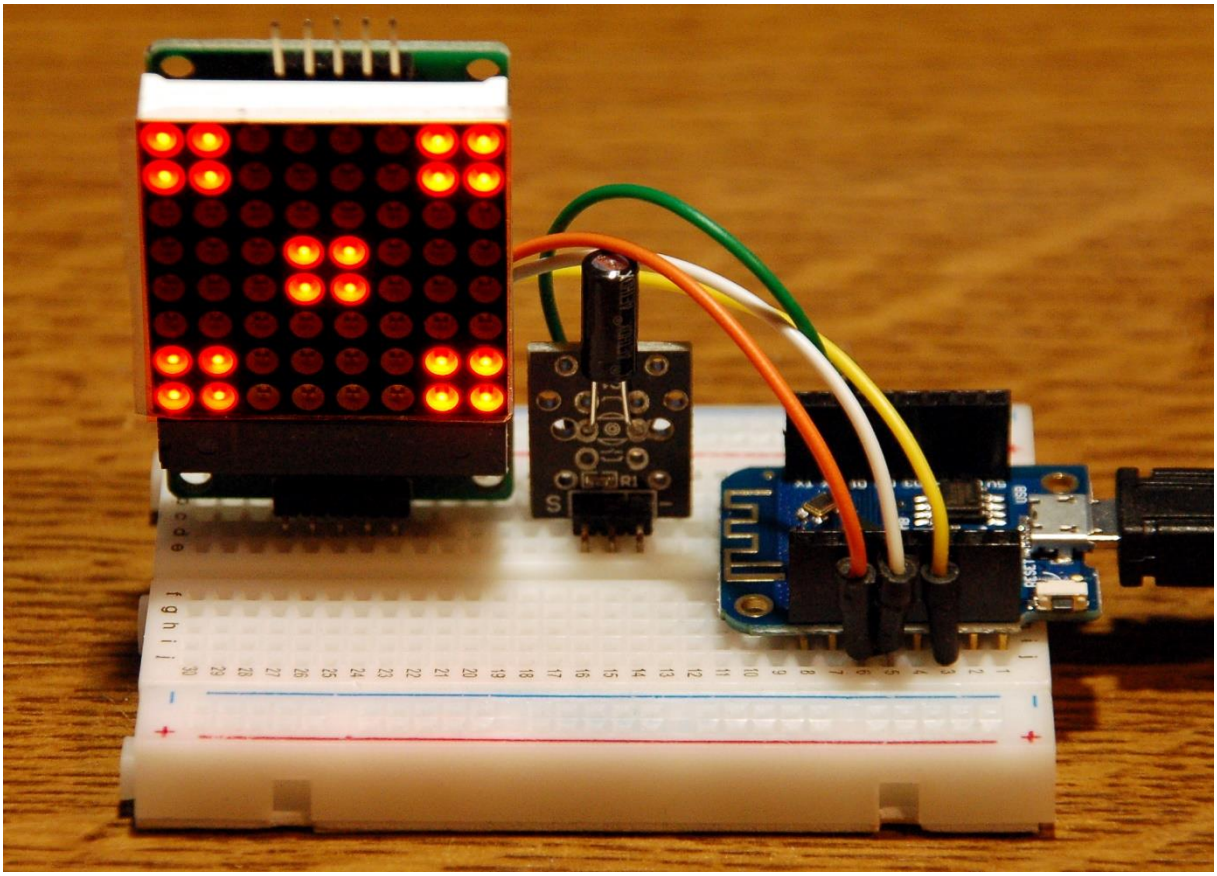


Abbildung 5: Würfel mit Matrixelement und D1 mini

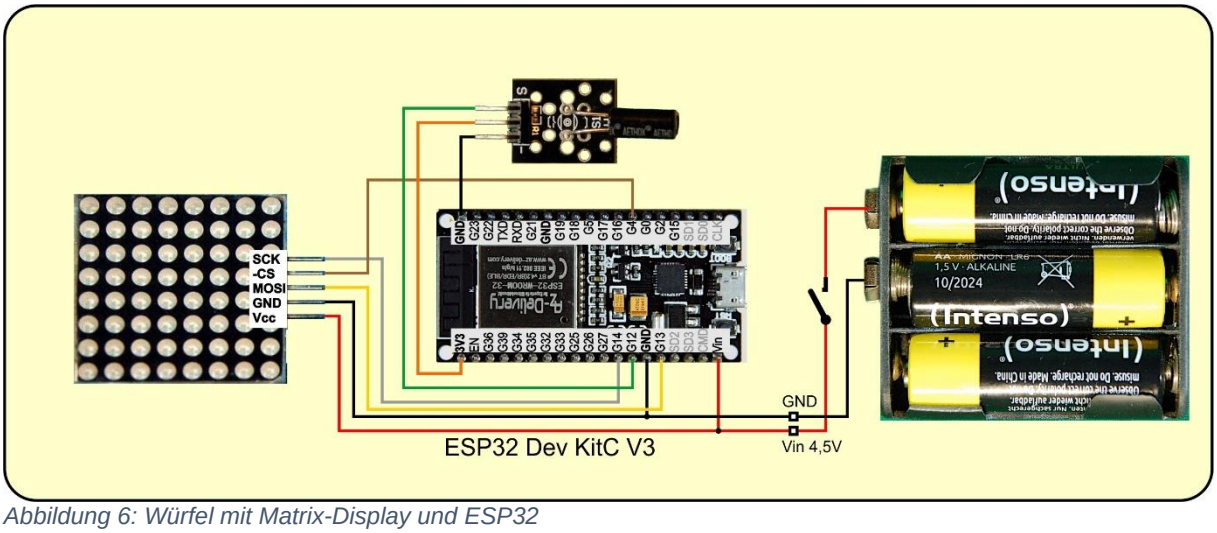


Abbildung 6: Würfel mit Matrix-Display und ESP32

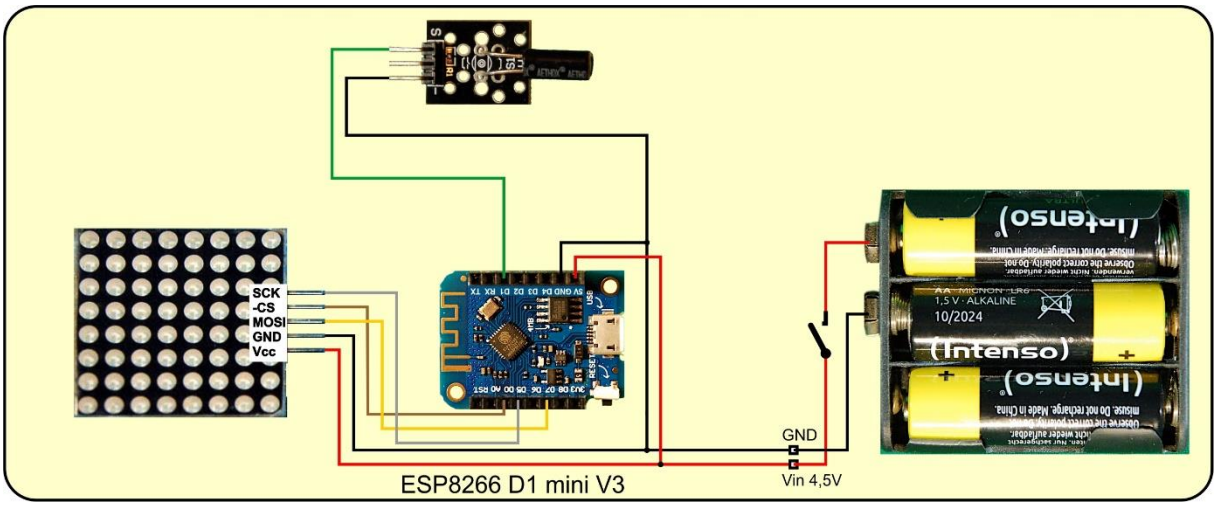


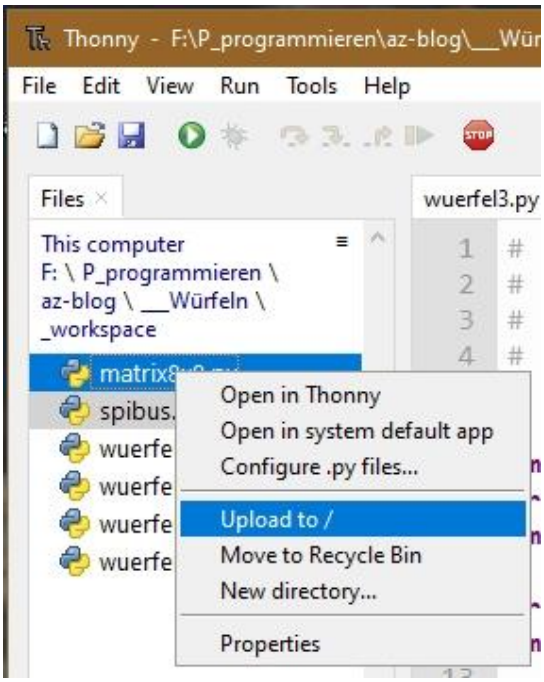
Abbildung 7: Würfel mit Matrix-Display und ESP8266 D1 mini V3

Ein Baustein mit 64 LEDs in einer 8 mal 8 Anordnung ersetzt die 7 LEDs. Für die Ansteuerung der 8 mal 8-LED-Matrix benötigen wir nicht etwa 64 Leitungen, sondern außer der Spannungsversorgung mit 5V und GND nur drei weitere, also insgesamt 5 Kabel. Das liegt daran, dass unter dem eigentlichen Display ein Treiberbaustein MAX7219 verborgen ist, der die LEDs nach unseren Wünschen zum Leuchten bringt. Wie das genau funktioniert, wird in einem eigenen Blogpost mit dem Titel "Das Mammut-Matrix-Display unter MicroPython" behandelt. Die Wünsche teilen wir dem MAX7219 über den sogenannten SPI-Bus (Serial Peripheral Interface) mit. Die Daten werden über die besagten drei Leitungen MOSI, SCK und CS übertragen. CS sagt dem MAX7219, dass in Kürze Daten anmarschieren, auf der Leitung MOSI (Master Out Slave IN) liegen die Datenpegel an, 0 oder 1, und auf der SCK-Leitung, sagt der ESP32 oder ESP8266 dem MAX7219, wann letzterer das Datenbit lesen soll. Die Hintergründe sind für dieses Projekt nicht wichtig, deshalb gehe ich auch nicht näher darauf ein. Das Einrichten des SPI-Busses überlasse ich dem Modul **spibus**. Es enthält eine Funktion **definePins**(), die dafür sorgt, dass die Leitungen entsprechend den Schaltplänen initialisiert werden. Die Auswahl des Controllertyps erfolgt automatisch.

Außerdem müssen wir die Klasse MATRIX aus dem Modul matrix8x8 importieren, damit, die Befehle zum Ansteuern des Displays verfügbar werden.

Was ist eigentlich ein Modul? Module sind in MicroPython das, was in der Arduino-IDE eine Bibliothek oder Library ist, also eine Sammlung von Programm- und Datenstrukturen, die man zusätzlich zu den eigenen Anweisungen nutzen kann. Dort wie hier sind das meistens Funktionen, die als Bindeglied zwischen Programm und angeschlossener Hardware dienen, sogenannte Treiber.

Die Module, die wir in den vorangegangenen Programmen genutzt haben, waren bereits fest in die Firmware von MicroPython integriert. Bei matrix8x8 und spibus ist das anders. Diese beiden liegen als MicroPython-Programmdateien vor. Am besten Sie laden matrix8x8.py und spibus.py gleich einmal herunter und verfrachten sie in das Arbeitsverzeichnis von Thonny. Dann markieren Sie die beiden mit gedrückter STRG-Taste und öffnen mit einem Rechtsklick das Kontextmenü.



Dann schicken Sie die Dateien mit Klick auf **Upload to /** in den Flash des Controllers. Kommen wir jetzt zum Programm.

```
import sys,os
from machine import Pin
import random

from matrix8x8 import MATRIX
import spibus

spi,cs=(spibus.definePins())
d=MATRIX(spi,cs,1)
kontakt=Pin(12,Pin.IN, Pin.PULL_UP)
```

Nach den Importen deklariere ich wieder den Eingang **kontakt** für die Leitung vom Schalter. Die Funktion **definePins**() aus dem Modul **spibus** legt abhängig vom Controllertyp die Leitungen für den SPI-Bus fest, richtet die dafür nötigen Objekte ein und gibt dann das SPI-Bus-Objekt sowie das Pinobjekt für die CS-Leitung zurück. Das Rückgabeobjekt ist ein Tuple, das ich aber gleich in die beiden Variablen **spi** und **cs** entpacken lasse.

Dann erzeuge ich mit diesen Daten ein Matrix-Display-Objekt **d** mit einem 8 mal 8-Element. Diese Programmzeilen kann man auch in REPL kopieren und mit <ENTER> abschicken.

>>> import sys,os
from machine import Pin
import random

from matrix8x8 import MATRIX
import spibus

kontakt=Pin(4,Pin.IN)
spi,cs=(spibus.definePins())
d=MATRIX(spi,cs,1)

Hardware-Bus 1: Pins fest vorgegeben
SCK Pin(14), MISO Pin(12), MOSI Pin(13), CSPin(16)

Konstruktor von MATRIX: 8 x 8 Pixel

Die erzeugten Objekte sind jetzt abfragbar.

>>> spi
HSPI(id=1, baudrate=4000000, polarity=0, phase=0)
>>> cs
Pin(16)
>>> d
<MATRIX object at 3fff0a70>

Im nächsten Abschnitt definiere ich in Form von Tuples unter Verwendung der Binärschreibweise jeweils acht Bytes, die das Muster der entsprechenden Würfelseite ergeben, 1 ist wieder hell, 0 ist dunkel. Danach fasse ich die sechs Muster-Tuples im Tuple **state** zusammen, damit ich die Muster durch ihren Index abrufen kann.

>>> sechs
(195, 195, 0, 195, 195, 0, 195, 195)
>>> state[5]
(195, 195, 0, 195, 195, 0, 195, 195)

Das tuple **sechs** steht in **state** an der 5. Position. Sie erinnern sich, die Zählung in seriellen Datenstrukturen wie Tuples, Listen und Strings beginnt bei 0.

In meiner Mainloop brauche ich einen Timer, der das Anzeigen des Musters verzögert, während gewürfelt wird. Ein solches Objekt liefert die Funktion **TimeOut**(). Sie gibt die Adresse einer anderen Funktion (**compare()**) zurück, die ich an beliebigen Stellen abfragen kann, ob die eingestellte Zeit schon abgelaufen ist oder nicht. Die Hintergründe sind in einer Anleitung für Einsteiger zu kompliziert, daher lasse ich dieselben weg. Wer dennoch tiefer einsteigen möchte, den verweise ich gerne auf ein PDF-Dokument, das die Wirkungsweise erklärt. Wichtig ist vorerst nur, dass die von **TimeOut**() zurückgegebene Funktion **compare**() dann mit **True** antwortet, wenn die Differenz zwischen der aktuellen Systemzeit in Millisekunden und der Startzeit größer ist als die Anzahl Millisekunden in **delay**.

Die Funktion **switch**() kann ich mir sparen. Ich verwende statt dieser eine Methode aus dem Modul **matrix8x8**. Dazu gleich mehr.

```
delay=1000
n=1
```

```
zeigen=TimeOut(delay)
```

In **delay** lege ich die Verzögerungszeit für die Anzeige fest und mit **n=1** erzeuge ich schon mal die Variable für die Würfelaugen, damit ich in der while-Schleife keinen Abbruch durch einen Indizierungsfehler in das Tuple **state** bekomme. In **zeigen** lege ich die Adresse der Funktion ab, die mir **TimeOut**() anliefert und die darauf getrimmt ist, nach Ablauf der Millisekunden in **delay** den Wert **True** zurückzugeben. Sie werden gleich sehen, wie das funktioniert.

```
while 1:
    if kontakt.value()==1:
        d.clear()
        n=random.getrandbits(8) % 6
        zeigen=TimeOut(delay)
    elif zeigen():
        d.shape(0,state[n])
        d.show()
    else :
        pass
```

Die Schleife ist wieder das Herz des Programms. Die Abfrage auf den Kontakt kennen wir schon. Die Geometrien des Matrixelements und des Kontaktmoduls haben mich aber dazu bewogen, die Abfrage des Kontakts umzukehren von geschlossen auf geöffnet. Die beiden Bausteine lassen sich dann nämlich brav nebeneinander in ein kleines Breadboard stecken. Wenn der Kontakt geöffnet ist, mache ich das Display aus, oder besser, ich lösche die Anzeige. Das macht die Methode **clear**() aus dem Modul matrix8x8. Danach wird eine Zahl von 0 bis 5 gewürfelt, das kennen wir schon. Jedes Mal, wenn oder so lange wie gewürfelt, beziehungsweise geschüttelt wird, stelle ich den Timer nach.

Jetzt kommt was Neues, die anfangs angekündigte Ausbaustufe der if-Struktur. Mit **elif** kann ich im else-Zeig von if erneut prüfen, ob eine Situation vorliegt, auf die vorher noch nicht getestet wurde. Hier ist es der Aufruf der Funktion **zeigen**(). Solange seit dem letzte Stellen der Ablaufzeit, wo immer das auch stattgefunden hat, die 1000 ms noch nicht abgelaufen sind, liefert **zeigen**() ein **False** zurück. Ist die Zeit um, bekommen wir ein **True**.

Dann darf das Muster, auf das der Index n im Tuple **state** zeigt, durch die Methode **shape**() aus dem Paket matrix8x8 zur Anzeige transportiert werden. Das nachfolgende **show**() lässt die entsprechenden LEDs aufleuchten.
Ein else: schließt die if-Konstruktion ab. Damit MicroPython keinen Syntaxfehler meldet habe ich die Anweisung pass eingefügt, die nur der Form halber dasteht, aber nichts ausführt. Eigentlich hätte ich das else: auch komplett weglassen können, aber uneigentlich wäre es mir dann nicht vergönnt gewesen, die ganze Schönheit einer if-Struktur zeigen zu können.

Jetzt ganz schnell, Programm wuerfeln3.py in einem Editorfenster öffnen, starten, "würfeln" und staunen, was das Matrixdisplay hervorbringt. Ich schätze mal, dass Sie neugierig geworden sind, was das Modul matrix8x8 noch so alles zu bieten hat. Öffnen Sie die Datei einfach in einem Editorfenster. Schauen Sie sich die

Funktionsdefinitionen mit den Parameterlisten an, dann haben Sie einen Anhaltspunkt, was Sie mit welchen Argumenten aufrufen können. Die volle Wirkung zeigt sich leider erst bei Displays mit mehreren Elementen, zum Beispiel rollen, shiften, blinken, Textausgabe …

Viel Vergnügen beim Forschen und Ausprobieren und natürlich beim Würfeln.