Inhalt

Bevor es losgeht ...

Kaum ein elektronisches Gerät seiner Preisklasse hat in den vergangenen Monaten so viel von sich reden gemacht wie der Raspberry Pi. Der Raspberry Pi ist, auch wenn er kaum größer ist als eine Kreditkarte, ein vollwertiger Computer, und es gibt ihn zu einem sehr günstigen Preis. Nicht nur die Hardware ist günstig, die Software ist es noch mehr. Das Betriebssystem und alle im Alltag notwendigen Anwendungen werden kostenlos zum Download angeboten.

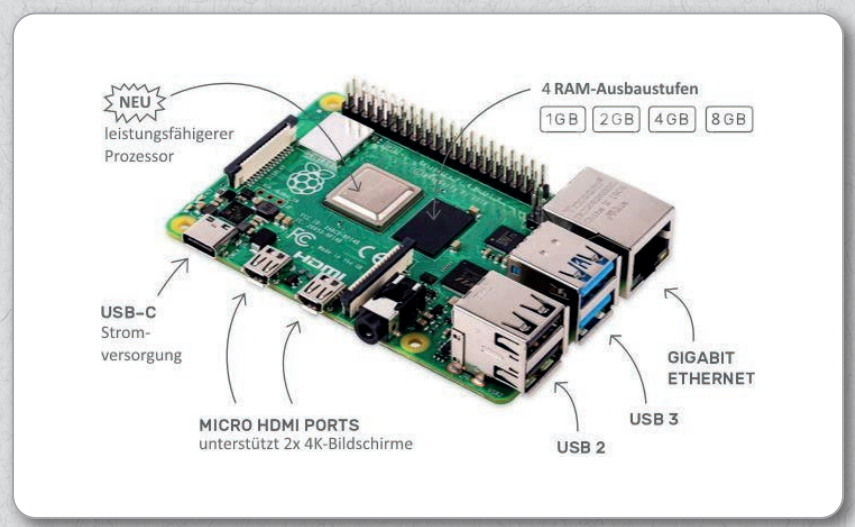


Abb. E.1: *Der Raspberry Pi 4 – ein PC im Miniformat.*

Mit dem speziell angepassten Linux ist der Raspberry Pi ein Strom sparender, lautloser PC-Ersatz. Seine frei programmierbare GPIO-Schnittstelle macht den Raspberry Pi besonders interessant für Hardwarebastler und die Maker-Szene.

Was braucht man?

Wenn Sie diesen Text lesen, haben Sie sich sicher schon etwas mit dem Raspberry Pi beschäftigt. Deshalb fassen wir die Systemvoraussetzungen für das Maker Kit nur kurz zusammen.

Raspberry Pi 4

Natürlich brauchen Sie einen Raspberry Pi 4. Dieser ist in drei verschiedenen RAM-Ausbaustufen lieferbar. Für die Scratch-Experimente aus diesem Maker Kit muss der Raspberry Pi 4 mindestens 2 GB RAM haben, für die Python-Experimente spielt die RAM-Größe keine Rolle.

Raspberry Pi 400

Der Raspberry Pi 400 entspricht technisch weitgehend einem Raspberry Pi 4, der in eine Tastatur eingebaut ist. Hier brauchen Sie keine USB-Tastatur. Die Anschlüsse für Maus, HDMI, GPIO und USB-C-Netzteil sind wie beim Raspberry Pi 4 auf der Rückseite vorhanden.

Abb. E.2: *Raspberry Pi 400*

USB-Typ-C-Handyladegerät

Der Raspberry Pi 4 kann über ein leistungsfähiges Smartphone-Netzteil mit USB-Anschluss Typ C mit Strom versorgt werden. Das Netzteil muss 5 V und nach Spezifikationen mindestens 3.000 mA liefern. Sind keine USB-Geräte angeschlossen, reichen auch 2.500 mA.

So macht sich ein zu schwaches Netzteil bemerkbar

Wenn der Raspberry Pi bootet, dann aber oben rechts auf dem Bildschirm ein gelbes Blitzsymbol erscheint, deutet das auf eine zu schwache Stromversorgung hin.

Speicherkarte

Die Speicherkarte enthält das Betriebssystem. Eigene Daten und installierte Programme werden ebenfalls darauf gespeichert. Die MicroSD-Speicherkarte sollte mindestens 8 GB groß sein und nach Herstellerangaben mindestens den Class-4-Standard unterstützen. Dieser Standard gibt die Geschwindigkeit der Speicherkarte an. Eine aktuelle Class-10-Speicherkarte macht sich in der Performance deutlich bemerkbar.

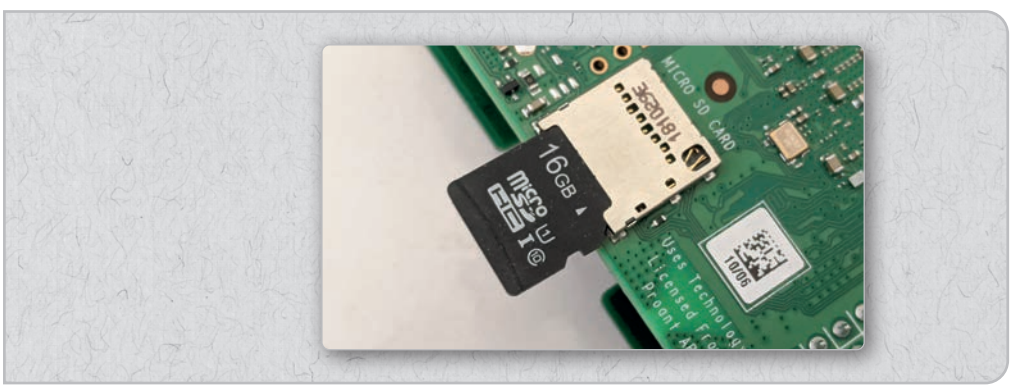


Abb. E.3: *MicroSD-Karte 16 GB Class 10 im Raspberry Pi 4. Die Zahl im Kreis gibt die Klassifizierung der Speicherkarte an.*

Tastatur

Jede gängige Tastatur mit USB-Anschluss kann genutzt werden. Kabellose Tastaturen funktionieren manchmal nicht, da sie zu viel Strom oder spezielle Treiber benötigen.

Beim Raspberry Pi 400 benötigen Sie keine zusätzliche Tastatur.

Maus

Verwenden Sie eine Maus mit USB-Anschluss. Einige Tastaturen haben zusätzliche USB-Anschlüsse für Mäuse, sodass Sie keinen weiteren Anschluss belegen müssen. Diesen können Sie dann z. B. für einen USB-Stick nutzen.

Netzwerkkabel

Zur Verbindung mit dem Router im lokalen Netzwerk wird ein Netzwerkkabel benötigt. Zur Ersteinrichtung ist dieses auf jeden Fall noch erforderlich, später kann man auch WLAN einsetzen. Ohne Internetzugang sind viele Funktionen des Raspberry Pi nicht sinnvoll nutzbar.

HDMI-Kabel

Der Raspberry Pi kann per HDMI-Kabel an Monitore oder Fernseher angeschlossen werden. Der Raspberry Pi 4 hat zwei Micro-HDMI-Anschlüsse, für die besondere Anschlusskabel notwendig sind. Zum Anschluss an Computermonitore mit DVI-Anschluss gibt es spezielle HDMI-Adapter. HDMI-Kabel sind im Elektronikhandel zu Preisen erhältlich, die fast dem Preis des Raspberry Pi selbst entsprechen. Bei Onlineversendern (z. B. `amzn.to/2Wnm76G`) bekommt man sie einschließlich Versand für wenige Euro. Der Unterschied: für die Verwendung am Raspberry Pi außer dem Preis keiner.

Raspberry-Pi-OS-Betriebssystem

Wir gehen bei allen Projekten davon aus, dass die aktuellste Version des Raspberry-Pi-OS-Betriebssystems auf der Speicherkarte installiert ist. Zur Installation verwenden Sie am besten das Programm *Raspberry Pi Imager* von `www.raspberrypi.org/downloads`.

Klicken Sie auf *OS wählen* und wählen Sie unter *Raspberry Pi OS (other)* die Variante *Raspberry Pi OS (64-bit)*.

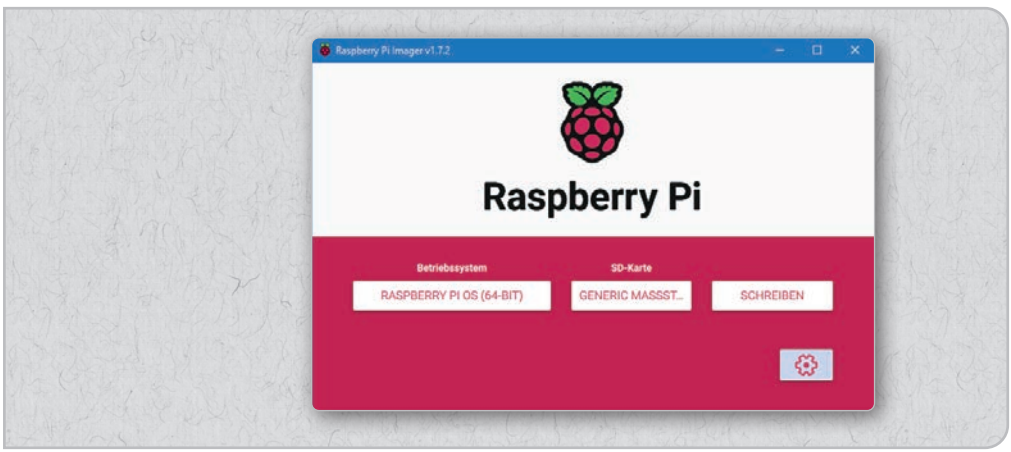


Abb. E.4: *Installation der Speicherkarte mit Raspberry Pi Imager auf dem PC.*

Klicken Sie auf *SD-Karte wählen* und wählen Sie die Speicherkarte im Kartenleser aus. In den meisten Fällen wird nur eine Speicherkarte zur Auswahl angeboten. Die Speicherkarte wird automatisch formatiert, bisher darauf befindliche Daten müssen vorher gesichert werden.

Klicken Sie dann auf *Write*. Jetzt wird das Betriebssystem heruntergeladen und automatisch auf die Speicherkarte übertragen. Warten Sie, bis der Vorgang komplett abgeschlossen ist.

Stecken Sie die Speicherkarte in den Raspberry Pi und schließen Sie dann erst die Stromversorgung an, um ihn zu booten. Ein Startassistent fragt nach ein paar Grundeinstellungen, etwa nach Sprache und Netzwerkverbindung.

Fast wie Windows – die grafische Oberfläche

Viele schrecken bei dem Wort Linux erst einmal zurück, weil sie befürchten, kryptische Befehlsfolgen per Kommandozeile eingeben zu müssen wie vor 30 Jahren unter DOS. Weit gefehlt! Als offenes Betriebssystem bietet Linux den Entwicklern freie Möglichkeiten, eigene grafische Oberflächen zu entwickeln. So ist man als Anwender des im Kern immer noch kommandozeilenorientierten Betriebssystems nicht auf eine Oberfläche festgelegt.

Das Raspberry-Pi-OS-Linux für den Raspberry Pi verwendet eine angepasste Variante der Oberfläche LXDE (*Lightweight X11 Desktop Environment*), die einerseits sehr wenige Systemressourcen benötigt und andererseits mit ihrem Startmenü und dem Dateimanager der gewohnten Windows-Oberfläche sehr ähnelt.

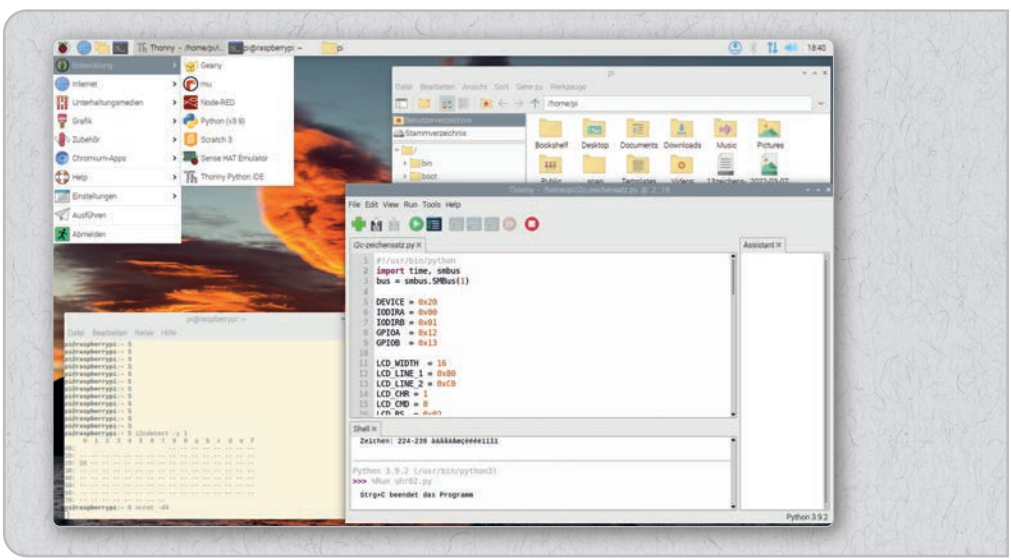


Abb. E.5: *Der Desktop auf dem Raspberry Pi.*

Linux-Anmeldung

Selbst die bei Linux typische Benutzeranmeldung wird im Hintergrund erledigt. Falls Sie sie doch einmal brauchen: Der Benutzername lautet `pi` und das Passwort `raspberry`.

Das Menüsymbol links oben öffnet das Startmenü, die Symbole daneben den Webbrowser, den Dateimanager und ein Kommandozeilenfenster. Das Startmenü ist wie unter Windows mehrstufig aufgebaut. Häufig verwendete Programme lassen sich mit einem Rechtsklick auf dem Desktop ablegen.

Raspberry Pi ausschalten

Theoretisch kann man beim Raspberry Pi einfach den Stecker ziehen, und er schaltet sich ab. Besser ist es jedoch, ihn wie einen PC sauber herunterzufahren. Wählen Sie dazu im Menü *Shutdown*.

Beispielprogramme zum Maker Kit herunterladen und nutzen

Zu diesem Buch bieten wir Zusatzmaterial zum Download, wie unter anderem alle Beispielprogramme aus den folgenden Kapiteln sowie Zusatzkapitel.

1. Besuchen Sie mit dem Chromium-Browser auf dem Raspberry Pi die Seite *www.buch.cd*.
2. Geben Sie dort diesen Code ein: 67112-7.
3. Folgen Sie den Anweisungen zum Download. Der Chromium-Browser speichert die Zip-Datei standardmäßig unter `/home/pi/Downloads`. Übernehmen Sie diese Vorgabe.
4. Nach erfolgreichem Download klicken Sie unten links im Browser auf die heruntergeladene Datei und wählen *Öffnen*.

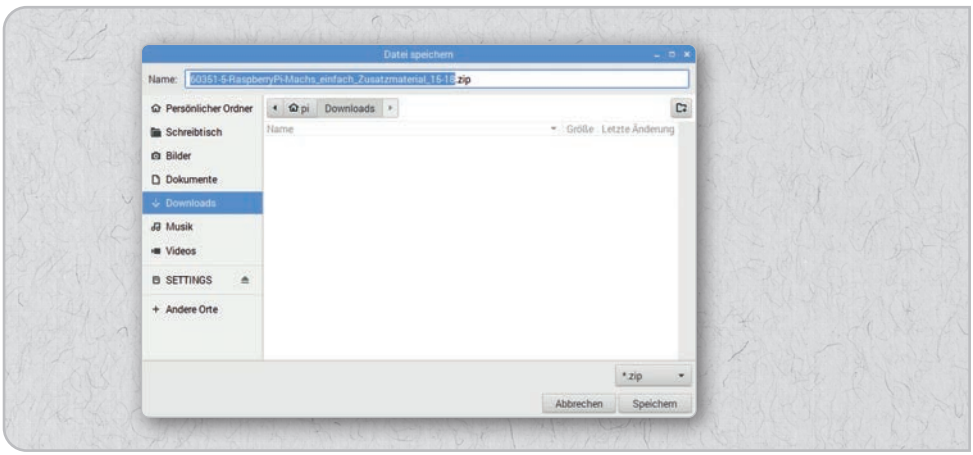


Abb. E.6: *Downloadordner in Chromium wählen.*

5. Die Datei wird im vorinstallierten *Xarchiver* geöffnet. Klicken Sie mit der rechten Maustaste auf den angezeigten Ordner und wählen Sie im Menü *Entpacken*. Tragen Sie im Feld *Entpacken nach* das Home-Verzeichnis `/home/pi` ein.

6. Klicken Sie auf *Entpacken*. Danach liegen die Dateien in Ihrem Home-Verzeichnis.

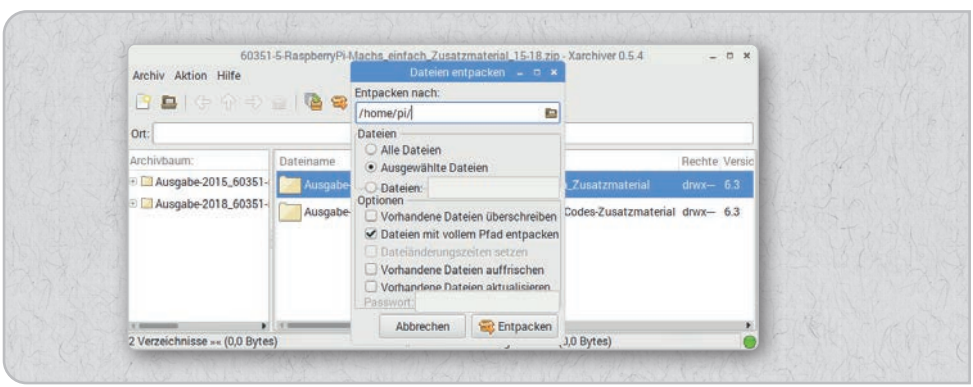


Abb. E.7: *Heruntergeladenes Zip-Archiv entpacken.*

Das erste Programm mit Python

Zum Einstieg in die Programmierung ist auf dem Raspberry Pi die Programmiersprache Python vorinstalliert. Python überzeugt durch seine klare Struktur, die einen einfachen Einstieg ins Programmieren erlaubt, es ist aber auch eine ideale Sprache, um »mal schnell« etwas zu automatisieren, was man sonst von Hand erledigen würde. Da keine Variablendeklarationen, Typen, Klassen oder komplizierten Regeln zu beachten sind, macht das Programmieren wirklich Spaß.

Die Entwicklungsumgebung *Thonny Python IDE* für Python 3 wird über den Menüpunkt *Entwicklung/Thonny Python IDE* gestartet. Dort erscheint ein dreigeteiltes Programmfenster.

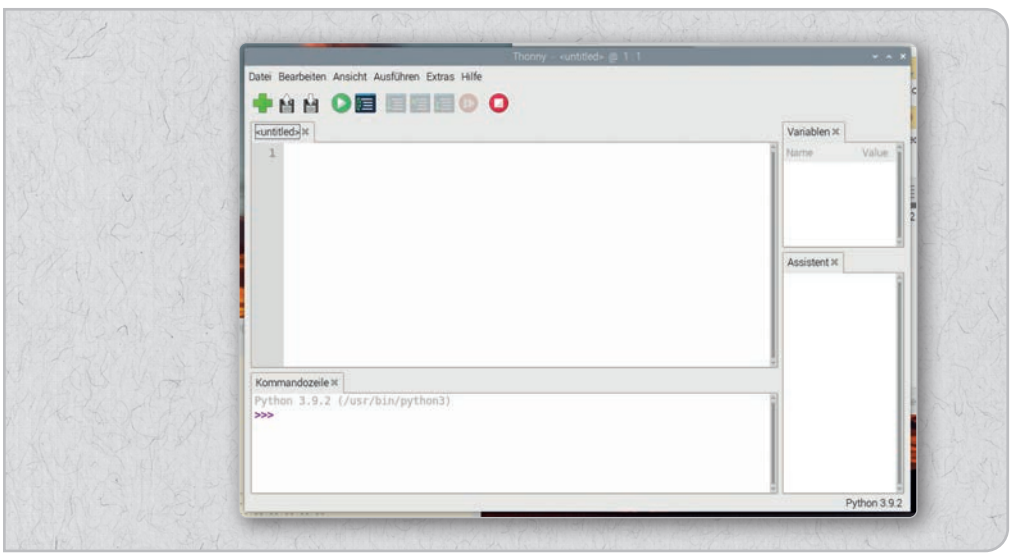


Abb. E.8: *Die Thonny Python IDE.*

Startet die Thonny Python IDE nur mit einem einfachen zweigeteilten Fenster und großen Icons, klicken Sie auf den eingeblendeten Link oben rechts, um auf die Standardoberfläche umzuschalten.

Wählen Sie im Menü *Tools/Options*, wählen Sie in der Liste *Language* die Sprache *Deutsch*, bestätigen Sie mit *OK* und starten Sie die Thonny Python IDE neu mit deutschsprachiger Oberfläche.

In diesem Fenster öffnen Sie vorhandene Python-Programme, schreiben neue oder arbeiten direkt Python-Kommandos interaktiv ab, ohne ein eigentliches Programm schreiben zu müssen. Geben Sie z. B. am Prompt im unteren Teilfenster *Shell* Folgendes ein:

```
>>> 1+2
```

Es erscheint sofort die richtige Antwort: `3`

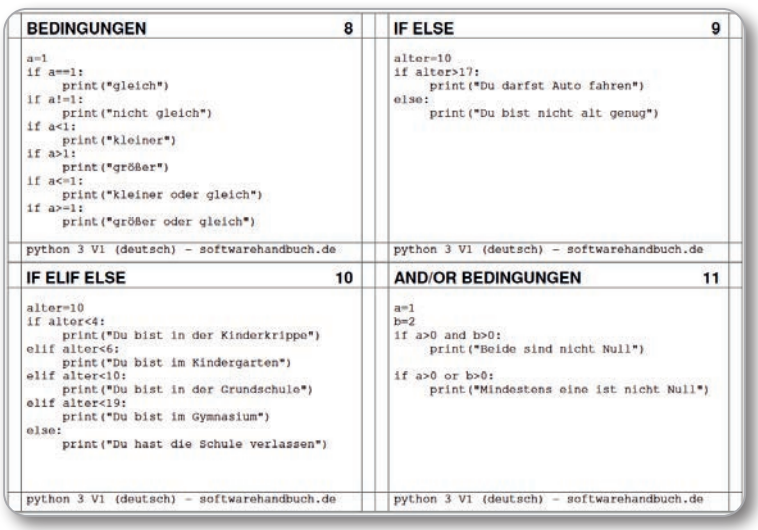
Auf diese Weise lässt sich Python als komfortabler Taschenrechner verwenden, was aber noch nichts mit Programmierung zu tun hat. Üblicherweise fangen Programmierkurse mit einem *Hallo Welt*-Programm an, das auf den Bildschirm den Satz »Hallo Welt« schreibt. Das ist in Python derart einfach, dass es sich nicht einmal lohnt, dafür eine eigene Überschrift einzufügen. Tippen Sie im Shell-Fenster einfach folgende Zeile:

```
>>> print("Hallo Welt")
```

Dieses erste »Programm« schreibt `Hallo Welt` in die nächste Bildschirmzeile.

Python-Flashcards

Python ist die ideale Programmiersprache, um den Einstieg in die Programmierung zu erlernen. Nur die Syntax und die Layoutregeln sind etwas gewöhnungsbedürftig. Zur Hilfestellung im Programmieralltag werden die wichtigsten Syntaxelemente der Sprache Python in Form kleiner »Spickzettel« kurz beschrieben. Diese basieren auf den Python-Flashcards von David Whale. Was es damit genau auf sich hat, finden Sie bei `bit.ly/pythonflashcards3`. Diese Flashcards erklären keine technischen Hintergründe, sondern beschreiben nur anhand ganz kurzer Beispiele die Syntax dazu, wie etwas gemacht wird.



BEDINGUNGEN 8

```
a=1
if a==1:
     print("gleich")
if a!=1:
     print("nicht gleich")
if a<1:
     print("kleiner")
if a>1:
     print("größer")
if a<=1:
     print("kleiner oder gleich")
if a>=1:
     print("größer oder gleich")
```

python 3 V1 (deutsch) - softwarehandbuch.de

IF ELSE 9

```
alter=10
if alter>17:
     print("Du darfst Auto fahren")
else:
     print("Du bist nicht alt genug")
```

python 3 V1 (deutsch) - softwarehandbuch.de

IF ELIF ELSE 10

```
alter=10
if alter<4:
     print("Du bist in der Kinderkrippe")
elif alter<6:
     print("Du bist im Kindergarten")
elif alter<10:
     print("Du bist in der Grundschule")
elif alter<19:
     print("Du bist im Gymnasium")
else:
     print("Du hast die Schule verlassen")
```

python 3 V1 (deutsch) - softwarehandbuch.de

AND/OR BEDINGUNGEN 11

```
a=1
b=2
if a>0 and b>0:
     print("Beide sind nicht Null")

if a>0 or b>0:
     print("Mindestens eine ist nicht Null")
```

python 3 V1 (deutsch) - softwarehandbuch.de

Abb. E.10: *Ausschnitt aus den Python-Flashcards.*

Hier sehen Sie auch gleich, dass die Python-Shell zur Verdeutlichung automatisch verschiedene Textfarben verwendet. Python-Kommandos sind lila, Zeichenketten grün und Ergebnisse blau. Später werden Sie noch weitere Farben entdecken.

Abb. E.9: *Hallo Welt in Python (oberhalb ist noch die Ausgabe der Berechnung zu sehen).*

Die Thonny Python IDE ist eine komplette Python-Shell und Entwicklungsumgebung. Für den Start in die Programmierung sind keine zusätzlichen Komponenten nötig.

- Das Symbol *Öffnen* öffnet ein gespeichertes Python-Programm. Python-Programme werden direkt im Quellcode verarbeitet und haben die Endung `.py`.

- Das Symbol *Neu* öffnet ein neues Eingabefenster, sodass man gleich ein eigenes Python-Programm schreiben kann.

- Das Symbol *Aktuelles Skript ausführen* oder F5 startet das Programm. Eventuelle Fehler werden im Fenster *Shell* angezeigt.

1 Die erste LED leuchtet am Raspberry Pi

Die 40-polige Stiftleiste an der Längsseite des Raspberry Pi 4 bietet die Möglichkeit, direkt Hardware anzuschließen, um z. B. über Taster Eingaben zu machen oder programmgesteuert LEDs leuchten zu lassen. Diese Stiftleiste wird als GPIO bezeichnet. Die englische Abkürzung für *General Purpose Input Output* bedeutet auf Deutsch einfach »allgemeine Ein- und Ausgabe«.

Vorsicht

Verbinden Sie auf keinen Fall irgendwelche GPIO-Pins miteinander und warten ab, was passiert, sondern beachten Sie unbedingt folgende Hinweise:

Einige GPIO-Pins sind direkt mit Anschlüssen des Prozessors verbunden, ein Kurzschluss kann den Raspberry Pi komplett zerstören. Verbindet man über einen Schalter oder eine LED zwei Pins miteinander, muss immer ein Vorwiderstand dazwischengeschaltet werden.

Verwenden Sie für Logiksignale stets Pin 1, der +3,3 V liefert und bis 50 mA belastet werden kann. Pin 6 ist die Masseleitung für Logiksignale.

Jeder GPIO-Pin mit einer Nummer kann als Ausgang (z. B. für LEDs) oder als Eingang (z. B. für Taster) geschaltet werden. GPIO-Ausgänge liefern im Logikzustand *1* eine Spannung von +3,3 V, im Logikzustand *0* 0 V. GPIO-Eingänge liefern bei einer Spannung bis +1,7 V das Logiksignal *0*, bei einer Spannung zwischen +1,7 V und +3,3 V das Logiksignal *1*.

Pin 2 und Pin 4 liefern +5 V zur Stromversorgung externer Hardware, z. B. eines LC-Displays. Hier kann so viel Strom entnommen werden, wie das Netzteil des Raspberry Pi liefert. Dieser Pin darf nicht mit einem GPIO-Eingang verbunden werden.

Von diesen 40 Pins lassen sich die 22, die nur mit Nummern bezeichnet sind, wahlweise als Eingang oder Ausgang programmieren und so für vielfältige Hardwareerweiterungen nutzen. Die übrigen sind für die Stromversorgung und andere Zwecke fest eingerichtet.

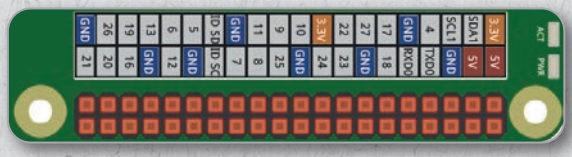


Abb. 1.1: *Belegung der GPIO-Schnittstelle. Die abgerundete Ecke unten rechts entspricht der Ecke der Raspberry-Pi-Platine.*

1.1 | Bauteile im Paket

Das Paket enthält diverse elektronische Bauteile, mit denen sich die beschriebenen Experimente (und natürlich auch eigene) aufbauen lassen. An dieser Stelle werden die Bauteile nur kurz vorgestellt. Die erforderliche praktische Erfahrung im Umgang damit bringen dann die wirklichen Experimente.

Benötigte Bauteile

- 3 x Steckplatine
- 1 x LCD-Anzeige
- 1 x Pfostenverbinder (16-polig)
- 1 x Portexpander MCP 23017
- 10 x LED
- 4 x Taster
- 1 x Widerstand 10 kOhm (Braun-Schwarz-Orange)
- 1 x Widerstand 560 Ohm (Grün-Blau-Braun)
- 1 x Potenziometer 15 kOhm
- 12 x Verbindungskabel (Steckplatine – Raspberry Pi)
- Schaltdraht

1.1.1 | Steckplatinen

Für den schnellen Aufbau elektronischer Schaltungen (ohne löten zu müssen) sind drei Steckplatinen im Paket enthalten. Hier können elektronische Bauteile direkt in ein Lochraster mit Standardabständen eingesteckt werden. Bei diesen Platinen sind die äußeren Längsreihen über Kontakte (X und Y) miteinander verbunden.

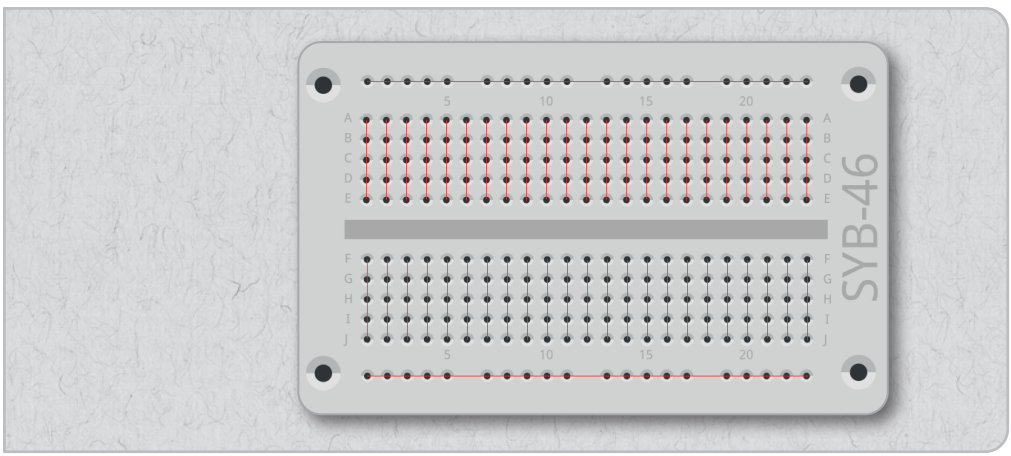


Abb. 1.2: *Die Steckplatine aus dem Paket mit einigen Verbindungen, beispielhaft angedeutet.*

Diese Kontaktreihen werden oft als Plus- oder Minuspol zur Stromversorgung der Schaltungen genutzt. In den anderen Kontaktreihen sind jeweils fünf Kontakte (A bis E und F bis J) quer miteinander verbunden, wobei in der Mitte der Platine eine Lücke ist. Deshalb können in der Mitte größere Bauelemente, wie z. B. der Portexpander, eingesteckt und nach außen hin verdrahtet werden.

1.1.2 | Verbindungskabel

Die farbigen Verbindungskabel haben alle auf einer Seite einen dünnen Drahtstecker, mit dem sie sich auf die Steckplatine stecken lassen. Auf der anderen Seite befindet sich eine Steckbuchse, die auf einen GPIO-Pin des Raspberry Pi passt.

Außerdem ist Schaltdraht im Paket enthalten. Damit stellen Sie kurze Verbindungsbrücken her, über die Kontaktreihen auf der Steckplatine verbunden werden. Schneiden Sie den Draht je nach Experiment mit einem kleinen Seitenschneider auf die passenden Längen ab. Um die Drähte besser in die Steckplatine stecken zu können, empfiehlt es sich, die Drähte leicht schräg abzuschneiden, sodass eine Art Keil entsteht. Entfernen Sie dann an beiden Enden auf einer Länge von etwa einem halben Zentimeter die Isolierung.

1.1.3 | Widerstände und ihre Farbcodes

Widerstände werden in der Digitalelektronik im Wesentlichen zur Strombegrenzung an den Ports eines Mikrocontrollers sowie als Vorwiderstände für LEDs verwendet. Die Maßeinheit für Widerstände ist Ohm. 1.000 Ohm entsprechen einem Kiloohm, abgekürzt kOhm, 1.000 kOhm entsprechen einem Megaohm, abgekürzt MOhm.

FARBE	WIDERSTANDSWERT IN OHM			
	1. RING (ZEHNER)	2. RING (EINER)	3. RING (MULTIPLIKATOR)	4. RING (TOLERANZ)
Silber			$10^{-2} = 0{,}01$	±10 %
Gold			$10^{-1} = 0{,}1$	±5 %
Schwarz		0	$10^{0} = 1$	
Braun	1	1	$10^{1} = 10$	±1 %
Rot	2	2	$10^{2} = 100$	±2 %
Orange	3	3	$10^{3} = 1.000$	
Gelb	4	4	$10^{4} = 10.000$	
Grün	5	5	$10^{5} = 100.000$	±0,5 %
Blau	6	6	$10^{6} = 1.000.000$	±0,25 %
Violett	7	7	$10^{7} = 10.000.000$	±0,1 %
Grau	8	8	$10^{8} = 100.000.000$	±0,05 %
Weiß	9	9	$10^{9} = 1.000.000.000$	

Abb. 1.1: *Die Tabelle zeigt die Bedeutung der farbigen Ringe auf Widerständen.*

Im Paket sind Widerstände in zwei verschiedenen Werten enthalten:

WERT	1. RING (ZEHNER)	2. RING (EINER)	3. RING (MULTIPL.)	4. RING (TOLERANZ)	VERWENDUNG
560 Ohm	Grün	Blau	Braun	Gold	Vorwiderstand für die Stromversorgung der LCD-Anzeige
10 kOhm	Braun	Schwarz	Orange	Gold	Pull-down-Widerstand für GPIO-Eingänge

Abb. 1.2: *Farbcodes der Widerstände im Lernpaket.*

1.1.4 | LEDs

An die GPIO-Ports können für Lichtsignale und Lichteffekte LEDs (LED = *Light Emitting Diode*, zu Deutsch Leuchtdiode) angeschlossen werden. Dabei muss zwischen dem verwendeten GPIO-Pin und der Anode der LED ein Vorwiderstand eingebaut werden, um den Durchflussstrom zu begrenzen und damit ein Durchbrennen der LED zu verhindern. Zusätzlich schützt der Vorwiderstand auch den GPIO-Ausgang des Raspberry Pi, da die LED in Durchflussrichtung fast keinen Widerstand bietet und deshalb der GPIO-Port bei einer Verbindung mit Masse schnell überlastet werden könnte. Die Kathode der LED verbindet man mit der Masseleitung an einem der GND-Pins des Raspberry Pi.

Die LEDs in diesem Maker Kit haben die Vorwiderstände bereits eingebaut. Sie können sie direkt an die GPIO-Pins anschließen.

LED in welcher Richtung anschließen?

Die beiden Anschlussdrähte einer LED sind unterschiedlich lang. Der längere ist der Pluspol, die Anode, der kürzere der Minuspol, die Kathode. Einfach zu merken: Das Pluszeichen hat einen Strich mehr als das Minuszeichen und macht damit den Draht quasi etwas länger. Außerdem sind die meisten LEDs auf der Minusseite abgeflacht wie ein Minuszeichen. Leicht zu merken: Kathode = kurz = Kante.

1.1.5 | LC-Display

Im Paket ist ein zweizeiliges LC-Display enthalten, das wie die meisten derartigen Displays zum Quasi-Standard HD44780 kompatibel ist. Dabei handelt es sich um die Bezeichnung des in das Display eingebauten Controllers, der Zeichen in ein bis vier Zeilen darstellt, ohne dass sich der Benutzer um die Ansteuerung der einzelnen Pixel zu kümmern braucht.

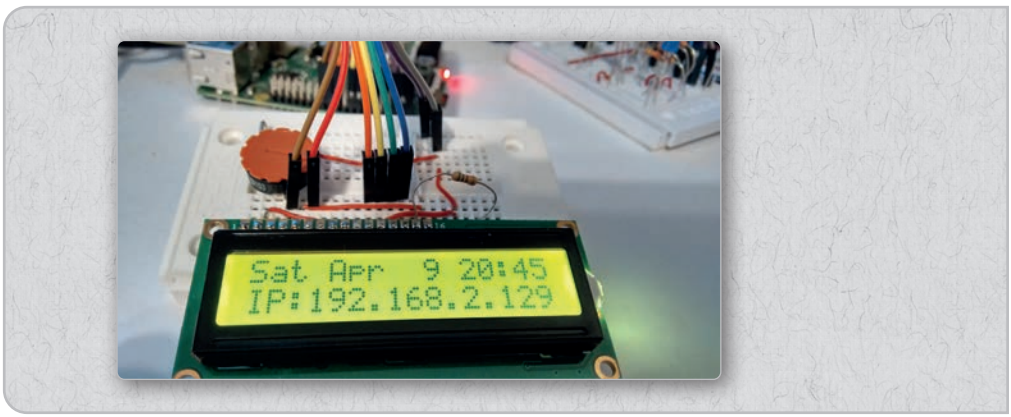


Abb. 1.3: *Zweizeiliges LC-Display auf einer Steckplatine am Raspberry Pi.*

Das Display hat eine 16-polige Anschlussleiste. Löten Sie hier den mitgelieferten Pfostenverbinder mit den kürzeren Pins so an, dass die längeren Pins nach unten frei herausstehen. Mit ihnen wird das Display später auf die Steckplatine gesteckt.

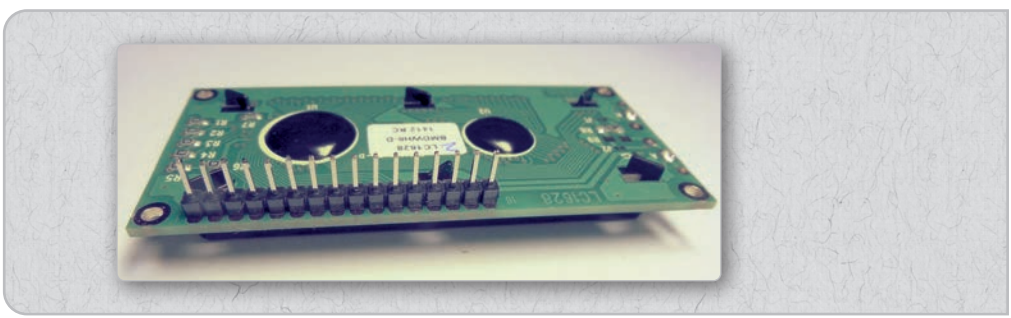


Abb. 1.4: *Display mit angelötetem Pfostenverbinder.*

Löten – einfach und richtig

Wer sich mit Hardwarebasteleien rund um den Raspberry Pi beschäftigt, wird ab und an auch mal etwas löten müssen. Für den Profi ist das kein Problem, für den Anfänger eigentlich auch nicht, sofern er ein paar wichtige Tricks beachtet. »Löten ist einfach« ist ein unterhaltsamer Comic mit Basiswissen für Hobbylöter, der nach der Creative-Commons-Lizenz CC-BY-SA frei weitergegeben werden darf (siehe letzte Seite der PDF-Datei). Dieser Comic ist im Download für dieses Maker Kit enthalten.

1.2 | GPIO mit Python

Um die GPIO-Ports in Python-Programmen zu nutzen, muss die Python-GPIO-Bibliothek installiert sein. In aktuellen Raspberry-Pi-OS-Versionen ist sie bereits standardmäßig vorinstalliert.

1.3 | LED mit Python ein-/ausschalten

Schließen Sie wie auf dem Bild eine LED an GPIO-Port 25 (Pin 22) an und verbinden Sie den Minuspol der LED über die Masseschiene der Steckplatine mit der Masseleitung des Raspberry Pi (Pin 6).

Benötigte Bauteile

- 1 x Steckplatine,
- 1 x LED (eingebauter Vorwiderstand),
- 2 x Verbindungskabel

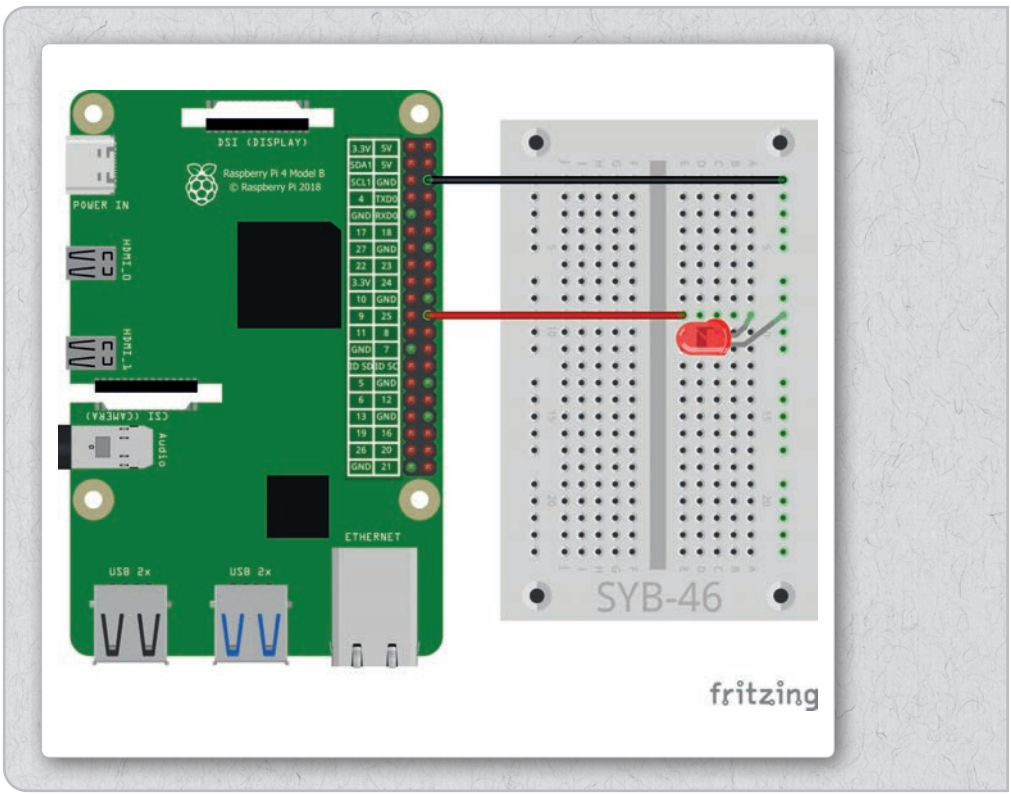


Abb. 1.5: *Eine LED an GPIO-Port 25.*

Das Programm `01led.py` lässt die LED zehnmal blinken:

```
#!/usr/bin/python
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(25, GPIO.OUT)
```

```
for i in range(10):
    GPIO.output(25, 1)
    time.sleep(0.2)
    GPIO.output(25, 0)
    time.sleep(0.2)
GPIO.cleanup()
```

1.3.1 | So funktioniert es

Das Beispiel zeigt die grundlegenden Funktionen der `RPi.GPIO`-Bibliothek.

```
#!/usr/bin/python
```

Diese Zeile steht am Anfang (fast) jedes Python-Skripts, um die Datei als solche zu identifizieren. Sie wäre zwar in diesem Fall nicht unbedingt nötig, aber gewöhnen Sie sich an, sie immer als erste Zeile in ein Python-Skript zu schreiben.

```
import RPi.GPIO as GPIO
```

Die Bibliothek `RPi.GPIO` muss in jedes Python-Programm importiert werden, in dem sie genutzt werden soll. Durch diese Schreibweise können alle Funktionen der Bibliothek über das Präfix `GPIO` angesprochen werden.

```
import time
```

Die Python-Bibliothek `time` hat nichts mit GPIO-Programmierung zu tun. Sie enthält Funktionen zur Zeit- und Datumsberechnung, unter anderem auch eine Funktion `time.sleep()`, mit der sich auf einfache Weise Wartezeiten realisieren lassen.

```
GPIO.setmode(GPIO.BCM)
```

Am Anfang jedes Programms muss definiert werden, wie die GPIO-Ports bezeichnet sind. Üblicherweise verwendet man die Standardnummerierung `BCM`.

```
GPIO.setup(25, GPIO.OUT)
```

Die Funktion `GPIO.setup()` initialisiert einen GPIO-Port als Ausgang oder als Eingang. Der erste Parameter bezeichnet den Port je nach vorgegebenem Modus `BCM` oder `BOARD` mit seiner GPIO-Nummer oder Pinnummer. Der zweite Parameter kann entweder `GPIO.OUT` für einen Ausgang oder `GPIO.IN` für einen Eingang sein.

Nummerierung der GPIO-Ports

Die Bibliothek `RPi.GPIO` unterstützt zwei verschiedene Methoden zur Bezeichnung der Ports. Im Modus `BCM` werden die bekannten GPIO-Portnummern genutzt, die auch auf Kommandozeilenebene oder in Shell-Skripten verwendet werden. Im alternativen Modus `BOARD` entsprechen die Bezeichnungen den Pinnummern von 1 bis 40 auf der Raspberry-Pi-Platine. Dieser Modus wird aber nur von der Python-Bibliothek unterstützt, nicht von Scratch und anderen Programmen.

```
for i in range(10):
```

Jetzt startet eine Schleife, die zehnmal durchläuft. `for`-Schleifen wie diese sind in vielen Programmiersprachen bekannt. Der Zähler `i` nimmt nacheinander von `0` aufsteigend ganze Zahlen an. Die Schleife wird beendet, wenn der bei `range` angegebene Wert erreicht ist. Der eingerückte Programmcode wird in jedem Schleifendurchlauf wiederholt.

```
    GPIO.output(25, 1)
```

Auf Port `25` wird eine `1` ausgegeben. Die angeschlossene LED leuchtet. Statt der `1` können auch die vordefinierten Werte `True` oder `GPIO.HIGH` ausgegeben werden.

Einrückungen sind in Python wichtig

In den meisten Programmiersprachen werden Programmschleifen oder Entscheidungen eingerückt, um den Programmcode übersichtlicher zu machen. In Python dienen diese Einrückungen nicht nur der Übersichtlichkeit, sondern sie sind für die Programmlogik zwingend erforderlich. Man braucht keine speziellen Satzzeichen, um Schleifen oder Entscheidungen zu beenden.

```
time.sleep(0.2)
```

Diese Funktion aus der am Anfang des Programms importierten `time`-Bibliothek bewirkt eine Wartezeit von 0,2 Sekunden, bevor das Programm weiterläuft.

```
GPIO.output(25, 0)
```

Zum Ausschalten der LED gibt man den Wert `0` bzw. `False` oder `GPIO.LOW` auf dem GPIO-Port aus.

```
time.sleep(0.2)
```

Auch bei ausgeschalteter LED wartet das Programm wieder 0,2 Sekunden. Danach ist die Schleife beendet, weil dies die letzte eingerückte Zeile ist. Die Schleife beginnt von Neuem, und die LED schaltet sich wieder kurz ein.

```
GPIO.cleanup()
```

Am Ende eines Programms müssen alle GPIO-Ports zurückgesetzt werden. Diese Zeile erledigt das für alle vom Programm initialisierten GPIO-Ports auf einmal. Ports, die von anderen Programmen initialisiert wurden, bleiben unberührt. So wird der Ablauf dieser anderen, möglicherweise parallel laufenden Programme nicht gestört.

Python-Programme brauchen keine eigene Anweisung zum Beenden. Sie enden einfach nach dem letzten Befehl bzw. nach einer Schleife, die nicht mehr ausgeführt wird und der keine weiteren Befehle folgen.

GPIO-Warnungen abfangen

Soll ein GPIO-Port verwendet werden, der nicht zurückgesetzt, sondern möglicherweise von einem anderen oder einem abgebrochenen Programm noch geöffnet ist, kommt es zu Warnungen, die den Programmfluss nicht unterbrechen. Bei der Programmierung sind diese Warnungen nützlich, um Fehler zu finden. Im fertigen Programm können sie den Anwender verwirren. Die GPIO-Bibliothek bietet mit `GPIO.setwarnings(False)` die Möglichkeit, diese Warnungen zu unterdrücken.

2 Das erste Projekt mit Scratch

Scratch ist eine intuitive Programmierumgebung, mit der Kinder und Programmiereinsteiger schnell Ideen umsetzen können, ohne sich vorher mit Programmiertheorie auseinandersetzen zu müssen. Passend für die aus Kindern und Jugendlichen bestehende Zielgruppe eignet sich Scratch besonders für interaktive Animationen und Spiele. Eine grafische Oberfläche gibt bereits alle Grundlagen vor, sodass man sich um die Programmierung der Benutzeroberfläche des eigenen Programms keine Gedanken zu machen braucht. Die Scratch-Skripte werden nicht als Text geschrieben, sondern aus vorgefertigten Elementen zusammengeklickt.

Scratch ist seit der ersten Betriebssystemversion, damals noch als Raspbian bezeichnet, vorinstalliert. Speziell für den Raspberry Pi 4 wurde eine Offlineversion des aktuellen Scratch 3 entwickelt. Dafür ist ein Raspberry Pi 4 mit mindestens 2 GB RAM erforderlich.

Starten Sie *Scratch 3* über das Menü *Entwicklung*. Scratch startet mit einem viergeteilten Programmfenster:

Klicken Sie oben links auf die Weltkugel und schalten Sie die Benutzeroberfläche auf Deutsch um.

1. Links befindet sich die Blockpalette mit allen Elementen, aus denen ein Scratch-Skript zusammengesetzt werden kann. Da ausschließlich vordefinierte Blöcke verwendet werden, kann der Benutzer keine Syntaxfehler begehen, die beim Einstieg in andere Programmiersprachen sehr ärgerlich und frustrierend sein können.

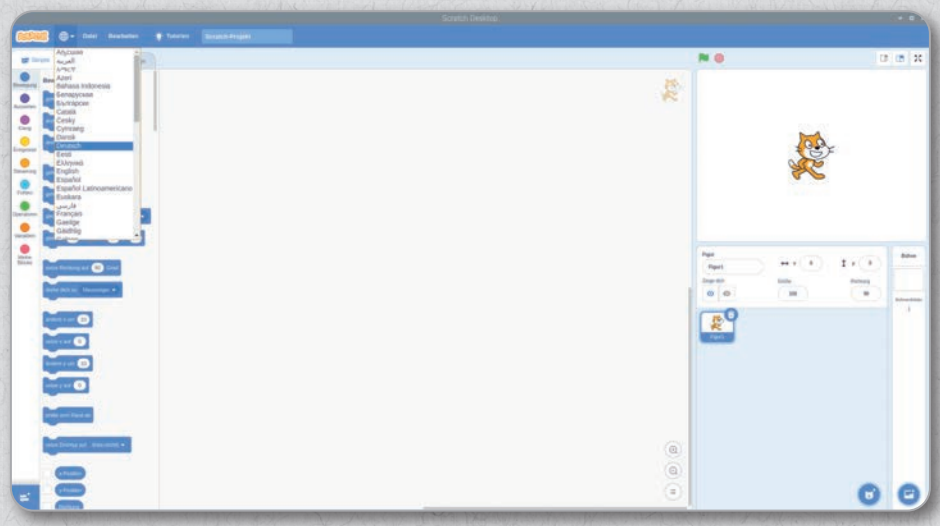


Abb. 2.1: *Scratch 3 starten.*

2. Das mittlere Feld ist am Anfang noch leer. Hier entsteht später das Skript.

3. Rechts oben ist die sogenannte Bühne, die Oberfläche, auf der das Skript abläuft. Die Katze, die dort zu sehen ist, stellt beispielhaft ein Objekt dar, das mit Scratch animiert werden kann.

4. Das Objektfenster unten rechts zeigt die im Skript verwendeten Objekte. Hier können Sie später auch selbst eigene Objekte gestalten.

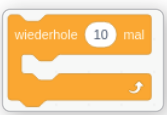
In einem allerersten einfachen Skript soll die Katze einmal im Kreis herumlaufen und dabei ihre Farbe verändern.

1. Klicken Sie im Objektfenster unten rechts auf die Katze. Diese wird nun hervorgehoben und erscheint als *Figur1* im Objekt- oder Figurenfenster. Alle Befehle im Skriptfenster beziehen sich dann auf dieses Objekt.

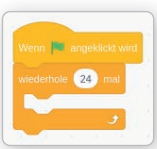
2. Klicken Sie in der Blockpalette oben auf das gelbe Symbol *Ereignisse*. Damit werden die Blöcke angezeigt, die auf Ereignisse reagieren.

3. Ziehen Sie den abgebildeten Block aus der Blockpalette in das Skriptfenster. Auf der Scratch-Bühne befindet sich oben links ein grünes Fähnchen. Es dient üblicherweise dazu, ein Programm zu starten. Das abgebildete Element bewirkt, dass die folgenden Skriptelemente ausgeführt werden, wenn der Benutzer auf das grüne Fähnchen klickt.

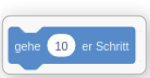
4. Die kreisförmige Bewegung wird aus einzelnen Gehen- und Drehen-Schritten zusammengesetzt. Diese werden so oft wiederholt, bis die Katze einen ganzen Kreis gegangen ist. Klicken Sie in der Blockpalette links außen auf das orangefarbene Symbol *Steuerung*. Ziehen Sie für die Wiederholungsschleife den abgebildeten Block in das Skriptfenster und docken Sie ihn unten an das dort bereits vorhandene Element an.



5. In jedem Bewegungsschritt soll sich die Katze um 15° drehen. Dabei dreht sie sich in 24 Schritten genau einmal um sich selbst. Tippen Sie in das weiße Zahlenfeld des *Wiederhole*-Blocks und ändern Sie den vorgegebenen Wert auf *24*.



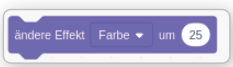
6. Damit die Katze eine Kreisbewegung ausführt, muss sie zuerst einen Schritt gehen, sich danach um 15° drehen, wieder einen Schritt gehen usw. Klicken Sie links in der Blockpalette auf das blaue Symbol *Bewegung* und ziehen Sie den abgebildeten Block in das Skriptfenster. Docken Sie ihn innerhalb der Schleife an. Ändern Sie dann noch die Schrittweise auf *20*.



7. Ziehen Sie danach den Block für die Drehbewegung gegen den Uhrzeigersinn in das Skriptfenster. Platzieren Sie ihn so über den vorhandenen Blöcken, dass er sich innerhalb der Schleife nach der Gehbewegung einklinkt.

8. Das Skript sollte jetzt wie abgebildet aussehen. Probieren Sie aus, ob es auch wie erwartet funktioniert. Klicken Sie dazu rechts oberhalb der Bühne auf das grüne Fähnchen. Die Katze wird eine Kreisbewegung gehen.

9. Nun fehlt nur noch die gewünschte Veränderung der Farbe. Klicken Sie dazu links in der Blockpalette auf das violette Symbol *Aussehen*. Jetzt werden Blöcke angeboten, die das Aussehen des aktiven Objekts beeinflussen. Ziehen Sie den abgebildeten Block *ändere Effekt Farbe um …* in das Skriptfenster in die Wiederholung hinter den *Drehe*-Block.



10. Lassen Sie das Skript jetzt wieder ablaufen, ändert die Katze im Lauf ihrer Bewegung zyklisch ihre Farbe. Am Ende der 24 Wiederholungen steht die Katze wieder an ihrer ursprünglichen Position und hat auch wieder ihre ursprüngliche Farbe.

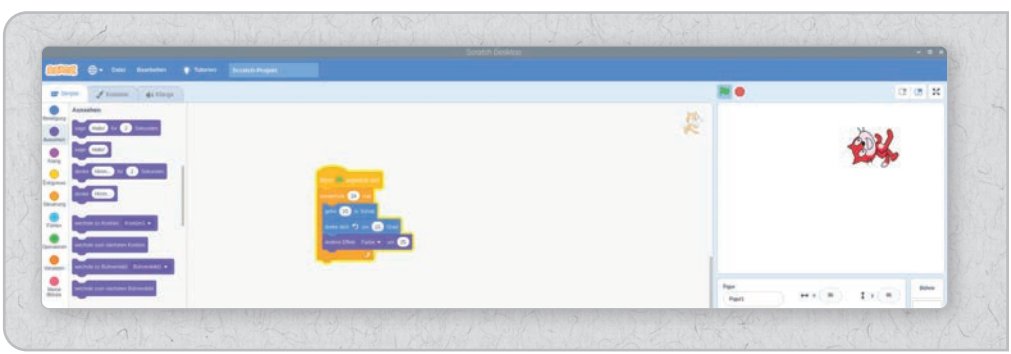


Abb. 2.2: *Gehen, drehen, Farbe ändern.*

3 Scratch und GPIO

Scratch bietet auf den ersten Blick keine Unterstützung für die GPIO-Schnittstelle. Dazu müssen Erweiterungen eingebunden werden. Wir verwenden für die folgenden Experimente die Erweiterung *Raspberry Pi GPIO*, die grundlegende Unterstützung für verschiedene Hardwarekomponenten am GPIO-Port bietet.

Klicken Sie auf das blaue Symbol unten links in Scratch 3, um Erweiterungen zu installieren. Wählen Sie auf dem nächsten Bildschirm die Erweiterung *Raspberry Pi GPIO*.

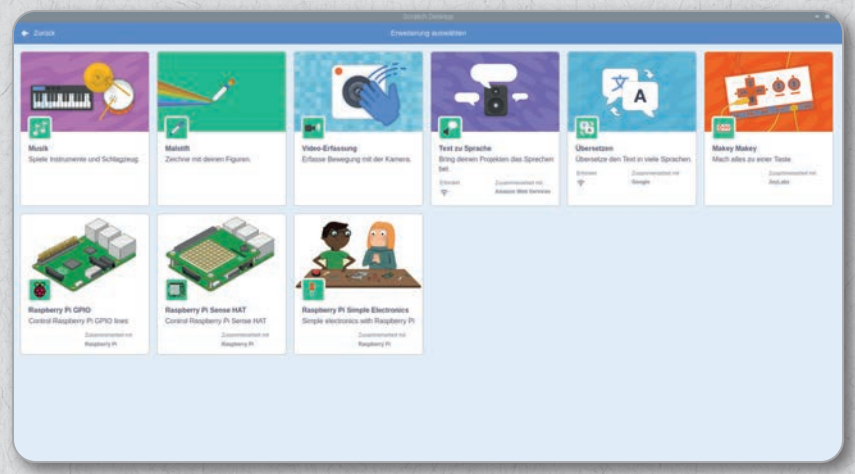


Abb. 3.1: *Erweiterungen auswählen und installieren.*

Auf einer neuen Blockpalette ganz unten erscheinen vier Scratch-Blöcke zur Steuerung von GPIO-Pins.

Abb. 3.2: *Die Erweiterung Raspberry Pi GPIO.*

3.1 | Die erste LED blinkt in Scratch

Im Vergleich mit dem Versuch, mit einem Windows-PC eine extern angeschlossene LED zum Blinken zu bringen, ist Scratch wirklich ganz einfach. Bauen Sie die abgebildete Schaltung auf einer Steckplatine auf.

Benötigte Bauteile

- 1 x Steckplatine,
- 1 x LED rot,
- 2 x Verbindungskabel

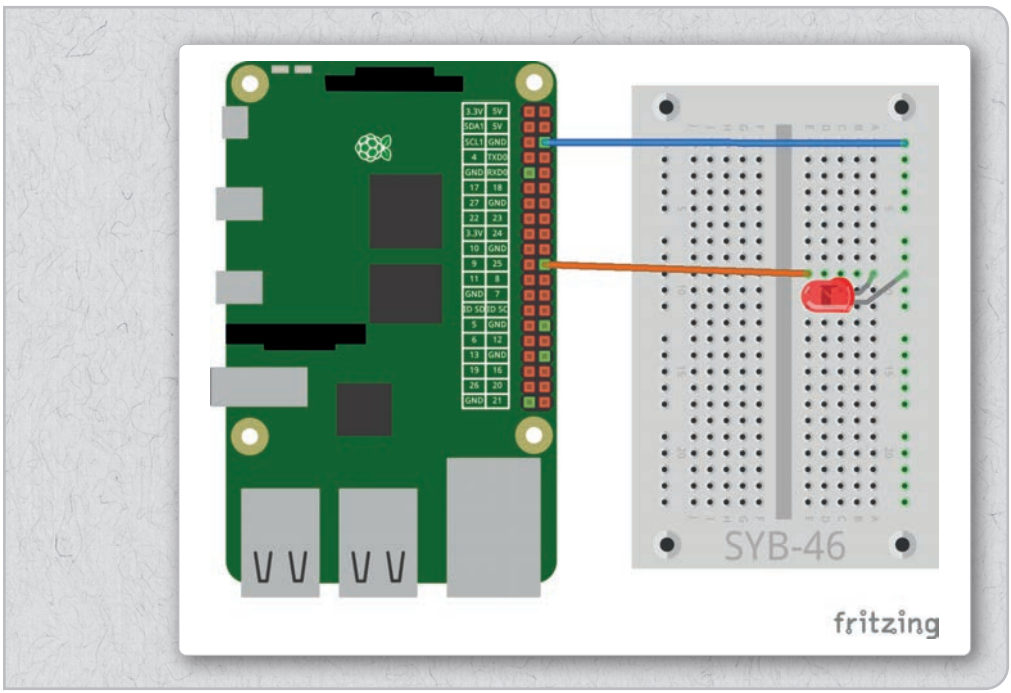


Abb. 3.3: *Die Schaltung entspricht der Schaltung aus dem letzten Python-Programm.*

Öffnen Sie über den Menüpunkt *Datei/Hochladen von deinem Computer* das Skript `031ed02` in Scratch und starten Sie es mit einem Klick auf das grüne Fähnchen. Die LED blinkt. Sie können das Programm natürlich auch selbst aus Scratch-Blöcken zusammensetzen.

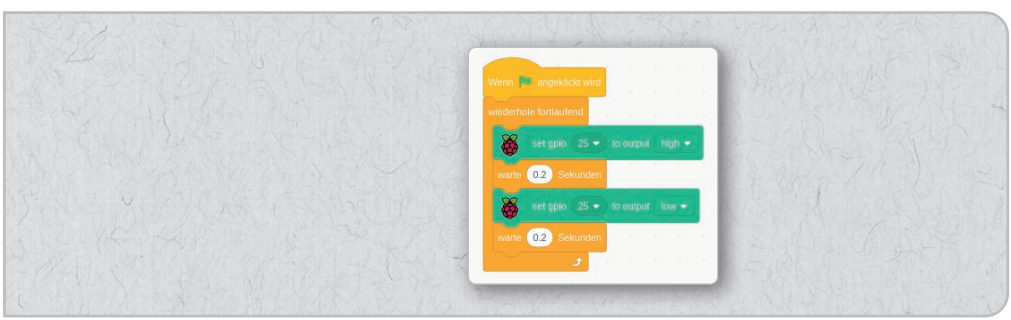


Abb. 3.4: *Dieses Programm lässt die LED blinken.*

3.1.1 | So funktioniert es

Das Programm startet mit einem Klick auf das grüne Fähnchen eine Schleife vom Typ *Wiederhole fortlaufend*.

Diese Schleife wird so lange wiederholt, bis der Benutzer auf das rote Stoppsymbol oberhalb der Scratch-Bühne klickt.

Innerhalb der Schleife schaltet ein Block *set gpio 25 to output high* den GPIO-Pin 25 auf *High*. Die LED leuchtet. In diesem Block kann der GPIO-Pin frei ausgewählt werden. Im hinteren Listenfeld wählen Sie zwischen *High* und *Low*, in Python als 1 und 0 bezeichnet.

Anschließend wartet das Programm 0,2 Sekunden, bis der nächste Block *set gpio 25 to output low* den GPIO-Pin 25 auf *Low* setzt. Auch bei ausgeschalteter LED wartet das Programm 0,2 Sekunden. Danach startet die Endlosschleife neu und schaltet die LED wieder ein.

4 Fußgängerampel

Eine einzelne LED ein- und wieder auszuschalten, mag im ersten Moment ganz spannend sein, aber dafür braucht man eigentlich keinen Computer. Eine Verkehrsampel mit ihrem typischen Leuchtzyklus von Grün über Gelb nach Rot und dann über eine Lichtkombination Rot-Gelb wieder zu Grün zeigt weitere Programmiertechniken in Python. Das nächste Experiment

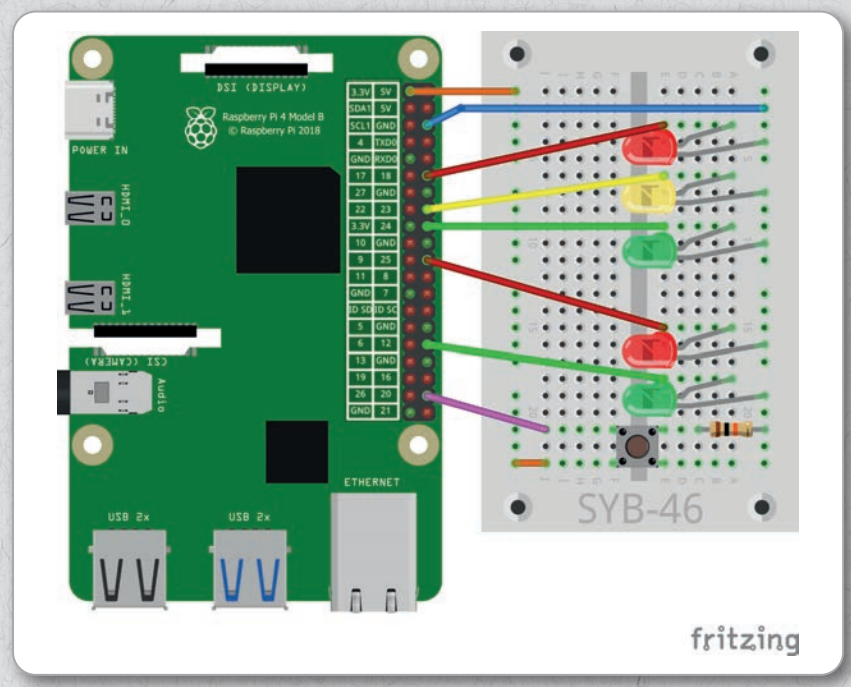


Abb. 4.1: *Verkehrsampel mit Fußgängerampel.*

stellt eine einfache Ampelschaltung mit Fußgängerampel, die während der Rotphase der Verkehrsampel eine Grünphase für Fußgänger anzeigt, auf einem Steckbrett mit fünf LEDs dar.

Bauen Sie für das folgende Experiment fünf LEDs sowie einen Taster wie abgebildet auf einer Steckplatine auf.

Benötigte Bauteile

- 1 x Steckplatine,
- 2 x LED rot,
- 1 x LED gelb,
- 2 x LED grün,
- 1 x Taster,
- 1 x 10-kOhm-Widerstand (Braun-Schwarz-Orange),
- 8 x Verbindungskabel,
- 1 x kurze Drahtbrücke

4.1 | Taster am GPIO-Anschluss

GPIO-Ports können nicht nur Daten ausgeben, zum Beispiel über LEDs, sie können auch zur Dateneingabe verwendet werden. Dazu müssen sie im Programm als Eingang definiert werden. Zur Eingabe verwenden wir im nächsten Projekt einen Taster, der direkt auf die Steckplatine gesteckt wird. Der Taster hat vier Anschlusspins, wobei jeweils die beiden einander gegenüberliegenden (großer Abstand) miteinander verbunden sind. Solange die Taste gedrückt ist, sind alle vier Anschlüsse miteinander verbunden. Im Gegensatz zu einem Schalter rastet ein Taster nicht ein. Die Verbindung wird beim Loslassen sofort wieder getrennt.

Liegt auf einem als Eingang definierten GPIO-Port ein +3,3-V-Signal an, wird es als logisch `True` bzw. `1` ausgewertet. Theoretisch könnten Sie also über einen Taster den jeweiligen GPIO-Port mit dem +3,3-V-Anschluss des Raspberry Pi verbinden.

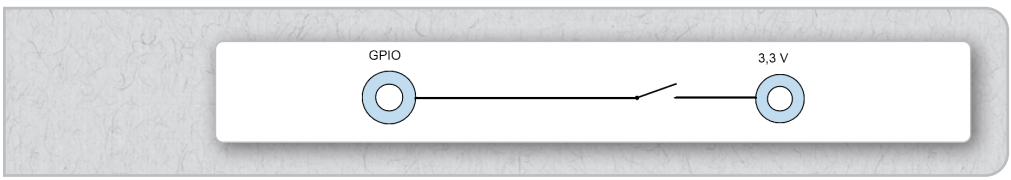


Abb. 4.2: *Ein Taster verbindet einen GPIO-Eingang mit +3,3 V.*

In den meisten Fällen funktioniert diese simple Schaltung bereits, allerdings hätte der GPIO-Port bei offenem Taster keinen eindeutig definierten Zustand. Wenn ein Programm diesen Port abfragt, kann es zu zufälligen Ergebnissen kommen. Um das zu verhindern, schließt man einen vergleichsweise sehr hohen Widerstand – üblicherweise 10 kOhm – gegen Masse. Dieser sogenannte Pull-down-Widerstand zieht den Status des GPIO-Ports bei geöffnetem Taster wieder nach unten auf 0 V. Da der Widerstand sehr hoch ist, besteht, solange der Taster gedrückt ist, auch keine Kurzschlussgefahr. Ist der Taster gedrückt, sind +3,3 V und die Masseleitung direkt über diesen Widerstand verbunden.

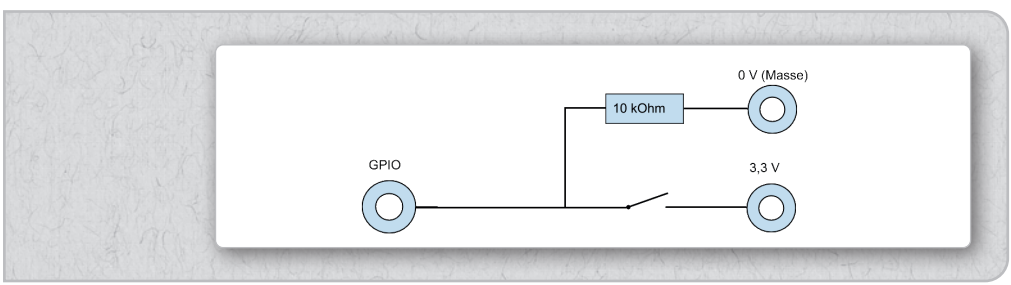


Abb. 4.3: *Taster mit Schutzwiderstand und Pull-down-Widerstand an einem GPIO-Eingang.*

Die in der Abbildung der aufgebauten Schaltung untere Kontaktleiste des Tasters ist über die Plusschiene der Steckplatine mit der +3,3-V-Leitung des Raspberry Pi (Pin 1) verbunden. Zur Verbindung des Tasters mit

der Plusschiene verwenden wir, um die Zeichnung übersichtlich zu halten, eine kurze Drahtbrücke. Alternativ können Sie einen der unteren Kontakte des Tasters direkt mit einem Verbindungskabel mit Pin 1 des Raspberry Pi verbinden.

Die in der Abbildung obere Kontaktleiste des Tasters ist mit dem GPIO-Port 20 und über einen 10-kOhm-Pull-down-Widerstand (Braun-Schwarz-Orange) mit 0 V Masse verbunden.

4.2 | Fußgängerampel mit Python

Im Ruhezustand soll die Verkehrsampel Grün zeigen, die Fußgängerampel Rot. Drückt man auf die Taste, schaltet die Verkehrsampel nacheinander über Gelb auf Rot. Danach erst schaltet die Fußgängerampel auf Grün. Nach einer Grünphase von zwei Sekunden schaltet die Fußgängerampel wieder auf Rot. Dann beginnt der Schaltzyklus der Verkehrsampel über Rot-Gelb nach Grün. Anschließend wartet das Programm mit dieser Einstellung, bis der Benutzer erneut die Taste drückt. Das Programm `04ampel1.py` steuert die Ampelanlage.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import RPi.GPIO as GPIO
import time

rot    = 0
gelb   = 1
gruen  = 2
f_rot  = 3
f_gru  = 4
taste  = 5
Ampel=[18,23,24,25,12,20]

GPIO.setmode(GPIO.BCM)
GPIO.setup(Ampel[rot], GPIO.OUT, initial=0)
GPIO.setup(Ampel[gelb], GPIO.OUT, initial=0)
GPIO.setup(Ampel[gruen], GPIO.OUT, initial=1)
```

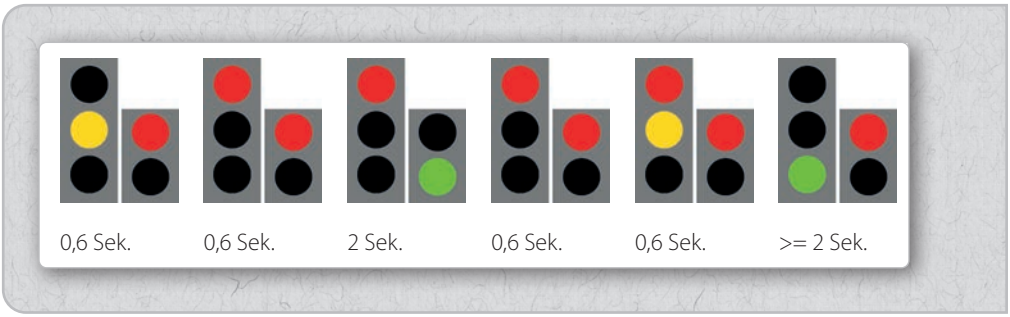
```
GPIO.setup(Ampel[f_rot], GPIO.OUT, initial=1)
GPIO.setup(Ampel[f_gru], GPIO.OUT, initial=0)
GPIO.setup(Ampel[taste], GPIO.IN)

print ("Taste drücken, um Fußgängerampel einzuschalten,
Strg+C beendet das Programm")

try:
    while True:
        if GPIO.input(Ampel[taste])==1:
            GPIO.output(Ampel[gruen],0)
            GPIO.output(Ampel[gelb],1)
            time.sleep(0.6)
            GPIO.output(Ampel[gelb],0)
            GPIO.output(Ampel[rot],1)
            time.sleep(0.6)
            GPIO.output(Ampel[f_rot],0)
            GPIO.output(Ampel[f_gru],1)
            time.sleep(2)
            GPIO.output(Ampel[f_gru],0)
            GPIO.output(Ampel[f_rot],1)
            time.sleep(0.6)
            GPIO.output(Ampel[gelb],1)
            time.sleep(0.6)
            GPIO.output(Ampel[rot],0)
            GPIO.output(Ampel[gelb],0)
            GPIO.output(Ampel[gruen],1)
            time.sleep(2)
except KeyboardInterrupt:
    GPIO.cleanup()
```

Die Verkehrsampel leuchtet grün, die Fußgängerampel rot, genau so, wie es echte Ampeln stundenlang tun würden, wenn kein Fußgänger kommt und auf den Knopf drückt.

Drücken Sie auf den Taster. Jetzt startet der Ampelzyklus, der in unserem Programm, wie auch bei einer echten Ampel, aus sechs unterschiedlichen Lichtmustern besteht, die unterschiedlich lang leuchten.



Mit dem letzten Lichtmuster – Fußgängerampel rot, Verkehrsampel grün – erreicht die Ampel wieder den Standardzustand. Das Programm muss allerdings dafür sorgen, dass auch für diesen Zustand immer eine Mindestzeit eingehalten wird. Selbst wenn ständig Fußgänger auf den Knopf drücken, müssen die Autos auch mal fahren dürfen. In unserer Modellampel sind das zwei Sekunden, bei einer wirklichen Ampel ist das Intervall natürlich deutlich länger.

4.2.1 | So funktioniert es

Die erste Zeile des Programms ist neu:

```
# -*- coding: utf-8 -*-
```

Damit die deutschen Umlaute im Wort `Fußgängerampel` in der Programmausgabe korrekt angezeigt werden – unabhängig davon, wie die IDLE-Oberfläche beim Benutzer eingestellt ist –, wird am Anfang eine Codierung zur Darstellung der Sonderzeichen definiert. Diese Zeile sollte in allen Programmen enthalten sein, die Texte ausgeben, in denen sich Umlaute oder andere landestypische Sonderzeichen befinden.

Die folgenden Zeilen sind bereits bekannt, sie importieren die Bibliotheken `RPi.GPIO` für die Ansteuerung der GPIO-Ports und `time` für Zeitverzögerungen.

```
rot    = 0
gelb   = 1
gruen  = 2
f_rot  = 3
f_gru  = 4
taste  = 5
```

Diese Zeilen definieren die drei Variablen `rot`, `gelb`, `gruen` für die drei LEDs der Verkehrsampel, `f_rot`, `f_gru` für die LEDs der Fußgängerampel sowie `taste` für den Taster. Damit braucht man sich im Programm keine Nummern der GPIO-Ports zu merken, sondern kann die LEDs einfach über ihre Farben ansteuern.

```
Ampel=[18,23,24,25,12,20]
```

Zur Ansteuerung der LEDs und des Tasters wird eine Liste eingerichtet, die die GPIO-Nummern in der Reihenfolge enthält, in der sie zuvor definiert wurden. Da die GPIO-Ports nur an dieser einen Stelle im Programm auftauchen, können Sie das Programm ganz einfach umbauen, wenn Sie andere GPIO-Ports nutzen möchten. Danach wird die Nummerierung der GPIO-Ports auf `BCM` gesetzt.

```
GPIO.setmode(GPIO.BCM)
GPIO.setup(Ampel[rot], GPIO.OUT, initial=0)
GPIO.setup(Ampel[gelb], GPIO.OUT, initial=0)
GPIO.setup(Ampel[gruen], GPIO.OUT, initial=1)
GPIO.setup(Ampel[f_rot], GPIO.OUT, initial=1)
GPIO.setup(Ampel[f_gru], GPIO.OUT, initial=0)
GPIO.setup(Ampel[taste], GPIO.IN)
```

Nacheinander werden die verwendeten GPIO-Ports als Ausgänge initialisiert. Dabei verwenden wir keine GPIO-Portnummern, sondern die zuvor definierte Liste. Innerhalb einer Liste werden die einzelnen Elemente über Zahlen, mit 0 beginnend, indiziert. Die Variablen `rot`, `gelb` und `gruen` enthalten die Zahlen `0`, `1` und `2`, die als Indizes für die Elemente der Liste benötigt werden. Auf diese Weise lassen sich die verwendeten GPIO-Ports über Farben adressieren:

- `Ampel[rot]` entspricht GPIO-Port `18` mit der roten LED.
- `Ampel[gelb]` entspricht GPIO-Port `23` mit der gelben LED.
- `Ampel[gruen]` entspricht GPIO-Port `24` mit der grünen LED.
- `Ampel[f_rot]` entspricht GPIO-Port `25` mit der roten LED der Fußgängerampel.
- `Ampel[f_gru]` entspricht GPIO-Port `12` mit der grünen LED der Fußgängerampel.
- `Ampel[taste]` entspricht GPIO-Port `20` mit der Taste.

Der GPIO-Port des Tasters wird als Eingang definiert. Diese Definition erfolgt ebenfalls über `GPIO.setup`, diesmal aber mit dem Parameter `GPIO.IN`.

Die `GPIO.setup`-Anweisung kann einen optionalen Parameter `initial` enthalten, der dem GPIO-Port bereits beim Initialisieren einen logischen Status zuweist. Damit schalten wir in diesem Programm die grüne LED der Verkehrsampel sowie die rote LED der Fußgängerampel bereits von Anfang an ein. Die anderen LEDs beginnen das Programm im ausgeschalteten Zustand.

```
print ("Taster drücken, um Fußgängerampel einzuschalten,
Strg+C beendet das Programm")
```

Jetzt wird eine kurze Bedienungsanleitung auf dem Bildschirm ausgegeben, die mitteilt, dass der Benutzer den Taster drücken soll. Das Programm läuft automatisch. Die Tastenkombination Strg+C soll es beenden.

Um abzufragen, ob der Benutzer mit Strg+C das Programm beendet, verwenden wir eine `try...except`-Abfrage. Dabei wird der unter `try:` eingetragene Programmcode zunächst normal ausgeführt. Wenn währenddessen eine Systemausnahme auftritt – das kann ein Fehler sein oder auch die Tastenkombination Strg+C –, wird abgebrochen, und die `except`-Anweisung am Programmende wird ausgeführt.

```
except KeyboardInterrupt:
  GPIO.cleanup()
```

Durch die Tastenkombination Strg+C wird ein `KeyboardInterrupt` ausgelöst und die Schleife automatisch verlassen. Die letzte Zeile schließt die verwendeten GPIO-Ports und schaltet damit alle LEDs aus. Danach wird das Programm beendet.

Durch das kontrollierte Schließen der GPIO-Ports tauchen keine Systemwarnungen oder Abbruchmeldungen auf, die den Benutzer verwirren könnten. Der eigentliche Ampelzyklus läuft in einer Endlosschleife:

```
while True :
```

Solche Endlosschleifen benötigen immer eine Abbruchbedingung, da das Programm sonst nie beendet werden würde.

```
    if GPIO.input(Ampel[taste])==1:
```

Innerhalb der Endlosschleife ist eine Abfrage eingebaut. Die folgenden Anweisungen werden erst ausgeführt, wenn der GPIO-Port mit dem Taster den Wert `True` annimmt, der Benutzer also den Taster drückt. Bis zu diesem Zeitpunkt bleibt die Verkehrsampel in ihrer Grünphase stehen.

```
        GPIO.output(Ampel[gruen],0)
        GPIO.output(Ampel[gelb],1)
        time.sleep(0.6)
```

Jetzt wird die grüne LED aus- und dafür die gelbe LED eingeschaltet. Diese leuchtet dann allein für 0,6 Sekunden.

```
        GPIO.output(Ampel[gelb],0)
        GPIO.output(Ampel[rot],1)
        time.sleep(0.6)
```

Jetzt wird die gelbe LED wieder ausgeschaltet und dafür die rote LED eingeschaltet, und erneut wird 0,6 Sekunden gewartet.

```
        GPIO.output(Ampel[f_rot],0)
        GPIO.output(Ampel[f_gru],1)
        time.sleep(2)
```

Erst danach schaltet die Fußgängerampel von Rot auf Grün um. Diese Phase dauert zwei Sekunden. Danach schaltet die Fußgängerampel auf Rot, und mit jeweils 0,6 Sekunden Verzögerung schaltet die Verkehrsampel über Rot-Gelb auf Grün um.

Am Ende der Schleife leuchtet die Verkehrsampel wieder grün, die Fußgängerampel rot. Die Schleife beginnt in der Grünphase der Ampel nach einer Wartezeit von zwei Sekunden von Neuem. Um zu verhindern, dass die Grünphase fast ausfällt, wenn der Taster unmittelbar nach der Gelbphase wieder gedrückt wird, ist diese Verzögerung am Ende der Schleife eingebaut. Natürlich können Sie alle Zeiten beliebig anpassen. In der Realität hängen die Ampelphasen von den Maßen der Kreuzung und den Verkehrsströmen ab. Die Gelb- und die Rot-Gelb-Phase sind üblicherweise je zwei Sekunden lang.

4.2.2 | Internen Pull-down-Widerstand nutzen

Der Raspberry Pi 4 verfügt über eingebaute Pull-down-Widerstände an den GPIO-Ports, die per Software eingeschaltet werden können. Auf diese Weise kann man sich externe Pull-down-Widerstände sparen.

Entfernen Sie den 10-kOhm-Pull-down-Widerstand aus der Schaltung. Der Ampelzyklus wird immer mal wieder zufällig starten, ohne dass Sie den Taster zu drücken brauchen.

Der interne Pull-down-Widerstand wird bei der Initialisierung des GPIO-Eingangs über einen zusätzlichen Parameter in der Funktion `GPIO.setup()` eingeschaltet.

```
GPIO.setup(Ampel[taste], GPIO.IN, GPIO.PUD_DOWN)
```

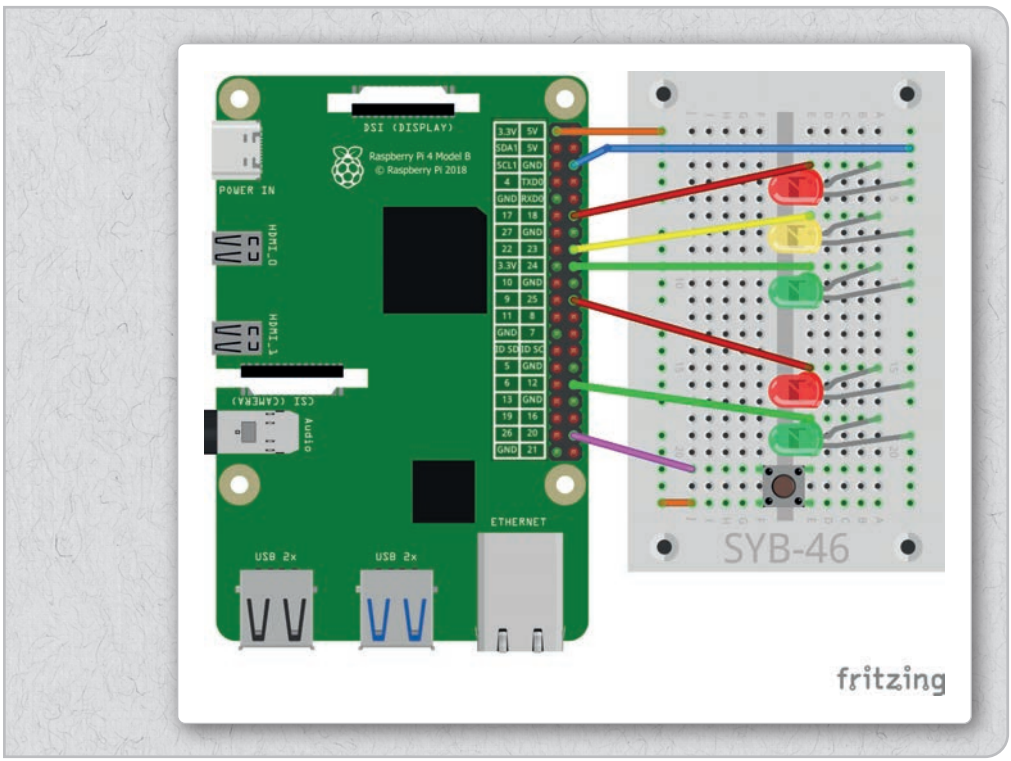


Abb. 4.4: *Die Ampel ohne Pull-down-Widerstand.*

Das Programm `04ampel2.py` enthält diese Änderung bereits. Starten Sie das Programm, und die Ampel lässt sich mit dem Taster steuern, auch ohne dass ein externer Pull-down-Widerstand eingebaut ist.

5 Fußgängerampel mit Scratch

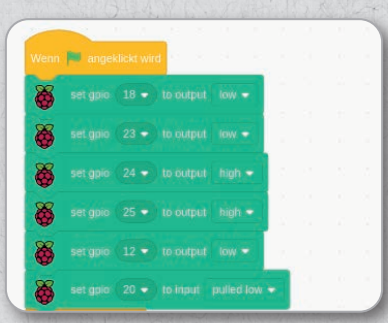
Die gleiche Fußgängerampel lässt sich auch mit Scratch programmieren. Da die Initialisierung der GPIO-Pins in Scratch einfacher ist, wirkt das Scratch-Programm `05ampel` noch etwas übersichtlicher als die Python-Version.

Um mehr Platz für das Skript zu haben, können Sie mit den Symbolen rechts unten im Skriptfenster zoomen.

Abb. 5.1: *Das Scratch-Skript für die Fußgängerampel.*

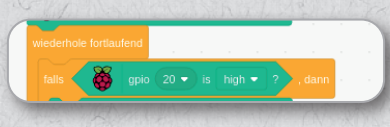
5.1 | So funktioniert es

Wenn der Benutzer auf das grüne Fähnchen klickt, werden als Erstes die sechs verwendeten GPIO-Pins initialisiert. Wie in der Python-Version der Ampel verwenden wir die Ports 18, 23, 24, 25 und 12 als Ausgänge für die LEDs und den Port 20 mit eingeschaltetem Pull-down-Widerstand als Eingang für den Taster.



Bei der Initialisierung werden die grüne LED der Verkehrsampel und die rote LED der Fußgängerampel eingeschaltet. Die anderen drei LEDs werden ausgeschaltet. Im Block *set gpio … to input …* wählen Sie im Listenfeld die Option *pulled low*, um den internen Pull-down-Widerstand einzuschalten.

Jetzt beginnt wie im Python-Skript eine Endlosschleife, die darauf wartet, dass der Benutzer die Taste drückt, und dann den Ampelzyklus startet.



An dieser Stelle wird der Status des Tasters abgefragt.

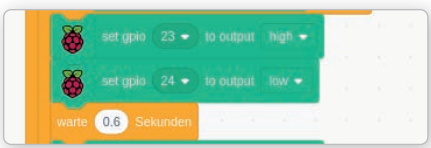
Dazu verwenden wir den Scratch-Block *falls*, der ähnlich wie eine `if`-Abfrage in Python funktioniert.

Innerhalb des Blocks werden die Blöcke eingefügt, die ausgeführt werden sollen, wenn die Abfrage wahr ist.

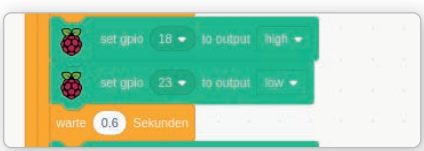
Für die Abfrage selbst ist im *falls*-Block ein längliches Feld mit spitzen Enden vorgesehen. Hier muss ein Block mit dieser Form eingefügt werden. Wählen Sie den Block *gpio … is high* aus der *Raspberry Pi GPIO*-Erweiterung und ziehen Sie ihn auf das Platzhalterfeld im Block *falls*.

In unserem Fall soll GPIO-Pin 20 *high* sein. Ein gedrückter Taster gibt auch hier, wie in Python, den Wert *high* zurück.

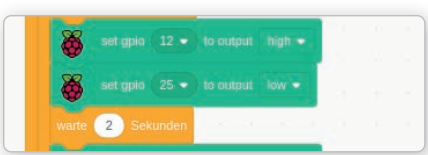
Wenn diese Abfrage wahr wird, der Benutzer also den Taster gedrückt hat, beginnt der Ampelzyklus. Im ersten Schritt wird die grüne LED der Verkehrsampel (GPIO 24) ausgeschaltet und dafür die gelbe (GPIO 23) eingeschaltet. Danach wartet das Skript 0,6 Sekunden.



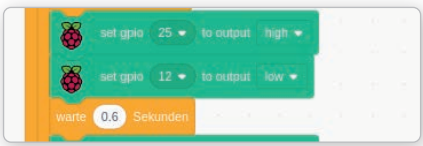
Nach der Gelbphase schaltet die Verkehrsampel auf Rot. Dazu wird die gelbe LED (GPIO 23) ausgeschaltet und dafür die rote (GPIO 18) eingeschaltet. Auch diese Phase dauert 0,6 Sekunden.



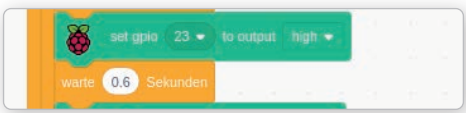
Im nächsten Schritt schaltet die Fußgängerampel auf Grün, wobei die Verkehrsampel unverändert Rot anzeigt. Die grüne LED der Fußgängerampel (GPIO 12) wird eingeschaltet, die rote (GPIO 25) aus. Die Grünphase der Fußgängerampel dauert zwei Sekunden.



Am Ende der Grünphase wird die rote LED der Fußgängerampel (GPIO 25) wieder eingeschaltet, die grüne (GPIO 12) aus. Diese Schaltphase dauert 0,6 Sekunden.



Die rote LED der Verkehrsampel leuchtet an dieser Stelle bereits. Für die jetzt folgende Rot-Gelb-Phase braucht nur die gelbe LED (GPIO 23) zusätzlich eingeschaltet zu werden.

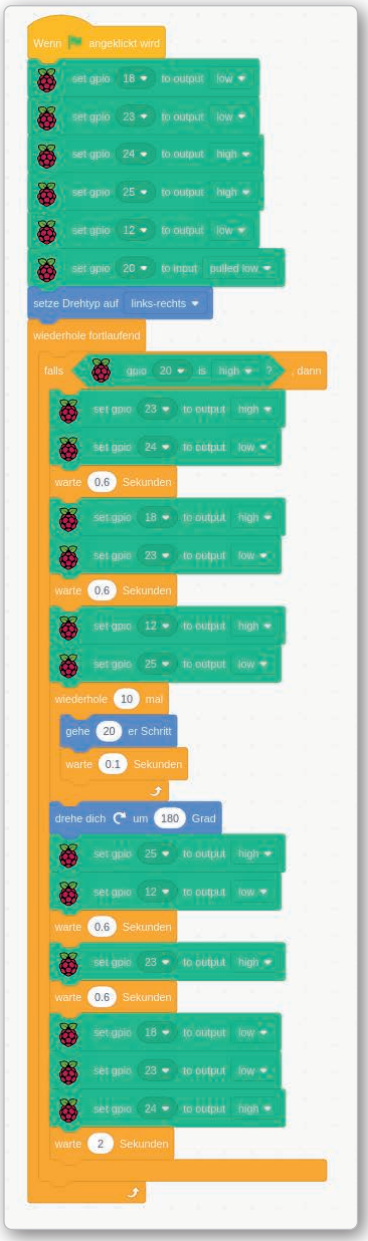


Nach der Rot-Gelb-Phase werden die rote LED (GPIO 18) und die gelbe LED (GPIO 23) der Verkehrsampel ausgeschaltet und die grüne LED (GPIO 24) eingeschaltet.

Nach dem Start der Grünphase wartet die Ampelschaltung zwei Sekunden, bevor die Befehlsfolge innerhalb der *falls*-Abfrage beendet wird. Damit wird verhindert, dass die Grünphase fast ausfällt, wenn der Taster unmittelbar nach der Gelbphase wieder gedrückt wird.

Danach startet die Endlosschleife neu und ruft je nach ermitteltem Sensorwert den Ampelzyklus auf.

5.2 | Die Katze bewegt sich zur Ampel



Im nächsten Schritt wird sich die Katze, die Symbolfigur von Scratch, passend zur Ampelschaltung bewegen und damit zeigen, dass sich klassische Scratch-Funktionen mit GPIO-Blöcken in einem Skript frei kombinieren lassen.

Das Skript `05ampel_katze` enthält dazu ein paar zusätzliche Blöcke.

Die Ampelsteuerung entspricht der in der vorherigen Version des Skripts. Lediglich während der Grünphase der Fußgängerampel werden ein paar neue Blöcke eingefügt, um die Katze zu bewegen.

Sobald die Fußgängerampel auf Grün schaltet, soll die Katze von ihrer aktuellen Position auf die andere Straßenseite gehen und sich dort um 180° drehen, um in der nächsten Grünphase der Fußgängerampel wieder zurückzugehen.

Anstatt einen 200er-Schritt auf einmal zu gehen oder eher zu springen, wird die Bewegung in zehn 20er-Schritte aufgeteilt, nach denen die Katze jeweils 0,1 Sekunden wartet. Durch diese Wiederholung erscheint die Bewegung flüssiger. Diese Blöcke ersetzen die Wartezeit von zwei Sekunden, sie dauern zusammen genauso lang.

Nach zehn Wiederholungen soll sich die Katze um 180° drehen. Damit sie nach der Drehung nicht auf dem Kopf steht, legt ein Block *setze Drehtyp auf links-rechts* am Anfang des Programms bei den Initialisierungen den Drehtyp für die aktuelle Figur fest.

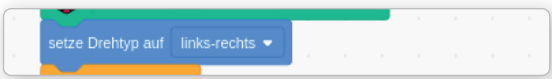


Abb. 5.2: *Dieser Block richtet die Katze nach rechts und links aus, anstatt sie frei zu drehen.*

Wenn Sie jetzt das Skript starten und auf die Taste der Ampel drücken, läuft die Katze in der ersten Grünphase der Fußgängerampel nach rechts und dreht sich dort. Beim nächsten Drücken läuft sie wieder nach links auf ihre Startposition. Da nur relative Bewegungen verwendet werden, braucht man im Skript nicht zwischen dem Gang nach rechts und dem nach links zu unterscheiden.

6 Spielwürfel mit LEDs

Die typischen Spielwürfel, die eins bis sechs Augen zeigen, kennt jeder und hat jeder zu Hause. Wesentlich cooler ist ein elektronisch gesteuerter Würfel, der auf Tastendruck die Augen leuchten lässt – aber nicht einfach eine bis sechs LEDs in einer Reihe, sondern in der Anordnung eines Spielwürfels. Diese haben Augen in der typischen quadratischen Anordnung, wozu man sieben LEDs braucht.

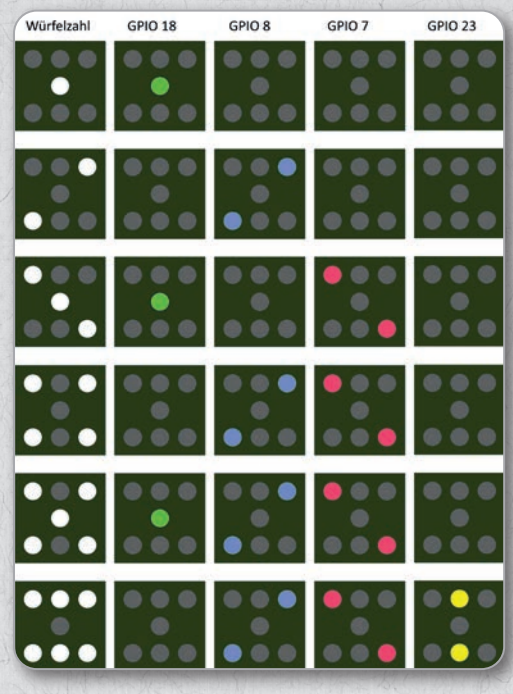


Abb. 6.1: *Die möglichen Würfelergebnisse für einen Würfel aus sieben LEDs.*

Bauen Sie für das folgende Experiment sieben LEDs sowie einen Taster wie abgebildet auf einer Steckplatine auf. Für die Ansteuerung der LEDs werden nur vier statt sieben GPIO-Pins benötigt, da ein Würfel zur Darstellung gerader Zahlen die Augen paarweise nutzt.

Benötigte Bauteile

- 1 x Steckplatine,
- 2 x LED rot,
- 2 x LED gelb,
- 2 x LED blau,
- 1 x LED grün,
- 1 x Taster,
- 7 x Verbindungskabel,
- 5 x Drahtbrücke (unterschiedliche Längen)

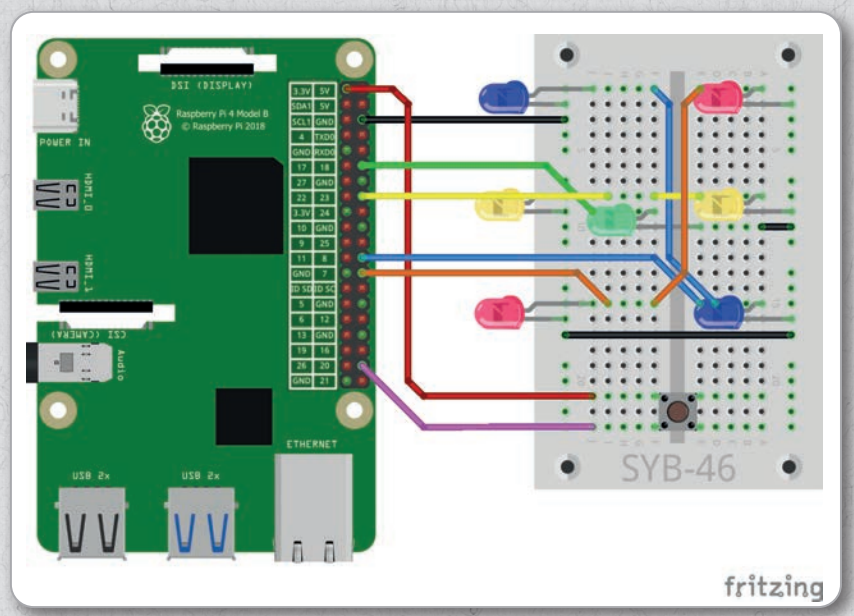
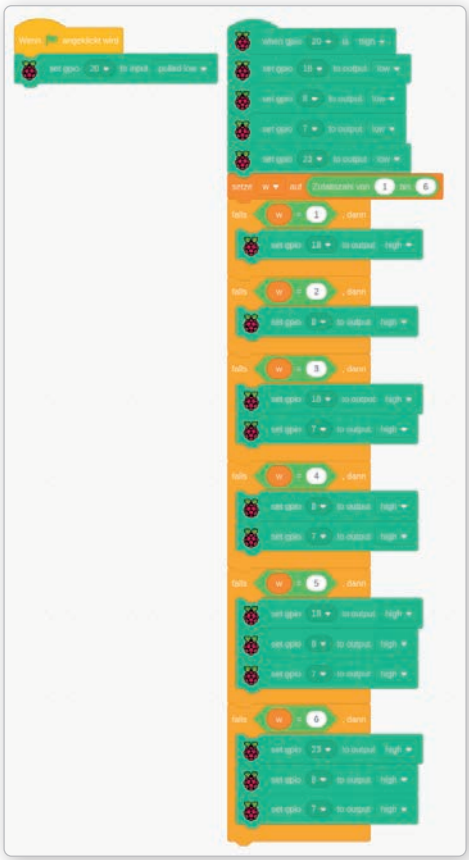


Abb. 6.2: *LED-Würfel mit Taste auf einer Steckplatine.*

6.1 | Würfeln mit Scratch



Das Scratch-Skript `06wuerfel` läuft insoweit ähnlich wie die Fußgängerampel, als ein Tastendruck verschiedene LEDs auslöst. Für das Erzeugen, Speichern und Abfragen der Zufallszahl werden neue Blöcke verwendet.

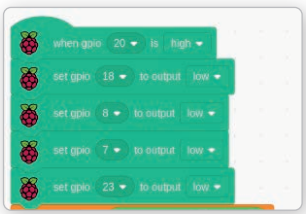
Abb. 6.3: *Das Scratch-Skript für den LED-Würfel.*

6.1.1 | So funktioniert es

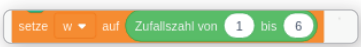
Wenn der Benutzer auf das grüne Fähnchen klickt, wird GPIO-Pin 20 als Eingang für den Taster mit internem Pull-down-Widerstand initialisiert.

Bei Verwendung der *Raspberry Pi GPIO*-Erweiterung ist nicht unbedingt eine Schleife nötig, die ständig darauf wartet, dass der Taster gedrückt wird.

Der Block *when gpio 20 is high* startet automatisch die darunterhängenden Programmblöcke, wenn dieser Pin den Wert *High* annimmt, der Taster also mit der +3,3-V-Leitung verbunden ist.



Danach werden als Erstes die vier GPIO-Ports 18, 8, 7 und 23 mit den LEDs ausgeschaltet. Am Anfang sind die LEDs alle aus, aber später bleibt ein angezeigtes Würfelergebnis so lange bestehen, bis der Benutzer wieder die Taste drückt.

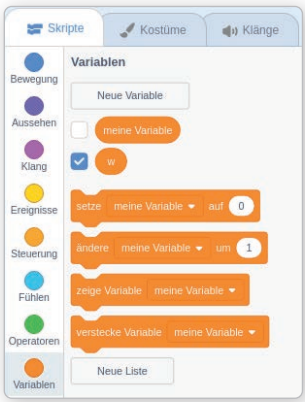


Anschließend wird eine zufällige Zahl zwischen Eins und Sechs erzeugt und in der Variablen `w` gespeichert.

Wie entstehen Zufallszahlen?

Gemeinhin denkt man, in einem Programm könne nichts zufällig geschehen. Wie kann ein Programm dann in der Lage sein, zufällige Zahlen zu generieren? Teilt man eine große Primzahl durch irgendeinen Wert, ergeben sich ab der x-ten Nachkommastelle Zahlen, die kaum noch vorhersehbar sind und sich auch ohne jede Regelmäßigkeit ändern, wenn man den Divisor regelmäßig erhöht. Dieses Ergebnis ist zwar scheinbar zufällig, lässt sich aber durch ein identisches Programm oder den mehrfachen Aufruf desselben Programms jederzeit reproduzieren. Nimmt man aber eine aus einigen dieser Ziffern zusammengebaute Zahl und teilt sie wiederum durch eine Zahl, die sich aus der aktuellen Uhrzeitsekunde oder dem Inhalt einer beliebigen Speicherstelle des Rechners ergibt, kommt ein Ergebnis heraus, das sich nicht reproduzieren lässt und daher als Zufallszahl bezeichnet werden kann.

Variablen müssen in Scratch erst einmal angelegt werden. Klicken Sie in der Blockpalette links auf das orangefarbene Symbol *Variablen* und dann auf *Neue Variable*. Geben Sie der Variablen einen Namen, in unserem Beispiel `w` (wie Würfel).



In der Blockpalette werden verschiedene Blöcke zur Arbeit mit Variablen angezeigt.

Aktivieren Sie den Schalter links neben der Variablen `w`, wird diese Variable automatisch auf der Bühne in einem kleinen orangefarbenen Feld angezeigt. So sehen Sie hier immer die gewürfelte Zahl und können leicht kontrollieren, ob die LEDs funktionieren.

Ziehen Sie jetzt den Block *setze ... auf* in das Skript. Wählen Sie im Listenfeld die Variable *w* aus.

Ziehen Sie aus der Blockpalette der Operatoren den Block *Zufallszahl von ... bis ...* auf das Zahlenfeld im orangefarbenen Block *setze ... auf*. Tragen Sie dann in die beiden Zahlenfelder eine `1` und eine `6` ein, da in diesem Bereich die Zufallszahl liegen soll.

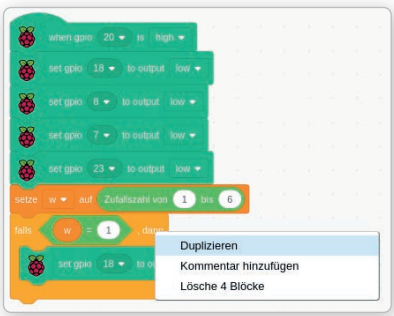
Nachdem die Zahl gewürfelt wurde, folgen sechs *falls*-Blöcke für jeden möglichen Würfelwert.

Jeder dieser Blöcke schaltet die entsprechenden LEDs ein, wenn eine bestimmte Zahl gewürfelt wurde.

Ziehen Sie den grünen Block = in das Abfragefeld des *falls*-Blocks. Und ziehen Sie dann den Block der Variablen *w* aus der Blockpalette *Variablen* in das erste Feld des Gleichheitsoperators. In das zweite Feld schreiben Sie

eine 1. Jetzt wird der Block innerhalb der Klammer immer dann abgearbeitet, wenn das Würfelergebnis eine 1 war. Innerhalb des *falls*-Blocks setzen Sie einen Block *set gpio … to outpout high* und geben dort die entsprechenden GPIO-Ports der LEDs für dieses Würfelergebnis an.

Mit einem Rechtsklick auf den *falls*-Block können Sie diesen duplizieren und brauchen nur noch die Würfelergebnisse und die passenden LEDs zu ändern.



Nachdem alle sechs möglichen Fälle geprüft wurden, zeigen die LEDs ein Würfelergebnis an. Es bleibt so lange bestehen, bis der Taster wieder gedrückt wird.

6.2 | Würfeln mit Python

Das Python-Programm `06ledwuerfel.py` stellt den gleichen Würfel in Python dar. Der Schaltungsaufbau und die verwendeten GPIO-Ports entsprechen denen aus dem Scratch-Skript.

6.2.1 | So funktioniert es

Das Programm enthält viele bekannte Elemente. Wir gehen hier nur auf die neuen ein.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
```

```
import RPi.GPIO as GPIO
import time, random
```

Bei der Initialisierung wird zusätzlich das Modul `random` importiert, das die Möglichkeit bietet, Zufallszahlen zu erzeugen.

```
GPIO.setmode(GPIO.BCM)
LED = [18,8,7,23]
TASTE = 20
```

Die Nummerierung der GPIO-Ports wird wie in den vorherigen Beispielen auf `BCM` gesetzt, und im Array `LED` werden die Nummern für die mittlere LED sowie die drei LED-Paare definiert. Der GPIO-Port des Tasters wird in der Konstanten `TASTE` definiert.

```
for x in range(4):
  GPIO.setup(LED[x], GPIO.OUT)
GPIO.setup(TASTE, GPIO.IN, GPIO.PUD_DOWN)
print ("Knopf drücken, um zu würfeln. Strg+C beendet das Programm")
```

Die GPIO-Ports der LEDs werden als Ausgänge definiert, der GPIO-Port des Tasters als Eingang mit Pull-down-Widerstand. Danach erscheint auf dem Bildschirm für den Benutzer ein kurzer Hinweis dazu, wie das Programm funktioniert.

```
try:
  while True :
    if GPIO.input(TASTE)==1:
```

Jetzt startet die Endlosschleife, die darauf wartet, dass der Benutzer den Taster drückt.

```
      for x in range(4):
        GPIO.output(LED[x], 0)
      time.sleep(0.5)
```

Hat er dies getan, werden alle LEDs ausgeschaltet, und es wird 0,5 Sekunden gewartet.

```
w = random.randrange(1,7)
print ("Gewürfelte Zahl:" + str(w))
```

Danach wird eine zufällige Zahl zwischen 1 und 6 erzeugt und auf dem Bildschirm angezeigt. Diese Anzeige kann auch weggelassen werden.

```
if w == 1:
  GPIO.output(LED[0], 1)
if w == 2:
  GPIO.output(LED[1], 1)
if w == 3:
  GPIO.output(LED[0], 1)
  GPIO.output(LED[2], 1)
if w == 4:
  GPIO.output(LED[1], 1)
  GPIO.output(LED[2], 1)
if w == 5:
  GPIO.output(LED[0], 1)
  GPIO.output(LED[1], 1)
  GPIO.output(LED[2], 1)
if w == 6:
  GPIO.output(LED[1], 1)
  GPIO.output(LED[2], 1)
  GPIO.output(LED[3], 1)
```

Abhängig von der gewürfelten Zahl, werden bestimmte LED-Gruppen eingeschaltet.

```
print ("Knopf drücken, um zu würfeln. Strg+C beendet das Programm")
```

Unabhängig vom Würfelergebnis wird der Benutzer erneut aufgefordert, die Taste zu drücken. Danach springt das Programm wieder zum Schleifenanfang. Die LEDs bleiben eingeschaltet, bis der Benutzer die Taste drückt.

7 Scratch-Katze mit GPIO-Tasten steuern

Die Katze, das Scratch-Symbol, kann auf der Bühne laufen und dabei Spuren hinterlassen. Die Katze symbolisiert dabei die verschiedenartigen Objekte, die in Scratch animiert werden können.

Das Scratch-Skript `07tasten` zeigt, wie man Scratch-Animationen über GPIO-Tasten steuern kann.

Bauen Sie für das folgende Experiment vier Taster so auf, dass sie die vier Richtungstasten links, oben, rechts und unten darstellen.

Benötigte Bauteile

- 1 x Steckplatine,
- 4 x Taster,
- 5 x Verbindungskabel,
- 4 x Drahtbrücke (unterschiedliche Längen)

Die vier Tasten sollen die Katze jeweils einen Schritt in die vier Richtungen bewegen. Dabei sollen die Bewegungen als Linien aufgezeichnet werden. Auf diese Weise kann man verfolgen, wie sich die Katze bewegt hat, oder auch gezielt bestimmte Grafiken zeichnen.

Um diese Linien zu zeichnen, installieren Sie zusätzlich in Scratch die Erweiterung *Malstift*.

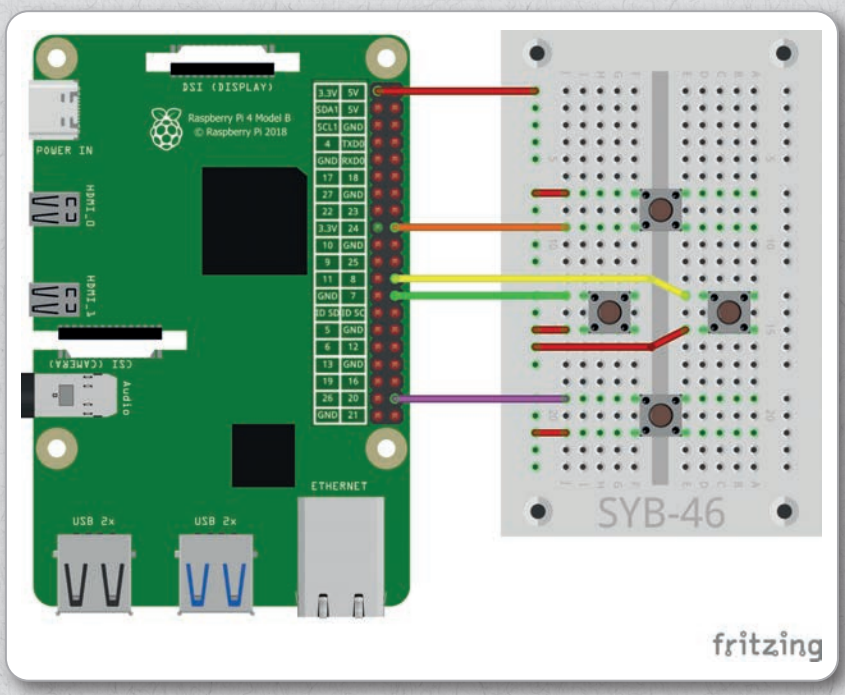


Abb. 7.1: *Vier Taster auf einer Steckplatine.*

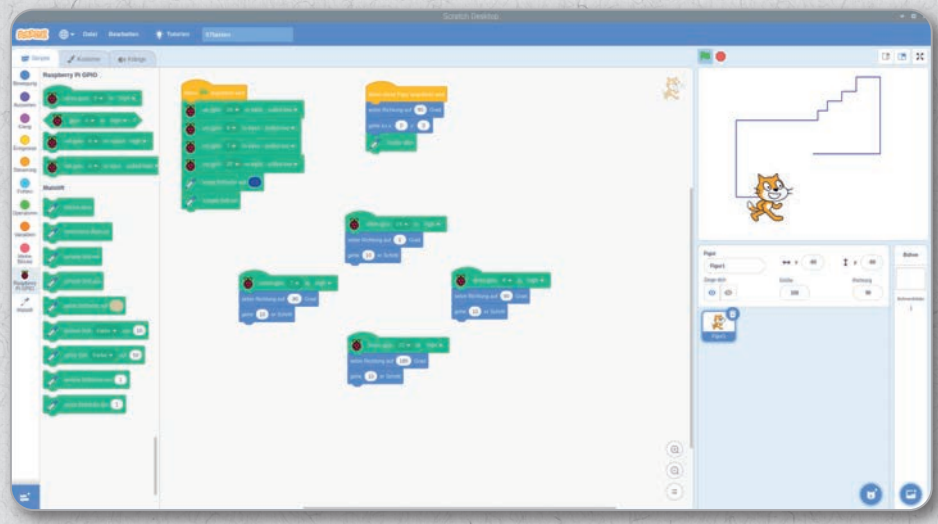
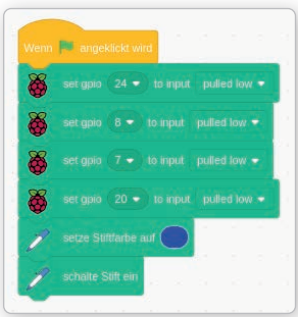


Abb. 7.2: *Die Katze bewegt sich gesteuert von GPIO-Tasten.*

7.1 | So funktioniert es

Vier einzelne Programmblöcke fragen die vier Tasten ab. Wurde eine davon gedrückt, dreht sich die Katze in die entsprechende Richtung und geht einen 10er-Schritt.

Wenn der Benutzer auf das grüne Fähnchen klickt, werden als Erstes die vier verwendeten GPO-Pins initialisiert. Wir verwenden hier die Ports 24, 8, 7 und 20 als Eingänge mit eingebauten Pull-down-Widerständen für die Taster.

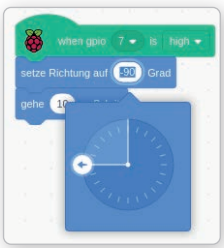


Nach der Initialisierung wird der Malstift auf Blau gesetzt und eingeschaltet, was bedeutet: Ab jetzt hinterlassen alle Bewegungen eine Spur auf der Bühne. Diese Blöcke finden Sie in der Erweiterung *Malstift*.

Jede Taste wird über einen *when gpio … is high*-Block aus der Erweiterung *Raspberry Pi GPIO* einzeln abgefragt.

In der Grundeinstellung läuft die Figur nach rechts, was in Scratch 90° bedeutet. Wurde die Taste an GPIO-Port 7 (nach links) gedrückt, dreht sich die Katze in die Richtung –90°, was im Scratch-Koordinatensystem nach links bedeutet. Die Winkel können im Block *setze Richtung auf … Grad* aus

der Blockpalette *Bewegung* interaktiv eingestellt werden. Hier werden absolute Richtungen angegeben, da die Ausrichtung der Katze vor dem Drücken der Taste nicht gespeichert wird. Nach der Drehung geht die Katze einen 10er-Schritt.



Auf die gleiche Weise werden die anderen drei Tasten abgefragt, die Katze wird entsprechend gedreht und bewegt.

Irgendwann ist der Bildschirm voller Linien und die Katze möglicherweise dabei, am Rand der Bühne zu verschwinden. Für diesen Fall legen Sie im Skriptbereich noch einen weiteren Skriptblock an, der mit dem Hauptskript nicht verbunden ist.

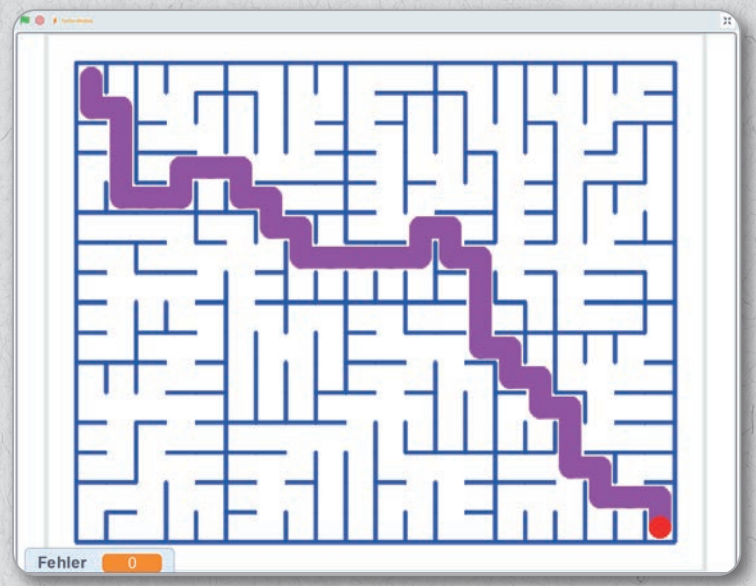
Dieses Skript wird nicht durch das grüne Fähnchen, sondern durch Klick auf die als *Figur1* bezeichnete Katze gestartet. Es stellt den Anfangszustand wieder her, indem es zunächst die Katze in die Standardrichtung 90° nach rechts dreht und sie auf die Koordinaten *x:0, y:0* im Zentrum der Bühne zurücksetzt. Zuletzt werden alle Malspuren gelöscht.

Dieses Skript kann jederzeit durch Klick auf die Katze ausgelöst werden. Die anderen Skriptblöcke brauchen nicht beendet zu werden und laufen weiter. Sie können also nach dem Zurücksetzen der Katze sofort wieder eine Taste drücken und die Katze weiterlaufen lassen.

8 Weg durch ein Labyrinth

Seit Jahrtausenden faszinieren Labyrinthe und Irrgärten unterschiedlichster Formen die Menschen. Besonderes Interesse löst diese Form der Grafik bei Künstlern, aber auch bei Mathematikern und Logikern aus. Im Computerzeitalter stellen sowohl das Gestalten als auch das Lösen solcher Labyrinthe immer wieder interessante Herausforderungen an Programmierer.

Dieses Scratch-Projekt liefert einen ersten Einblick in die Geometrie und Logik von Labyrinthen.



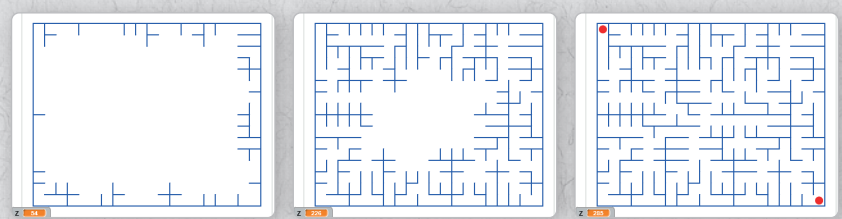
Das Scratch-Programm generiert im ersten Schritt ein zufälliges Labyrinth. Anschließend können Sie mit den vier Pfeiltasten der Tastatur den roten Punkt durch das Labyrinth bewegen, ohne an den Wänden anzustoßen.

Es gibt immer genau einen Weg durch das Labyrinth

Jedes Labyrinth sieht etwas anders aus, und durch jedes gibt es genau einen Weg von links oben nach rechts unten – probieren Sie es aus.

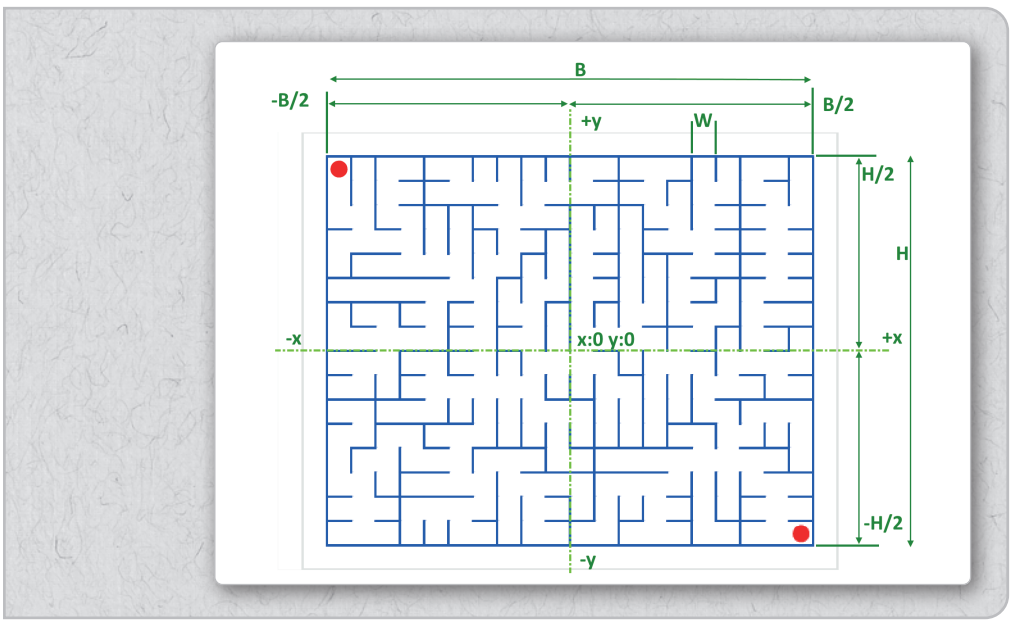
Dieser garantierte Weg basiert auf einem einfachen Prinzip. Beim Generieren des Labyrinths wird an den Rändern begonnen. Von einem zufälligen Punkt auf einer Wand aus wird ein neues Wandsegment in den freien Raum gezogen. Es darf nicht an einer bestehenden Wand enden. So wird nacheinander immer zufällig ein Wandpunkt (im Raster des Labyrinths) ausgewählt, und von dort werden Wände auf angrenzende freie Rasterpunkte gezogen – so lange, bis alle Rasterpunkte belegt sind. Dadurch, dass nie eine Wand zwischen zwei Wänden gezogen wird, kann es keine durchgehende Sperre geben, die einen Lösungsweg verhindert. Umgekehrt fängt keine Wand an einem freien Punkt im Raum an, sodass es auch keine Inseln geben kann, woraus sich zwei unterschiedliche Wege ergeben würden.

Dieses Prinzip kann beim Generieren eines Labyrinths genau mitverfolgt werden.



8.1 | Das Koordinatensystem des Labyrinths

Die Scratch-Bühne hat zwar eine genau definierte Größe, trotzdem ist dieses Programm durch die Verwendung von Variablen so allgemein gehalten, dass es theoretisch auf unterschiedlichen Bühnengrößen laufen könnte. Außerdem verliert man bei der Berechnung mit Variablen nicht so schnell die Übersicht wie bei absoluten Zahlen, bei denen man irgendwann nicht mehr weiß, wie sie sich zusammensetzen.



Zur Beschreibung bestimmter Punkte im Labyrinth werden die Variablen *B* (= Breite), *H* (= Höhe) und *W* (= Wegbreite) verwendet. Die Wegbreite legt das Raster des Labyrinths fest.

- Der Mittelpunkt des Labyrinths liegt bei den Koordinaten `x:0` und `y:0`.
- Das Labyrinth hat eine Breite von *B*, gezählt in Wegbreiten, nicht in Koordinateneinheiten. Die Breite in Koordinateneinheiten ergibt sich durch Multiplikation mit der Wegbreite *W*. `B=20` und `W=20`, daraus ergibt sich eine Gesamtbreite von `400` Koordinateneinheiten.

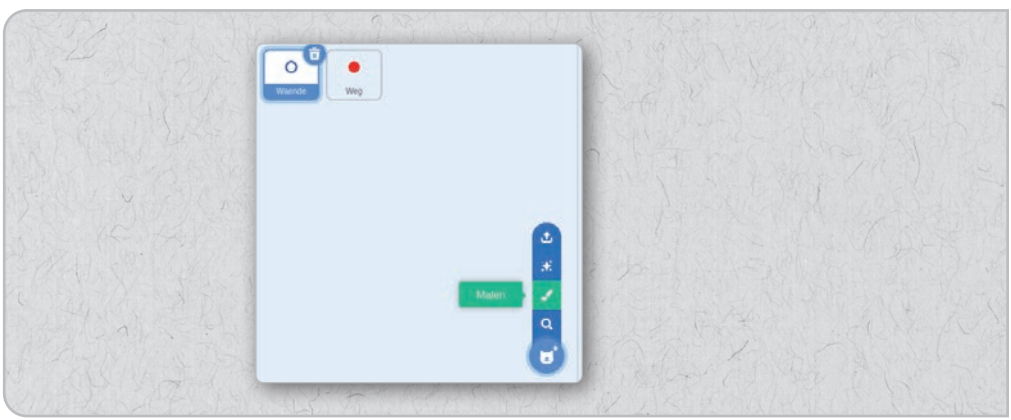
- Das Labyrinth hat eine Höhe von *H*, ebenfalls gezählt in Wegbreiten. Die Höhe in Koordinateneinheiten ergibt sich wieder durch Multiplikation mit der Wegbreite *W*. `H=16` und `W=20`, daraus ergibt sich eine Gesamthöhe von `320` Koordinateneinheiten.

8.2 | Das Labyrinth zeichnen

Das Programm `08labyrinth1` verwendet zwei einfache Figuren, eine zeichnet die Wände und ist selbst nicht sichtbar, mit der anderen bewegt man sich später durch das Labyrinth und zeichnet dabei den Weg.

Legen Sie in Scratch ein neues Projekt an und löschen Sie die Katze. Installieren Sie die Erweiterung *Malstift*, mit der die Linien des Labyrinths gezeichnet werden.

Im Programm verwenden wir zwei einfache Objekte: Ein blauer hohler Kreis (*Waende*) zeichnet die Wände und ist selbst nicht sichtbar, mit dem anderen Objekt, dem roten gefüllten Kreis (*Weg*), bewegt man sich später durch das Labyrinth und zeichnet dabei den Weg.



Klicken Sie in der Figurenliste unten rechts auf das Symbol *Malen*. Scratch öffnet ein eigenes Grafikmodul, mit dem sich neue Figuren erstellen lassen. Verwenden Sie den Vektormodus für die beiden Figuren.

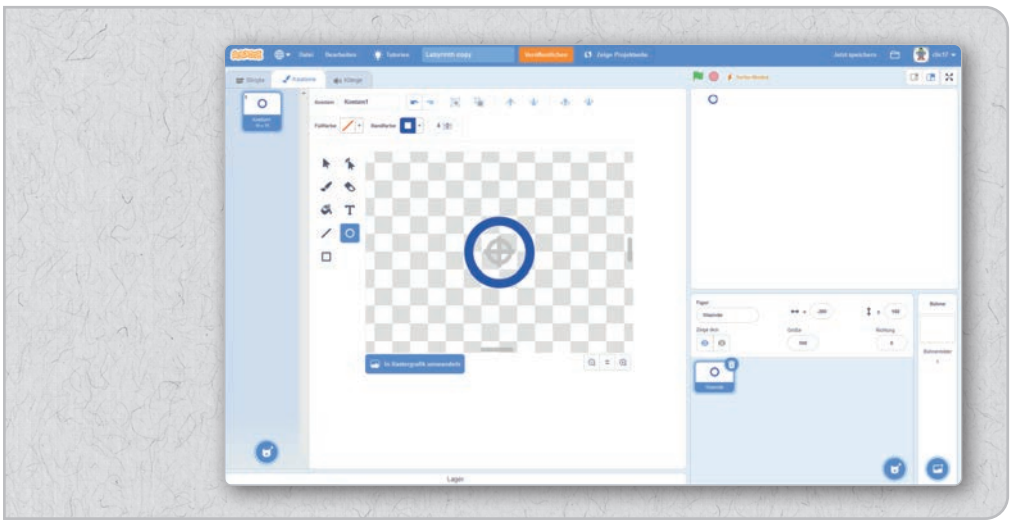
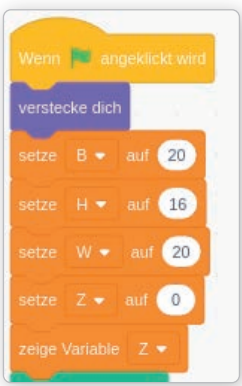


Abb. 8.1: *Das Vektorgrafikmodul in Scratch.*

Beim Klick auf das grüne Fähnchen wird die Figur *Waende* versteckt, da sie beim Zeichnen nur stört und deshalb besser unsichtbar genutzt wird. Dann werden vier Variablen festgelegt:

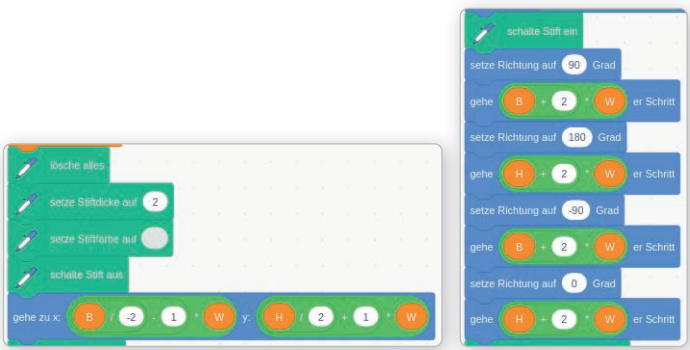


- *B* bezeichnet die Breite des Labyrinths, gezählt in Wegbreiten, nicht in Koordinateneinheiten.
- *H* bezeichnet die Höhe des Labyrinths, ebenfalls gezählt in Wegbreiten.
- Die Wegbreite *W* legt das Raster des Labyrinths fest.
- Der Zähler *Z* zählt mit, wie viele Rasterpunkte bereits an das Labyrinth angeschlossen sind, und stellt so fest, wann es fertig gezeichnet ist.

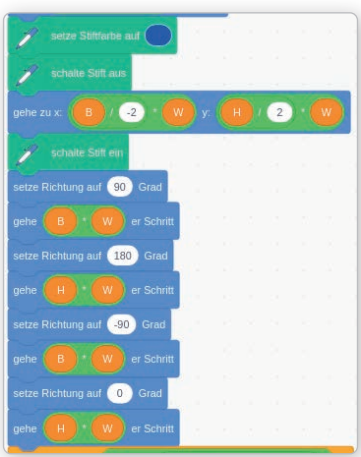
Jede Scratch-Figur kann bei ihrer Bewegung Linien zeichnen, wenn der Malstift aus der gleichnamigen Erweiterung eingeschaltet ist. Solange er ausgeschaltet ist, hinterlassen Objekte keine Malspuren.

Als Erstes zeichnen wir ein hellgraues Rechteck mit einem Abstand von einer Wegbreite um das eigentliche Labyrinth herum. Dies wird später verhindern, dass beim Erzeugen des Labyrinths Mauersegmente außen am Labyrinth angebaut werden. Dazu werden zuerst alle Malspuren früherer Programmläufe gelöscht. Danach wird die Stiftfarbe auf Hellgrau gesetzt, und der Malstift wird zunächst ausgeschaltet, um die Figur an die linke obere Ecke dieses Rechtecks zu bringen.

Anschließend werden die vier Seiten des Rechtecks gezeichnet.

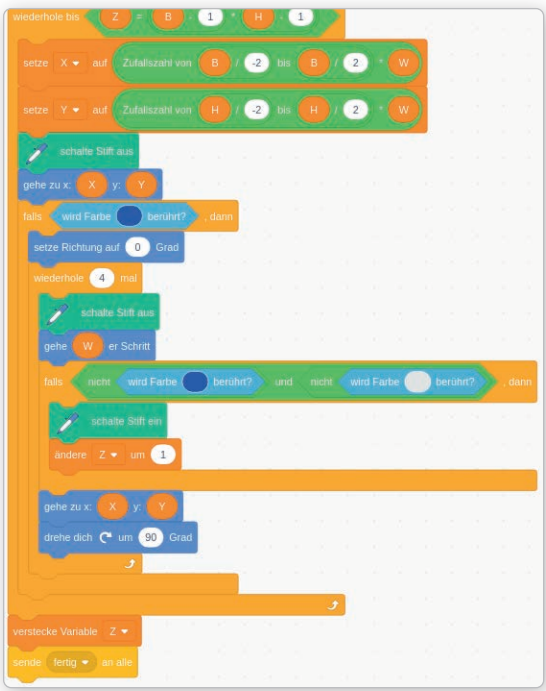


Als Nächstes zeichnen wir in Blau den Umriss des Labyrinths. Dies funktioniert ganz ähnlich, da es auch ein Rechteck ist, dessen Mittelpunkt im Nullpunkt des Koordinatensystems liegt.



Jetzt kommt der eigentlich interessante Programmteil, der das Labyrinth aufbaut. Da wir, bedingt durch den Zufallsgenerator, vorher nicht wissen, wie lange es dauert, das Labyrinth aufzubauen, verwenden wir eine *wiederhole bis* ...-Schleife, die so lange läuft, bis alle inneren Rasterpunkte des Rechtecks an das Netz der Labyrinthmauern angeschlossen sind. Es gibt `B-1 * H-1` Rasterpunkte innerhalb des Rechtecks, die die Variable *Z* zählt.

In jedem Schleifendurchlauf wird ein Punkt zufällig ermittelt, von dem aus neue Labyrinthmauern gebaut werden. Voraussetzung ist natürlich, dass der Punkt bereits an einer Mauer liegt, da Mauern nach den Baureglen nicht frei im Raum beginnen können. Die Koordinaten dieses zufällig ermittelten Punkts werden in den Variablen *X* und *Y* gespeichert.



Liegt der zufällig ermittelte Punkt auf einer Mauer, können von dort neue Mauern gezeichnet werden. Um das zu überprüfen, verwenden wir den Block *wird Farbe ... berührt*. Wenn das unsichtbare Objekt, das die Mau-

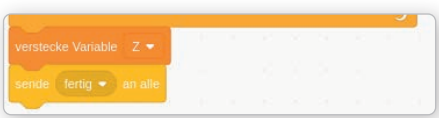
ern zeichnet, die blaue Farbe berührt, mit der die Mauern gezeichnet werden, steht es auf einem Rasterpunkt mit einer Mauer. Das Objekt ist klein genug, um keine Nachbarmauer auf dem nächsten Rasterpunkt zu berühren. Es kann auch bei der hier verwendeten Programmlogik nie zwischen zwei Rasterpunkten stehen.

Trifft die Bedingung zu, versucht das Programm, innerhalb der Schleife möglichst viele Mauern von dem ermittelten Punkt aus in die vier Richtungen zu zeichnen. Es können aber höchstens drei Mauern neu gezeichnet werden, da ein Punkt nur als Anfangspunkt verwendet wird, wenn er bereits an einer Mauer liegt. Die vier möglichen Richtungen werden mit einer weiteren Schleife abgearbeitet.

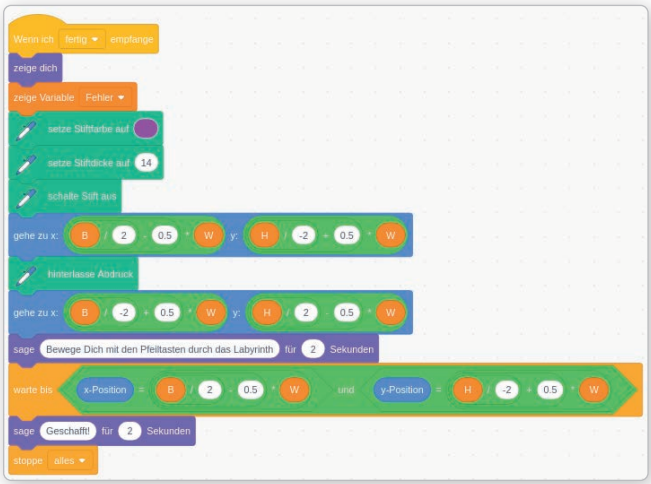
8.3 | Durch das Labyrinth laufen

Um durch das Labyrinth zu laufen, verwenden wir ein zweites Objekt *Weg*, das mit den vier Pfeiltasten der Tastatur in den vier Richtungen gesteuert wird. Beim Klick auf das grüne Fähnchen wird dieses Objekt zunächst versteckt, bis das Labyrinth fertig ist.

Das Objekt soll erst auftauchen, wenn das Labyrinth fertig gezeichnet ist. Dazu brauchen wir Blöcke, die es einem Objekt möglich machen, an ein anderes eine Nachricht zu senden. Bisher galten alle Programmblöcke immer nur für ein Objekt. Das Programm des Objekts *Waende* enthält dazu am Ende einen Block *sende fertig an alle*. Darauf können andere Figuren reagieren.



Das Hauptprogramm zum Bewegen des Objekts *Weg* durch das Labyrinth startet, wenn die rote Figur *Weg* von der blauen Figur *Waende* die Nachricht `fertig` empfängt:



Am Anfang zeigt sich die rote Figur *Weg* wieder. Auch die Variable *Fehler* wird jetzt angezeigt, damit man jederzeit sehen kann, wie oft man schon gegen eine Mauer gelaufen ist.

Danach setzt das Programm Stiftfarbe und -dicke, hinterlässt in der unteren rechten Ecke einen Abdruck der Figur und startet dann in der linken oberen Ecke des Labyrinths. Dabei wird mit einem *sage ... für ... Sek*-Block eine kurze Erklärung in einer Sprechblase angezeigt.

Ab jetzt braucht das Hauptprogramm nur noch abzuwarten, bis die rote Figur am Ziel angekommen ist. Die Tastatursteuerung und die Bewegung der Figur werden über eigenständige Programmblöcke geregelt.

Die beiden Bedingungen lauten:

- Die *x-Position* der Figur muss in der letzten Spalte des Labyrinths ganz rechts außen sein.

- Die *y-Position* der Figur muss in der untersten Zeile des Labyrinths sein.

Wenn beide Bedingungen gleichzeitig erfüllt sind, befindet sich die Figur in der rechten unteren Ecke des Labyrinths. Die *x-Position*, *y-Position* und auch die *Richtung* können wie Variablen zum Rechnen verwendet werden.

Wenn die rote Figur am Ziel angekommen ist, liefert ein *sage ... für ... Sek*-Block eine Erfolgsmeldung, danach werden alle Programmaktivitäten beendet.

Das Programm ist aber noch nicht fertig. Jetzt müssen die Pfeiltasten abgefragt, und danach muss die Figur entsprechend bewegt werden. Hierfür verwenden wir *Wenn Taste ... gedrückt*-Blöcke von der Blockpalette *Ereignisse*. Dieser Block bietet eine lange Auswahlliste an Tasten, die Aktionen auslösen können.

Die Aktionen, die durch die Pfeiltasten ausgelöst werden, unterscheiden sich nur in der Richtung, sind sonst aber gleich. Deshalb bauen wir dazu einen Programmblock, der von allen vier Pfeiltasten aufgerufen wird. Am Ende jeder Tastenabfrage findet sich wieder ein *sende ... an alle*-Block, der diesmal den Befehl `gehe` sendet.

Als Erstes geht die Figur einen halben Schritt weit in die durch den Taster festgelegte Richtung. Würde die Figur einen ganzen Schritt im Labyrinthraster gehen, würde sie auf jeden Fall auf dem benachbarten Wege-

feld landen. Durch den Trick mit dem halben Schritt gibt es zwei Möglichkeiten:

- Die Figur kommt mitten in einer Wand zum Stehen.
- Die Figur kommt auf einem Weg zwischen zwei Rasterfeldern zum Stehen.

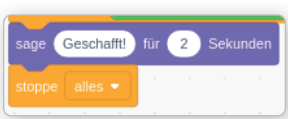
Diese beiden möglichen Ergebnisse haben natürlich völlig unterschiedliche Folgen.

Berührt die Figur nach dem halben Schritt die blaue Farbe einer Mauer, ist sie auf dem falschen Weg. Sie dreht sich um 180° und geht den halben Schritt wieder zurück an ihren ursprünglichen Platz.

Im anderen Fall – wenn die Figur keine blaue Farbe berührt, also auf dem Weg zwischen zwei Rasterfeldern steht – soll sie auf das nächste Rasterfeld laufen und dabei eine Spur hinterlassen. Da sie auf dem ersten halben Schritt keine Spur hinterlassen hat, geht sie den halben Schritt wieder zurück, wie es auch beim Berühren der Mauer erfolgt, und schaltet dann, am ursprünglichen Platz angekommen, den Malstift ein. Anschließend dreht sich die Figur erneut um 180° und bewegt sich mit eingeschaltetem Malstift einen ganzen Schritt auf das nächste Rasterfeld.

Die Abfrage der vier Taster wird so lange wiederholt, bis die rote Figur die untere rechte Ecke des Labyrinths erreicht hat und an dieser Stelle den roten Abdruck berührt. Diese Bedingung steht in der *wiederhole bis ...*-Schleife.

Zum Schluss liefert ein *sage ... für ... Sek*-Block eine Erfolgsmeldung, danach werden alle Programmaktivitäten beendet.

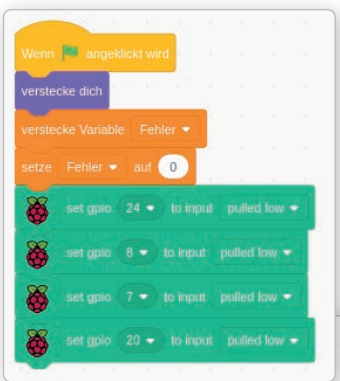


8.4 | Mit eigenen Tasten durch das Labyrinth laufen

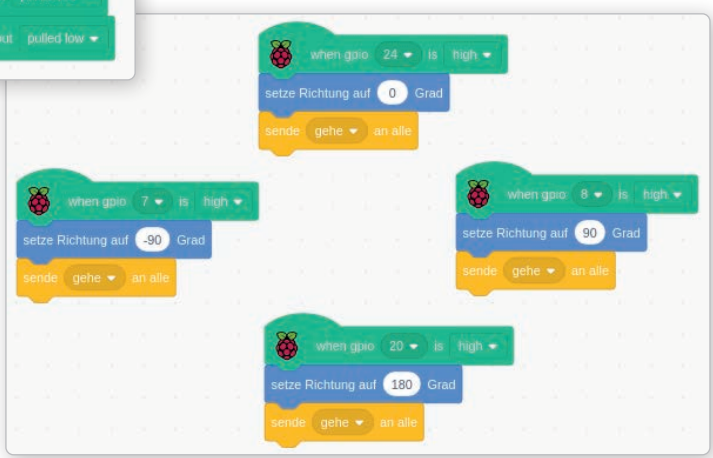
Die zweite Variante des Programms, `08labyrinth2`, baut auf die gleiche Weise ein Labyrinth. Nach Fertigstellung bewegt sich die rote Figur aber nicht mit den Pfeiltasten der Tastatur, sondern über vier Taster, die an GPIO-Pins angeschlossen sind. Der Schaltungsaufbau entspricht dem letzten Experiment.

Diese Programmvariante benötigt die Erweiterung *Raspberry Pi GPIO*. Dank dieser Erweiterung sind nur geringfügige Änderungen notwendig.

Beim Klick auf das grüne Fähnchen werden zusätzlich die vier verwendeten GPIO-Pins als Eingänge mit eingebauten Pull-down-Widerständen initialisiert.



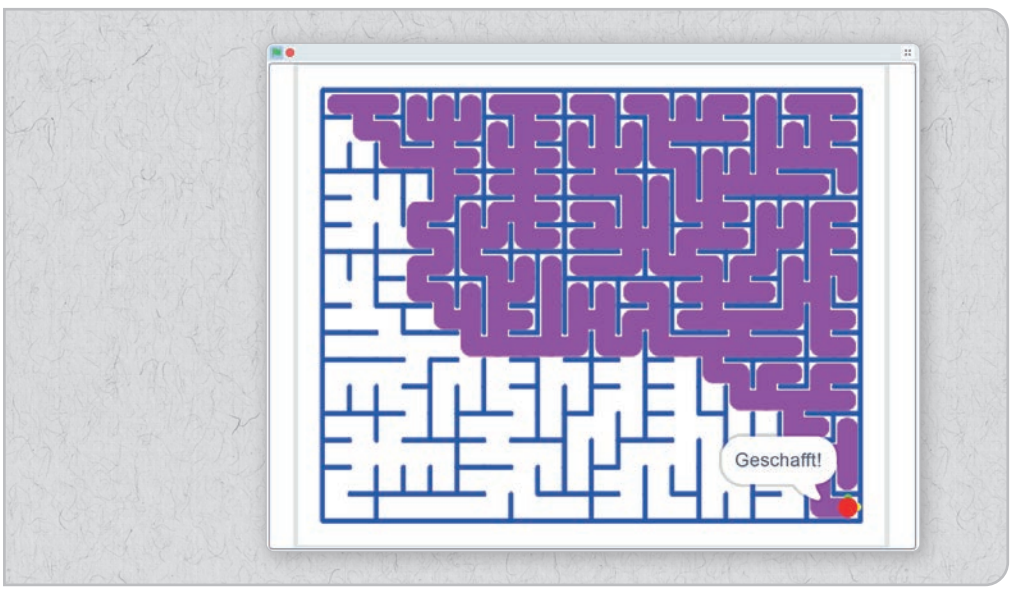
Die vier Blöcke, die bisher die Pfeiltasten der Tastatur abgefragt haben, verwenden jetzt Blöcke vom Typ *when gpio … is high* aus der Erweiterung *Raspberry Pi GPIO*.



8.5 | Automatisch den Weg durch das Labyrinth finden

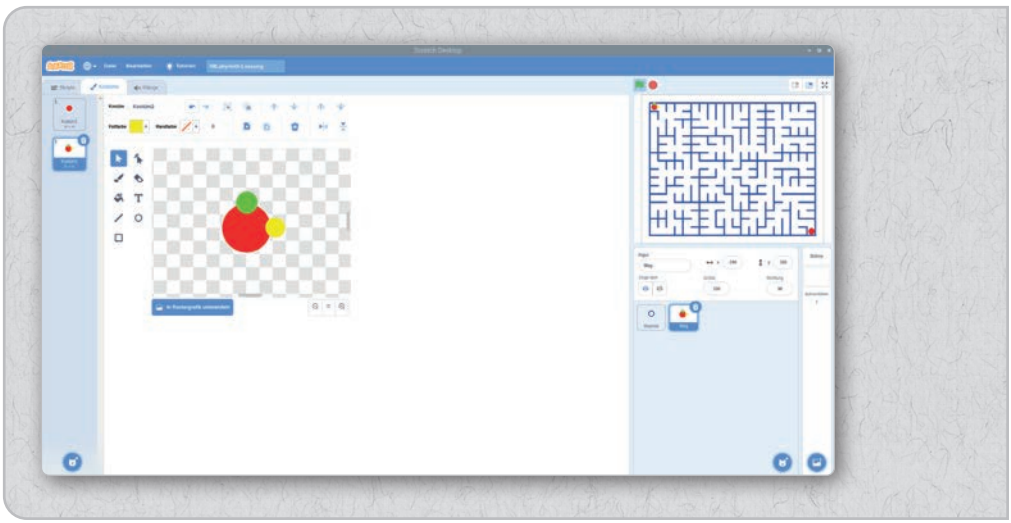
Den Weg durch ein Labyrinth zu finden, macht Spaß. Deutlich interessanter ist es natürlich, das Programm den Weg finden zu lassen.

Für jedes Labyrinth, das keine Inseln, sondern einen eindeutigen Weg hat, gibt es eine sichere Lösung: »Gehe immer mit der linken Hand an der Wand entlang, dann bleibt die rechte frei für eine Taschenlampe.« Nach diesem Prinzip wird das Scratch-Programm `08labyrinth_Loesung` einen Weg durch das Labyrinth finden.



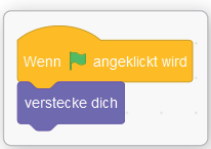
Die rote Figur *Weg* bekommt zur Suche nach dem Weg zwei Sensoren: ein gelbes »Auge«, das noch vorne blickt, und eine grüne »Hand«, die sich an der Wand entlangtastet.

Duplizieren Sie dazu ein zweites Kostüm auf der Registerkarte *Kostüme* und fügen Sie ihm, wie in der Abbildung zu sehen, zwei farbige Kreise hinzu. Dabei ist es wichtig, dass die beiden farbigen Flächen weit genug aus dem roten Quadrat herausragen, um benachbarte Wände zu berühren.

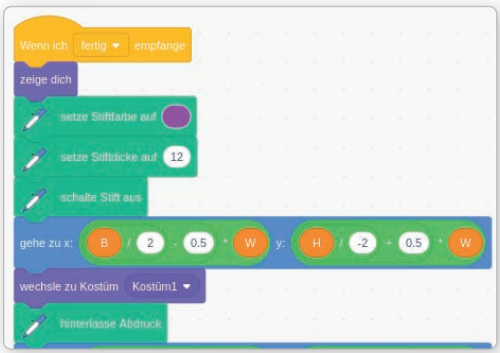


Das Programm der blauen Figur *Waende* zum Zeichnen des Labyrinths bleibt unverändert. Die rote Figur *Weg* bekommt ein komplett neues Programm, um den Weg zu finden. Die Variable *Fehler* wird nicht mehr benötigt, das Programm macht keine Fehler.

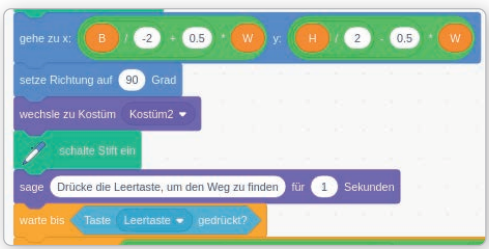
Wenn das grüne Fähnchen angeklickt wird, wird die Figur *Weg* versteckt.



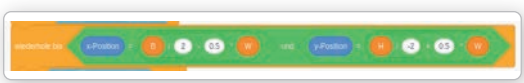
Sie tritt erst wieder in Aktion, wenn das Labyrinth fertig gezeichnet ist. Hier werden ein paar Grundeinstellungen getroffen. Die Figur zeigt sich. Der Malstift wird auf Lila und etwas breiter gesetzt und erst einmal ausgeschaltet. Die Figur richtet sich in die Startrichtung 90° aus, bewegt sich in die untere rechte Ecke und hinterlässt dort einen Abdruck. Dazu wird das *Kostüm1* eingeschaltet, damit dieser Abdruck nur als roter Kreis erscheint.



Nachdem die Figur an den Start oben links bewegt wurde, wird das *Kostüm2* gewählt, bei dem die beiden Sensoren sichtbar sind. Außerdem wird der Malstift eingeschaltet, damit die Figur auf ihrem Weg eine Spur hinterlässt. Jetzt ist die rote Figur bereit, auf den Weg geschickt zu werden. Ein Block *sage* ... teilt dem Benutzer mit, dass er die Leertaste drücken muss, um zu starten. Anschließend wartet das Programm darauf, dass der Benutzer die Leertaste auch wirklich drückt.



Danach startet eine neue *wiederhole bis* ...-Schleife, die die Bewegungsschritte der roten Figur so lange wiederholt, bis diese in der rechten unteren Ecke des Labyrinths angekommen ist.



Bevor wir das Programm bauen, das den Weg findet, hier zunächst die Logik, nach der die Suche im Labyrinth funktioniert.

So sucht sich das Programm seinen Weg

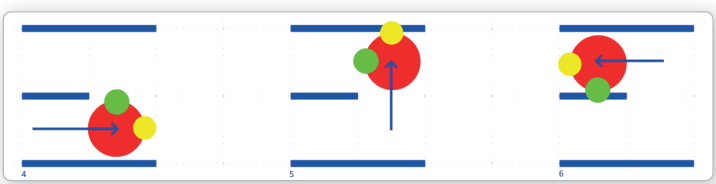
Wenn sich die Figur durch das Labyrinth bewegt, kann es zwei Fälle geben:

1. Die Figur steht auf einem Feld, auf dem der grüne Sensor links eine Wand berührt.

2. Hier muss das Programm noch unterscheiden, ob die Figur vor einem freien Feld steht (1) und geradeaus weitergehen kann oder ob sie vor einer Wand steht (2) und nach rechts abbiegen muss – links ist ja die Wand.

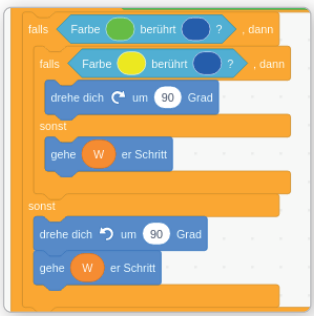
3. Die Figur ist einen Schritt auf ein Feld gegangen, auf dem der grüne Sensor links keine Wand mehr berührt. In diesem Fall – wenn links keine Wand ist – muss sie immer links abbiegen (3).

4. Dies gilt auch für den Fall, dass die Figur nicht an einer Ecke, sondern neben einem Wandende (4) steht.



5. In diesem Fall kann sie ebenfalls links abbiegen und einen Schritt weitergehen. Danach steht sie vor der gleichen Situation: Links ist keine Wand (5). Also soll sich die Figur erneut nach links drehen und dann noch mal einen Schritt weitergehen. Jetzt steht sie mit dem grünen Sensor wieder an einer Wand (6).

Diese Bewegungsregeln werden in zwei ineinandergeschachtelten *falls ... dann ... sonst*-Abfragen verarbeitet:



Falls der grüne Sensor eine blaue Wand berührt, wird geprüft, ob auch der gelbe Sensor eine blaue Wand berührt.

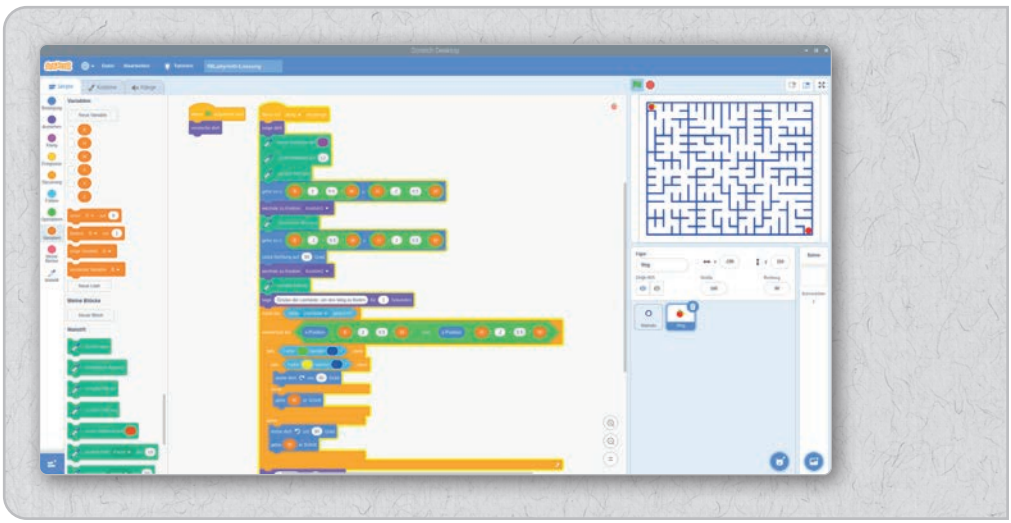
Ist das der Fall, steht die Figur in einer Ecke, in der es nur nach rechts weitergeht, sie muss sich also um 90° nach rechts drehen.

Berührt nur der grüne Sensor eine Wand, der gelbe aber nicht, geht es geradeaus weiter. Die Figur macht einen Schritt, also eine Rastereinheit *W* weit.

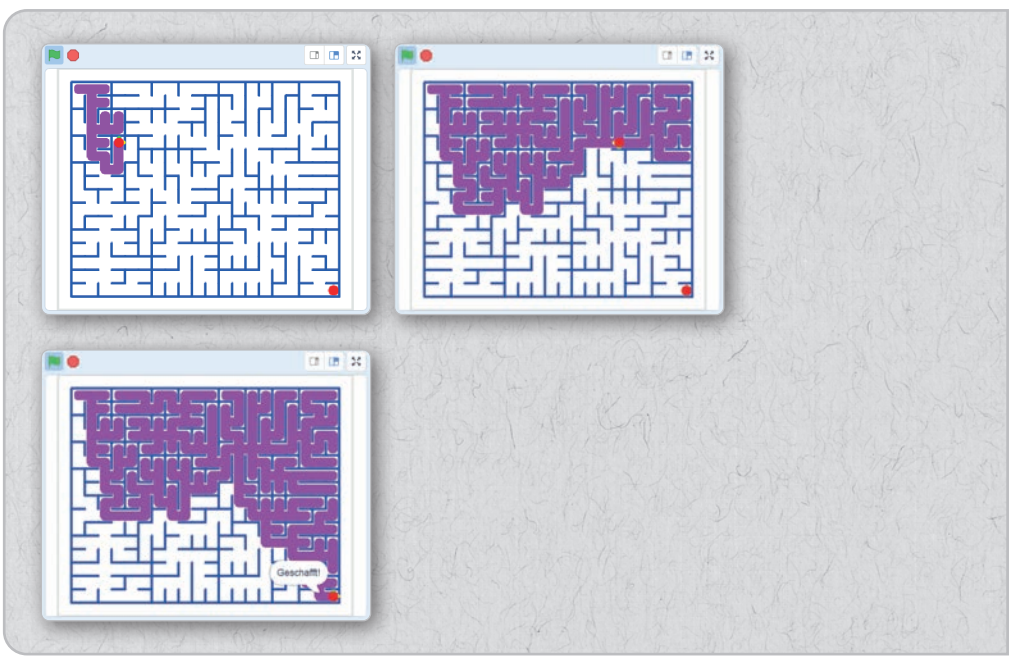
Berührt der grüne Sensor keine Wand, muss die Figur nach links abbiegen und einen Schritt gehen, um wieder eine Wand zu finden.

Diese Bewegungsmuster werden so lange wiederholt, bis die Figur den Zielpunkt in der rechten unteren Ecke erreicht hat. Zum Schluss wird noch kurz eine Meldung angezeigt, und das Programm ist zu Ende.

Starten Sie das Programm mit einem Klick auf das grüne Fähnchen. Warten Sie, bis das Labyrinth fertig gezeichnet ist und die Aufforderung zum Loslaufen erscheint. Drücken Sie dann einmal auf die Leertaste, und die rote Figur wird loslaufen.



Jetzt können Sie mitverfolgen, wie sich die Figur immer an der linken Wand entlanghangelt und so etwa die Hälfte des Labyrinths durchwandert, bis sie am Ziel ankommt.



9 ERSTES EXPERIMENT MIT DEM LC-DISPLAY

Die nächsten Experimente zeigen, wie Sie mit Python Texte auf dem LC-Display darstellen können. Zuerst müssen Sie den beiliegenden Pfostenverbinder wie auf Seite 25 beschrieben am Display anlöten.

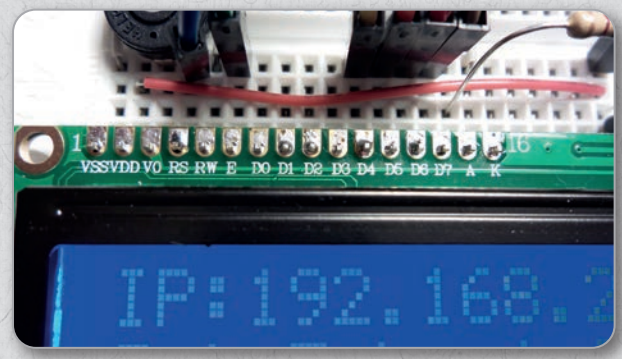


Abb. 9.1: *Die 16 Anschlusspins des LC-Displays.*

9.1 | Pinbelegung eines HD44780-kompatiblen Displays

PIN	FUNKTION	BESCHREIBUNG
1	VSS	Stromversorgung Masseleitung 0 V
2	VDD	Stromversorgung +5 V
3	V0	Kontrasteinstellung, 0 V ... 5 V
4	RS	Register Select

PIN	FUNKTION	BESCHREIBUNG
5	RW	Read/Write, wenn vom Display nichts ausgelesen wird, mit 0 V verbinden
6	E	Enable (Umschaltsignal)
7	D0	Datenbit 0 (im 4-Bit-Modus nicht benötigt)
8	D1	Datenbit 1 (im 4-Bit-Modus nicht benötigt)
9	D2	Datenbit 2 (im 4-Bit-Modus nicht benötigt)
10	D3	Datenbit 3 (im 4-Bit-Modus nicht benötigt)
11	D4	Datenbit 4
12	D5	Datenbit 5
13	D6	Datenbit 6
14	D7	Datenbit 7
15	A	Hintergrundbeleuchtung, 560-Ohm-Vorwiderstand erforderlich
16	K	Hintergrundbeleuchtung Masseleitung

Benötigte Bauteile

- 1 x Steckplatine,
- 1 x LC-Display,
- 1 x 560-Ohm-Widerstand (Grün-Blau-Braun),
- 1 x Potenziometer,
- 8 x Verbindungskabel,
- 7 x Drahtbrücke (unterschiedliche Längen)

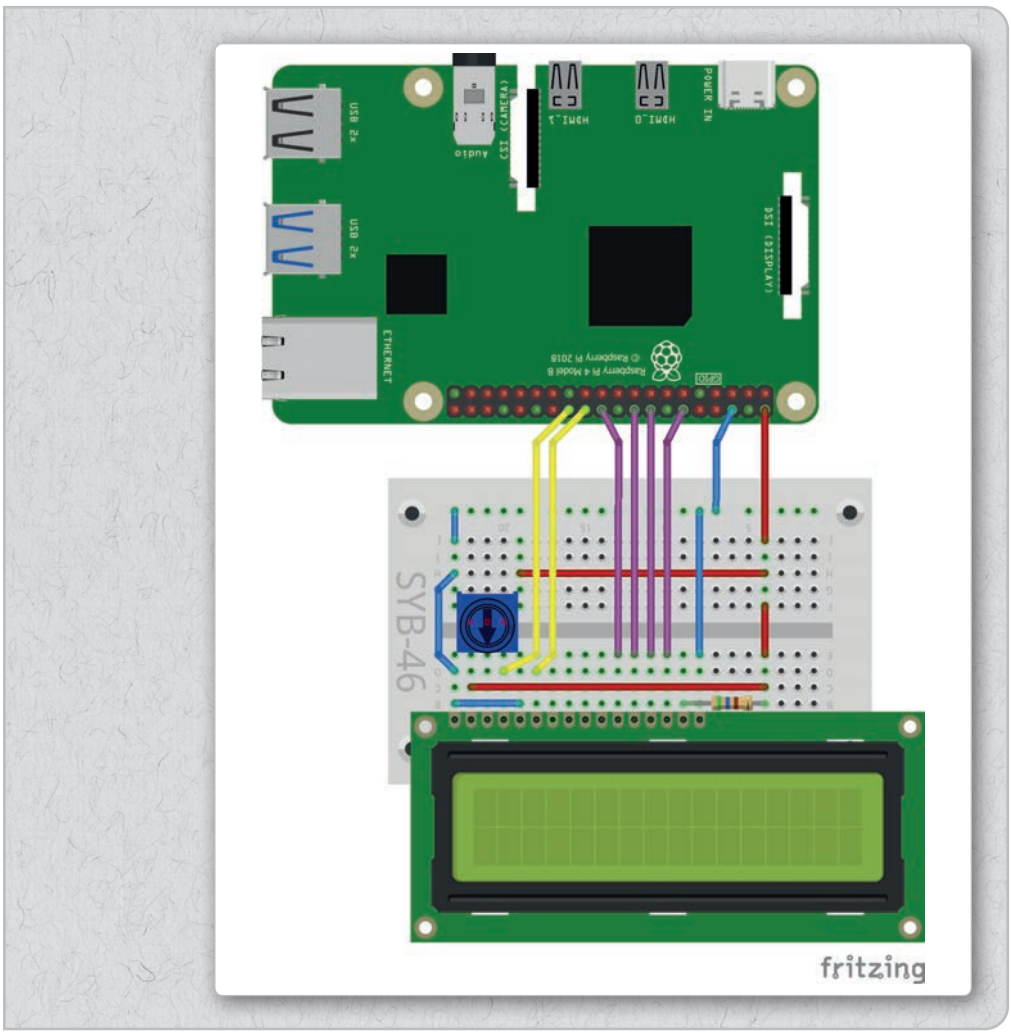


Abb. 9.2: *Gelb: Steuerleitungen, Violett: Datenleitungen, Rot: +5 V, Blau: Masse.*

Zusätzlich zu den Verbindungskabeln zum Raspberry Pi benötigen Sie Drahtbrücken unterschiedlicher Länge, mit denen mehrere Pins für die gemeinsame Stromversorgung und die Masseleitung zusammengeführt werden. Schneiden Sie sie vom mitgelieferten Schaltdraht ab und entfernen Sie die Isolierung an beiden Enden auf einer Länge von etwa 5 mm.

DISPLAYPIN	SIGNAL	RASPBERRY-PI-GPIO
1	VSS	0 V
2	VDD	+5 V
3	V0	Potenziometer
4	RS	GPIO 7
5	RW	0 V
6	E	GPIO 8
7	D0	im 4-Bit-Modus nicht benötigt
8	D1	im 4-Bit-Modus nicht benötigt
9	D2	im 4-Bit-Modus nicht benötigt
10	D3	im 4-Bit-Modus nicht benötigt
11	D4	GPIO 25
12	D5	GPIO 24
13	D6	GPIO 23
14	D7	GPIO 18
15	A	Vorwiderstand 560 Ohm
16	K	0 V

Der LED der Hintergrundbeleuchtung – Pin 15 – wird ein 560-Ohm-Widerstand als Schutz gegen Überlastung vorgeschaltet. Zur Kontrastregelung bauen Sie ein 15-kOhm-Potenziometer ein, das dem Pin 3 eine Spannung zwischen +5 V und 0 V zuführt.

Schaltung genau überprüfen

Bevor Sie den Raspberry Pi und damit die Schaltung mit Strom versorgen, prüfen Sie alle Anschlussdrähte noch einmal genau. Eine falsch angeschlossene +5-V-Leitung kann das Display beschädigen. Die maximale Spannung am Kontrastpin darf nur weniger als +5 V betragen. Drehen Sie also das Potenziometer nie ganz auf.

Das Display unterstützt zwei Modi zur Übertragung der Daten. Im 8-Bit-Modus kann ein komplettes Zeichen auf einmal übertragen werden. Meistens wird aber der 4-Bit-Modus verwendet, da dieser vier Ports am sendenden Gerät spart. Hier werden die Daten eines Zeichens in zwei Blöcken hintereinander übertragen. Pi_Scratch verwendet generell den 4-Bit-Modus.

9.2 | LC-Display mit Python ansteuern

In Python wird das Display sehr hardwarenah direkt über die GPIO-Ports und entsprechende High- oder Low-Signale angesteuert, was anfangs etwas umständlich wirkt. Dafür lernen Sie, die Funktionsweise von LC-Displays genauer zu verstehen. Das Display wird bei diesem Projekt im 4-Bit-Modus betrieben.

Abb. 9.3: *Einfache Textdarstellung auf dem LC-Display.*

Das erste einfache Python-Beispiel `09display01.py` zeigt zwei statische Zeichenfolgen auf dem Display. Das hier beschriebene Grundprogramm dient auch als Basis für die weiteren Programmbeispiele mit dem LC-Display.

```
#!/usr/bin/python
import RPi.GPIO as GPIO
import time

LCD_RS = 7
LCD_E  = 8
LCD_D4 = 25
LCD_D5 = 24
LCD_D6 = 23
LCD_D7 = 18

GPIO.setmode(GPIO.BCM)
GPIO.setup(LCD_E,  GPIO.OUT)
GPIO.setup(LCD_RS, GPIO.OUT)
GPIO.setup(LCD_D4, GPIO.OUT)
GPIO.setup(LCD_D5, GPIO.OUT)
GPIO.setup(LCD_D6, GPIO.OUT)
GPIO.setup(LCD_D7, GPIO.OUT)

LCD_WIDTH = 16
LCD_LINE_1 = 0x80
LCD_LINE_2 = 0xC0
LCD_CHR = 1
LCD_CMD = 0
E_PULSE = 0.00005
E_DELAY = 0.00005
INIT  = 0.01

def lcd_enable():
  time.sleep(E_DELAY)
  GPIO.output(LCD_E, 1)
  time.sleep(E_PULSE)
  GPIO.output(LCD_E, 0)
  time.sleep(E_DELAY)

def lcd_byte(bits, mode):
  GPIO.output(LCD_RS, mode)
  GPIO.output(LCD_D4, bits&0x10==0x10)
```

```
  GPIO.output(LCD_D5, bits&0x20==0x20)
  GPIO.output(LCD_D6, bits&0x40==0x40)
  GPIO.output(LCD_D7, bits&0x80==0x80)
  lcd_enable()
  GPIO.output(LCD_D4, bits&0x01==0x01)
  GPIO.output(LCD_D5, bits&0x02==0x02)
  GPIO.output(LCD_D6, bits&0x04==0x04)
  GPIO.output(LCD_D7, bits&0x08==0x08)
  lcd_enable()

def lcd_string(message):
  message = message.ljust(LCD_WIDTH," ")
  for i in range(LCD_WIDTH):
    lcd_byte(ord(message[i]),LCD_CHR)

def lcd_anzeige(z1, z2):
  lcd_byte(LCD_LINE_1, LCD_CMD)
  lcd_string(z1)
  lcd_byte(LCD_LINE_2, LCD_CMD)
  lcd_string(z2)

LCD_INIT = [0x33, 0x33, 0x32, 0x28, 0x0C, 0x06, 0x01]
for i in LCD_INIT:
  lcd_byte(i,LCD_CMD)
  time.sleep(INIT)

lcd_anzeige("0123456789ABCDEF", " www.franzis.de")
GPIO.cleanup()
```

9.3 | So funktioniert es

Dass das Programm funktioniert, lässt sich einfach ausprobieren. Jetzt stellen sich natürlich Fragen: Was passiert im Hintergrund? Was bedeuten die einzelnen Programmzeilen?

Am Anfang werden bereits bekannte Bibliotheken importiert, um die GPIO-Schnittstelle zu nutzen.

```
LCD_RS = 7
LCD_E  = 8
LCD_D4 = 25
LCD_D5 = 24
LCD_D6 = 23
LCD_D7 = 18
```

Diese Zeilen definieren die Konstanten, in denen die Nummern der GPIO-Ports für Steuerleitungen und Datenleitungen des Displays gespeichert sind.

```
GPIO.setmode(GPIO.BCM)
```

Danach wird Nummerierung der GPIO-Pins auf `BCM`-Standard gesetzt.

```
GPIO.setup(LCD_E,  GPIO.OUT)
GPIO.setup(LCD_RS, GPIO.OUT)
GPIO.setup(LCD_D4, GPIO.OUT)
GPIO.setup(LCD_D5, GPIO.OUT)
GPIO.setup(LCD_D6, GPIO.OUT)
GPIO.setup(LCD_D7, GPIO.OUT)
```

Die für das Display verwendeten Ports werden alle als Ausgang definiert. Die Portnummern werden über die zuvor definierten Variablen angegeben. Diese Methode hat den Vorteil, dass die tatsächlichen GPIO-Ports nur an dieser einen Stelle im Programm auftauchen. So können Sie das Programm ganz einfach umbauen, wenn Sie andere GPIO-Ports nutzen möchten.

```
LCD_WIDTH = 16
LCD_LINE_1 = 0x80
LCD_LINE_2 = 0xC0
LCD_CHR = 1
LCD_CMD = 0
E_PULSE = 0.00005
E_DELAY = 0.00005
INIT  = 0.01
```

Diese Zeilen definieren weitere Konstanten, die im Programm verwendet werden.

KONSTANTE	BEDEUTUNG
LCD_WIDTH	Breite des Displays in Zeichen.
LCD_LINE_1	Speicheradresse der 1. Zeile.
LCD_LINE_2	Speicheradresse der 2. Zeile.
LCD_CHR	Schaltet das Display auf Pin RS in den Zeichenmodus.
LCD_CMD	Schaltet das Display auf Pin RS in den Steuerungsmodus.
E_PULSE	Dauer eines Steuerimpulses auf Pin E.
E_DELAY	Wartezeit zwischen zwei Steuerimpulsen auf Pin E.
INIT	Wartezeit zwischen zwei Initialisierungskommandos.

```
def lcd_enable():
```

Jetzt wird eine Funktion definiert, die später im Programm aufgerufen wird. Sie sendet einen kurzen Steuerimpuls an den Pin E (*enable*). Damit liest das Display die Daten auf den GPIO-Ports aus, um sie anzuzeigen. Im 4-Bit-Modus wird der gleiche Impuls verwendet, um zwischen den oberen und den unteren Bits eines darzustellenden Bytes umzuschalten.

```
    time.sleep(E_DELAY)
```

Zuerst wird eine kurze in `E_DELAY` gespeicherte Zeit (0,05 ms) gewartet. Das Display hat eine gewisse Trägheit. Man kann nicht unmittelbar nach dem letzten Zeichen auf das andere Halbbyte umschalten.

```
    GPIO.output(LCD_E, 1)
```

Jetzt wird auf Pin E das Logiksignal `1` ausgegeben.

```
    time.sleep(E_PULSE)
    GPIO.output(LCD_E, 0)
```

Das Signal liegt über die in `E_PULSE` gespeicherte Zeit am Pin E an, danach wird der Pin wieder auf `0` zurückgesetzt.

```
time.sleep(E_DELAY)
```

Das Programm wartet noch die in `E_DELAY` gespeicherte Zeit ab, bevor die Funktion beendet wird, damit danach sofort wieder Daten an das Display gesendet werden können.

```
def lcd_byte(bits, mode):
```

Diese Funktion sendet ein Byte an das Display. Dieses Byte kann ein Zeichen oder ein Steuerbefehl sein. Im Parameter `bits` bekommt die Funktion das zu sendende Byte, der Parameter `mode` gibt an, ob es sich um ein Zeichen oder um einen Steuerungsbefehl handelt. Die beiden Werte, die `mode` annehmen kann, sind in den Konstanten `LCD_CHR` (Zeichen) und `LCD_CMD` (Steuerungsbefehl) festgelegt.

```
GPIO.output(LCD_RS, mode)
```

Der zu verwendende Modus wird auf den Pin RS (*Register Select*) ausgegeben. Damit wird das Display in den Zeichenmodus oder den Steuerungsmodus geschaltet.

```
GPIO.output(LCD_D4, bits&0x10==0x10)
GPIO.output(LCD_D5, bits&0x20==0x20)
GPIO.output(LCD_D6, bits&0x40==0x40)
GPIO.output(LCD_D7, bits&0x80==0x80)
```

Das Programm verwendet den 4-Bit-Modus des Displays. Zunächst werden die oberen 4 Bit des zu sendenden Bytes auf den Datenleitungen ausgegeben. Dazu wird der Python-Operator `&` (das bitweise UND) verwendet. Nacheinander wird für jedes der vier Bits geprüft, ob es `1` oder `0` ist. Das Ergebnis der Prüfung ist entsprechend `True` oder `False`. Dieser Wert wird dann auf der jeweiligen Datenleitung ausgegeben.

BIT-NR.	4 HIGH	3 HIGH	2 HIGH	1 HIGH	4 LOW	3 LOW	2 LOW	1 LOW
Pin	D7	D6	D5	D4	D7	D6	D5	D4
Binär	1000 0000	0100 0000	0010 0000	0001 0000	0000 1000	0000 0100	0000 0010	0000 0001
Dezimal	128	64	32	16	8	4	2	1
Hex	0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01

```
    lcd_enable()
```

Diese zuvor definierte Funktion liest die Daten von den GPIO-Ports und schaltet dann das Display auf das untere Halbbyte um.

```
    GPIO.output(LCD_D4, bits&0x01==0x01)
    GPIO.output(LCD_D5, bits&0x02==0x02)
    GPIO.output(LCD_D6, bits&0x04==0x04)
    GPIO.output(LCD_D7, bits&0x08==0x08)
```

Nach dem gleichen Schema werden danach die unteren 4 Bit des zu sendenden Bytes auf den Datenleitungen ausgegeben …

```
    lcd_enable()
```

… und danach wird wieder auf das obere Halbbyte umgeschaltet.

```
def lcd_string(message):
```

Diese Funktion schreibt eine Zeichenfolge auf das Display. Die Zeichenfolge wird vom aufrufenden Programm im Parameter `message` an die Funktion übergeben.

```
    message = message.ljust(LCD_WIDTH," ")
```

Die Zeichenkette wird bis auf die Länge des Displays rechts mit Leerzeichen aufgefüllt. Die Länge des Displays, in unserem Fall 16 Zeichen, ist in der Konstanten `LCD_WIDTH` gespeichert. Die Methode `.ljust` kann in jeder

Zeichenkette angewendet werden, der zweite Parameter gibt das Zeichen an, mit dem kürzere Zeichenketten aufgefüllt werden. Längere Zeichenketten werden automatisch auf die angegebene Länge abgeschnitten.

```
for i in range(LCD_WIDTH):
   lcd_byte(ord(message[i]),LCD_CHR)
```

Diese Schleife läuft so oft durch, wie das Display Zeichen hat. In jedem Durchlauf wird ein Zeichen der Zeichenkette `message` zuerst mit der Python-Funktion `ord()` in seinen ASCII-Zahlenwert umgewandelt und dann im Zeichenmodus `LCD_CHR` über die zuvor definierte Funktion `lcd_byte()` auf das Display ausgegeben.

```
def lcd_anzeige(z1, z2):
```

Diese Funktion bekommt vom aufrufenden Programm zwei Zeichenketten und sorgt dafür, dass diese auf den beiden Zeilen des Displays dargestellt werden.

```
  lcd_byte(LCD_LINE_1, LCD_CMD)
  lcd_string(z1)
```

Zuerst wird die Adresse der oberen Zeile des Displays im Steuerungsmodus `LCD_CMD` auf das Display ausgegeben. Damit wird festgelegt, dass die folgenden Ausgaben im Zeichenmodus auf der oberen Zeile erscheinen. Anschließend schreibt die Funktion `lcd_string()` die erste Zeichenkette auf das Display.

```
  lcd_byte(LCD_LINE_2, LCD_CMD)
  lcd_string(z2)
```

Auf die gleiche Weise wird anschließend die zweite Zeichenkette in die untere Zeile des Displays geschrieben.

Nachdem die vier Funktionen `lcd_enable()`, `lcd_byte()`, `lcd_string()` und `lcd_anzeige()` definiert sind, startet jetzt das eigentliche Programm.

```
LCD_INIT = [0x33, 0x33, 0x32, 0x28, 0x0C, 0x06, 0x01]
```

Bevor das Display verwendet werden kann, muss es initialisiert werden. Eine Folge von Initialisierungskommandos bringt es in einen eindeutig definierten Zustand und setzt es auf den 4-Bit-Modus.

```
for i in LCD_INIT:
    lcd_byte(i,LCD_CMD)
    time.sleep(INIT)
```

Diese Schleife schreibt nacheinander die in der Liste `LCD_INIT` eingetragenen Initialisierungskommandos im Steuerungsmodus `LCD_CMD` auf das Display. Zur Datenübertragung wird die weiter oben definierte Funktion `lcd_byte()` verwendet. Zwischen den Initialisierungskommandos wird 0,01 Sekunden gewartet.

Nachdem alles definiert und initialisiert ist, kann es wirklich losgehen.

```
lcd_anzeige("0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ",
" www.franzis.de")
```

Diese Zeile gibt die beiden Zeichenketten an, die dargestellt werden sollen. Dabei spielt die Länge keine Rolle, da die Zeichenketten von der weiter oben definierten Funktion `lcd_string()` automatisch auf eine Länge von jeweils 16 Zeichen gebracht werden.

```
GPIO.cleanup()
```

Danach ist das Programm auch schon zu Ende. Als Letztes muss, wie in allen Python-Programmen, die GPIO-Schnittstelle geschlossen werden, um Warnungen beim nächsten Start eines Programms zu verhindern. Nach Abschluss bleibt die Anzeige auf dem Display aber trotzdem bestehen. Das Display hat keinen ständigen Kontakt zu den GPIO-Ports, sondern liest die Daten nur aus, wenn das Enable-Signal umgeschaltet wird. Danach bleibt die Anzeige im Display wieder unverändert, unabhängig davon, ob Daten an den Datenleitungen anliegen.

10 LC-Display im 8-Bit-Modus

Das nächste Projekt zeigt, wie das LC-Display im 8-Bit-Modus verwendet wird. Verbinden Sie dazu, wie in der Abbildung gezeigt, mit vier weiteren Verbindungskabeln die Datenleitungen `D0...D3` des Displays mit den GPIO-Ports `7`, `8`, `9` und `10`.

Zur Übersicht auch hier noch einmal eine Tabelle der Datenleitungen des Displays und der verwendeten GPIO-Ports. Die Leitungen `D4...D7` sind genau so wie im vorhergehenden Projekt angeschlossen.

DISPLAYPIN	SIGNAL	RASPBERRY PI GPIO
4	RS	GPIO 7
6	E	GPIO 8
7	D0	GPIO 21
8	D1	GPIO 20
9	D2	GPIO 16
10	D3	GPIO 12
11	D4	GPIO 25
12	D5	GPIO 24
13	D6	GPIO 23
14	D7	GPIO 18

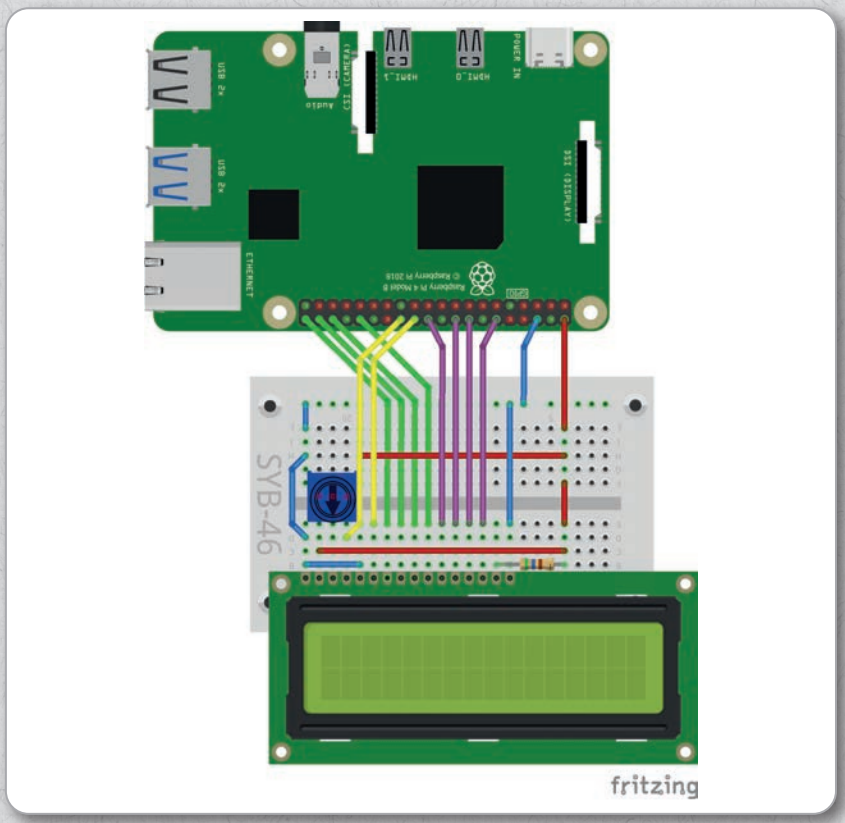


Abb. 10.1: *LC-Display im 8-Bit-Modus. Die zusätzlichen Leitungen sind grün dargestellt.*

Benötigte Bauteile

- 1 x Steckplatine,
- 1 x LC-Display,
- 1 x 560-Ohm-Widerstand (Grün-Blau-Braun),
- 1 x Potenziometer,
- 12 x Verbindungskabel,
- 6 x Drahtbrücke (unterschiedliche Längen)

10.1 | So funktioniert es

Das Programm `10display08.py` basiert auf dem letzten Programm. Wir zeigen hier nur die Änderungen für den 8-Bit-Modus.

```
LCD_RS = 25             LCD_D3 = 10
LCD_E  = 24             LCD_D4 = 23
LCD_D0 = 7              LCD_D5 = 17
LCD_D1 = 8              LCD_D6 = 27
LCD_D2 = 9              LCD_D7 = 22
```

Bei den Definitionen der GPIO-Portnummern werden vier weitere Variablen für die zusätzlichen vier Datenleitungen deklariert.

```
GPIO.setmode(GPIO.BCM)
GPIO.setup(LCD_E,  GPIO.OUT)
GPIO.setup(LCD_RS, GPIO.OUT)
GPIO.setup(LCD_D0, GPIO.OUT)
GPIO.setup(LCD_D1, GPIO.OUT)
GPIO.setup(LCD_D2, GPIO.OUT)
GPIO.setup(LCD_D3, GPIO.OUT)
GPIO.setup(LCD_D4, GPIO.OUT)
GPIO.setup(LCD_D5, GPIO.OUT)
GPIO.setup(LCD_D6, GPIO.OUT)
GPIO.setup(LCD_D7, GPIO.OUT)
```

Auch diese vier GPIO-Ports werden als Ausgänge initialisiert. Die weiteren Variablendeklarationen sowie die Funktion `lcd_enable()` bleiben unverändert.

```
def lcd_byte(bits, mode):
  GPIO.output(LCD_RS, mode)
  GPIO.output(LCD_D0, bits&0x01==0x01)
  GPIO.output(LCD_D1, bits&0x02==0x02)
  GPIO.output(LCD_D2, bits&0x04==0x04)
  GPIO.output(LCD_D3, bits&0x08==0x08)
  GPIO.output(LCD_D4, bits&0x10==0x10)
  GPIO.output(LCD_D5, bits&0x20==0x20)
```

```
GPIO.output(LCD_D6, bits&0x40==0x40)
GPIO.output(LCD_D7, bits&0x80==0x80)
lcd_enable()
```

Die Funktion `lcd_byte()` errechnet weiterhin die 8 Bit jedes Zeichens, gibt diese jetzt aber nicht mehr in zwei Blöcken, sondern auf einmal auf acht Datenleitungen aus. Die Funktionen `lcd_string()` und `lcd_anzeige()` sind wieder unverändert, weil sie sich nur mit der Umsetzung von Zeichenketten und nicht mit der direkten Ansteuerung der Displayhardware befassen.

```
LCD_INIT = [0x33, 0x33, 0x32, 0x3C, 0x0C, 0x06, 0x01]
```

Um das Display im 8-Bit-Modus zu verwenden, muss sich der Initialisierungsstring im vierten Zeichen unterscheiden:

INITIALISIERUNGSSTRING	
[0x33, 0x33, 0x32, 0x28, 0x0C, 0x06, 0x01]	4-Bit-Modus
[0x33, 0x33, 0x32, 0x3C, 0x0C, 0x06, 0x01]	8-Bit-Modus

Die restlichen Programmanweisungen können unverändert aus der Programmversion für den 4-Bit-Modus übernommen werden.

4 Bit oder 8 Bit – welcher Modus ist besser?

Vergleicht man den Verkabelungs- und Programmieraufwand für den 4-Bit-Modus und den 8-Bit-Modus, stellt man schnell fest, dass der 4-Bit-Modus die bessere Wahl ist. Im 4-Bit-Modus werden nur sechs statt zehn GPIO-Ports benötigt, und das Programm ist sogar um ein paar Zeilen kürzer – zumindest bei Verwendung von Python. Aus diesen Gründen verwenden wir für die folgenden Beispiele auch wieder den 4-Bit-Modus. Pi_Scratch unterstützt den 8-Bit-Modus nicht.

Natürlich werden Sie sich fragen, warum es überhaupt noch einen 8-Bit-Modus gibt, wenn der 4-Bit-Modus doch offenbar nur Vorteile bietet. Die Hersteller von LC-Displays könnten sich vier Anschlüsse sparen. Im Experiment »LC-Display am Portexpander« sehen Sie jedoch, dass der 8-Bit-Modus in bestimmten Anwendungsfällen auch seine Stärken hat.

11 IP-Adresse des Raspberry Pi anzeigen

Betreibt man einen Raspberry Pi headless ohne Monitor, ist es nicht immer ganz einfach, die IP-Adresse herauszufinden, die man für eine SSH-Verbindung zur Administration des Servers benötigt. Mit dem LC-Display und einem Python-Skript lässt sich diese Aufgabe elegant lösen.

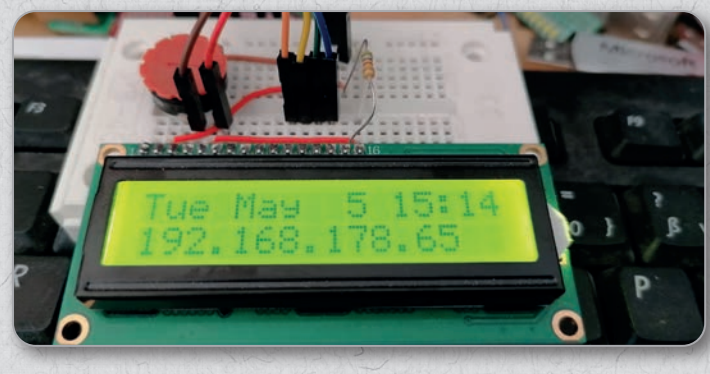


Abb. 11.1: *Das LC-Display zeigt über ein Python-Skript die aktuelle Systemzeit und die IP-Adresse an.*

11.1 | So funktioniert es

Das Programm `11status.py` übernimmt die Funktionen zur Displaysteuerung im 4-Bit-Modus aus dem weiter oben beschriebenen Programm `09display01.py`. Auch der Schaltungsaufbau und die verwendeten GPIO-Ports sind die gleichen. Im Folgenden werden nur die neuen Programmelemente beschrieben.

```
#!/usr/bin/python
import RPi.GPIO as GPIO
import time, os
```

Ganz am Anfang wird zusätzlich die Bibliothek `os` importiert, mit der man Linux-Kommandos aus Python-Programmen heraus aufrufen und auswerten kann.

```
pause = 2
```

Bei den Variablendeklarationen wird zusätzlich eine Variable `pause` eingeführt, die das Aktualisierungsintervall der Uhrzeit festlegt. Der Wert `2` bedeutet, dass die Uhrzeit, die nur auf Minuten genau angezeigt wird, alle zwei Sekunden aktualisiert wird. Dies ist ein guter Kompromiss zwischen Genauigkeit und Systemauslastung.

Die Funktionen zum Zugriff auf das Display wie auch die Initialisierungssequenz werden unverändert übernommen.

Anstelle einer einfachen statischen Textausgabe wird jetzt eine Endlosschleife verwendet, die immer wieder die Uhrzeit und auch die IP-Adresse aktualisiert.

```
try:
  while True:
```

Diese Endlosschleife wird so lange wiederholt, bis die weiter unten bei `except` festgelegte Abbruchbedingung eintritt.

```
    zeile1 = time.asctime()
```

Zur Ausgabe von Datum und Uhrzeit wird die Funktion `time.asctime()` aus dem Modul `time` verwendet. Sie liefert Datum und Uhrzeit in einer Zeichenfolge, z. B. `Tue May 5 15:14:59 2020`. Die ersten 16 Stellen dieser Zeichenfolge zeigen Wochentag, Monat, Tag, Stunden und Minuten auf dem 16-stelligen Display an. Die Ausgabe dieser Funktion wird in der neuen Variablen `zeile1` gespeichert.

```
    zeile2 = os.popen("hostname -I").readline()[:-2]
```

In der Variablen `zeile2` wird die IP-Adresse gespeichert. Dafür wird die Funktion `os.popen()` aus dem Modul `os` verwendet, die die Ausgabe eines beliebigen Kommandozeilenbefehls als Zeichenkette liefert. Der Kommandozeilenbefehl `hostname -I` liefert die IP-Adresse. Diese Zeichenkette wird bis auf die letzten zwei Zeichen verwendet, das Zeilenendezeichen `\n` der Ausgabe sowie ein Leerzeichen am Ende werden abgeschnitten.

```
    lcd_anzeige(zeile1, zeile2)
```

Die Inhalte dieser beiden Variablen werden dann über die bereits im Grundprogramm definierte Funktion `lcd_anzeige()` auf dem Display ausgegeben.

```
    time.sleep(pause)
```

Nach der Ausgabe wartet das Programm die als Pause definierte Zeit (Standard: zwei Sekunden). Danach startet die Schleife neu und zeigt wieder die neue Zeit an.

```
except KeyboardInterrupt:
  GPIO.cleanup()
```

Drückt der Benutzer die Tastenkombination Strg+C, wird ein `KeyboardInterrupt` ausgelöst und die Schleife automatisch verlassen. Die letzte Zeile schließt die verwendeten GPIO-Ports. Danach wird das Programm beendet. Nach dem Beenden des Programms bleibt die Anzeige auf dem Display stehen, die Uhr läuft aber nicht mehr weiter. Die Anzeige selbst wird durch den im Display eingebauten Controller aufrechterhalten.

Durch das kontrollierte Schließen der GPIO-Ports tauchen beim nächsten Start eines Programms, das die GPIO-Schnittstelle nutzt, keine Systemwarnungen oder Abbruchmeldungen auf, die den Benutzer verwirren könnten.

11.2 | Programm automatisch starten

Dieses Programm ist besonders dann nützlich, wenn am Raspberry Pi kein Monitor angeschlossen ist. In diesem Fall kann es beim Booten automatisch gestartet werden.

Bearbeiten Sie im Dateimanager die Textdatei `autostart` im Verzeichnis `/etc/xdg/lxsession/LXDE-pi`. Da diese Datei nur mit Superuserrechten bearbeitet werden kann, wechseln Sie zuerst mit dem Dateimanager in das Verzeichnis und wählen dann im Menü *Werkzeuge/Ausführen eines Befehls im aktuellen Ordner...* Geben Sie hier die abgebildete Befehlsfolge ein.

Fügen Sie am Ende der Datei die abgebildete Zeile zum Aufruf des Programms mit dem Python-Kommandozeileninterpreter hinzu:

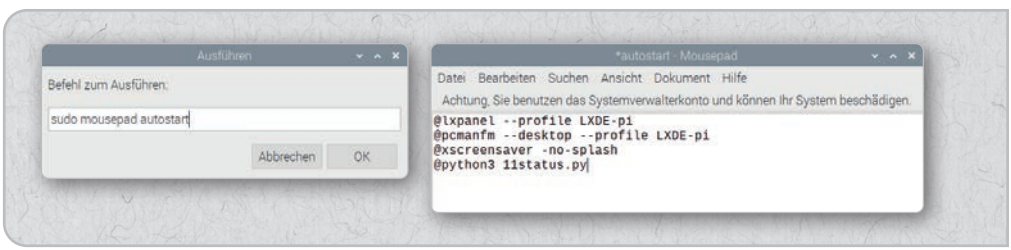


Abb. 11.2: *Links: Datei autostart mit Superuserrechten bearbeiten. Rechts: Der Autostartaufruf für die Statusanzeige.*

Damit werden kurz nach dem Booten des Raspberry Pi die aktuelle Uhrzeit sowie die IP-Adresse automatisch auf dem LCD-Modul angezeigt.

Um den automatischen Start abzuschalten, löschen Sie die Zeile wieder, oder Sie tragen ganz am Anfang der Zeile ein #-Zeichen ein:

```
# @python3 11status.py
```

Die automatisch gestartete Statusanzeige lässt sich nicht über die Tastenkombination Strg+C beenden, da kein Python-Shell-Fenster existiert.

12 Laufschrift auf dem LC-Display

LC-Displays und ähnliche zeichenorientierte Anzeigeelemente werden gern für Laufschriften verwendet. Auf diese Weise ist es möglich, auch längere Texte in den relativ kurzen Zeilen der Anzeige darzustellen.

Dieses Experiment stellt einen beliebigen Text als Laufschrift in der unteren Zeile des Displays dar, in der oberen Zeile läuft die aus dem letzten Experiment bekannte Uhr.

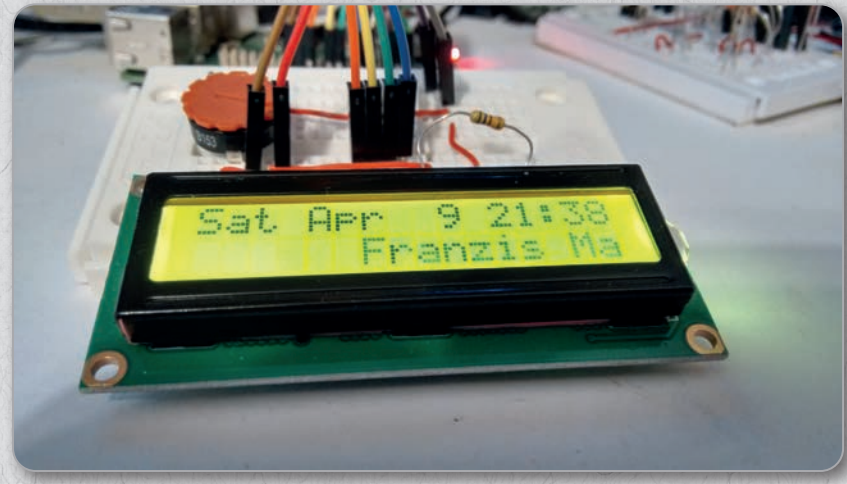


Abb. 12.1: *Uhr und Laufschrift auf dem LC-Display.*

12.1 | So funktioniert es

Das Programm `12laufschrift.py` basiert auf dem Programm `11status.py`. Für die Laufschrift braucht nur wenig verändert zu werden.

```
pause = 0.3
```

Die Pause wird auf 0,3 Sekunden verkürzt, sie gibt jetzt das Intervall an, in dem die Laufschrift um einen Buchstaben vorrückt. Verlängert man die Pause, läuft die Laufschrift langsamer.

```
tx = " " * LCD_WIDTH + input("Bitte Text eingeben: ")
```

Diese Zeile fragt über die Funktion `input()` nach einer Benutzereingabe.

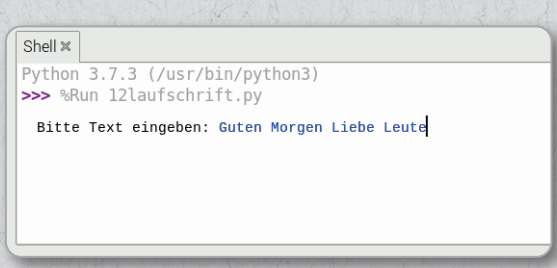


Abb. 12.2: *Die Python-Shell wartet auf eine Benutzereingabe.*

Der eingegebene Text wird in der Variablen `tx` gespeichert, davor werden so viele Leerzeichen gesetzt, wie das Display breit ist – in unserem Fall 16. In Python kann man einfach eine Zeichenkette mit einer Zahl multiplizieren und sie damit mehrfach wiederholen. Durch Verwendung der Konstanten `LCD_WIDTH` anstelle einer Zahl ist das Programm auch auf breiteren Displays nutzbar.

```
try:
    while True:
        for j in range(len(tx)):
```

Innerhalb der Hauptschleife des Programms läuft jetzt eine weitere Schleife über die Länge des Texts.

```
        zeile1 = time.asctime()
        zeile2 = tx[j:(j + LCD_WIDTH)]
        lcd_anzeige(zeile1, zeile2)
```

In jedem Durchlauf wird in der oberen Zeile des Displays die aktuelle Uhrzeit angezeigt, in der unteren Zeile ein 16 Zeichen langer Ausschnitt von `tx`. Dieser beginnt jedes Mal ein Zeichen weiter hinten. Da `tx` am Anfang 16 Leerzeichen enthält, beginnt die Laufschrift mit einem leeren Display, der Text wandert von rechts Zeichen für Zeichen ins Bild. Am Ende des Texts sind keine zusätzlichen Leerzeichen erforderlich, diese werden von der bereits bekannten Funktion `lcd_string()` bei Bedarf automatisch am Ende angehängt.

```
        time.sleep(pause)
```

Nachdem der Textausschnitt dargestellt wurde, wartet das Programm 0,3 Sekunden, danach erscheint der nächste um ein Zeichen verschobene Textausschnitt.

Die innere Schleife endet, nachdem das letzte Zeichen von `tx` am linken Rand des Displays verschwunden ist, und startet danach neu. Die äußere Endlosschleife läuft, bis der Benutzer mit Strg+C das Programm abbricht.

Deutsche Umlaute auf dem LC-Display 13

Die LC-Displays verschiedener Hersteller verwenden diverse unterschiedliche Zeichensätze, die sich nur in den ersten 128 Zeichen, dem Standard-ASCII-Zeichensatz, gleichen. Das im Maker Kit enthaltene LCD-Modul hat einen Zeichensatz, der die deutschen Umlaute sowie das ß zwar enthält, aber nicht an den Stellen, an denen ein PC diese Zeichen erwartet. Das können Sie einfach ausprobieren, wenn Sie im Programm `12laufschrift.py` einen Text eingeben, der Umlaute enthält.

13.1 | Zeichensatz des LC-Displays anzeigen

Um in späteren Programmen die Umlaute richtig zuordnen zu können, müssen deren Positionen im Zeichensatz bekannt sein. Das Programm `13zeichensatz.py` basiert auf dem Programm `11status.py` und zeigt den kompletten Zeichensatz eines LC-Displays an. Da die Programmteile zur Ansteuerung des LC-Displays bereits bekannt sind, hier nur die neuen Elemente.

```
try:
    while True:
        for j in range(16):
            zeile1 = "Zeichen: " + str(j*16) + "-" + str(j*16+15)
            zeile2 = ""
            for k in range(16):
                zeile2 += chr(j*16+k)
            print(zeile1 + " " + zeile2)
            lcd_anzeige(zeile1, zeile2)
            time.sleep(pause)
```

Das Programm zeigt seitenweise je 16 Zeichen des Zeichensatzes an. Die Bilder zeigen, wo die Umlaute im Zeichensatz liegen.

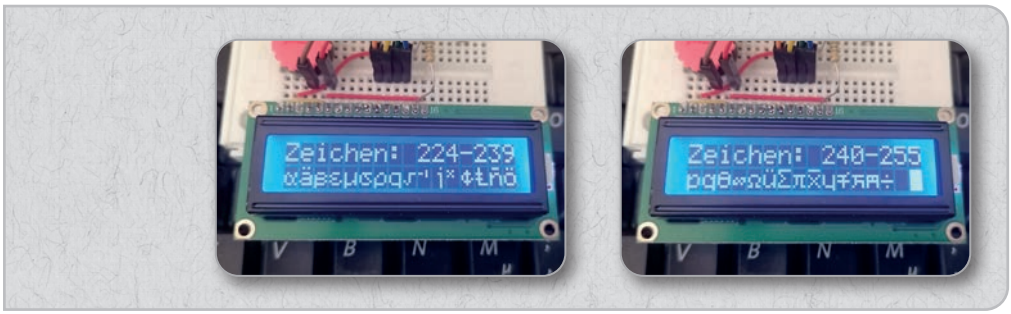


Abb. 13.1: *Umlaute im Zeichensatz des LCD-Moduls.*

ZEICHEN	DEZIMALCODE
ä	225
ö	239
ü	245
ß	226

13.1.1 | So funktioniert es

Das Programm zeigt nacheinander 16 Displayseiten. Auf jeder Seite zeigt die obere Zeile den Nummernbereich der angezeigten Zeichen, die untere Zeile zeigt 16 aufeinanderfolgende Zeichen.

Innerhalb der Endlosschleife läuft eine Schleife mit dem Zähler J, die in jedem Durchlauf eine der 16 Seiten darstellt.

Zur Anzeige der oberen Zeile `zeile1` wird der Schleifenzähler `j` mit 16 multipliziert. Daraus ergibt sich die Nummer des ersten angezeigten Zeichens. Die Nummer des letzten angezeigten Zeichens ist um 15 höher. Die Python-Standardfunktion `str()` schreibt eine Zahl in eine Zeichenfolge, um daraus die komplette Zeile zusammenzubauen.

Die Variable `zeile2` für die untere Zeile wird zu Beginn des Durchlaufs auf die Zeichenfolge "" gesetzt, und damit wird der bisherige Inhalt gelöscht.

Innerhalb dieser Schleife baut eine weitere Schleife mit dem Schleifenzähler `k` die Zeichenfolge für die untere Zeile aus 16 Zeichen zusammen. Die Python-Standardfunktion `chr()` ermittelt ein Zeichen anhand seiner Nummer im Zeichensatz. Durch Hochzählen des Schleifenzählers `k` entsteht so in der Variablen `zeile2` eine Zeichenfolge aus 16 aufeinanderfolgenden Zeichen.

Danach zeigt `print()` die Zeichennummern sowie die 16 Zeichen im Shell-Fenster an, die bereits aus früheren Programmen bekannte Funktion `lcd_anzeige()` zeigt die gleichen Daten auf dem LC-Display. Im Vergleich sehen Sie die Unterschiede zwischen dem Standardzeichensatz von Linux und dem Zeichensatz des LC-Displays.

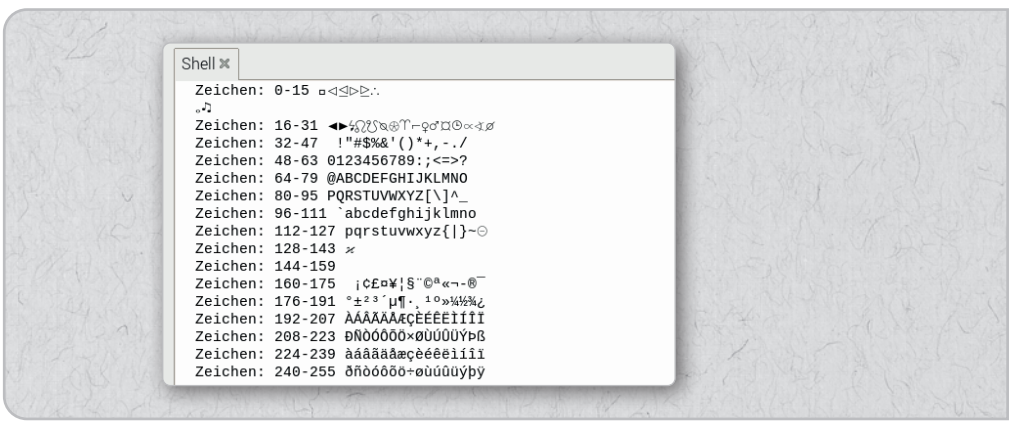


Abb. 13.2: *Der Linux-Zeichensatz im Shell-Fenster.*

13.2 | Umlaute durch passende Zeichen ersetzen

Das Programm `13umlaute.py` zeigt, wie sich Umlaute in Zeichenketten austauschen lassen, um sie auf dem LC-Display richtig darzustellen. Das Programm basiert auf dem Programm `12laufschrift.py`.

Die beiden Zeilen des LC-Displays zeigen, oben unverändert und unten mit ausgetauschten Zeichen, einen Satz an, der am Anfang des Programms in der Variablen `tx` festgelegt wird.

```
tx = "Zwölf Boxkämpfer jagen Victor quer über den großen Sylter Deich"
```

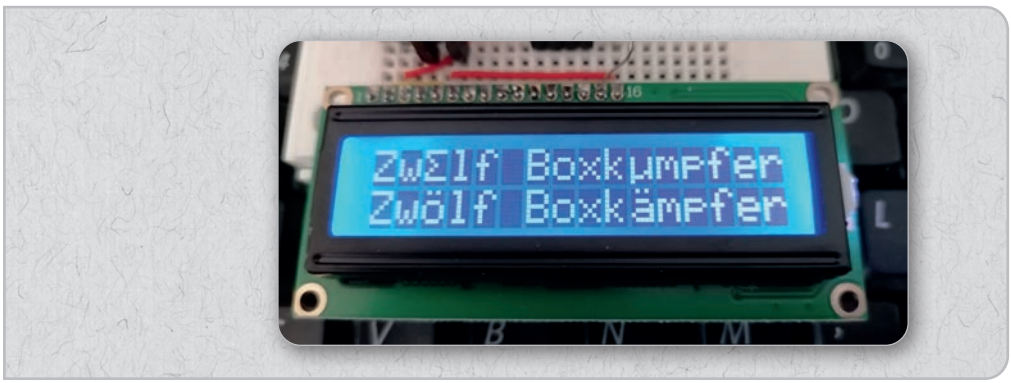


Abb. 13.3: *Text im Originalzeichensatz und mit umgesetzten Umlauten.*

Pangramme – alle Buchstaben des Alphabets in einem Satz

Wundern Sie sich nicht über diesen scheinbar sinnlosen Satz: *Zwölf Boxkämpfer jagen Victor quer über den großen Sylter Deich.* Der Satz, ein sogenanntes Pangramm, enthält alle Buchstaben des Alphabets, Umlaute und ß eingeschlossen. Ein echtes Pangramm, in dem jeder Buchstabe der deutschen Sprache genau einmal vorkommt, lautet: *„Fix, Schwyz!" quäkt Jürgen blöd vom Paß.* (Ein Schweizer Sportfan feuert von einer Bergstraße seine Mannschaft an.) Solche Pangramme werden häufig zum Testen von Schriftarten, Druckern oder eben LCD-Modulen verwendet.

13.2.1 | So funktioniert es

Da die Programmteile zur Ansteuerung des LC-Displays bereits bekannt sind, hier nur die neuen Elemente.

```
z1 = " " * LCD_WIDTH + tx
z2 = z1.replace("ä", chr(225))
z2 = z2.replace("ö", chr(239))
z2 = z2.replace("ü", chr(245))
z2 = z2.replace("ß", chr(226))
z2 = z2.replace("Ä", chr(225))
z2 = z2.replace("Ö", chr(239))
z2 = z2.replace("Ü", chr(245))
```

Die Variable `z1` enthält den Text mit vorangestellten Leerzeichen, um ihn als Laufschrift von rechts in das LC-Display laufen zu lassen.

Die Python-Methode `.replace()` ersetzt in einer Zeichenkette alle Vorkommen eines bestimmten Zeichens durch ein anderes, das in diesem Programm über seine Nummer im Zeichensatz des LC-Displays angegeben wird. In der Variablen `z1` werden auf diese Weise alle `"ä"` ersetzt, und das Ergebnis wird in `z2` gespeichert.

In den nächsten Schritten werden in `z2` auch die anderen Umlaute ersetzt. Da der Zeichensatz des LC-Displays keine großen Umlaute enthält, werden stattdessen die kleinen genutzt.

```
try:
    while True:
        for j in range(len(tx)):
            zeile1 = z1[j:(j + LCD_WIDTH)]
            zeile2 = z2[j:(j + LCD_WIDTH)]
            lcd_anzeige(zeile1, zeile2)
            time.sleep(pause)
```

Die Hauptschleife des Programms zeigt, ähnlich wie das Programm `12laufschrift.py`, zwei Laufschriften an, oben der unveränderte Text, bei dem Umlaute auf dem LC-Display fehlerhaft dargestellt werden, und unten der Text mit ausgetauschten Umlauten.

14 Erweiterte Statusanzeige

Das nächste Programm erweitert die Statusanzeige um Anzeigen des freien Speicherplatzes auf der SD-Karte und auf bis zu drei angeschlossene USB-Sticks. USB-Sticks benötigen so wenig Strom, dass man auch mehrere davon über einen Hub ohne eigene Stromversorgung gleichzeitig am Raspberry Pi anschließen kann. Verwendet man eine Tastatur mit eingebautem USB-Anschluss für die Maus, passen noch drei USB-Sticks ohne Hub an den Raspberry Pi.

Abb. 14.1: *Statusanzeige für drei USB-Sticks, am Raspberry Pi angeschlossen.*

Die Schaltung entspricht der aus den vorhergehenden 4-Bit-Experimenten.

Dateisysteme auf USB-Sticks

USB-Sticks sind üblicherweise im vfat-Dateisystem formatiert. Linux kann dieses Dateisystem lesen und schreiben, nur die Rechteverwaltung gibt es unter vfat nicht. Lassen Sie dieses Dateisystem auf dem USB-Stick bestehen, können Sie leicht Daten von beliebigen PCs auf Ihren Server übertragen, indem Sie nur den USB-Stick umstecken. Windows kann dagegen mit einem Linux-formatierten USB-Stick nichts anfangen.

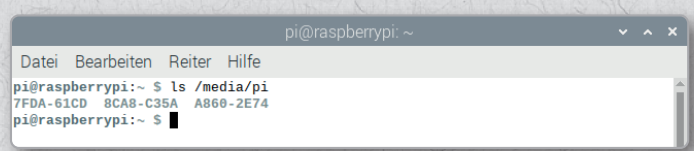
Im Definitionsbereich des Programms `14status_sdusb.py` werden die drei Namen der angeschlossenen USB-Medien festgelegt. Damit das Programm funktioniert, müssen die drei USB-Sticks in diese drei Zeilen am Anfang des Programms vor dem ersten Programmstart eingetragen werden.

```
HD_2 = "/media/pi/7FDA-61CD"
HD_3 = "/media/pi/8CA8-C35A"
HD_1 = "/media/pi/2C33-0235"
```

Die UUIDs der USB-Sticks werden verwendet, solange die USB-Sticks keine festen Namen haben.

In einem LXTerminal-Fenster können Sie die UUIDs anzeigen lassen:

```
Ls /media/pi
```



14.1 | So funktioniert es

Das Programm `14status_sdusb.py` basiert auf dem Programm `11status.py`, zeigt aber zusätzlich nach der Uhrzeit und der IP-Adresse im Wechsel von zwei Sekunden noch den freien Speicherplatz auf der SD-Karte und den drei angeschlossenen USB-Sticks an. Dazu enthält es einige zusätzliche Programmzeilen:

```
pause = 2
```

Die Pause bis zur Aktualisierung der Anzeige wird auf zwei Sekunden gesetzt. In dieser Zeit kann man die Werte auf der Anzeige bequem lesen und wartet nicht zu lange auf die nächste Anzeigeseite.

```
HD_1 = "/media/pi/7FDA-61CD"
HD_2 = "/media/pi/8CA8-C35A"
HD_3 = "/media/pi/A860-2E74"
```

In die drei Programmzeilen müssen vor dem ersten Start die UUIDs der angeschlossenen USB-Sticks eingetragen werden.

Die wichtigsten Änderungen finden sich in der Hauptschleife des Programms, die jetzt aus drei Blöcken besteht, die nacheinander unterschiedliche Informationen anzeigen.

```
try:
  while True:
    sd1 = os.statvfs(,/‘)
    hd1 = os.statvfs(HD_1)
    hd2 = os.statvfs(HD_2)
    hd3 = os.statvfs(HD_3)
    sd1x = sd1.f_bsize * sd1.f_bavail // 1048576
    hd1x = hd1.f_bsize * hd1.f_bavail // 1048576
    hd2x = hd2.f_bsize * hd2.f_bavail // 1048576
    hd3x = hd3.f_bsize * hd3.f_bavail // 1048576
    zeile1 = time.asctime()
    zeile2 = os.popen("hostname -I").readline()[:-2]
    lcd_anzeige(zeile1, zeile2)
```

```
time.sleep(pause)
zeile1 = "S:" + str(sd1x) + "MB "
zeile1 += "2:" + str(hd2x) + "MB"
zeile2 = "1:" + str(hd1x) + "MB "
zeile2 += "3:" + str(hd3x) + "MB"
lcd_anzeige(zeile1, zeile2)
time.sleep(pause)
```

Am Anfang jedes Schleifendurchlaufs wird der freie Speicherplatz auf der SD-Karte und den angeschlossenen USB-Sticks ermittelt.

```
sd1 = os.statvfs(,/‘)
hd1 = os.statvfs(HD_1)
hd2 = os.statvfs(HD_2)
hd3 = os.statvfs(HD_3)
```

Das Statistikmodul `os.statvfs()` aus der `os`-Bibliothek liefert diverse statistische Informationen zum Dateisystem, die hier für das Hauptverzeichnis bei jedem Schleifendurchlauf aufs Neue als Objekt in die Variable `sd1` geschrieben werden. Die Speicheranzeige für das Hauptverzeichnis ermittelt nicht den Speicherplatz auf gemounteten externen Datenträgern.

Auf die gleiche Weise werden die Daten der USB-Sticks ermittelt und in die Variablen `hd1`, `hd2` und `hd3` geschrieben.

Die Methode `sd1.f_bsize` liefert die Größe eines Speicherblocks in Byte aus den in der Variablen `s` gespeicherten Daten des Dateisystems. `sd1.f_bavail` gibt die Anzahl freier Blöcke an. Das Produkt aus beiden Werten gibt demnach die Anzahl freier Bytes an, die hier ganzzahlig mit dem Python 3-Operator `//` durch 1.048.576 geteilt wird, um die Anzahl freier Megabytes zu erhalten. Dieser Wert wird in der neuen Variablen `sd1x` gespeichert. Auf die gleiche Weise wird der freie Speicherplatz auf den USB-Sticks errechnet und in die Variablen `hd1x`, `hd2x` und `hd3x` geschrieben.

```
sd1x = sd1.f_bsize * sd1.f_bavail // 1048576
hd1x = hd1.f_bsize * hd1.f_bavail // 1048576
hd2x = hd2.f_bsize * hd2.f_bavail // 1048576
hd3x = hd3.f_bsize * hd3.f_bavail // 1048576
```

Danach laufen im Wechsel zwei Anzeigeblöcke ab, die die Daten in den beiden Zeilen des LCD-Moduls darstellen.

Im ersten Block zeigt das Display wie im Programm `11status.py` Datum, Uhrzeit und IP-Adresse. Danach wartet das Programm die am Anfang festgelegte Pause von zwei Sekunden.

```
zeile1 = time.asctime()
zeile2 = os.popen("hostname -I").readline()[:-2]
lcd_anzeige(zeile1, zeile2)
time.sleep(pause)
```

Der zweite Block zeigt den freien Speicherplatz auf der SD-Karte und den angeschlossenen USB-Sticks. Die Textblöcke werden aus den drei Zeichenketten `"S:"` bzw. `"2:"`, dem errechneten Wert und `"MB "` zusammengesetzt. Jeweils zwei der so zusammengebauten Texte werden mit dem Operator `+=` zu einer Zeichenkette zusammengehängt. Die obere Zeile zeigt die Werte `S:` für die SD-Karte und `2:` für den zweiten USB-Stick, die untere Zeile zeigt die Werte `1:` für den ersten USB-Stick und `3:` für den dritten USB-Stick.

```
zeile1 = "S:" + str(sd1x) + "MB "
zeile1 += "2:" + str(hd2x) + "MB"
zeile2 = "1:" + str(hd1x) + "MB "
zeile2 += "3:" + str(hd3x) + "MB"
lcd_anzeige(zeile1, zeile2)
time.sleep(pause)
```

Die bereits bekannte Funktion `lcd_anzeige()` schreibt die Daten auf das Display. Diese Anzeige bleibt wie gehabt zwei Sekunden lang stehen, danach werden wieder Uhrzeit und IP-Adresse angezeigt. Haben Sie weniger als drei USB-Sticks angeschlossen, entfernen Sie einfach die jeweiligen Programmzeilen.

Lassen Sie das Programm laufen. Wenn Sie große Dateien auf die Speicherkarte oder die USB-Sticks übertragen, werden Sie sehen, wie sich die Megabytezahlen bei jedem Anzeigezyklus ändern.

Interaktive Statusanzeige mit Tasten

15

Die Statusanzeige blättert bisher automatisch zwischen den beiden Seiten mit je zwei Anzeigezeilen. Zwei zusätzliche Taster machen die Anzeige interaktiv steuerbar. Mit der einen Taste blättert man zeilenweise nach oben, mit der anderen nach unten. So sind jeweils zwei Zeilen sichtbar, die auch automatisch aktualisiert werden.

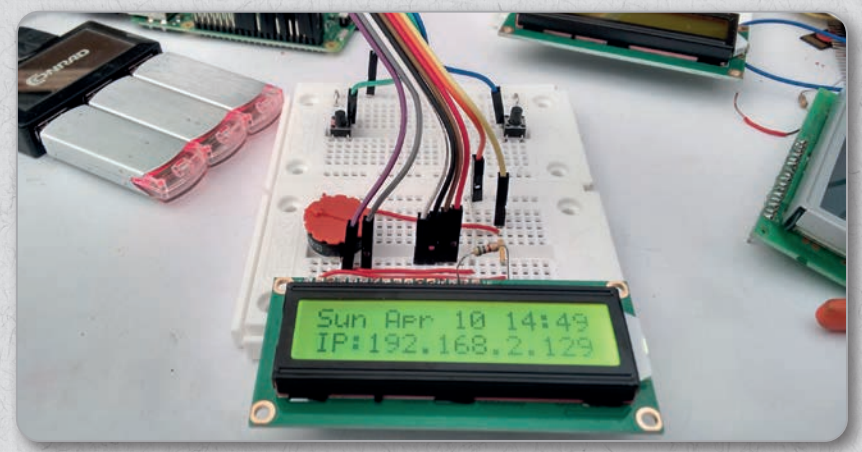


Abb. 15.1: *Interaktive Statusanzeige mit zwei Tastern zum Blättern.*

Bauen Sie auf einem zweiten Steckbrett zwei Taster wie abgebildet auf. Die beiden Taster sind so auf dem Steckbrett angeordnet, dass sie trotz des breiten Kabelstrangs der Verbindungskabel zum Raspberry Pi gut erreichbar sind.

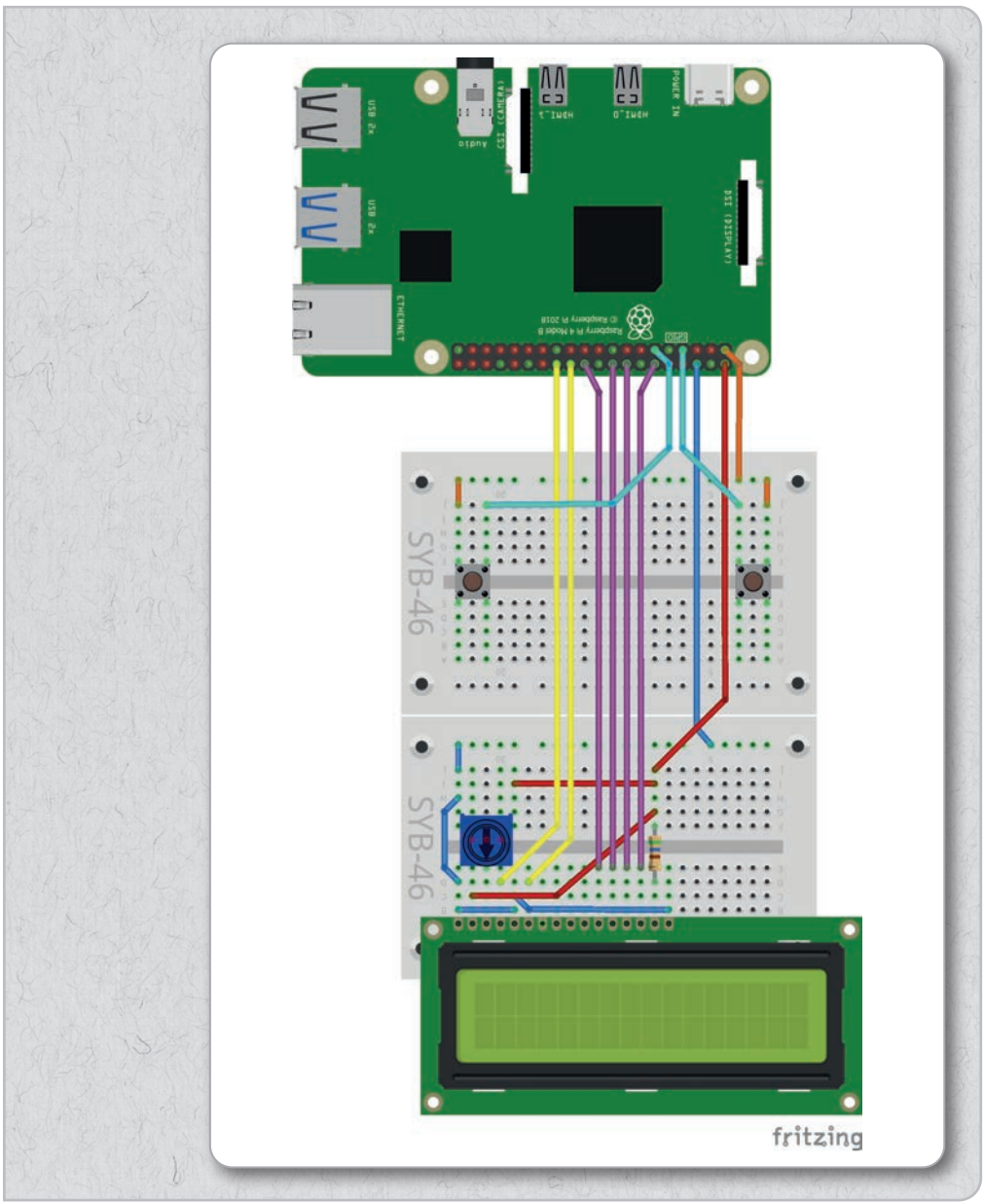


Abb. 15.2: *LC-Display mit zwei Tastern auf zwei Steckplatinen.*

Benötigte Bauteile

- 2 x Steckplatine,
- 1 x LC-Display,
  1 x 560-Ohm-Widerstand (Grün-Blau-Braun),
- 1 x Potenziometer,
  2 x Taster,
- 11 x Verbindungskabel,
- 8 x Drahtbrücke (unterschiedliche Längen)

Der Schaltungsaufbau entspricht weitgehend den vorhergehenden Experimenten mit dem LC-Display. Es werden auch die gleichen GPIO-Ports verwendet. Die beiden cyanfarben dargestellten Leitungen verbinden die Taster mit den noch freien GPIO-Ports 4 und 17.

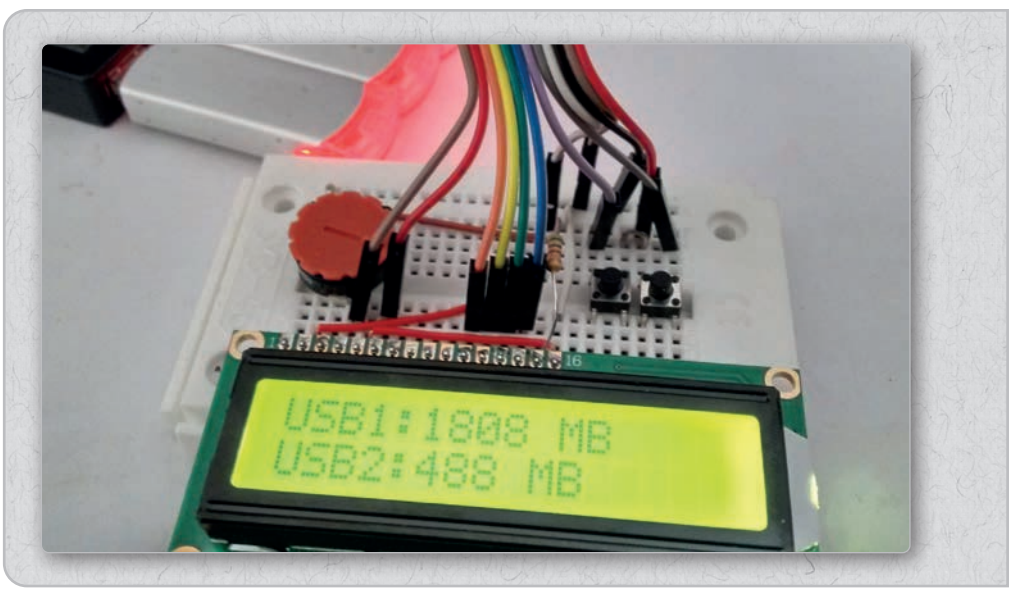


Abb. 15.3: *LC-Display mit zwei Tastern auf einer Steckplatine.*

Wer es etwas handlicher haben möchte, kann die beiden Taster auch noch auf der gleichen Steckplatine wie die übrigen Bauteile unterbringen, wenn man alles ein wenig enger aufbaut.

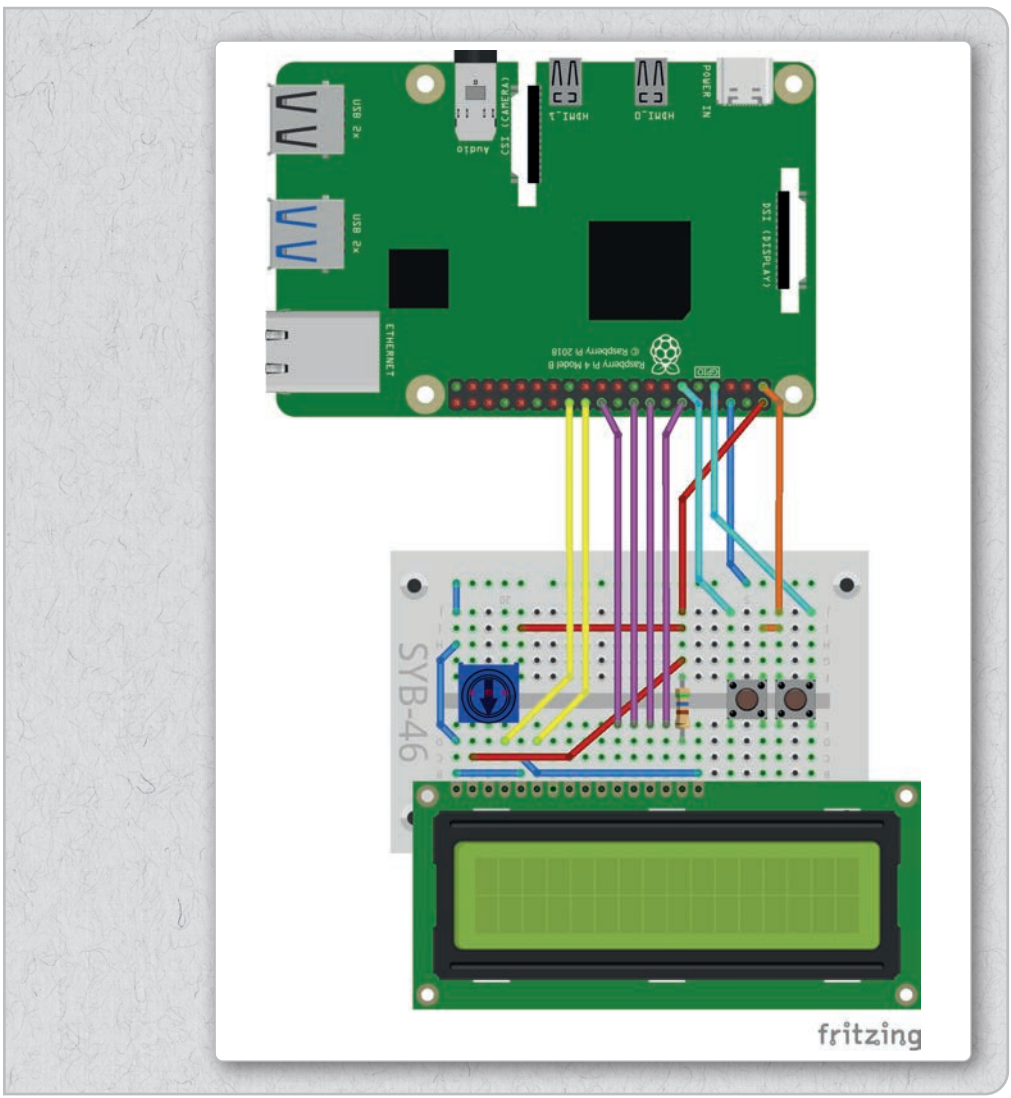


Abb. 15.4: *LC-Display mit zwei Tastern auf einer Steckplatine.*

Zwei verschiedene Stromversorgungen

Achten Sie beim Aufbau besonders genau auf den korrekten Anschluss der Stromversorgung, da die Schaltung beide Stromversorgungsleitungen des Raspberry Pi nutzt, +3,3 V für die GPIO-Ports der Taster und +5 V für das LC-Display. Ein +5-V-Signal darf keinesfalls zurück auf einen GPIO-Eingang gelangen.

15.1 | So funktioniert es

Das Programm `15status_sdusb.py` basiert auf dem Programm `14status_sdusb.py` und ist um die Steuerung durch die beiden Taster erweitert. Um die Anzeige übersichtlicher zu gestalten, da jetzt mehr Platz vorhanden ist, zeigt jede Zeile nur noch den freien Speicherplatz eines USB-Sticks. So ergeben sich auch keine Schwierigkeiten, falls auf einem USB-Stick mal 1.000 MB oder mehr Speicherplatz frei sind.

```
T1 = 4
T2 = 17
```

Dazu werden am Anfang im Definitionsbereich zwei weitere LED-Ports für die beiden Taster definiert …

```
GPIO.setup(T1, GPIO.IN, GPIO.PUD_DOWN)
GPIO.setup(T2, GPIO.IN, GPIO.PUD_DOWN)
```

… und später als Eingänge mit internen Pull-down-Widerständen initialisiert.

```
pause = 0.1
```

Die Pause wird auf 0,1 Sekunden verkürzt, da sie diesmal das Abfrageintervall der Taster festlegt. Bei zu langen Pausen müsste der Benutzer sehr lange auf die Taster drücken, damit der Tastendruck vom Programm registriert wird.

```
z1 = 0
zeile = ["", "", "", "", "", ""]
```

Jetzt werden noch eine Variable `n` und ein Array `z` mit sechs leeren Zeichenketten definiert, die später für die Auswahl und die Darstellung der insgesamt sechs verschiedenen Anzeigezeilen benötigt werden. In der Hauptschleife sind die wichtigsten Unterschiede gegenüber früheren Programmversionen zu finden.

```
try:
  while True:
    sd1 = os.statvfs(,/‘)
    hd1 = os.statvfs(HD_1)
    hd2 = os.statvfs(HD_2)
    hd3 = os.statvfs(HD_3)
    sd1x = sd1.f_bsize * sd1.f_bavail // 1048576
    hd1x = hd1.f_bsize * hd1.f_bavail // 1048576
    hd2x = hd2.f_bsize * hd2.f_bavail // 1048576
    hd3x = hd3.f_bsize * hd3.f_bavail // 1048576
    z[0] = time.asctime()
    z[1] = os.popen("hostname -I").readline()[:-2]
    z[2] = "SD  :" + str(sd1x) + " MB"
    z[3] = "USB1:" + str(hd1x) + " MB"
    z[4] = "USB2:" + str(hd2x) + " MB"
    z[5] = "USB3:" + str(hd3x) + " MB"
```

Die Texte für die sechs verschiedenen Anzeigezeilen werden aus den entsprechenden Werten zusammengesetzt und in den sechs Feldern des Arrays `z[]` abgelegt. Die Variable `n` gibt an, welche Zeilen dargestellt werden sollen. Am Anfang steht diese Variable auf `0`. Das bedeutet, `z[0]` und `z[1]` erscheinen in der Anzeige.

```
    if GPIO.input(T1)==1:
      n += 1
      if n > 4:
        n = 4
```

Drückt der Benutzer die Taste 1, wird `n` um 1 erhöht, die Anzeige blättert einen Schritt nach unten und zeigt jetzt `z[1]` und `z[2]`. Erreicht `n` durch mehrfaches Drücken der Taste den Wert 4, werden `z[4]` und `z[5]` angezeigt. Höhere Werte sind nicht möglich, da keine weiteren Textzeilen definiert sind. Deshalb wird `n` automatisch wieder auf `4` gesetzt, sollte der Benutzer ein weiteres Mal die Taste 1 drücken.

```
if GPIO.input(T2)==1:
  n -= 1
  if n < 0:
    n = 0
```

Nach dem gleichen Prinzip blättert die andere Taste nach oben, indem der Wert `n` bei jedem Tastendruck um 1 reduziert wird. Würde der Wert kleiner als 0, würde er automatisch auf `0` gesetzt. In diesem Fall werden `z[0]` und `z[1]` angezeigt.

```
lcd_anzeige(z[n], z[n + 1])
time.sleep(pause)
```

Am Ende jedes Schleifendurchlaufs werden die in `n` definierte und die folgende Zeile auf dem LC-Display angezeigt. Nach einer kurzen Wartezeit startet die Schleife neu, aktualisiert die Anzeige und prüft, ob der Benutzer eine Taste gedrückt hat.

16 Lauflicht mit dem Portexpander

Lauflichter sind immer wieder beliebte Effekte, um Aufmerksamkeit zu erregen, sei es im Partykeller oder in professioneller Leuchtwerbung. Mit dem Raspberry Pi und ein paar LEDs lässt sich so etwas leicht realisieren. Das Problem ist, dass die GPIO-Ports schnell knapp werden.

16.1 | Der Portexpander MCP23017

Im Paket befindet sich ein integrierter Schaltkreis mit der Bezeichnung MCP23017. Dabei handelt es sich um einen Portexpander, der 16 programmierbare GPIO-Ports liefert, selbst aber nur über zwei Steuerleitungen vom Raspberry Pi angesteuert wird.

Der MCP23017-Chip hat auf jeder Seite 14 Anschlüsse, von denen auf jeder Seite acht GPIO-Ports sind. Der Rest sind Steuerleitungen und Stromversorgung.

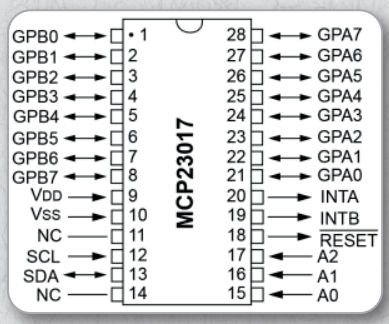


Abb. 16.1: *Pinbelegung des MCP23017.*

In der Linksammlung `www.softwarehandbuch.de/raspberry-pi` finden Sie das Datenblatt zu diesem Chip.

Kerbe beachten

Der MCP23017 hat wie alle integrierten Schaltkreise dieser Bauform an einer Schmalseite eine Kerbe. Diese dient der Orientierung, um den Baustein in der richtigen Richtung einzubauen. Wenn Sie die Schaltungen auf den Steckplatinen entsprechend den Abbildungen in diesem Buch nachbauen, ist die Kerbe immer rechts.

Die Anschlüsse `GPA0...GPA7` und `GPB0...GPB7` sind zwei je 8 Bit breite GPIO-Schnittstellen `GPIOA` und `GPIOB`, die binärcodiert angesteuert werden können. Das bedeutet, man steuert nicht jeden Port einzeln an, sondern schreibt eine 8 Bit lange Zahl im Ganzen auf die `GPIOA`- oder `GPIOB`-Schnittstelle.

Die Anschlüsse `VDD` und `VSS` sind die Stromversorgung für den Portexpander. `VDD` muss an +3,3 V angeschlossen werden, `VSS` an 0 V.

Der `RESET`-Anschluss muss im Normalbetrieb auf `True` stehen, also mit +3,3 V verbunden sein.

Die Anschlüsse `A0`, `A1`, `A2` legen die Adresse des Portexpanders fest. Mit den drei Leitungen lassen sich binär die Zahlen von 0 bis 7 darstellen. Ist eine Leitung mit 0 V verbunden, hat sie den Wert 0, bei der Verbindung mit +3,3 V hat sie den Wert 1. Dazu wird `0x20` addiert, sodass sich die i2c-Adressen `0x20...0x27` verwenden lassen.

16.2 | Das i2c-Protokoll

i2c (auch als I^2C, »I-Quadrat-C« = »Inter Chip Communication« bezeichnet) ist ein standardisiertes serielles Kommunikationsprotokoll, mit dem Elektronikchips über eine Zweidrahtverbindung Daten untereinander austauschen können. Auf diesem sehr einfachen Datenbus braucht jedes Gerät

eine eindeutige Adresse, damit die Daten entsprechend den Geräten zugeordnet werden können. Der Raspberry Pi unterstützt das i2c-Protokoll, es muss aber erst aktiviert werden.

Rufen Sie über den Menüpunkt *Einstellungen* das Programm *Raspberry Pi Konfiguration* auf und setzen Sie dort auf der Registerkarte *Schnittstellen* den Schalter *I2C* auf *Aktiviert*.

Abb. 16.2: *i2c in der Raspberry Pi-Konfiguration aktivieren.*

Starten Sie danach den Raspberry Pi neu, um i2c zu initialisieren. Geben Sie nach dem Neustart in einem LXTerminal-Fenster folgenden Befehl ein, um die i2c-Unterstützung zu testen:

```
i2cdetect -y 1
```

Wenn die i2c-Unterstützung aktiv ist, erscheint eine Tabelle der i2c-Geräte am Bus, die am Anfang leer ist, solange kein Gerät angeschlossen ist.

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Erhalten Sie eine Fehlermeldung, ist das i2c-Protokoll auf dem Raspberry Pi nicht korrekt aktiviert. Installieren Sie in diesem Fall zwei Pakete nach:

```
sudo apt-get update
sudo apt-get install i2c-tools python-smbus
```

16.3 | LEDs am Portexpander

Das erste Experiment mit i2c und dem Portexpander lässt acht LEDs als Lauflicht nacheinander aufleuchten. Bauen Sie die Schaltung wie in der Abbildung auf.

Benötigte Bauteile

- 2 x Steckplatine,
- 1 x Portexpander MCP23017,
- 2 x LED rot,
- 2 x LED orange,
- 2 x LED gelb,
- 2 x LED grün,
- 4 x Verbindungskabel,
- 15 x Drahtbrücke (unterschiedliche Längen)

Achten Sie darauf, den MCP23017 mit der Kerbe nach rechts einzubauen. Die farbigen Drahtbrücken verbinden die farblich passenden LEDs mit den acht Anschlüssen der `GPIOA`-Schnittstelle. Alle blau dargestellten Drahtbrücken sind mit 0 V verbunden, die roten mit +3,3 V – hier neben der Stromversorgung auch der `RESET`-Pin.

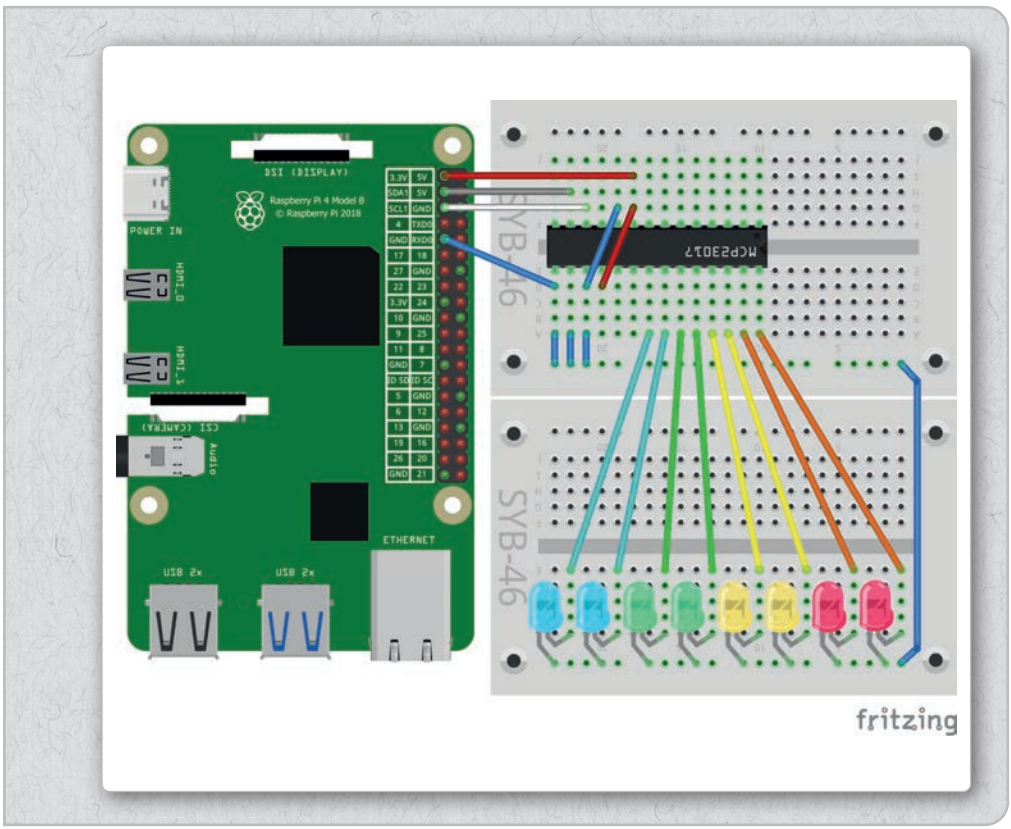


Abb. 16.3: *Lauflicht mit MCP23017-Portexpander. (15i2c_lauflicht_Steckplatine.png)*

Die drei kurzen blauen Drahtbrücken verbinden die drei Adressleitungen mit 0 V und geben dem Portexpander damit die i2c-Adresse `0x20`. Lassen Sie, nachdem Sie die Schaltung mit dem Raspberry Pi verbunden haben, noch einmal `i2cdetect` laufen.

```
i2cdetect -y 1
```

Jetzt erscheint das neue Gerät mit der Adresse `0x20` in der i2c-Geräteliste.

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Das Programm `16i2c_01.py` lässt die LEDs als Lauflicht leuchten.

```
#!/usr/bin/python
import time, smbus

DEVICE = 0x20
IODIRA = 0x00
GPIOA  = 0x12

bus = smbus.SMBus(1)
bus.write_byte_data(DEVICE,IODIRA,0x00)
bus.write_byte_data(DEVICE,GPIOA,0x00)

while True:
  j = 1
  for i in range(8):
    bus.write_byte_data(DEVICE,GPIOA,j)
    j *= 2
    time.sleep(0.1)
```

16.3.1 | So funktioniert es

Programme, die ausschließlich i2c nutzen, brauchen die GPIO-Schnittstelle nicht zu initialisieren. Nicht einmal die `RPi.GPIO`-Bibliothek wird benötigt.

```
import time, smbus
```

Für i2c muss die Bibliothek `smbus` importiert werden. Die Bibliothek `time` dient der Berechnung der Blinkdauer. Anschließend werden drei Hardwareadressen als Konstanten definiert.

`DEVICE = 0x20`	i2c-Geräteadresse des Portexpanders.
`IODIRA = 0x00`	8-Bit-Register, das die Richtung (Ausgang/Eingang) der GPIOA-Ports festlegt.
`GPIOA = 0x12`	8-Bit-Datenregister der GPIOA-Ports.

```
bus = smbus.SMBus(1)
```

Diese Zeile definiert ein Objekt namens `bus` vom Typ `smbus.SMBus()`. Darüber wird der i2c-Bus mit allen angeschlossenen Geräten angesprochen.

Die Methode `bus.write_byte_data()` schreibt ein Byte auf den i2c-Bus. Alle 8 Bits eines der beiden GPIO-Ports müssen immer gleichzeitig angesteuert werden. Der i2c-Bus verarbeitet immer ganze Bytes aus 8 Bits. Ein Byte kann binär, dezimal oder hexadezimal angegeben werden.

```
bus.write_byte_data(DEVICE,IODIRA,0x00)
```

Diese Zeile schreibt auf das in der Konstanten `DEVICE` definierte Gerät (hier das einzige vorhandene i2c-Gerät) in alle Bits des Registers `IODIRA` den Wert `0`. Damit werden alle acht `GPIOA`-Ports als Ausgänge definiert. Steht ein Bit auf `1`, würde der entsprechende Port als Eingang definiert.

```
bus.write_byte_data(DEVICE,GPIOA,0x00)
```

Diese Zeile schreibt auf das gleiche Gerät in alle Bits des Datenregisters `GPIOA` den Wert `0`. Damit werden alle acht `GPIOA`-Ports auf `0` gesetzt, die angeschlossenen LEDs also ausgeschaltet.

Für das Lauflicht werden jetzt nacheinander die acht LEDs einzeln eingeschaltet. Dazu muss das `GPIOA`-Datenregister nacheinander genau die Zahlenwerte annehmen, bei denen ein einzelnes Bit auf `1` steht.

LED LEUCHTET	BINÄR	DEZIMAL	HEX
1	0000 0001	1	0x01
2	0000 0010	2	0x02
3	0000 0100	4	0x04
4	0000 1000	8	0x08
5	0001 0000	16	0x10
6	0010 0000	32	0x20
7	0100 0000	62	0x40
8	1000 0000	128	0x80

In der Tabelle ist leicht zu erkennen, dass der Dezimalwert, bei 1 beginnend, einfach bei jedem Schritt verdoppelt werden muss, um die passende Zahl zu erhalten.

```
while True:
```

Das Lauflicht wird in einer Endlosschleife wiederholt, bis der Benutzer das Programm mit Strg+C abbricht.

```
  j = 1
```

Am Anfang jedes Schleifendurchlaufs wird der Wert für das Datenregister auf `1` gesetzt, damit nur die erste LED leuchtet.

```
  for i in range(8):
```

Innerhalb der Endlosschleife läuft eine weitere Schleife je achtmal, wobei in jedem Durchlauf eine LED eingeschaltet wird.

```
bus.write_byte_data(DEVICE,GPIOA,j)
```

Dazu wird der aktuelle Wert `j` in das `GPIOA`-Datenregister geschrieben. Im ersten Durchlauf hat das erste Bit den Wert `1`, die anderen 7 Bit den Wert `0`.

```
j *= 2
```

Danach wird der Wert `j` verdoppelt und damit das nächsthöhere Bit auf `1` gesetzt.

```
time.sleep(0.1)
```

Dieser Zustand wird für 0,1 Sekunden beibehalten. Dann startet der nächste Schleifendurchlauf, und die nächste LED leuchtet für 0,1 Sekunden. Nach acht Durchläufen der inneren Schleife startet der nächste Durchlauf der äußeren Schleife, wobei der Wert `j` erneut mit `1` startet und wieder die erste LED der Reihe blinkt.

Das Programm kann jederzeit mit Strg+C oder dem roten Stoppsymbol in der Thonny Python IDE abgebrochen werden. Eine Bereinigung der GPIO-Ports ist nicht nötig, deshalb fehlt auch das `try…except`-Konstrukt im Programm.

Binäruhr

17

Binäre Uhren zeigen die Uhrzeit in binärcodierter Form statt in Ziffern an. Solche Uhren sehen einfach cool aus, obwohl oder gerade weil das Ablesen für Anfänger etwas gewöhnungsbedürftig ist.

Das Prinzip der Zeitdarstellung ist einfach. Jede LED steht für ein Bit der Binärzahl. Zur Darstellung der maximal zwölf Stunden werden vier LEDs benötigt (8, 4, 2, 1), zur Darstellung der maximal 59 Minuten sechs LEDs (32, 16, 8, 4, 2, 1). Jetzt braucht man nur noch die Werte der leuchtenden LEDs zu addieren, und man erhält die Werte für Stunden und Minuten. Die Zeit wird wie auf einer Analoguhr im 12-Stunden-Format angezeigt.

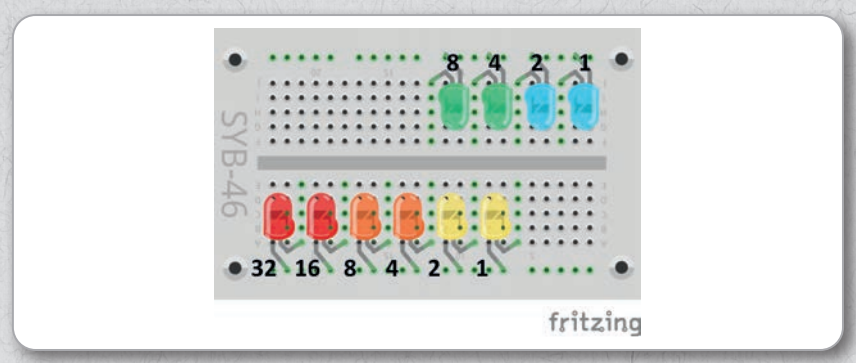


Abb. 17.1: *Aufbau einer Binäruhr mit zehn LEDs.*

In diesem Experiment bauen wir eine solche Binäruhr aus zehn LEDs auf, die über den Portexpander gesteuert werden.

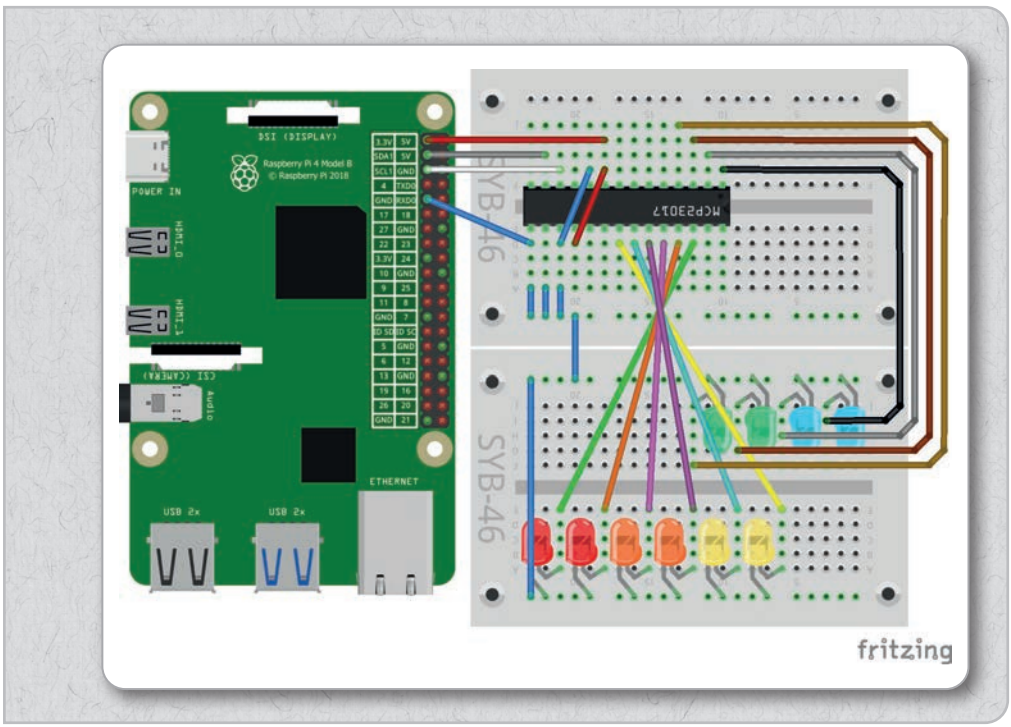


Abb. 17.2: *Binäruhr mit MCP23017-Portexpander.*

Benötigte Bauteile

- 2 x Steckplatine,
- 1 x Portexpander MCP23017,
- 2 x LED rot,
- 2 x LED orange,
- 2 x LED gelb,
- 2 x LED grün,
- 2 x LED blau,
- 4 x Verbindungskabel,
- 17 x Drahtbrücke (unterschiedliche Längen)

Die Anschlussdrähte der LEDs verlaufen diesmal nicht parallel zu den Ports des Portexpanders, sondern genau über Kreuz. Dies hat den Vorteil, dass die LED ganz rechts mit dem niedrigsten Bitwert auch mit dem Port mit dem niedrigsten Bitwert verbunden ist. Die weiteren LEDs folgen mit aufsteigenden Bitwerten, was die Programmierung deutlich vereinfacht. Die Tabelle zeigt, welche LED wo angeschlossen ist.

LED	PORT AM PORTEXPANDER	LED	PORT AM PORTEXPANDER
Stunde 8	GPIOB3	Minute 16	GPIOA4
Stunde 4	GPIOB2	Minute 8	GPIOA3
Stunde 2	GPIOB1	Minute 4	GPIOA2
Stunde 1	GPIOB0	Minute 2	GPIOA1
Minute 32	GPIOA5	Minute 1	GPIOA0

Die Anschlüsse für die Adressleitungen und den MCP23017 wurden unverändert aus dem letzten Experiment übernommen.

Das Programm `17binaeruhr.py` ist sehr einfach gehalten, da der Portexpander von sich aus bereits mit binären Daten arbeitet.

```
#!/usr/bin/python
import time, smbus

DEVICE = 0x20
IODIRA = 0x00
IODIRB = 0x01
GPIOA  = 0x12
GPIOB  = 0x13

bus = smbus.SMBus(1)
bus.write_byte_data(DEVICE,IODIRA,0x00)
bus.write_byte_data(DEVICE,IODIRB,0x00)
bus.write_byte_data(DEVICE,GPIOA,0x00)
bus.write_byte_data(DEVICE,GPIOB,0x00)

m1 = 60
```

```
while True:
  zeit = time.localtime()
  m = zeit.tm_min
  h = zeit.tm_hour
  if h > 12:
    h = h - 12
  if m1 <> m:
    bus.write_byte_data(DEVICE,GPIOB,h)
    bus.write_byte_data(DEVICE,GPIOA,m)
    m1 = m
  time.sleep(1)
```

17.1 | So funktioniert es

Die Initialisierung des i2c-Bus läuft weitgehend wie im vorherigen Experiment, allerdings mit dem Unterschied, dass diesmal beide GPIO-Ports des Portexpanders verwendet werden.

`DEVICE = 0x20`	i2c-Geräteadresse des Portexpanders.
`IODIRA = 0x00`	8-Bit-Register, das die Richtung (Ausgang/Eingang) der GPIOA-Ports festlegt.
`IODIRB = 0x01`	8-Bit-Register, das die Richtung (Ausgang/Eingang) der GPIOB-Ports festlegt.
`GPIOA = 0x12`	8-Bit-Datenregister der GPIOA-Ports.
`GPIOB = 0x13`	8-Bit-Datenregister der GPIOB-Ports.

Das Programm läuft in einer Endlosschleife, die regelmäßig die aktuelle Uhrzeit des Raspberry Pi ausliest. Unterscheidet sich die aktuelle Minute von der zuletzt auf den LEDs dargestellten Minute, werden andere LEDs eingeschaltet.

```
while True:
  zeit = time.localtime()
```

In jedem Durchlauf wird die aktuelle Zeit in das Objekt `zeit` geschrieben. Dazu wird die Funktion `time.localtime()` aus der `time`-Bibliothek

verwendet. Das Ergebnis ist eine Datenstruktur, die aus verschiedenen Einzelwerten besteht.

```
m = zeit.tm_min
h = zeit.tm_hour
```

Die beiden für die Digitaluhr relevanten Werte Minuten und Stunden werden aus der Struktur in die Variablen `m` und `h` geschrieben.

```
if h > 12:
  h = h - 12
```

Ist die Stundenangabe im 24-Stunden-Format größer als 12, wird 12 subtrahiert, um die Zeit im 12-Stunden-Format in der Variablen `h` zu speichern.

```
if m1 !=<> m:
  bus.write_byte_data(DEVICE,GPIOB,h)
  bus.write_byte_data(DEVICE,GPIOA,m)
  m1 = m
```

Hat die aktuelle Minute `m` einen anderen Wert als die zuletzt dargestellte Minute `m1`, wird die Stunde auf den Port `GPIOB` und die Minute auf den Port `GPIOA` geschrieben. Hier können direkt die Zahlenwerte verwendet werden, der Portexpander rechnet sie automatisch in Binärzahlen um und gibt sie entsprechend auf den Ports aus.

Danach wird `m1` auf die gerade dargestellte Minute gesetzt. Vor dem ersten Start der Schleife bekommt `m1` den Wert `60`, den die Minutenanzeige im laufenden Betrieb nie erreicht. Damit wird sichergestellt, dass unabhängig von der aktuellen Uhrzeit bereits im ersten Schleifendurchlauf eine neue Zeit ermittelt und über die LEDs angezeigt wird.

```
time.sleep(1)
```

Am Ende der Schleife wartet das Programm eine Sekunde. Damit wird verhindert, dass die Schleife in Intervallen von Sekundenbruchteilen immer wieder aufgerufen wird und damit den Prozessor so stark auslastet, dass der Raspberry Pi für nichts anderes als dieses Programm zu nutzen ist.

18 Binäruhr plus LCD-Uhr

Die Binäruhr benötigt auf dem Raspberry Pi nur die beiden GPIO-Ports 2 und 3 zur Ansteuerung des Portexpanders über den i2c-Bus. Alle anderen GPIO-Ports bleiben ungenutzt, auch die, die wir in früheren Experimenten zur Steuerung des LC-Displays verwendet haben. Daher lassen sich beide Schaltungen zu einer kombinieren und beide Programme zu einem, das die aktuelle Zeit parallel auf der Binäruhr und dem LC-Display zeigt. Für den Schaltungsaufbau benötigen Sie fast alle Bauteile aus dem Paket.

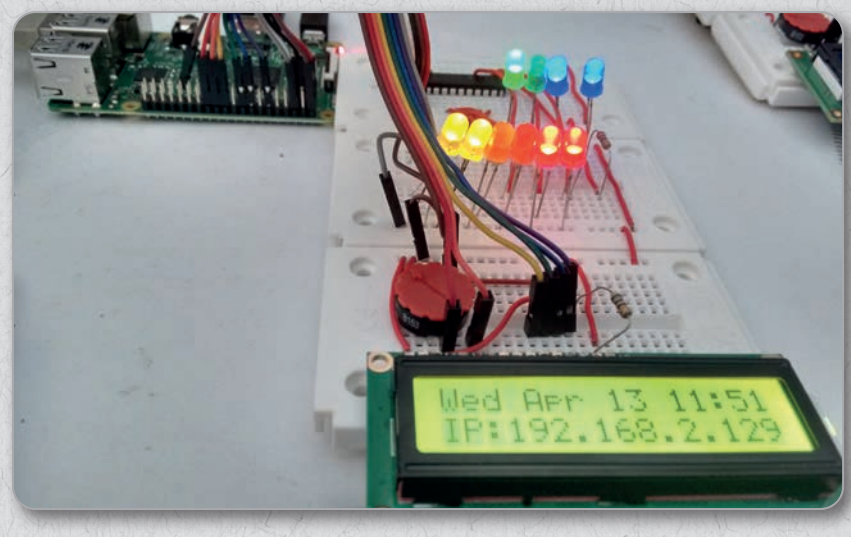


Abb. 18.1: *Binäruhr mit zusätzlicher LCD-Anzeige.*

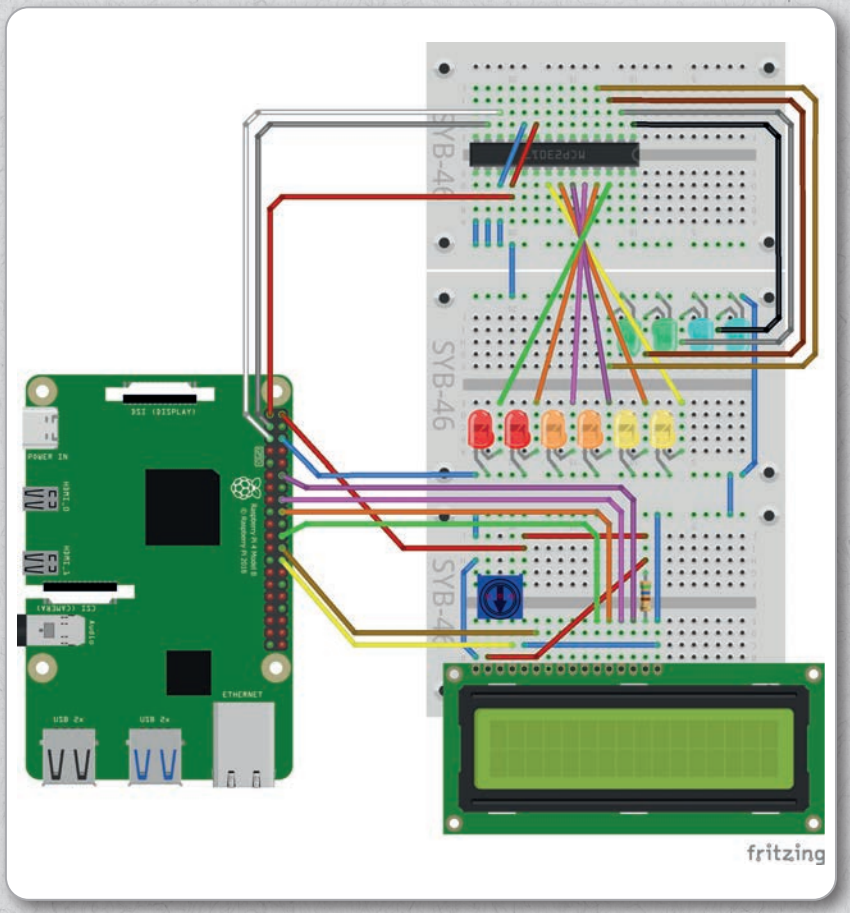


Abb. 18.2: *Binäruhr mit MCP23017-Portexpander und LCD-Uhr in einer Schaltung.*

Benötigte Bauteile

- 3 x Steckplatine,
- 1 x Portexpander MCP23017,
- 1 x LC-Display,
- 1 x 560-Ohm-Widerstand (Grün-Blau-Braun),
- 1 x Potenziometer,
- 2 x LED rot,
- 2 x LED orange,
- 2 x LED gelb, 2 x LED grün,
- 2 x LED blau,
- 11 x Verbindungskabel,
- 24 x Drahtbrücke (unterschiedliche Längen)

Zwei verschiedene Stromversorgungen

Achten Sie beim Aufbau besonders genau auf den korrekten Anschluss der Stromversorgung, da die Schaltung beide Stromversorgungsleitungen des Raspberry Pi nutzt, +3,3 V für den Portexpander und +5 V für das LC-Display.

18.1 | So funktioniert es

Das Programm `18uhr.py` kombiniert die beiden Programme der Binäruhr und der LCD-Uhr.

```
#!/usr/bin/python
import RPi.GPIO as GPIO
import time, os, smbus

LCD_RS = 7
LCD_E  = 8
LCD_D4 = 25
LCD_D5 = 24
LCD_D6 = 23
LCD_D7 = 18

GPIO.setmode(GPIO.BCM)
GPIO.setup(LCD_E,  GPIO.OUT)
GPIO.setup(LCD_RS, GPIO.OUT)
GPIO.setup(LCD_D4, GPIO.OUT)
GPIO.setup(LCD_D5, GPIO.OUT)
GPIO.setup(LCD_D6, GPIO.OUT)
GPIO.setup(LCD_D7, GPIO.OUT)

LCD_WIDTH = 16
LCD_LINE_1 = 0x80
LCD_LINE_2 = 0xC0
LCD_CHR = 1
LCD_CMD = 0
E_PULSE = 0.00005
```

```
E_DELAY = 0.00005
INIT  = 0.01
pause = 2

def lcd_enable():
    time.sleep(E_DELAY)
    GPIO.output(LCD_E, 1)
    time.sleep(E_PULSE)
    GPIO.output(LCD_E, 0)
    time.sleep(E_DELAY)

def lcd_byte(bits, mode):
    GPIO.output(LCD_RS, mode)
    GPIO.output(LCD_D4, bits&0x10==0x10)
    GPIO.output(LCD_D5, bits&0x20==0x20)
    GPIO.output(LCD_D6, bits&0x40==0x40)
    GPIO.output(LCD_D7, bits&0x80==0x80)
    lcd_enable()
    GPIO.output(LCD_D4, bits&0x01==0x01)
    GPIO.output(LCD_D5, bits&0x02==0x02)
    GPIO.output(LCD_D6, bits&0x04==0x04)
    GPIO.output(LCD_D7, bits&0x08==0x08)
    lcd_enable()

def lcd_string(message):
    message = message.ljust(LCD_WIDTH," ")
    for i in range(LCD_WIDTH):
        lcd_byte(ord(message[i]),LCD_CHR)

def lcd_anzeige(z1, z2):
    lcd_byte(LCD_LINE_1, LCD_CMD)
    lcd_string(z1)
    lcd_byte(LCD_LINE_2, LCD_CMD)
    lcd_string(z2)

LCD_INIT = [0x33, 0x33, 0x32, 0x28, 0x0C, 0x06, 0x01]
for i in LCD_INIT:
    lcd_byte(i,LCD_CMD)
    time.sleep(INIT)
```

```
DEVICE = 0x20
IODIRA = 0x00
IODIRB = 0x01
GPIOA  = 0x12
GPIOB  = 0x13

bus = smbus.SMBus(1)
bus.write_byte_data(DEVICE,IODIRA,0x00)
bus.write_byte_data(DEVICE,IODIRB,0x00)
bus.write_byte_data(DEVICE,GPIOA,0x00)
bus.write_byte_data(DEVICE,GPIOB,0x00)

m1 = 60

try:
    while True:
        zeit = time.localtime()
        m = zeit.tm_min
        h = zeit.tm_hour
        if h > 12:
            h = h - 12
        if m1 != m:
            bus.write_byte_data(DEVICE,GPIOB,h)
            bus.write_byte_data(DEVICE,GPIOA,m)
            m1 = m
        zeile1 = time.asctime()
        zeile2 = os.popen(„hostname -I").readline()[:-2]
        lcd_anzeige(zeile1, zeile2)
        time.sleep(pause)

except KeyboardInterrupt:
  GPIO.cleanup()
```

Das Programm enthält keine neuen Funktionen. Nach den Initialisierungen für das Display wird der i2c-Bus initialisiert. In der Hauptschleife folgen nacheinander die bekannten Routinen für die Zeitanzeige auf der Binäruhr und dem Display.