

Python auf dem Microcontroller

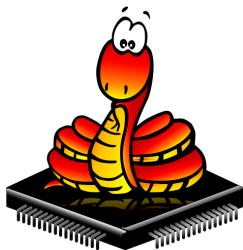
Alexander Böhm
alexander.boehm@malbolge.net

Chemnitzer Linux-Tage, 16. März 2019

Motivation

- Python ist schnell und einfach
- Microcontroller-Entwicklung im IoT
 - mehr Leistung
 - mehr eingebaute Geräte
 - energieeffizienter
 - mehr Möglichkeiten
- Verschmelzung von Desktop- und Microcontroller-Welt

MicroPython



- Python 3-Interpreter
- Subset von Funktionen
- Standardbibliotheken ähnlich CPython
- Inline-Assembler
- verschiedene Erweiterungsmöglichkeiten
- bekannte Forks
 - CircuitPython
 - PyCom

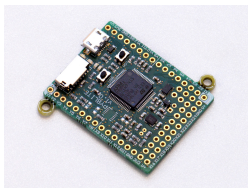
Plattformen

- PyBoard / STM32
- ESP8266
- ESP32
- PIC16
- ARM
- Unix
- ...

Betrachtete Plattformen

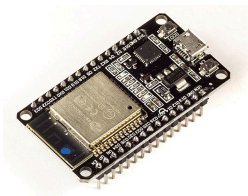
- **PyBoard / STM32**
- ESP8266
- **ESP32**
- PIC16
- ARM
- **Unix**
- ...

PyBoard



- STM32 mit 168 Mhz (Cortex M4)
- 192 KiB RAM
- 1024 KiB Flash
- MicroSD-Card Slot
- 3-Achsen Beschleunigungssensor
- 24 GPIO-PINs, DACs, etc.
- I2C, SPI, RTC, etc.

Espressif ESP32



- Dualcore, Xtensa 32Bit, 240 MHz
- 520 KiB RAM
- 4MB Flash
- Wifi, BLE
- 24 GPIO-PINs, DACs, etc.
- Hall-Sensor
- I2C, SPI, RTC, etc.

Hello PyBoard

Anschluss an den PC

- Prompt über serielle Schnittstelle
 - ähnlich Python-Prompt auf PC
 - verschiedene Eingabemodi
 - Autovervollständigung
- nur PyBoard: Dateisystem als USB-Massenspeicher
 - *boot.py*: Wird direkt nach Start des Bootloaders ausgeführt
 - *main.py*: Hauptprogramm

Module *uos*

- Basisinformation über Board und Version
- Zugriff auf Dateisystem

```
>>> import uos
>>> uos.uname()
(sysname='pyboard',
 nodename='pyboard',
 release='(1).(9).(4)',
 version='b33f108cd-dirty on 2019-03-07',
 machine='PYBv1.1 with STM32F405RG')
```

Module *machine*

- Klassen für gängige Schnittstellen für Zugriff auf Peripherie
- über Boards einheitlich

```
>>> # Nutzung I2C
>>> from machine import I2C, Pin
>>> i2c = I2C(freq=400000, scl=Pin(22), sda=Pin(21))
>>> i2c.scan()
[...]
```

- weitere Module erweitern Anschlussfähigkeiten
- Board-spezifisch

module *network* - WLAN bei ESP32

■ WiFi-Interface aktivieren

```
import network
sta_if = network.WLAN(network.STA_IF)
sta_if.active(True)
```

■ Nach APs scannen

```
sta_if.scan()
[...]
```

■ Mit AP verbinden

```
sta_if.connect('WiFi-SSID', 'WiFi-Pass')
```

Unterschiede zu CPython

Bytes und Bytearrays

- für *bytearrays*
- Slice-Operator unterstützt keine Zuweisung

```
# nur CPython!  
ar = bytearray(4)  
ar[0:2] = [1, 2]
```

Bytes und Bytearrays

- *bytes* hat eine *format*-Methode

```
# nur MicroPython!
```

```
b'My byte data: {:s}'.format('upy')
```

- *bytes* hat keine Keyword-Unterstützung

```
# nur CPython!
```

```
bytes('123', encoding='ASCII')
```

```
# CPython und MicroPython
```

```
bytes('123', 'ASCII')
```

Exception-Handling

- Basis-*Exception*-Klassen haben keine `__init__`-Methode
- Unterstützung für typisierte Exceptions

```
# nur CPython!
```

```
class MyException(Exception):  
    def __init__(self):  
        Exception.__init__(self, 'My exception')
```

```
raise MyException()
```


Beispiel Unterschiede Exception

```
class MyException(ValueError):
    def __init__(self, msg):
        ValueError.__init__(self)
        self.msg = msg

    def __str__(self):
        return 'Your %s' % (self.msg)

try:
    raise MyException('Boom!')
except MyException as e:
    # Hier landet CPython 3.7
    print('It\'s my fault:', e)
except ValueError:
    print('Think about your values')
except:
    # Hier landet micropython
    print('Some undefined occured')
```

Inline Assembler

- anwendbar auf Funktionen, auch in Klassen
- Definition über Decorator

```
@micropython.asm_thumb  
def my_asm_func():  
    . . .
```

- Unterstützung für Untermenge von ARM-Thumb-2
 - freier Zugriff auf Register *R0-R7*
 - eingeschränkter Zugriff für *R8-R15*
 - Instruktionen für verschiedene Wortbreiten
 - Logik, bedingt. Sprung, Stack, Floating Point Instructions

Beispiel Inline Assembler

- Funktionsrückgabewerte immer im Register *r0*

```
@micropython.asm_thumb  
def ret_answer():  
    mov(r0, 42)  
    # return 42
```

Funktionsparameter

- Bis zu 3 Funktionsparameter werden unterstützt
- Parameternamen vordefiniert *r0*, *r1*, *r2*
- Name entspricht Register

```
@micropython.asm_thumb
def simple_add(r0, r1):
    nop()
    add(r0, r0, r1)
    nop()
```

Speicherzugriff

- Über *LDR* und *STR* möglich

```
@micropython.asm_thumb
```

```
def inc_2nd_int(r0):
```

```
    ldr(r2, [r0, 4])
```

```
    add(r2, r2, 1)
```

```
    str(r2, [r0, 4])
```

```
ar = array.array('i', [1, 2, 3])
```

```
inc_2nd_elm(uctypes.addressof(ar))
```

```
# list(ar) == [1, 3, 3]
```

Nicht definierte Befehle

- Nur gebräuchliche Befehl in MicroPython enthalten
- Frei definierbare Bytes

```
@micropython.asm_thumb  
def f():  
    data(2, 0x202A)  
    # return 42
```

Werkzeuge und nützliche Erweiterungen

Module *pyboard*

- für CPython
 - Fernsteuerung über Prompt (z.B. über serielle Schnittstelle)
 - Nutzung RAW-REPL-Modus
 - Kein Ausgabe von Ausdrücken
 - Ähnlich Paste-Mode
 - Ausdrücke werden direkt ausgeführt
- nützlich für Wrapper-Funktionen bspw. I2C/SPI-Proxy

adafruit-ampy

- Verfügbar über Pip
- Dateiverwaltung (bspw. für ESP32)

Dateien anzeigen

```
ampy -p /dev/ttyACM0 ls
```

Datei auf Board speichern

```
ampy -p /dev/ttyACM0 put fib.py
```

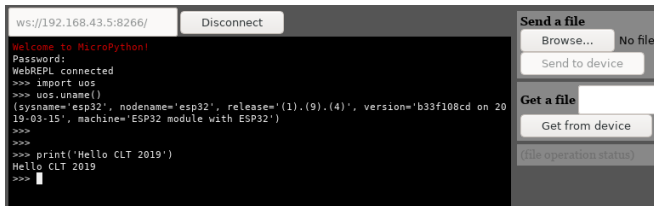
- Ausführen von Skripten auf Boards

Datei ausführen

```
ampy -p /dev/ttyACM0 run fib.py
```

Web-REPL bei ESP32

- Einrichtung über Module *webrepl_setup*
- REPL-Prompt über WebSockets
- Board mit WLAN verbunden
- Client im Browser



The screenshot displays a web browser window with the URL `ws://192.168.43.5:8266/` and a `Disconnect` button. The main terminal area shows the following text:

```
Welcome to MicroPython!  
Password:  
WebREPL connected  
>>> import uos  
>>> uos.uname()  
(sysname='esp32', nodename='esp32', release='(1).(9).(4)', version='b33f108cd on 20  
19-03-15', machine='ESP32 module with ESP32')  
>>>  
>>> print('Hello CLT 2019')  
Hello CLT 2019  
>>> █
```

The sidebar on the right contains the following controls:

- Send a file** section: `Browse...` button, `No file` text, and `Send to device` button.
- Get a file** section: `Get from device` button.
- (file operation status)** section: A greyed-out area for status updates.

upip

- Paketmanagement
- ähnlich zu Pip, Pakete bei PyPI
- Zz. 75 Paket verfügbar¹

```
# Installiert serial module in aktuelles Verzeichnis  
micropython -m upip \  
    install -p . micropython-serial
```

¹Stand 09.03.2019

my-cross

- Cross-Compiler
- Erzeugung optimierten Bytecode (mpy-Files)
- mpy-Files können in Flash geladen werden (Frozen Bytecode)

```
mpy-cross optimize.py
```

Optimierungen und Performance

Vergleich natives C

- Plattform Linux, x86-64
- Betrachtung nur Berechnung
- I/O außerhalb der Betrachtung
- Berechnung Quersumme über Byte-Array (16k)
- 10.000 Wiederholungen

Implementierung	Dauer	Faktor
Python 2	14,1s	45
Python 3	15,7s	50
Cython	8,0s	26
MicroPython	10,1s	32
MicroPython (opt.)	3,8s	12
MicroPython (C-Implementierung)	1,2s	4
C	0,3s	1

Laufzeitvergleich, Optimierung für MicroPython nicht Python 3-kompatibel

Vergleich Arduino

- Ansteuerung OLED-Display via I2C
- Bildschirm wird mit Zeichen vollgeschrieben
- Video vlnr.
 - Arduino ESP32
 - MicroPython ESP32
 - MicroPython PyBoard

Implementierung	Durchläufe/s	Faktor
Arduino (ESP32)	32,1	1,0
MicroPython (ESP32)	5,4	5,9
MicroPython (PyBoard v1.1)	20,9	1,5

Emitter

- native Instruktionen, statt VM-Bytecode

- Native

- Keine Generators
- Kein Context mit *with*
- ca. doppelte so schnell

```
@micropython.native  
def foo(arg):  
    # ...
```

- Viper

- Typisierung
- Zahlreiche Beschränkungen
- Pointer-Unterstützung
- Beschleunigung v.a. für Integer/Bit-Operationen

```
@micropython.viper  
def foo(arg: int) -> int  
    # ...
```


Frozen Module

- Flash als Ablage für Module (z.B. via USB)

- Was passiert beim Laden eines Modules?

- Module Code wird aus Flash gelesen
- Compiler übersetzt Code in Bytecode
- Python-VM führt Bytecode aus

→ Problem

- Compiler benötigt temporären Speicher und Laufzeit
- Arbeitsspeicher für VM-Bytecode des Modules

Frozen Bytecode

- Python-Files vor Ausführungszeit bekannt
 - wenn Firmware erzeugt wird
 - zur Laufzeit des Boards
 - Compiler erzeugt Bytecode für Plattform
 - Bytecode wird in Flash geschrieben
 - Module wird direkt im Flash ausgeführt
- Vorteil: Nur noch nicht dynamische Teile des Moduls müssen im RAM gehalten werden

Erweiterung mit nativen C-Implementierungen

- Module können in C geschrieben werden
- Zugriff wie bei regulären Modulen
- Instruktionen native auf Board ausgeführt

Unix Binary schnell selbst gebaut

```
$ git clone https://github.com/micropython/micropython
$ cd micropython
$ git submodule update --init
$ cd ports/unix
$ make
...
$ ./micropython
MicroPython b33f108cd-dirty on 2019-03-09; linux version
Use Ctrl-D to exit, Ctrl-E for paste mode
>>>
```

PyBoard flashen

- Eigene Module in Frozen Module verzeichnis
\$ `cp mymodule.py ports/stm32/modules`
- Firmware kompilieren
\$ `cd ports/stm32`
\$ `make`
- Board in Update-Modus setzen
(*BOOT0* mit 3.3V, Reset)
- Firmware flashen
\$ `make deploy`

Zusammenfassung & Fazit

- MicroPython bringt Python 3 auf Microcontroller
- Vorteile
 - REPL-Prompt
 - kein ständiges flashen
 - Verkürzung Entwicklungszyklen
 - diverse Optimierungs- und Erweiterungsmöglichkeiten
- Nachteile
 - Performance schlechter
 - weniger Hardware unterstützt
 - kleinere Community
- Folien, Quelltexte, Nachfragen auf
 - github.com/aboehm/CLT2019
 - alexander.boehm@malbolge.net

Fingerprint: C2AA 3A42 66D1 11B2 7C37 74EB 2438 B8AD FDF4 5447

Nützliche Quellen

- MicroPython Website
micropython.org
- MicroPython Documentation
docs.micropython.org/en/latest/index.html
- MicroPython Core Libraries
github.com/micropython/micropython-lib
- Adafruit CircuitPython
github.com/adafruit/circuitpython
- Inline Assembler-Wiki für STM32-basierte PyBoards
wiki.micropython.org/platforms/boards/pyboard/assembler
- WebREPL-Client
github.com/micropython/webrepl

Kontakt für Nachfragen

Alexander Böhm

alexander.boehm@malbolge.net

Fingerprint: C2AA 3A42 66D1 11B2 7C37 74EB 2438 B8AD FDF4 5447

[Github @aboehm](#)