Python & MicroPython: Programmieren lernen für Einsteiger

Digitale Logik und Schaltungstechnik

Teil 3

Der Raspberry Pi findet als vielseitiger Einplatinencomputer in der Digitaltechnik eine Vielzahl von Anwendungen. Ein wichtiger Einsatzbereich ist unter anderem die Verwendung als kostengünstige Lernplattform. Die Digitaltechnik ist zur unverzichtbaren Grundlage der modernen Elektronik geworden und die Programmiersprache „Python" kann verwendet werden, um Grundlagen dieser zentralen Disziplin zu vermitteln. Auch wenn die klassische Digitaltechnik mit einzelnen Logikgattern oder Zähler-ICs zunehmend in den Hintergrund tritt – ohne solide Kenntnisse der digitalen Schaltungstechnik können weder Mikrocontroller noch FPGAs (Free Programmable Gate Arrays), EPLDs (Electronically Programmable Logic Devices), Prozessoren, Displays oder Speichersysteme entwickelt und aufgebaut werden.

Der Raspberry Pi eignet sich bestens dazu, digitale Schaltungen und Prototypen zu entwickeln und zu testen, da er über GPIO-Pins verfügt, die als digitale Ein- und Ausgänge dienen können. Dadurch ist er als Steuerungseinheit für verschiedene Sensoren und Aktoren in digitalen Schaltungen verwendbar. Zudem kann mit dem Raspberry Pi z. B. ein Smart-Home-System aufgebaut werden, in dem python-gesteuerte Hardware als zentrale Steuereinheit dient, um verschiedene IoT-Geräte wie beispielsweise digitale Sensoren für Temperatur, Feuchtigkeit, Bewegung und Lichtsteuerung auszuwerten.

In diesem Beitrag soll deshalb die Funktion der Pins als Eingänge genauer beschrieben werden. Dabei werden die folgenden Themen und Python-Programmstrukturen vorgestellt:

- Logische Funktionen in Python:
  AND, OR, NOT, NAND, NOR, EXOR
- Bitshift-Funktionen («, »)
- Binärzähler und Sieben-Segment-Displays

Logik mit Python

Für einfache Logikschaltungen wie UND, ODER, NICHT (engl. AND, OR und NOT) wurden über Jahrzehnte hinweg sogenannte TTL- oder CMOS-Bausteine eingesetzt. Für umfangreichere Aufgaben entstanden so regelrechte „TTL-Gräber“ (Bild 1), deren Name auf die regelmäßige Anordnung der digitalen ICs zurückgeht, die an Gräberfelder großer Friedhöfe erinnerte.

Mit dem Aufkommen programmierbarer Bausteine fand diese Ära ihr Ende. Heute können sogar die komplexesten Aufgaben von einem einzelnen Controller oder Mikroprozessor übernommen werden. Für die Programmierung mit Python stehen umfangreiche Logikfunktionen als Softwarevariante zur Verfügung, z. B.:

a	not a
True	False
False	True

a	b	a and b	a or b
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Auch die Bitoperatoren können in Python durch die folgenden Anweisungen dargestellt werden:

& bitweise AND-Verknüpfung
| bitweise OR- Verknüpfung
^ bitweise XOR-Verknüpfung
~ bitweises NOT

In Python wie auch in vielen anderen Programmiersprachen gibt es also zwei Arten von Operatoren: Standardoperatoren und bitweise Operatoren. Der Hauptunterschied zwischen ihnen liegt in der Art und Weise, wie sie Operanden verarbeiten.

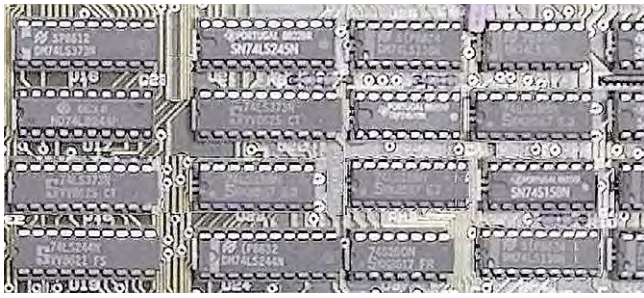

Bild 1: Klassisches „TTL-Grab“

Bitweise Operatoren führen Operationen auf der Bit-Ebene aus. Sie behandeln ihre Operanden als Bitsequenzen und führen alle Funktionen auf den entsprechenden Bits durch. Sie sind nützlich, um spezifische Manipulationen an den Bits von Ganzzahlen vorzunehmen, insbesondere in Bezug auf Flags, Masken und Portmanipulationen. Zunächst kann man sich mit den Operatoren über die Konsole vertraut machen. Das folgende Beispiel verdeutlicht den Unterschied.

So liefert die Eingabe

```
>>> a=4              # a = 0100 binär
>>> b=10             # b = 1010 binär
>>> print(bool(a and b))
```

den Wert „True“ als Ergebnis.

Dagegen liefert

```
>>> print(bool(a & b))
```

den Wert „False“. Die Standardverknüfung (a and b) liefert „True“, da beide Operanden ungleich null, also „True“ sind. Das bitweise AND (&) liefert „False“

```
        0100
&       1010
=       0000
```

da an keiner Binärstelle von a und b gleichzeitig eine Eins steht. Das Ergebnis ist also die Bitfolge 0000 und damit „False“.

Hier ist in der Programmierpraxis also große Aufmerksamkeit gefordert, da die beiden Verknüpfungen leicht verwechselt werden können.

Die Shift-Operatoren sind ebenfalls verfügbar:

```
<< Bits nach links verschieben
>> Bits nach rechts verschieben
```

Die folgenden Beispiele verdeutlichen wieder die Anwendung:

```
>>> a = 0b01010
>>> bin(a<<1)
```

liefert

```
'0b10100'
```

als Resultat. Die Bitfolge wurde also um eine Binärstelle nach links verschoben. Mathematisch entspricht dies einer Multiplikation mit 2:

```
>>> a = 0b1010
>>> a
10
>>> a<<1
20
```

LED-Logik

Über die IO-Pins können die Logik-Operatoren auch in der realen Welt eingesetzt werden. Das folgende Python-Programm emuliert ein AND-Gatter (AND_emulation.py):

```
from gpiozero import LED, Button
import time

LED1 = LED(2)                          # LED @ pin 2
button1 = Button(4, pull_up=False)     # button @ pin 3
button2 = Button(17, pull_up=False)    # button @ pin 4

print(button1); print(button2)

while True:
    if (button1.is_pressed & button2.is_pressed):
        LED1.on()
    else:
        LED1.off()
```

Die Taster müssen hierzu an die Pins 4 und 17 des Pi angeschlossen werden. Die LED bzw. das LED-Modul wird über Pin 2 angesteuert. Den Schaltplan dazu zeigt Bild 2.

Um den Eingängen in jedem Fall definierte Pegel zu geben, wurden jeweils 10-kΩ-Pull-down-Widerstände (siehe Teil 2 der Beitragsserie: GPIOs steuern die Welt) verwendet. Die LED erhält einen 470-Ω-Vorwiderstand, dieser kann bei Verwendung von PAD-Modulen entfallen.

Nach dem Start des Programms ist die LED zunächst aus bzw. es leuchtet die rote LED am Logic-Level-Modul, falls dieses verwendet wird (siehe Teil 2 der Beitragsserie: GPIOs steuern die Welt). Wird entweder Taster 1 oder Taster 2 gedrückt, ändert sich nichts. Erst wenn Taster 1 und Taster 2 gleichzeitig betätigt werden, geht die LED an bzw. das Logic-Level-Modul ändert seinen Port-Zustand von rot auf grün.

Hinweis: Der Pin GPIO3 kann hier nicht als Eingang verwendet werden, da er intern fest mit einem Pull-up-Widerstand verdrahtet ist. Deshalb werden die Ports 4 und 17 eingesetzt.

Bitshift in Aktion

Um die etwas abstrakten Bitshift-Operationen praktisch anzuwenden, kann das folgende Programm verwendet werden (BitshiftChaser.py). Zur Demonstration der Shift-Operation wird hier vorgegriffen. Die im Programm verwendeten Zeichenkettenoperationen usw. werden jedoch in späteren Beiträgen ausführlich erläutert.

```
from gpiozero import LED
from time import sleep

led_pins = [2,3,4,17,27,22,10,9]
leds = [LED(pin) for pin in led_pins]

n=1

while True:
    bit_pattern = str(bin(n)[2:])
    bit_pattern = ".join(reversed(bit_pattern))
    print(bit_pattern)
    for i in range(len(bit_pattern)):
       if bit_pattern[i] == '1':
           leds[i].on()
       else:
           leds[i].off()

    n=n<<1     # bitshift left

    sleep(.1)

    if n>=255:
        for i in range(len(bit_pattern)):
             leds[i].off()
        sleep(.1)
        n=1
```

Das Programm sorgt dafür, dass ein Lichtpunkt eine Reihe von LEDs von links nach rechts durchläuft. Nach dem Einbinden der Bibliotheken und dem Initialisieren der verwendeten LED-Pins wird die Zählervariable „n“ auf 1 gesetzt. In der folgenden Endlosschleife wird nun ein passendes Bitmuster erzeugt:

```
bit_pattern = str(bin(n)[2:])
```

Dies liefert eine Binärzahl (bin()) als Zeichenmuster, beispielsweise für n = 5 die Zeichenfolge:

```
n = 5      bit_pattern = 101
```

Die Anweisung [2:] sorgt dafür, dass das Binärzahlenpräfix (0b...) entfernt wird. Mit

```
bit_pattern = ''.join(reversed(bit_pattern))
```

wird die Reihenfolge des Bitmusters umgedreht, damit die Bit-Reihenfolge mit den LEDs übereinstimmt. Die folgende „for“-Schleife sorgt dafür, dass immer genau die LED eingeschaltet wird, an deren Stelle eine „1“ im Bitmuster steht. Dann folgt die entscheidende Anweisung

```
n = n << 1
```

in welcher der eigentliche Bitshift ausgeführt wird. Die verbleibenden Programmzeilen sorgen dann nur noch dafür, dass beim Erreichen der letzten LED wieder von vorne begonnen wird. Um das Programm zu testen, müssen nun lediglich 8 LEDs (oder das Level-Modul) an die Ports

2, 3, 4, 17, 27, 22, 10 und 9

angeschlossen werden (Bild 3).

Parallel dazu kann das Bitmuster in der Thonny-Shell beobachtet werden (Bild 4).

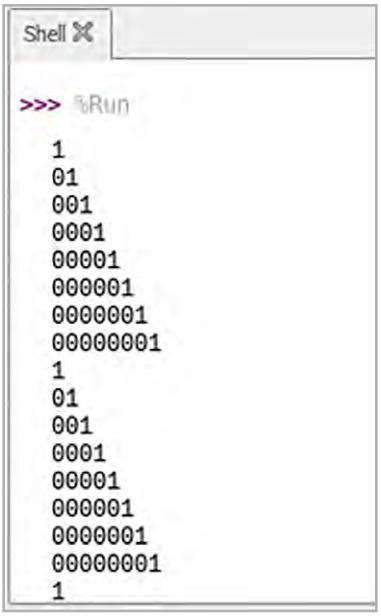

Bild 4: Ausgabe Bitshift-Funktion in der Shell

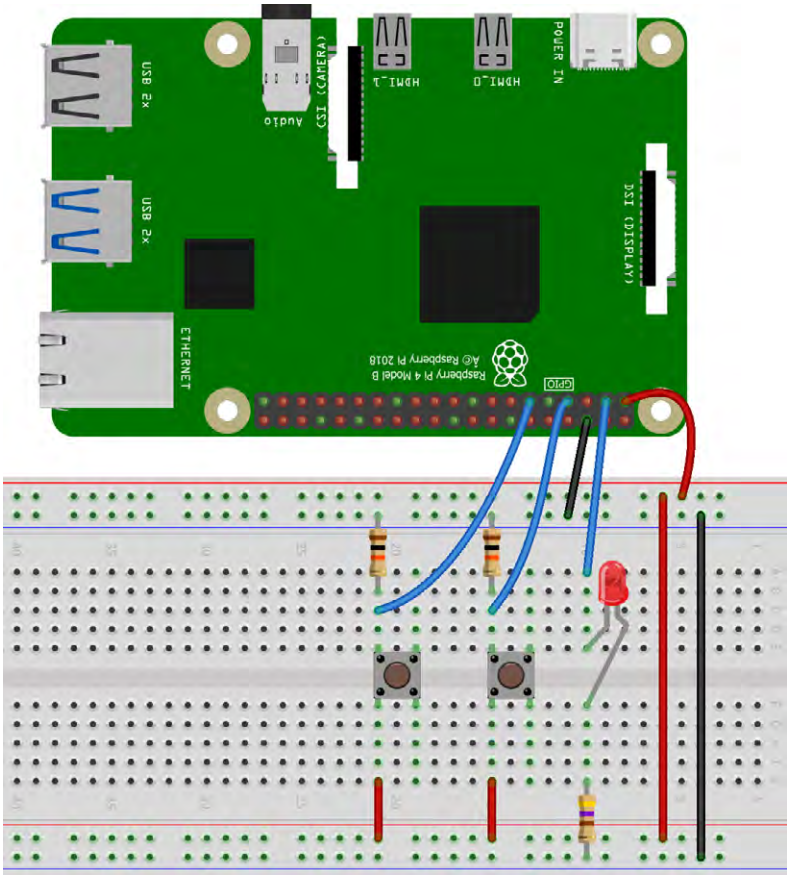

Bild 2: Schaltbild einer Gatterfunktion mit dem Raspberry Pi

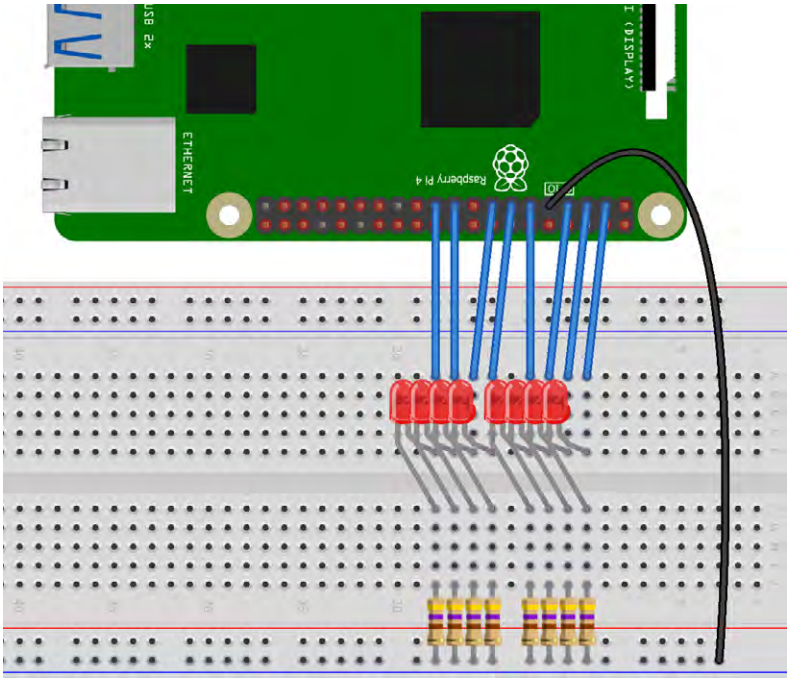

Bild 3: Schaltbild zum Aufbau eines Lauflichts

Logik-Analyzer in Python

Dass mit Python nicht nur veraltete Hardware ersetzt werden kann, sondern auch andere komplexere und nützliche Anwendungen möglich sind, soll in diesem Abschnitt demonstriert werden. Mit dem folgenden Aufbau entsteht beispielsweise ein Logik Analyzer. Er ist in der Lage, die Wahrheitstabelle eines beliebigen Gatters aufzuzeichnen. Hierfür müssen lediglich zwei IO-Pins als Ausgänge und ein Pin als Eingang konfiguriert werden. Das folgende Programm zeigt eine einfache Version eines solchen Logik-Analyzers (LogicAnalyzer.py):

```
import RPi.GPIO as GPIO
import time

output1_pin=2
output2_pin=3
input_pin=4

GPIO.setmode(GPIO.BCM)
GPIO.setup(output1_pin, GPIO.OUT)
GPIO.setup(output2_pin, GPIO.OUT)
GPIO.setup(input_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)

while True:
 print("A B Out")
 # 00
 GPIO.output(output1_pin, GPIO.LOW)
 GPIO.output(output2_pin, GPIO.LOW)
 print(0,0, GPIO.input(input_pin))
 time.sleep(1)

 # 01
 GPIO.output(output1_pin, GPIO.LOW)
 GPIO.output(output2_pin, GPIO.HIGH)
 print(0,1, GPIO.input(input_pin))
 time.sleep(1)

 # 10
 GPIO.output(output1_pin, GPIO.HIGH)
 GPIO.output(output2_pin, GPIO.LOW)
 print(1,0, GPIO.input(input_pin))
 time.sleep(1)

 # 11
 GPIO.output(output1_pin, GPIO.HIGH)
 GPIO.output(output2_pin, GPIO.HIGH)
 print(1,1, GPIO.input(input_pin))
 time.sleep(3)

 print()
```

Hier wird wieder die RPi.GPIO importiert, um auf die GPIO-Pins des Raspberry Pi zugreifen zu können. Es werden drei Pins konfiguriert:

- output1_pin (Pin 2) als Ausgang
- output2_pin (Pin 3) als Ausgang
- input_pin (Pin 4) als Eingang mit internem Pull-Up-Widerstand

In der Endlosschleife (while True) werden die folgenden Aktionen wiederholt:

1. **"A B Out"** wird ausgegeben, um anzuzeigen, welche Kombinationen von output1_pin und output2_pin derzeit aktiv sind.
2. Die GPIO-Pins werden in allen möglichen Kombinationen angesteuert:
   "00": output1_pin und output2_pin werden auf LOW gesetzt, um die Kombination 00 zu repräsentieren. Dann wird der Zustand des input_pin (Pin 4) ausgegeben.
   "01": output1_pin wird auf LOW und output2_pin auf HIGH gesetzt, um die Kombination 01 zu repräsentieren. Dann wird der Zustand des Input-Pins ausgegeben.
   "10": output1_pin wird auf HIGH und output2_pin auf LOW gesetzt, um die Kombination 10 zu repräsentieren. Der Zustand wird wieder ausgegeben.
   "11": output1_pin und output2_pin werden auf HIGH gesetzt, um die Kombination 11 zu repräsentieren. Es folgt wieder die Ausgabe des Zustands.

Nach jeder Ausgabe wird eine Pause von einer Sekunde bzw. drei Sekunden nach der Kombination „11" eingelegt, um die Ausgaben lesbar zu halten.

Bild 5 zeigt die Schaltung unter Verwendung eines PAD-Moduls (AND-Gatter).

Hinweis: Bitte genau auf den korrekten Aufbau achten. Wenn durch fehlerhafte Verdrahtung z. B. zwei Ausgänge verbunden werden, kann dies zu Hardwaredefekten führen. Im Zweifelsfall können sicherheitshalber vor alle Raspberry-Pi-Eingänge 1-kΩ-Widerstände eingefügt werden.

Wenn damit ein UND-Gatter analysiert wird, sieht das Ergebnis so aus wie in Bild 6. Ein Vergleich mit den darin dargestellten Logiktabellen bestätigt die korrekte Funktion.

Graphische Darstellung von Logiktabellen: Time Charts

In Datenblättern und in der technischen Literatur werden die Pegelverläufe von Logikschaltungen häufig grafisch in ihrem zeitlichen Verlauf („time charts") dargestellt. Bild 7 zeigt ein entsprechendes Beispiel.

Mit kleineren Modifikationen kann auch der Logik-Analyzer aus dem letzten Abschnitt Time Charts liefern (Bild 8).

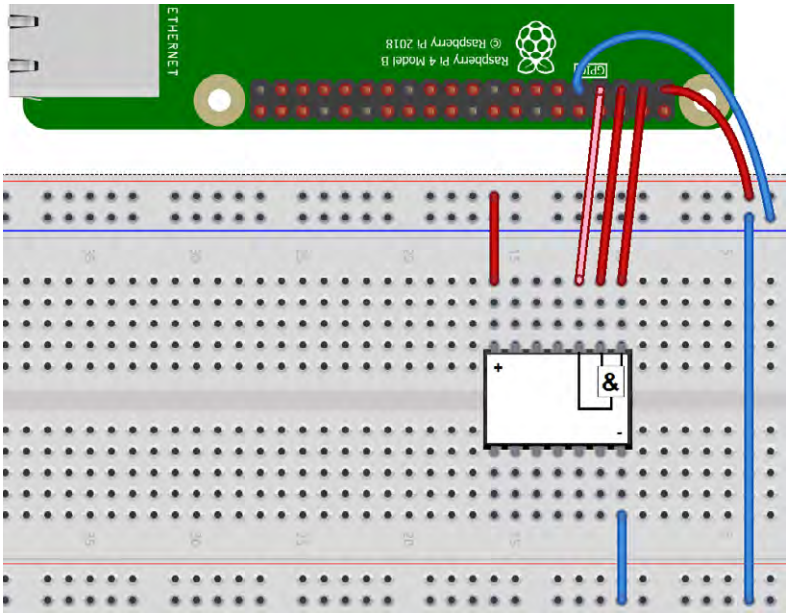

Bild 5: Schaltung zum Logik-Analyzer mit PAD-AND-Gate-Modul

Bild 6: UND-Gatter im Logik-Analyzer

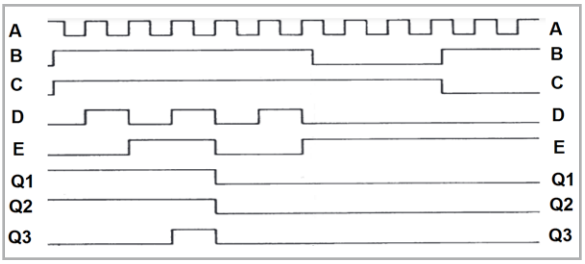

Bild 7: Time Chart einer digitalen Schaltung

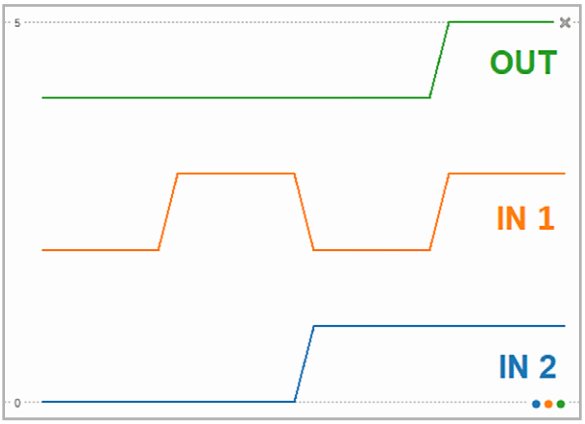

Bild 8: Time Chart einer AND-Funktion in der Thonny-Shell

Der Code dazu (Logic-Analyzer_graphical.py) sieht wie folgt aus:

```
import RPi.GPIO as GPIO
import time

output1_pin=2
output2_pin=3
input_pin=4
length=7

# available pins: 2,3,4,17,27,22,10,9
GPIO.setwarnings(False)
GPIO.setmode(GPIO.BCM)
GPIO.setup(output1_pin, GPIO.OUT)
GPIO.setup(output2_pin, GPIO.OUT)
GPIO.setup(input_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)

# 00
for n in range(length):
    GPIO.output(output1_pin, GPIO.LOW)
    GPIO.output(output2_pin, GPIO.LOW)
    print(0, 2, 4+GPIO.input(input_pin))
# 01
for n in range(length):
    GPIO.output(output1_pin, GPIO.LOW)
    GPIO.output(output2_pin, GPIO.HIGH)
    print(0, 3, 4+GPIO.input(input_pin))
# 10
for n in range(length):
    GPIO.output(output1_pin, GPIO.HIGH)
    GPIO.output(output2_pin, GPIO.LOW)
    print(1, 2, 4+GPIO.input(input_pin))
# 11
for n in range(length):
    GPIO.output(output1_pin, GPIO.HIGH)
    GPIO.output(output2_pin, GPIO.HIGH)
    print(1, 3, 4+GPIO.input(input_pin))
```

Im Vergleich zur numerischen Ausgabe wurde hier dafür gesorgt, dass keine störenden Zeichen ausgegeben werden. Die einzelnen Grafikkanäle wurden zudem durch entsprechende Offsetzahlen voneinander getrennt. Weitere Informationen und Details zu grafischen Ausgaben folgen in späteren Beiträgen.

Binärzähler

Neben den Logikfunktionen spielen vor allem Zähler und Timer eine wichtige Rolle in der Digitaltechnik. Als nächste Stufe soll daher ein Digitalzähler in Python realisiert werden. Diese Version ersetzt die früher übliche Kaskadierung von Flip-Flops und kann daher den Hardware-Aufwand in digitalen Systemen erheblich reduzieren.

Das folgende Programm (BinaryCounter.py) zählt an den Ports 2, 3, 4, 17, 27, 22, 10, 9 im Binärcode:

```
0 0 0 0   0 0 0 0     0
0 0 0 0   0 0 0 1     1
0 0 0 0   0 0 1 0     2
0 0 0 0   0 0 1 1     3
0 0 0 0   0 1 0 0     4
...
1 1 1 1   1 1 1 1     255
```

Wird der Wert 255 erreicht, beginnt die Zählung wieder bei null.

```
import RPi.GPIO as GPIO
import time

LED=[2,3,4,17,27,22,10,9]
EXP2=[1,2,4,8,16,32,64,128]

print ("Binary counter")
GPIO.setmode(GPIO.BCM) # access pins by their number

for n in LED:  # Configure all pins to output mode
    GPIO.setup(n, GPIO.OUT)

cnt = 0

try:
    while True:
        for n in range(1,8):
            GPIO.output(LED[n], cnt & EXP2[n])
        time.sleep(0.1)
        cnt += 1 %  256

except KeyboardInterrupt:
    GPIO.cleanup()
    print ("Bye…")
```

Nach dem Import der Bibliotheken RPi.GPIO und „time“ werden zwei Listen definiert. Eine legt die Pins fest, an denen die Binärzahlen ausgegeben werden. Die andere enthält eine Folge von Zweierpotenzen. Dann werden die GPIO-Pins im BCM-Modus (Broadcom-Konfiguration) konfiguriert und als Ausgänge definiert:

```
        for n in LED:
            GPIO.setup(n, GPIO.OUT)
```

Anschließend wird die Hauptschleife gestartet. In jeder Iteration wird der binäre Zähler auf den LEDs dargestellt. Dabei wird der Zählwert (cnt) bitweise mit den Zweierpotenzen aus der EXP2-Liste verknüpft, um den Zustand der LEDs entsprechend zu setzen. Nach jeder Iteration wird eine kurze Pause von 0,1 Sekunden eingelegt, damit der aktuelle Wert abgelesen werden kann. Der Zähler wird unverzüglich bei Erreichen der Zahl 256 zurückgesetzt, um sicherzustellen, dass er im Bereich von 0 bis 255 bleibt. Der Rest des Programms dient wieder dem Abbruch bei einer entsprechenden Tastaturanweisung (Ctrl+C). Der Aufbau ist in Bild 9 dargestellt. Die LEDs sind hier wie üblich an den folgenden Ports angeschlossen (siehe z. B. Bild 3):

LED	1	2	3	4	5	6	7	8
Pi Port #	2	3	4	17	27	22	10	9

Nach dem Starten des Programms in Thonny leuchten die grünen LEDs im Anzeigemodul in der Folge eines Binärzählers (s. o.) auf.

Bild 9: Der Binärzähler zeigt die Zahl 17 (10001000).

Sieben-Segment-Display

Wenn ein Binärzähler zur Verfügung steht, ist es nur noch ein kleiner Schritt zu den bekannten Sieben-Segment-Displays. Diese zeigen Zahlen durch eine spezielle Anordnung von LEDs an. Besonders schön ist dies im PAD-Sieben-Segment-Modul zu sehen, da hier alle LEDs einzeln sichtbar sind (Bild 10).

Bei anderen Sieben-Segment-Anzeigen sind die LEDs als rechteckige Leuchtfelder ausgeführt, sodass die Ziffern noch deutlicher zu erkennen sind (Bild 11).

Um die Binärzahlen 0 bis 9 als Ziffern anzuzeigen, müssen nur die LEDs 1 bis 4 mit den Eingängen A, B, C und D des Ziffernmoduls verbunden werden (Bild 12). Nach dem Start des Programms zählt das Ziffernmodul nun automatisch von 0 bis 9.

Diese Anwendung ist ein schönes Beispiel dafür, wie eine Software-Logikfunktion im Raspberry Pi mit der Hardware-Logik eines Ziffernanzeigemoduls effizient kombiniert werden kann.

Übungen und Anregungen

- Ändern Sie das AND-Programm so ab, dass eine ODER oder ein EXOR-Funktion entsteht!
- Wie ist es möglich, mehr als zwei digitale Eingänge zu verknüpfen?
- Wie muss das Programm zum LED-Lauflicht aussehen, damit der Lichtpunkt von rechts nach links läuft?

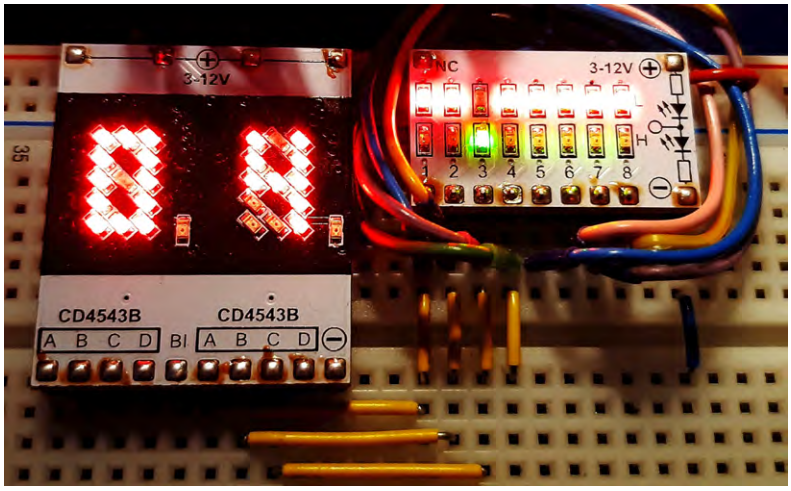

Bild 12: Der Digitalzähler zeigt die Zahl 4.

Material	Artikel-Nr.
z. B. Raspberry Pi 4 Model B, 8 GB RAM	250567
Experimentier-/Steckboard EXSB1	153753
Digitalbausteine und Ziffernanzeige sind im Prototypenadapter-Set PAD6 (CMOS-Logik) enthalten	158980

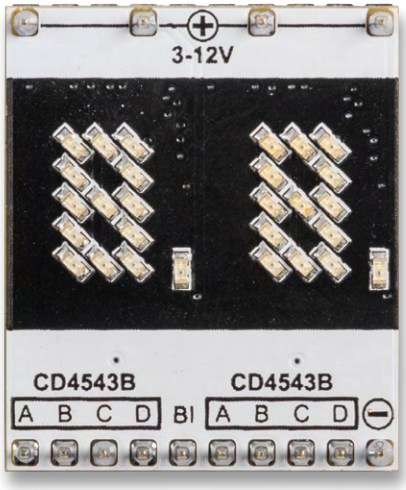

Bild 10: Zweistelliges Sieben-Segment-Display als PAD-Modul

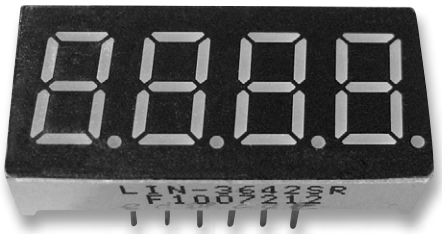

Bild 11: Kommerzielles vierstelliges Sieben-Segment-Display

- Wie kann man den Punkt von rechts nach links und zurück „pendeln" lassen?
- Testen Sie alle weiteren Gatter wie ODER, NOR, EXOR etc. mit dem Logik-Analyzer!
- Wie kann man den Logik-Analyzer auf mehrere Kanäle erweitern?
- Kann man auch Logikgatter mit fünf Eingängen testen? Wenn ja, wie?
- Wie muss man den Binärzähler erweitern, um im Sieben-Segment-Display von 0 bis 99 zu zählen?
- Wie könnte ein vierstelliges Sieben-Segment-Display angesteuert werden?

Ausblick

Nachdem in diesem Beitrag die Logikfunktionen in Python detailliert vorgestellt und praktisch angewendet wurden, sollen im nächsten Beitrag Programmabläufe und -strukturen genauer unter die Lupe genommen werden. Bereits in diesem Artikel wurde im Vorgriff von verschiedenen Schleifen und Entscheidungstechniken Gebrauch gemacht. Im nächsten Beitrag sollen diese grundlegenden Programmelemente detailliert erläutert werden, sodass auch bisher noch nicht vollständig erklärte Programmteile verständlich werden. In weiteren Beiträgen werden dann die in diesem Artikel bereits verwendeten Stringoperationen behandelt, sodass dann auch diese letzte Lücke geschlossen wird. ELV

Weitere Infos

Download-Paket