13 Ausnahme-Situationen behandeln

Bei der Ausführung unserer Server-Programme zum Steuern der LED können unterschiedliche Probleme auftreten, die zu einem Abbruch oder Fehlverhalten des Programms führen. Drei von diesen Ausnahmesituation wollen wir hier behandeln:

- Der ESP32 kann sich nicht mit dem Access Point verbinden.
- Der ESP32 kann das Binding nicht durchführen.
- Wenn mehrere Browser gleichzeitig zum Steuern der LED benutzt werden, können sie sich ggf. gegenseitig blockieren.

Die im Folgenden besprochenen Änderungen bzw. Ergänzungen können bei beiden Programm-Dateien, `sta_led_server_0.py` und `sta_led_server_1.py`, vorgenommen werden.

Wlan-Verbindung zum Access Point

Eine Verbindung zum Access Point kann nicht zustande kommen, wenn der SSID bzw. das Passwort falsch angegeben worden sind. In diesem Fall sollte der Nutzer auf dieses Problem explizit hingewiesen werden. Dazu ändern wir die Verbindungsüberprüfung folgendermaßen ab:

```
if not station.isconnected():
  print('Wlan not connected! SSID and password correct?')
else:
  print(station.ifconfig())
  ...
```

Damit haben wir natürlich nicht auf alle möglichen Fehlerquellen hingewiesen. Es dürfte sich hier aber um die häufigste handeln.

Fehler beim Binding

Wir hatten in Kapitel 8 schon darauf hingewiesen, dass es bei einem erneuten Start des Server-Programms zu einem Abbruch kommen kann; dabei wird im Terminal darauf hingewiesen, dass es sich um einen **OSError** (Betriebssystem-Fehler) handelt, der bei der Ausführung von

```
s.bind(('', 80))
```

aufgetaucht ist. Diese Fehlermeldung ist für den Anwender wenig hilfreich. Unser Programm sollte hier einen konkreteren Hinweis geben. Leider beendet der ESP32 mit dem Auftreten des Fehlers das Programm. Nun ist es zu spät, um noch eine entsprechende Meldung am Terminal auszugeben.

Eine Reihe von Fehlern rufen sogenannte **Exceptions** (Ausnahmen) hervor (vgl. K4). Diese können nicht nur durch Fehler, sondern auch anderweitig erzeugt werden, z. B. beim Abbrechen eines Programms mit Ctrl-C.

Mit der **`try-except-Konstruktion`** können wir solche Exceptions abfangen und dadurch auf die jeweilige Situation reagieren. Wie man dabei vorgeht, schauen wir uns konkret für den vorliegenden Fall an:

```
...
# Server bereit stellen
s = socket.socket()

try:
  s.bind(('', 80))
  s.listen(5)
  print('Server ready...')
  weiter = True
except OSError:
  print('OSError when binding')
  print('Server not ready!')
  print('Please reboot!')
  exit() # Programmende

# Daten empfangen und senden...
while True:
  ...
```

In dem **try-Block** stehen die Befehle, bei denen es gegebenenfalls zu einer Ausnahme kommen kann: Wenn einer der dort auftauchenden Befehle eine Exception hervorruft, dann wird dieser Befehl abgebrochen und das Programm springt sofort zum except-Teil (ohne dass eine Fehlermeldung vom Micropython-System im Terminal ausgegeben wird).

Hinter dem Schlüsselwort except steht – und das ist jetzt neu – eine weitere Angabe, nämlich `OSError`. Diese Angabe kennzeichnet den Exception-Typ. Sie sorgt dafür, dass der **except-Block** nur ausgeführt wird, wenn die Exception vom Typ `OSError` ist. Falls im try-Block eine Exception mit einem anderen Typ auftritt, dann greift unsere try-except-Konstruktion nicht: Das Programm verhält sich dann so, als ob sie gar nicht vorhanden wäre. Würde hinter `except` kein Exception-Typ stehen, dann würde der except-Block bei jeder Art von Exception ausgeführt (vgl. K4).

Wenn nun eine derartige OSError-Exception ausgelöst wird, dann werden im Terminal die Hinweise 'OSError when binding', 'Server not ready!' und 'Please reset!' ausgegeben. Das gesamte Programm wird anschließend durch den Befehl **`exit()`** beendet; diese `exit`-Funktion muss übrigens zu Beginn des Programms aus dem `sys`-Modul importiert werden. Eine Fehlermeldung vom Micropython-System gibt es jetzt nicht mehr.

LEDs mit mehreren Clients gleichzeitig steuern

Eigentlich sollte es kein Problem sein, wenn zwei (oder noch mehr) Browser (quasi-)gleichzeitig zum Steuern der LED eingesetzt werden. Schließlich sollte der Server auf jeden Request reagieren, egal von welchem Browser er kommt.

Es gibt nun aber Browser (z. B. der Chrome-Browser auf meinem Android-Handy), die eine merkwürdige Eigenart besitzen: Nachdem sie die Response vom Server erhalten haben, zeigen sie die Webseite wie erwartet an. Was man vielleicht nicht erwartet, ist: Direkt im Anschluss daran bauen sie *automatisch* wieder eine Verbindung zum Server auf; ein Request wird jedoch erst dann abgesendet, wenn der Nutzer einen entsprechenden Link (oder Button) betätigt. In der Zwischenzeit wartet der Server mit der Funktion `recv` auf Daten. Und solange er keine Daten empfängt, kann auch die Schleife nicht weiter durchlaufen werden. Deswegen kann in dieser Zeit auch keine neue Verbindung – etwa zu einem anderen Client (Browser) – hergestellt werden: der Server ist für andere Clients blockiert!

Es ist müßig darüber nachzudenken, warum mein Handy-Chrome-Browser so agiert; vielleicht steckt dahinter die Absicht, weitere Requests möglichst rasch erledigen zu können. Viel wichtiger ist: Wie kann man einer derartige Blockade begegnen?

Nun, ein erster Schritt besteht darin, in der `getrequest`-Funktion für das Socket-Objekt `connection` ein **Timeout** festzulegen. Schreiben wir

```
connection.settimeout(5.0)
```

vor die Schleife mit dem Lesevorgang, dann wird bei jeder Ausführung der `recv`-Methode maximal 5 Sekunden auf das Eintreffen von Daten gewartet. Treffen die Daten nicht rechtzeitig ein, so wird eine Exception vom Typ `OSError` mit dem Error-Code 110 ausgelöst.

Sämtliche Befehle, angefangen vom Empfangen des Request bis hin zum Senden der HTML-Datei, schieben wir nun in einen try-Block:

```
try:
  request = getrequest(connection)
  ...
  connection.send(html())
```

Gibt es ein Timeout beim Empfang des Requests, werden die restlichen Befehle des try-Blocks nicht ausgeführt. Stattdessen geht es dann weiter zum except-Teil:

```
except OSError as e:
  errno = e.args[0]
  if errno == 110: # Timeout
      print('getrequest-timeout: OSError 110')
    else:
      print('OSError: ',errno)
```

Schauen wir uns zunächst die except-Zeile an: Hier finden wir hinter `except OSError` den Zusatz **`as`** e. Was bedeutet dieser Zusatz? Als im try-Teil ein Timeout aufgetreten ist, wurde ein OSError-Objekt erzeugt. Dabei erhielt dieses Objekt bestimmte Eigenschaften, ganz ähnlich wie es z. B. bei der Erzeugung eines Pin-Objektes geschieht: Wie das Pin-Objekt eine Pin-Nummer (und weitere Eigenschaften) erhält, so erhält das OSError-Objekt eine Fehlernummer (`errno`), in unserem Fall 110. Mit dem Zusatz `as e` definieren wir eine Variable, welche auf dieses OSError-Objekt verweist. Im except-Block greifen wir nun auf die in `e` gespeicherte Fehlernummer zu: Sie ist als erstes Element in dem Argumente-Tupel `args` zu finden: `errno = e.args[0].`

Ist die Fehlernummer tatsächlich 110, dann lassen wir eine entsprechende Meldung ausgeben; andernfalls wird die vorliegende Fehlernummer ausgegeben. Wer mag, kann für diesen Fall noch eine detailliertere Fehler-Behandlung implementieren.

Egal ob es im try-Block nun einen OSError gegeben hat oder die Befehlsfolge dort regulär durchlaufen werden konnte, auf jeden Fall muss am Ende die Verbindung wieder geschlossen werden. Dies geschieht mit Hilfe eines **`finally`**-Teils, den wir unter den except-Teil schreiben:

```
finally:
  connection.close()
```

Alle drei Teile der try-except-finally-Konstruktion befinden sich innerhalb der Endlos-Schleife. Deswegen wird eine Blockade, so wie sie oben geschildert wurde, höchstens 5 Sekunden dauern. Mit Beginn des nächsten Schleifendurchlaufs kann dann wieder eine neue Verbindung – egal mit welchem Client (Browser) – aufgebaut werden.

Das so erweiterte Programm finden Sie unter: `sta_led_server_2.py` (Button-Version).