c't Heft 3/2022 S. 154-159 / Praxis - Raspi-GPIO in Python

Auf Tritt

GPIO-Pins des Raspi in Python programmieren

Den Erfolg im Bastel-Business verdankt der Raspberry Pi seinem GPIO-Anschluss. Damit programmieren Sie leicht eine Mini-Lightshow oder fragen einen Buzzer ab und lassen dem Kandidaten ein Licht aufgehen. Mit der richtigen Python-Bibliothek und unserer Anleitung gelingen auch komplexe Aufgaben und der Raspi steuert bei Ihrem nächsten Live-Auftritt Kamera und Mikrofon.

Ein Raspberry Pi, eine LED, ein Widerstand und wenige Zeilen Python-Code sind alles, was Sie für eine blinkende Lightshow benötigen. Möglich macht das der 40-polige GPIO-Anschluss (General Purpose Input/Output, universelle Ein-/Ausgabe), an dem Sie ohne großen Aufwand LEDs, aber auch Taster, Schalter, Servomotoren, Sensoren und sogar TFT-Displays anschließen können. Die zugehörige Python-Bibliothek GPIO ist auf Raspberry Pi OS bereits vorinstalliert. Es gibt aber auch Bibliotheken für etliche andere Programmiersprachen. Wir haben uns für Python entschieden, weil die Programmiersprache leicht zu erlernen und dank zahlloser Bibliotheken nahezu grenzenlos erweiterbar ist. Damit sind die Möglichkeiten keineswegs auf triviale Dinge wie eine blinkende LED beschränkt. Mit dem nachfolgenden Beispielprogramm steigen Sie gleich ins Show-Business ein und steuern über einen zweckentfremdeten Gitarrenfußschalter für wenige Euro Kamera, Mikrofon, Aufnahme oder gar den kompletten Live-Stream in OBS Studio Ihres Streaming-Rechners.

c't Hardcore

Von den 40 Pins des GPIO-Ports stehen Ihnen 28 für Schalt- und Steueraufgaben zur Verfügung, die übrigen sind Masseanschlüsse oder dienen der Spannungsversorgung. Bei letzteren lauert auch gleich die erste Stolperfalle: An den Pins 2 und 4 liegt die Eingangsspannung (Vin) des Raspi von 5 Volt an. Hierüber können Sie die Eingangsspannung des Raspi für eigene Schaltungen abgreifen oder aber den Raspi mit Strom ohne ein zusätzliches Netzteil versorgen - so arbeitet zum Beispiel das PoE-Hat (Power over Ethernet). Die Schaltpins und Eingänge arbeiten jedoch nur mit der Versorgungsspannung 3,3 Volt (VCC), die an den Pins 1 und 17 zur Verfügung steht - bei mehr gehen sie kaputt! Trotz der niedrigen Spannung dürfen Sie eine LED nicht direkt an den GPIO-Pins betreiben, Sie benötigen zusätzlich einen Vorwiderstand zwischen 220 und 470 Ohm, um den Stromfluss zu begrenzen.

c't kompakt

Die Python-Bibliothek RPi.GPIO steuert LEDs und fragt Taster und Schalter mit wenigen Befehlen ab.
Wie Sie asynchrone Funktionen über GPIO-Event-Handler aufrufen und externe Programme wie OBS Studio per WebSocket ansteuern.

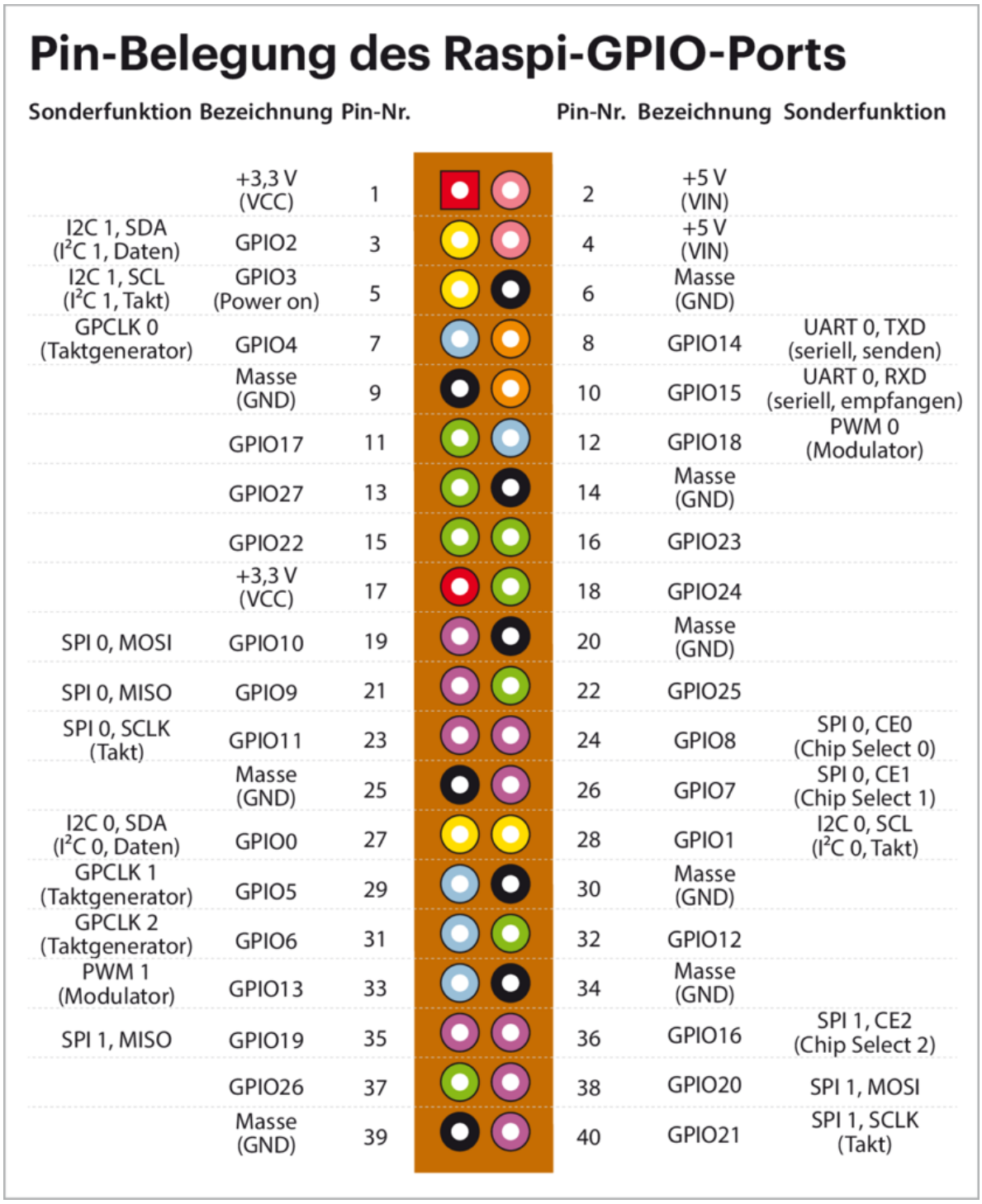
Licht an

Prinzipiell können Sie die LED an jeden der 28 schaltbaren GPIO-Pins anschließen, doch einige besitzen Zusatzfunktionen wie Taktgenerator, Modulator oder sind Teil eines Bussystems wie SPI oder I2C. Im Kasten "Raspberry Pi als Schaltzentrale" haben wir diese Besonderheiten zusammengefasst und die GPIO-Pins farblich so gekennzeichnet, dass Sie etwaige Sonderfunktionen leicht erkennen können. Um eine Schaltung später nicht umgestalten zu müssen, weil Sie zusätzlich ein bestimmtes Bussystem brauchen, sollten Sie für einfache Schaltaufgaben wie eine LED einen der grünen GPIO-Schaltpins ohne besondere Zusatzfunktion benutzen. In den nachfolgenden Beispielen ist es Pin 37.

Raspberry Pi als Schaltzentrale

Der 40-polige GPIO-Anschluss ist es, was den Raspberry Pi vom herkömmlichen PC unterscheidet und für viele Basteleien besonders interessant macht: Über die Pins lassen sich nicht nur LEDs und andere Komponenten ein- und ausschalten, sie können auch als Eingänge für Taster und Schalter dienen, modulierte Signale erzeugen und über die Bussysteme SPI (Serial Peripherial Interface) und I²C (Inter-Integrated Circuit) Displays ansteuern oder Messwerte komplexer Sensoren auslesen. Dafür besitzen etliche Pins bis zu fünf Funktionen: Pin 3 und 5 zum Beispiel können LEDs oder Optokoppler schalten, als Taster-Eingänge fungieren oder als I²C-Bus Sensoren auslesen.

In der Übersicht der GPIO-Pins haben wir nur die wichtigsten Sonderfunktionen aufgeführt. Diese soll Ihnen helfen, etwa für eine LED den geeigneten GPIO-Pin zu finden: Zwar könnten Sie sie an Pin 3 oder 5 betreiben, doch damit wäre der I²C-Bus blockiert für den Fall, dass Sie Ihre Schaltung später noch erweitern wollen. Besser geeignet sind dafür die grün markierten Pins, etwa 11, 16 oder 37, die keine häufig benötigte Zusatzfunktion haben.



Pin-Belegung des Raspi-GPIO-Ports

Sonderfunktion	Bezeichnung	Pin-Nr.		Pin-Nr.	Bezeichnung	Sonderfunktion
	+3,3 V (VCC)	1		2	+5 V (VIN)	
I2C 1, SDA (I²C 1, Daten)	GPIO2	3		4	+5 V (VIN)	
I2C 1, SCL (I²C 1, Takt)	GPIO3 (Power on)	5		6	Masse (GND)	
GPCLK 0 (Taktgenerator)	GPIO4	7		8	GPIO14	UART 0, TXD (seriell, senden)
	Masse (GND)	9		10	GPIO15	UART 0, RXD (seriell, empfangen)
	GPIO17	11		12	GPIO18	PWM 0 (Modulator)
	GPIO27	13		14	Masse (GND)	
	GPIO22	15		16	GPIO23	
	+3,3 V (VCC)	17		18	GPIO24	
SPI 0, MOSI	GPIO10	19		20	Masse (GND)	
SPI 0, MISO	GPIO9	21		22	GPIO25	
SPI 0, SCLK (Takt)	GPIO11	23		24	GPIO8	SPI 0, CE0 (Chip Select 0)
	Masse (GND)	25		26	GPIO7	SPI 0, CE1 (Chip Select 1)
I2C 0, SDA (I²C 0, Daten)	GPIO0	27		28	GPIO1	I2C 0, SCL (I²C 0, Takt)
GPCLK 1 (Taktgenerator)	GPIO5	29		30	Masse (GND)	
GPCLK 2 (Taktgenerator)	GPIO6	31		32	GPIO12	
PWM 1 (Modulator)	GPIO13	33		34	Masse (GND)	
SPI 1, MISO	GPIO19	35		36	GPIO16	SPI 1, CE2 (Chip Select 2)
	GPIO26	37		38	GPIO20	SPI 1, MOSI
	Masse (GND)	39		40	GPIO21	SPI 1, SCLK (Takt)

Etliche Pins des GPIO-Anschlusses haben wichtige Sonderfunktionen, weshalb man einfache Schaltaufgaben den grün markierten Pins übertragen sollte.

Manchmal überschneiden sich auch Anforderungen: Pin 5 ist nicht nur die Taktleitung des I²C-Busses. Schließt man diesen Pin gegen Masse kurz, so schaltet er den Raspi wieder ein, nachdem er heruntergefahren wurde. Möchte man den Raspi also mit einem Power-Taster versehen, muss man auf den I²C-Bus verzichten oder getrennte Tasten zum Ein- und Ausschalten verwenden mit dem Risiko, dass bei Fehlbedienung die I²C-Buskommunikation zusammenbricht. Auf den zweiten I²C-Bus auf den Pins 27 und 28 (Gelb) sollte man aber nicht ausweichen, denn diese Pins sind für die Kommunikation mit dem EPROM eines HATs (Hardware Attached on Top, Huckepackplatine) reserviert, um falls notwendig automatisch zusätzliche Konfigurationsparameter oder Treiber zu laden. Manchmal gibt es auch Alternativen: So lassen sich die PWM-Generatoren auf den Pins 12 und 33 auch auf die Pins 32 und 35 verlegen, indem Sie in der Datei config.txt der Boot-Partition des Raspi zusätzliche Parameter angeben.

Eine häufige Stolperfalle ist, dass manche Softwareprojekte die internen GPIO-Port-Nummern des Broadcom-Chips (BCM) und nicht die auf der Platine aufgedruckten Pin-Nummern verwenden. GPIO 17 am Chip etwa ist mit Pin 11 der 40-poligen Steckerleiste verbunden - das führt zu Verwirrungen. Bei eigenen Projekten sollten Sie die Pin-Nummern von der Platine bevorzugen, denn das erleichtert den Nachbau.

Die nächste anstehende Entscheidung ist, welche Schaltlogik Sie verwenden wollen. Zur Wahl stehen Active High und Active Low. Bei Active High liefert der GPIO-Schaltpin die Versorgungsspannung für die LED. LED und Vorwiderstand werden dann zwischen Pin 37 und Masse geschaltet. Wird der Pin auf High-Pegel geschaltet, also 3,3 Volt, leuchtet die LED. Bei Active Low hingegen werden LED und Vorwiderstand an Pin 1 oder 17 angeschlossen und darüber mit Spannung versorgt - damit die LED leuchtet, müssen Sie den Pin 37 auf Low-Potenzial schalten, also Masse. Beim nachfolgenden Beispiel haben wir uns für Active High entschieden, deshalb schließen Sie die LED nebst Vorwiderstand zwischen Pin 37 und Pin 39 des GPIO-Ports an.

Für die Ansteuerung in Python genügen wenige Kommandos, die Sie direkt in den Python-Interpreter eingeben können. Dazu starten Sie ihn als Benutzer pi mit dem Befehl *python3* und erhalten dann das Python-Prompt, bestehend aus drei Größer-Zeichen. Im ersten Schritt binden Sie die GPIO-Bibliothek des Raspi ein:

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

Wie im Kasten "Raspberry Pi als Schaltzentrale" beschrieben sollen die Pins über ihre Pin-Nummer auf der Platine des Raspis referenziert werden, dies legen Sie mit folgendem Befehl fest:

```
GPIO.setmode(GPIO.BOARD)
```

Pin 37 alias GPIO26 ist für die LED-Ansteuerung vorgesehen, es handelt sich also um einen Ausgang. Da wir uns für Active High entschieden haben und die LED im Normalfall nicht leuchten soll, initialisieren Sie den Pin mit Low-Potenzial:

```
GPIO.setup(37, GPIO.OUT,
           initial=GPIO.LOW)
```

Um die LED einzuschalten, müssen Sie nur noch den Pin auf High-Potenzial setzen:

```
GPIO.output(37, GPIO.HIGH)
```

Zum Ausschalten setzen Sie den Ausgang wieder auf Low:

```
GPIO.output(37, GPIO.LOW)
```

Bevor Sie den Python-Interpreter mit Strg+D verlassen, sollten Sie mit *GPIO.cleanup()* noch "aufräumen" und den GPIO-Anschluss wieder in den Grundzustand zurückversetzen.

Auf und ab

Alle GPIO-Schaltpins lassen sich darüber hinaus auch als digitale Eingänge benutzen. Sie können also feststellen, ob an einem Pin High-Potenzial (3,3 Volt) oder Low-Potenzial (0 Volt, Masse) anliegt. Mit den folgenden Zeilen initialisieren Sie Pin 5 als Eingang:

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BOARD)
```

```
GPIO.setup(5, GPIO.IN)
```

Anschließend können Sie abrufen, welcher Pegel an dem Pin anliegt:

```
while True:
    if GPIO.input(5):
        print(" Pin 5: High")
    else:
        print(" Pin 5: Low")
        break
```

Schließen Sie einen Taster an Pin 5 und Pin 6 (Masse) an, so ist Pin 5 im Normalfall High und Low, wenn Sie den Taster gedrückt halten - und die Schleife endet.

Zugegeben, mit diesen Zeilen brechen Sie die zuvor verkündete Regel, für einfache Schaltaufgaben keinen Pin eines Bussystems zu verschwenden. Denn Pin 5 gehört zum zweiten I2C-Bus des Raspi. Er hat aber noch eine weitere Sonderfunktion: Ein dort angeschlossener Taster startet den Raspi neu, wenn er zuvor heruntergefahren wurde. Damit das auch klappt, während der Raspi heruntergefahren ist, wurde an Pin 5 werksseitig ein sogenannter Pull-up-Widerstand angeschlossen und mit VCC verbunden.

Der Pull-up-Widerstand bewirkt, dass Pin 5 im Normalfall stets auf High-Potenzial liegt. Erst wenn Sie den Pin mittels Taster gegen Masse kurzschließen, erkennt der Raspi Low - dieser Eingang arbeitet also mit der Logik Active Low. Dabei verhindert ein Pull-up-Widerstand von 1,8 kOhm bei den Pins 3 und 5, dass während des Kurzschlusses zu viel Strom fließt und der Raspi beschädigt wird. Wenn Sie also einen Taster zum Starten und Herunterfahren Ihres Raspi haben wollen, ist Pin 5 trotz Sonderfunktion der Richtige. Anstelle von *break* müssten Sie zum Herunterfahren Python nur den Linux-Befehl *shutdown -h now* ausführen lassen.

Bei anderen Pins gibt es keine fest verdrahteten Widerstände, sondern schaltbare: Hier können Sie bei der Initialisierung des Pins festlegen, ob Sie einen Widerstand brauchen und ob er als Pull-up-Widerstand gegen VCC oder als Pull-down-Widerstand gegen Masse geschaltet werden soll. Die folgende Zeile aktiviert Pin 16 als Eingang und schaltet einen Pull-up-Widerstand gegen VCC, ähnlich wie er bei Pin 5 zum Einsatz kommt:

```
GPIO.setup(16, GPIO.IN,
           pull_up_down=GPIO.PUD_UP)
```

Hitzkopf

Den Zustand eines Pins mit einer *while* -Schleife abzufragen hat eine ganz erhebliche Nebenwirkung: Sie bringt den SoC (System on Chip) des Raspi regelrecht zum Glühen und lastet das System vollständig aus - der Raspi ist nur noch damit beschäftigt, den Pin abzufragen. Ein naheliegender, aber sehr verpönter Weg ist es, mittels *time.sleep()* die Schleife zu bremsen. Doch dann reagiert der Raspi nur noch verzögert auf den Tastendruck. Die sinnvolle Lösung ist, dass sich der Raspi selbst ganz nebenbei um die Überwachung kümmert und das Programm bei jeder Zustandsänderung des Pins benachrichtigt. Dazu müssen Sie für den gewünschten Pin ein Event registrieren und eine Funktion bereitstellen, die aufgerufen wird, sobald sich an dem Pin etwas tut. Das folgende Beispiel fährt den Raspi herunter, wenn Sie die an Pin 5 angeschlossene Power-Taste drücken:

```
import RPi.GPIO as GPIO
import time
from subprocess import call
def buttonPress(pin):
    if not GPIO.input(5):
        print(" Shutdown")
        call(['shutdown', '-h',
             'now'], shell=False)
GPIO.setmode(GPIO.BOARD)
GPIO.setup(5, GPIO.IN)
GPIO.add_event_detect(
    5, GPIO.BOTH,
```

```
    callback=buttonPress
)
while True:
    time.sleep(5)
```

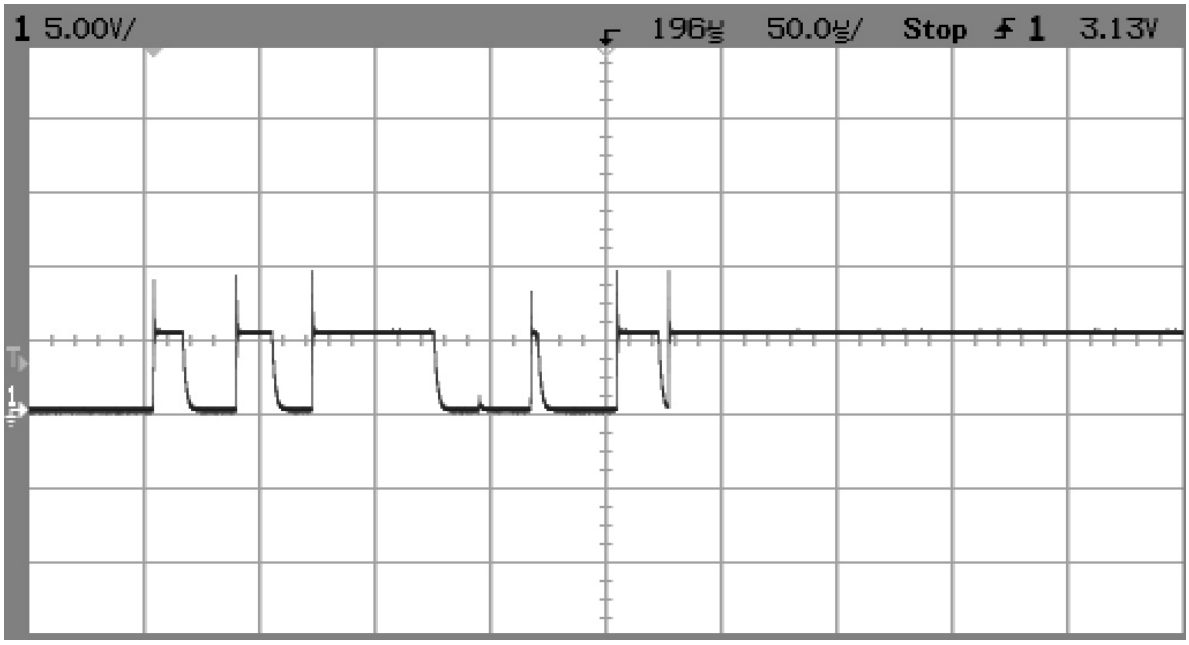
Der Aufruf der Funktion *GPIO.add_event_detect()* registriert durch den Parameter *GPIO.BOTH* die Funktion *buttonPress()* beim Event-Handler für den Fall, dass der Raspi an Pin 5 entweder einen fallenden (*GPIO.FALLING*) oder einen steigenden (*GPIO.RISING*) Spannungspegel erkennt. Dabei erhält *buttonPress()* die Nummer des Pins, an dem die Pegeländerung erkannt wurde. Wenn Sie dieses Beispielprogramm unter dem Namen pishutdown.py speichern und dann mit *sudo python3 pishutdown.py* aufrufen, können Sie Ihren Raspi herunterfahren und durch erneuten Tastendruck wieder aufwecken.

Abgeprallt

Ein generelles Problem von mechanischen Tastern und Schaltern ist das sogenannte Prellen: Der Schließkontakt überbrückt nicht sofort und dauerhaft die Ruhekontakte. Der bewegliche Schließer unterbricht und schließt die Kontakte mehrfach hintereinander, bis sich am Ende ein stabiler Zustand einstellt. Der Raspberry Pi fragt die GPIO-Pins so schnell ab, dass er diese Zustandswechsel als mehrfache schnelle Betätigung interpretiert. Das Prellen dauert üblicherweise nur wenige Millisekunden und kann schaltungstechnisch zum Beispiel dadurch verhindert werden, dass man einen Kondensator mit 10 bis 100 nF parallel zum Taster oder Schalter anlötet. Für die Funktion *add_event_detect()* gibt es den Parameter *bouncetime* , mit dem Sie angeben können, für wie viele Millisekunden der Raspi solch einen schnellen Zustandswechsel ignorieren soll - im nachfolgenden Beispiel 500 Millisekunden:

```
GPIO.add_event_detect(
    5, GPIO.BOTH,
    callback=buttonPress,
    bouncetime=500
)
```

Da wohl niemand den Power-Taster im Stakkato benutzt, ist eine *bouncetime* von einer halben Sekunde kein Problem. Bei anderen Anwendungen wie Benutzereingaben sollten Sie den Wert aber senken, etwa auf 100 Millisekunden - und dann zusätzlich einen Kondensator anschließen. Bei Tastern sollten Sie außerdem überlegen, ob Sie durch die Angabe von *GPIO.BOTH* wirklich auf beide Signalflanken reagieren wollen oder nicht nur auf eine. Dabei müssen Sie wiederum die Schaltlogik beachten: Bei Active Low bedeutet *GPIO.FALLING* , dass eine Taste gedrückt wird, und *GPIO.RISING* , dass Sie sie loslassen.



Mechanische Taster und Schalter stellen den Kontakt nicht sofort stabil her. Vielmehr gibt es beim Betätigen eine Reihe von Kontaktunterbrechungen, sogenanntes Prellen, das der Raspi als schnelles Umschalten interpretiert.

Event-Schleifen

Das vorangegangene Beispiel schummelt aber noch immer, schließlich enthält es am Ende eine *while* -Endlosschleife mit einem *sleep()* -Befehl. Die dadurch erzeugte Last ist zwar verschwindend gering, die saubere Lösung ist jedoch, die *while* -Schleife durch folgende Konstruktion zu ersetzen:

```
try:
    loop = asyncio.get_event_loop()
    loop.run_forever()
except:
    loop.close()
```

Am Anfang des Programms, das Sie vollständig auf ct.de/yuap zum Download finden, müssen Sie außerdem noch mit *import asyncio* die Bibliothek für asynchrone Ein-/Ausgabe laden. Diese Bibliothek ist der Schlüssel für eventgesteuerte Programmierung, bei der Funktionen asynchron etwa auf Prozessoren mit mehreren Kernen parallel ausgeführt werden können.

Typischerweise enthalten solche Programme lediglich einige Zeilen zur Initialisierung und unzählige Funktionen, die als Event-Handler dienen. Während der Initialisierung wird der Event-Loop angelegt, etwa mit der Funktion *get_event_loop()* und dann falls nötig Event-Handler registriert. Anschließend sorgt die Funktion *run_forever()* dafür, dass das Programm weiter läuft, selbst wenn aktuell keine Befehle auszuführen sind. Sollten später Aufgaben hinzukommen, so werden diese zum laufenden Event-Loop hinzugefügt und abgearbeitet, sobald sie an die Reihe kommen.

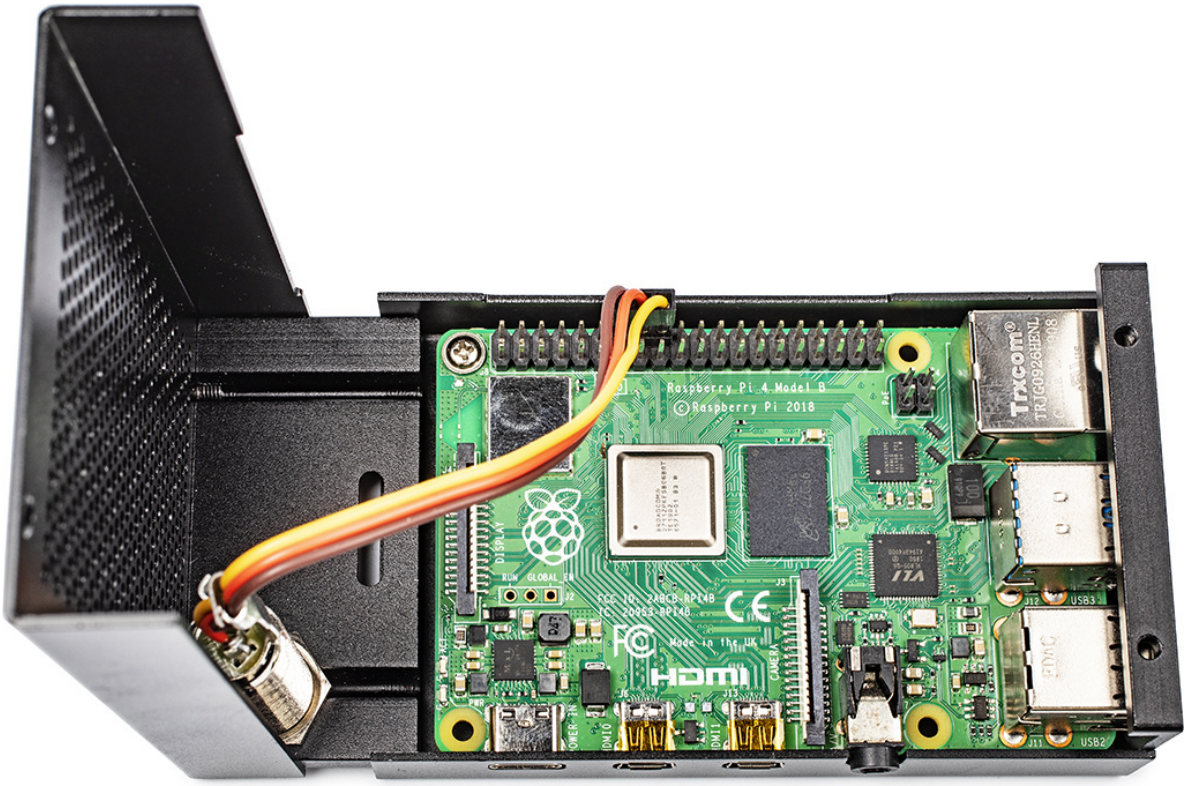
Vorhang auf

Viele moderne Python-Bibliotheken arbeiten asynchron, so auch die Bibliothek "simpleobsws", mit der Sie Befehle an den WebSocket-Server der Streaming-Software OBS Studio schicken können. Wenn Sie die Bibliothek mit den Befehlen

```
sudo apt-get install python3-pip
pip3 install simpleobsws
```

installiert haben, kann der Raspi Ihren Auftritt im Internet künftig mit einem Gitarrenfußschalter unterstützen, sodass Sie auf Tritt zwischen zwei Kameraeinstellungen um- oder Ihr Mikrofon stummschalten können.

Doppel-Gitarrenfußschalter wie der Lead Foot FS-2 für knapp 15 Euro haben einen 3-poligen 1/4-Zoll-Klinkenstecker, bei denen jeweils ein Schalter mit der Spitze respektive dem ersten Kontaktring verbunden ist, den Masseanschluss nutzen beide gemeinsam. Um den Stecker nicht abschneiden zu müssen, haben wir in ein GeeekPi Metal Case für ebenfalls knapp 15 Euro eine Stereo-Klinkenbuchse eingebaut. Das GeeekPi-Gehäuse ist dafür besonders gut geeignet, da die Seitenwand mit dem MicroSD-Slot ein massiver, vier Millimeter starker Aluminiumblock ist. Hier können Sie gut ein Loch für die Klinkenbuchse BKL 1109029 und ein Feingewinde M12 ? 1 mm bohren oder, falls Sie keinen passenden Gewindebohrer haben, die Buchse mit der beiliegenden Scheibe und Mutter verschrauben. Die Klinkenbuchse von BLK ist zwar mit rund 3 Euro teurer als andere, sie ist aber auch besonders robust und umschließt den Klinkenstecker auf voller Länge. Wenn Sie sie nach dem Einbau mit Schrumpfschlauch ummanteln, gibt es selbst dann keine Kurzschlüsse mit der Raspi-Platine, wenn Sie bei einem besonders beeindruckenden Auftritt versehentlich auf den Stecker oder den Raspi treten und das Gehäuse eindellen.



Die gut vier Millimeter dicke, massive Alu-Seitenwand des GeeekPi-Gehäuses eignet sich gut, um eine 1/4-Zoll-Klinkenbuchse zum Anschluss eines Gitarrenfußschalters stabil einzubauen.

Für den Anschluss benötigen Sie ein dreipoliges Kabel, ideal sind Servo-Kabel aus dem Modellbau, und drei GPIO-Pins: einmal Masse und zweimal einen der grün markierten Schaltpins. Wir haben uns für die Schaltpins 16 und 18 entschieden; ob Sie als Masseleitung Pin 14 oder 20 benutzen, bleibt Ihnen überlassen - so können Sie leicht durch Umdrehen des Servo-Kabels die Schalter 1 und 2 vertauschen.

Fußschalter für OBS Studio

```
import RPi.GPIO as GPIO
import asyncio
import simpleobsws
import logging
Switch1 = 16
Switch2 = 18
BounceTime = 500
ObsIP = '192.168.178.20'
ObsPort = 4444
ObsPass = '****'
SceneName = ['Welcome', 'Gameplay']
AudioSource = 'Mikrofon'
logging.basicConfig(level=logging.INFO)
loop = asyncio.get_event_loop()
async def toggleAudioMute():
    global ObsIP, ObsPort, ObsPass
    global AudioSource
    ws = simpleobsws.obsws(host=ObsIP, port=ObsPort, password=ObsPass)
    await ws.connect()
    result = await ws.call('GetSourcesList')
    if result.get('status') == 'ok':
        for src in result.get('sources'):
            if src.get('name') == AudioSource:
                break
    if src.get('name') != AudioSource:
        logging.error(" No audio source '%s'.", AudioSource)
        return
    data = {'source':AudioSource}
    result = await ws.call('ToggleMute', data)
```

```
    if result.get('status') == 'ok':
        logging.info(" Toggled mute on '%s'.", AudioSource)
    await ws.disconnect()
async def switchToScene(SceneNr):
    global ObsIP, ObsPort, ObsPass
    global SceneName
    ws = simpleobsws.obsws(host=ObsIP, port=ObsPort, password=ObsPass)
    await ws.connect()
    result = await ws.call('GetCurrentScene')
    if result.get('name') == SceneName[SceneNr]:
        logging.info(" Scene '%s' already active.", SceneName[SceneNr])
        return
    data = {'scene-name':SceneName[SceneNr]}
    result = await ws.call('SetCurrentScene', data)
    if result.get('status') == 'ok':
        logging.info(" Switched to scene '%s'.", SceneName[SceneNr])
    await ws.disconnect()
def buttonPress(pin):
    global loop
    if loop is None:
        return
    state = GPIO.input(pin)
    if pin == Switch1:
        asyncio.run_coroutine_threadsafe(switchToScene(state), loop)
    if pin == Switch2:
        asyncio.run_coroutine_threadsafe(toggleAudioMute(), loop)
try:
    GPIO.setmode(GPIO.BOARD)
    GPIO.setup(Switch1, GPIO.IN, pull_up_down=GPIO.PUD_UP)
    GPIO.setup(Switch2, GPIO.IN, pull_up_down=GPIO.PUD_UP)
    GPIO.add_event_detect(Switch1, GPIO.BOTH,
        callback=buttonPress, bouncetime=BounceTime)
    GPIO.add_event_detect(Switch2, GPIO.BOTH,
        callback=buttonPress, bouncetime=BounceTime)
    state = GPIO.input(Switch1)
    loop.run_until_complete(switchToScene(state))
    loop.run_forever()
except:
    loop.close()
    GPIO.cleanup()
```

Welche Pins Sie als Schalt-Pins benutzen, ist im Listing auf Seite 159 (Download auf ct.de/yuap) in den globalen Variablen *Switch1* und *Switch2* (Zeile 6 und 7) festgelegt. Gleich darunter steht die *BounceTime* zum Entprellen. Die IP-Adresse Ihres Streaming-Rechners und das Passwort für den WebSocket-Server von OBS stehen gleich darunter.

Die Initialisierung der GPIO-Schnittstelle erfolgt am Ende des Listings in den Zeilen 64 bis 70. Der größte Unterschied zu den bisherigen Beispielen ist, dass *GPIO.add_event_detect()* zweimal hintereinander aufgerufen wird, einmal für *Switch1* und einmal für *Switch2* . Als Event-Handler dient in beiden Fällen die Funktion *buttonPress()* ab Zeile 53.

Direkt nach der GPIO-Initialisierung wird die Funktion *switchToScene()* aufgerufen: Damit ist sichergestellt, dass OBS Studio auf jene Kameraszene umschaltet, die vom Fußschalter gerade ausgewählt ist. Da es sich um eine asynchrone Funktion handelt, zu erkennen am Prefix *async def* in Zeile 38, erfolgt der Aufruf in Zeile 72 mittels *run_until_complete()* aus dem Event-Loop heraus. Anschließend startet in Zeile 73 die Endlosschleife.

Asynchronisiert

Die GPIO-Bibliothek des Raspi ist allerdings noch nicht auf asynchrone Arbeitsweise ausgelegt, folglich darf die beim Event-Handler registrierte Funktion *buttonPress()* in Zeile 53 auch nicht als asynchron definiert sein. Die Funktionen *switchToScene()* und *toggleAudioMute()* hingegen nutzen etliche asynchrone Funktionen der WebSocket-Bibliothek von OBS Studio, weshalb sie in den Zeilen 19 und 38 als asynchron definiert sind. Nicht-asynchrone Funktionen können jedoch nicht ohne Weiteres asynchrone aufrufen - die Lösung ist, die Funktion *asyncio.run_coroutine_threadsafe()* zu benutzen, damit sie in der Event-Loop *loop* ausgeführt, aber nicht mittendrin von einem konkurrierenden Aufruf unterbrochen werden.

Die Arbeitsweise der beiden asynchronen Funktionen *toggleAudioMute()* und *switchToScene()* ist sehr ähnlich: Sie nehmen Kontakt zu OBS Studio auf dem Streaming-Rechner auf, überprüfen, was der aktuelle Zustand ist, senden dann falls nötig den Befehl und schließen die Verbindung wieder.

Der Verbindungsaufbau zum WebSocket-Server erfolgt in den Zeilen 22 und 41, indem eine neue Instanz des WebSocket-Objekts erzeugt wird. IP-Adresse, Port und Passwort sind dabei die notwendigen Parameter. In Zeile 24 ruft *toggleAudioMute()* mittels *ws.call('GetSourcesList')* die Liste der Audioquellen ab, die in der gerade aktiven Szenensammlung von OBS Studio gibt, und prüft für jede, ob es sich um die gesuchte handelt. Falls ja, bricht die Schleife ab - dann enthält die Variable *src* den Namen der gesuchten Audioquelle. Gibt es die gesuchte Audioquelle nicht, so läuft die Schleife in Zeile 26 bis zum letzten Element durch und der Namensvergleich in Zeile 29 schlägt fehl.

Hat *toggleAudioMute()* die richtige Audioquelle gefunden, bettet sie deren Namen in Zeile 32 in ein Dictionary ein und ruft in Zeile 33 den Befehl *ws.call('ToggleMute', data)* mit dem Dictionary als zweiten Parameter auf. Die Zeile 34 überprüft, ob OBS Studio den Befehl akzeptiert und verarbeitet hat, in Zeile 35 wird dann die Verbindung zum WebSocket-Server geschlossen.

Die Funktion *switchToScene()* ruft in Zeile 44 mittels *ws.call('GetCurrentScene')* ab, welche Szene aktuell in OBS Studio läuft. Da Sie die Szenen bereits per Maus in OBS umgeschaltet haben könnten, ist dieser Schritt erforderlich. Nur dann, wenn die zu aktivierende Szene eine andere ist (Zeile 44), sendet *switchToScene()* wiederum per *ws.call()* in Zeile 48 den Umschaltbefehl an OBS Studio. Die Zeile 49 ermittelt, ob OBS den Befehl akzeptiert hat, anschließend wird die Verbindung getrennt.

Anstatt die Szene zu wechseln oder das Mikrofon stummzuschalten, könnten Sie auf Tritt zum Beispiel auch den Stream starten oder beenden. Welche Befehle der WebSocket-Server von OBS Studio unterstützt, finden Sie in der Protokollspezifikation (Link siehe ct.de/yuap).

Fazit

Mit der GPIO-Bibliothek und wenigen Zeilen Python-Code wird der Raspberry Pi zur universellen Schaltzentrale. Dank umfangreicher Zusatzbibliotheken sind Sie nicht darauf beschränkt, etwa nur eine LED blinken zu lassen, sondern können sogar Schaltaufgaben in Drittprogrammen wie OBS Studio auf anderen Rechnern übernehmen. So wird der Raspi zur universellen Netzwerk-Fernbedienung, die Sie für den perfekten Auftritt individuell anpassen können.

Listings und Dokumentation: ct.de/yuap

Mirko Dölle

Quelle:	c't Heft 3/2022 S. 154-159
ISSN:	0724-8679
Ressort:	Praxis
Rubrik:	Raspi-GPIO in Python
Dokumentnummer:	2131315255726389454