Python & MicroPython: Programmieren lernen für Einsteiger

Dateneingabe und Tastensteuerung

Teil 7

Der Raspberry Pi eignet sich hervorragend für die Interaktion mit verschiedenen Eingabegeräten. Ob einfache Taster oder komplexere Eingabeeinheiten wie Rotary Encoder und Matrix-Tastaturen – die Möglichkeiten, den Raspberry Pi mit der realen Welt zu verknüpfen, sind vielfältig. In diesem Artikel sollen drei wichtige Arten von Eingabegeräten für den Raspberry Pi näher betrachtet werden: Taster/Druckknöpfe bzw. Schalter, Rotary Encoder und Matrix-Tastaturen mit bis zu 16 Tasten.

Eingabegeräte

Druckknöpfe und Schalter gehören zu den grundlegendsten und am häufigsten verwendeten Eingabeeinheiten. Sie sind ideal für einfache Steuerungsfunktionen wie das Ein- und Ausschalten von Geräten oder das Starten bestimmter Programme oder Programmteile.

Rotary Encoder dagegen erlauben die Erfassung kontinuierlicher Drehbewegungen. Dabei kann sowohl die Richtung als auch die Geschwindigkeit der Drehung erfasst werden. Diese Komponenten eignen sich daher perfekt für Anwendungen in der Robotik, der Maschinensteuerung oder als Scroll-Räder für die Menüauswahl in komplexeren Programmen.

Matrix-Tastaturen sind nützlich, wenn mehrere Eingabetasten benötigt werden. Sie bestehen aus einer Anordnung von Tasten, die in Zeilen und Spalten organisiert sind. Eine 4x4-Matrix-Tastatur zum Beispiel bietet 16 Tasten, die sich auf vier Zeilen und vier Spalten verteilen. Sie eigenen sich zum Bau von Zugangssicherungen, Geldautomaten, Fernbedienungen oder Telefonanlagen etc.

Egal ob einfache Druckknöpfe, Rotary Encoder für präzise Steuerungen oder Matrix-Tastaturen mit mehreren Tasten zum Einsatz kommen sollen – mit dem passenden Python-Programm und einem Raspberry Pi lassen sich diese Eingabegeräte problemlos anschließen und vielseitig nutzen.

RPi.GPIO oder GPIOZero?

Für die Ansteuerung der Pins stehen beim Raspberry Pi zwei Python-Bibliotheken, RPi.GPIO und GPIOZero, zur Verfügung. Beide ermöglichen einen effizienten Einsatz der GPIO-Pins (General Purpose Input/Output) des Raspberry Pi. Sie haben aber unterschiedliche Ansätze und Zielgruppen. Ein Vergleich der beiden Bibliotheken zeigt die Unterschiede:

- **RPi.GPIO:**
  Diese Library ist älter und erfordert mehr manuelle Konfiguration und Programmierung, was für Anfänger schwieriger sein kann. Der Code für die Einrichtung eines GPIO-Pins, um eine LED einzuschalten, sieht so aus:

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setup(18, GPIO.OUT)
GPIO.output(18, GPIO.HIGH)
```

- **GPIOZero:**
  Diese Bibliothek wurde entwickelt, um die Nutzung der GPIO-Pins so einfach wie möglich zu gestalten, und ist besonders für Anfänger geeignet. Sie bietet eine hohe Abstraktionsebene, sodass viele Aufgaben mit weniger Code erledigt werden können. Der Beispielcode für die gleiche Aufgabe wie oben sieht so aus:

```
from gpiozero import LED
led = LED(18)
led.on()
```

Bei RPi.GPIO muss der Benutzer alle GPIO-Einstellungen explizit festlegen. RPi.GPIO bietet somit eine sehr genaue Kontrolle über jedes Detail, wie z. B. das Setzen von Pull-up- oder Pull-down-Widerständen und das Handling von GPIO-Interrupts. Diese Feinsteuerung erfordert aber mehr Codezeilen und ein besseres Verständnis der GPIO-Hardware.

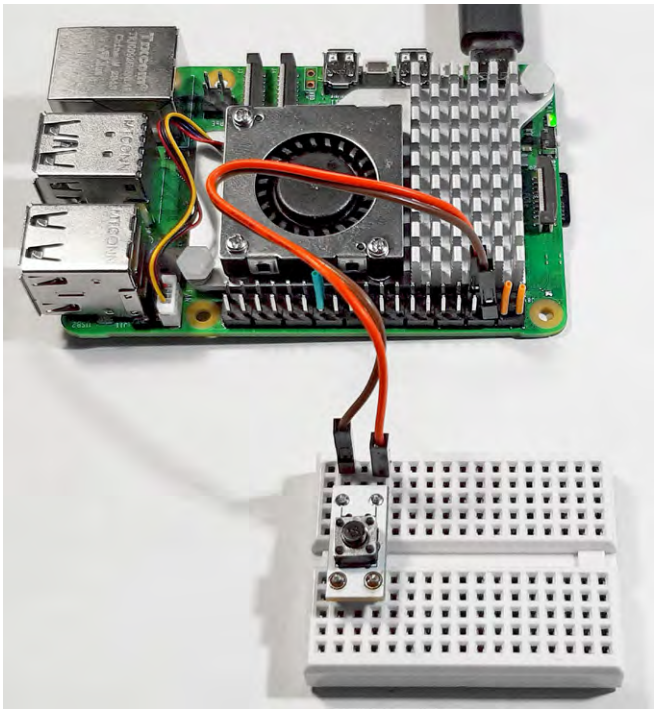

Bild 1: Taster und LED am Raspberry Pi

GPIOZero abstrahiert viele dieser technischen Details, sodass einfache Aktionen wie das Schalten einer LED oder das Lesen eines Buttons mit weniger Code durchgeführt werden können. Komplexere Aktionen können durch Kapselung in leicht zu verwendende Klassen und Methoden erledigt werden. Für einfache Projekte, bei denen es darum geht, schnell mit Hardware zu interagieren, ist GPIOZero die bessere Wahl. Wer hingegen komplexere oder speziellere Anforderungen hat, ist mit RPi.GPIO möglicherweise besser bedient.

Die RPi.GPIO-Bibliothek ist eine der ältesten und am weitesten verbreiteten GPIO-Bibliotheken für den Raspberry Pi, und es gibt derzeit keine offizielle Ankündigung, dass sie „aussterben" könnte oder nicht mehr unterstützt wird. Allerdings gibt es einige Trends und Überlegungen, die in der Zukunft relevant sein könnten. RPi.GPIO wird sicher nicht in Kürze vollkommen verschwinden, aber die Library könnte allmählich an Bedeutung verlieren. Die Bibliothek ist seit vielen Jahren unverändert und bietet eine sehr grundlegende, aber stabile Funktionalität. Da sie älter ist, wird sie möglicherweise nicht mehr aktiv weiterentwickelt, insbesondere wenn neue Modelle oder Hardwareänderungen beim Raspberry Pi auftreten.

Auf dem Raspberry Pi 5 ist GPIOZero bereits die bevorzugte Bibliothek für die GPIO-Steuerung, da sie vielseitiger und einfacher zu bedienen ist. Wer also noch die alte RPi.GPIO-Bibliothek verwenden sollte, muss auf ältere Pi-Modelle zurückgreifen. GPIOZero ist somit aus heutiger Sicht die zukunftssichere Variante. Aus diesem Grund wird im Folgenden auch die GPIOZero-Version im Vordergrund stehen.

Alles auf Knopfdruck

Ein Druckknopfschalter schließt den Stromkreis nur, wenn er gedrückt wird, während ein klassischer Schalter seinen Zustand behält, bis er manuell wieder verändert wird. Beide Komponenten können mit GPIO-Pins des Raspberry Pi verbunden werden und sind leicht mit Python auslesbar.

Mit der GPIOZero-Bibliothek ist das Einlesen von Drucktastern mit dem Raspberry Pi besonders einfach und benutzerfreundlich. Die Library vereinfacht den Umgang mit der GPIO-Hardware erheblich, sodass man mit minimalem Code auskommt. Bild 1 zeigt, wie ein Taster mit dem Raspberry Pi verbunden werden kann.

Die einfachste Konfiguration ist, einen Pin des Tasters mit einem GPIO-Pin des Raspberry Pi und den anderen Pin des Tasters mit GND zu verbinden. Wenn der Taster gedrückt wird, wird der GPIO-Pin auf LOW (0 V) gezogen, andernfalls bleibt er auf HIGH (3,3 V). Diese Konfiguration wird deshalb auch als „LOW"-Aktiv bezeichnet.

Das folgende Programm zeigt, wie man einen Tasterzustand einlesen kann (Button.py):

```
from gpiozero import Button
from signal import pause

button = Button(4)
def on_button_press():
    print("Taste gedrückt!")

button.when_pressed = on_button_press

pause()
```

In diesem Beispiel wird ein Button-Objekt erstellt, das den GPIO-Pin 4 abfragt.

Die Funktion `when_pressed` registriert eine Callback-Funktion. Diese wird immer dann aufgerufen, wenn die Taste gedrückt wird.

Tastenprellen (Bouncing)

Beim Einlesen von Tastern ist das sogenannte Tastenprellen (engl. „Bouncing") ein unerwünschtes, aber leider häufiges Phänomen, das bei mechanischen Schaltern auftritt.

Wenn ein Taster gedrückt wird, „prellen" die Kontaktelemente häufig voneinander ab, und es kann zu mehreren sehr schnellen Schaltvorgängen kommen, obwohl der Taster nur einmal betätigt wurde. Dies kann zu unerwünschten Mehrfachauslösungen führen. Bild 2 zeigt diesen Prellvorgang auf einem Oszilloskop.

Wenn das obenstehende Programm läuft, kann man häufig feststellen, dass die Ausgabe „Taste gedrückt!" zwei- oder sogar mehrfach erfolgt, auch wenn die Taste nur einmal betätigt wurde. Dies ist natürlich im Allgemeinen unerwünscht. GPIOZero hat bereits eine einfache Lösung für dieses Problem integriert: „Debouncing". Man kann den Debounce-Mechanismus einfach konfigurieren, indem man den `bounce_time`-Parameter setzt (Debounce.py).

```
from gpiozero import Button
from signal import pause

button = Button(4, bounce_time=0.1)

def on_button_press():
    print("Taste gedrückt, ohne Prellen!")

button.when_pressed = on_button_press
pause()
```

In diesem Beispiel wurde eine `bounce_time=0.1` gewählt. Das bedeutet, dass GPIOZero nach dem ersten Tastendruck für 100 Millisekunden (= 0,1 s) keine weiteren Ereignisse registriert, was das Prellen effektiv unterdrückt.

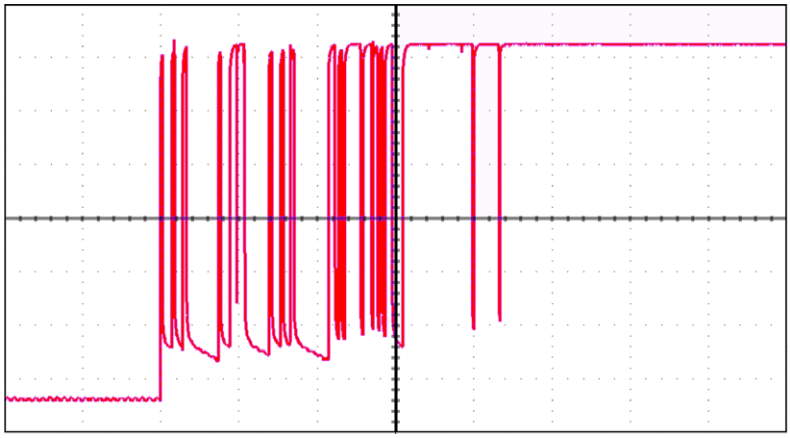

Bild 2: Tastenprellen

Die integrierte Unterstützung für das Entprellen macht es also sehr einfach, zuverlässig und stabil Eingaben zu erfassen. Wenn mechanische Tasten verwendet werden, ist es immer eine gute Idee, das Prellen zu berücksichtigen und den `bounce_time`-Parameter zu nutzen, um unerwünschte Mehrfachauslösungen zu vermeiden. Mit diesen Grundlagen kann man problemlos Taster einlesen und nur auf gewünschte Eingabeereignisse reagieren.

Die Prellzeit (bounce time) eines Tasters hängt von den mechanischen Eigenschaften des Tasters ab. Preisgünstige oder weniger präzise Taster neigen zu längeren Prellzeiten, während qualitativ hochwertige Taster weniger stark prellen. Zudem weisen abgenutzte oder alte Taster, längere Prellzeiten auf. Auch Umweltbedingungen wie Temperatur, Luftfeuchtigkeit oder Schmutz können die mechanische Funktionsweise eines Tasters beeinflussen und das Prellen verstärken.

Typische Prellzeiten liegen zwischen 5 und 50 Millisekunden. Mit 100 ms liegt man also auf der sicheren Seite. Übertrieben lange Entprellzeiten führen auch dazu, dass der Taster träge reagiert. Sie sollten also vermieden werden. Um die Prellzeit zu optimieren, kann man folgendermaßen vorgehen:

- Man startet mit einer typischen Debounce-Zeit von 50 ms.
- Das Schaltverhalten wird beobachtet und die Prellzeit schrittweise reduziert, um eine schnellere Reaktionszeit des Tasters zu erzielen.
- Sobald der Taster Mehrfachauslösungen produziert, muss die Zeit wieder erhöht werden.

Wenn das Prellen besonders stark auftritt und Software-Debouncing nicht ausreicht, kann auch eine Hardware-Entprellung hinzufügt werden. Dies geschieht in der Regel durch das Einfügen eines Kondensators parallel zum Taster oder das Verwenden eines Schmitt-Trigger-Schaltkreises.

Schalten einer LED

Viele moderne Geräte wie Handys, Tablets oder PCs lassen sich mit einem Taster ein- und ausschalten. Diese Variante wird häufig bevorzugt, da klassische Schalter größer, teurer und störanfälliger sind. Zudem kann man bei der Taster-Variante eine automatische Abschaltfunktion vorsehen. Dies ist bei einem Schalter nicht möglich, da Schalter immer manuell betätigt werden müssen. In modernen elektronischen Geräten werden Schalter daher zunehmend durch Taster ersetzt.

Das folgende Programm erlaubt es, eine LED mit einem Taster ein- und auszuschalten (Button_LED_Switcher.py):

```
from gpiozero import Button, LED
from signal import pause

led = LED(23)
button = Button(4, bounce_time=0.1)

led_status = False

def toggle_led():
    global led_status
    if led_status:
        led.off()
        print("LED ausgeschaltet")
    else:
        led.on()
        print("LED eingeschaltet")
    led_status = not led_status

button.when_pressed = toggle_led
pause()
```

Hier wird zunächst ein LED-Objekt für den GPIO-Pin 23, an dem die LED angeschlossen ist, erstellt. Dann wird ein Button-Objekt für den GPIO-Pin 4, der mit dem Taster verbunden ist, erstellt inklusive einer Entprellzeit von 0,1 Sekunden.

Die Funktion `toggle_led()` schaltet die LED um, wenn der Taster gedrückt wird, d. h., wenn die LED an ist, wird sie ausgeschaltet und umgekehrt.

Mit `button.when_pressed` wird die Funktion `toggle_led()` ausgeführt, sobald der Taster gedrückt wird. Die Funktion `pause()` hält das Programm am Laufen, damit es ständig auf ein Taster-Ereignis wartet.

Mit diesem Code kann also eine LED mit einem Tastendruck ein- und ausgeschaltet werden (Bild 3). Die GPIOZero-Bibliothek ermöglicht es dabei, mit minimalem Aufwand und übersichtlichem Code die Aufgabe umzusetzen.

Taster zum Drehen: Rotary Encoder

Ein Rotary Encoder oder Drehgeber misst die Drehbewegung seiner Achse und wandelt diese in elektrische Signale um. Er ermöglicht es, die Position, Geschwindigkeit oder Richtung einer Drehung zu erfassen.

Insbesondere Inkremental-Encoder sind sehr preisgünstig und weit verbreitet. Dieser Typ gibt bei jeder Drehbewegung Pulse aus. Die Anzahl der Pulse pro Umdrehung bestimmt die Genauigkeit. Die Drehrichtung kann durch zwei zeitversetzte Signale (A und B) bestimmt werden.

Derartige Encoder werden in vielen Systemen verwendet, um präzise Steuerung oder Positionsfeedback zu bieten. Vor allem bei der Motorsteuerung, in der Robotik oder in Bedieneinheiten von z. B. CNC-Maschinen sind sie häufig zu finden.

Bild 4 zeigt, wie man einen Rotary Encoder an den Raspberry Pi anschließen kann:

Die Bedieneinheit MEXB-BP1 stellt einen passenden Encoder zur Verfügung (s. Abschnitt Material), aber auch andere Inkrementalgeber können hier problemlos verwendet werden. Dabei werden die folgenden Leitungen benötigt:

Encoder	Raspberry Pi
clk_pin (A)	17
dt_pin (B)	18
C	GND
sw_pin	4 (optional)

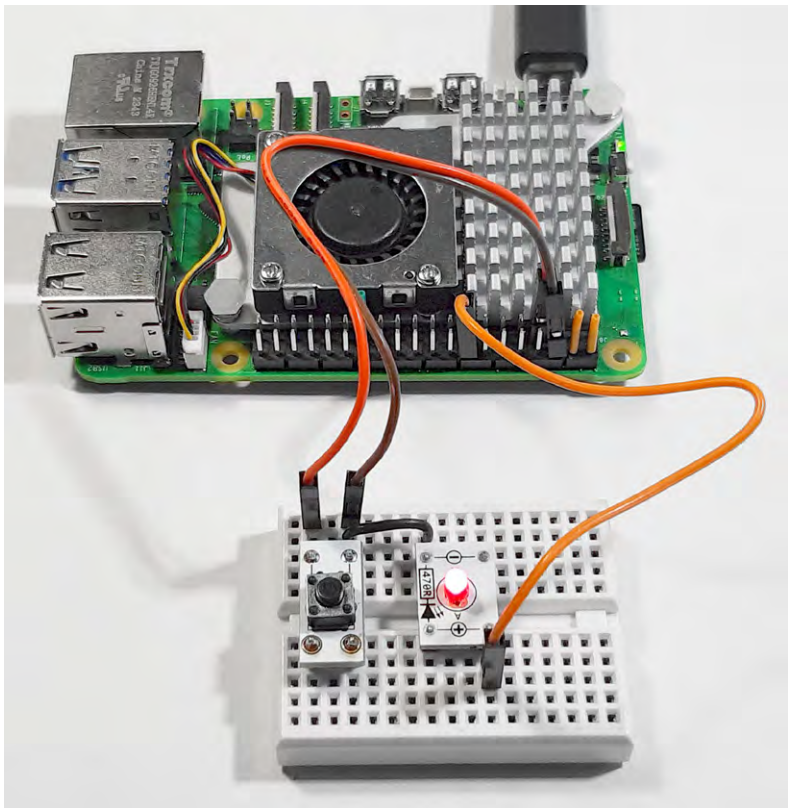

Bild 3: Raspberry Pi schaltet eine LED.

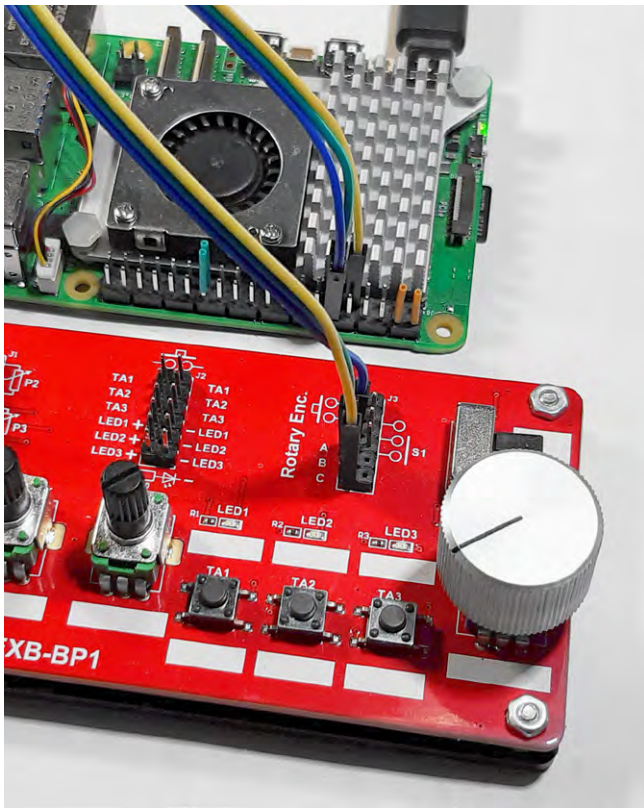

Bild 4: Rotary Encoder am Raspberry Pi

Das zugehörige Programm sieht so aus (RotEnc.py):

```
from gpiozero import RotaryEncoder, Button
from signal import pause

clk_pin = 17  # CLK (A)
dt_pin = 18   # DT (B)
sw_pin = 4    # Optional: Button Pin

encoder = RotaryEncoder(clk_pin, dt_pin, max_steps=100)
button = Button(sw_pin, pull_up = True, bounce_time= None)

def turned():
    print(f"Position: {encoder.steps}")

def button_pressed():
    print("Button gedrückt!")

encoder.when_rotated = turned
button.when_pressed = button_pressed
pause()
```

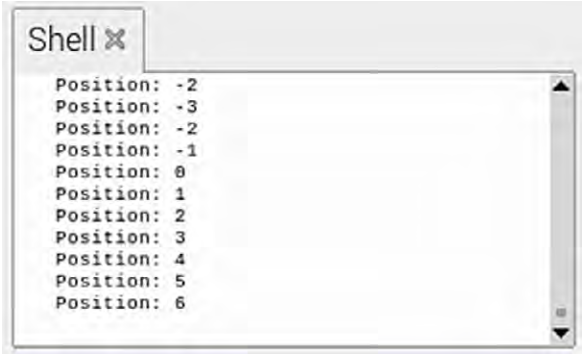

Bild 5: Rotary-Encoder-Positionsdaten

Die Ausgabe der relativen Position des Encoders erfolgt in die Shell (Bild 5).

Viele Encoder besitzen eine Zusatzfunktion: Beim Drücken auf die Achse wird ein zusätzlicher Schalter aktiviert. Diese Funktion ist im obigen Programm ebenfalls implementiert und kann bei Bedarf genutzt werden. Hierfür muss zusätzlich der Achsenschalter des Drehencoders mit Pin 4 des Raspberry Pi verbunden werden.

Im letzten Beitrag dieser Serie wurde die MatPlotLib-Bibliothek ausführlich dargestellt. In Kombination mit dem Rotary Encoder können nun die Eingangsdaten des Bauelements grafisch dargestellt werden.

Das folgende Programm (RotEnc_to_MatPlotLib.py) liefert die Grundlage dazu:

```
from gpiozero import RotaryEncoder, Button
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import time

clk_pin = 17  # CLK (A)
dt_pin = 18   # DT (B)

encoder = RotaryEncoder(clk_pin, dt_pin, max_steps=100)
button = Button(sw_pin, pull_up = True, bounce_time= None)

positions = []
times = []

start_time = time.time()

def update_plot(frame):
    current_time = time.time() - start_time
    current_position = encoder.steps

    positions.append(current_position)
    times.append(current_time)

    ax.clear()
    ax.plot(times, positions, label='Encoder Position')
    ax.set_xlabel('Zeit (s)')
    ax.set_ylabel('Position (Schritte)')
    ax.legend()

fig, ax = plt.subplots()
ani = FuncAnimation(fig, update_plot, interval=100)
plt.show()
```

Das Programm erfasst die Drehposition eines Rotary Encoders in Echtzeit und stellt diese grafisch dar. Der `RotaryEncoder` wird mit den Pins (17 und 18) und einer maximalen Schrittzahl von 100 initialisiert.

Danach werden zwei Listen, `positions` und `times`, erstellt, um die Position des Encoders und die dazugehörige Zeit zu speichern. Die Zeitmessung beginnt mit `start_time`, um die Zeitdifferenz während der Laufzeit zu berechnen.

Die Funktion `update_plot` wird in regelmäßigen Abständen (alle 100 Millisekunden) von `FuncAnimation` aufgerufen.

Diese Funktion liest die aktuelle Position des Encoders (encoder.steps) und berechnet die verstrichene Zeit seit `start_time`. Zudem aktualisiert sie das Diagramm, das so die Position des Encoders über die Zeit darstellt. Via MatPlotLib wird damit eine Echtzeit-Grafik angezeigt, in der auf der x-Achse die Zeit in Sekunden und auf der y-Achse die Encoder-Position (in Schritten) dargestellt ist. Die Grafik wird kontinuierlich aktualisiert, während der Rotary Encoder gedreht wird. Das Programm zeigt somit die Position des Rotary Encoders in Echtzeit als Funktion der Zeit in einem animierten Diagramm an (Bild 6).

Digitaler LED-Dimmer

Bei einem klassischen LED-Dimmer kommt meist ein Potentiometer zusammen mit einem stromverstärkenden Transistor zum Einsatz. Diese Variante hat den Nachteil, dass man die Auflösung nicht einstellen kann, da sie sich aus dem Drehbereich des Potentiometers ergibt. Bei einer modernen Variante eines LED-Dimmers mithilfe eines Rotary Encoders kann man dagegen die Auflösung und die Schrittweite der LED-Helligkeit sehr flexibel wählen.

Das folgende Programm (RotEnc_to_LED.py) zeigt, wie man die Helligkeit einer LED über einen Rotary Encoder verändern kann.

```
from gpiozero import RotaryEncoder, PWMLED
from signal import pause

clk_pin = 17  # CLK (A)
dt_pin = 18   # DT (B)
led_pin = 23  # PWM-fähiger Pin für die LED

encoder = RotaryEncoder(clk_pin, dt_pin, max_steps=10)  # 100 Schritte insgesamt
led = PWMLED(led_pin)

def update_led_brightness():
 brightness = max(0, min(1, encoder.steps / 10))  # Begrenzung auf Werte zwischen 0 und 1
 led.value = brightness
 print(f"Helligkeit: {brightness * 100:.0f}%")

encoder.when_rotated = update_led_brightness

print("Drehen Sie den Rotary Encoder, um die LED-Helligkeit anzupassen.")
pause()
```

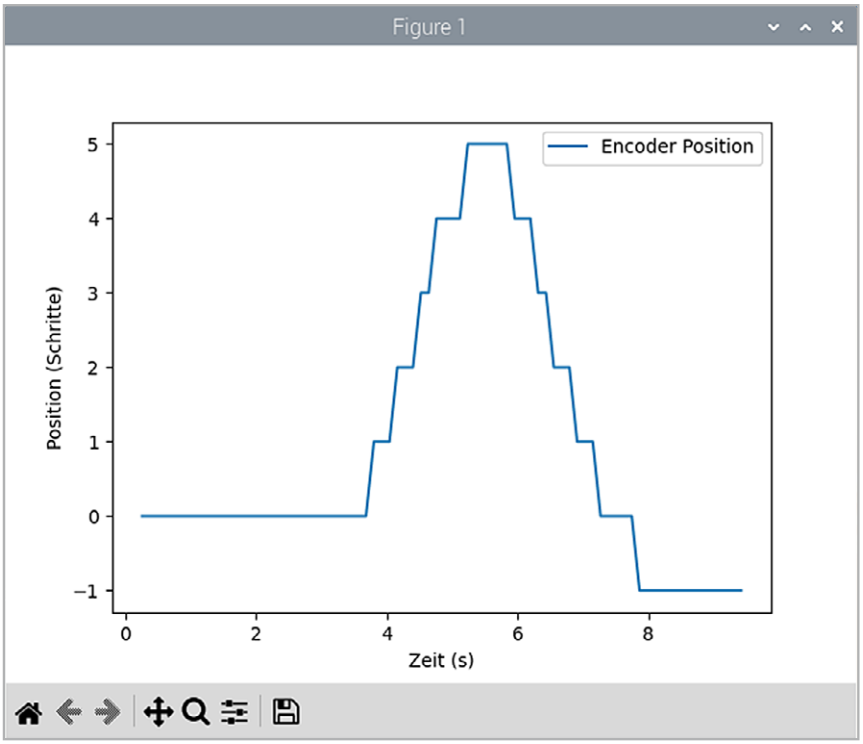

Bild 6: Grafische Darstellung der Rotary-Encoder-Positionsdaten

Hinweis: Die im Text grün markierten Programme sind im Download-Paket enthalten.

Der Rotary Encoder wird über die GPIO-Pins 17 (A: CLK) und 18 (B: DT) abgefragt. Mit `max_steps=100` wird der Encoder auf 100 Schritte begrenzt.

Die LED ist an einem PWM-fähigen Pin (GPIO 23) angeschlossen, sodass ihre Helligkeit zwischen 0 und 100 % stufenlos verändert werden kann.

Die Funktion `update_led_brightness` wird immer dann aufgerufen, wenn der Rotary Encoder gedreht wird. Sie berechnet die Helligkeit der LED, basierend auf der aktuellen Position des Encoders (zwischen 0 und 100 Schritten).

Der Wert für die LED wird als PWM-Wert (zwischen 0 und 1) an die LED weitergegeben, um die Helligkeit anzupassen.

`pause()`: Hält das Programm wieder im Hintergrund am Laufen, sodass es jederzeit auf Ereignisse vom Rotary Encoder reagieren kann.

Zur Kontrolle wird der aktuelle Helligkeitslevel zusätzlich in der Shell ausgegeben.

Auf der Hardwareseite muss die Schaltung nach Bild 3 lediglich um eine LED (mit Vorwiderstand) an Pin 23 ergänzt werden. Nach dem Starten des Programms kann deren Helligkeit dann mit dem Rotary Encoder verändert werden. Im Abschnitt „Ergänzungen und Übungen" finden sich einige Anregungen, wie man das Programm an verschiedene Anforderungen anpassen kann.

Matrix-Tastaturen

Matrix-Tastaturen sind nützlich, wenn mehrere Eingabetasten benötigt werden. Sie bestehen aus einer Anordnung von Tasten, die in Zeilen und Spalten organisiert sind. Eine 4x4-Matrix-Tastatur zum Beispiel bietet 16 Tasten, die sich über vier Zeilen und vier Spalten verteilen.

Eine Matrix-Tastatur verwendet ein spezielles Netzwerk, um die Tasten miteinander zu verbinden. Jede Taste schließt einen anderen Strompfad zwischen einer Zeile und einer Spalte. Dies ermöglicht es dem Raspberry Pi zu erkennen, welche Taste gedrückt wurde, ohne für jede Taste einen eigenen GPIO-Pin zu benötigen.

Wird beispielsweise eine 4x4-Matrix verwendet, bräuchte man bei einer direkten Verschaltung der Tasten 17 Leitungen (eine für jede Taste plus eine Ground-Leitung). Bei einer Matrix-Verschaltung hingegen werden nur acht Leitungen (vier Zeilen und vier Spalten) benötigt.

Zum Auslesen der Matrix werden spezielle Bibliotheken verwendet, um die Tastendrücke zu dekodieren und auf die gedrückten Tasten zu reagieren.

Um eine Matrix-Tastatur an den Raspberry Pi anzuschließen, sind also mehrere GPIO-Pins erforderlich, je nachdem wie viele Zeilen und Spalten die Tastatur hat. Mit Python-Bibliotheken können Tastendrücke einfach ausgelesen und verarbeitet werden.

Bild 7 zeigt Matrix-Tastaturen in mehreren Varianten. Diese Tastaturen haben gemeinsam, dass sie jeweils 16 bzw. zwölf Tasten und acht bzw. sieben Anschlussleitungen besitzen. Sie können z. B. gemäß der folgenden Tabelle an die GPIO-Pins eines Raspberry Pi angeschlossen werden.

Zeilenleitungen – IO-Pins:	17, 27, 22, 5
Spaltenleitungen – IO-Pins:	6, 13, 19, 26

Bei der 12-Tasten-Matrix kann eine Spaltenleitung entfallen. Bild 8 zeigt eine Aufbauvorschlag für den Anschluss der Folientastatur.

Um die Tastatur auszulesen, kann das folgende Beispielprogramm (KeyMatrix.py) verwendet werden. Es nutzt die Bibliothek `gpiozero` und gibt die jeweils gedrückte Taste auf der Konsole aus.

```
from gpiozero import DigitalOutputDevice, Button
from time import sleep

# GPIO-Pins für die Zeilen und Spalten definieren
rows = [17, 27, 22, 5]
columns = [6, 13, 19, 26]

key_map = [
 ['1', '2', '3', 'A'],
 ['4', '5', '6', 'B'],
 ['7', '8', '9', 'C'],
 ['*', '0', '#', 'D']
]

row_pins = [DigitalOutputDevice(pin) for pin in rows]
col_pins = [Button(pin, pull_up=False) for pin in columns]

def scan_matrix():
 for row_num, row_pin in enumerate(row_pins):
 row_pin.on()  # Aktiviere die aktuelle Zeile
 for col_num, col_pin in enumerate(col_pins):
 if col_pin.is_pressed:
 print(f"Taste {key_map[row_num][col_num]} gedrückt")
 row_pin.off()  # Deaktiviere die aktuelle Zeile
 sleep(0.1)

try:
 while True:
 scan_matrix()
 sleep(0.1)  # kurze Pause zwischen den Scans
except KeyboardInterrupt:
 print("Programm beendet.")
```

Nach dem Import der Bibliotheken

```
gpiozero uind time import
```

erfolgt die Definition der GPIO-Pins:

```
rows = [17, 27, 22, 5]
```

Die GPIO-Pins für die Zeilen (oben) sind je nach Verkabelung anpassbar, ebenso die GPIO-Pins für die Spalte:

```
columns = [6, 13, 19, 26]
```

Die Definition der Tastenbelegung erfolgt in einem Array:

```
key_map = [
    ['1', '2', '3', 'A'],
    ['4', '5', '6', 'B'],
    ['7', '8', '9', 'C'],
    ['*', '0', '#', 'D']
]
```

Diese sogenannte `key_map` besteht aus einem 4x4-Array, das die Zuordnung der Tasten in der Matrix definiert. Jede Taste hat eine spezifische Position in der Matrix. Zum Beispiel befindet sich die Taste '1' in der ersten Zeile und ersten Spalte, während sich 'D' in der vierten Zeile und vierten Spalte befindet. Natürlich können hier, je nach Tastenbeschriftung auch andere Werte gewählt werden. Es folgt die Initialisierung der GPIO-Pins:

```
row_pins = [DigitalOutputDevice(pin) for pin in rows]
col_pins = [Button(pin, pull_up=False) for pin in columns]
```

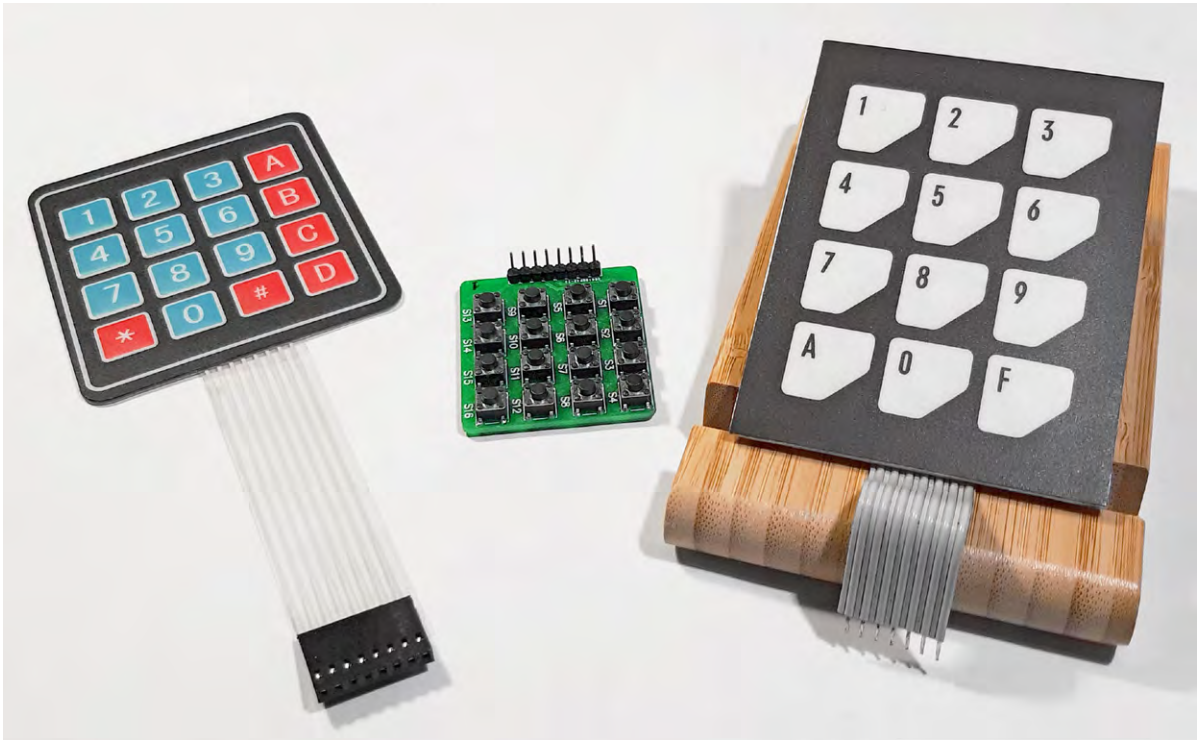

Bild 7: Matrix-Tastaturen

- Die Zeilen-Pins werden als `DigitalOutputDevice`-Objekte initialisiert, sodass sie als digitale Ausgänge verwendet werden können. Sie steuern die Stromversorgung der Zeilen der Tastenmatrix.
- Die Spalten-Pins werden dagegen als Button-Objekte initialisiert, die als Eingänge fungieren. Der Parameter `pull_up=False` bedeutet, dass kein interner Pull-up-Widerstand verwendet wird (dieser ist normalerweise nur bei Einzeltasten nützlich).

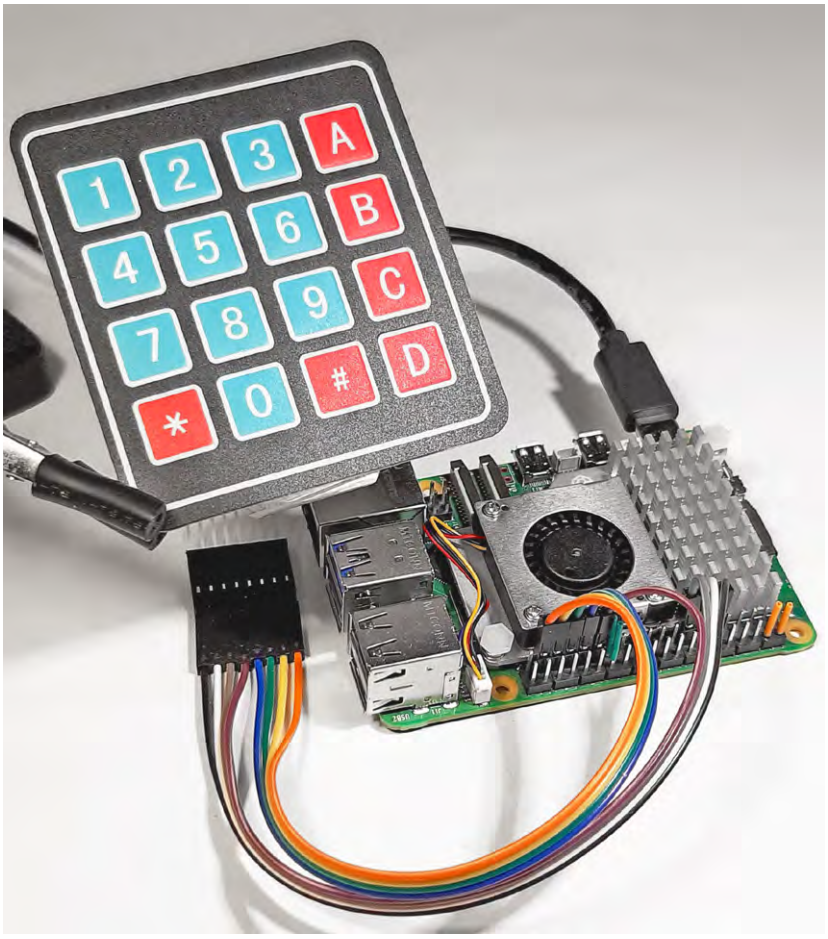

Bild 8: Folientastatur am Raspberry Pi

Die Funktion `scan_matrix()` dient zur Überprüfung der Tastenmatrix. Diese Funktion durchläuft die Zeilen der Matrix, aktiviert jeweils eine Zeile und überprüft, ob eine Taste in einer der Spalten gedrückt wurde.

- `row_pin.on()`: Aktiviert die Stromversorgung für die aktuelle Zeile.
- `col_pin.is_pressed`: Überprüft, ob die Taste in der aktuellen Spalte gedrückt wurde. Wenn ja, wird die entsprechende Taste aus der `key_map` ausgegeben.
- `row_pin.off()`: Deaktiviert die Zeile, bevor zur nächsten Zeile übergegangen wird.
- Die Endlosschleife ruft die `scan_matrix()`-Funktion auf, um kontinuierlich auf Tastendrücke zu prüfen. Die Funktion `sleep(0.1)` sorgt für eine kurze Pause zwischen den Scans, um den Raspberry Pi nicht zu überlasten und eine zu schnelle Abfrage zu vermeiden. Falls das Programm mit „Strg + C“ beendet wird, gibt es eine entsprechende Ausgabe („Programm beendet“) aus.

Bild 9 zeigt die Ausgabe auf der Shell, nachdem die Tasten 1, 2, 3, und A, B, C auf der Matrix-Tastatur gedrückt wurden.

Bild 9: Testausgabe der Folientastatur am Raspberry Pi

Ergänzungen und Übungen

- Bestimmen Sie die optimale Prellzeit zum Schalten einer LED mit einem bestimmten Tastertyp.
- Anpassungsmöglichkeiten für die LED-Helligkeitssteuerung:
  Pinbelegung: Ändern Sie die clk_pin, dt_pin und led_pin je nach Hardware-Setup.
  Maximale Schritte: Passen Sie max_steps so an, dass die LED sehr feinfühlig gesteuert werden kann.
- Welche Möglichkeiten gibt es, eine falsche Zuordnung der Tasten bei einer Folientastatur zu korrigieren (Hardware, Software)?

Ausblick

In diesem Artikel wurde bereits die Technik der Pulsweitenmodulation (PWM) zur Ansteuerung der Helligkeit einer LED verwendet. Im nächsten Beitrag soll dieses häufig eingesetzte Verfahren genauer betrachtet werden. Zudem sollen sogenannte Digital-Analog-Converter zum Einsatz kommen. Diese Technologien gestatten es, analoge Spannungen auszugeben und so z. B. die Helligkeit von LEDs zu verändern. Es lassen sich aber nicht nur LEDs steuern, sondern z. B. auch die Geschwindigkeit von Motoren. So können z. B. Roboter oder Lüfter sehr präzise angesteuert werden. ELV

Material

Raspberry Pi mit Netzteil

z. B. Raspberry Pi 4 Model B, 8 GB RAM	Artikel-Nr. 250567
z. B. Raspberry Pi 4 USB-Netzteil Typ C	Artikel-Nr. 250962
Bedienpanel MEXB-BP1	Artikel-Nr. 157431

Kleinteile (LEDs, Taster etc.) finden sich z. B. im PAD-PRO-EXSB Professional Set

Folientastatur oder selbst aufgebaute Tastenmatrix

Zum Download-Paket

Python & MicroPython: Alle bisherigen Beiträge im Überblick

Teil 1: Erste Schritte
Zum Beitrag

Teil 2: GPIOs steuern die Welt
Zum Beitrag

Teil 3: Digitale Logik
Zum Beitrag

Teil 4: Ablaufsteuerung und Programmstrukturen
Zum Beitrag

Teil 5: Erfassung analoger Werte
Zum Beitrag

Teil 6: Grafikkunst mit MatPlotLib
Zum Beitrag