Programmieren eingebetteter Systeme

MicroPython

Sebastian Stigler

- MicroPython
- ESP8266
- Installation
- Interagieren mit Peripherie

MicroPython

Was ist MicroPython

» MicroPython zielt darauf ab, Python auf Microcontroller zu bringen.

» Es setzt den Python 3.4 Standard mit einigen Features neuerer Versionen in punkto Syntax um. Die meisten Funktionalitäten sind (wenn auch abgespeckt) identisch zu denen in CPython.

» MicroPython beinhaltet ein Auswahl an Standardmodulen, die für den Microcontroller optimiert und verschlankt wurden. Man erkennt Sie am vorangestellten kleinem `U`. Soweit umgesetzt, funktionieren diese Module wie die CPython Module. Mit `help('modules')` kann man sich alle vorhandenen Module anzeigen lassen.

Kategorien mit einigen Beispielen [siehe: 1]:

- Syntax
  - Spaces (req. space between lit. numbers and keywords)
  - Unicode (no name escapes `\N{LATIN SMALL LETTER A}`)
- Core Language
  - Classes (no `__del__`, multiinheritance not fully sup.)
  - Functions (no user-def. attributes)
  - Generator
  - Runtime
  - import
- Builtin Types
  - Exception (no exc. chaining, no user-def. attributes . . .)
  - bytearray (support `.format()` methode)
  - bytes
  - float
  - int
  - list
  - str
  - tuple
- Modules
  - array
  - builtins
  - deque
  - json (no exception when obj. is not serialisable)
  - struct (`pack` doesn't check for to few or to many args)
  - sys

ESP8266

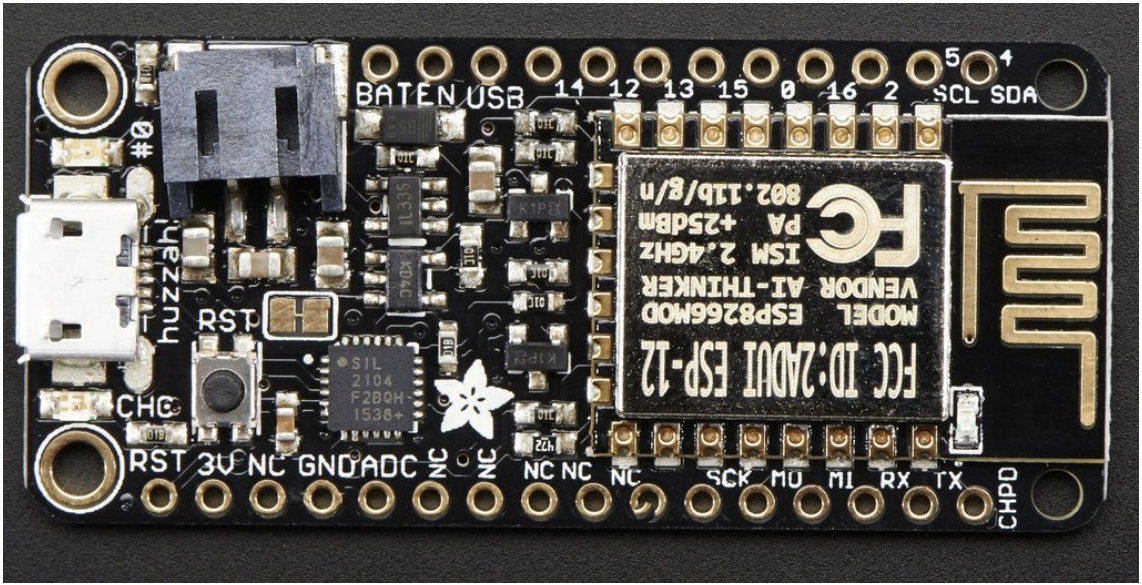

Abbildung 1: Adafruit Feather HUZZAH ESP8266

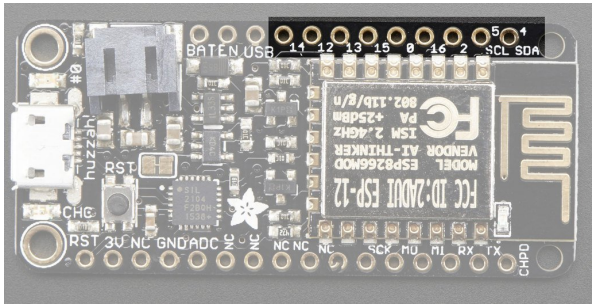

Abbildung 2: GPIOs

» Es gibt 9 GPIOs: **0**, **2**, **4**, **5**, **12**, **13**, **14**, **15**, **16**. Die meisten dieser Pins haben einen, per Software zuschaltbaren, Pullup Widerstand.
» Alle GPIOs sind **nicht 5V kompatible**, sondern dürfen nur mit 3.3V betrieben werden (In/Out).
» **GPIO 0** hat **keinen Pullup** und ist mit der LED oberhalb des USB Anschlusses verbunden. Achtung die LED leuchtet, wenn der Pin LOW ist!
» **GPIO 2** ist mit der Blauen LED auf dem WLAN Modul verbunden und hat einen **festen Pullup** verbaut. Auch hier geht die LED an, wenn der Pin LOW ist.
» **GPIO 15** hat einen fest verbauten Pulldown Widerstand.
» **GPIO 0, 2, 15** werden auch benutzt um den Betriebsmodus der ESP8266 beim Booten zu wählen. Sie sollten nur als Output verwendet werden (außer man weiss genau was man tut!)
» **GPIO 16**, wenn er mit RESET verbunden ist, wird verwendet um aus den Deep-sleep Modus auf zuwachen.

Installation

Notwendige Programme

Erstellen und aktivieren einer virtuellen Umgebung[1]:

```
$ python3 -m venv pesvenv
```

Aktivieren einer virtuellen Umgebung:

```
$ . ./pesvenv/bin/activate
```

Nun sollte am Anfang des Prompts `(pesvenv)` stehen.
Installieren von `wheel`[2], `esptool` und `rshell` in die virtuelle Umgebung:

```
(pesvenv)$ pip install wheel
(pesvenv)$ pip install esptool
(pesvenv)$ pip install rshell
```

[1] Evtl. muss noch mit `sudo apt-get install python3-venv` das venv Modul installiert werden
[2] ist Vorraussetzung für `esptool` und `rshell`

Download und Installation von MicroPython

Link: https://micropython.org/download
Dort zum Abschnitt *ESP8266* gehen und die aktuellste stabile Version im `.bin` Format herunterladen (z.B. `esp8266-20190125-v1.10.bin`)
Mit den folgenden Kommandos (in der virtuellen Umgebung) wird der *ESP8266* gelöscht und anschließend mit MicroPython geflashed:

```
(pesvenv)$ esptool.py --port /dev/ttyUSB0 erase_flash
(pesvenv)$ esptool.py --port /dev/ttyUSB0 --baud 115200 write_flash \
                      --flash_size=detect 0 esp8266-20190125-v1.10.bin
```

Mit Hilfe von von `rshell` kann man u.a.[siehe: 2]:

- Die Dateien auf den Microcontroller und Hostrechner anzeigen lassen (`ls`)
- MicroPython Dateien auf den Mircocontroller kopieren (`cp`)
- Dateien editieren (`edit`[3])
- Eine interactive Konsole ausführen (`repl`[4])

Eine Verbindung kann mit folgendem Kommando aufgebaut werden

```
(pesvenv)$ rshell -p /dev/ttyUSB0
```

Ein Besonderheiten beim *ESP8266*

Das lokale Verzeichnis auf dem Microcontroller kann **nicht** über `/flash` sondern über `/pyboard` erreicht werden!

[3] Über `sudo update-alternatives --config editor` oder die Umgebungsvariable `EDITOR` setzt man den default Editor

[4] Read Eval Print Loop

Interagieren mit Peripherie

Pin Klasse kontrolliert I/O Pins


```
from machine import Pin


# create an output pin on pin #0
p0 = Pin(0, Pin.OUT)

# set the value low then high
p0.value(0)
p0.value(1)

# create an input pin on pin #2, with a pull up resistor
p2 = Pin(2, Pin.IN, Pin.PULL_UP)

# read and print the pin value
print(p2.value())

# reconfigure pin #0 in input mode
p0.mode(p0.IN)

# configure an irq callback
p0.irq(lambda p:print(p))
```

Praktische Probleme beim Einlesen eines Pins

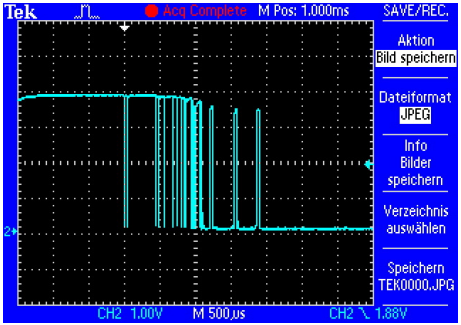

Abbildung 3: Joystick: Prellen beim Betätigen

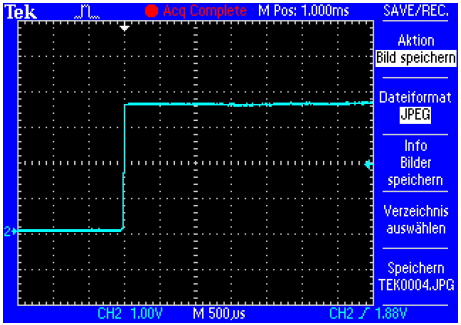

Abbildung 4: Joystick: Prellen beim Loslassen

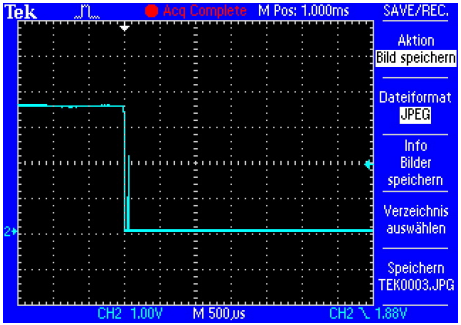

Abbildung 5: Feuertaste: Prellen beim Betätigen

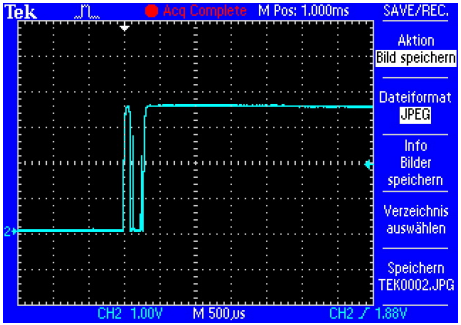

Abbildung 6: Feuertaste: Prellen beim Loslassen

In Mensch-Computer-Interaktion lernt man, dass eine Verzögerung von weniger als 100ms nicht als solche wahrgenommen wird.

Idee zum Entprellen (engl. debouncing)

- Lese den Pin alle 6 ms aus.
- Merke die letzen 12 Werte.
- Wenn diese alle gleich sind, dann ist das der tatsächliche Wert des Pins.

Damit wird die Reaktion auf ein Signal im besten Fall um $12 \cdot 6ms$ also $72ms$ verzögert. Somit kann der Schalter immer noch $28ms$ prellen, ohne dass der Benutzer ein Verzögerung wahrnimmt.

Die Timer Klasse


```
from machine import Timer


# create a virtual (software) Timer
tim = Timer(-1)  # negativ id indicates virtual timer

# some callback
def my_callback(obj):
    print(obj)

# initialize a periodic timer with a dutycycle of 500ms
# to initialize a one shot timer use Time.ONE_SHOT as mode
tim.init(period=500, mode=Timer.PERIODIC, callback=my_callback)

# stop the timer
tim.deinit()
```

Interrupt Service Routine (ISR)

» Halte den Code so kurz und einfach wie möglich.
» Vermeide Speicher allokierung: Kein Anhängen an Listen, Einfügen in Dicts oder Fließkomma Arithmetik.
» Benutze `micropython.schedule` als Workaround wenn man Speicher allokieren muss.
» Wenn die IRS mehrere Bytes mit dem Hauptprogramm austauscht, dann sollte man ein `bytearray` im Hauptprogramm anlegen. Bei Integers kann man ein `array` (`array.array`) verwenden.
» Allokiere einen Notfallbuffer für Exceptions

```
import micropython
micropython.alloc_emergency_exception_buf(100)
```

```
from machine import I2C, Pin


# init i2c with the scl pin and the sda pin
scl = Pin(5)
sda = Pin(4)
bus = I2C(scl=scl, sda=sda, freq=400000)  # freq=400000 is default value

# scan the i2c bus for slaves
i2c.scan()

# write 3 bytes to slave with 7-bit address 42
i2c.writeto(42, b'123')
i2c.writeto(42, bytes([0x31, 0x32, 0x33])  # like the line above
                                           # but with hex values


# read 4 bytes from slave with 7-bit address 42 into data
data = i2c.readfrom(42, 4)
```

Network: In ein bestehendes WLAN einwählen


```
from machine import idle
from network import WLAN, STA_IF


SSID = 'MyNetworkName'
PASSWORD = '12345678'

wifi = WLAN(STA_IF)
wifi.active(True)
wifi.connect(SSID, PASSWORD)
while not wifi.isconnected():
    idle()
print('Connection to wifi:', wifi.ifconfig())
```

Network: Den eingebauten AP deaktivieren

Ans Ende (!!) der Datei `boot.py`

```
from network import WLAN, AP_IF


ap_if = WLAN(AP_IF)
ap_if.active(False)
```

Network: Mac addresse


```
from network import WLAN
import ubinascii


mac = ubinascii.hexlify(WLAN().config('mac'), ':').decode()
print(mac)
```

MQTT: publish


```
from umqtt.simple import MQTTClient


# mqtt broker ip 1.2.3.4,  port 1883
client = MQTTClient('ClientID', '1.2.3.4', port=1883)
client.connect()

client.publish(b'topicname', b'message')

# [...]
client.disconnect()
```

MQTT: subscribe


```
# import time  # for non-blocking wait
from umqtt.simple import MQTTClient


def sub_cb(topic, message):
    print((topic, message))


# mqtt broker ip 1.2.3.4,  port 1883
client = MQTTClient('ClientID', '1.2.3.4', port=1883)
client.set_callback(sub_cb)
client.connect()

while True:

    client.wait_msg()  # Blocking wait for message

    # client.check_msg()  # Non-blocking wait for message
    # # Then need to sleep to avoid 100% CPU usage (in a real
    # # app other useful actions would be performed instead))
    # time.sleep(1)
```

Literaturverzeichnis I

[1] Damien P. George, Paul Sokolovsky und contributors. *MicroPython Documentation*. URL: http://docs.micropython.org/en/latest/index.html.

[2] Dave Hylands. *rshell Documentation*. URL: https://github.com/dhylands/rshell.

[3] Nicholas H. Tollervy. *Programming with MicroPython*. O'Reilly, 2017.