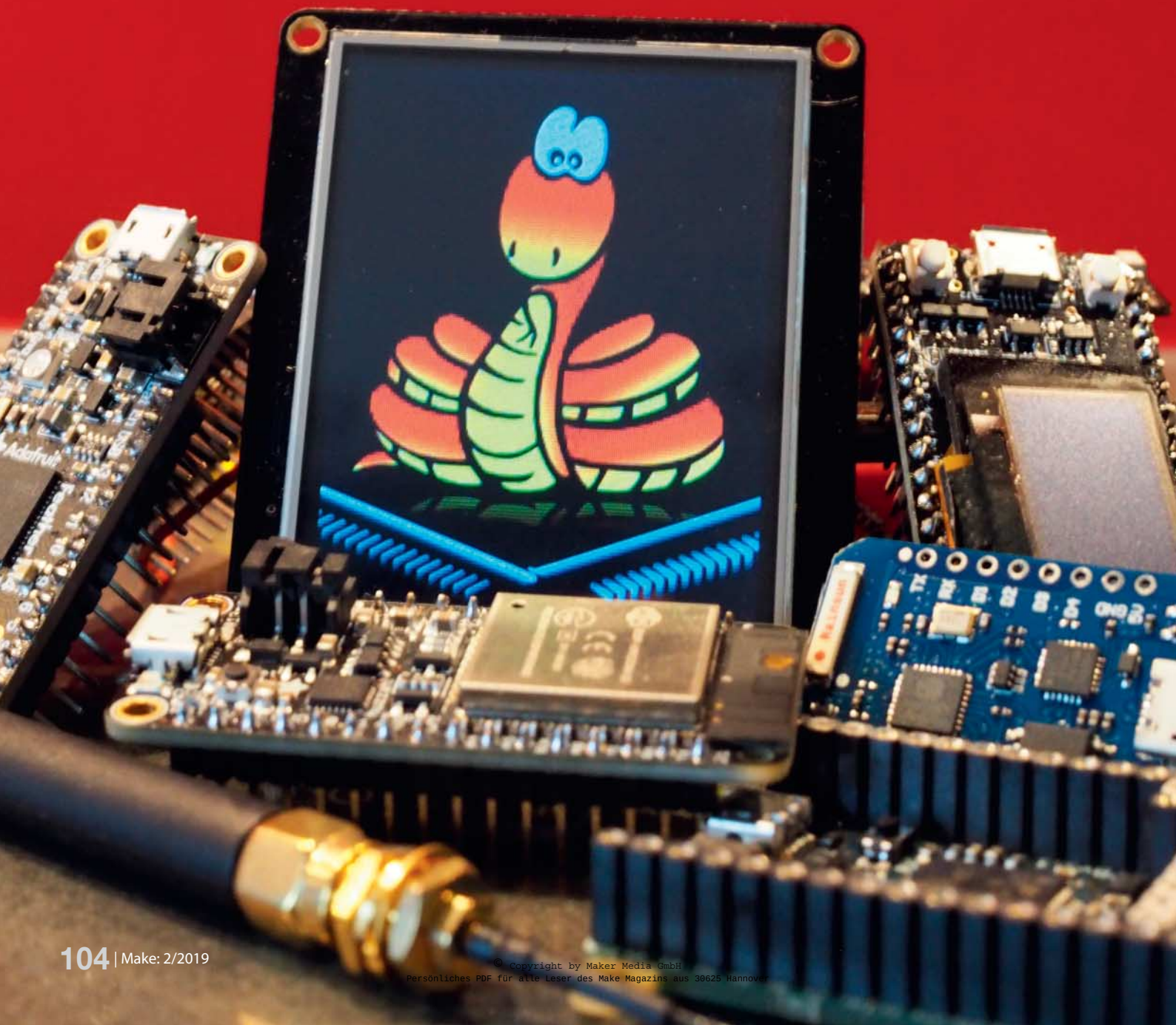
Einstieg in MicroPython

Keine Lust mehr auf C++? Dann probieren Sie doch mal MicroPython. Dank leistungsfähigerer Mikrocontroller mit vergleichsweise viel Speicher bietet sich diese Skriptsprache für viele Projekte als mächtige und einfach zu erlernende Alternative zum Arduino und C++ an.

von Thomas Euler



Mehr Funktionalität für komplexere Projekte und dabei eine einfach zu lesende Programmiersprache – neue Mikrocontroller, die auf MicroPython setzen, machen nach 15 Jahren dem bisherigen Platzhirsch Arduino klare Konkurrenz. Wir erklären die notwendigen Grundlagen und Besonderheiten der Python-3-Variante für Mikrocontroller, geben eine Übersicht über aktuelle MicroPython-Distributionen und geeignete Boards. Am Beispiel des ESP32-Mikrocontrollers zeigen wir, welche Programmierumgebung sich für den Einstieg eignet und demonstrieren, wie man die Hardware anspricht. In einem Artikel im nächsten Heft bauen wir außerdem einen MicroPython-gesteuerten Laufroboter. Links zu allen genannten Webseiten und Programmen finden Sie unter der URL in der Kurzinfo.

Mit einer Crowdfunding-Kampagne stellte der Ingenieur Damien George Ende 2013 MicroPython und das dazugehörige pyboard vor. Inzwischen arbeitet George Vollzeit am MicroPython-Ökosystem, und es gibt Portierungen von MicroPython für eine Reihe von Mikrocontrollern, darunter sowohl die beliebten WLAN-Boards ESP8266 und ESP32 von Espressif als auch ARM-Cortex-M4-Boards. Erste Anlaufstelle ist die Webseite micropython.org, auf der sich neben der kompletten Dokumentation die aktuellen Firmwares für die verschiedenen Mikrocontrollerarchitekturen beziehungsweise -boards finden. Auch einen Emulator gibt es, um MicroPython ohne Hardware auszuprobieren. Der Quellcode ist Open Source und steht komplett auf GitHub zur Verfügung.

MicroPython enthält eine sehr kompakte Implementierung des Python-Interpreters; dieser begnügt sich bereits mit 256kB Flash-Speicher und läuft ab 16kB RAM. Trotzdem ist er im Kern auf maximale Kompatibilität zum Standard-Python (CPython) ausgelegt. Syntax und Sprachumfang entsprechen weitgehend dem von Python 3.4, so dass sich Python-Programmierer sofort zurechtfinden. Zusätzlich sind ein paar Sprachelemente jenseits von CPython definiert, die den Speicherbedarf gering halten und die Ausführungsgeschwindigkeit steigern. Eingebaut ist auch ein komfortabler Kommandozeileninterpreter (REPL für „read-eval-print-loop"), der eingegebenen Code direkt ausführt, ohne dass er gespeichert und auf das Board geladen werden muss. Mit diesem kann man zum Beispiel die Ansteuerung von neuer Hardware testen oder einfach nur MicroPython erkunden.

Neben dem Sprachkern gibt es spezielle Python-Bibliotheken wie `machine` und `micropython` für die Ansteuerung der Hardware. So lassen sich – wie bei der Arduino-Plattform – digitale und analoge Signale ausgeben und/oder lesen, mit externen Komponenten über Busse wie SPI oder I²C kommunizieren und zusätzliche Hardware wie WLAN-Chips oder TFT-Bildschirme steuern. Welche hardwarenahen Funktionen zur Verfügung stehen oder nachgerüstet werden können, hängt vom Board ab. Dieses sollte man daher zum Beginn des Projekts mit Bedacht wählen. Ist etwa WLAN-Unterstützung zwingend, bietet sich ein ESP32-basiertes Board mit eingebautem WLAN an. Wenn nur wenige Anschlusspins benötigt werden und der Speicherbedarf gering ist, reicht viel-

Kurzinfo

» MicroPython in Python programmieren
» Passenden Mikrocontroller finden und einsetzen
» Beispiel: Servo und Distanzsensor mit ESP32 ansteuern

Checkliste

Zeitaufwand:
ca. eine Stunde (je nach Board)

Kosten:
ab 11 Euro (je nach Board)

Programmieren:
Grundkenntnisse in Python

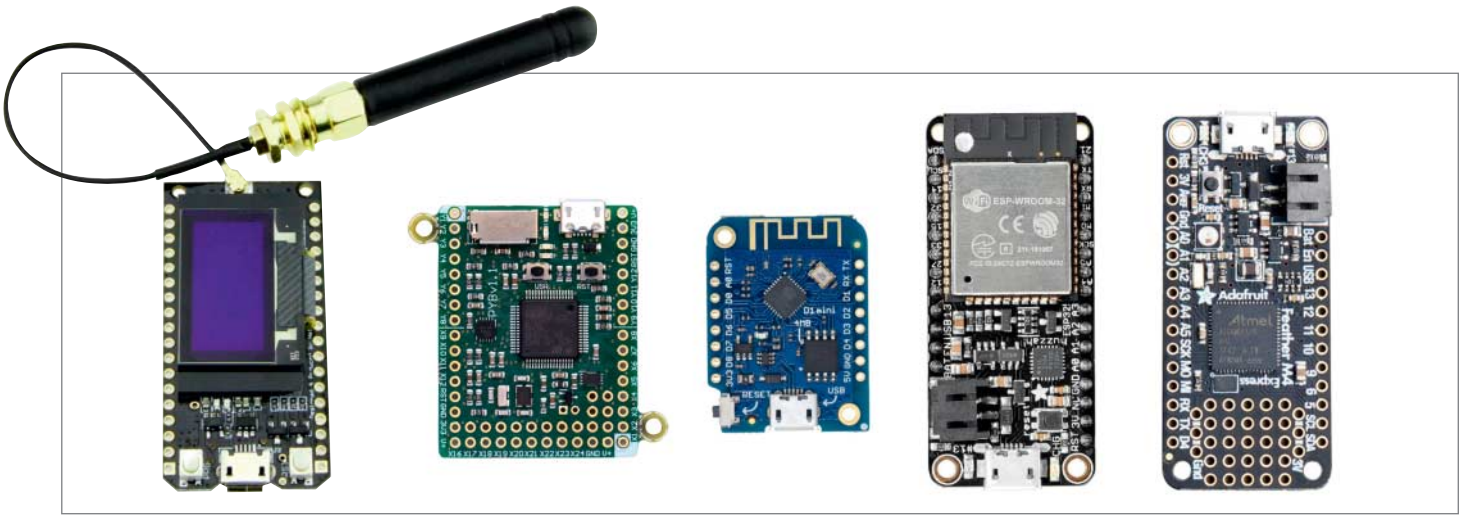
Alles zum Artikel im Web unter
make-magazin.de/xfpz

Material

» **MicroPython-fähiger Mikrocontroller** (z. B. pyboard, Adafruit HUZZAH32)
» **Distanzsensor Sharp GP2D120**
» **Kleiner Modellbau-Servo**

Mehr zum Thema

» Daniel Bachfeld, Arduino-Alternativen, Make 5/18, S. 28
» Florian Schäffer, Servos für jeden Zweck, Make 2/18, S. 46



MicroPython-Boards, von links: TTGO LORA32 mit WLAN/Bluetooth/LoRa und kleinem OLED-Display, pyboard, WeMos D1 mini Pro mit WLAN/Bluetooth, Feather HUZZAH32 mit WLAN/Bluetooth und Feather M4 Express mit kleinem Experimentierfeld

leicht ein günstigeres ESP8266-Board. Für rechenintensivere Programme eignet sich eines der Boards auf Cortex-M4-Basis, die eine bessere Unterstützung zum Rechnen mit Gleitkommazahlen bieten. Wird Wert auf die vollständige Implementierung von MicroPython gelegt, ist das offizielle pyboard die bessere Wahl. Es gibt auch alternative Distributionen mit einer eigenen Reihe von Boards, wie die CircuitPython-Plattform.

CircuitPython von Adafruit

Die US-amerikanische Elektronikfirma Adafruit pflegt mit CircuitPython einen MicroPython-Zweig („fork") für ihre eigenen Boards, die den Einstieg in die Mikrocontroller-Programmierung zum Beispiel in der Schule einfacher machen sollen. Dieser Anspruch spiegelt sich in der hervorragenden Online-Dokumentation mit vielen praktischen Beispielen wieder, hat aber auch zur Folge, dass die Bibliotheken anders gestaltet und oft inkompatibel zu anderen Distributionen sind. Während etwa in MicroPython `machine` spezifische Funktionen für den Hardwarezugriff auf bestimmte Boards enthält, sind bei CircuitPython die entsprechenden Klassen auf mehrere Bibliotheken verteilt: Die Pins sind etwa in `board` definiert, während `digitalio` und `analogio` für digitale und analoge Ein- und Ausgaben verantwortlich sind. Innerhalb des Adafruit-Universums kann man so Programme auf verschiedenen Boards laufen lassen und muss keine oder nur wenige Anpassungen vornehmen. Das ist zum Beispiel hilfreich, wenn man mitten in einem Projekt auf leistungsfähigere Hardware umsteigen will.

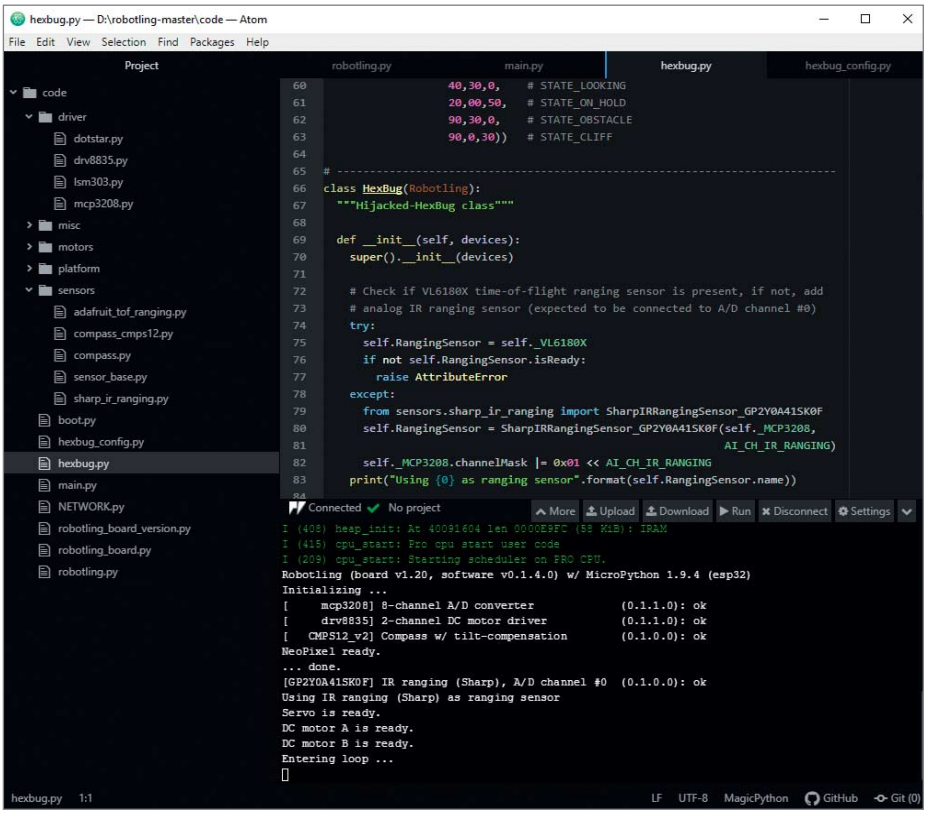
Adafruit hat außerdem eine einheitliche Softwarearchitektur für Hardware-Erweiterungen geschaffen, die das Erstellen von Treibern vereinfacht. Die Bibliotheken sind über Github herunterladbar und auch eine gute Anlaufstelle, wenn man MicroPython um einen Treiber ergänzen will. Wie MicroPython ist CircuitPython Open Source. Zum Programmieren empfiehlt Adafruit den Editor Mu, der über vorprogrammierte Modi für die CircuitPython-Boards verfügt und sich ähnlich wie die Arduino-IDE bedienen lässt. Zur CircuitPython-Installation gibt es einen einfachen Drag-and-Drop-Mechanismus, bei dem die Firmware einfach auf das „Bootlaufwerk" des Boards gezogen wird. Mu gibt es für Windows, macOS und Linux und wird in zahlreichen Tutorials vorgestellt.

Arbeitsumgebung einrichten

Auf allen Boards muss vor dem ersten Einsatz von MicroPython jeweils eine passende Firmware installiert werden. Anschließend können die eigenen Skripte aufgespielt werden (siehe Kasten S. 107) Einige Boards, wie das pyboard und viele Adafruit-Boards, melden dabei einen Teil ihres Flash-Speichers als Laufwerk an, sobald man sie über ein USB-Kabel mit dem Computer verbindet. Auf dieses Laufwerk kopiert man die eigenen Pythonskripte als Dateien mit der Endung `.py`. Dabei muss sich das Hauptprogramm in einer Datei namens `main.py` befinden, denn diese wird von

Übersicht MicroPython-Boards

Name	pyboard v1.1	Feather HUZZAH32	Feather M4 Express	WeMos D1 mini Pro	WiPy 3.0
Hersteller	MicroPython	Adafruit	Adafruit	WeMos	pycom
Mikrocontroller	STM32F405RG (Cortex M4) 32 Bit	Tensilica LX6 (ESP32, WROOM32-Modul) 32 Bit	ATSAMD51J19 (Cortex M4) 32 Bit	Tensilica L106 (ESP8266EX) 32 Bit	Tensilica LX6 (ESP32, WROOM32-Modul) 32 Bit
Anzahl Kerne	1	2	1	1	2
Taktfrequenz	168MHz	240MHz	120MHz	80-160MHz	240MHz
Fließkomma-Hardware	ja	ja	ja	ja	ja
Betriebsspannung	3,3V	3,3V	3,3V	3,3V	3,3V
RAM	192kB	520kB	192kB	64 + 96kB	520kB + 4MB
Flash internal / external	1MB / –	– / 4MB (SPI)	512kB / 2MB (SPI)	– / 16MB	– / 8MB
als USB-Laufwerk	ja	nein	ja	nein	nein
GPIOs /mit PWM	30 / 20	21 / 21	21 / 16	9 / 8	24 / 18
Hardware UART / SPI / I^2C	5 / 2 / 2	≥1/ 2 / 2	1 / ≥1 / ≥1	1 / 2 / 1	2 / 2 / 2
A/D-Wandler	3 × 12 Bit (16 Pins)	2 × 12 Bit (12 Pins)	2 × 12 Bit (6 Pins)	1 × 10 Bit (1 Pin)	1 × 12 Bit (8 Pins)
D/A-Wandler	2 × 12 Bit (2 Pins)	2 × 8 Bit (2 Pins)	2 × 12 Bit (2 Pins)	–	2 × 8 Bit (2 Pins)
Onboard-LEDs	4	1 + 1 (Ladezustand)	1 + 1 (Ladezustand) + 1 Neopixel	1	1RGB
WLAN / Bluetooth	nein / nein	ja (802.11b/g/n) / ja	nein / nein	ja (802.11b/g/n/e/i) / nein	ja (802.11b/g/n) / ja
Besonderheiten	Echtzeituhr, Mikro-SD-Kartenslot, Beschleunigungssensor, 3,3V-Spannungswandler, Batterieanschluss	Echtzeituhr, SD-Kartenunterstützung, Hall-Sensor, 2 × I^2S Audio, Batterieanschluss, LiPo-Ladeschaltung, PCB-Antenne	Echtzeituhr, 2 × I^2S Audio, Batterieanschluss, LiPo-Ladeschaltung, Kamerainterface	Echtzeituhr, 1 × I^2S Audio, Batterieanschluss, LiPo-Ladeschaltung, PCB-Antenne, Antennenanschluss	Echtzeituhr, SD-Kartenunterstützung, Hall-Sensor, 2 × I^2S Audio, PCB-Antenne, Antennenanschluss
Platinengröße [mm] ohne Header	33 × 43 × 4	51 × 23 × 8	51 × 23 × 8	35 × 25 × 8	42 × 20 × 4
Firmware-Distribution (Portierung)	MicroPython (pyboard)	MicroPython (ESP32)	CircuitPython	MicroPython (ESP8266)	MicroPython (via pycom-Software)
Link	store.micropython.org	www.adafruit.com	www.adafruit.com	www.wemos.cc	pycom.io
Preis (ca.)	34 €	24 €	27 €	11 €	20 €
Vertrieb (Beispiel)	bei elektor	bei Eckstein	bei Eckstein	bei Eckstein	pycom.io



Screenshot von Atom mit pymakr eingerichtet und einem ESP32 verbunden; links werden die Dateien des aktuellen Projekts angezeigt, rechts oben befindet sich der Editor und darunter der pymakr-Bereich mit dem REPL-Fenster.

MicroPython automatisch beim nächsten Reset ausgeführt – das kann ein Hardware-Reset oder „Soft-Reset“ (Ctrl-D) über den REPL-Direktinterpreter sein. Ein laufendes Programm unterbricht man mit Ctrl-C. Noch vor `main.py` und der USB-Konfiguration wird übrigens `boot.py` ausgeführt. Hier kann man beeinflussen, über welche Verbindung man mit der REPL kommunizieren möchte: über USB oder – falls unterstützt – über WLAN (WebREPL).

Andere Boards, wie der hier verwendete ESP32-Controller, melden zwar kein Flash-Laufwerk an, aber auch dort lädt man die `.py`-Skripte über USB hoch. Dafür gibt es Kommandozeilenprogramme wie ampy von Adafruit. Komfortabler bewerkstelligt dies pymakr von pycom, einem weiteren Hersteller von MicroPython-Boards. Pymakr ist ein Plug-in für den Open-Source-Editor Atom (siehe unsere Übersicht auf Seite 66) und ergänzt diesen um die REPL und die Fähigkeit, ganze Programverzeichnisse hochzuladen oder einzelne Skripte auf dem Board zu starten. Atom wird damit zu einer bequemen und mächtigen Entwicklungsumgebung für MicroPython. Der einzige Nachteil ist, dass Atom derzeit nicht auf Deutsch verfügbar ist. Für die Installation von pymakr startet man Atom, ruft den Eintrag *Settings* im *File*-Menü auf, geht auf die *Install*-Seite, gibt dort pymakr in die Suchmaske ein und wählt das entsprechende Paket aus der Liste zur Installation aus. Leider benimmt sich pymakr bei Updates gelegentlich störrisch. Hier hilft es, die Erweiterung zu deinstallieren, Atom neu zu starten und die aktuelle Version frisch zu installieren. Wenn die Installation erfolgreich war, erscheint unterhalb des Editorfensters ein Bereich, über den man mit dem REPL des Boards interagieren kann (siehe Screenshot). Hat man MicroPython auf dem HUZZAH32-Board installiert (siehe Kasten S. 107), schließt man dieses über ein USB-Kabel an den PC an. Ist ein Akku angeschlossen, zeigt die gelbe LED auf dem HUZZAH32-Board den Ladezustand an. Dauerhaftes Leuchten bedeutet, dass der Akku geladen wird; flackert die LED, ist der Akku voll oder es ist kein Akku angeschlossen.

ESP32 unter Windows 10 einrichten

Für das HUZZAH32-Feather-Board bietet Adafruit noch keine CircuitPython-Firmware, weshalb wir MicroPython verwenden. Alle nötigen Webseiten finden Sie in der Kurzinfo. Zunächst laden wir die aktuelle Python-3-Distribution und den Programmier-Editor Atom herunter und installieren sie. Wer bereits Python 3 oder einen anderen Editor nutzt, kann auf diesen Schritt verzichten. Für den ESP32 brauchen wir außerdem den CP2104-USB-Treiber von Silicon Labs und starten nach der Installation den PC neu. Nun werden von der Kommandozeile des Betriebssystems aus einige Python-Pakete ergänzt (Zeile 1 unten).

Schließlich benötigen wir die aktuelle MicroPython-Firmware für den ESP32. Dann kann man das Board über ein USB-Kabel mit dem PC verbinden und den Namen der seriellen Schnittstelle des Boards (z. B. COM10) beispielsweise im Windows-Gerätemanager ermitteln. Nun wird der Flash-Speicher des ESP32 komplett gelöscht und dann die Firmware übertragen (Zeile 2 und 3). Die Schnittstelle und der Name der Firmware-Datei müssen angepasst werden. Wenn die Firmware nicht im aktuellen Verzeichnis liegt, muss man den entsprechenden Dateipfad ergänzen. Um zu überprüfen, ob sie korrekt geladen wurde und MicroPython läuft, stellt man über ein Terminalprogramm (z. B. PuTTY) eine serielle Verbindung zum Board her (115200 Baud). Erscheint die MicroPython-Eingabeaufforderung >>> kann man über REPL mit Python auf dem ESP32 interagieren.

```
1 pip install msgpack, esptool, adafruit-ampy
2 python -m esptool --port COM10 erase_flash
3 python -m esptool --port COM10 --baud 460800 write_flash --flash_mode dio --flash_size=detect 0x1000
  esp32-20180511-v1.9.4.bin
```

Die bei der Installation der Firmware ermittelte serielle Schnittstelle muss in der Konfigurationsseite von pymakr unter *Device Address* eingetragen werden, etwa COM10. Die übrigen Einstellungen kann man zunächst so lassen. Ein Klick auf den pymakr-Bereich stellt die Verbindung zur REPL auf dem Board her und das Erscheinen der Eingabeaufforderung >>> zeigt den Erfolg an. Bekommt man keine Verbindung, hilft manchmal ein Klick auf „Disconnect“ oder das Lösen und neuerliche Anstecken des USB-Kabels. Wenn auf dem Board bereits ein Programm läuft, klappt zwar die Verbindung, aber es erscheint keine Eingabeaufforderung. In diesem Fall drückt man Ctrl-C (notfalls mehrmals), um das laufende Programm zu unterbrechen.

Um `.py`-Skripte laufen zu lassen, öffnet man sie im Editor und klickt oberhalb des REPL-Bereichs auf *Run*. Ganze Verzeichnisse kann man mit *Upload* in den Flash-Speicher des Boards hochladen; das Quellverzeichnis legt man auf der pymakr-Konfigurationsseite fest (*Sync Folder*). Hier kann man festlegen, welche Dateitypen hochgeladen werden sollen (*Pyignore list* und *Upload file types*). Die Konfigurationsseite findet man über *File*, *Settings*, *Packages* und die Suchmaske oder bei angeschlossenem Board über den Knopf *Settings* (*Global settings*) über der REPL. Mit dem Knopf *More* kann man sich anzeigen lassen, über welche serielle Schnittstelle das Board erreichbar ist, oder auch die Firmware-Version ausgeben.

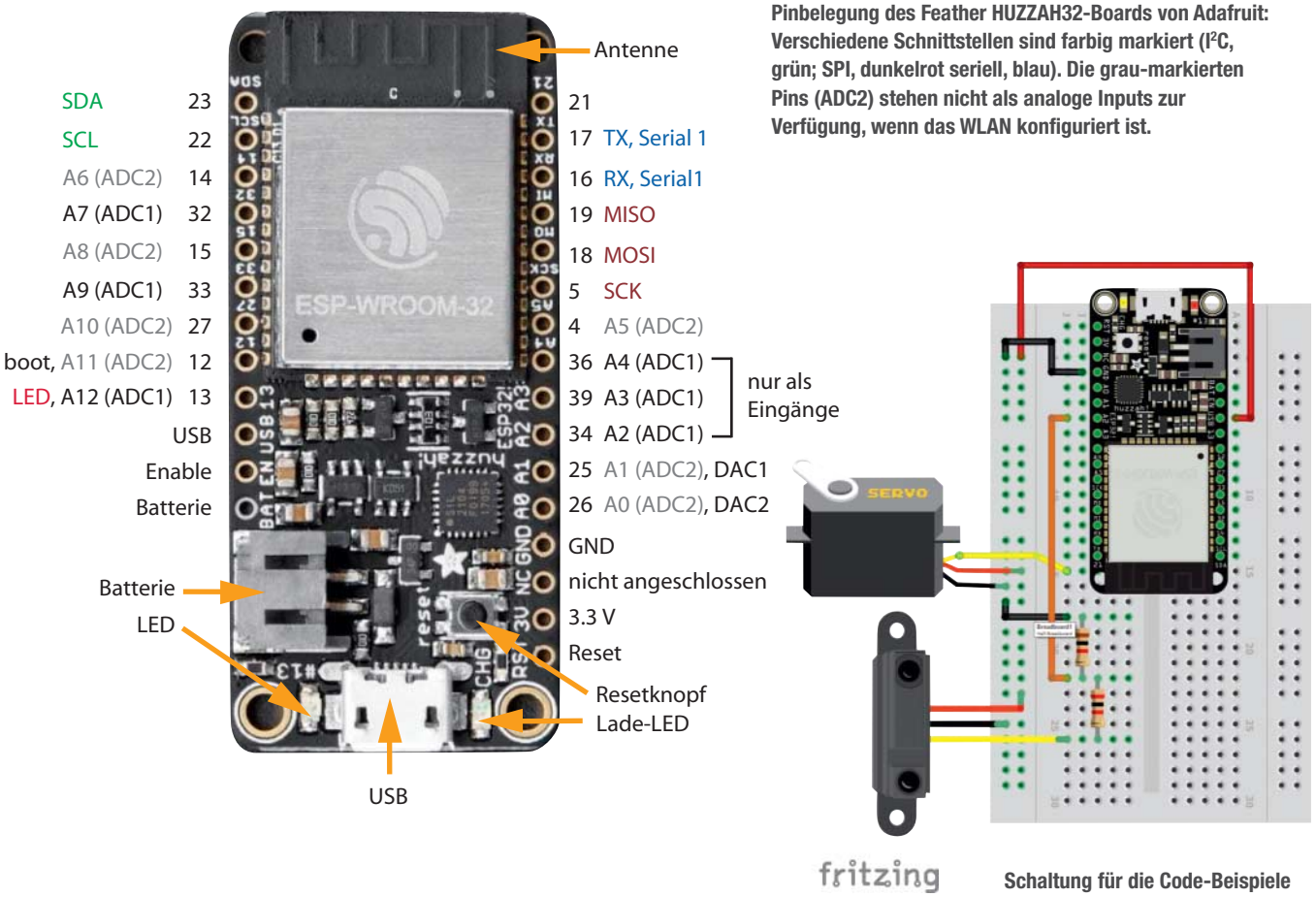
Erste Schritte

ESP32-Boards gibt es in verschiedenen Bauformen, die sich in Bezeichnung und Verfügbarkeit der Pins unterscheiden. Die folgenden Beispiele sind mit dem HUZZAH32 getestet. Für andere Boards müssen die Pin-Nummern anhand der Herstellerangaben überprüft und im Zweifelsfall angepasst werden. Zu beachten ist auch, dass zwar fast alle Pins mehrere Funktionen unterstützen (general purpose input/output, GPIO), aber nicht alle. Manche Pins sind reine Eingänge, andere spielen beim Bootprozess eine Rolle. Außerdem fällt ein Teil der analogen Eingänge weg, sobald die WLAN-Funktionen zum Einsatz kommen. Achtung: Die Pins des ESP32 vertragen maximal 3,6 Volt und können nur sehr wenig Strom liefern. Daher funktionieren die Beispiele nur, wenn das Board am USB-Kabel hängt. Distanzsensor und Modellbauservo ziehen ihren Strom in den Beispielen aus dem USB-5V-Pin. Hier reicht ein einfacher Modellbauservo. Weitere Informationen gibt es in unserer Servo-Übersicht in Make 2/18, Seite 46.

blink.py

```
 1 import machine
 2 import time
 3
 4 pin = machine.Pin(13,
   machine.Pin.OUT)
 5
 6 for i in range(10):
 7   pin.value(not pin.value())
 8   time.sleep(0.5)
 9
10 pin.value(False)
```

Das erste Beispiel `blink.py` demonstriert den Zugriff auf den GPIO-Pin 13. An diesem Pin hängt beim HUZZAH32 die eingebaute rote LED. Zunächst importieren wir die Module `machine` (Zeile 1) und `time` (Zeile 2). Nun wird ein Pin-Objekt `pin` für Pin 13 erzeugt und für die digitale Ausgabe konfiguriert (Zeile 4). In der Schleife (Zeilen 6–8) wird der letzte Zustand des Pins `pin` abgefragt, invertiert und neu gesetzt (Zeile 7), wobei `True` die LED an-



Pinbelegung des Feather HUZZAH32-Boards von Adafruit: Verschiedene Schnittstellen sind farbig markiert (I²C, grün; SPI, dunkelrot seriell, blau). Die grau-markierten Pins (ADC2) stehen nicht als analoge Inputs zur Verfügung, wenn das WLAN konfiguriert ist.

Schaltung für die Code-Beispiele

und `False` sie abschaltet. Nachdem mit der Funktion `sleep` für 0,5 Sekunden gewartet wurde (Zeile 8), beginnt der nächste Durchlauf. Die Schleife läuft 10-mal durch, die LED blinkt also 5-mal. Am Ende wird sie wieder ausgeschaltet (Zeile 10).

Im zweiten Beispiel `distance.py` messen wir den Abstand zu einem Objekt mit Hilfe eines analogen Distanzsensors (GP2D120 von Sharp) und ändern die Helligkeit der roten LED abhängig vom gemessenen Abstand. Im Code werden nur die benötigten Klassen und Funktionen aus den Modulen `machine` und `time` importiert (Zeilen 1, 2). Für Pin 13 wird ein Objekt der Klasse `PWM` generiert (Zeile 4). Diese erzeugt durch Pulsbreitenmodulation (pulse width modulation, PWM) an einem digitalen Pin ein quasi-analoges Signal, in dem die Spannung mit hoher Frequenz an- und ausgeschaltet wird. Die Höhe dieses Signals – bei Pin 13 auch die LED-Helligkeit – hängt davon ab, wie lange die Spannung während eines Zyklus an ist (`duty cycle`) oder Tastverhältnis. Für Pin 34 – auf dem HUZZAH32 als analoger Eingang A2 gekennzeichnet – wird ein Objekt der Klasse ADC (analog-digital converter) erzeugt und so konfiguriert, dass eine am Pin anliegende Spannung mit einer Auflösung von 10 Bit digitalisiert wird (Zeilen 6, 7). In einer Endlosschleife (Zeilen 10–12) misst und digitalisiert `adc.read` die Spannung am Pin 34 und übergibt das Resultat an das PWM-Objekt. Dessen Methode `duty` erwartet einen Wert zwischen 0 und 1023. Das heißt, je näher ein Objekt am Sensor ist, desto heller wird die rote LED. Das `try-except`-Konstrukt (Zeilen 9–14) stellt sicher, dass die von `pwm` belegten Ressourcen freigegeben werden (Zeile 15), wenn der Benutzer die Schleife per Ctrl-C unterbricht. Da der Standardmessbereich der Analog-Eingänge 0 bis 1 Volt beträgt, verwenden wir einen Spannungsteiler aus zwei Widerständen, um die Ausgangsspannung des Distanzsensors (max. rund 3 Volt) anzupassen).

Das letzte Beispiel `servo_test.py` verwendet die Klasse `Servo` aus dem Modul `servo.py`. Letzteres zeigt, wie man eine einfache Klasse definiert, die die Steuerung eines Modellbauservos verkapselt. Die Klassenmethode `__init__`, die automatisch aufgerufen wird, sobald ein neues Objekt dieser Klasse erzeugt wird, benötigt einen PWM-fähigen GPIO-Pin (`pin`) und generiert ein internes PWM-Objekt (`self._pwm`) mit Standardparametern für die Servoansteuerung (Zeilen 4–8). Viele Servos erwarten ein 50Hz-Steuersignal: Der Stellwinkel des Servos wird dabei über die Pulsbreite festgelegt, die üblicherweise zwischen 500 und 2500µs liegen muss. Die Methode `setPosition_us` nimmt eine Servo-Position als µs-Wert (`pos_us`) entgegen und berechnet daraus das Tastverhältnis (`duty`) für das PWM-Objekt (10–16). Dabei wird sichergestellt, dass der `pos_us` innerhalb des erlaubten Bereichs bleibt (zwischen `min_us` und `max_us`). Die Methode `deinit` erlaubt den Servo abzuschalten, wenn er nicht mehr benötigt wird, und so Ressourcen freizugeben (Zeilen 18–20). Nachdem in `servo_test.py` das Servo-Objekt `servo` an Pin 21 erzeugt wurde (Zeile 5), wird die Position im Sekundenabstand dreimal neu gesetzt (Zeilen 7–12). Der Modellbauservo sollte sich entsprechend bewegen.

Performance-Fragen

Während bei Arduino-Boards ein in C++ geschriebenes Programm in Maschinencode kompiliert wird und so quasi als Firmware direkt auf dem Mikrocontroller läuft, gibt es bei MicroPython einen Overhead: Ein `.py`-Skript im Flash-Speicher wird erst in Bytecode übersetzt, um anschließend in einer Laufzeitumgebung ausgeführt zu werden. Dieser Umweg ist wegen der leistungsfähigeren Hardware zunächst kaum spürbar. Wenn es um komplexere und zeitkritische Programme geht, ist es an der Zeit, sich die MicroPython-spezifischen Möglichkeiten der Code-Optimierung anzusehen – diese reichen vom Einsatz typisierter Datenstrukturen fester Größe bis hin zur Zusammenstellung eigener Firmware mit eigenen Modulen. Dabei sollte man nicht generell auf Pythons mächtige Sprachkonstrukte verzichten, sondern ein Programm zuerst zum Laufen bringen, und dann zeitkritische Teile gezielt optimieren. Ausführliche Informationen findet man in der offiziellen Doku. Auch die ESP32-Seite von Boris Lovosevic (siehe Kurzlink) ist ein guter Ausgangspunkt. Dort gibt es vorkompilierte Firmware-Versionen mit unterschiedlicher Speicheraufteilung sowie eine Anleitung für eigene MicroPython-Versionen.

Ausblick

Im nächsten Heft bauen wir einen kleinen autonomen Laufroboter, der MicroPython versteht. Als Basis dient eine ferngesteuerte Spielzeugspinne der Firma HexBug. Diese bekommt eine einfache Platine aufgesetzt, die viele Anschlußmöglichkeiten für Motoren und Sensoren bietet und als „Gehirn“ einen Adafruit-Feather-Mikrocontroller nutzt. —*hch*

distance.py

```
1  from machine import Pin, PWM, ADC
2  from time import sleep
3
4  pwm = PWM(Pin(13))
5
6  adc = ADC(Pin(34))
7  adc.width(ADC.WIDTH_10BIT)
8
9  try:
10   while True:
11     pwm.duty(adc.read())
12     sleep(0.2)
13
14 except KeyboardInterrupt:
15   pwm.deinit()
```

servo.py

```
1  from machine import PWM
2
3  class Servo(object):
4    def __init__(self, pin, freq_Hz=50, min_us=600, max_us=2400):
5      self._min_us = min_us
6      self._max_us = max_us
7      self._freq_Hz = freq_Hz
8      self._pwm = PWM(pin, freq=freq_Hz, duty=0)
9
10   def setPosition_us(self, pos_us):
11     if pos_us == 0:
12       duty = 0
13     else:
14       pos_us = min(self._max_us, max(self._min_us, pos_us))
15       duty = pos_us *1024 *self._freq_Hz //1000000
16     self._pwm.duty(duty)
17
18   def deinit(self):
19     self._pwm.duty(0)
20     self._pwm.deinit()
```

servo_test.py

```
1  from machine import Pin
2  from time import sleep
3  from servo import Servo
4
5  servo = Servo(Pin(21))
6
7  servo.setPosition_us(1300)
8  sleep(1)
9  servo.setPosition_us(2100)
10 sleep(1)
11 servo.setPosition_us(1800)
12 sleep(1)
13
14 servo.deinit()
```