c't Heft 6/2022 S. 24-29 / Titel - Raspberry Pi: Sensoren und Aktoren

Bitgebastel

Raspi: Servos und Sensoren ansteuern

Der GPIO-Anschluss des Raspberry Pi kann mehr, als nur LEDs zum Leuchten bringen. Schon mit wenigen Zeilen Python können Sie Ultraschallsensoren einbinden, Signallaufzeiten messen oder Servos und andere Elektronikkomponenten ansteuern. Denn genau dafür wurde der Mini-Computer gemacht.

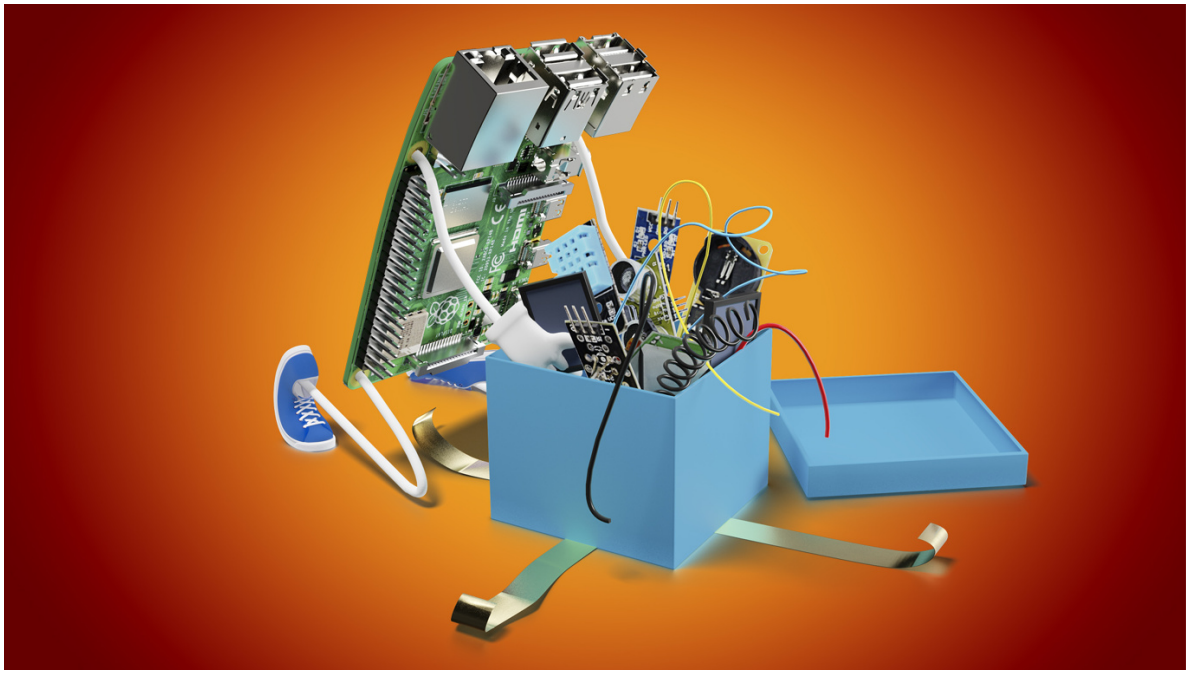


Bild: Andreas Martini

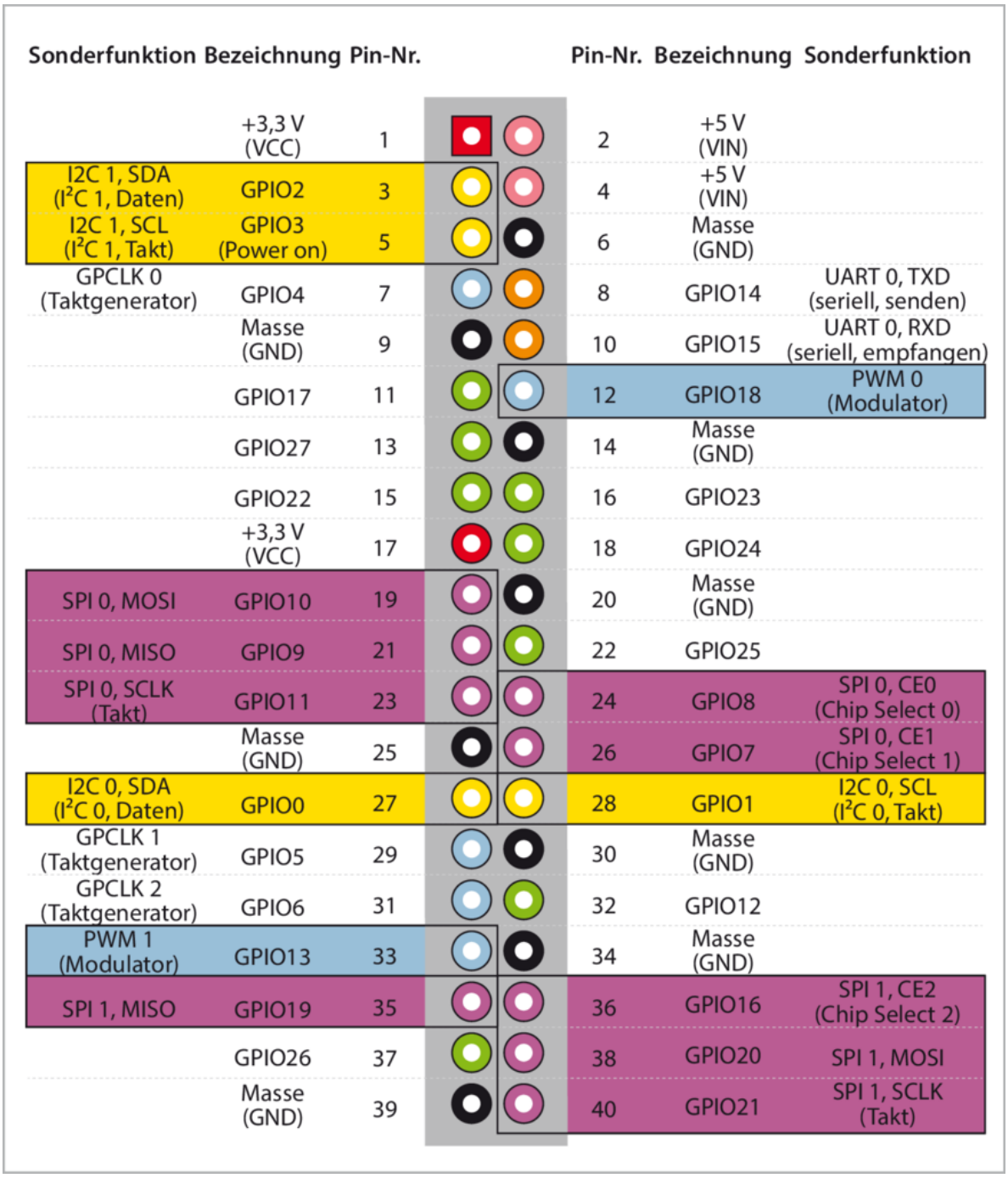
Zum Basteln ist er da: Mit dem Raspberry Pi entwarf die gleichnamige Foundation einen günstigen Mini-Computer als Lernplattform, um mehr Menschen in aller Welt an die Computertechnik heranzuführen. Der ursprünglich 26-polige GPIO-Anschluss, der später auf 40 Pins erweitert wurde, bietet zahlreiche Anschlussmöglichkeiten für zusätzliche Hardware: LEDs, Taster und Schalter, aber auch komplexe Komponenten wie Sensoren, Displays oder RFID-Leser. Dazu haben viele der insgesamt 29 "Allzweck"-(General-Purpose-/GP-)Schaltpins Zusatzfunktionen, etwa die Pins 3 und 5, die auch als I2C-Bus (Inter-Integrated Circuit) fungieren, oder die Pins 35, 36, 38 und 40, die als zweiter SPI-Bus (Serial Peripheral Interface) dienen.

Weil Hardware-Bus-Controller Geld respektive Platz auf dem System on Chip (SoC) des Raspi kosten, gibt es davon nur wenige - beim Ur-Raspi mit nur 26 Pins waren jeweils nur ein I2C- und ein SPI-Bus über den GPIO-Anschluss verfügbar. Erst mit der Erweiterung auf 40 Pins kamen ein weiterer I2C- und SPI-Bus hinzu, die jedoch für Aufsteckplatinen alias Hats gedacht sind. Um manche andere Bussysteme wie zum Beispiel CAN-Bus zu nutzen, benötigen Sie zusätzliche Controllerchips oder teure Hats. Für einfachere Bus-Systeme wie den 1-Wire-Bus, der zum Beispiel für Temperatur- oder Feuchtigkeitssensoren benutzt wird, ist das nicht erforderlich. Der Raspi ist leistungsfähig genug, um das jeweilige Busprotokoll in Software nachzubilden. Fachleute nennen das "Bit Banging".

Bit für Bit

Beim Bit Banging wird ein Schaltausgang in den richtigen Zeitabständen ein- und ausgeschaltet oder an einem Eingang die Zeit gemessen, die zwischen

Pegeländerungen liegt. Beim 1-Wire-Bus übernimmt das Modul *w1-gpio* des Linux-Kernels die Aufgabe, standardmäßig den Pin 7 des Raspi mit dem richtigen Timing ein- und auszuschalten, und belastet damit die CPU. Sie müssen allerdings zuvor den 1-Wire-Bus auf der Kommandozeile mit Befehl *raspi-config* oder über die grafische Raspi-Konfiguration aktivieren und den Raspi neu starten.



Der Raspberry Pi verfügt lediglich über zwei I2C- und SPI-Busse sowie zwei PWM-Modulatoren. Andere Bussysteme wie 1-Wire müssen per Bit Banging in Software implementiert werden.

Gibt es kein fertiges Kernel-Modul für ein bestimmtes Protokoll, so können Sie dieses per Bit Banging selbst implementieren. Für die Programmierung der GPIO-Anschlüsse eignet sich die Programmiersprache Python gut, die in diesem Artikel vorgestellten Beispiele sind allesamt auf Python Version 3 ausgelegt. Mit der unter Raspberry Pi OS vorinstallierten Bibliothek RPi.GPIO haben Sie direkten Zugriff auf die einzelnen Pins. In der Python-Einführung aus [1] haben wir gezeigt, wie Sie schon mit wenigen Zeilen Python-Code eine LED zum Blinken bringen - das war bereits eine sehr einfache Form des Bit Banging.

Da ein Python-Programm im Userspace ausgeführt wird und nicht wie ein Kernel-Modul im Kernel-Space, sollte das Timing des Protokolls nicht zu anspruchsvoll sein - schließlich kann das Python-Programm jederzeit von privilegierten Prozessen unterbrochen werden. Das kann im Extremfall zu so großen zeitlichen Abweichungen

(Jitter) führen, dass die Kommunikation zwischen Raspi und der angesteuerten Komponente zusammenbricht. Bit Banging per Python eignet sich vor allem für niedrige Frequenzen und geringe Datenraten.

Als Beispiel für ein per Bit Banging in Python implementiertes Protokoll dient der Ultraschall-Entfernungsmesser HC-SR04, den Sie bei verschiedenen Elektronik-Shops für unter 5 Euro bekommen. Das Modul arbeitet autark, Sie müssen sich also nicht darum kümmern, einen Ultraschallton zu erzeugen. Stattdessen müssen Sie gemäß dem Protokoll, das in den Datenblättern aufgeführt ist, über den Trigger-Pin eine Messung anstoßen und erhalten dann am Echo-Pin ein Signal zurück, anhand dessen Sie die Entfernung ablesen können.

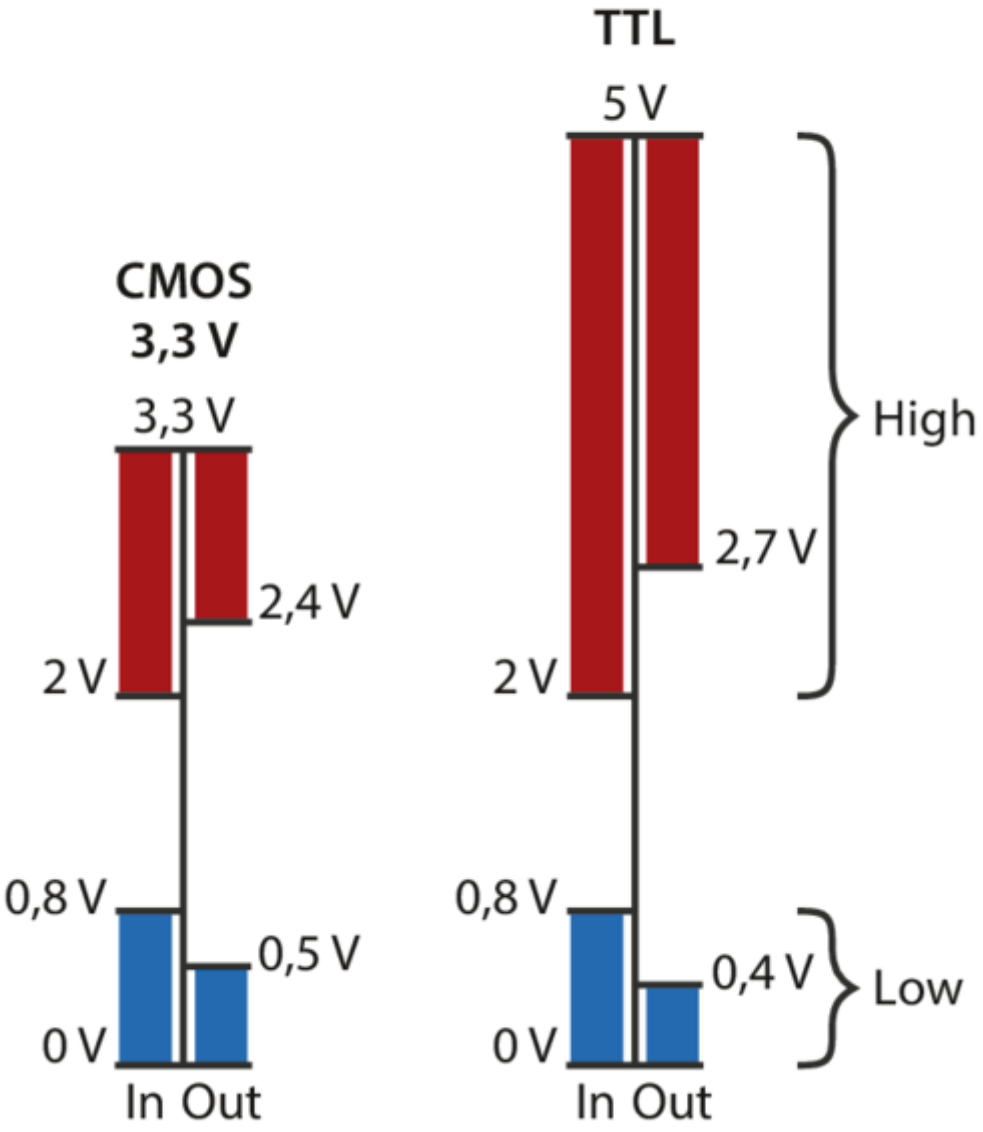
Hoch spannend

Dieser Sensor wird unter anderem auch in Arduino-Projekten eingesetzt, was ein wichtiger Hinweis ist: Der Arduino Uno und viele andere Arduino-Boards arbeiten mit TTL-Pegeln (Transistor-Transistor-Logic), die bis zu 5 Volt betragen. Zu viel für den Raspberry Pi, dessen Eingänge nur 3,3 Volt verkraften. Wenn Sie also eine Komponente wie einen Sensor oder ein Display aus einem Arduino-Projekt für den Raspi verwenden, müssen Sie prüfen, ob es davon eventuell eine Version mit 3,3 Volt gibt oder die Logik-Pegel an den Raspi anpassen. Dafür gibt es Pegelwandler als fertige Baugruppen, doch manchmal geht es auch einfacher.

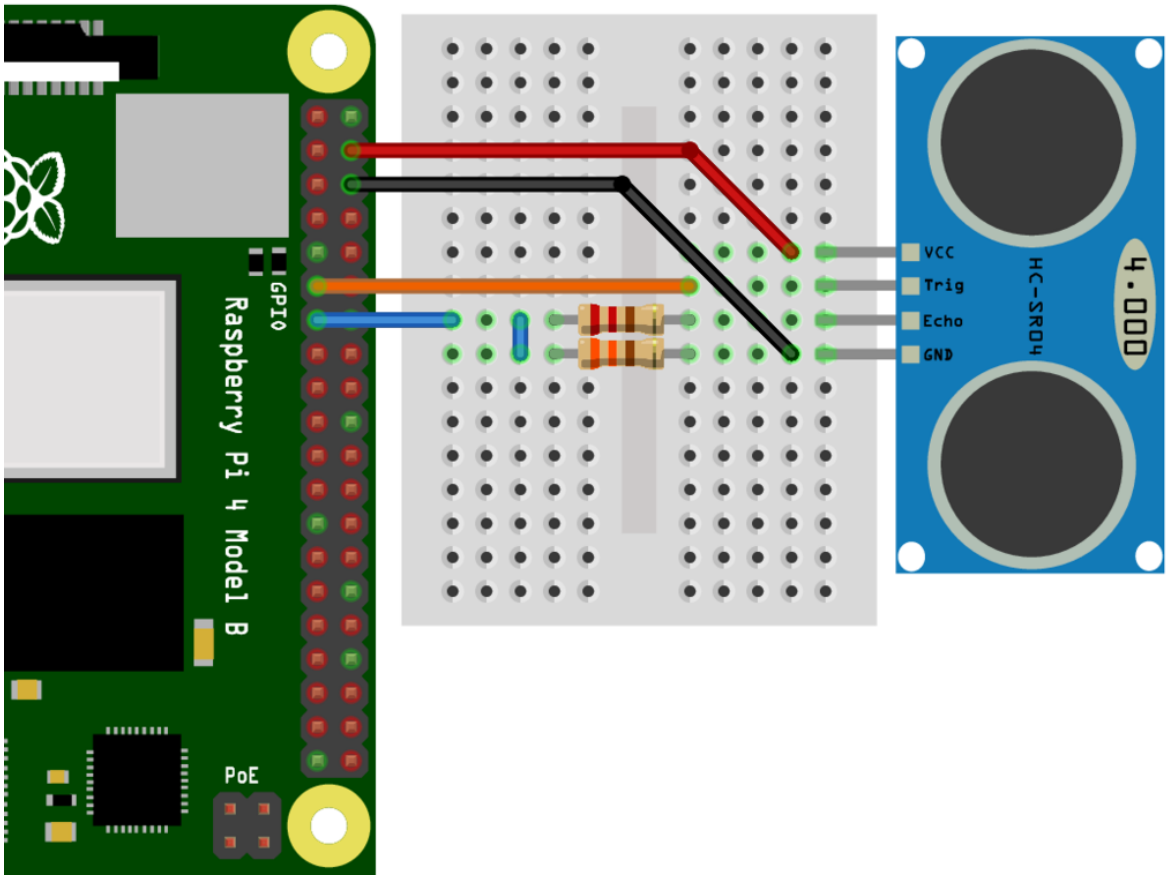
Beim Ultraschall-Entfernungsmesser HC-SR04 genügen zwei Widerstände für die Anpassung an den Raspi. Dabei können Sie ausnutzen, dass High- und Low-Pegel über eine gewisse Bandbreite erkannt werden. Bei TTL etwa wird an einem Eingang ein Pegel von 0 bis 0,8 Volt als Low-Signal erkannt, Spannungen von 2 bis 5 Volt als High-Signal. Bei Werten dazwischen ist nicht definiert, ob sie von einer Schaltung als High- oder als Low-Signal erkannt oder idealerweise als Störsignal interpretiert und ignoriert wird, der Störabstand beträgt an TTL-Eingängen also 1,2 Volt. An TTL-Ausgängen ist der Störabstand noch größer, ein Low-Signal hat hier höchstens 0,4 Volt und ein High-Signal mindestens 2,7 Volt. Die CMOS-Logikpegel liegen bei ebenfalls 0 bis 0,8 Volt am Eingang für ein Low-Signal und 2 bis 3,3 Volt für ein High-Signal, die Ausgänge liefern 0 bis 0,5 Volt bei einem Low-Signal und 2,4 bis 3,3 Volt für ein High-Signal.

Logikpegel

Viele Komponenten und Module etwa aus Arduino-Projekten arbeiten mit TTL-Pegeln. Der Raspberry Pi verträgt allerdings nur CMOS-Logikpegel bis maximal 3,3 Volt, weshalb an Eingängen ein Spannungsteiler erforderlich ist. Die Spannung an den Ausgängen des Raspi hingegen genügt, um auch TTL-Komponenten direkt anzusteuern.



Damit genügt das 3,3-Volt-Signal am Ausgang eines Raspi auch, um einen Sensor mit TTL-Eingang wie den HC-SR04 anzusteuern. Eine Anpassung für den Trigger-Eingang des Ultraschallsensors ist also unnötig. Am Echo-Ausgang hingegen benötigen Sie einen Spannungsteiler etwa aus einem 220- und einem 330-Ohm-Widerstand, der den TTL-Pegel von 5 Volt auf Raspi-freundliche 3,3 Volt herunterteilt. Dabei sind die konkreten Widerstandswerte nicht entscheidend, Sie könnten genauso gut einen 2,2-kOhm- und einen 3,3-kOhm- oder einen 5,1-kOhm- und einen 10-kOhm-Widerstand verwenden. Wichtig ist nur, dass die Widerstandswerte im Verhältnis von 1 zu 2 stehen, denn dann fallen ein Drittel der Spannung am kleineren und zwei Drittel am größeren Widerstand ab - womit der Signalpegel von 5 Volt auf 3,3 Volt am größeren der beiden Widerstände sinkt. Da der Raspi ein High-Signal auch noch bei 2,5 Volt erkennt, könnten Sie sogar zwei gleiche Widerstände als Spannungsteiler benutzen.



Der Ultraschall-Entfernungsmesser HC-SR04 arbeitet mit TTL-Pegeln, die die Eingänge des Raspberry Pi nicht vertragen. Deshalb müssen Sie die Spannung am Echo-Ausgang mit einem Spannungsteiler aus einem 220- und einem 330-Ohm-Widerstand auf 3,3 Volt reduzieren.

Welche GPIO-Pins des Raspi Sie für Trigger und Echo benutzen, steht Ihnen grundsätzlich frei. Sie sollten für solche einfachen Aufgaben aber keine höherwertigen Anschlüsse von Bussystemen blockieren. Die Pins 11 als Schaltausgang für Trigger und 13 als Eingang für das Echo eignen sich prima für den Ultraschallsensor. Um flexibel zu bleiben und den Python-Code leicht verständlich zu halten, tragen Sie die Pin-Nummern in Variablen ein und initialisieren damit die Pins:

```
import RPi.GPIO as GPIO
from time import sleep, time
triggerPin = 11
echoPin = 13
GPIO.setmode(GPIO.BOARD)
GPIO.setup(triggerPin, GPIO.OUT,
           initial=GPIO.HIGH)
GPIO.setup(echoPin, GPIO.IN)
```

Sie müssen den Code nicht abtippen, wie immer finden Sie alle Beispiele auf ct.de/yyp7

zum Download. Mit *initial=GPIO.HIGH* in der vorletzten Zeile legen Sie fest, dass der Trigger-Ausgang bei der Initialisierung auf High-Pegel gelegt wird. Prinzipiell ist es eine gute Idee, Ausgänge bei der Initialisierung auf einen definierten Pegel zu legen, ob High oder Low hängt aber ganz von der Anwendung ab. Der Entfernungsmesser wird getriggert, indem man die Leitung kurzzeitig auf Masse zieht - daher ist es sinnvoll, dass der Pin ansonsten auf High verbleibt und deshalb auch mit High-Pegel initialisiert wird.

Dauerlauf

Um den Raspi als digitalen Zollstock einzusetzen, soll die Messung im Sekunden-Intervall wiederholt werden. Dazu eignet sich eine *while* -Schleife mit Fehlerbehandlung, die Sie mit der Tastenkombination Strg+C jederzeit unterbrechen dürfen:

```
try:
  while True:
    print(" Entfernung: %.1f cm"
          % Dist())
    sleep(1)
except KeyboardInterrupt:
  GPIO.cleanup()
```

Python strukturiert den Code durch Einrückungen, Standard sind vier Leerzeichen pro Ebene. Damit die Listings nicht ausufern, verwenden wir in den Beispielen nur zwei Leerzeichen pro Ebene, was ebenfalls erlaubt ist. Drücken Sie später im laufenden Programm Strg+C, so erkennt die Fehlerbehandlung das Signal *KeyboardInterrupt* und schließt den GPIO-Anschluss des Raspi ordnungsgemäß, bevor das Programm endet. Für die Entfernungsmessung selbst ist die Funktion *Dist()* zuständig:

```
def Dist():
  GPIO.output(triggerPin, GPIO.LOW)
  sleep(0.000010)
  GPIO.output(triggerPin, GPIO.HIGH)
  while GPIO.input(echoPin) == 0:
    t0 = time()
  while GPIO.input(echoPin) == 1:
    t1 = time()
  return (t1-t0)*34300/2
```

Die ersten drei Zeilen der Funktion ziehen den Trigger-Pin gemäß der Spezifikationen des HC-SR04 für mindestens zehn Mikrosekunden auf Low-Pegel, bevor er für die nächste Messung wieder auf High-Pegel zurückgesetzt wird. Voraussetzung ist, dass der Pin schon zuvor auf High lag - was die Initialisierung sichergestellt hat.

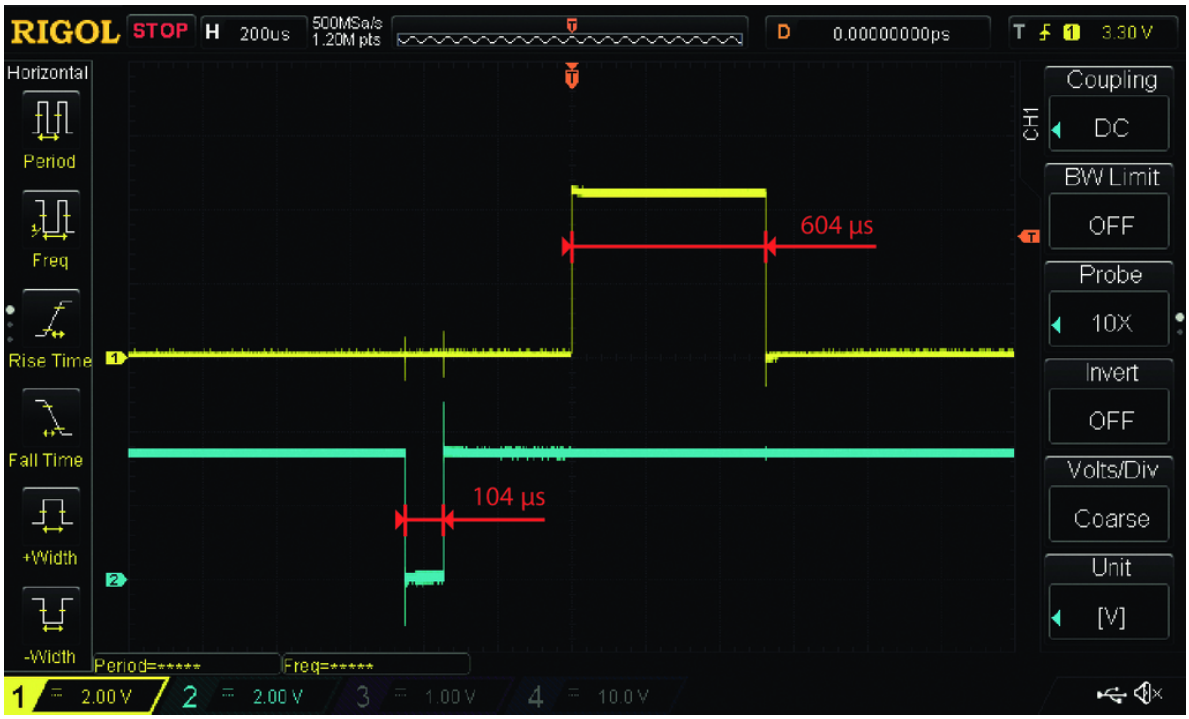
Der Rest der Funktion wertet die Antwort des Ultraschall-Entfernungsmessers auf dem Echo-Pin aus. Dieser hält die Echo-Leitung auf Low-Pegel, bis das Ultraschallsignal ausgesendet wird, und zieht es dann so lange auf High, bis das Echo empfangen wurde. Den Beginn der Messung signalisiert also der Wechsel von Low auf High, das Ende der Messung umgekehrt der Wechsel von High auf Low.

Dementsprechend aktualisiert die erste *while* -Schleife den Start-Zeitpunkt so lange, wie das Echo-Pin noch auf Low liegt. Die Schleife endet genau dann, wenn die Messung beginnt; in der Variablen *t0* steht somit der Zeitpunkt knapp vor Beginn der Entfernungsmessung. Die zweite *while* -Schleife verwendet die umgekehrte Logik, sie aktualisiert die Variable *t1* , solange die Messung läuft. Ist die Messung beendet, endet auch die Schleife und in *t1* steht der letzte Zeitpunkt, an dem die Messung noch lief.

Aus der Differenz von *t1* und *t0* lässt sich nun die Laufzeit des Ultraschallsignals berechnen. Multipliziert man das Ergebnis mit der Schallgeschwindigkeit von 34.300 Zentimetern pro Sekunde, erhält man die gesamte vom Ultraschallsignal zurückgelegte Strecke von Sender zum reflektierenden Objekt und zurück zum Empfänger. Das Objekt befindet somit sich auf halber Distanz.

Maß genommen

Speichern Sie das Listing unter dem Namen zollstock.py oder laden Sie es über ct.de/yyp7 herunter und starten Sie es mit dem Befehl *python3 zollstock.py* . Wie das vom Raspi erzeugte Trigger-Signal und das Echo des Moduls in Wirklichkeit aussehen, zeigt unsere Oszilloskopaufnahme: Das in Türkis dargestellte Trigger-Signal hat eine Breite von gut 100 Mikrosekunden statt der programmierten 10 Mikrosekunden - dafür ist der Raspi offenkundig nicht schnell genug. Da das Ultraschallmodul lediglich eine Mindest-Signalbreite von 10 Mikrosekunden verlangt, es aber auch mehr sein dürfen, wird die Messung trotzdem angestoßen.



Für ein Trigger-Signal (Türkis) von nur zehn Mikrosekunden ist der Raspberry Pi nicht schnell genug; 100 Mikrosekunden sind dem Ultraschall-Entfernungsmesser aber auch recht. Die Antwort, ein 600 Millisekunden breites Rechtecksignal (Gelb), zeigt ein Objekt in zehn Zentimetern Entfernung an.

Rund 300 Mikrosekunden nach Ende des Trigger-Signals beginnt das Modul damit, die Ultraschalltöne auszusenden und setzt deshalb den Echo-Pin auf High (gelber Signalverlauf). Die horizontale Auflösung des Oszilloskops beträgt 200 Mikrosekunden pro Kästchen ("H 200us" links oben). Nach 3 Kästchen oder umgerechnet 600 Mikrosekunden fällt der Echo-Pin wieder auf Low zurück - das Echo wurde empfangen, die Messung ist beendet. Daraus ergibt sich eine Entfernung von rund zehn Zentimetern zum gemessenen Objekt.

Wichtig ist, dass die Verkabelung stimmt: Haben Sie das Modul falsch angeschlossen, blockiert das Programm in einer der beiden *while* -Endlosschleifen der Funktion *Dist()* und lastet den Raspi schlimmstenfalls voll aus. Ein entlastendes *sleep()* fehlt mit voller Absicht in beiden *while* -Schleifen; damit die Messung möglichst präzise ist, soll der Raspi so oft wie möglich den Status des Echo-Pins überprüfen und die Zeiten aktualisieren.

Weil die *while* -Schleifen zur Messung der Signalbreite in einem Python-Anwendungsprogramm im sogenannten Userspace laufen, müssen sie sich die Ressourcen des Raspi mit anderen Anwendungen sowie den Aufgaben des ohnehin privilegierten Kernels teilen. Bei einem stark belasteten System kann das zu Abweichungen im Zentimeter-Bereich führen. Präziser messen Sie, indem Sie den Raspi selbst den Echo-Pin überwachen lassen und lediglich bei einer Pegeländerung von Low nach High oder High nach Low *t0* respektive *t1* erfassen. Dazu fügen Sie unmittelbar nach der Initialisierung des Echo-Pins folgende Zeile ein:

```
GPIO.add_event_detect(echoPin,
     GPIO.BOTH, callback=echoEvent)
```

Damit startet ein sogenannter Event-Handler die Funktion *echoEvent()* , sobald der Raspi eine ansteigende oder fallende Signalflanke (*GPIO.BOTH*) am Echo-Pin erkennt - also sobald das Ultraschallmodul den Beginn oder das Ende einer Messung signalisiert:

```
t0 = None
t1 = None
def echoEvent(pin):
  global t0, t1
  if (GPIO.input(pin)
     and not t0):
    t0 = time()
  elif (not GPIO.input(pin)
       and t0
       and not t1):
    t1 = time()
    print(" Entfernung: %.1f cm"
          % ((t1-t0)
            *34300/2))
    t0 = None
    t1 = None
```

Der Event-Handler ruft die Funktion mit der Nummer des betreffenden Pins als Parameter auf. Bei steigender Flanke wurde eine neue Messung gestartet, bei fallender eine laufende beendet. Diese beiden Fälle lassen sich leicht unterscheiden, indem Sie den aktuellen Pegel des Pins überprüfen: Ist der Pin High, so wurde die Funktion wegen einer steigenden Signalflanke am Anfang einer Messung aufgerufen und Sie speichern die Zeit in der globalen Variable *t0* . Ist der Pegel des Pins Low, war es die fallende Signalflanke am Ende der Messung, dann landet die aktuelle Zeit in der Variablen *t1*

Stückwerk

Doch es ist nicht garantiert, dass die Funktion *echoEvent()* am Stück abgearbeitet wird, denn der Event-Handler ist währenddessen weiter scharf geschaltet. Sollte der Raspi, während *echoEvent()* gerade ausgeführt wird, wieder eine Signalflanke entdecken, so würde er die aktuell laufende Funktion unterbrechen und sie neu starten - wodurch die Variablen *t0* oder *t1* überschrieben würden. Deshalb prüft die Funktion, ob bereits eine t0 gesetzt ist - falls ja, läuft bereits eine Messung und die steigende Signalflanke wird ignoriert.

Bei einer fallenden Signalflanke, die das Ende einer Messung anzeigt, muss zuvor eine t0 gesetzt worden sein - *t1* hingegen muss leer sein, denn andernfalls läuft noch die Auswertung einer früheren Messung. Erst wenn das Ergebnis der früheren Messung angezeigt wurde, leert *echoEvent()* die Variablen *t0* und *t1* , wonach die nächste Messung angestoßen werden kann. Das geschieht über eine verkürzte Variante der Funktion *Dist()* , die nur noch das Trigger-Signal sendet, aus der bekannten *while* - Schleife heraus:

```
def Dist():
  global triggerPin
  GPIO.output(triggerPin, GPIO.LOW)
  sleep(0.000010)
  GPIO.output(triggerPin, GPIO.HIGH)
try:
  while True:
    Dist()
    sleep(1)
except KeyboardInterrupt:
  GPIO.cleanup()
```

Servo unter Kontrolle

Auch Modellbau-Servos und Fahrtenregler für Motoren kann der Raspberry Pi per Bit

Banging direkt ansteuern, Sie benötigen dafür im Idealfall nicht einmal zusätzliche Elektronikkomponenten. Gerade Miniatur-Servos arbeiten oft schon bei 3,3 Volt Versorgungsspannung, die Sie an Pin 1 des Raspi abgreifen und am roten Servo-Kabel einspeisen können. Ihre volle Kraft entfalten Servos aber erst bei 5 Volt, die an den Pins 2 und 4 des Raspi anliegen - Servos mit größeren Stellkräften und besonders schnelle Digital-Servos sollten Sie aber besser über ein externes 5-Volt-Netzteil versorgen, da diese sehr hohe Spitzenströme ziehen. Der Masseanschluss (Braun oder Schwarz) eines Servos gehört in jedem Fall, gegebenenfalls zusammen mit dem Masseanschluss des externen Netzteils, an Pin 6 des Raspi oder an einen der anderen Masseanschlüsse. Die Steuerleitung (Gelb oder Weiß) können Sie mit einem beliebigen GPIO-Ausgang verbinden, zum Beispiel mit Pin 11. Die maximal 3,3 Volt der Raspi-Schaltausgänge genügen den meisten Servos.

Die Ansteuerung von Modellbau-Servos und Motorreglern erfolgt über Rechtecksignale mit definierter Pulsbreite und Pulsabständen. Bei einer Pulsbreite von 1,5 Millisekunden dreht sich der Servo in die Mittelstellung. Verkürzt man die Pulsbreite auf nur 0,5 Millisekunden, so dreht sich der Servo um 90 Grad in die eine Richtung, verlängert man die Pulsbreite auf 2,5 Millisekunden, dreht er sich um 90 Grad in die andere Richtung - jeweils von der Mittelstellung aus betrachtet. Bei Reglern bestimmt die Pulsbreite die Geschwindigkeit und die Drehrichtung des Motors, 1,5 Millisekunden Pulsbreite bedeuten hier Stopp. Ein Servo hält die zuvor eingenommene Position so lange, wie sich das Signal mindestens alle 25 Millisekunden (40 Hz) wiederholt - die meisten Servos unterstützen aber auch kürzere Abstände bis hinunter zu 5 Millisekunden (200 Hz), wodurch der Servo deutlich schneller reagieren kann.

Der Bereich für die Pulsbreite von 0,5 bis 2,5 Millisekunden bleibt allerdings stets gleich, unabhängig von der Pulshäufigkeit, also der Pulsfrequenz. Um einen Servo mit dem Raspi in die Mittelstellung zu bringen, müssen Sie lediglich den GPIO-Pin für 1,5 Millisekunden auf High schalten und anschließend zum Beispiel für 9 Millisekunden auf Low. Das erledigt folgendes Programm:

```
import RPi.GPIO as GPIO
from time import sleep
servopin = 11
GPIO.setmode(GPIO.BOARD)
GPIO.setup(servopin, GPIO.OUT,
           initial=GPIO.LOW)
for i in range(2):
  GPIO.output(servopin, GPIO.HIGH)
  sleep(0.0015)
  GPIO.output(servopin, GPIO.LOW)
  sleep(0.009)
GPIO.cleanup()
```

Da der GPIO-Pin mit einem Low-Signal initialisiert wurde, genügen bestenfalls zwei Pulse, also zwei Durchläufe (*range(2)*) der *for* -Schleife, um den Servo in die Mittelstellung zu bringen. Da es in der Praxis gelegentlich zu Störungen kommt, empfehlen wir mindestens fünf Schleifendurchläufe. Damit der Servo die Position auch dann hält, wenn beispielsweise die Kraft eines Ruders auf den Servoarm einwirkt, müssen Sie das Pulssignal hingegen ständig wiederholen - etwa mit vom Ultraschall-Entfernungsmesser bekannten *while* -Endlosschleife mit Abbruchmöglichkeit per Strg+C anstelle der *for* -Schleife:

```
try:
  while True:
    ...
except KeyboardInterrupt:
  GPIO.cleanup()
```

Der Raspberry Pi ist schnell genug, um so kurze Schlafzyklen von 1,5 Millisekunden einigermaßen genau einhalten zu können: Wie die Abbildung zeigt, hat das

Rechtecksignal tatsächlich eine Breite von 1,58 Millisekunden; die horizontale Auflösung des Oszilloskops beträgt 1 Millisekunde pro Kästchen ("H 1.00ms" links oben). Auch die Wiederholung weicht mit 10,16 Millisekunden nur geringfügig vom Soll ab.

Wie präzise das Timing des Servo-Signals ist, hängt beim Bit Banging wesentlich von der Systemauslastung des Raspi ab. Schon ohne Last ist das Rechtecksignal fast 0,1 Millisekunden länger als programmiert.

Da es sich bei Raspberry Pi OS aber nicht um ein Echtzeit-Linux handelt (Real-Time Linux), sind je nach Auslastung des Minirechners sehr viel größere Abweichungen möglich. Denn während das Python-Programm schläft, kümmert sich das Betriebssystem um andere Prozesse, die im Hintergrund laufen, um Dateisystemoperationen, Datenverkehr über Ethernet und WLAN und vieles mehr.

Modellbau-Servos sind aber empfindlich genug, um auf Abweichungen im Bereich von Mikrosekunden zu reagieren. Deshalb kommt es abhängig von der Systemauslastung des Raspi dazu, dass der Servoarm zittert. In seltenen Fällen kann der Servo sogar von einem Anschlag zum anderen fahren, wenn es durch Multitasking zu größeren Timing-Problemen kommt. Gerade bei den schnellen Digital-Servos kann das ein Modell oder andere Hardware beschädigen.

PWM statt Bit Banging

Die Lösung ist, anstelle von Bit Banging einen der beiden PWM-Signalgeneratoren (Pulse Width Modulation) für die Ansteuerung der Servos zu benutzen, beispielsweise den an Pin 12 des Raspi. Einmal programmiert, wiederholt der PWM-Generator das Signal selbsttätig so lange, bis Sie es ändern oder wieder abschalten. Der PWM-Generator kennt zwei Parameter: die Frequenz, mit der er das Signal wiederholt, und die Einschaltdauer (Duty Cycle), die angibt, wie viel Prozent des Signaldurchgangs ein High-Pegel ausgegeben werden soll. Eine Frequenz von 100 Hz und ein Duty Cycle von 50 Prozent bedeuten, dass während einer Periodendauer von 10 Millisekunden die ersten 5 Millisekunden ein High-Pegel ausgegeben wird, danach ein Low-Pegel. Das ist zu viel für einen Servo. Für die Mittelstellung muss die Einschaltdauer, also Pulsbreite, bei einer Frequenz von 100 Hz genau 15 Prozent betragen:

```
import RPi.GPIO as GPIO
from time import sleep
servopin = 12
GPIO.setmode(GPIO.BOARD)
GPIO.setup(servopin, GPIO.OUT)
servo = GPIO.PWM(servopin, 100)
servo.start(15)
sleep(0.1)
servo.stop()
```

```
GPIO.cleanup()
```

Durch den Aufruf der Funktion *GPIO.PWM()* legen Sie fest, welchen Pin und welche Frequenz Sie verwenden wollen. Für Servos haben Sie die Wahl zwischen 40 Hz, was einem Pulsabstand von 25 Millisekunden entspricht, und 200 Hz, wodurch der Abstand auf 5 Millisekunden schmilzt. Häufig findet man in Beispielen eine Frequenz von 50 Hz, diese waren früher bei Modellbauempfängern üblich. Mit 100 Hz reagiert das Servo nicht nur schneller, es erleichtert auch die Berechnung des Duty Cycle - 5 Prozent entsprechen 0,5 Millisekunden, 15 Prozent 1,5 Millisekunden, also Mittelstellung, und 25 Prozent 2,5 Millisekunden. Die Prozentangabe muss übrigens nicht ganzzahlig sein, *10.51* ist ebenfalls zulässig. Das Beispiel ist ausreichend präzise, um frisch ausgepackte Servos vor dem Einbau zuverlässig in die Mittelstellung zu fahren.

Der *sleep()* -Aufruf vor *servo.stop()* sorgt dafür, dass der PWM-Generator überhaupt anläuft: Der Raspi ist so schnell, dass er das PWM-Signal wieder abschalten würde, noch ehe das Signal einmal vollständig ausgesendet wurde. Der Servo würde also ohne *sleep()* nicht reagieren. Das hat auch Auswirkungen für die korrekte Ausnahmebehandlung von Strg+C, falls bei einem solchen Programmabbruch das Servo wieder in die Mittelstellung zurückgefahren werden soll, bevor der PWM-Generator abgeschaltet und der GPIO-Anschluss wieder freigegeben werden:

```
try:
  ...
except KeyboardInterrupt:
  servo.ChangeDutyCycle(15)
  sleep(0.1)
  servo.stop()
  GPIO.cleanup()
```

Der Vorteil von PWM gegenüber Bit Banging ist, dass Sie mehrere Servos gleichzeitig ansteuern können: Einmal programmiert laufen die PWM-Generatoren eigenständig und halten das Timing auch dann ein, wenn der Raspi belastet wird. Allerdings ist damit die Zahl der Servos auf zwei begrenzt, mehr Hardware-PWM-Generatoren besitzt der Raspberry Pi nicht. Beim Bit Banging könnten Sie theoretisch mit allen 29 GPOI-Schaltpins Servos ansteuern, in der Praxis würde das Timing dabei aber viel zu ungenau. Die Lösung für mehr Servos am Raspberry Pi ist deshalb ein externer Servo-Controller, den Sie über den I2C-Bus des Raspberry Pi ansteuern. Passende Zusatzplatinen für 8 oder gar 16 Servos gibt es für 10 bis 20 Euro.

Fazit

Der Raspi hat nur wenige eingebaute Hardware-Controller für externe Bussysteme. Per Bit Banging kann das leistungsfähige SoC des Raspberry Pi jedoch viele gebräuchliche Bussysteme emulieren, indem er einzelne Pins des GPIO-Anschlusses schnell genug ein- und ausschaltet oder einen der PWM-Generatoren benutzt, um das Timing des Bus-Protokolls zu emulieren. Man darf es aber damit nicht übertreiben und mit zu vielen Pins Bit Banging betreiben, sonst fällt der Raspi mit zunehmender Systemlast schnell aus dem Rahmen dessen, was die Protokolle erlauben. Für einzelne Sensoren oder Servos reicht die Leistung allemal.

Literatur

1. Mirko Dölle, Auf Tritt, GPIO-Pins des Raspi in Python programmieren, c't 3/2022, S. 154

Listings zum Download: ct.de/yyp7

Mirko Dölle

Quelle:	c't Heft 6/2022 S. 24-29
ISSN:	0724-8679
Ressort:	Titel
Rubrik:	Raspberry Pi: Sensoren und Aktoren
Dokumentnummer:	2202617585862701465