

Physical Computing

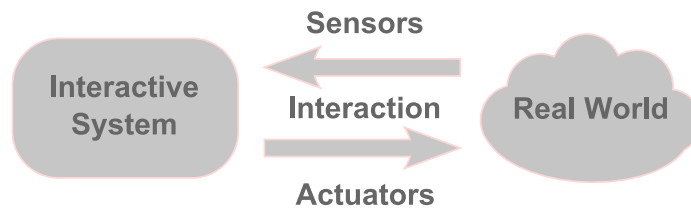
M. Bakera, R. Gummlich, A. Alef

23. November 2020

Inhaltsverzeichnis

	Ansteuerung mit Python . .	24
	Test des SPI-Bus	25
	Typische Probleme	25
Bitoperationen	3	
Bitweise logische Operationen	5	
Bits einer Ganzzahl in Python ausgeben	6	
Bit setzen und zurücksetzen	7	
Bit lesen	10	
Einzelnes Bit toggeln (umschalten)	10	
Tauschen von Bytes	13	
Zweierkomplement	14	
Bus-Systeme	17	
I ² C	17	
Master-Slave-System	17	
Leitungen und Pins	17	
Register und Adressen . . .	19	
i2ctools	19	
Ansteuerung mit Python . .	20	
Lesen aus Word-Registern .	21	
Typische Probleme	22	
SPI	22	
Master-Slave-System	22	
Leitungen und Pins	22	
Lesen und Schreiben	24	
	Bauteile prüfen (TODO)	25
	Software-Dummies	25
	MQTT	28
	Broker	30
	Clients für die Kommandozeile . .	30
	Clients in Python	32
	Sicherheit	35
	Software-Versionierung	39
	Zentrale Ansätze	40
	Verteilte Ansätze	41
	Git	42
	Git-Server	43
	Web-Grundlagen	45
	Software-Bibliotheken	46
	Python-Bibliotheken mit pip installieren	47

Matplotlib zum Zeichnen von	
Grafiken	47
Das Webframework Bottle	51
Fehlersuche	65



Physical Computing vereint die beiden Themenschwerpunkte **Programmierung** und **Elektrotechnik** und kombiniert unterschiedliche Aspekte der Hard- mit der Softwareentwicklung. Hierbei ergeben sich neue Möglichkeiten, aus Anwendungen heraus mit Systemen der realen Welt zu interagieren. Sensoren können Messwerte liefern, die von einer Anwendung verarbeitet werden und dazu führen, dass Aktoren wie z.B. ein Motor so angesteuert werden, dass eine Jalousie geschlossen oder eine Thermostat herunter geregelt wird.

Bei der Entwicklung von komplexen Systemen kommen verschiedene Technologien, Hardwarekomponenten, Software-Bibliotheken und Protokolle zum Einsatz, die in diesem kleinen Skript erläutert werden. Es erfolgt immer nur ein kurzer Abriss – ein weiteres Studium sowie praktische Übungen müssen in jedem Fall zusätzlich erfolgen, um ein tieferes Verständnis zu entwickeln.

Innerhalb des Skripts gibt es immer wieder Aufträge, die du nach und nach selbständig bearbeiten solltest, um die Themen auch praktisch umzusetzen. Dies kann alleine, zu zweit oder auch in einer Lerngruppe passieren. Trage nach dem Bearbeiten das jeweilige Datum ein. Nicht jeder Auftrag muss erfolgreich gelöst werden. Wichtiger ist, regelmäßig an den Aufträgen zu arbeiten.¹



Auftrag 1: Lies die obigen Zeilen und trage unten das heutige Datum ein.

Bearbeitet am:

Bitoperationen

Bei der Programmierung in einer höheren Programmiersprache macht man sich häufig keine Gedanken, wie Daten repräsentiert werden. Allgemein bekannt ist, dass Daten im Speicher in Bytes organisiert sind. Bytes sind „nahezu unteilbar“, auch wenn wir wissen, dass jedes Byte aus 8 Bits besteht. Betrachten wir nun einzelne Daten und ihre

¹Das Hand-Symbol wurde von Adrien Coquet erstellt und vom „Noun Project“ zu freien Verwendung zur Verfügung gestellt.

```
i=18
j = 4
str="qwertz"
pi= 3.14

print("Wert: ",i, " Typ: ",type(i),
      " Bytes: ",sys.getsizeof(i), bin(i))
print("Wert: ",j, " Typ: ",type(j),
      " Bytes: ",sys.getsizeof(j), bin(j))
print("Wert: ",str, " Typ: ",type(str),
      " Bytes: ",sys.getsizeof(str))
print("Wert: ",pi, " Typ: ",type(pi),
      " Bytes: ",sys.getsizeof(pi))
```

Abbildung 1: Datentypen, Platzbedarf und binäre Ausgabe

```
>>>
Wert:  18  Typ:  <class 'int'>  Bytes:  28 0b10010
Wert:   4  Typ:  <class 'int'>  Bytes:  28 0b100
Wert:  qwertz  Typ:  <class 'str'>  Bytes:  55
Wert:  3.14  Typ:  <class 'float'>  Bytes:  24
```

Abbildung 2: Datentypen, Platzbedarf und binäre Ausgabe, Konsolausgabe

Datentypen, so fällt auf, dass sie i. a. Vielfache von Bytes in Anspruch nehmen (und im Arbeitsspeicher besetzen). Abb. 1 zeigt ein Pythonprogramm, das die Datentypen, ihren Platzbedarf und soweit möglich, die binäre Darstellung ausgibt, Abb. 2 dessen Ausgabe in der Konsole.

In vielen Fällen kann es sinnvoll sein, Zugriff auf einzelne Bits eines Bytes lesend oder schreibend zu haben. Dies ist vor allem bei HW-nahen Anwendungen (Embedded Systems) sinnvoll, früher auch, als Speicher um ein Vielfaches teurer war und man deswegen in einem Byte möglicherweise gezwungen war, verschiedenartige binäre Informationen unterzubringen.

Ein Beispiel: In einem Byte soll die Konfiguration einer Ampel gespeichert werden. Die unteren drei Bits (Bit 0 bis Bit 2) sollen anzeigen, ob die Ampel aktiv (eingeschaltet) ist, die Bits 3 bis 5 zeigen an, ob die Ampel blinken soll (Wert = 1) oder Dauerlicht hat (Wert = 0).

Bit Nr.	7	6	5	4	3	2	1	0
Wert	0	0	0	1	0	0	1	0

Tabelle 1: Beispiel: Bits einer Ampelsteuerung, gelb, Blinkbetrieb

In der obigen Tabelle kann man die Bitkombination sehen. Offensichtlich ist die mittlere Ampel (üblicherweise gelb) aktiv und soll blinken. Liest man diese Bitkombination als Dualzahl, so ergibt sich der Wert

$$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 1 \cdot 16 + 1 \cdot 2 = 18$$

Nun soll die Ampel von Blinkbetrieb abstellen und die rote Ampel auf Dauerbetrieb einschalten. Dies führt zu folgenden Schritten

Bit Nr.	7	6	5	4	3	2	1	0
Wert	0	0	0	0	0	1	0	0

Tabelle 2: Beispiel: Bits einer Ampelsteuerung, rot, kein Blinkbetrieb

$$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 0 \cdot 4 = 4$$

Man könnte nun für ein Umschalten des Ampelbetriebes eine Zahl i verwenden, wie oben im Python-Programm zu sehen ist, manuell die Bitkombinationen in Dualzahlen umrechnen und entsprechend setzen oder zurücksetzen. Einfacher aber wäre es, die Bits direkt in der Variablen lesen, setzen oder zurücksetzen zu können.

Bitweise logische Operationen

Beim Zugriff auf einzelne Bits einer Variablen (üblicherweise eine Integerzahl) macht man sich die gängigen logischen Operatoren und Schiebebefehle (Bitshift) zu nutze:

- Logisches UND (AND, and, &)
- Logisches ODER (OR, or, ||)
- Exklusiv-Oder (XOR, \vee , ^)
- Bitshift nach links (<<)
- Bitshift nach rechts (>>)

```
>>> import sys
>>> i=18
>>> bin(18)
'0b10010'
>>> i=-i
>>> bin(i)
'-0b10010'
```

Abbildung 3: Duale und Hexausgabe

 Auftrag 2: Mache Dir an einer Wahrheitstabelle klar, wie die obigen logischen Operatoren UND, ODER und XOR wirken.

Bearbeitet am:

 Auftrag 3: Überlege Dir , wie sich ein Bitshift um eine Stelle mathematisch auf den Gesamtwert des Zahl auswirkt.

Bearbeitet am:

Bits einer Ganzzahl in Python ausgeben

In Python gibt es in der Bibliothek sys (import sys) einen Befehl, der die bitweise Ausgabe von Ganzzahlen (int, integer) ermöglicht (vgl. Abb. 3)

Beachte, dass das Binärmuster als Dualzahl mit mathematischem Vorzeichen ausgegeben wird. Bitintern werden negative Zahlen anders dargestellt. Außerdem werden nur die mathematisch wesentlichen Stellen ausgegeben, führende Nullen oder Einsen (negative Zahl im Zweierkomplement) werden unterdrückt. Insofern entspricht das ausgegebene Bitmuster nur mathematisch dem Bitmuster.

 Auftrag 4: Gib die Zahlen 0, 1, 2, 4, 8 usw. binär aus.

Bearbeitet am:

Mit dem dualen Zahlensystemen eng verwandt ist das hexadezimale. Jeweils 4 Bits des dualen Systemes ergeben eine Stelle des hexadezimalen und umgekehrt. Besonde-

rer Vorteil: jeder 4. Übertrag des dualen Zahlensystems fällt mit den Überträgen des hexadezimalen zusammen. Das führt zu folgendem „Kochrezept“: Wenn man eine duale Zahlenkolonne hat, kann man vom LSB (least significant bit, das Bit der geringsten Stellenwertigkeit, rechteste Stelle) ausgehend je 4 Stellen in eine Hexzahl umsetzen und umgekehrt. Dazu ist es ausreichend, sich die Kombinationen von 0000 = 0x0 bis 1111=0xF zu merken und mechanisch zu ersetzen. Anmerkung: Hexzahlen werden, wie in vielen Programmiersprachen üblich, in diesem Skript mit einem führenden 0x eingeleitet.

Bit setzen und zurücksetzen

Beim Setzen² oder Zurücksetzen eines Bits in einer Zahl ist der Anspruch, dass nur dieses bestimmte Bit gesetzt oder zurückgesetzt wird, die Werte der übrigen Bits aber unangetastet bleiben.

Hierzu arbeitet man neben der Zahl, die in entsprechender Art durch Setzen und Zurücksetzen der Bits verändert werden soll, mit einer sogenannten Bitmaske, kurz Maske. Die Maske verhüllt alle Stellen der Zahl und lässt nur das Bit sichtbar, das verändert (oder gelesen) werden soll. Dies natürlich im übertragenen Sinne. Man belegt die Maske mit Bits so, dass sie an allen Stellen neutral wirkt, in denen keine Veränderung erfolgen soll, an der bewußten zu verändernden Stelle sich aber durchsetzt. Was neutral bzw. durchsetzen ist, kann nur im Zusammenhang mit der logischen Operation gesehen werden. Eine 1 ist im Zusammenhang mit einem logischen Und neutral, während eine 0 mit einem logischen Oder neutral wirkt.

Die Bits werden von rechts nach links und mit 0 anfangend durchnummeriert. Wenn also vom Bit Nr. 4 hier gesprochen wird, ist eigentlich das 5. gemeint.

Bit Nr.	7	6	5	4	3	2	1	0
Maske $m = 1$	0	0	0	0	0	0	0	1
Bitshift der Maske m um 2 Stellen nach links	0	0	0	0	0	1	0	0
Ampel a vorher	0	0	0	1	0	0	0	0
$a \leftarrow a \parallel m$	0	0	0	1	0	1	0	0

Tabelle 3: Beispiel: Setzen des roten Bits mit der Nummer 2

Wir haben mit der Maske m ein Bitmuster durch Verschieben nach links erzeugt, das

²Mit Setzen ist hier das Zuweisen des Wertes „1“ gemeint, mit Zurücksetzen das Zuweisen einer „0“, analog zum englischen Sprachgebrauch „set“ und „reset“.

```
# Bit setzen
m=1
a=16
m=m<<2
print ("Maske:           ", m, bin(m))
print ("Zielbyte vorher:  ", a, bin(a))
a=a|m
print ("Zielbyte nachher: ", a, bin(a))
```

Abbildung 4: Bit setzen

eine „1“ an der Stelle hat, an der in dem anderen Byte die „1“ gesetzt werden soll. Durch die Veroderung wird die „1“ in das Ampelbyte übernommen, während die „0“en neutral wirken, so dass die anderen Bits unverändert bleiben.

Das Zurücksetzen eines Bits funktioniert, indem man eine Maske erzeugt, die eine „0“ an der Bitstelle enthält, die zurückgesetzt werden soll und ansonsten mit „1“en gefüllt ist (neutrales Verhalten). Als logische Operation wird dieses Mal das logische Und verwendet.

Bit Nr.	7	6	5	4	3	2	1	0
Maske $m = 1$	0	0	0	0	0	0	0	1
Bitshift der Maske m um 4 Stellen nach links	0	0	0	1	0	0	0	0
Bitweises Invertieren $m \leftarrow \sim m$	1	1	1	0	1	1	1	1
Ampel a vorher	0	0	0	1	0	1	0	0
$a \leftarrow a \& m$	0	0	0	0	0	1	0	0

Tabelle 4: Beispiel: Zurücksetzen eines gelben Bits mit der Nummer 4

Das Codestück (Abb. 4) und seine Ausgabe (Abb. 4) zeigen die Möglichkeiten des Setzens des Bits Nr. 2 in Python:

Das Zurücksetzen des Bits Nr. 4 wird in Abb 6 und Abb. 7 veranschaulicht.

Das Programm führt zu folgender Ausgabe:

```
0b1
0b100
Maske:           4 0b100
Zielbyte vorher: 16 0b10000
Zielbyte nachher: 20 0b10100
```


Abbildung 5: Bit setzen, Konsolausgabe

```
# Bit Nr 4 zuruecksetzen
m=1
m=m<<4
m = ~m
print ("Maske:           ", m, bin(m))
print ("Zielbyte vorher: ", a, bin(a))
a=a&m
print ("Zielbyte nachher: ", a, bin(a))
```

Abbildung 6: Bit zurücksetzen

```
Maske:           -17 -0b10001\footnote{kjdfjbfsgkdfj}
Zielbyte vorher:  20 0b10100
Zielbyte nachher:  4 0b100
```

Abbildung 7: Bit zurücksetzen, Konsolausgabe

 Auftrag 5: Setze in einer beliebigen Zahl ein höheres Bit und setze es wieder zurück. Natürlich darf der Wert am Ende nicht verändert sein ...

Bearbeitet am:


Bit lesen

Beim Lesen eines Bits werden ähnliche Mechanismen verwendet, wie beim Setzen und beim Zurücksetzen. Es wird wieder mit einer 1-Maske gearbeitet, die auf die entsprechende Bitstelle nach links verschoben wird. Diese Maske wird mit dem zu untersuchenden Byte und-verknüpft. Ist die entstehende Zahl eine 0, so war auch das untersuchte Bit 0, anderenfalls (Zahl > 0), ist es eine 1.

Bit Nr.	7	6	5	4	3	2	1	0
Maske $m = 1$	0	0	0	0	0	0	0	1
Bitshift der Maske m um 4 Stellen nach links	0	0	0	1	0	0	0	0
Zu lesendes Ampelbyte	0	0	0	1	0	1	0	0
$x \leftarrow a \& m$	0	0	0	1	0	0	0	0

Tabelle 5: Beispiel: Lesen des Bits mit der Nummer 4

Man sieht, dass durch die und-Verknüpfung zwischen Maske m und Ampelbyte a (genannt x) lediglich das selektierte Bit übrigbleibt.

 Auftrag 6: Überlege Dir, wie man die ein Bit lesenden oder setzenden Codestücke verwenden kann, um das untere Byte einer Zahl mit dem nächsten Byte zu vertauschen.

Bearbeitet am:

Einzelnes Bit toggeln (umschalten)

Unter Toggeln versteht man das Umschalten eines Bitwertes, also von 0 auf 1 bzw. von 1 auf 0. Hierbei kommt das Exklusiv-Oder zur Anwendung. Dieses führt bei Gleichheit zu einer 0, bei Ungleichheit zu einer 1. Deswegen nennt man es auch Antivalenz. Man erzeugt eine Maske mit einer 1 an der zu toggelnden Stelle (restliche Stellen = 0).

```

# Bit Nr 4 lesen
m=1
m=m<<4
print ("Maske:           ", m, bin(m))
print ("zu lesendes Byte: ", a, bin(a))
x=a&m
print ("Ergebnis:        ", x, bin(x))
# Bit Nr 2 lesen
m=1
m=m<<2
print ("Maske:           ", m, bin(m))
print ("zu lesendes Byte: ", a, bin(a))
x=a&m
print ("Ergebnis:        ", x, bin(x))

```

Abbildung 8: Bestimmtes Bit lesen

```

0b1
0b100
Maske:           4 0b100
Zielbyte vorher: 16 0b10000
Zielbyte nachher: 20 0b10100
Maske:          -17 -0b10001
Zielbyte vorher: 20 0b10100
Zielbyte nachher: 4 0b100
Maske:           16 0b10000
zu lesendes Byte: 4 0b100
Ergebnis:        0 0b0
Maske:           4 0b100
zu lesendes Byte: 4 0b100
Ergebnis:        4 0b100

```

Abbildung 9: Bestimmtes Bit lesen, Konsolausgabe

```
# 2. Bit toggeln
m=1
a = 0xF
m=m<<2
print ("Maske:           ", m, bin(m))
print ("zu lesendes Byte: ", a, bin(a))
a=a^m
print ("Ergebnis:         ", a, bin(a))
a=a^m
print ("Ergebnis:         ", a, bin(a))
```

Abbildung 10: Bestimmtes Bit toggeln

- Fall 01: die zu toggeln Stelle sei 0. Dann wird sie mit der Masken-1 „ver-x-odert“ und führt zu einer 1.
- Fall 11: die zu toggeln Stelle sei 1. Dann wird sie mit der Masken-1 „ver-x-odert“ und führt zu einer 0.
- Fall 10: diese nicht zu toggeln Stelle sei 1. Dann wird sie mit der Masken-0 „ver-x-odert“ und bleibt bei einer 1.
- Fall 00: diese nicht zu toggeln Stelle sei 0. Dann wird sie mit der Masken-0 „ver-x-odert“ und bleibt bei einer 0.

Bit Nr.	7	6	5	4	3	2	1	0
Maske	0	0	0	1	0	0	0	0
Ampelbyte	0	0	0	1	0	1	0	0
das erste Mal toggeln: $a \leftarrow a \hat{m}$	0	0	0	0	0	0	0	0
das zweite Mal toggeln: $a \leftarrow a \hat{m}$	0	0	0	1	0	0	0	0

Tabelle 6: Beispiel: Toggeln eines Bits

```

Maske:           4 0b100
zu lesendes Byte: 15 0b1111
Ergebnis:       11 0b1011
Ergebnis:       15 0b1111


```

Abbildung 11: Bestimmtes Bit toggeln, Konsolausgabe

Bit Nr.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x = 43527	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1

y = 1962	0	0	0	0	0	1	1	1	1	0	1	0	1	0	1	0
----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Abbildung 12: Beispiel: Tausch der ersten beiden Bytes in einer Zahl

 **Auftrag 7:** Toggeln Sie das MSB einer int-Zahl und lassen Sie sich den Wert ausgeben. Hinweis: mit `sys.getsizeof(i)` erhalten Sie die Anzahl der Bytes, die Ihre Zahl in verbraucht. Durch Multiplikation mit 8 erhalten sie die Gesamtzahl der Bits.

Bearbeitet am:

Tauschen von Bytes

Das (Aus)Tauschen von Bytes lässt sich in drei Schritte untergliedern:

- Das Extrahieren eines, des ersten Bytes und Abspeichern in einer Zwischenvariablen,
- das Extrahieren des zweiten Bytes und Abspeichern (Überschreiben) an der Stelle des ersten Bytes (Originalposition) und
- das Schreiben des zwischengespeicherten ersten Bytes an die Stelle des zweiten.

Bit Nr.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$i = 43527 = 0xAA07$	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1
Maske $m = 0xFF00$	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
Zwischenvariable $z = m \& i$	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0
$z = z >> 8$	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0
$i = i << 8$	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0
$i = i z = 0x07AA$	0	0	0	0	0	1	1	1	1	0	1	0	1	0	1	0

Tabelle 7: Beispiel: Vertauschen des ersten Bytes einer Zahl mit dem zweiten

```
# Bytetausch von 0xAA07 (43520) auf 0x07AA (1962)
m = 0xff00
i = 0xaa07
print("i ", hex(i))
z = i & m      # oberes Byte in z zwischenspeichern
z = z >> 8     # z nach rechts schieben
i = i << 8     # unteres Byte hochschieben
i = i | z      # Ex-oberes Byte unten einblenden
i = 0x00ffff & i # 3. Byte loeschen
print("i vertauscht ", hex(i))
```

Abbildung 13: Bytes tauschen

Zweierkomplement

Das Dualzahlensystem im mathematischen Sinne kennt nur positive Zahlen und die 0. Um nun auch mit negativen Zahlen arbeiten zu können, gibt es verschiedene Ansätze. Der naheliegende Ansatz, das MSB (most significant bit, also das am weitesten links stehende Bit) als Vorzeichen zu interpretieren, hat den Nachteil, dass es zwei Möglichkeiten gäbe, die 0 zu kodieren, nämlich einmal die +0 und einmal die -0. Diese fehlende Eindeutigkeit ist das ko.-Kriterium für diesen Ansatz. Eine bessere und insbesondere eindeutige Lösung ist die Speicherung im Zweierkomplement (engl. 2's complement). Der positive Zahlenbereich bei n Stellen läuft von 0 bis $2^{n-1} - 1$, der negative von 2^{n-1} bis -1 . Man kann folgende Algorithmen anwenden, um das Zweierkomplement zu bilden:

- Man invertiert die ganze Zahl bitweise und addiert eine 1 oder

i	0xaa07
i vertauscht	0x7aa

Abbildung 14: Bytes tauschen, Konsolausgabe

- man liest die Zahl von rechts bis zur ersten 1, die man beibehält und invertiert alle weiteren Stellen links davon.


Die erste Vorgehensweise eignet sich für die Arbeit mit Papier und Stift, die zweite ist mit Programmiersprachenelementen einfach realisierbar (Abb.15).

Auf jeden Fall ist eine negative Zahl durch eine führende 1 (MSB) erkennbar.

Bit Nr.	i	7	6	5	4	3	2	1	0
$i = 1$	1	0	0	0	0	0	0	0	1
$i = \sim i$		1	1	1	1	1	1	1	0
$i = i + 1$	-1	1	1	1	1	1	1	1	1
zurück $i = \sim i$		0	0	0	0	0	0	0	0
zurück $i = i + 1$	1	0	0	0	0	0	0	0	1

Tabelle 8: Beispiel: Bilden des Zweierkomplements

Tabelle zeigt die binären Mechanismen der Zweierkomplementbildung hin (von 1 zu -1) und zurück (von -1 nach 1). Man erkennt, dass die Vorgehensweise in beide Richtungen funktioniert. Auch das entsprechenden Python-Programm (Abb.15) führt zum korrekten Ergebnis (Abb.16)

 Auftrag 8: Erstellen Sie eine negative Zahl in einem Pythonscript und testen Sie das MSB.

Bearbeitet am:

```
# Alternative 1 (ueber Bitoperationen):
i = 1
# Umwandlung ins Negative
print("vorher  : ", i, bin(i))
i = ~i # bitweise invertieren
print("invertiert", i, bin(i))
i = i+1# plus 1
print("negativ  ", i, bin(i))
# zurueck ins Positive
i = ~i # bitweise invertieren
print("invertiert", i, bin(i))
i = i+1# plus 1
print("positiv  ", i, bin(i))

# Alternative 2 (ueber Python-Mechanismen:
i = 1
print("vorher  : ", i, bin(i))
i = -i
print("nachher   ", i, bin(i))
i = -i
print("wieder zurueck  ", i, bin(i))
```

Abbildung 15: Python-Programm zur Bildung des Zweierkomplements

```
vorher  :  1 0b1
invertiert -2 -0b10
negativ   -1 -0b1
invertiert 0 0b0
positiv   1 0b1
vorher  :  1 0b1
nachher   -1 -0b1
wieder zurueck  1 0b1
```

Abbildung 16: Python-Programm zur Bildung des Zweierkomplements, Konsolausgabe

Bus-Systeme

In einem Protokoll wird die Kommunikation zwischen zwei Kommunikationspartnern geregelt. Diese Partner können zwei Computersysteme sein – etwa ein Webserver und ein Browser – oder auch ein Hardwarebauteil, das mit einem anderen Bauteil oder dem Raspberry Pi kommuniziert. Die wesentlichen und wichtigen Protokolle werden in den folgenden Abschnitten vorgestellt.

I²C

I²C steht für Inter-Integrated Circuit und beschreibt eine Schnittstelle, die für die Kommunikation zwischen integrierten Schaltungen (ICs) und Mikrocontrollern (z.B. Raspberry Pi oder Arduino) benutzt wird. Ursprünglich wurde sie in den 1980er Jahren von Philips entwickelt und der Name markenrechtlich bis ins Jahr 2006 geschützt. Daher existieren alternative Bezeichnungen wie TWI (two-wire interface) oder SMBus (System Management Bus). Bei der Programmierung wird dieses Detail später noch einmal eine Rolle spielen, da die Python-Bibliothek für die Ansteuerung von I²C Bauteilen den Namen `smbus` trägt.

Master-Slave-System

Bei einem Bussystem teilen sich mehrere Teilnehmer dasselbe Kommunikationsmedium (die Busleitung). Damit es bei der Kommunikation nicht zu Kollisionen kommt, wird ein Buszugriffsverfahren benötigt.

Beim I²C handelt es sich um ein Master-Slave-System. Der Master (bei uns der Raspberry Pi) steuert die Kommunikation über die Busleitung. Die Slaves (alle anderen Bauteile, die am Bus angeschlossen sind) kommunizieren nur, wenn sie vom Master angesprochen werden. Dazu hat jeder Slave eine eigene 7-Bit-lange Adresse.

Leitungen und Pins

ICs werden bei I²C über zwei Leitungen SDA und SCL angeschlossen. Über die Taktleitung SCL (serial clock) gibt der Master die Geschwindigkeit der Kommunikation vor. Die Daten werden über die Datenleitung SDA (serial data) übertragen. Die Kommunikation findet über diese Leitung in beide Richtungen (vom Master zum Slave und vom Slave zum Master) statt. Die Slaves senden erst Daten, wenn sie vom Master aufgefordert wurden.

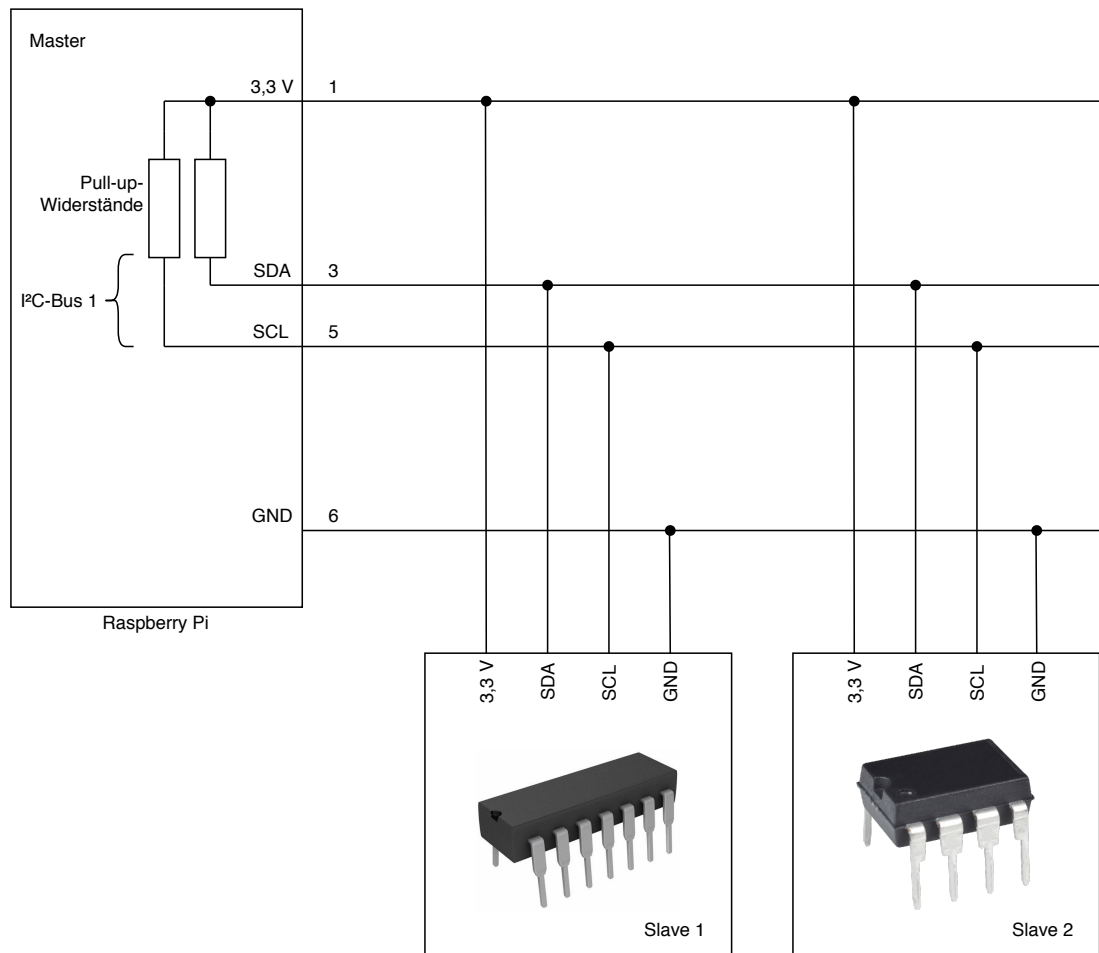


Abbildung 17: Blockschaltbild: Master und zwei Slaves am I²C Bus

Beim Raspberry Pi sind nicht alle GPIO-Pins für I²C geeignet. Intern verfügt er über zwei I²C Schnittstellen. Die beiden Schnittstellen werden mit I²C 0 und I²C 1 bezeichnet und sind an den folgenden Pins verfügbar:

I ² C Bus	Pin: Board	BCM	Bedeutung
1	3	2	SDA
	5	3	SCL
0	27	0	SDA
	28	1	SCL

Zusätzlich müssen die ICs mit Energie versorgt werden und benötigen daher eine Spannung von 3,3 V. Die Masse muss ebenfalls an alle ICs angeschlossen werden.

Abb. 17 zeigt, wie zwei Slaves am I²C Bus 1 des Pi angeschlossen werden. Die Pull-up-Widerstände an den Busleitungen sind beim Raspberry Pi bereits integriert und müssen nicht zusätzlich angeschlossen werden. Manche I²C Bauteile verfügen über weitere Pins, die unbedingt beschaltet werden müssen. Häufig wird darüber die Slaveadresse des Bauteils eingestellt. Diese Informationen müssen im Datenblatt nachgelesen werden.



Auftrag 9: Zeichne eine normgerechte und vollständig beschriftete Schaltung mit einem beliebigen I²C Bauteil, welches am Raspberry Pi angeschlossen werden soll.

Bearbeitet am:

Register und Adressen

Auf dem I²C Bus ist jeder IC unter einer eindeutigen Adresse erreichbar. Diese Adresse wird häufig als Hexadezimalzahl wie 0x68 angegeben.

Jeder I²C Chip verfügt zudem über unterschiedliche Register, die beschrieben und gelesen werden können. Diese kann man sich wie kleine Speicherstellen vorstellen, über die der IC konfiguriert, verändert oder abgefragt werden kann. Die exakte Bedeutung der einzelnen Register muss im Einzelfall dem Datenblatt des Bauteils entnommen werden.

i2ctools

Ein an den Pi angeschlossener IC kann leicht über die i2c-tools detektiert und angesprochen werden. Das ist eine Sammlung kleine Kommandozeilentools, die beim Debugging und der Fehlersuche helfen. Eine Installation ist leicht über die Paketverwaltung möglich, falls die Programme noch nicht installiert sind.


```
sudo apt-get install i2c-tools
```

Ein IC an Bus 1 kann mit dem Befehl `i2cdetect` ermittelt werden. Da alle I²C Chips über eine Adresse eindeutig identifizierbar sind, melden sie sich unter dieser und tauchen in der Ausgabe auf. Die Adresse wird häufig in hexadezimaler Schreibweise angegeben.


```
$ i2cdetect -y 1
```

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00:				--	--	--	--	--	--	--	--	--	--	--	--	--
10:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
20:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
30:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
40:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
50:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
60:	--	--	--	--	--	--	--	--	68	--	--	--	--	--	--	--
70:	--	--	--	--	--	--	--	--								

Hier taucht ein IC unter der Adresse 0x68 auf. Mit den Kommandozeilentools `i2cget` und `i2cset` können die Register eines Bausteins gelesen und beschrieben werden.

 Auftrag 10: Installiere die `i2ctools` und schließe einen Baustein an. Ermittle und notiere die zugehörige Adresse. Teste auch den zweiten I²C Bus.

Bearbeitet am:

 Auftrag 11: Beschreibe das I²C Protokoll und erläutere darin die folgenden Begriffe: Adresse, SDA, SCL, Register, Master, Slave.

Bearbeitet am:

Ansteuerung mit Python

Wenn der Baustein auf dem Bus erkannt wird, kann er mit einem einfachen Pythonprogramm angesteuert werden. Hierfür wird die Bibliothek `smbus`³ verwendet. Das folgende Programm zeigt, wie ein Byte mit dem Inhalt 0x01 an ein Register 0x00 eines Bauteils

³Dies steht für „System Management Bus“, eine alternative Bezeichnung für den I²C Bus.

```
import smbus

I2C_BUS = 1
DEVICE_ADDR = 0x68
DEVICE_REGISTER = 0x00
DATA = 0x01

bus = smbus.SMBus(I2C_BUS)
bus.write_byte_data(DEVICE_ADDR, DEVICE_REGISTER, DATA)
```

Abbildung 18: Ansteuerung von I²C mit Python

gesendet wird, das am I²C Bus 1 unter der Adresse 0x68 angeschlossen ist. Der Effekt hängt natürlich vom Bauteil ab und muss im Datenblatt recherchiert werden.

Ein Beispiel für die Ansteuerung ist in Abb. 18 zu sehen. Neben der Methode `write_byte_data` zum Schreiben gibt es eine Methode `read_byte_data(dev_addr, register)`, über die ein Register eines IC ausgelesen werden kann. Die Methode liefert als Ergebnis ein Byte zurück.



Auftrag 12: Greife auf einen I²C Chip aus einem Python-Programm heraus zu.

Bearbeitet am:

Lesen aus Word-Registern

In manchen I²C Chips sind die Register zwei Byte (2 Byte = 1 Word) groß. Mit den `i2c-tools` lässt sich so ein Register mit dem Zusatz „w“ ausgelesen. Im Beispiel wird das Register 0x00 eines Bauteils mit der Adresse 0x68 am I²C Bus 1 gelesen.

```
$ i2cget -y 1 0x68 0x00 w
```

In Python gibt es in der Bibliothek `smbus` die Methode `read_word_data(dev_addr, register)`, um solch ein Register auszulesen.

Bei beiden Varianten ist unbedingt zu beachten, dass die beiden Bytes in verkehrter Reihenfolge gelesen werden. Das niederwertige Byte steht vor dem höherwertigen Byte. Um den gelesenen Wert korrekt interpretieren zu können, müssen die beiden Bytes also vertauscht werden. Wie das funktioniert, wird im Kapitel zu Bitoperationen ab Seite 3 beschrieben.

Typische Probleme

Sollte der I²C Bus am Pi nicht aktiviert sein, kann er über einen Aufruf von `sudo raspi-config` in den Einstellungen aktiviert werden. Die entsprechende Einstellung befindet sich im Abschnitt „Interfacing Options“.

SPI

SPI steht für Serial Peripheral Interface und beschreibt eine weitere serielle Schnittstelle, die für die Kommunikation zwischen integrierten Schaltungen (ICs) und Mikrocontrollern (z.B. Raspberry Pi oder Arduino) benutzt wird. Das Bussystem wurde 1987 von Motorola (heute NXP Semiconductors) entwickelt.

Master-Slave-System

Auch beim SPI teilen sich alle Bauteile dieselben Busleitungen. Der Zugriff wird wie beim I²C nach dem Master-Slave-Prinzip geregelt.

Anders als beim I²C haben die SPI Bauteile allerdings keine Adresse, die bei der Kommunikation übermittelt werden muss. Die Slaves werden stattdessen vom Master (dem Raspberry Pi) über „Chip-Select-“ (CS) bzw. „Chip-Enable-“ (CE) Leitungen angesprochen. Dazu wird jeder Slave über eine separate CE-Leitung mit dem Master verbunden.

Das Chip-Enable-Signal ist „Low-aktiv“. Das bedeutet, dass vom Raspberry Pi ein Low-Signal (0 V) an die CE-Leitung angelegt werden muss, um mit dem dazugehörigen Bauteil zu kommunizieren. Alle andere CE-Leitungen werden in dieser Zeit auf High (3,3 V) gesetzt, damit kein weiterer Slave bei der Kommunikation stört.

Leitungen und Pins

Zum SPI-Bus gehören drei Leitungen. Wie beim I²C gibt es beim SPI ebenfalls eine Taktleitung, die häufig mit SCLK (serial clock) bezeichnet wird. Zusätzlich gibt es zwei Datenleitungen. Die MOSI-Leitung ist für die Datenübertragung vom Master zum Slave („Master out Slave in“) vorgesehen. Über die MISO-Leitung senden die Slaves Daten zum Master („Master in Slave out“). Zusätzlich werden für jeden Slave eine separate Chip-Enable-Leitung (CE) und die Spannungsversorgung benötigt.

Der Raspberry Pi verfügt über zwei SPI Schnittstellen, welche mit SPI 0 und SPI 1 bezeichnet werden. Die Schnittstelle verfügt über zwei bzw. drei CE-Pins (CE0, CE1 und CE2). Die folgende Tabelle zeigt, welche Pins der GPIO-Schnittstelle dafür vorgesehen sind:

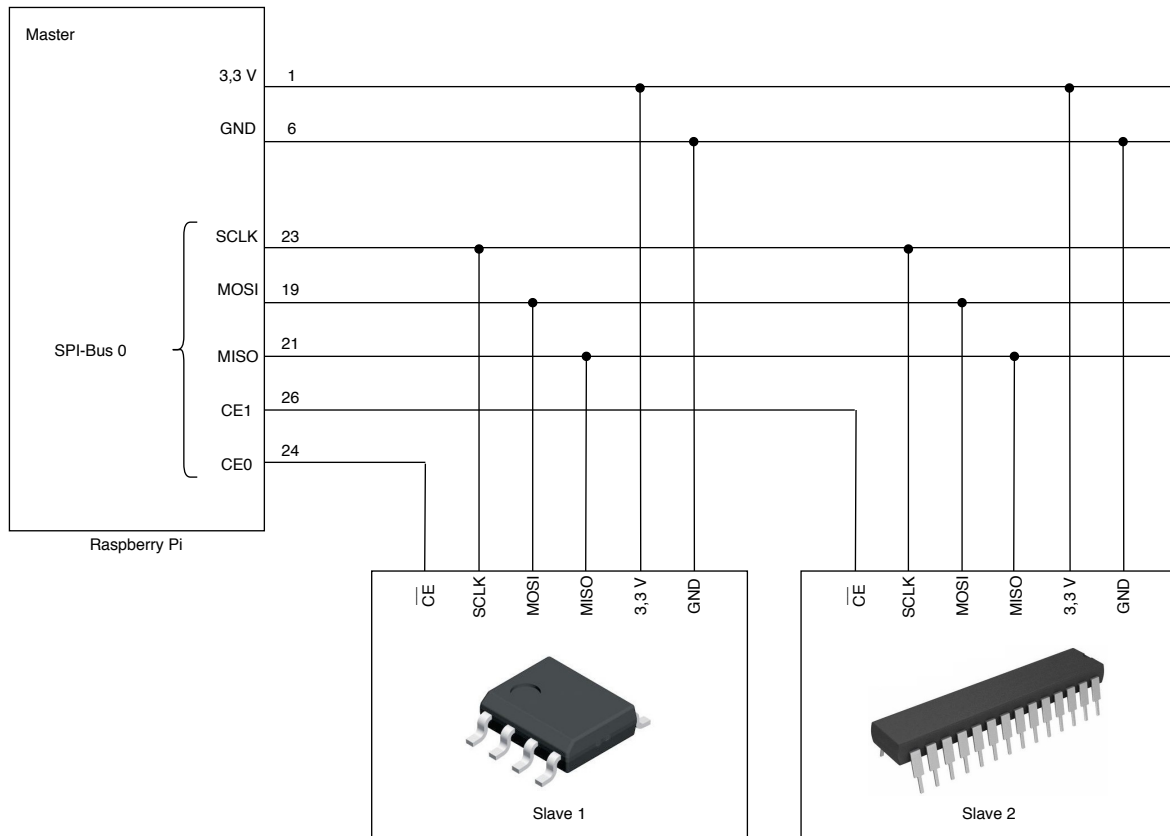




Abbildung 19: Blockschaltbild: Master und zwei Slaves am SPI Bus

SPI Bus	Pin: Board	BCM	Bedeutung
0	19	10	MOSI
	21	9	MISO
	23	11	SCLK
	24	8	CE0
	26	7	CE1
1	38	20	MOSI
	35	19	MISO
	40	21	SCLK
	12	18	CE0
	11	17	CE1
	36	16	CE2

Abb. 19 zeigt, wie zwei Slaves am SPI Bus 0 des Pi angeschlossen werden. Ein Blick in die Datenblätter der SPI Bauteile ist auch hier unverzichtbar, um herauszufinden, welche Pins des jeweiligen Chips angeschlossen werden müssen. In den Datenblättern mancher Bauteile werden der MOSI-Pin mit DI (data in) und der MISO-Pin mit DO (data out) bezeichnet. Die damit angegebene Signalflussrichtung bezieht sich auf das Slave-Bauteil.

 Auftrag 13: Vergleiche den SPI Bus mit dem I²C Bus. Nenne Gemeinsamkeiten und Unterschiede. Gehe dabei auf die Buszugriffsverfahren ein.

Bearbeitet am:

 Auftrag 14: Zeichne eine normgerechte und vollständig beschriftete Schaltung mit einem beliebigen SPI Bauteil, welches am Raspberry Pi angeschlossen werden soll.

Bearbeitet am:

Lesen und Schreiben

- synchrone Datenübertragung

Ansteuerung mit Python

- spidev-Bibliothek

Test des SPI-Bus

- MISO und MOSI kurzschließen

Typische Probleme

- Sollte der SPI Bus am Pi nicht aktiviert sein, kann er über einen Aufruf von `sudo raspi-config` in den Einstellungen aktiviert werden. Die entsprechende Einstellung befindet sich im Abschnitt „Interfacing Options“.
- Sollte die Kommunikation mit einem IC augenscheinlich nicht funktionieren, so kann es sein, dass die Frequenz des Taktsignals begrenzt werden muss. In Python setzt man das Attribut `max_speed_hz` des SPI-Objektes auf einen moderaten Wert. Im Beispiel wird die Frequenz auf 1 MHz gesetzt. Die maximal zulässige Frequenz hängt vom Bauteil ab und kann im Datenblatt nachgelesen werden.

```
spi = spidev.SpiDev()  
spi.open( ... )  
spi.max_speed_hz = 1000000
```

Bauteile prüfen (TODO)

Software-Dummies

Es ist erstrebenswert, Software und Hardware gleichzeitig und arbeitsteilig zu entwickeln. So kann die Software schon geschrieben werden, während eine Schaltung erst noch entwickelt werden muss. Zu diesem Zweck werden in der Softwareentwicklung sogenannte Dummies für eine noch zu entwickelnde Komponente – in unserem Falle eine Hardwarekomponente – erstellt.

Nehmen wir einen Taster als Beispiel für eine sehr einfache Hardwarekomponente. Wir wollen den Taster in unserer Software benutzen, ohne dass wir die Hardware besitzen – dafür müssen wir ihn softwareseitig simulieren. Daher realisieren wir den Taster über eine entsprechende Klasse **Taster**. Damit die Software auch ohne den Taster erstellt werden kann, wird eine Dummyklasse entwickelt, die jedoch keine echten, sondern nur simuliert Werte liefert. Dies können z.B. Zufallswerte, immer die gleichen Werte oder selbst gewählte Werte in einer bestimmten Reihenfolge sein.

Für das Beispiel soll die Klasse lediglich über eine Methode `ist_gedrueckt` verfügen, die den Status des Tasters liefert: „gedrückt“=`True` und „nicht gedrückt“=`False`. Ein Klassendiagramm befindet sich in Abb. 20.

Den Dummy für den Taster erstellen wir nun in einer Datei `dummy.py` mit dem Inhalt in Abb. 21 (oben). Die Methode `ist_gedrueckt` liefert an dieser Stelle immer `True` zurück und simuliert damit einen dauerhaft gedrückten Taster.

Beachte, dass die im Konstruktor übergebene Pin-Nummer nicht weiter benötigt wird und daher auch nicht in einem Attribut gespeichert werden muss. Lediglich eine Ausgabe zur Information ist vorhanden.

Wir können dieser Klasse nun in einem Projekt sofort verwenden und mit den Werten etwas machen – z.B. können wir sie einfach auf der Konsole ausgeben.

```
# main.py

from dummy import Taster

taster = Taster(23)
while True:
    print(taster.ist_gedrueckt())
```

Wichtig ist hierbei, dass unser Programm bereits Werte liefert und nicht auf eine Hardware angewiesen ist. Diese wird parallel in einer anderen Datei entwickelt, welche wir `hardware.py` nennen. Sie enthält die tatsächliche Implementierung des Tasters, durch die der Dummy später ersetzt wird (vgl. Abb 21, unten).

Sobald die Entwicklung an der realen Klassen abgeschlossen ist, muss lediglich der Import im Hauptprogramm angepasst werden. Wir können hierfür eine Konfigurationsvariable `USE_DUMMY` nutzen, um zu entscheiden, aus welcher Datei (`dummy.py` oder `hardware.py`) die Klasse Taster bezogen werden soll. Somit kann zwischen dem Dummy und der realen Klasse entschieden werden und das Programm ist durch Ändern dieser Variable auch auf einem Rechner ohne GPIOs weiterhin lauffähig.

Der Rest des Programmes bleibt, wie er ist, da beide Klassenbeschreibungen den gleichen Klassennamen, den gleichen Methodennamen und die gleichen Arten von Rückgabewerten enthalten. Beide Varianten der Klasse Taster sind also kompatibel zueinander.

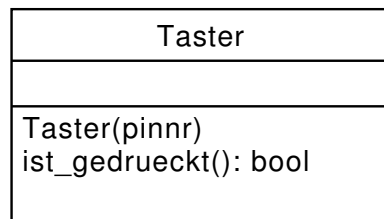


Abbildung 20: Klassendiagramm für einen Taster

```
# dummy.py

class Taster:
    def __init__(self, pinnr):
        print("Taster erstellt an GPIO pin", pinnr)

    def ist_gedrueckt(self):
        return True
```

```
# hardware.py

import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BOARD)

class Taster:
    def __init__(self, pin):
        GPIO.setup(pin, GPIO.IN)
        self.pin = pin

    def ist_gedrueckt(self):
        taster_status = GPIO.input(self.pin)
        if taster_status == GPIO.HIGH:
            return True
        else:
            return False
```


Abbildung 21: Quelltext für den Dummy und die reale Implementierung eines Tasters

```
# main.py


USE_DUMMY = False

if USE_DUMMY:
    from dummy import Taster
else:
    from hardware import Taster

...
```

 Auftrag 15: Erstelle einen Dummy für einen Taster, der zufällig den Wert True oder False zurückliefert, wenn die Methode `ist_gedrueckt` aufgerufen wird.

Bearbeitet am:

 Auftrag 16: Erstelle einen Dummy für ein Bauteil des aktuellen Projektes. Zeichne dafür zunächst ein Klassendiagramm und realisiere im Anschluss den Dummy und die reale Klasse.

Bearbeitet am:

Wenn du das Thema noch weiter vertiefen möchtest, eignen sich folgende Begriffe für weitere Recherchen: „Mock“ bzw. „Mock-Objekt“, „Stub“ oder ganz allgemein „Testgetriebene Entwicklung“ (im Englischen „Test-Driven Development“ (TDD)).

Python selbst wird bereits mit dem Paket `unittest.mock`⁴ ausgeliefert, welches während eines Komponenten-Tests die Ersetzung von Quelltext ermöglicht. Damit wird ein Verfahren realisiert, welches dem oben beschriebenen ähnelt.

Message Queuing Telemetry Transport (MQTT)

MQTT⁵ ist ein Protokoll, das auf Port 1883 und 8883 (mit Verschlüsselung) läuft und für die Übertragung von Sensordaten zwischen Maschinen entwickelt wurde. An der Entwicklung sind IBM und andere große IT-Unternehmen beteiligt. Der prinzipielle Aufbau ist in Abbildung 22 zu sehen. Sie zeigt einen zentralen Server, der bei MQTT Broker

⁴ <https://docs.python.org/3/library/unittest.mock>

⁵ Mehr Informationen unter <https://de.wikipedia.org/wiki/MQTT>

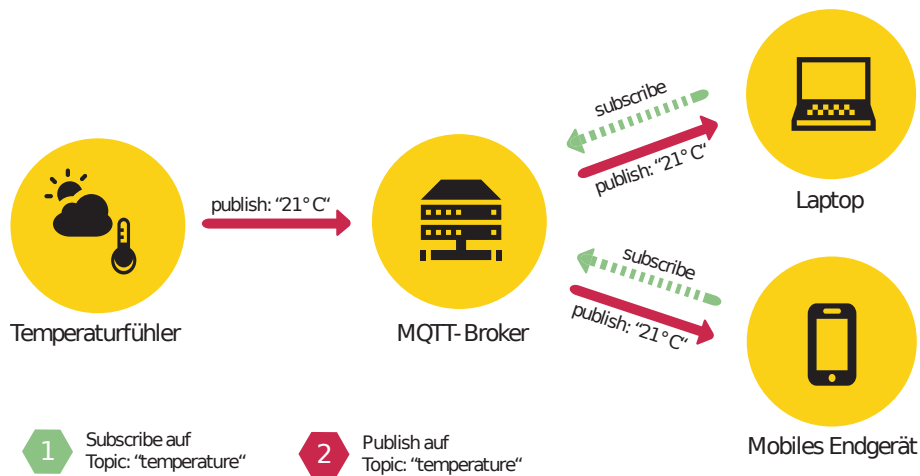


Abbildung 22: Ein typischer Anwendungsfall von MQTT (Quelle: <https://heise.de/-2168152>)

genannt wird. Clients können sich mit dem Broker verbinden und mit ihm kommunizieren. Unter Clients verstehen wir im Rahmen von MQTT Geräte wie Smartphones, Desktoprechner aber auch kleine Sensoren und Maschinen. Die Clients veröffentlichen (publish) Nachrichten in Themen (topics). Andere Clients können wiederum diese Themen abonnieren (subscribe) und werden über neue Nachrichten vom Broker informiert.


Diese Form der Kommunikation beruht auf dem »Publish-Subscribe«-Entwurfsmuster, bei dem Sender Nachrichten auf einem Bus senden und andere Teilnehmer diese Nachrichten wiederum mitlesen können. Hierbei ist zunächst unerheblich, wie die einzelnen Teilnehmer miteinander kommunizieren, da der Sender nicht weiß, wer die Nachrichten empfängt und was er damit macht.

Eine Filterung der Nachrichten kann allgemein nach dem Inhalt, Themen oder einer Mischung aus beiden erfolgen. Bei MQTT werden die Nachrichten auf der Basis von Themen und Unterthemen abonniert.

Ein Artikel bei Heise⁶ und ein Artikel bei DZone⁷ beschreiben das MQTT-Protokoll ausführlich.

⁶<https://heise.de/-2168152> (mehrere Seiten)

⁷<https://dzone.com/articles/mqtt-the-nerve-system-of-iot> (englisch)

 Auftrag 17: Lies die beiden Artikel bei Heise und dZone unter den angegebenen Links und schreibe eine kurze Zusammenfassung, die folgende Begriffe erläutert: MQTT, Client, Broker, Topic, publish, subscribe, Quality of Service (QoS), Last Will/Testament, Retained Messages.

Bearbeitet am:


Broker

Bei der Kommunikation ist ein Server beteiligt (Broker genannt), der zwischen den Clients vermittelt und die Nachrichten von den Clients empfangen und an diese verteilen kann. Die Referenz-Implementierung für einen solchen Broker ist »mosquitto«, die über den Paket-Manager auf dem Pi installiert werden kann.

```
sudo apt install mosquitto
```

Für Testzwecke kann auch ein öffentlicher Broker von Eclipse verwendet werden.

```
MQTT_BROKER = "mqtt.eclipse.org"
MQTT_PORT = 1883
```

 Auftrag 18: Installiere mosquitto auf dem Pi. Prüfe, ob der Broker funktioniert und notiere, wie diese Prüfung durchgeführt wurde (z.B. mit Hilfe von systemctl).

Bearbeitet am:

Clients für die Kommandozeile

Mit dem Broker können nun verschiedene Clients kommunizieren. Hierbei kann ein Client ein Topic abonnieren oder Nachrichten in ein Topic schreiben. Es existieren Clients für die Kommandozeile, die über das Paket mosquitto-clients auf dem Pi installiert werden können.

```
sudo apt install mosquitto-clients
```

Nun stehen zwei Befehle `mosquitto_sub` zum Abonnieren (subscribe) von Topics und `mosquitto_pub` zum Veröffentlichen (publish) von Daten in Topics zur Verfügung. Ohne weiteren Programmieraufwand können die Befehle ausgeführt werden, um den Broker zu testen.

```
mosquitto_sub -v -h mqtt.eclipse.org
               -t Erdgeschoss/Wohnzimmer/Temp

mosquitto_pub -h mqtt.eclipse.org
               -t Erdgeschoss/Wohnzimmer/Temp
               -m 23
```


Abbildung 23: Eine Nachricht an einen Broker senden und ein Topic abonnieren.

Option	Bedeutung
-h	Ein Host kann explizit angegeben werden. Diese Option kann entfallen, wenn der Broker auf localhost verwendet werden soll.
-v	Steht für verbose (ausführlich) und gibt beim Empfang von Daten zusätzlich das Topic an.
-t	Das betroffene Topic. Ein Topic ist ähnlich einer URL oder einem Dateipfad aufgebaut und verweist z.B. auf einen Sensor an einem bestimmten Ort.
-m	Die Nachricht (message), die an den Broker gesendet werden soll.

Abbildung 24: Wichtige Optionen für `mosquitto_sub` und `mosquitto_pub`

Öffne zunächst ein Fenster und gib den Befehl aus Abb. 23 (oben) ein, um ein Topic zu abonnieren. Der Befehl wartet bis Daten im Topic eintreffen und gibt sie dann auf der Konsole aus. Öffne ein weiteres Fenster und gib den Befehl aus Abb. 23 (unten) ein. Dieser sendet nun einen Wert in das Topic.

Der empfangene Werte wird zusammen mit dem Topic im ersten Fenster angezeigt. Wichtige Optionen sind in Tabelle 24 gelistet.

 **Auftrag 19:** Installiere die Kommandozeilentools und teste sie wie beschrieben. Schau auch in die man-page und notiere weitere relevante Optionen.

Bearbeitet am:


Neben den Clients für die Kommandozeile existieren auch grafische Clients. Ein Programm, das für alle Betriebssysteme verfügbar ist, nennt sich »MQTT-Explorer«⁸.

Clients in Python

Nun erstellen wir zwei Clients in Python, die mit dem Broker kommunizieren sollen. Einer abonniert ein Topic und wartet auf eintreffende Nachrichten (Subscriber) und ein anderer veröffentlicht Nachrichten in dieses Topic (Publisher). Wir legen dafür ein gemeinsames Topic in einer Variable fest.

```
TOPIC = "Erdgeschoss/Wohnzimmer/Temp"
```

Als Client-Bibliothek für Python nutzen wir `paho-mqtt`⁹, die sich mit `pip` installieren lässt. `Pip` ist ein mächtiges Tool für die Installation von Python-Bibliotheken, welches auf Seite 47 beschrieben wird.

 **Auftrag 20:** Installiere `paho-mqtt` auf dem Pi und dem eigenen Laptop. Versichere dich auch, dass die Installation erfolgreich war, indem du eine Python-Konsole startest und darin `paho.mqtt` importierst.

Bearbeitet am:

Subscriber Wir beginnen mit einem Client, der ein Topic abonniert und bei neuen Nachrichten eine Ausgabe auf die Konsole schreibt. Hierfür geben wir zwei callback-Methoden an. Das sind Methoden, die aufgerufen werden, sobald ein Ereignis auftritt.

⁸<https://mqtt-explorer.com/>

⁹<https://pypi.python.org/pypi/paho-mqtt>

In unserem Fall sollen die Methoden auf eine erfolgreiche Verbindung und eine neue Nachricht reagieren.

1. `my_connect_method` soll aufgerufen werden, sobald eine Verbindung zum Broker zustande gekommen ist.
2. `my_message_method` soll bei jeder neuen Nachricht aufgerufen werden.

Abbildung 25 zeigt in den Zeilen 5-10 die Implementierung der Methoden. In den Zeilen 12-15 wird ein Client erzeugt und die eben erstellten Callback-Methoden festgelegt. Schließlich verbinden wir den Client in Zeile 17 mit dem Broker und warten auf neue Nachrichten.

Sobald der Client verbunden ist, muss eine Endlosschleife gestartet werden, die regelmäßig neue Nachrichten abrufen. Es gibt verschiedene Varianten, diese Netzwerkschleife (englisch *network loop*) zu starten, die in der Dokumentation unter dem Abschnitt *network-loop*¹⁰ genauer beschrieben werden. Exemplarisch sollen hier nur zwei Beispiele genannt werden.

- `loop_start` startet den Abruf im Hintergrund. Das Programm läuft nach dem Ausführen normal weiter.
- `loop_forever` startet die Netzwerkschleife im Vordergrund und blockiert damit die Anwendung ab diesem Aufruf. Weitere Anweisungen, die nach diesem Aufruf kommen, werden nicht ausgeführt.

In unserem Beispiel zeigt Zeile 19, dass wir eine Schleife im Vordergrund starten.



Auftrag 21: Erstelle einen Subscriber in einer Datei `sub.py` und abonniere ein Topic. Starte die Netzwerkschleife. Nutze `mosquitto_pub`, um eine Nachricht an das Topic zu senden.

Bearbeitet am:

Publisher Während der Subscriber läuft, erstellen wir nun einen weiteren Client in einer neuen Datei. Er soll in regelmäßigen Abständen eine Nachricht in das Topic schreiben, weshalb wir ihn Publisher nennen. Er ist in Abb. 26 zu sehen.

¹⁰<https://pypi.org/project/paho-mqtt/#network-loop>

```
1 # subscriber.py
2
3 import paho.mqtt.client as mqtt
4
5 def my_connect_method(client, userdata, flags, rc):
6     print("Connected. Subscribing to topic", TOPIC)
7     client.subscribe(TOPIC)
8
9 def my_message_method(client, userdata, msg):
10     print("Message received:", msg.topic, msg.payload)
11
12 client = mqtt.Client()
13
14 client.on_connect = my_connect_method
15 client.on_message = my_message_method
16
17 client.connect(MQTT_BROKER, MQTT_PORT, keepalive=60)
18
19 client.loop_forever()
```

Abbildung 25: Ein Subscriber für MQTT in Python


```
1 # publisher.py
2
3 import paho.mqtt.client as mqtt
4 import time
5
6 publisher = mqtt.Client()
7 publisher.connect(MQTT_BROKER, MQTT_PORT)
8 publisher.loop_start()
9
10 while True:
11     publisher.publish(topic=TOPIC, payload=22)
12     time.sleep(1)
```

Abbildung 26: Ein Publisher für MQTT in Python

Da der Subscriber lauscht, können wir nun eine Nachricht veröffentlichen. Dies erledigt die Methode `publish` in regelmäßigen Abständen (Zeile 10). Diesmal starten wir die Netzwerkschleife mit `loop_start` im Hintergrund (Zeile 7). Das ist auch sinnvoll so, da ein Aufruf von `loop_forever` dazu führen würde, dass das Programm an dieser Stelle blockiert und die restlichen Zeilen niemals ausgeführt würden.

Wir sehen die Ausgabe aus unseren definierten Callback-Methoden, sobald die Nachricht eintrifft.

```
Message received: Ergeschoss/Wohnzimmer/Temp b'22'
```


 Auftrag 22: Erstelle zwei Dateien `pub.py` und `sub.py`, die jeweils einen Publisher und einen Subscriber enthalten. Der Publisher soll alle zwei Sekunden die aktuelle Zeit aus `time.time()` in das Topic `sensor/time` und den String `OK` in das Topic `sensor/status` senden. Sobald eine neue Nachricht in einem der beiden Topics eintrifft, soll der Subscriber diesen Wert auf der Konsole ausgeben.

Bearbeitet am:

Sicherheit

Sicherheit ist ein Thema, das viele Bereiche der Soft- und Hardwareentwicklung gleichzeitig betrifft und nicht nur an einer Stelle relevant ist. Es gehört zu einem guten Entwicklungsstil, den Sicherheitsaspekt stets mit zu bedenken und nicht nur „am Ende“ noch irgendwie zu integrieren. Die verschiedenen Ebenen betreffen die Hardware, Software, Kommunikation und Vernetzung.

In Abbildung 27 und auf den folgenden Seiten ist ein Ausschnitt aus dem Buch *A Reference Guide to the Internet of Things*¹¹ (S. 70 ff.) zu finden, der unterschiedliche Aspekte beschreibt, wie Sicherheit auf verschiedenen Ebenen umgesetzt werden kann.

 Auftrag 23: Lies die Seiten aus dem Buch zum Thema Sicherheit und beschreibe die verschiedenen Möglichkeiten, um Geräte zu schützen.

Bearbeitet am:

¹¹Eine vollständige Version ist online verfügbar: <https://bridgera.com/wp-content/uploads/2018/10/IoTeBook3.pdf>

Physical Things

You should plan for the in-life supportability of your devices (the physical things) during the design phase. This may include planning for how you will perform firmware updates, battery charging or replacement, and sensor calibration. It also includes device provisioning, on-boarding, diagnostics, and security patching.

Security

Security issues are of special importance in IoT projects. Edge devices tend to get less attention from systems administrators and, thus, can make an attractive target for hackers. Properly and consistently monitoring devices and planning regular updates to firmware and security protocols can be challenging. However, these precautionary measures are paramount to protecting the system and everything connected to it.

These issues can be complex. It will be critical to involve technical staff in the development process. This will help build a system you can efficiently monitor, maintain, and update.

Security Considerations

As more and more Internet-connected devices enter our homes and businesses, it is important to remember that they bring with them security risks. The IoT is an ever-growing phenomenon. In the rush for convenience, security and privacy are often overlooked. IoT security is still in its infancy, leaving the door open for malicious attacks.

Securing an IoT device is not an easy task. There are many IoT devices out there with different hardware configuration and operating systems. These variations make it impossible to deploy a one-size-fits-all solution. IoT devices generally come with very limited processing capabilities. This makes it hard to run state of the art encryption-based security solutions inside the devices. To add to this challenge, many devices are not designed to get regular firmware updates, thus you cannot apply security patches.

Securing the IoT device to stop hackers from gaining access to the device and data is the main challenge. The same security practices for a web application or software platform will not work. However, you can and should apply these practices to the software layer of an IoT system.

Most IP-enabled devices keep a port open for incoming messages. This provides an opening for any outside attack. Fortunately, there are multiple ways to prevent

Abbildung 27: Auszug aus *A Reference Guide to the Internet of Things*

attacks to a device. Device communication is generally limited to exchanging messages with IoT platforms or other devices/hubs. If the device has enough resources to support public/private key encryption, its communication channel can use TLS/SSL encryption. For this type of communication, private key encryption is stored at the device level. Here you'll also find a digital certificate to ensure authenticity. Digital certificates issued by a trusted Certification Authority (CA) expire after a certain period. There should be an easy way to remotely update the digital certificate in the device if these types of certificates are used.

Updating the digital certificate for the device's firmware requires physical access to the device or an Over the Air (OTA) firmware update capability. An alternative is to generate X.509 certificates to use for TLS/SSL communication. These certificates are easier to manage and do not expire. It is important to use separate private keys for each device. This helps to quarantine only the compromised device without shutting down the whole operation.

Implementing a TLS/SSL solution requires a lot of resources, such as memory and processing power. Unfortunately, most low cost, IoT devices won't support this on their own. There are other, more common ways to install security for these types of IoT devices. A good way to prevent a widespread attack is to use a unique piece of information which should be a part of every communication to the device. Some IoT devices send a password/passcode along with every command. The device generally ships with a default password. Not changing the default password before deployment would be a huge mistake. [Note: This was the cause of the DDoS IoT attack in October of 2016].

The IoT software responsible for communicating with these devices should change the default password at the time of activation. It should also change the password frequently using remote device management. The passwords for the devices should be kept securely (encrypted) in the IoT software layer. If each device has a unique password for device-to-server communication, a widespread attack would be virtually impossible.

IP white-listing filters untrusted messages at the device level. This is an easy measure to implement in IoT platforms using a static IP address, or range of static IPs. [Note: You must request a static IP for a mobile IoT device. Otherwise, the cellular/m2m carriers decide the IP address, and it is generally dynamic.] With a static IP address, the known static IP list filters the messages coming into the IoT software layer from devices. Generally, communication between devices does not happen directly, rather it goes through software or a broker. IP-based filtering at the software or broker level is a great buffer to prevent attacks.

As mentioned, most IP-enabled, IoT devices keep a port open for the incoming traffic. Keeping this port open, in listening mode, drains the battery. Some applications prefer to be in sleep mode to save battery. SMS text messaging can send an instruction to wake a device when no open communication port is available. Sending SMS text messages through SMPP gateways is a common commercial solution. Sending SMS text messages requires the use of a phone number, or a set of phone numbers. IoT devices can keep a list of trusted numbers to quickly discard messages from unknown sources.

Software-Versionierung

Wenn mehrere Programmierer an einem Projekt zusammenarbeiten, müssen sie sich abstimmen, da nicht alle gleichzeitig eine Datei editieren können. Wer dies schon einmal probiert hat, merkt schnell, dass Probleme entstehen, wenn eine Stelle in einer Datei gleichzeitig bearbeitet wird.

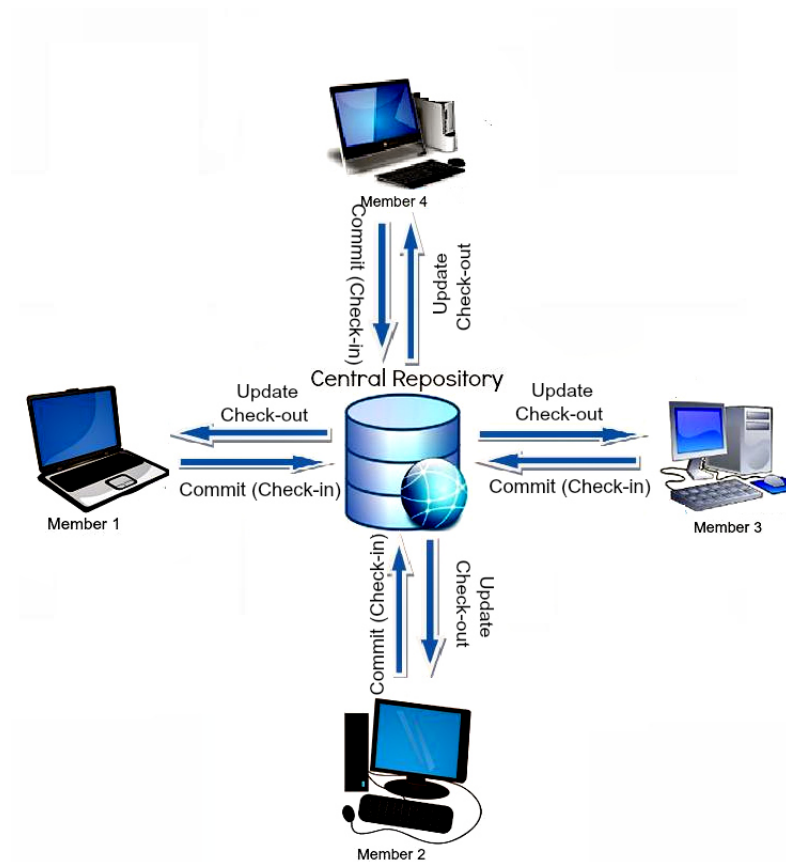
Zudem müssen Fehler, die während der Entwicklung entstehen, korrigierbar sein. Sobald ein Fehler entstanden ist, muss der Ursprung von einzelnen Quelltextzeilen erklärbar sein: Wer hat den Fehler verursacht? Wer könnte das Problem am besten lösen? Auch wenn die Korrektur eines Fehlers länger dauert, möchte man auf einen alten Stand zurückkehren, um den Fehler zu beheben und eine alte Version der Software noch einmal ausliefern zu können.

Bevor wir uns den verschiedenen Systemen zuwenden, müssen einige Begriffe geklärt werden. Daher beginnen wir mit einer Erklärung wichtiger Fachbegriffe. Der Quelltext einer Software wird bei Versionskontrollsystemen in sogenannten Repositories abgespeichert. Das stellt man sich am besten wie eine kleine Datenbank vor, die neben dem Quelltext auch Informationen über die Autoren, die Bearbeitungszeiten und Versionsnummern abspeichert. Auch Verzweigungen (branch genannt) in Produktspezialisierungen sind in einem Repository möglich. So hat man häufig einen stabilen Zweig aus dem jeweils die neue Hauptversion der Software erzeugt wird und Entwicklungszweige, in die jeweils neue Features integriert oder Bugs behoben werden. Die Änderungen, die ein Entwickler am Quelltext durchführt, werden in vielen Systemen als commit bezeichnet. Hierbei kann es sich um Änderungen an einer oder mehreren Dateien handeln.

Probleme entstehen immer dann, wenn zwei Programmierer im Repository in einer Datei an einer Stelle eine Änderung vornehmen wollen und sich nicht abgesprochen haben. Welche Änderung soll dann gelten? Diese Probleme werden Konflikte genannt.

Um Softwareentwickler bei der Arbeit zu unterstützen und um Konflikte zu vermeiden, gibt es Software, die diese Verwaltungsaufgaben übernimmt. Diese Software wird Versionskontrollsystem (im englischen version control system (VCS)) genannt und lässt sich grob in zwei unterschiedliche Ansätze aufteilen: zentrale und verteilte Ansätze.

Zentrale Ansätze

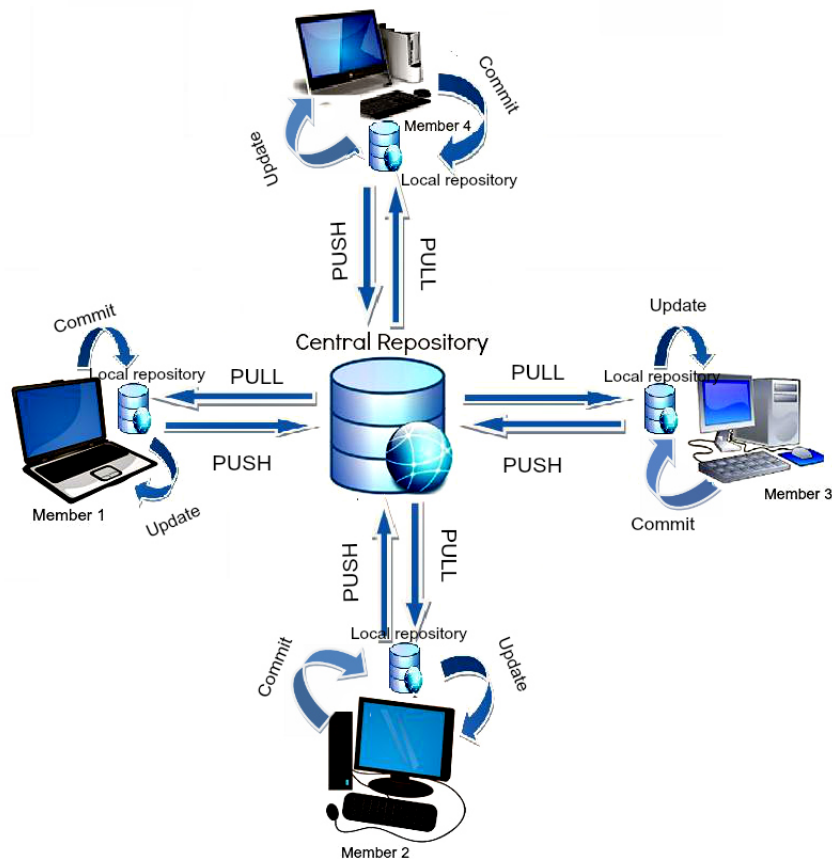


Bei den zentralen Ansätzen¹² wird der Quelltext zentral an nur einer Stelle in einer Art Server gespeichert und die Entwickler greifen auf diesen zentralen Server zu. Vertreter dieser Gattung sind z.B. „Concurrent Versions System“ (cvs) oder „Subversion“ (svn).

Ein Vorteil zentraler Systeme besteht darin, dass immer nur der jeweils aktuelle Stand der Software benötigt wird und nicht die vollständige Versionshistorie. Wenn die Software über viele Jahre oder Jahrzehnte gewachsen ist, können diese Historien viel Platz beanspruchen. Als Nachteil sei zu erwähnen, dass für die Arbeit immer eine Netzwerkverbindung benötigt wird, da jeder commit mit dem zentralen Repository abgeglichen werden muss.


¹²Die Abbildung in diesem und dem nächsten Abschnitt entstammt der Seite <https://scmquest.com/centralized-vs-distributed-version-control-systems/>

Verteilte Ansätze



Bei verteilten Ansätzen hat jeder Entwickler ein eigenes Repository mit allen Versionsständen der Software. Die Änderungen werden im lokalen Repository vorgenommen und regelmäßig mit einem anderen Repository abgeglichen. Typische Softwaresysteme, die einen verteilten Ansatz verfolgen, sind z.B. „Mercurial“ oder „Git“.


Ein Vorteil der verteilten Ansätze ist, dass nun auch eine Arbeit ohne Netzwerkanbindung möglich ist. Da das Abgleichen aber nur ab und zu durchgeführt wird, kommt es häufiger zu Konflikten.

 Auftrag 24: Stelle die Unterschiede zwischen einem zentralen und einem verteilten Ansatz kurz dar: Nenne für jeden Ansatz (a) einen Vorteil, (b) einen Nachteil und (c) einen Vertreter.

Bearbeitet am:

Git


Git ist ein prominenter Vertreter der verteilten Ansätze für Softwareversionierung und kommt auch im Unternehmensumfeld häufig zum Einsatz.

 Auftrag 25: Unter der Adresse <https://pel-daniel.github.io/git-init> wird ein Demo-Ablauf mit typischen git-Operationen vorgestellt. Schau dir den Ablauf an und notiere die vorgestellten git-Befehle mit deren Bedeutung. Auf der Seite befindet sich auch die Übersicht aus Abbildung 28, die noch einmal die wichtigsten Befehle zusammenfasst. Mache ruhig Anmerkungen an dieser Übersicht.

Bearbeitet am:

Auf dem Pi, Linux- und MacOS-Systemen ist git gewöhnlich schon installiert. Für Windows gibt es verschiedene Möglichkeiten, git zu installieren. In der nächsten praktischen Übung wird erklärt, wie die Installation unter Windows vollzogen werden kann.

Das Programm `Git-it`¹³ ist ein Programm zum Erlernen von git. Neben eine Beschreibung, wie der eigene Rechner eingerichtet werden muss, enthält es auch einen Kurs, der mit praktischen Übungen in die Arbeit mit git einführt.

 Auftrag 26: Installiere git auf dem Laptop und prüfe, ob es auch auf dem Pi korrekt installiert ist, indem du `'git --version'` in einer Konsole eingibst.

Mache dann die praktischen Übungen aus dem `Git-it` Kurs. Die Übungen bis zum Kapitel „Forks and Clones“ reichen für den Einstieg. Interessierte können aber natürlich auch die restlichen Übungen machen.

Bearbeitet am:

Ein typischer Ablauf einer Sitzung mit Git ist in Abbildung 29 zu sehen.

¹³<https://github.com/jlord/git-it-electron>

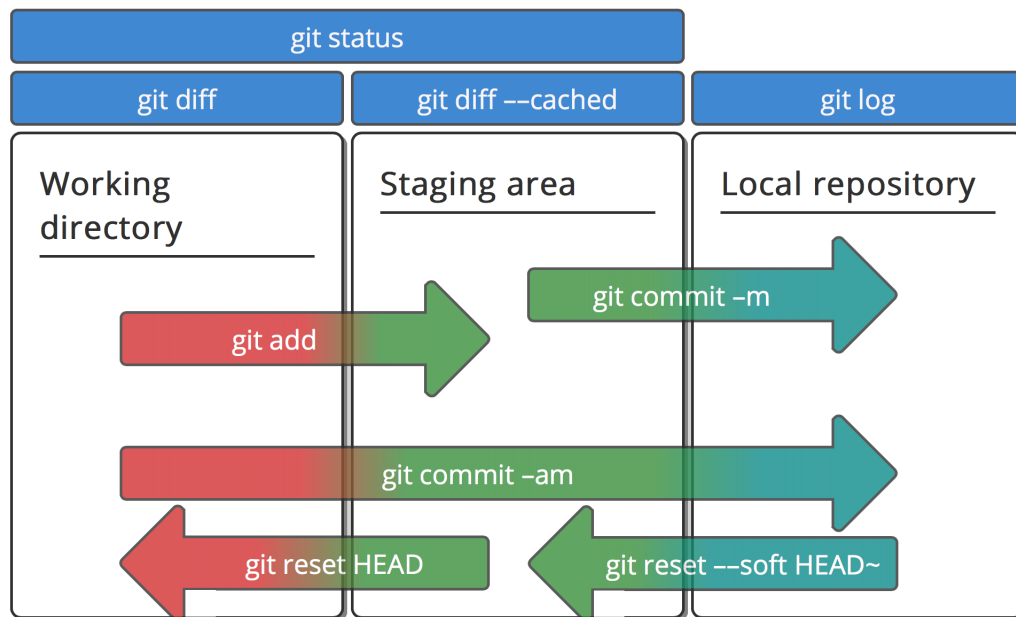


Abbildung 28: Wichtige git-Befehle

☞ Auftrag 27: Recherchiere die Bedeutung des Begriffes git. Versuche zusätzlich, herauszufinden, wer die Software ursprünglich geschrieben hat.

Bearbeitet am:

Git ist ein durchaus komplexes Thema und auch erfahrene Entwickler kennen nach vielen Jahren noch immer nicht alle Funktionen. Daher ist ein weiteres Studium empfohlen. Das Buch Pro Git¹⁴ ist eine umfangreiche Quelle, die es gedruckt oder auch als Nachschlagewerk online gibt. Ein gezieltes Nachschlagen oder vertiefendes Lesen kann ich nur empfehlen, wenn sich ein Problem mit einem Repository ergibt.

Git-Server

Eine der größten Quellen für OpenSource-Software ist die Seite Github¹⁵, welche von der Firma Microsoft betrieben wird. Kostenlos können dort Konten erstellt und genutzt

¹⁴ <https://git-scm.com/book/de/>

¹⁵ <https://www.github.com>


1. `git pull` - Synchronisation mit einem entfernten Repository.
2. (Änderungen an den Dateien durchführen. Hierbei kleinschrittig vorgehen.)
3. `git add` - Änderungen an den Dateien in der „staging area“ vormerken.
4. `git commit` - Vorgemerkte Änderungen in das lokale Repository sichern.
5. `git push` - Die Änderungen, des lokalen Repositories an das Repository auf dem Git-Server senden.

Abbildung 29: Ein typischer Ablauf während der Arbeit mit Git

werden. In immer mehr Firmen gehört ein gut gepflegtes Github-Profil zum guten Ton und ist Teil einer vollständigen Bewerbung als Softwareentwickler.

Github ist eine große aber natürlich nicht die einzige Plattform dieser Art. Unsere Schule verfügt über einen eigenen Git-Server¹⁶, der für eigene Projekte genutzt werden kann. Eine Anmeldung ist mit den Office365-Zugangsdaten möglich.

Wer einen eigenen Git-Server betreiben möchte, kann dies auch tun. So schwierig ist das nicht. In dem Pro Git Buch¹⁷ wird das Verfahren beschrieben, wie ein beliebiger Linux-Server ohne weitere Software für git genutzt werden kann. Eine weitere Option ist eine eigene Server-Software wie etwa gitea, gogs oder gitlab¹⁸. Für den Pi ist gitea eine schlanke und feature-reiche Option, die sich leicht installieren lässt.

 Auftrag 28: Erstelle für das aktuelle Projekt ein git-Repository und veröffentliche es mit `git push` auf einem Server. Erstelle dafür ein Konto auf einem der vorgestellten git-server – auch der eigene Server ist natürlich möglich. Notiere dir die Adresse des Repositories auf dem git-server. Aus dem Git Repository heraus kann diese mit `git remote -v` angezeigt werden.

Bearbeitet am:

¹⁶<https://it.tbs1.de/git> - basierend auf gitea.

¹⁷<https://git-scm.com/book/de/v1/Git-auf-dem-Server-Einrichten-des-Servers>

¹⁸ gitea: <https://gitea.io>, gogs: <https://gogs.io> oder gitlab: <https://gitlab.com>

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"></meta>
    <title>Titel der Webseite</title>
  </head>

  <body>
    Hier kommt der Inhalt der Webseite hinein
  </body>
</html>
```

Abbildung 30: Grundgerüst einer HTML-Seite

Web-Grundlagen

Das World-Wide-Web (kurz: WWW) ist sicherlich jedem durch schillernde und bunte Webseiten bekannt. Man kann es mit einem Browser erkunden und den Verlinkungen von Seite zu Seite folgen, indem man sie schlicht anklickt.

Weniger bekannt sind meist die Technologien, die das WWW ausmachen und im Hintergrund arbeiten. Damit wir den späteren Ausführungen zum Webframework „bottle“ folgen können, müssen wir uns zunächst ein paar Konzepte klarmachen.

HTML steht für „Hypertext Markup Language“. Alle Webseiten im WWW werden als Texte beschrieben, die nach einem ganz bestimmten Format aufgebaut sind – eben HTML. Ein Grundgerüst einer solchen Seite ist in Abb. 30 zu sehen.

Die Elemente einer Seite werden Tags genannt und tauchen immer paarweise auf. Es gibt eine öffnende und eine schließende Version eines Tags: `<title>` (öffnendes Titlelement), `</title>` (schließendes Titlelement).



Auftrag 29: Erstelle auf Grundlage des HTML-Grundgerüsts in Abb. 30 eine Datei mit Namen `webseite.html`. Öffne die Seite in einem Browser und prüfe, ob sie korrekt angezeigt wird. Recherchiere drei Tags und füge sie der Datei hinzu. Ergänze schließlich einen Kommentar in der Datei, der im Browser nicht angezeigt wird.

Bearbeitet am:

HTTP ist eine Abkürzung für „Hypertext Transfer Protocol“. Innerhalb des WWW kommunizieren Clients in Form von Webbrowsern mit Webservern¹⁹. Sie tauschen hauptsächlich Texte über dieses Protokoll aus. Hierbei sendet der Client eine Anfrage (request) an den Server, der diese mit einer Antwort (response) beantwortet.

Der Request erfolgt in Form einer sogenannten HTTP-Methode. Die häufigste Methode in HTTP ist die Methode GET, die eine Information in Form einer Webseite beim Webserver erfragt. Wenn der Client allerdings etwas an den Server sendet, tut er dies mit einer anderen Methode z.B. mit der Methode POST.²⁰ Dies ist z.B. dann der Fall, wenn eine Suchanfrage in ein Suchfeld einer Suchmaschine eingegeben wird.

Als Antwort wird häufig eine HTML-Seite geschickt, wenngleich auch andere Dateiformate wie JSON, XML oder Binärformate wie PNG oder JPG möglich sind. Neben dem Inhalt, ist jede Antwort mit einem Statuscode versehen, der aus drei Ziffern besteht und einen Hinweis darauf gibt, ob die Anfrage erfolgreich war oder nicht. Es gibt eine ganze Menge von Statuscodes, sodass man sie in Gruppen eingeteilt hat: 1xx: Informationen, 2xx: erfolgreiche Operation, 3xx: Umleitung, 4xx: Client-Fehler, 5xx: Server-Fehler, 9xx: Proprietäre Fehler. Am bekanntesten oder häufigsten sind die Statuscodes 200 (OK) und 404 (Not Found).

 Auftrag 30: Ermittle, (a) wer das HTTP-Protokoll entwickelt hat, (b) wann dies geschehen ist und (c) in welchem Land die Entwicklung stattfand.

Bearbeitet am:

Software-Bibliotheken

Für die Physical-Computing-Projekte kommen verschiedene Python-Bibliotheken zum Einsatz, die entweder auf dem eigenen Laptop oder dem Pi installiert werden – manchmal sogar auf beiden Geräten. Hierbei unterscheidet sich der Installationsprozess jeweils abhängig vom Betriebssystem auf dem eigenen Laptop. Da es sich um Python-Bibliotheken handelt, können diese häufig mit dem Kommandozeilentool `pip`²¹ installiert werden.

¹⁹ Verbreitete Browser sind z.B. Firefox oder Google Chrome. Typische Webserver sind auf Linux-Systemen z.B. Apache oder nginx. Fallen dir weitere ein?

²⁰ Neben den beiden Methoden GET und POST existieren noch weitere Methoden wie HEAD, PUT, DELETE, TRACE, OPTIONS, CONNECT.

²¹ Weitere Informationen unter <https://pip.pypa.io/>

Python-Bibliotheken mit pip installieren

So wie `apt` bzw. `apt-get` ein Paket-Manager für Linux und den Pi ist, so ist `pip` ein Paket-Manager für Python-Bibliotheken – die Abkürzung steht für „pip installs packages“²². Er hilft dabei, Bibliotheken zu installieren, zu deinstallieren und sie auf dem aktuellen Stand zu halten. Die Installation eines beliebigen Paketes erfolgt abhängig vom Betriebssystem aber leider immer etwas unterschiedlich, weshalb wir die Unterschiede in diesem Abschnitt kurz darlegen.

Unter Linux (also auch auf dem Pi) und MacOS kann ein Paket mit dem folgenden Aufruf in einem Konsolenfenster installiert werden.

```
pip3 install PAKETNAME
```

Unter Windows muss zunächst eine Eingabeaufforderung geöffnet werden. Danach ist das Vorgehen leider nicht ganz einheitlich und hängt davon ab, mit welchen Optionen Python ursprünglich installiert wurde. In fast allen Fällen sollte der folgenden Aufruf funktionieren.


```
py -m pip install PAKETNAME
```

Sollte der genannte Aufruf unter Windows doch nicht funktionieren, so könnte einer der folgenden Aufrufe helfen.

```
pip install PAKETNAME
```

oder

```
python -m pip install PAKETNAME
```

 **Auftrag 31:** Installiere auf dem Pi und Laptop die folgenden Pakete: `paho-mqtt`, `matplotlib`, `bottle`. Finde ferner heraus, wie sich bestehende Pakete wieder deinstallieren lassen und notiere diesen Befehl.

Bearbeitet am:

Matplotlib zum Zeichnen von Grafiken

²² Hierbei handelt es sich um eine sogenannte rekursive Abkürzung, da der abzukürzende Begriff Teil der Erklärung ist.



Die Bibliothek matplotlib²³ kann für die Erzeugung von Diagrammen genutzt werden. Sie kann mit Hilfe von pip auf dem eigenen Rechner oder dem Pi über den Paketname `matplotlib` installiert werden.

Man sollte nach der Installation prüfen, ob diese erfolgreich war. Dies gelingt am besten mit einem Python-Interpreter: IDLE ist für diesen Zweck vollkommen ausreichend. Bei der Eingabe der folgenden Zeilen darf keine Fehlermeldung erscheinen.

```
import matplotlib
import matplotlib.pyplot as plt
```

In einem einfachen Beispielprogramm, das in Abbildung 31 zu sehen ist, werden wir ein paar Daten festlegen, diese mit matplotlib in einen Graphen zeichnen und das Ergebnis abspeichern.

Wir erstellen eine Liste mit X- und Y-Werten, die in einem Koordinatensystem gezeichnet werden soll (Zeile 3-4).

Mit Hilfe der Funktion `plot` können einfache Plots erstellt werden (Zeile 6). Hierfür müssen lediglich die X- und Y-Werte übergeben werden. Der Plot wird anschließend mit der Funktion `savefig` als PNG- und SVG-Bild abgespeichert (Zeile 7-8), um ihn weiterverwenden zu können. PNG ist ein Bitmapformat während SVG ein Vektorformat ist, das sich beliebig nah heranzoomen lässt. Das erzeugte Bild ist unten in Abbildung 31 abgebildet.

Bevor ein neuer Plot erstellt werden kann, muss mit `plt.clf()` der bisherige Plot zurückgesetzt werden. Für viele weitere Plots und zahlreiche Konfigurationsmöglichkeiten hilft ein Blick in die Beispiele²⁴ und die Beschreibung der Methode `plot`²⁵.

Zeitstempel werden häufig als Werte auf der X-Achse verwendet, da gemessene Sensordaten einem Zeitpunkt zugeordnet werden müssen. In Abb.32 ist zu sehen, wie mit Hilfe des Python-Paketes `datetime` Messwerte über einen Zeitraum von mehreren Sekunden gesammelt und anschließend in einem Graphen dargestellt werden.

Als X-Werte kommen diesmal Zeitwerte zum Einsatz, während die Y-Werte unverändert sind. In einem realen Szenario würde man hier natürlich echte Sensorwerte verwenden.

Die `datetime`-Objekte der verschiedenen Zeitpunkte können wie gewöhnliche Zahlen in der Liste der X-Werte gesammelt werden. Matplotlib erkennt diesen Datentyp und stellt ihn daher als Zeitwert auf der X-Achse korrekt dar.

²³<https://matplotlib.org/>

²⁴https://matplotlib.org/tutorials/introductory/sample_plots.html

²⁵https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot

```
1 import matplotlib.pyplot as plt
2
3 xs = [-3,-2,-1,0,1,2,3]
4 ys = [9,4,1,0,1,4,9]
5
6 plt.plot(xs, ys)
7 plt.savefig("plot.png")
8 plt.savefig("plot.svg")
```

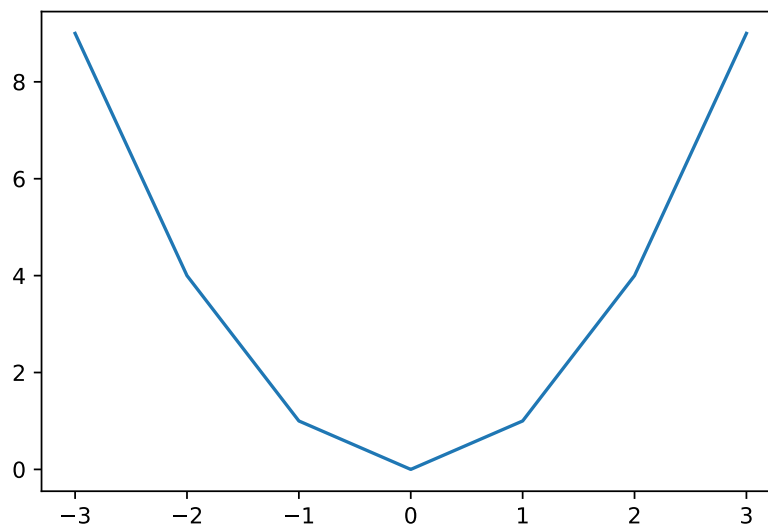


Abbildung 31: Beispielprogramm für Matplotlib

```
1 import datetime
2 import time
3
4 xs = []
5 ys = [9,4,1,0,1,4,9]
6
7 for i in range(7):
8     zeitstempel = datetime.datetime.now()
9     xs.append(zeitstempel)
10    time.sleep(1)
```

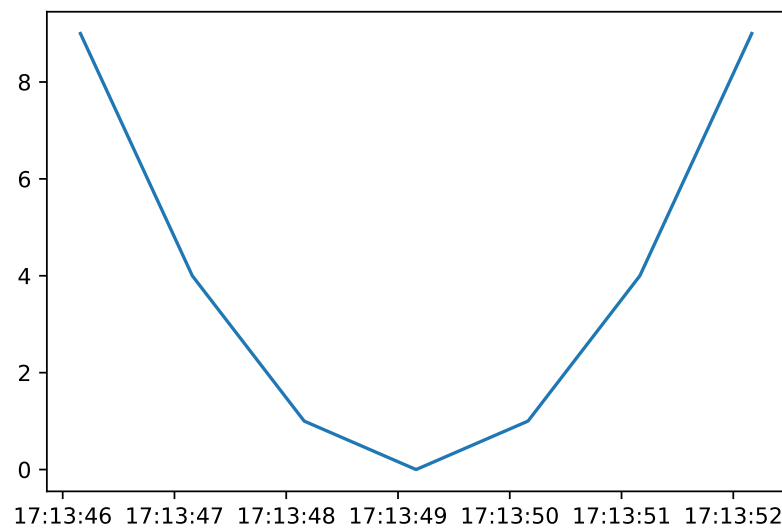



Abbildung 32: Beispielprogramm mit Zeitstempeln für Matplotlib

```
1 import bottle
2
3 @bottle.route("/")
4 def index():
5     return "OK"
6
7 bottle.run(host="127.0.0.1", port=8081, reloader=True)
```

Abbildung 33: Einfacher Bottle-Webserver

 Auftrag 32: Installiere matplotlib auf dem Pi und dem eigenen Laptop und zeichne einen Graphen. Dieser Graph soll 100 Zufallswerte aus dem Bereich zwischen -50 und +50 enthalten, die über einen Zeitraum von 10 Sekunden generiert wurden. Auf der X-Achse werden die Zeitpunkte der Zufallswerte, auf der Y-Achse die Werte selbst angezeigt.

Bearbeitet am:

Das Webframework Bottle

Bottle²⁶ ist ein Webframework für die Entwicklung von Webanwendungen. An dieser Stelle setzen wir daher das Wissen aus dem Abschnitt „Web-Grundlagen“ ab Seite 45 voraus.

Es kann mit pip über den Paketname `bottle` installiert werden. Ein einfacher Import in einer Pythonkonsole hilft, um zu prüfen, ob die Installation erfolgreich war: `import bottle`

Einfacher Webserver Für jeden Pfad, der mit einem Web-Browser aufgerufen werden kann, muss eine Route in Form einer Methode in Bottle deklariert werden. Über `@bottle.route` kann eine Route des Webservers mit einer Pythonfunktion verknüpft werden. Diese Methode muss einfaches HTML in Form eines String an den Browser zurücksenden. Abb.33 zeigt einen einfachen Webserver.



Wir beginnen mit der Route für `/` und binden diese an die Funktion `index` (Zeile 3). Zum Starten des Webservers wird die Methode `bottle.run` verwendet (Zeile 7). Neben dem Parameter für den `host` und `port` gibt es die Option `reloader=True`. Sie startet den

²⁶<https://bottlepy.org/>

Server automatisch neu, sobald sich eine der Python-Dateien geändert hat. Während der Entwicklung ist dies besonders hilfreich.

Es sollte nun die folgende Ausgabe zu sehen sein. Sie enthält auch die URL, über die unsere Anwendung nun erreicht werden kann. Wir sehen auch einen GET-Request, der auf die Route „/“ getätigt wurde.

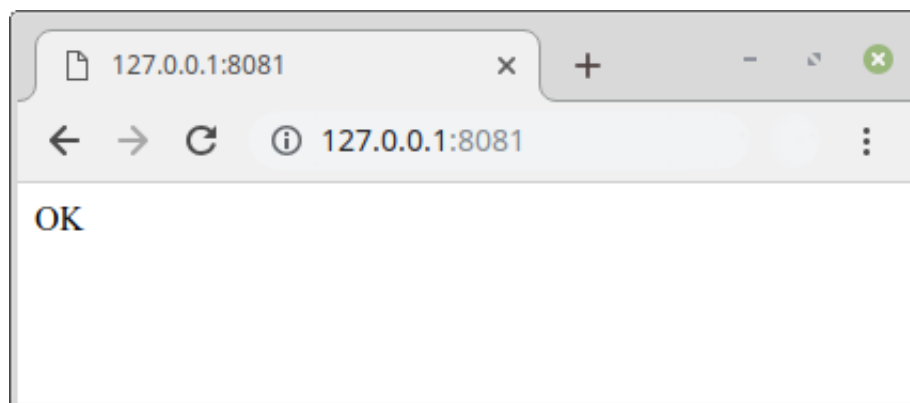
```
Bottle v0.12.13 server starting up (using WSGIRefServer())...

Listening on http://127.0.0.1:8081/

Hit Ctrl-C to quit.

127.0.0.1 - - [30/Sep/2018 09:30:53] "GET / HTTP/1.1" 200 2
```

Unter `http://127.0.0.1:8081/` ist die Seite nun erreichbar. Wenn man sie darunter aufruft, zeigt ein Blick in den Quelltext, dass der Text „OK“ genau so auftaucht, wie er in der Methode angelegt wurde.



HTML-Ausgabe Unser Browser kann jedoch mehr als nur Text anzeigen. Er kann HTML interpretieren. Passen wir die Methode `index` wie in Abb. 34 an und lassen valides HTML zurückgeben.²⁷ Nach dem Start zeigt die Webseite nun eine HTML-Seite mit Quelltext an, wie er in der Methode `index` festgelegt wurde.

²⁷Zur Erinnerung: Wenn eine Zeichenkette in Python mit drei Anführungszeichen (""") umschlossen wird, kann sie sich über mehrere Zeilen erstrecken.

```
...  
@bottle.route("/")  
def index():  
    html = """  
        <!DOCTYPE html>  
        <html>  
            <head><title>Bottle Demo</title></head>  
            <body>  
                <h1>Bottle-Demo</h1>  
            </body>  
        </html>  
    """  
  
    return html  
...
```

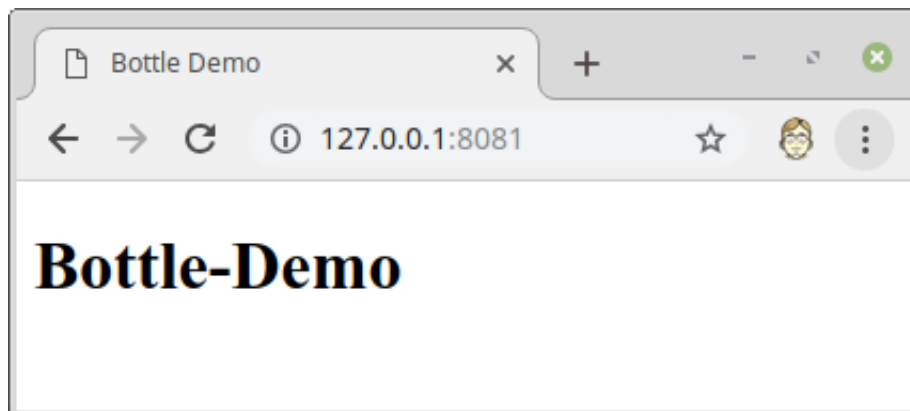


Abbildung 34: HTML-Ausgabe mit bottle

☞ Auftrag 33: Installiere Bottle auf dem eigenen Laptop und dem Pi. Erstelle dann eine Webanwendung, die unter der Route `http://localhost:8888/hallo` einen mit HTML fett-formatierten String „**Welt**“ anzeigt. Ein einfaches Hallo-Welt der Webentwicklung.

Bearbeitet am:

Dynamische Webseiten Nun soll die Seite etwas dynamischer werden und einen Zeitstempel ausgeben, der sich bei jedem Aufruf ändert.

```
...
import time

@bottle.route("/")
def index():
    html = """
        <!DOCTYPE html>
        <html>
            <head><title>Bottle Demo</title></head>
            <body>
                <h1>Bottle-Demo</h1>
                {sekseit1970} Sekunden seit dem 1.1.1970.
            </body>
        </html>
    """

    # return HTML but replace variable in string before
    t = time.time()
    return html.format(sekseit1970=t)

...
```

Wenn wir den Server nun starten, wird bei jedem Aufruf der Zeitstempel im Ergebnis ersetzt.



Bilder ausgeben

Schließlich soll noch ein Bild auf der Webseite erscheinen. Wir platzieren dafür ein einfaches Bild von einem Ball neben der Python-Datei und passen den HTML-Quelltext

so an, dass er auf das Bild verweist.

```
@bottle.route("/")
def index():
    html = """
    <!DOCTYPE html>
    <html>
        <head><title>Bottle Demo</title></head>
        <body>
            <h1>Bottle-Demo</h1>
            
        </body>
    </html>
    """

    return html

bottle.run(host="127.0.0.1", port=8081)
```

In einem ersten Schritt schlägt die Anzeige jedoch fehl. Der Statuscode 404 hinter `/img/ball.gif` weist auf eine fehlende Route für das Bild hin.

```
Bottle v0.12.13 server starting up (using WSGIRefServer())...
Listening on http://127.0.0.1:8081/
Hit Ctrl-C to quit.

127.0.0.1 - - [... 09:35:42] "GET / HTTP/1.1" 200
127.0.0.1 - - [... 09:35:42] "GET /img/ball.gif HTTP/1.1" 404
```

In der Ausgabe und im HTML-Dokument taucht eine Route `/img/` auf, für die bisher noch keine Methode angegeben wurde. Dies holen wir nun nach. Diese Methode soll jedoch kein HTML, sondern eine Bilddatei `/img/ball.gif` liefern. Hierfür können wir die Methode `static_file` im `bottle`-Modul verwenden. Mit `root` wird das Wurzelverzeichnis für Bilder angegeben - in diesem Falle mit einem Punkt „`.`“ das aktuelle Verzeichnis.

```
@bottle.route("/img/ball.gif")
def image():
    return bottle.static_file(filename="ball.gif", root=".")
```

Wir testen auch diese Methode einmal und rufen sie direkt auf. Es fällt auf, dass nun kein HTML, sondern eine Binärdatei ausgeliefert wird.

```
print(image())
```

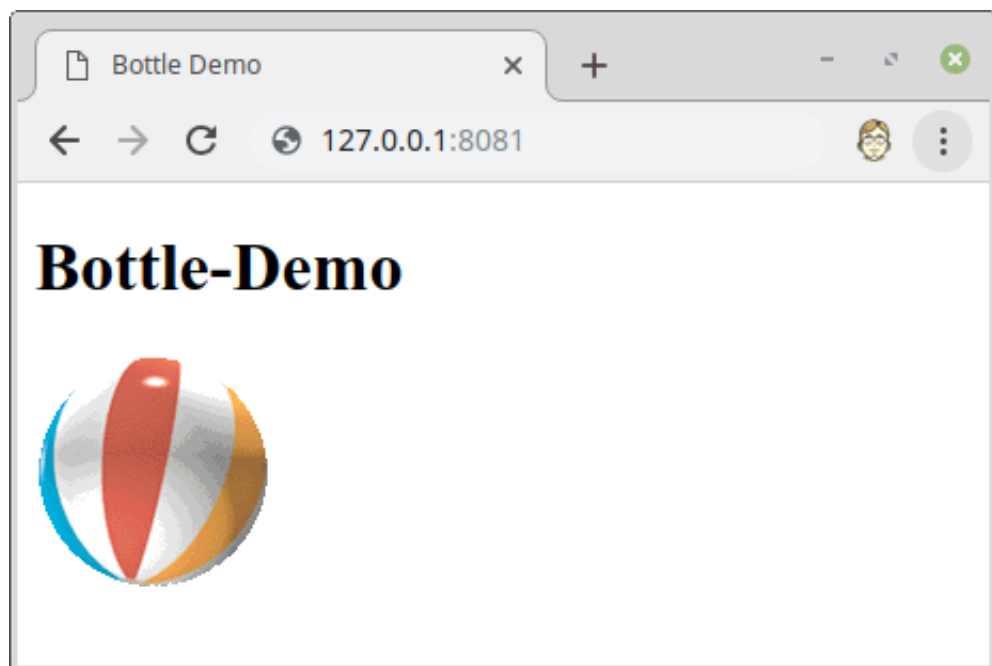
```
Content-Type: image/gif
```

```
Content-Length: 5015
```

```
Last-Modified: Thu, 09 Aug 2018 17:57:30 GMT
```

```
Accept-Ranges: bytes
```

Starten wir den Webserver nun erneut, so können wir das Bild auf der Webseite bewundern.



 Auftrag 34: Ergänze die bisher erstellte Webanwendung um ein Bild, das beim Aufruf von `http://localhost:8888/mein_bild.jpg` angezeigt wird.

Bearbeitet am:

Dynamische Routen Da es umständlich ist, für jedes Bild eine eigene Route festzulegen, können Routen auch dynamisch sein und einen Parameter enthalten – z.B. ein Dateiname für ein Bild. Dieser Parameter (in unserem Falle `dateiname`) taucht dann in der Routenbeschreibung in spitzen Klammern und in der Methode als Parameter auf.

```
@bottle.route("/img/<dateiname>") # dynamic route
def image(dateiname):
    return bottle.static_file(
        filename=dateiname, # parameter from method
        root=".")
```

Die Parameter von dynamischen Routen können sogar an einen Datentyp gebunden werden. Dies ist ein fortgeschrittenes Thema und wird in der Dokumentation von Bottle zu „Request Routing“²⁸ beschrieben.

Formulare In HTML können mit Formularen Daten an den Webserver übertragen werden. Hierfür wird das `form`-tag²⁹ in der HTML-Seite genutzt. Das `action`-Attribut darin gibt das Ziel des Formulars an. Dies ist die URL der Seite, an die die Formulardaten gesendet werden sollten. Mit `method` kann die HTTP-Methode für die Übertragung gesetzt werden. Für HTML-Formulare sind die Werte GET oder POST möglich.³⁰

Das Formular ist in Abbildung 35 dargestellt und enthält ein Eingabefeld und einen Button zum Absenden des Formulars. Es leitet mit einem POST Request von `/formular` and die Route `/formularauswerter` weiter, welche für die Auswertung der Formulardaten zuständig ist. Für beide Routen müssen wir also jeweils Methoden erstellen und diese mit einer Route verknüpfen.

Im ersten Schritt erstellen wir eine Seite, die das Formular enthält. Diese ist in Abb. 36 oben zu sehen. Im `form`-Tag des HTML-Strings wird mit `action=` die Route angegeben, an die die Formulardaten gesendet werden. Mit `name="eingabefeld"` wird der Name für das Eingabefeld festgelegt. Unter diesem Namen können wir anschließend auf die Formulardaten zugreifen.

Die zweite Route (in Abb. 36 unten) zeigt die „Gegenstelle“, welche die Auswertung der Daten übernimmt. Hierfür wird eine Methode inklusive einer Routenzuordnung angegeben. Diesmal wird jedoch `bottle.post` als Dekorator verwendet, damit diese Methode auf einen POST Request reagiert.

In der Variablen `bottle.request.forms` sind die Formulardaten zugänglich, die auf der Seite des sendenden Formulars festgelegt wurden. In unserem Beispiel also das Textfeld mit dem Namen `eingabefeld`.

Nun ist das Formular unter `http://localhost:8081/formular` erreichbar. In der Ausgabe, die nach dem Start erfolgt, ist jetzt auch deutlich die Kommunikation zwischen

²⁸<http://bottlepy.org/docs/dev/tutorial.html#request-routing>

²⁹<https://wiki.selfhtml.org/wiki/HTML/Formulare/form>

³⁰ HTTP-Methoden wurden im Abschnitt „Web-Grundlagen“ auf Seite 45 beschrieben.

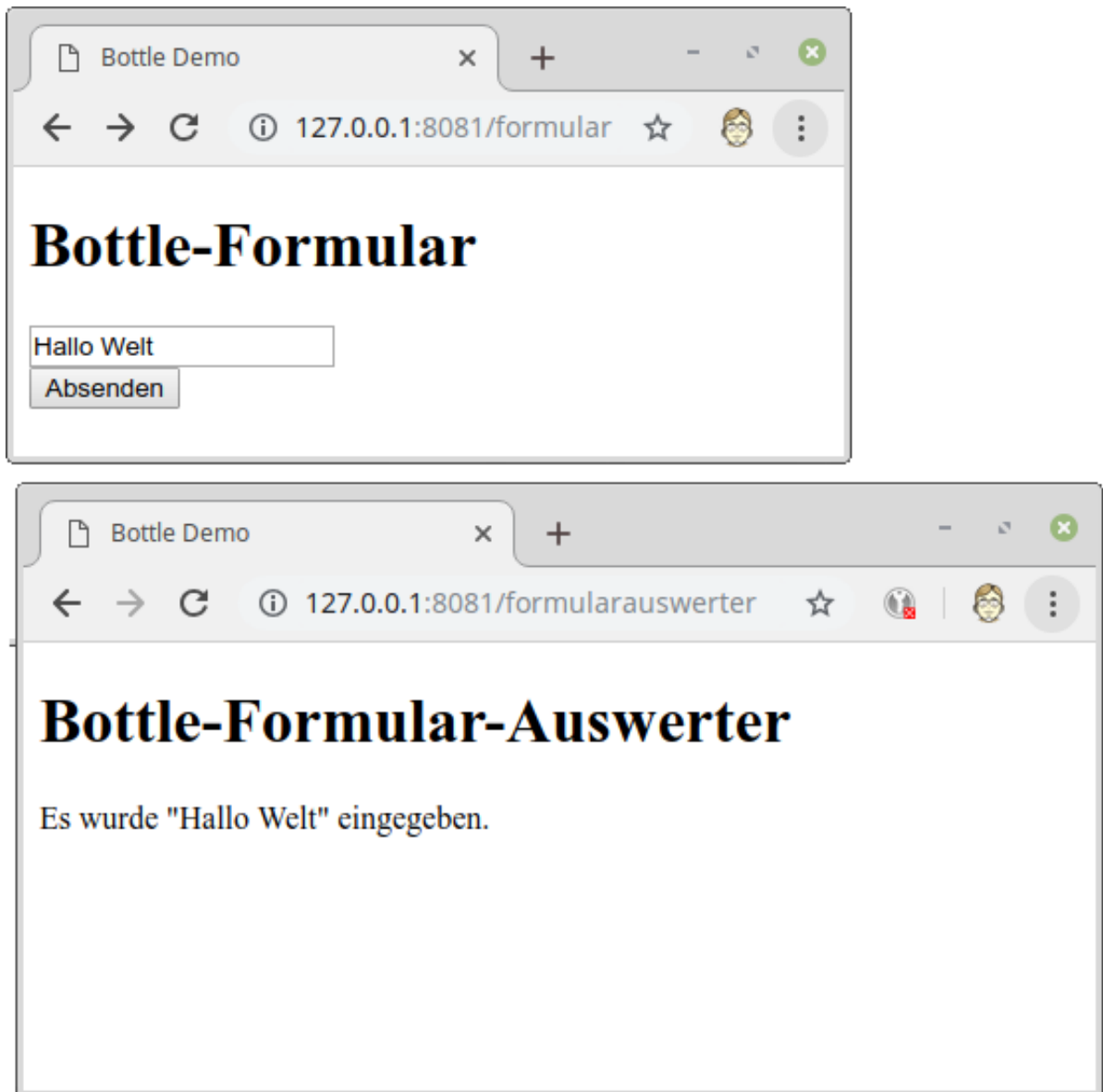


Abbildung 35: Eine Webseite mit Formularen

```
import bottle

@bottle.route("/formular")
def fomular():
    return """
    <!DOCTYPE html>
    <html>
        <head><title>Bottle Demo</title></head>
        <body>
            <h1>Bottle-Formular</h1>

            <form action="/formularauswerter" method="POST">
                <input type="textfield" name="eingabefeld"><br>
                <input type="submit" value="Absenden">
            </form>

        </body>
    </html>
    """

@bottle.post("/formularauswerter")
def fomularauswerter():
    html = """
    <!DOCTYPE html>
    <html>
        <head><title>Bottle Demo</title></head>
        <body>
            <h1>Bottle-Formular-Auswerter</h1>
            Es wurde "{req}" eingegeben.
        </body>
    </html>
    """
    formulareingabe = bottle.request.forms["eingabefeld"]

    return html.format(req=formulareingabe)


bottle.run(host="127.0.0.1", port=8081)
```

Abbildung 36: Formulare mit Bottle übertragen und auswerten


Browser und Webserver an den HTTP-Methoden GET und POST zu erkennen. Die Formulardaten sehen wir in dieser Darstellung nicht. Mit den Developertools von Firefox und Chrome könnten diese jedoch sichtbar gemacht werden.

```
Bottle v0.12.13 server starting up (using WSGIRefServer())...
Listening on http://127.0.0.1:8081/
Hit Ctrl-C to quit.
```

```
127.0.0.1 - [... 09:32:01] "GET /formular HTTP/1.1"
127.0.0.1 - [... 09:32:05] "POST /formularauswerter HTTP/1.1"
```

 Auftrag 35: Schreibe eine Web-Anwendung, die unter der Route `/eingabe` ein Namensfeld anzeigt. Bei Eingabe des Namens wird man zur Route `/hallo` geleitet, die eine Begrüßung anzeigt: „Hallo NAME“.

Bearbeitet am:

 Auftrag 36: Gehe auf die Webseite der Schule und öffne im Browser die Entwicklerkonsole. Gib dann im Suchfeld der Schul-Homepage ein Suchwort ein und versuche, den Request, der diese Anfrage überträgt, zu finden.

Bearbeitet am:

Templates dienen dem Zweck, HTML- und Python-Quelltext voneinander zu trennen. Dadurch können Änderungen am Design durchgeführt werden, ohne den Quelltext dafür ändern zu müssen. Der HTML-Quelltext wird in sogenannten Templates gespeichert während der Python-Quelltext in den Python-Dateien verbleibt.

Templates³¹ werden im Ordner `views` abgelegt. Dieser muss sich in dem Verzeichnis der Pythondatei befinden, die Bottle startet. Dort legen wir die HTML-Datei aus Abb. 37 (oben) mit dem Namen `index.tpl` ab. In dieser Datei befindet sich der Platzhalter `{{mein_text}}` im HTML-Quelltext, der noch ersetzt werden muss.

Das Template kann nun angezeigt und die Variable `mein_text` darin ersetzt werden. Dies erledigt die Methode `template` (vgl. Abb. 37 unten). Ihr wird als Parameter der Name des Template übergeben. Wichtig: Sowohl die Dateiendung `tpl` als auch der Ordner `views` werden hierbei nicht angegeben.

Nach einem Start wird das Template angezeigt und die Variable darin ersetzt.

³¹<http://bottlepy.org/docs/dev/tutorial.html#templates>

```
<!DOCTYPE html>
<html>
<head><title>Bottle Demo</title></head>
<body>
  <h1>Bottle-Template</h1>
  Hallo {{mein_text}}
</body>
</html>
```

```
@bottle.route("/")
def index():
    return bottle.template("index", mein_text="Welt")
```

Abbildung 37: Oben: HTML-Template für Bottle: `index.tpl` im Verzeichnis `views` (oben). Unten: Routendefinition für das Template

Templates mit Kontrollstrukturen werden benötigt, wenn das Template nicht nur Designelemente für die Darstellung enthält. Innerhalb des Templates können nicht nur Variablen angezeigt, sondern auch Kontrollstrukturen wie Schleifen oder Verzweigungen verwendet werden. Das ist besonders nützlich, wenn eine Liste mit Werten angezeigt werden soll.

Das nächste Beispiel zeigt ein Template in Abb. 38, in dem eine Liste durchlaufen wird. Hierfür wird Python-Quelltext mit einem vorangestellten `%` im HTML-Quelltext des Templates eingetragen. Da die sonst bei Python wichtige Einrückung im Template keine Rolle spielt, muss das Ende der Schleife mit `end` markiert werden. Zusätzlich wird die Länge der Liste mit Hilfe der Funktion `len` ausgegeben.

Beim Aufruf der Webseite wird nun eine Liste übergeben.

```
@bottle.route("/")
def index():
    return bottle.template("%index",
                           meine_liste=["eins", "zwei", "drei"])
```

Die Liste wird im Template in einer `for`-Schleife durchlaufen und jeder Eintrag wird in ein `li`-Element³² gesetzt.

³²li=list item

```
<!DOCTYPE html>
<html>
<head><title>Bottle Demo</title></head>
<body>
  <h1>Bottle-Template</h1>
  Eine Liste mit {{ len(meine_liste) }} Elementen
  <ul>
    % for eintrag in meine_liste:
      <li>{{ eintrag }}</li>
    % end
  </ul>
</body>
</html>
```

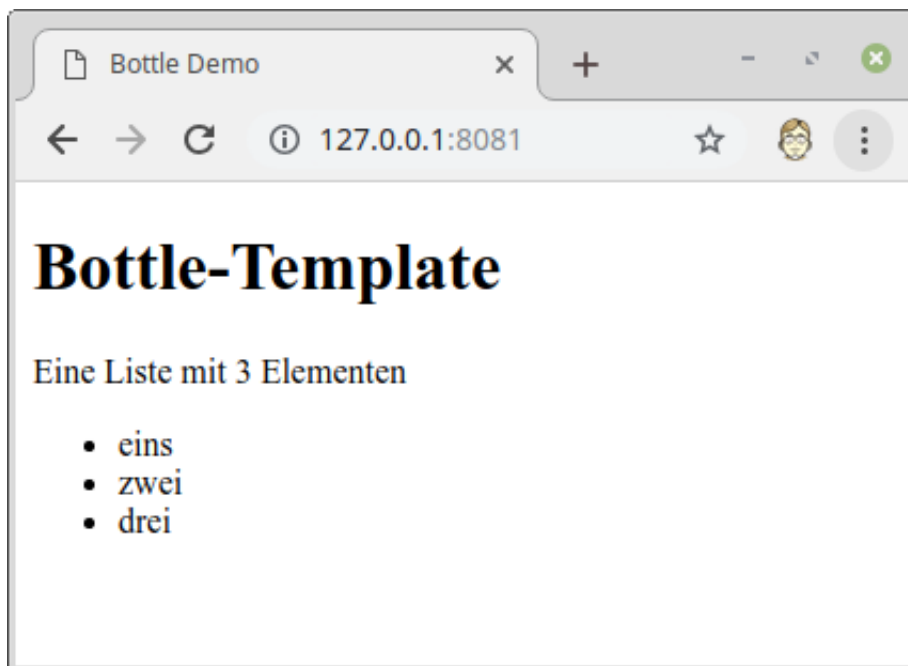



Abbildung 38: Ein HTML-Template mit Kontrollstrukturen

Mini-Projekt: Todo-Liste

 Auftrag 37: Erstelle mit Bottle eine App für die Verwaltung von Todos. Über ein Eingabefeld kann ein Todo der Liste hinzugefügt werden. Mit einem Click auf den Link „erledigt“ wird das TODO aus der Liste entfernt. Die Todos können in einer temporären Variable, einer Datei oder Datenbank verwaltet werden.

Bearbeitet am:

Der HTML-Quelltext und ein mögliches Design der Seite sind in Abbildung 39 abgebildet. Die Seite enthält die Todo-Liste und das Formular, um Todos hinzuzufügen. Auch ein Link zum Entfernen ist angegeben, der auf eine dynamische Route verweist.

TODO Liste

- [\[erledigt\]](#) Todo App erstellen
- [\[erledigt\]](#) Wichtige Dinge aufschreiben

```
<!DOCTYPE html>
<html lang="de">
<head>
  <meta charset="UTF-8">
  <title>TODO</title>
</head>
<body>
  <h1>TODO Liste</h1>

  <ul>
    <li>
      <a href="/del/0">[erledigt]</a>
      Todo App erstellen
    </li>
    <li>
      <a href="/del/1">[erledigt]</a>
      Wichtige Dinge aufschreiben
    </li>
  </ul>

  <form action="/" method="post">
    <input type="text" name="new_todo" />
    <input type="submit" value="TODO erstellen">
  </form>
</body>
</html>
```

Abbildung 39: Webseite und HTML-Quelltext für eine Todo-Liste

Fehlersuche

Im Zusammenspiel von Hard- und Software gibt es eine Reihe von Fehlerquellen, die man beachten sollte. Versuchen wir gedanklich von der Idee zur Umsetzung, die Quellen von Fehlern nach und nach zu identifizieren.

„Teile und Herrsche“ ist ein allgemeines Prinzip, nach dem ein Problem in kleine Teile aufgeteilt und anschließend die Teile einzeln analysiert – also beherrscht – werden. Bei jeder Fehlersuche ist es daher ratsam nach und nach die Bestandteile des Problems zu isolieren und einzeln zu betrachten.

Für ein Physical-Computing-Projekt bedeutet dies, dass es auf der obersten Ebene grob in die beiden Aspekte Hard- und Software aufgeteilt werden kann. Aber auch die Hard- und Software kann ihrerseits weiter untergliedert werden. Bei der Hardware wären die Bauteile, bei der Software Klassen die nächste Ebene.

Bei der Fehlersuche versucht man nun, unterschiedliche Bereiche zu identifizieren, die Probleme bereiten könnten. Danach können nach und nach die Probleme in diesen Bereichen ausgeschlossen oder der Fehler dort gefunden werden.

Im Folgenden schauen wir uns verschiedene dieser Bereiche an und stellen Möglichkeiten vor, Probleme in diesen Bereichen zu entdecken.

Die Projektidee ist die erste Quelle von möglichen Fehlern. Hier können logische Fehler sichtbar werden. Das sind Denkfehler, die entstehen, wenn man das Problem, das man lösen möchte, noch nicht vollständig verstanden hat.

Typische Fragen, die man sich stellen sollte, wären: Habe ich genug Zeit? Welche Bauteile könnten mein Problem lösen? Was tun die Bauteile? Welche GPIO-Pins benötige ich am Pi? Welche weiteren Bauteile benötige ich? Habe ich alle Randfälle bedacht? Benötige ich ein Gehäuse oder eine Platine?

Hier können auch Fehler auftreten, wenn man eine Annahme trifft, die gar nicht zu zutreffend ist.

Der Schaltplan ist die nächste Fehlerquelle. An dieser Stelle trifft man wichtige Entscheidungen darüber, wie Sensoren, Aktoren und der Raspberry Pi oder ein Mikrocontroller miteinander verbunden werden sollen. Dies schließt insbesondere die gewählten Pin-Nummern am Pi und am Sensor oder Aktor mit ein.

Eventuell werden weitere Bauteile wie Widerstände, Kondensatoren, Transistoren, Spulen, eine Spannungsversorgung, etc. benötigt. Die Bauteile gibt es in verschiedenen Dimensionierungen oder Ausführungen. Auch diese Werte werden im Schaltplan

festgelegt.

Manche Bauteile benötigen einen definierten Zustand an den Pins. Dies hat zur Folge, dass ggf. alle Pins beschaltet werden müssen. Ein nicht verbundener Pin ist nicht das gleiche wie ein Pin, der mit Masse verbunden ist. Durch statische Aufladungen kann ein solcher Pin mal ein High- und mal ein Low-Signal erhalten. Ist dieser Pin für das Zurücksetzen des Bauteils zuständig, so zieht dies unvorhersehbare und auch schwer zu findende Fehler nach sich. Ein Blick in das Datenblatt hilft dabei, zu entscheiden, wie die Pins jeweils zu beschalten sind.

Die aufgebaute Schaltung folgt dem Schaltplan und setzt diesen in der Realität um. Dies kann auf einem Steckbrett oder einer Platine erfolgen. Die Leitungen müssen nun mit Steckbrücken und die Knotenpunkte mit dem Steckbrett nachgebildet werden.

Nun können physikalische Probleme auftauchen: Kabel können defekt und Verbindungen mit dem Steckbrett wackelig sein.

Ein Multimeter kann mit einer Durchgangsprüfung helfen, Fehler in den Verbindungen zu finden. Es hilft auch, ein High- oder Low-Signal an einer Leitung oder einem Pin zu entdecken. Wenn kein Multimeter zur Hand ist oder mehrere Punkte betrachtet werden, hilft eine LED an der richtige Stelle in der Schaltung dabei, einen Hinweis auf die anliegenden Pegel zu geben.

Beim Verbindungen von Steckbrücken mit dem Pi können Abzählfehler entstehen. Die Wahl der GPIO-Pins kann hierbei helfen. Ein Block zusammenhängender Pins ist optisch leichter zu prüfen als beliebige Pins auf der GPIO-Leiste.

Defekte in der Hardware können immer wieder auftreten und sollten entsprechend berücksichtigt werden. Der Abschnitt „Bauteile prüfen (TODO)“ auf Seite 25 beschreibt wie einzelne Bauteile systematisch und isoliert getestet werden können, bevor sie in einer größeren Schaltung integriert werden. Eine Prüfung eines konkreten Bauteils sollte daher immer erfolgen.

Um ein Programm auch ohne die Hardware testen zu können, bietet sich ein Dummy für die Hardware an. Dies ist ein Programm, das z.B. selbst generierte aber eben nicht gemessene Werte für einen Sensor liefert. Diese können z.B. an einen Broker gesendet werden, so wie es auch die echte Hardware tun würde. Der Rest des Programms kann mit diesen Daten bereits getestet werden, selbst wenn die Hardware defekt oder nicht vorhanden ist. Im Abschnitt „Software-Dummies“ auf Seite 25 wird das Vorgehen genauer beschrieben.

Die Buskommunikation ermöglicht Bauteilen eine Kommunikation mit dem Pi oder einem Mikrocontroller. Als prominente Vertreter sind I²C und SPI in diesem Skript ab Seite 17 beschrieben. Durch eine falsche Beschaltung, z.B. durch das Vertauschen von Pins, kann diese Kommunikation gestört werden.

Die Netzkommunikation ermöglicht Programmen das Versenden und den Empfang von Daten. Bei Physical-Computing-Projekten werden die Protokolle MQTT und HTTP³³ häufig genutzt. Bei der Verwendung sollte die physikalische Netzwerkverbindung im Falle von Fehlern geprüft werden. Hierbei ist eine verkabelte Verbindung immer einem Versand über WiFi vorzuziehen. Auch die Konfiguration kann Fehler produzieren und sollte geprüft werden. Hier spielen Portnummern für HTTP und MQTT bzw. Topicnamen im Falle von MQTT eine zentrale Rolle.

Das Betriebssystem auf dem Raspberry Pi (Raspbian) fungiert als Steuerzentrale für die Schaltung. Es sollte auf einem aktuellen Stand und nicht durch eine defekte Konfiguration beeinträchtigt sein.

Im Hintergrund laufen vielleicht noch andere Programme, die Probleme verursachen und entsprechend beendet werden sollten. Laufende Python3-Programme lassen sich z.B. mit `pgrep python3` auflisten und mit `pkill python3` beenden. Wenn ein Programm im Hintergrund eine Ressource wie etwa einen Port blockiert, lässt sich dieses Problem etwas durch Beenden des entsprechenden Prozesses häufig lösen.

Auch ein Neustart des Betriebssystems kann häufig hartnäckige Probleme lösen.

Beteiligte Systeme sind andere Systeme, die für die Projektumsetzung relevant sind. Dies könnten ein Webdienst oder MQTT-Broker sein. Der Status des Brokers kann mit `service mosquitto status` geprüft werden. Ein Neustart ist mit `service mosquitto restart` möglich.

Das Programm, welches für die Projektumsetzung geschrieben wurde kann Fehler auf ganz unterschiedlichen Ebenen, enthalten. Wir betrachten ein paar wichtige Gruppen von Fehlerquellen innerhalb des Programms genauer.

Konfigurationsfehler betreffen die Festlegung von Portnummern, Dateinamen oder auch Zuordnungen von GPIO-Pins. Welche Zahlen auch immer für das Programm relevant sind, sollten in einer separaten Datei `config.py` ausgelagert und von anderen Dateien importiert werden. Über die sinnvolle Wahl von Namen für z.B. GPIO-Pins wird klar,

³³Details befinden sich in diesem Skript ab Seite 28 für MQTT und ab Seite 45 für HTTP.

welche Bedeutung eine Zahl in diesem Fall haben soll. Für einen Pin, der als Clock-Pin an einem I²C Bus einen Temperatursensor ausliest, wäre `GPIOPIN_CLK_TEMPERATURE` eine bessere Bezeichnung als schlicht die Zahl 5.

Syntaxfehler lassen sich schnell erkennen, da sie in einer IDE angezeigt werden und auch ein Programmstart die betroffenen Zeilen direkt nennt. Dies können fehlende Klammern, Doppelpunkte, Anführungszeichen, etc. sein.

Mächtige IDEs wie „PyCharm“ oder „VisualStudioCode“ zeigen häufig Fehler direkt an, wenn man den Quelltext darin öffnet. Ein sinnvolle Strategie ist es also, den Quelltext, der sich auf dem Pi befindet, im Problemfall auf den eigenen Laptop zu übertragen und dort zu prüfen.

Laufzeitfehler treten im laufen Betrieb auf. Hier könnte ein Problem mit der Netzwirkkommunikation, eine fehlende Bibliothek oder eine ganze Menge anderer Probleme auftauchen. Wichtig ist, die Fehlermeldung genau zu lesen und die Stelle im Quelltext zu finden und nachzuvollziehen. In vielen Fällen ist die Fehlermeldung aussagekräftig genug, um das Probleme zu lösen. Bei Problemen mit englischen Begriffen hilft auch häufig ein Blick in ein Wörterbuch.

Logische Fehler führen zu keinem Programmabsturz und sind daher auch leider schwerer zu entdecken. Hier hilft ein kleinschrittiges Vorgehen. Man beginnt sein Programm mit einer kleinen einfachen Funktionalität und erweitert es im Anschluss nach und nach in kleine Schritten. So kann ein logischer Fehler, wenn er sich einschleicht auf die letzte Änderung eingegrenzt werden.

Um einzelne Werte nachvollziehen zu können, helfen `print`-Anweisungen, um Ausgaben auf der Konsole zu machen. Sie helfen auch, wenn man unsicher ist, ob eine Methode überhaupt aufgerufen oder eine Verzweigung in einer Bedingung betreten wird. Eine Ausgabe an der richtigen Stelle, kann hier weiterhelfen. Danach kann man dies Anweisungen entweder wieder löschen oder auskommentieren.

Eine Checkliste für die Fehlersuche ist sinnvoll und hilft, Fehler systematisch zu finden. Eine Zusammenfassung der bisher genannten Probleme befindet sich in Abb. 40. Sobald ein Fehler auftritt, sollte man die Checkliste von vorne nach hinten abarbeiten.

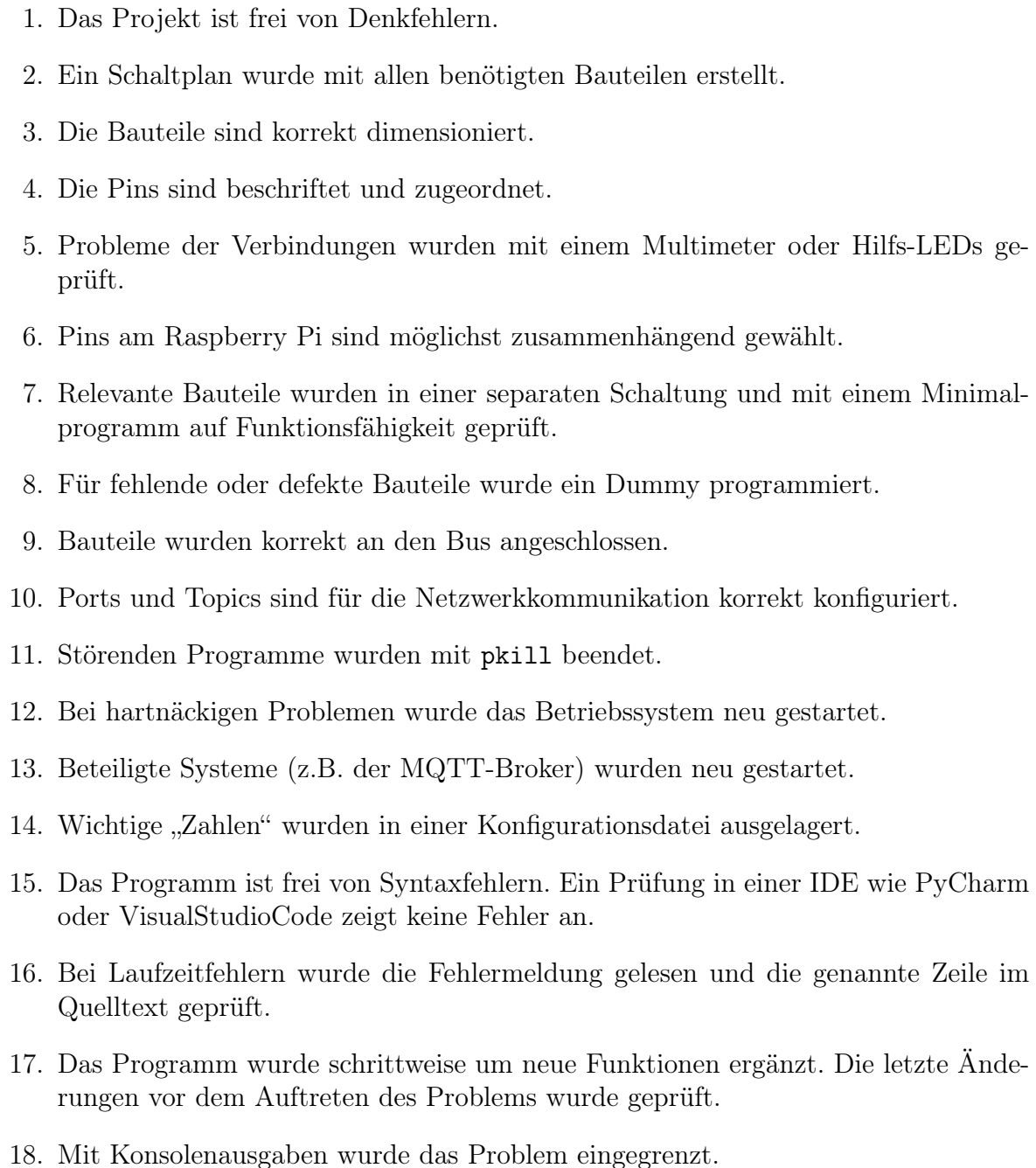
-
- 
1. Das Projekt ist frei von Denkfehlern.
 2. Ein Schaltplan wurde mit allen benötigten Bauteilen erstellt.
 3. Die Bauteile sind korrekt dimensioniert.
 4. Die Pins sind beschriftet und zugeordnet.
 5. Probleme der Verbindungen wurden mit einem Multimeter oder Hilfs-LEDs geprüft.
 6. Pins am Raspberry Pi sind möglichst zusammenhängend gewählt.
 7. Relevante Bauteile wurden in einer separaten Schaltung und mit einem Minimalprogramm auf Funktionsfähigkeit geprüft.
 8. Für fehlende oder defekte Bauteile wurde ein Dummy programmiert.
 9. Bauteile wurden korrekt an den Bus angeschlossen.
 10. Ports und Topics sind für die Netzwerkkommunikation korrekt konfiguriert.
 11. Störenden Programme wurden mit `pkill` beendet.
 12. Bei hartnäckigen Problemen wurde das Betriebssystem neu gestartet.
 13. Beteiligte Systeme (z.B. der MQTT-Broker) wurden neu gestartet.
 14. Wichtige „Zahlen“ wurden in einer Konfigurationsdatei ausgelagert.
 15. Das Programm ist frei von Syntaxfehlern. Ein Prüfung in einer IDE wie PyCharm oder VisualStudioCode zeigt keine Fehler an.
 16. Bei Laufzeitfehlern wurde die Fehlermeldung gelesen und die genannte Zeile im Quelltext geprüft.
 17. Das Programm wurde schrittweise um neue Funktionen ergänzt. Die letzte Änderungen vor dem Auftreten des Problems wurde geprüft.
 18. Mit Konsolenausgaben wurde das Problem eingegrenzt.

Abbildung 40: Checkliste für die Fehlersuche