ESP32: Messen und steuern mit MicroPython

Das ESP-Board als „Embedded System“

Mit dem ESP32 hat Espressif ein in der Community beliebtes SoC (System-on-Chip) geschaffen, das neben dem ESP8266 von immer mehr Anwendern für Elektronikprojekte genutzt wird. Nachdem im letzten Beitrag (ELVjournal 6/2019 [1]) erläutert wurde, wie der ESP32 mit MicroPython (µP) programmiert werden kann, soll im Folgenden beschrieben werden, wie man Sensoren auswerten, Messwerte erfassen, ein Display ansteuern oder einen Servo kontrollieren kann.

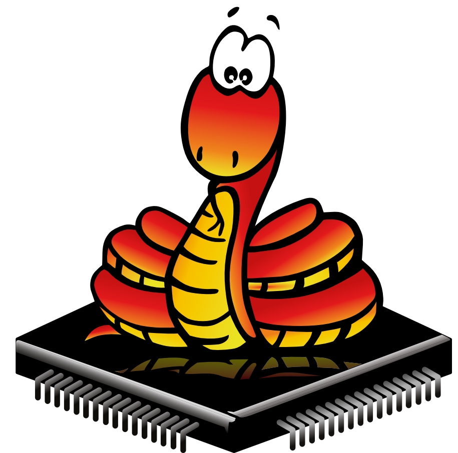
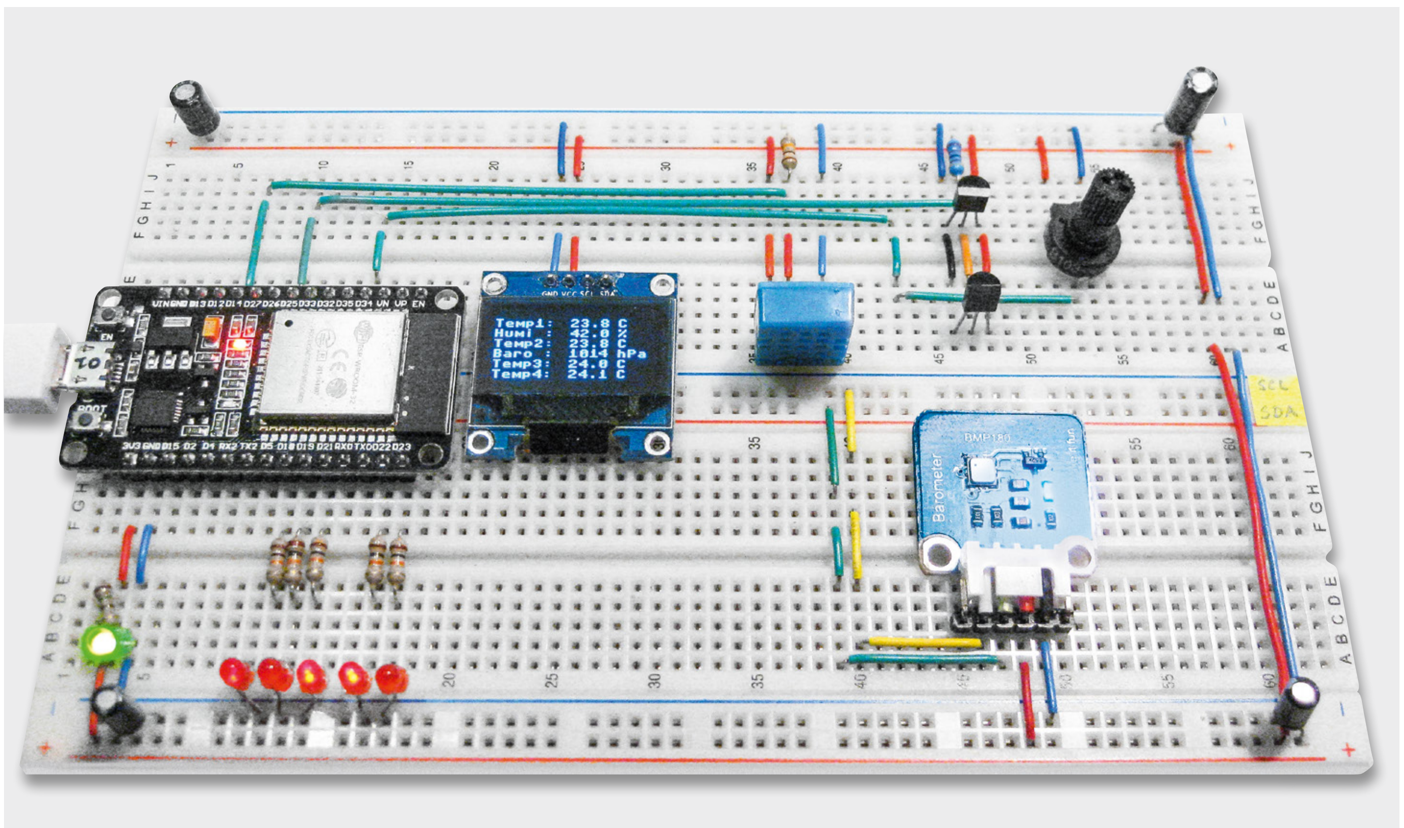


Abbildung: User „Neon22“ auf https://github.com



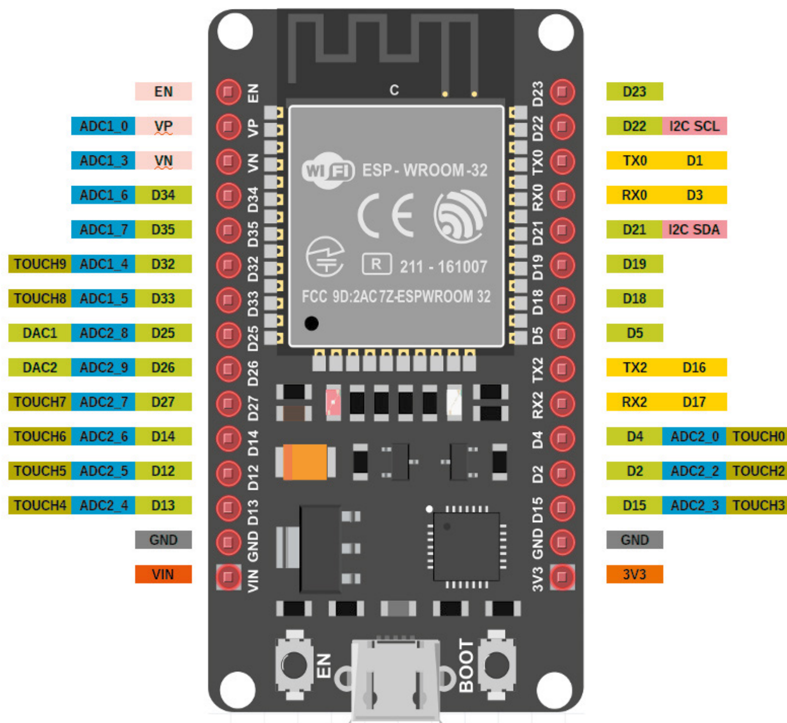


Bild 1: Pinbelegung des Joy-IT NodeMCU-Boards

Tabelle 1

GPIO	Input	Output	Bemerkung
0	pulled up	OK	outputs PWM signal at boot
1	TX pin	OK	debug output at boot
2	OK/ADC	OK	connected to on-board LED
3	OK	RX pin	HIGH at boot
4	OK /ADC	OK	
5	OK	OK	outputs PWM signal at boot
6	x	x	connected to integrated SPI flash
7	x	x	connected to integrated SPI flash
8	x	x	connected to integrated SPI flash
9	x	x	connected to integrated SPI flash
10	x	x	connected to integrated SPI flash
11	x	x	connected to integrated SPI flash
12	OK/ADC	OK	boot fail if pulled high
13	OK/ADC	OK	
14	OK/ADC	OK	outputs PWM signal at boot
15	OK/ADC	OK	outputs PWM signal at boot
16	OK	OK	
17	OK	OK	
18	OK	OK	
19	OK	OK	
21	OK	OK	
22	OK	OK	
23	OK	OK	
25	OK/ADC	OK / DAC	
26	OK/ADC	OK / DAC	
27	OK/ADC	OK	
32	OK/ADC	OK	
33	OK/ADC	OK	
34	OK/ADC		input only
35	OK/ADC		input only
36	OK		input only

General Purpose Input Output (GPIO)

Der ESP32-Chip verfügt über 48 Pins, die meist mit mehreren Funktionen belegt sind. Dabei ist allerdings zu beachten, dass nicht alle Pins auf sämtlichen ESP32-Entwicklungsplatinen verfügbar sind. So sind etwa auf dem Joy-IT NodeMCU-Board von den insgesamt vorhandenen 18 ADC-Kanälen nur 13 frei verfügbar. Zwei weitere liegen auf dem VP- bzw. VN-Anschlüssen des Boards und sind mit rauscharmen Vorverstärkern versehen. Zudem sind einige von ihnen nicht universell einsetzbar. Bild 1 zeigt die Belegung der Ein- und Ausgänge des Joy-IT NodeMCU-Boards (siehe auch „Material").

Obwohl verschiedene Entwicklungsplatinen unterschiedliche Pin-Konfigurationen aufweisen, haben die jeweils verfügbaren GPIOs (General Purpose Input Output) stets die identischen Funktionen. Dadurch wird die Portierung von Projekten von einer Board-Version zu einer anderen deutlich erleichtert.

Das Joy-IT-ESP32-Board stellt die folgenden Funktionen zur Verfügung:

- 15 Analog-Digital-Wandler-Kanäle (ADC)
- 2 SPI-Schnittstellen
- 2 UART-Schnittstellen
- 2 I^{2}C-Schnittstellen
- 16 PWM-Ausgangskanäle
- 2 Digital-Analog-Wandler (DAC)
- 2 I^{2}S-Schnittstellen
- 10 kapazitive Berührungssensoren

Die Funktionen ADC (Analog-Digital-Wandler) und DAC (Digital-Analog-Wandler) sind festen Pins zugeordnet. Die Anschlüsse für I^{2}C, SPI, PWM usw. können dagegen weitgehend frei konfiguriert und softwareseitig zugewiesen werden. Diese Funktionalität wird durch die Multiplexfunktion des ESP32-Chips ermöglicht. Sie stellt damit einen großen Fortschritt gegenüber den festen Pinbelegungen klassischer Mikrocontroller dar. Die Beschaltung bzw. das Layout für den ESP32 wird so in vielen Fällen erheblich erleichtert.

Obwohl viele Pin-Funktionen in der Software frei definierbar sind, hat sich ein gewisser Standard durchgesetzt (siehe Bild 1). Dadurch wird ein hohes Maß an Kompatibilität in verschiedenen Anwendungsbereichen erreicht. Man sollte sich daher an diese Konfiguration halten, solange keine schwerwiegenden Gründe dagegensprechen. Die wichtigsten Funktionszuordnungen sind in der Tabelle 1 zusammengefasst.

Die grün hervorgehobenen Pins können frei verwendet werden. Die gelb markierten zeigen beim Booten unter Umständen ein unerwartetes Verhalten. Wenn dies berücksichtigt wird, sind sie jedoch problemlos einsetzbar. Die rot markierten Pins sollten dagegen nicht als Ein- oder Ausgänge verwendet werden.

Der ESP32 verfügt über 10 integrierte kapazitive Berührungssensoren. Diese sind ladungsempfindlich und können daher die Berührung mit einem Finger detektieren. Die Pins können einfach in kapazitive Pads integriert werden und ersetzen mechanische Taster. Darüber hinaus sind die kapazitiven Touch-Pins auch verwendbar, um den ESP32 aus dem Tiefschlafmodus zu reaktivieren.

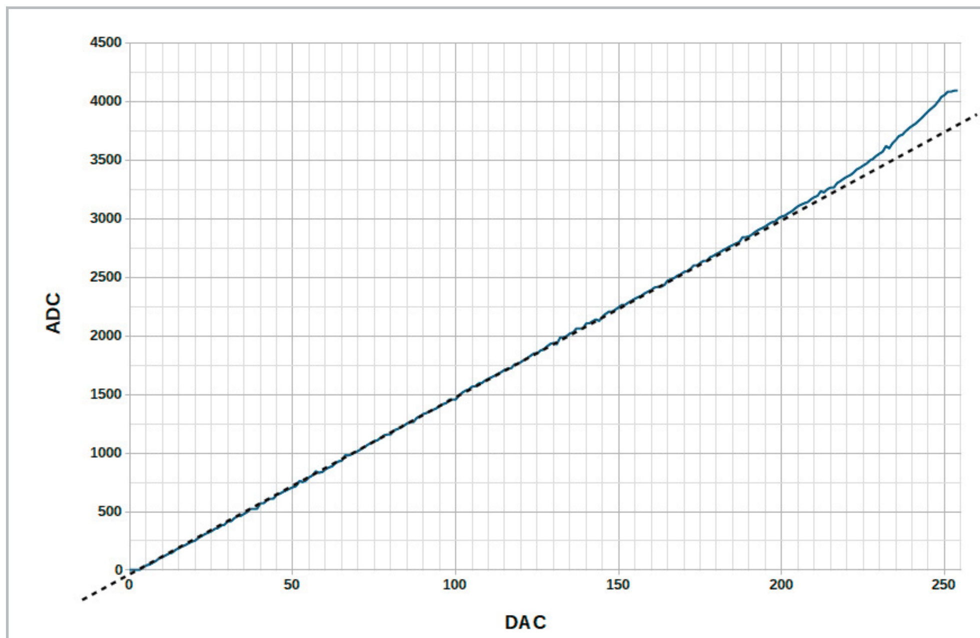


Bild 2: Nichtlinearität des integrierten ADCs

Die analoge Welt per ADC erfassen

Der ESP stellt bis zu 18 ADC-Eingänge zur Verfügung. Damit lassen sich analoge Spannungswerte mit hoher Auflösung erfassen. Idealerweise würden man bei Verwendung der ESP32-ADC-Pins ein lineares Verhalten erwarten. Dies ist jedoch nicht der Fall. Vielmehr zeigen sie das in Bild 2 gezeigte nichtlineare Verhalten. Dies hat zur Folge, dass insbesondere an den Randbereichen größere Messfehler auftreten. Diese können jedoch durch geeignete Maßnahmen wie die Linearisierung mit einem Ausgleichspolynom oder durch geeignete Einschränkung des Messbereichs deutlich reduziert werden.

Die ADCs haben standardmäßig eine Auflösung von 12 Bit. Der Wertebereich liegt damit für Spannungen von 0 und 3,3 V zwischen 0 und 2^{12}-1 = 4095. Die Auflösung kann softwareseitig geändert werden. Im Bedarfsfall sind auch 9 bis 11 Bit einstellbar.

Nichtlinearitäten

Die ausgeprägte Nichtlinearität des ADCs kann mithilfe des (linearen) DACs leicht grafisch dargestellt werden.

Das folgende Programm (alle Skripte zum Download gibt es unter [2]) liefert die entsprechenden Messdaten:

```
# ADC_DAC_tst.py

from machine import DAC, ADC, Pin
import time

dac0=DAC(Pin(25))
adc0=ADC(Pin(34))
adc0.atten(ADC.ATTN_11DB)

for n in range(0, 256):
  print(n, end =' ')
  dac0.write(n)
  time.sleep(0.1)
  print(adc0.read())
  time.sleep(0.1)
```

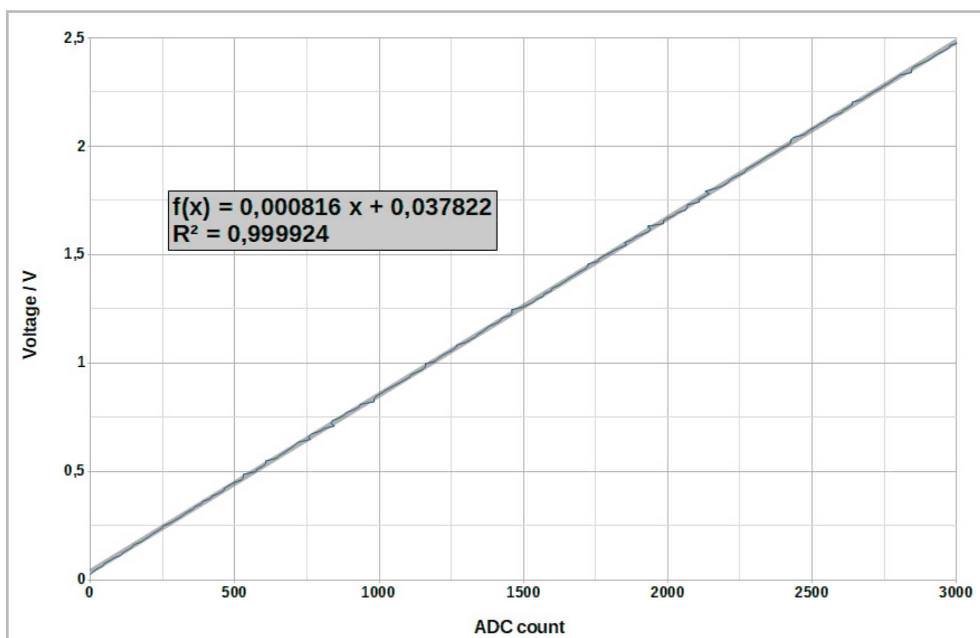


Bild 3: Kalibrationsgerade des ESP32 ADCs

Die damit gewonnenen Werte können mithilfe von Excel oder LibreOffice in einem Diagramm dargestellt werden (siehe Bild 3). Über ein Ausgleichspolynom können die Werte über den gesamten Messbereich hinweg linearisiert werden. Damit lässt sich allerdings keine sehr hohe Gesamtgenauigkeit erreichen. Besser ist es, nur die weitgehend linearen Anteile der Kennlinie zu verwenden. Man erkennt aus Bild 2, dass der ADC im Bereich bis ca. 3000 counts weitestgehend linear arbeitet. Beschränkt man sich auf diesen Bereich, dann ergibt sich die Kalibrationsgerade wie in Bild 3.

Mit der daraus abgeleiteten Regressionsformel:

voltage = 0.000816 * ADC_count + 0.037822

können nun Spannungswerte zwischen 200 mV und 2,5 Volt sehr präzise erfasst werden. Dies ist für viele Sensoranwendungen vollkommen ausreichend, da Werte von unter 0,2 Volt von vielen Sensoren ohnehin meist nicht erreicht werden. Das Messprogramm dazu sieht so aus:

```
# ADC_lin.py

from machine import Pin, ADC
from time import sleep

pot = ADC(Pin(34))
pot.atten(ADC.ATTN_11DB)      #Full range: 3.3v

while True:
  pot_value = pot.read()
  voltage = 0.000816*pot_value + 0.037822
  print(voltage)
  sleep(0.1)
```

Über ein Potenziometer können nun Testspannungswerte erzeugt werden. Die erfassten Werte erscheinen in der Ausgabekonsole der verwendeten Programmierumgebung (siehe Bild 4).

Mit einem hochwertigen und exakt kalibrierten Voltmeter können die Werte nachgemessen werden. Die Abweichungen sollten dabei deutlich unterhalb von 3 % bleiben. Damit steht dem Einsatz von analogen Messwandlern wie Photodioden, analogen Temperatursensoren oder Dehnungsmessstreifen nichts mehr im Wege.

Aber auch für den Anschluss digitaler Messwandler ist der ESP32 bestens gerüstet. Über geeignete Bussysteme können diese Sensoren direkt mit dem Prozessor kommunizieren.

Einfach und universell: der I²C-Bus

Eines der wichtigsten Beispiele dazu ist der I²C-Bus (meist als „I-Quadrat-C-Bus" ausgesprochen), ein von Philips 1982 entwickelter serieller Kommunikationsbus. Über dieses System können zwei oder mehr Geräte miteinander kommunizieren. Die an den Bus angeschlossenen Einheiten sind als Master oder als Slaves konfigurierbar. In der Regel hat ein Bus nur einen Master und einen oder mehrere Slaves. Der Standard erlaubt zwar auch komplexere Topologien, diese werden jedoch nur selten eingesetzt. Bild 5 zeigt zwei Komponenten am I²C-Bus des ESP32.

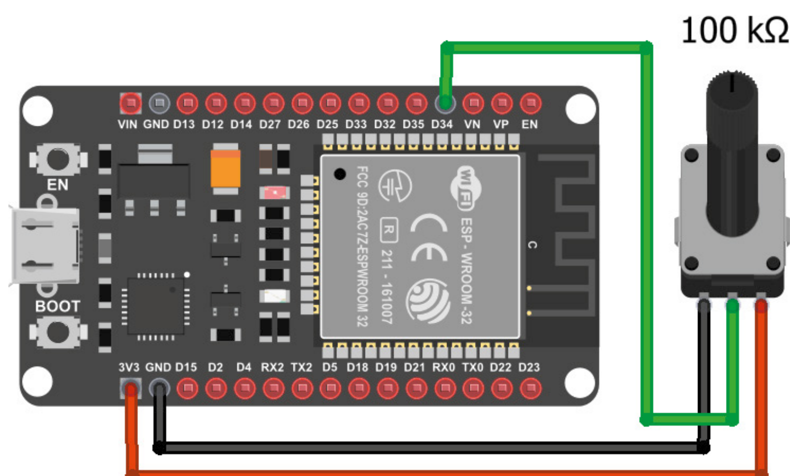


Bild 4: Potenziometer an einem Analogeingang des ESP32

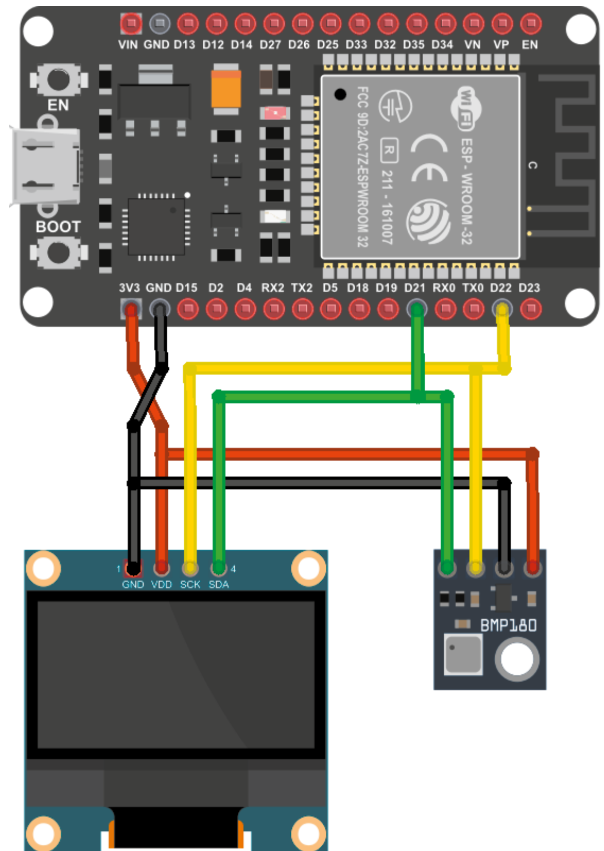


Bild 5: Display und Barometermodul am I²C-Bus des ESP32

Ein zentraler Punkt beim I2C-Bus ist, dass jedes an den Bus angeschlossene Gerät über eine eindeutige Adresse verfügen muss. Für die Übertragung von Daten stehen u. a. folgende gebräuchliche Geschwindigkeiten zur Verfügung:
1. Standard (100 Kbit/s)
2. Fast (400 Kbit/s)

Der I²C-Bus benötigt nur zwei Kommunikationsleitungen, welche die Geräte verbinden:

- SDA, Serial Data – die eigentliche Datenleitung
- SCL, Serial Clock – das vom Master erzeugte Taktsignal

Die beiden Leitungen müssen über Pull-up-Widerstände mit einer Referenzspannung (Vdd) verbunden sein. Alternativ können auch interne Pull-ups des ESP32 verwendet werden.

Beim ESP32 liegt der I²C-Bus standardmäßig an den Pins GPIO22 (SCL) und GPIO21 (SDA). Allerdings erlaubt die I²C-Bibliothek auch die Auswahl anderer Pins, falls es erforderlich sein sollte. Der I²C-Bus wird von MicroPython in vollem Umfang unterstützt.

Eine der wichtigsten Aufgaben bei der Arbeit mit dem I²C-Bus ist die Identifikation der Module und die Erfassung der zugehörigen Adresse. Das folgende Programm liefert die Adressen aller angeschlossenen I²C-Einheiten in dezimaler und hexadezimaler Schreibweise:

```
# I2Cscanner

import machine

i2c = machine.I2C(scl=machine.Pin(22), sda=machine.Pin(21))
print('Scaning i2c bus...')
devices = i2c.scan()

if len(devices) == 0:
  print("No i2c device found!")
else:
  print('i2c devices found:',len(devices))
  for device in devices:
    print("Decimal address: ",device," | Hex address: ",hex(device))
```

Falls beispielsweise ein OLED-Display und ein BMP180-Luftdrucksensor angeschlossen sind, liefert das Skript das in Bild 6 zu sehende Ergebnis.

```
Ready to download this file,please wait!
.....
download ok
exec(open('I2C_scanner.py').read(),globals())
Scan i2c bus...
i2c devices found: 2
Decimal address:  60  | Hexa address:  0x3c
Decimal address:  119  | Hexa address:  0x77
>>>
```

Bild 6: I²C-Module gefunden

Ein OLED-Display am ESP32

Als erstes Anwendungsbeispiel für den I²C-Bus kann ein OLED-Display an den ESP32 angeschlossen werden. Ein weit verbreiteter Typ ist die 0,96-Zoll-Einheit SSD1306 (siehe „Material“). Dieses verfügt über eine Auflösung von 128 x 64 Pixel und kann sowohl Texte und Daten als auch einfache Grafiken darstellen.

Das SSD1306 ist mit internem Display-RAM und einem eigenen Oszillator ausgestattet. Dadurch kann es ohne weitere externe Komponenten betrieben werden. Darüber hinaus verfügt es über eine Helligkeitsregelung mit 256 einstellbaren Stufen. Die wichtigsten Eigenschaften des Displays sind:

- Auflösung: 128 x 64 Punktmatrix
- Stromversorgung: 1,65 V bis 3,3 V
- Betriebstemperaturbereich: -40 °C bis +85 °C
- Integrierter 128 x 64-Bit-SRAM-Anzeigepuffer
- Kontinuierliche Bildlauffunktion in horizontaler und vertikaler Richtung
- On-Chip-Oszillator

Der große Vorteil von OLED-Anzeigen besteht darin, dass sie keine Hintergrundbeleuchtung benötigen. Da jedes einzelne Pixel in der Lage ist, Licht auszusenden, sind OLED-Displays auch bei ungünstigen Lichtverhältnissen noch gut ablesbar. Zudem ist der Kontrast im Vergleich zu Flüssigkristallanzeigen (LCDs) deutlich höher. Darüber hinaus verbrauchen die Pixel nur dann Energie, wenn sie tatsächlich leuchten. Damit arbeiten OLED-Displays im Vergleich zu anderen Anzeigen sehr energieeffizient.

Das hier verwendete Modell verfügt über lediglich vier Pins. Andere Versionen weisen zusätzliche Reset-Pins oder sogar eine weitere SPI-Schnittstelle auf. Für die meisten Anwendungen ist die einfache Bauform allerdings vollkommen ausreichend. Die folgende Tabelle zeigt alle erforderlichen Verbindungen (siehe auch Bild 5).

OLED-Pin	ESP32
Vin	3,3 V
GND	GND
SCL	GPIO 22
SDA	GPIO 21

Mit dem folgenden Skript können Textnachrichten und Grafikelemente auf das Display ausgegeben werden (Bild 7):

```
# text_frame.py

from machine import Pin, I2C
from ssd1306 import SSD1306_I2C

i2c = I2C(-1, scl=Pin(22), sda=Pin(21))
 # I2C Pin assignment

oled_width=128
oled_height=64
oled = SSD1306_I2C(oled_width, oled_height, i2c)

lin_hight = 9
col_width = 8

def text_write(text,lin, col):
   oled.text(text,col*col_width,lin*lin_hight)

oled.fill(0)
text_write("Programming", 0, 2)
text_write("ESP32", 2, 5)
text_write("using", 3, 5)
text_write("Micropython", 5, 2)
oled.rect(0, 39, 128, 20, 1)
oled.show()
```

Um das Display ansteuern zu können, müssen zunächst die erforderlichen Module importiert werden. Die Libraries „machine“ und „SSD1306“ sind üblicherweise als Standardbibliotheken bereits vorhanden. Im Bedarfsfall kann jedoch auch eine ssd1306.py-Datei separat auf das Board hochgeladen werden [3]. Die Pin-Deklaration erfolgt über:

i2c = I2C (-1, scl = Pin (22), sda = Pin (21))

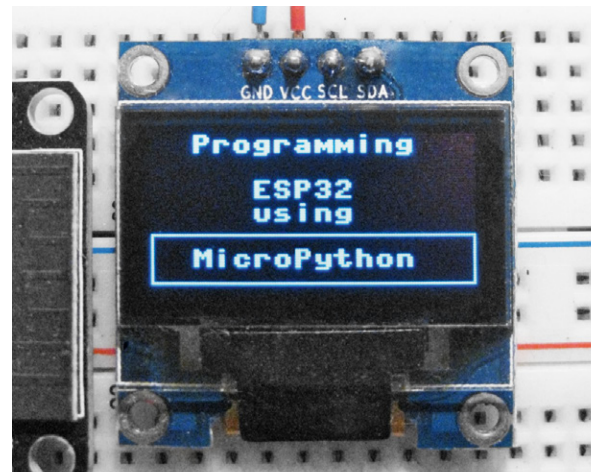


Bild 7: OLED-Display mit Textnachricht

Die Pixel-Auflösung der verwendeten Einheit wird mit folgenden Variablen

oled_width = 128
oled_height = 64

erfasst. Der Parameter „-1" weist darauf hin, dass das verwendete Modul weder über einen Reset- noch über einen Interrupt-Pin verfügt.

Nun kann ein SSD1306_I2C-Objekt mit dem Namen oled erstellt werden. Hier werden die zuvor definierten Daten übernommen:

oled = ssd1306.SSD1306_I2C (oled_width, oled_height, i2c)

Nach dem Initialisieren der Anzeige können mit der Funktion „text()" Informationen auf die Anzeige ausgegeben werden. Abschließend ist die Methode show() aufzurufen, um das Display zu aktualisieren. Die text()-Funktion akzeptiert die folgenden Argumente:

- Nachricht (Typ String)
- X-Position und Y-Position des Textfeldes in Pixeleinheiten
- Optional Textfarbe: 0 = schwarz und 1 = weiß

Mit der folgenden Zeile wird beispielsweise die Meldung „Hallo, ESP32!" in weißer Farbe angezeigt. Der Text beginnt mit x = 0 und y = 0:

oled.text ('Hallo, ESP32!', 0, 0)

Die show()-Methode sorgt schließlich dafür, dass die Änderungen auf dem Display sichtbar werden. Darüber hinaus enthält die Bibliothek noch weitere nützliche Methoden. Um den gesamten Bildschirm weiß darzustellen, kann die Funktion fill(1) verwendet werden. Das Löschen der Anzeige erfolgt über oled.fill (0). Damit werden alle Pixel auf schwarz gesetzt. Auch einfache grafische Darstellungen sind möglich. Zum Zeichnen einzelner Pixels wird die pixel()-Methode angeboten. Diese akzeptiert die folgenden Argumente:

- X-Koordinate: Pixelposition horizontal
- Y-Koordinate: Pixelposition vertikal
- Pixelfarbe: 0 = schwarz, 1 = weiß

So kann beispielsweise ein weißes Pixel in der oberen linken Ecke erzeugt werden: oled.pixel (0, 0, 1)

Die OLED-Farben können auch invertiert werden. Dabei wird Weiß zu Schwarz und umgekehrt. Dies wird über die invert()-Methode erreicht: oled.invert (True). Um zu den ursprünglichen Farben zurückzukehren, kann oled.invert (False) verwendet werden.

Grafikausgaben

Zusätzlich zu den einfachen Pixel-Befehlen stehen auch noch weitere Grafik-Anweisungen zur Verfügung. Horizontale und vertikale Linien können mit .hline () bzw. .vline () gezeichnet werden. Dabei werden die XY-Startposition sowie die Linienlänge und -farbe angegeben.

Das folgende Programm liefert einen rechteckigen Rahmen:

```
# draw frame

from machine import Pin, I2C
import ssd1306

i2c = I2C(-1, scl=Pin(22), sda=Pin(21))

oled_width = 128
oled_height = 64
oled = ssd1306.SSD1306_I2C(oled_width, oled_height, i2c)

oled.hline(0, 0, oled_width-1, 1)
oled.hline(0, oled_height-1, oled_width-1, 1)
oled.vline(0, 0, oled_height-1, 1)
oled.vline(oled_width-1, 0, oled_height-1, 1)
oled.show()
```

Diagonale Linien können mit der Methode .line(x1, y1, x2, y2, c) zwischen zwei festgelegten Punkten (x1, y1) und (x2, y2) gezogen werden. Der Parameter c steuert die Farbe der gezeichneten Linie.

Für einfache Grafiken können Bitmaps Pixel für Pixel in den Displaypuffer geschrieben werden. Das folgende Programm zeigt ein entsprechendes Beispiel, das Ergebnis auf dem Display ist in Bild 8 zu sehen:

```
# bitmap_bitmap.py

from machine import Pin, I2C
import ssd1306
import urandom

i2c = I2C(-1, scl=Pin(22), sda=Pin(21))

oled_width = 128
oled_height = 64
oled = ssd1306.SSD1306_I2C(oled_width, oled_
height, i2c)

# frame
oled.hline(0, 0, oled_width-1, 1)
oled.hline(0, oled_height-1, oled_width-1, 1)
oled.vline(0, 0, oled_height-1, 1)
oled.vline(oled_width-1, 0, oled_height, 1)
oled.show()

ICON = [
  [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0],
  [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0],
  [0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0],
  [0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
  [0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
  [0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0],
  [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0],
  [0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0],
  [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1],
]

for n in range(12):
  xofs = urandom.randint(1, oled_width-12)
  yofs = urandom.randint(1, oled_height-12)
  for y, row in enumerate(ICON):
    for x, c in enumerate(row):
      oled.pixel(x+xofs, y+yofs, c)

oled.show()
```

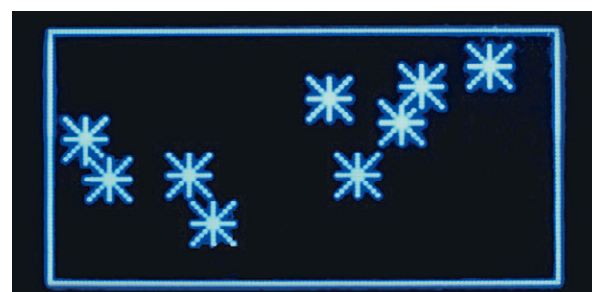


Bild 8: Bitmap-Graphiken auf dem OLED-Display

I²C-Sensoren: BMP180 für präzise Luftdruckmessung

Neben einem Display können auch verschiedene Sensoren über den I²C-Bus an den ESP32 angeschlossen werden. Ein Beispiel sind die Luftdrucksensoren der BMP/E-Serie der Firma Bosch. Diese verfügen neben dem Drucksensor selbst auch über eine präzise integrierte Temperaturerfassung. Der BMP180 ist der bekannteste Vertreter dieser Sensorbaureihe. Er wird in vielen Mobilgeräten wie Smartphones, Tablet-PCs oder elektronischen Höhenmessern eingesetzt. Durch den niedrigen Stromverbrauch ist der BMP180 besonders für batterie- oder akkubetriebene Geräte geeignet. Die Anwendungen des Sensors umfassen:

- Indoor-Navigation
- GPS-Erweiterung für Höhenbestimmung oder Hangerkennung usw.
- Höhenmesser und Sportgeräte, z. B. zur Höhenprofilerstellung
- Wettervorhersage
- vertikale Geschwindigkeitsanzeige (Anstiegs-/Sinkgeschwindigkeit)

Die wichtigsten technischen Daten des Sensors:

Druckbereich	300–1100 hPa
RMS-Rauschen	0,06 hPa, typ. (Ultra-Low-Power-Modus) 0,02 hPa, typ. (ultrahochauflösender Modus)
RMS-Höhenrauschen	0,5 m, typ. (Ultra-Low-Power-Modus) 0,17 m, typ. (ultrahochauflösender Modus)
Absolute Genauigkeit	Druck: -4,0 bis +2,0 hPa Temperatur: ± 1 °C, typ.
Stromaufnahme	3 µA, typ. (Ultra-Low-Power-Modus) 32 µA, typ. (erweiterter Modus) 650 µA, typischer Stand-by-Strom
Versorgungsspannung	1,62–3,6 V
Betriebstemperatur	-40 bis +85 °C
I²C-Übertragungsrate	3,4 MHz max.

Das folgende Programm liefert die beiden Messwerte für Druck und Temperatur:

```
# BMP180 pressure sensor

from machine import Pin, I2C
from bmp180 import BMP180
from time import sleep

i2c = I2C(-1, scl=Pin(22), sda=Pin(21))
bmp = BMP180(i2c)
bmp.oversample = 2
bmp.sealevel = 101325

while(1):
  temp = bmp.temperature
  p = bmp.pressure
  altitude = bmp.altitude
  print("Temp: ", temp, "C")
  print("Baro: ", p/100, "hPa")
  print()
  sleep(1)
```

Der das Auslesen des Sensors erforderliche Treiber steht zum kostenlosen Download zur Verfügung [4].

One-Wire-Protokoll für exakte Temperaturwerte

Neben dem I²C-Protokoll hat auch das sogenannte One-Wire-System weite Verbreitung gefunden. Ein wichtiger Sensor, der über diese Bussystem kommuniziert, ist der DS18B20. Die Datenleitung des Sensors wird über einem beliebigen I/O-Pin (z. B. Pin 25 in Bild 7) mit dem ESP32 verbunden. Zusätzlich ist lediglich ein Pull-up-Widerstand von 4,7-kΩ erforderlich. Auf diese Weise lassen sich nahezu beliebig viele Temperatursensoren parallel abfragen (Bild 9).

Das folgende Programm liefert die Werte aller angeschlossenen Sensoren:

```
# DS18x20_TempSens_demo.py

from machine import Pin
import onewire
import ds18x20
import time

ow = onewire.OneWire(Pin(25))  #Init wire
ow.scan()
ds=ds18x20.DS18X20(ow)
     #create ds18x20 object

while True:
  roms=ds.scan()          #scan ds18x20
  ds.convert_temp()        #convert temperature

  for rom in roms:
    print(ds.read_temp(rom))   #display

  time.sleep(1)
  print()
```

Die Python-Module zum Auslesen des Sensors sind wieder standardmäßig in der MicroPython-Firmware verfügbar.

Spezialprotokolle: Luftfeuchtemessung mit dem DHT11/22

Mit entsprechenden Libraries kann der ESP32 auch sehr spezielle Protokolle auslesen. Als Beispiel soll hier der bekannte Temperatur- und Luftfeuchtesensor DHT11/22 dienen. DHT-Komponenten (Digital Humidity & Temperature) sind kostengünstige digitale Sensoren und enthalten sowohl kapazitive Feuchtigkeitssensoren als auch Thermistoren zur Erfassung von Umgebungsluftparametern. Der integrierte Chip übernimmt neben der Analog-Digital-Wandlung auch die Datenkonversion für die One-Wire-Schnittstelle. Die Anwendungen des Sensors umfassen:

- Überwachung des Raumklimas
- Schutz vor Schimmelbildung
- Vermeidung von Wasserkondensation in Technikräumen

Der DHT11 kann lediglich einmal pro Sekunde, der DHT22 sogar nur alle zwei Sekunden aufgerufen werden. Ein gewisser Nachteil der Sensoren ist zudem, dass ihre Genauigkeit mit der Zeit abnimmt. Für das Auslesen der Daten werden nur drei der vier Anschluss-Pins verwendet. Die Beschaltung zeigt Bild 10.

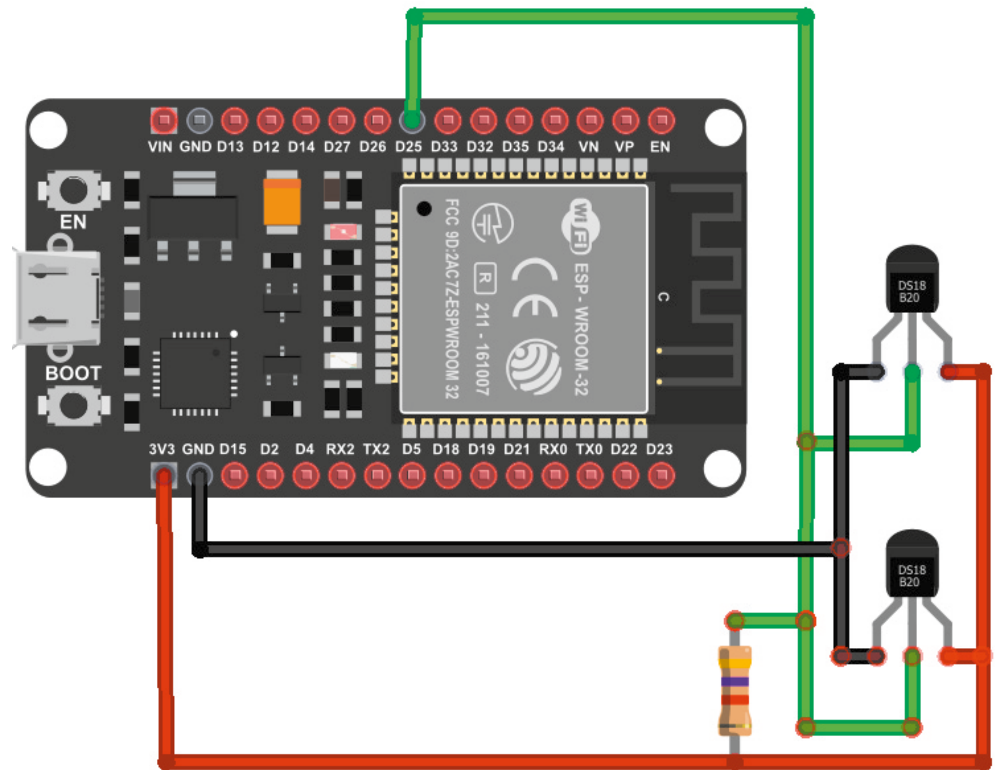


Bild 9: DS18x20 Temperatursensoren am ESP32

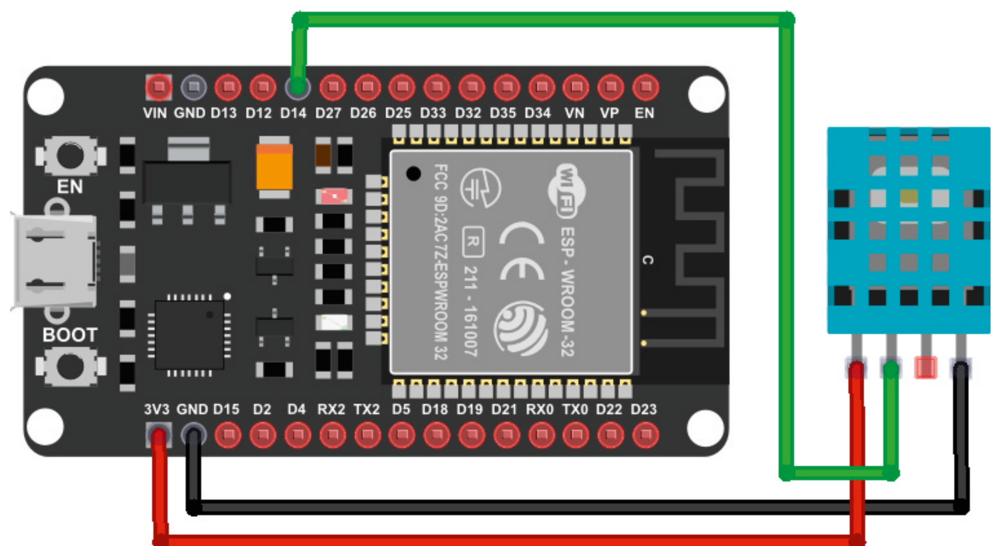


Bild 10: Luftfeuchtesensor DHT11 am ESP32

Die wichtigsten Daten der Sensoren:

	DHT11	**DHT22**
Temperaturbereich	0 bis 50 °C ±2 °C	-40 bis +80 °C ±0,5 °C
Luftfeuchtigkeitsbereich	20 bis 90 % ±5 %	0 bis 100 % ±2 %
Auflösung	Luftfeuchtigkeit: 1 % Temperatur: 1°C	Luftfeuchtigkeit: 0,1 % Temperatur: 0,1 °C
Betriebsspannung	3–5,5 VDC	3–6 VDC
Stromversorgung	0,5–2,5 mA	1–1,5 mA
Minimales Messintervall	1 Sekunde	2 Sekunden

Das Python-Modul zum Auslesen der DHT-Sensoren ist wieder in der MicroPython-Firmware enthalten. Die DHT-Sensoren bieten somit eine sehr einfache Möglichkeit, um Temperatur und Luftfeuchtigkeit zu bestimmen. Das folgende Programm liefert die aktuellen Messdaten des Sensors:

```
# DHT11_HumiTemp_sensor.py

from machine import Pin
from time import sleep
import dht

sensor = dht.DHT11(Pin(14))
#sensor = dht.DHT22(Pin(14))

while True:
 try:
  sleep(2)
  sensor.measure()
  temp = sensor.temperature()
  hum = sensor.humidity()
  print(‚Temperature: %3.1f C‘ %temp)
  print(‚Humidity: %3.1f %%‘ %hum)
  print()

 except OSError as e:
  print(‚Failed to read sensor‘)
```

Universelle Stand-alone-Klima-Messtation

Als praktische Anwendung kann nun eine vollständige Klimamesstation mit allen vorgestellten Sensoren aufgebaut werden. Das komplette Python-Programm dazu sieht so aus:

```
# ESP32_stand-alone_climate_station.py
# Baro: BMP180, Thermo: 2x DS18x20, Hygro: DHT11,
# Display: SSD1306

from machine import Pin, I2C
import ssd1306
import dht
from time import sleep
from bmp180 import BMP180
import onewire
import ds18x20

# i2c pin assignmet
i2c = I2C(-1, scl=Pin(22), sda=Pin(21))

# temp initialization DS18x20
ow = onewire.OneWire(Pin(25))  #Init wire
ow.scan()
ds=ds18x20.DS18X20(ow)        #create ds18x20 object

# baro initialization BMP180
bmp = BMP180(i2c)
bmp.oversample = 2
bmp.sealevel = 101325

# hygro initialization DHT11
sensor = dht.DHT11(Pin(14))
DHT_cal = 1

# OLED dimensions and initialization SSD1306
oled_width = 128
oled_height = 64
oled = ssd1306.SSD1306_I2C(oled_width, oled_height, i2c)
d1 = 60 # column distance 1
d2 = 90 # column distance 1
h1 = 9  # line hight

while True:
 try:
  oled.fill(0)
  oled.text(‚Climate Station‘, 0, 0)

  # DHT11: Temp & Humiture
  sensor.measure()
  temp = sensor.temperature()+DHT_cal
  hum = sensor.humidity()
  print(‚Temp1: %3.1f C‘ %temp)
  print(‚Humi : %3.1f %%› %hum)

  temperature_string = str(int(10*temp)/10)
  humidity_string = str(int(10*hum)/10)

  oled.text(‚Temp1:‘, 0 ,h1)
  oled.text(temperature_string, d1 ,h1)
  oled.text(‚ C‘, d2 ,h1)

  oled.text(‚Humi :›, 0 , 2*h1)
  oled.text(humidity_string, d1 ,2*h1)
  oled.text(‚ %‘, d2 ,2*h1)

  # BMP180: Temp & pressure
  temp = bmp.temperature
  pres = int(bmp.pressure/100)

  print(‚Temp2: %3.1f C‘ %temp)
  print(‚Baro : %3.0f hPa› %pres)

  temperature_string = str(int(10*temp)/10)
  pressure_string = str(pres)

  oled.text(‚Temp2:‘, 0 ,3*h1)
  oled.text(temperature_string, d1 ,3*h1)
  oled.text(‚ C‘, d2 ,3*h1)

  oled.text(‚Baro :›, 0 ,4*h1)
  oled.text(pressure_string, d1 ,4*h1)
  oled.text(‚ hPa‘, d2 ,4*h1)

  # DS18x20 2xTemp
  roms=ds.scan()

  ds.convert_temp()
  n = 1
  for rom in roms:
   print(‚Temp : %3.1f C› %ds.read_temp(rom))
   temperature_string = str(int(10*ds.read_temp(rom))/10)
   oled.text(‚Temp‘, 0 ,(4+n)*h1)
   oled.text(str(n+2), 32 ,(4+n)*h1)
   oled.text(‚:‘, 40 ,(4+n)*h1)
   oled.text(temperature_string, d1 ,(4+n)*h1)
   oled.text(‚ C‘, d2 ,(4+n)*h1)
   n = n + 1

  print()
  oled.show()
  sleep(1)

 except OSError as e:
  print(‚Failed to read sensor‘)
  oled.fill(0)
  oled.text(‚Sensor error‘, 0, 0)
  oled.show()
```

Alle Komponenten können auf einem größeren Breadboard untergebracht werden. Zusätzlich finden sogar noch Bauelemente wie Taster, Potenziometer und LEDs für weitere Anwendungen Platz (siehe Bild 11).

Unentbehrlich für Physical-Computing-Projekte und IoT: Servos

Servos sind inzwischen nicht nur im Modellbau weit verbreitet. Die Mini-Stellmotoren können mit geringem Hardware- und Code-Aufwand präzise positioniert werden. Prinzipiell ist dazu lediglich ein PWM-Signal mit einer Grundfrequenz von 50 Hz erforderlich. Da µP bereits über eine integrierte PWM-Signalerzeugung verfügt, ist die Erzeugung eines Servo-Signals kein Problem. Der passende Code sieht so aus:

```
# Servo_tst.py

import machine
from machine import Pin
from time import sleep

p4 = machine.Pin(18)
servo = machine.PWM(p4,freq=50)

# duty for servo is between 52...102
duty_min=52    # 52*20/1023 ms = 1.02 ms
duty_mid=77    # 77*20/1023 ms = 1.51 ms
duty_max=102  # 102*20/1023 ms = 1.99 ms

while True:
  for pos in (duty_min,duty_mid,duty_max):
    print(pos)
    servo.duty(pos)
    sleep(1)
```

Der Servo führt damit eine Bewegung vom Rechtsanschlag des Ruderhorns über die Mittelstellung zum Linksanschlag und wieder zurück aus.

Das Tastverhältnis des erzeugten Signals liegt zwischen 0 (0 %) und 210 -1 = 1023 (100 %). Da Servos ein High-Signal zwischen einer Millisekunde (Linksanschlag = 0° bzw. -90°) und zwei Millisekunden (180° bzw. +90°) benötigen, liegen die Duty-Cycle-Werte zwischen 52 und 102. Für Servos mit anderen Winkelbereichen (z. B. ±45° oder 210°) können die Duty-Cycle-Werte entsprechend angepasst werden.

Als Anwendungsbeispiel kann der Servo in einem Quasi-Analog-Anzeigeinstrument eingesetzt werden. Dazu wird anstelle des Ruderhorns ein Zeiger montiert. Mit einer geeigneten Skala können dann Werte wie auf einem klassischem Analogmessinstrument angezeigt werden. Bild 12 zeigt einen Aufbauvorschlag dazu.

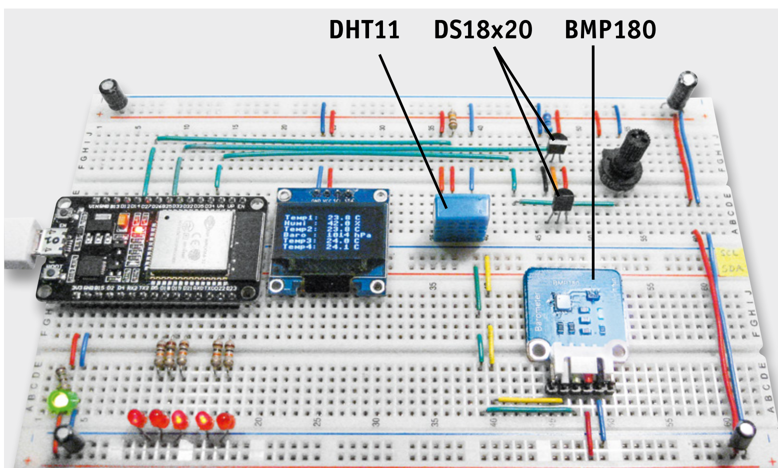


Bild 11: Klima-Messstation

Der Vorteil dieser Anzeigemethode liegt darin, dass man die Werte auch aus einer größeren Entfernung noch gut ablesen kann. Beim OLED-Display ist es bereits schwierig, die Zahlen aus einer Distanz von mehr als einem Meter abzulesen. Den Anzeigewert des Servo-Displays dagegen kann man auch noch aus mehreren Metern problemlos erkennen.

Das Programm dazu besteht aus einer Kombination des Auswertecodes für den DS18B20 und der Servosteuerung:

```
# Servo_thermometer.py

import machine
from machine import Pin, ADC
import onewire
import ds18x20
from time import sleep

p4 = machine.Pin(18)
servo = machine.PWM(p4,freq=50)

# duty for servo is between 52 and 102
duty_min = 52  # 52*20/1023  ms=1.02 ms
duty_mid = 77  # 77*20/1023  ms=1.51 ms
duty_max = 102 # 102*20/1023 ms=1.99 ms

# pot = ADC(Pin(34))
# pot.atten(ADC.ATTN_11DB)   #Full range: 3.3v

ow=onewire.OneWire(Pin(25))  #Init wire
ow.scan()
ds=ds18x20.DS18X20(ow)    #create ds18x20 object

while True:
  # pot_value = pot.read()
  # voltage = 0.000816*pot_value + 0.037822
  #print(voltage)

  roms=ds.scan()          #scan ds18x20
  ds.convert_temp()        #convert temperature
  for rom in roms:
    T=ds.read_temp(rom)
  print(T)             #test output
  pos = int(duty_max-(duty_max-duty_min)*T/50)
  servo.duty(pos)
  sleep(0.1)
```

Falls mehrere DS18B20-Sensoren am One-Wire-Bus angeschlossen sind, wird der mit der höchstwertigsten Kennung zur Auswertung herangezogen.

Der Anschluss des Servos erfolgt über Pin 18 (Signal). Darüber hinaus müssen nur noch die Spannungsversorgungsleitungen mit GND und VIN (5 V) am ESP-Board verbunden werden. Bei größeren Servos empfiehlt sich die Verwendung einer separaten Versorgung, da es sonst zu unerwünschten Spannungseinbrüchen kommen kann.

Ausblick

Nachdem in diesem Beitrag gezeigt wurde, wie man Sensoren oder einen Servo mit dem ESP in MicroPython lokal ansteuert bzw. auswertet, wird im nächsten Artikel erläutert, wie man den ESP32 in ein WLAN einbindet. Dann lassen sich die Werte drahtlos zu einem Router übertragen und zentral auswerten. Aber auch der umgekehrte Weg ist möglich.

Mit geeigneten Python-Programmen kann man LEDs oder – über entsprechende Leistungstransistoren oder Relais – auch beliebige Geräte oder Anlagen steuern.

Der Anwendung des ESP32 als universellem Steuergerät steht dann nichts mehr im Wege. Über entsprechende Sensoren kann so z. B. die Raumtemperatur individuell erfasst und sogar gesteuert werden. Aber auch die Markise kann bei Herannahen eines Sturmtiefs automatisch eingefahren werden. Dachfenster schließen sich selbstständig, sobald die ersten Regentropfen fallen, und die Jalousien regulieren über geeignete MicroPython-Programme vollautomatisch die Lichtverhältnisse in den Räumen. ELV

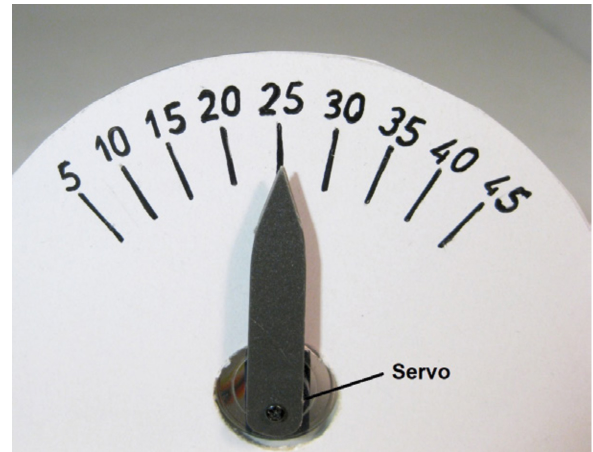


Bild 12: Quasi-Analogdisplay mit Servo-gesteuertem Zeiger

Weitere Infos:

[1] ELVjournal 6/2019: de.elv.com/journal/
[2] Download-Skripte: de.elv.com/Webcode #10304
[3] github.com/micropython/micropython/blob/master/drivers/display/ssd1306.py
[4] github.com/micropython-IMU/micropython-bmp180/blob/master/bmp180.py

Material	Bestell-Nr.
Entwicklungsplatine NodeMCU mit ESP32	14 51 64
Velleman 0,96"-OLED-Display	25 04 92
Temperatur-/Feuchtigkeitssensor DHT11	25 04 45
Temperatursensor DS18B20 1-Wire	25 04 44
Servomotor	14 51 68

ELV Newsletter abonnieren und € 5,– Bonus* sichern!

▶ Neueste Techniktrends
▶ Sonderangebote
▶ Tolle Aktionen und Vorteile
▶ Kostenlose Fachbeiträge

und vieles mehr …

*Sie erhalten einmalig € 5,- Bonus auf Ihre Bestellung, ab einem Warenwert von € 25,–. Der Gutschein gilt nicht in Verbindung mit anderen Aktionen und kann nicht ausgezahlt werden. Fachhändler und Institutionen, die bereits Sonderkonditionen erhalten, sind von diesem Bonus ausgeschlossen. Eine Auszahlung/Verrechnung mit offenen Rechnungen ist nicht möglich.

de.elv.com/newsletter
at.elv.com/newsletter · ch.elv.com/newsletter