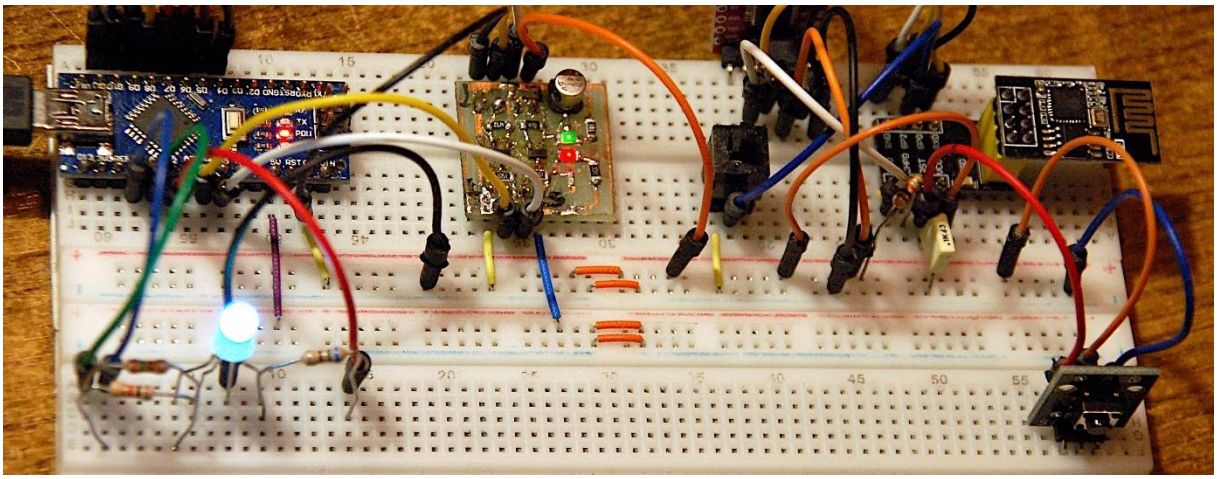

Abbildung 1: Arduino-Slave am ESP8266-01-Master

Diesen Beitrag gibt es auch als PDF-Dokument zum Download.

Im Teil 1 der Reihe Arduino goes ESP hatte ich schon angedeutet, dass man im Prinzip einen Arduino auch über den SPI-Bus direkt an einen ESP8266 oder ESP32 ankoppeln könnte. Diesen Ansatz werde ich heute in leicht abgewandelter Form wieder aufgreifen. Folge 1 und Folge 2 behandeln die Funkverbindung via NRF24L01-Modulen. Im heutigen Post werde ich den Arduino als I2C-Slave an einen ESP8266 als Master anschließen und auf diese Weise zwei Fliegen mit einer Klatsche erlegen. Zum einen wird der Arduino dadurch WLAN-fähig und zum anderen erhält der ESP8266-01 eine intelligente Porterweiterung. Neugierig geworden? Na dann, willkommen zu einer weiteren Episode zum Thema

MicroPython auf dem ESP32 und ESP8266

Untertitel:

Arduino goes ESP8266/32 via I2C (Teil 3)

Der Vorteil des I2C-Busses gegenüber dem SPI-Bus ist der, dass I2C nur zwei Leitungen statt wenigstens drei braucht. Das spielt in diesem Fall eine wesentliche Rolle, weil ich mir den kleinsten Vertreter der ESP-Familie ausgesucht habe, einen ESP8266-01. Der hat nämlich nur 2 quasi-freie GPIOs. Betrachten wir das also als Herausforderung. Hier in Teil 3 schaffen wir die Grundlagen. Wir entwickeln ein Programm für den Arduino nano, das diesen zu einem aktiven Allround-Sensor macht. Für den Arduino nutzen wir die Arduino-IDE. Für den ESP8266-01 schreiben wir eine Anwendung in MicroPython, welche es erlaubt die Register des Arduino über den I2C-Bus zu steuern und abzufragen. Aus der Sicht des ESP stellt der Arduino quasi eine intelligente Porterweiterung dar. Natürlich lassen sich außer den IO-Leitungen auch alle anderen Features des Arduino vom ESP8266-01 aus nutzen.

Hardware

1	Nano V3.0 CH340 Chip + Breadboardadapter für ESP-01 oder
1	ESP8266 01 esp-01 Wlan WiFi Modul mit Breadboardadapter
1	KY-009 RGB LED SMD Modul
2	KY-004 Taster Modul
1	Logic Level Converter TXS0108E 8 Kanal (*)
1	MB-102 Breadboard Steckbrett mit 830 Kontakten
1	FT232-AZ USB zu TTL Serial Adapter für 3,3V und 5V
2	Widerstand 2,2kΩ
1	Widerstand 560Ω
1	Widerstand 10kΩ
diverse	Jumperkabel
2	passende USB-Kabel
1	Batterie 4,5V oder 5V-Steckernetzteil

(*) alternative Möglichkeit weiter unten im Text

Aus dem Schaltplan der Entwicklungsumgebung des Projekts erkennen wir das Zusammenspiel der Teile.

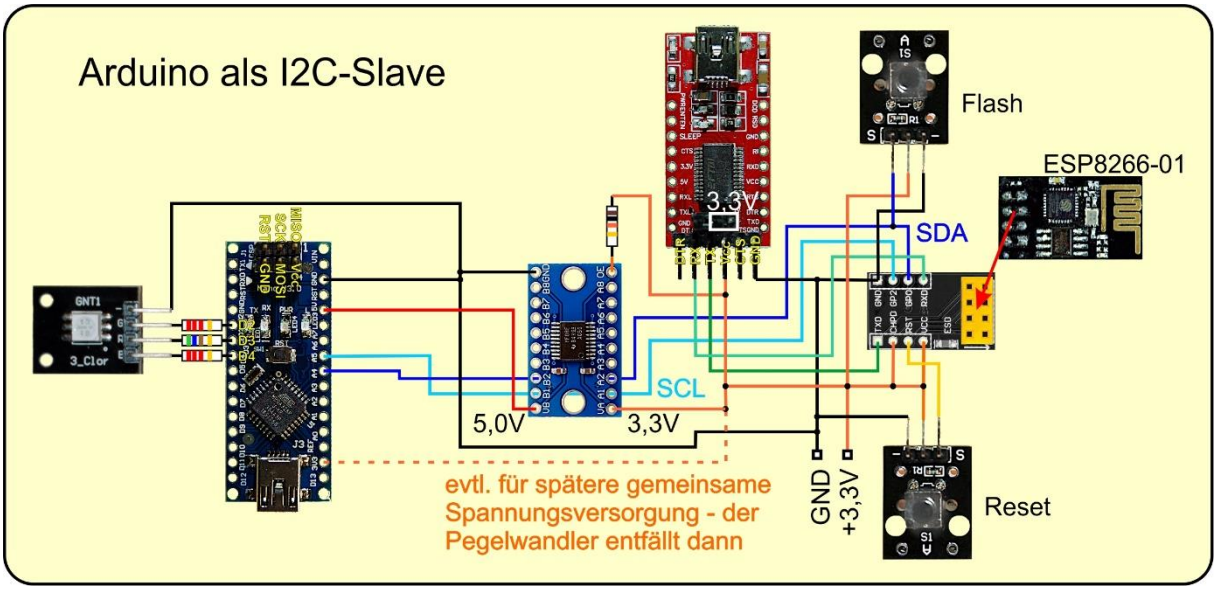

Abbildung 2: Entwicklungssystem

Ein kleines Problem stellen die beiden unterschiedlichen Betriebsspannungen von Arduino und ESP8266 dar. Das macht, zumindest während der Entwicklungsphase einen Levelshifter notwendig, welcher die Spannungspegel auf den I2C-Leitungen anpasst. Die Versorgungsspannung für den ESP8266-01 liefert der FTDI-USB-TTL-Adapter. Aber Vorsicht! Der Jumper am Ausgang gegenüber der USB-Buchse muss auf 3,3V gesetzt sein, sonst stirbt der Controller.

Letzterer stirbt auch, wenn die GPIO-Leitungen mehr als 3,6V Spannung führen. Was tun also, wenn der angegebene Typ des Level-Shifters nicht verfügbar ist? Die Lösung ist einfach. Auch in den Tagen hoch integrierter Chips ist ein diskreter Aufbau manchmal auch nicht verkehrt. Den Zusammenhang zeigt das folgende Schaltbild. Die beiden N-Kanal-MOSFETS 2N7000 sorgen dafür, dass ein 5V-Puls in einen 3,3V-Puls umgewandelt wird und umgekehrt. Andere MOSFETs sind dann auch

brauchbar, wenn deren Gate-Threshold-Spannung wie beim 2N7000 (2,1V) kleiner ist als die 3,3V der Versorgungsspannung. Wichtig: der Source-Anschluss des Transistors muss auf der Seite mit der niedrigeren Versorgungsspannung liegen.

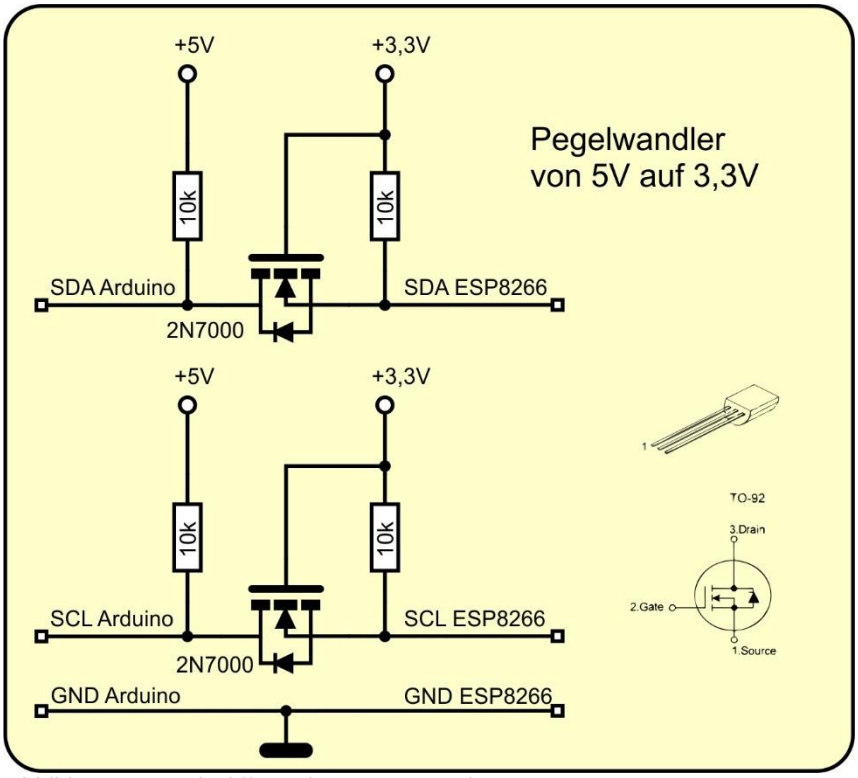

Abbildung 3: Level-Shifter mit CMOS-Transistor

Als Sockel für den ESP8266-01 verwenden wir einen breadboard-tauglichen Adapter, auf den das ESP-Modul in der angegebenen Richtung gesteckt wird. Zum Flashen der MicroPython-Firmware benötigen wir zwei Taster. Auf deren Gebrauch gehe ich später noch ein.

Der Arduino wird, wie der FTDI-Adapter, aus einem USB-Anschluss versorgt. Auf dem Arduino-Board befinden sich bereits ein USB-TTL-Adapter mit dem Chip CH340 und ein 3,3V-Regler. Damit wir die Steuerung der Ports überprüfen können, schließen wir eine RGB-LED über drei Widerstände an bestimmte IO-Leitungen an. Der Arduino hat drei Gruppen von IO-Anschlüssen, PortB, PortC und PortD. Die Bezeichnungen auf dem Board weichen davon ab, weshalb, das wissen nur die Arduino-Macher. Ich komme von der Assemblerseite und bin daher bei den AVR-Bezeichnungen B, C und D daheim. Für uns sind daher jetzt erst einmal die richtigen Zuordnungen wichtig, denn die brauchen wir nachher bei der Programmierung.

AVR Port	PD0 RXD	PD1 TXD	PD2	PD3	PD4	PD5	PD6	PD7
Arduino	D0 RXD	D1 TXD	D2	D3	D4	D5	D6	D7

AVR Port	PB0	PB1	PB2	PB3	PB4	PB5
Arduino	D8	D9	D10	D11	D12	D13

AVR Port	PC0	PC1	PC2	PC3	PC4	PC5		

Arduino	A0	A1	A2	A3	A4	A5	A6*	A7*

* Nur TQFP-Variante und nur auch dort nur analoge Eingänge

Jede Portgruppe besitzt drei Register, ein Ausgangsregister genannt PORT, ein Eingangsregister PIN und ein Datenrichtungsregister (DDR). Unser Ziel ist es, alle drei Arten in ganzer Breite zu setzen und abzufragen. Ebenfalls sollen einzelne Bitoperationen möglich sein. Kommandos und Daten wandern dabei über den I2C-Bus. Die Gruppe C tanzt etwas aus der Reihe. Die C-Leitungen dienen neben ihrer Funktion als digitale Ports auch als analoge Eingänge des Mutiplexers, der seinerseits den ADC (Analog-Digital-Converter) versorgt.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Für den Arduino
Arduino-IDE
arduino_als_slave.ino Zur Kommunikation mit dem ESP8266-01 und zum Abarbeiten von Kommandos

Verwendete Firmware für den ESP8266/ESP32:

MicropythonFirmware
Bitte eine Stable-Version aussuchen

Die MicroPython-Programme zum Projekt:

esp_i2c_master.py: Zur Kommunikation mit dem Arduino
arduino_als_slave.ino: Zur Kommunikation mit dem ESP8266-01

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung. Darin gibt es auch eine Beschreibung, wie die MicropythonFirmware (Stand 03.02.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

An dieser Stelle ein paar Takte zum Flashen des ESP8266-01. Der hat nämlich, anders als seine größeren Geschwister, keine Flash-Automatik an Bord. Hier ist Handarbeit gefragt.

Der Flashvorgang gliedert sich in zwei Teile, erstens Flash-Speicher löschen und zweitens Firmware übertragen. Für jeden Part muss beim ESP8266-01 folgendes passieren:

a) In Thonny die Vorbereitungen erledigen, wie hier beschrieben
b) Reset- und Flash-Taste drücken
c) In Thonny den Flashvorgang starten
d) Reset-Taste lösen, Flash-Taste halten, bis der Fortschritt angezeigt wird
e) Flash-Taste lösen
f) Warten bis erneut der Zugriff auf die COM-Schnittstelle gemeldet wird
g) Dann erneut die Punkte b) bis f) durchlaufen und
h) abschließend das Installer-Fenster schließen und die Options mit OK beenden

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Für den Arduino brauchen wir natürlich die Arduino-IDE. Hier kommt eine Kurzanleitung für die Installation derselben.

Einrichten der Arduino-IDE

Falls Sie noch nicht mit der Arduino-IDE gearbeitet haben und sich dieses Werkzeug noch nicht auf Ihrem Rechner befindet, folgen Sie bitte dieser Kurzanleitung.

Beginnen Sie mit dem Download der Installationsdatei über diesen Link.

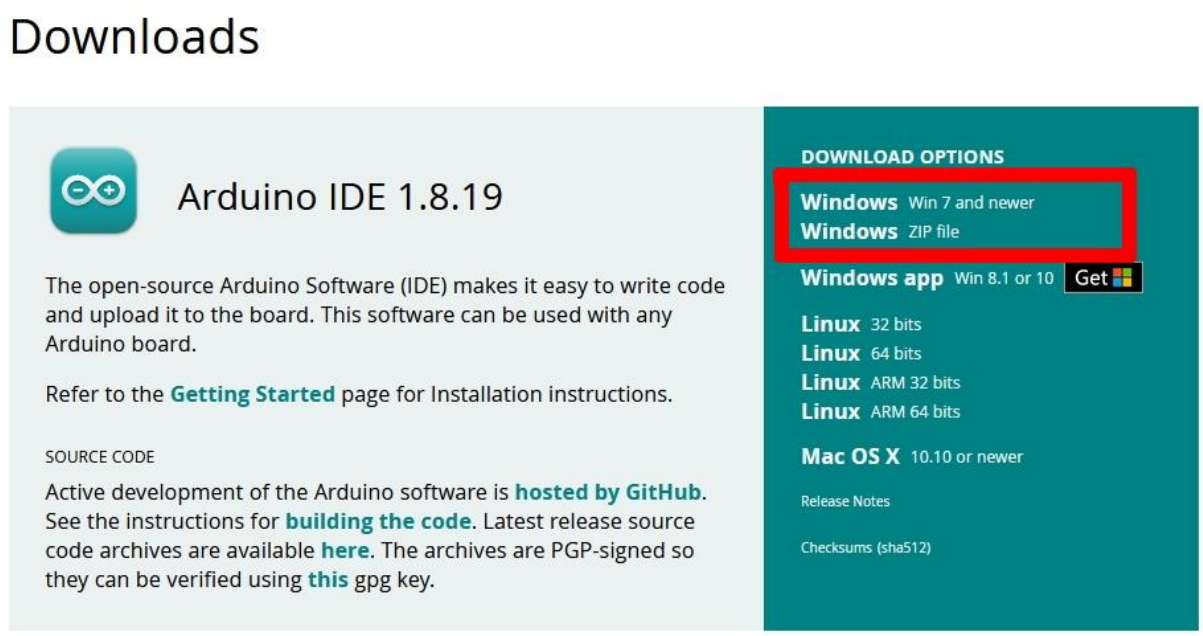

Abbildung 4: Arduino-IDE herunterladen

Klicken Sie die Version, die Ihrem Betriebssystem entspricht und speichern Sie die Datei in einem Verzeichnis Ihrer Wahl.

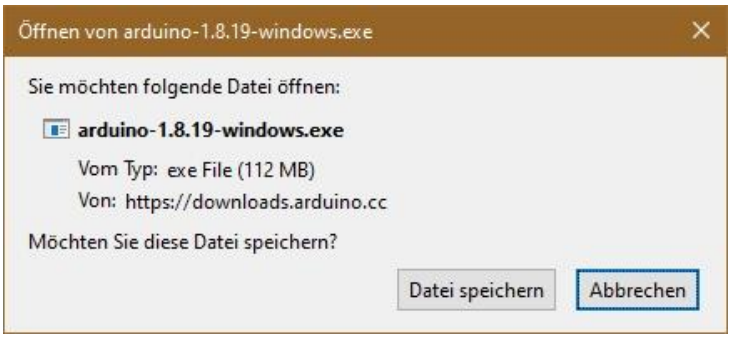

Abbildung 5: Installationsdatei speichern

Starten Sie die Installationsdatei und folgen Sie der Benutzerführung. Wir brauchen für dieses Projekt keine externen Libraries für den Arduino und sind hiermit sofort startklar.

In beiden Softwareteilen sind die einzelnen Aufgaben in Funktionen ausgelagert. Die Hauptschleife bleibt dadurch übersichtlicher und das gesamte Programm ist leichter zu pflegen. Beginnen wir mit dem MicroPython-Programm für den ESP8266-01.

Der ESP8266-01 als Master

Als Erstes sollten wir nach dem Flashen der Firmware dem ESP8266 abgewöhnen, nach einem Accesspoint zu suchen, weil das nach meinen Erfahrungen verschiedentlich zu merkwürdig eigensinnigem Verhalten des Moduls führt.

Auf der Kommandozeile von Thonny geben wir folgenden Befehl und dann "d" für disable ein. Diese Aktion sollte jedes Mal nach dem Flashen der Firmware erfolgen.

```
>>> import webrepl_setup
  WebREPL daemon auto-start status: disabled

  Would you like to (E)nable or (D)isable it running on boot?
  (Empty line to quit)
  > d
  No further action required
```

Nun zum Programm selbst. Wir gehen alles Schritt für Schritt durch.

```
from machine import SoftI2C, Pin
from time import ticks_ms, sleep

SCL=Pin(2)
SDA=Pin(0)
i2c=SoftI2C(scl=SCL, sda=SDA, freq=100000)

HWADR=i2c.scan()[0]
print("Arduino ist auf {:#x}".format(HWADR))
```

Der wichtigste Import ist der Treiber für die I2C-Schnittstelle. Es ist kein externes Modul nötig. Wir müssen also nichts zusätzlich aus dem Internet holen. Bei den Anschlüssen haben wir keine große Auswahl. Wir deklarieren die Pins und erzeugen ein I2C-Objekt. Solange kein weiteres I2C-Gerät außer unserem Arduino am Bus liegt, liefert uns der Scan-Befehl eindeutig seine Hardware-Adresse, die wir uns in hexadezimaler Schreibweise ausgeben lassen. Das ist ein erster Test, ob der Arduino auch ansprechbar ist – sobald der mit unserem Sketch gestartet wurde.

```
# Portgruppen
B=const(0); C=const(1); D=const(2)
DDRB=const(3); DDRC=const(4); DDRD=const(5)
# Kommandos
WritePort=const(0)
WriteDDR =const(1)
ReadPort =const(2)
ReadDDR  =const(3)

INPUT=const(0)
OUTPUT=const(1)
```

Wir definieren diverse Konstanten für die Portgruppen, die Kommandos und die Datenrichtung.

Der ESP8266-01 kann Kommandos für das Beschreiben oder Lesen von Registern an den Arduino senden. Diese Befehle müssen natürlich die Adresse und die zu schreibenden Daten enthalten. Die beiden grundlegenden Funktionen für diesen Zweck sind writeReg() und readReg().

```
def writeReg(command,port,value):
    buf=bytearray(3)
    buf[0]=command
    buf[1]=port
    buf[2]=value
    written=i2c.writeto(HWADR,buf)
    return written
```

Das I2C-Interface verarbeitet ausschließlich Strukturen, die auf dem bytes-Protokoll aufbauen. Wenn wir zum Beispiel versuchen, eine Zahl zu senden, bekommen wir folgende Fehlermeldung.

>>> i2c.writeto(HWADR,2)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: object with buffer protocol required

Wir erzeugen daher ein byte-Array und weisen den Elementen die übergebenen Parameterinhalte zu. Dann schicken wir die Hardwareadresse und das Array über die Leitung. Die Anzahl der übertragenen Bytes bekommen wir zurück.

```
def readReg(command,port,direction=INPUT):
    buf=bytearray(3)
    buf[0]=command
    buf[1]=port
    buf[2]=direction
    written=i2c.writeto(HWADR,buf)
    sleep(0.1)
    return i2c.readfrom(HWADR,1)
```

Ähnlich funktioniert das Einlesen von Registern. Zuerst senden wir mit dem Array das Kommando, die Adresse und die Datenrichtung. Mit INPUT wird das PIN-Register gelesen, das die logischen Pegel der als Eingang geschalteten Leitungen enthält. INPUT ist Standard, der Parameter direction kann beim Aufruf also auch weggelassen werden. Der Wert OUTPUT liest dagegen den Wert des Ausgangsregisters PORT ein.

Nach dem Fußvolk kommt jetzt die Kavallerie. Ab hier unterscheiden wir zwischen PORT, PIN und DDR. Um einen Port zu beschreiben müssen wir dem Arduino den Namen des Ports angeben und natürlich den Wert, der geschrieben werden soll.

```
def writePort(port,value):
    assert port in range(3)
    assert value in range(256)
    written=0
    try:
        written=writeReg(WritePort,port,value)
    except:
        pass
    return written
```

Wir versichern uns, dass als **port** nur die Nummern 0,1 oder 2 übergeben wurden und dass **value** im Bereich 0..255 incl. liegt. Stimmt eines von beiden nicht, wird eine Assertation-Exeption geworfen und das Programm abgebrochen. Während der Entwicklung ist das hilfreich, um Fehler aufzudecken. In einem späteren Dienstprogramm sollte es dazu nicht mehr kommen.

```
>>> writePort(5,34)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 57, in writePort
AssertionError:
```

Auch beim Schreiben auf den Bus kann es zu Fehlern kommen, etwa wenn unser Arduino gerade unpässlich ist und keine Daten empfangen kann. So etwas kann aber erst zur Laufzeit des Programms geprüft werden und darf dann nicht zum Abbruch führen. Deshalb setzen wir **written** schon einmal auf 0 und fangen etwaige Fehler durch die try-except-Struktur ab. An dem zurückgegebenen Wert der Funktion können wir dann sehen, ob Zeichen gesendet wurden. Eine analoge Funktion legen wir auch für die Datenrichtungsregister an.

```
def writeDDR(port,value):
    assert port in range(3)
    assert value in range(256)
    try:
        written=writeReg(WriteDDR,port,value)
    except:
        pass
    return written
```

Auch die Funktionen für das Lesen von Registern ähneln sich stark.

```
def readPort(port):
    assert port in range(3)
    try:
        inhalt=readReg(ReadPort,port,OUTPUT)
    except:
        pass
    return inhalt

def readInput(port):
    assert port in range(3)
    try:
```

```
        inhalt=readReg(ReadPort,port,INPUT)
    except:
        pass
    return inhalt

def readDDR(port):
    inhalt=None
    assert port in range(3)
    try:
        inhalt=readReg(ReadDDR,port)
    except:
        pass
    return inhalt
```

Als Rückgabewert erhalten wir die Inhalte des jeweiligen Registers.

Etwas aufwendiger gestaltet sich das Setzen einzelner Bits. Die Funktion soll die PORTs ebenso wie die DDRs bedienen und Bits setzen (1) aber auch löschen (0) können.

```
def setBit(port,pos,val):
    assert val in range(2)
    if port in range(3):
        cont=readPort(port)[0]
    else:
        assert 3<=port<=5
        cont=readDDR(port-3)[0]
    if val==1:
        cont |= 1<<pos
    else:
        cont &= 0<<pos
    if port in range(3):
        writePort(port,cont)
    else:
        writeDDR(port-3,cont)
```

Wir holen uns also das entsprechende Register, gemäß der Adresse.

B=const(0); C=const(1); D=const(2)
DDRB=const(3); DDRC=const(4); DDRD=const(5)

Dann setzen (durch Oderieren) oder löschen (durch Undieren) wir das bit an der Position **pos.** Denken Sie daran, dass x & 0 stets 0 ergibt und x | 1 stets 1. Nach der Operation schreiben das Register dorthin zurück wo es hergekommen ist. Ich habe die virtuellen Adressen der DDRs um 3 höhergelegt wie die Portadressen. Daher kommt die Subtraktion der 3 beim Zugriff auf die Richtungsregister.

Das Auslesen eines Bit-Werts ist unspektakulär und trotzdem trickreich. Wir holen das entsprechende Register, isolieren das Bit und schieben es in die Position von Bit0. Der Rückgabewert ist deshalb stets 0 oder 1.

Das komplette Programm für den ESP8266-01 liegt hier zum Download bereit.

Der Arduino-Slave

Dieses Projekt hat im Hinblick auf den Arduino einen Doppelnutzen. Zum einen wird gezeigt, wie man einen Arduino als I2C-Slave einsetzen kann. Zum anderen wird dadurch aber auch deutlich, wie ein handelsüblicher I2C-Baustein wie der SHT21 (Temperatur und rel. Luftfeuchte) oder der BMP280 (Luftdruck und Temperatur) intern arbeiten.

Damit der Arduino zum I2C-Slave wird, muss man nicht viel tun. Wir binden lediglich die Library Wire.h ein und erzeugen unter Angabe einer Hardware-Adresse (0x24) eine Wire-Instanz. Die Anschlüsse für scl und sda liegen beim Arduino fest auf den Pins A5 (PC5) und A4 (PC4). Alle anderen Port-Pins können wir als IO-Leitungen oder in einer ihrer Sonderfunktionen nutzen. Für ersteres ist der folgende Sketch zuständig.

```
#include <Wire.h>

void setup() {

  Wire.begin(0x24);
  Serial.begin(115200);
  Wire.onReceive(auftrag);
  Wire.onRequest(senden);
  Serial.println("Arduino-Slave on HWADR 0x24");
}

bool nachricht=false;
uint8_t received[]={0,0,0,0,0,0,0,0};
uint8_t nob=0;
uint8_t inhalt;
uint8_t command;
```

Nach der Initialisierung der seriellen Schnittstelle deklarieren wir zwei Eventhandler. Das sind Proceduren, die asynchron zum laufenden Hauptprogramm aufgerufen werden, wenn ein bestimmtes Ereignis eintritt. Das Hauptprogramm wird dadurch kurz unterbrochen. Der laufende Befehl wird noch beendet (1), dann springt der Programmzeiger an den Start des Event-Handlers (2), das Ereignis wird bedient (3) und danach erfolgt der Rücksprung ins unterbrochene Programm (4), das mit der Ausführung der nächstfolgenden Anweisung fortfährt (5).

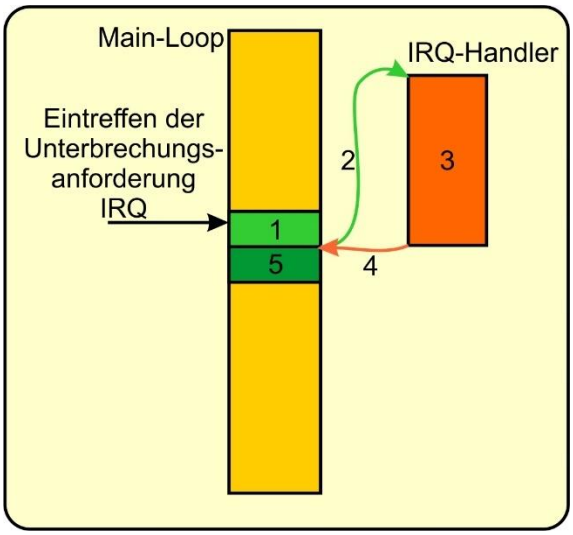

Abbildung 6: Das geschieht bei einem IRQ

Die Unterbrechungsanforderungen (aka Interrupt Request oder kurz IRQ) kommen in unserem Fall von der I2C-Schnittstelle des ATMega328. Der erste IRQ wird ausgelöst, wenn Zeichen Über den I2C-Bus ankommen. Wir werden in Kürze eine Prozedur **auftrag**() deklarieren, die dieses Ereignis bedient. Das zweite Ereignis tritt ein, wenn wir den Arduino auffordern, uns Daten zu senden. Der Event triggert die Prozedur **senden**(). Diese Art der Programmierung durch Event-Handler entbindet uns von der Aufgabe permanent das I2C-Interface selbst überwachen zu müssen (Polling).

Wir lassen uns die Hardware-Adresse im seriellen Monitor ausgeben, schließen die Setup-Prozedur ab und deklarieren noch eine Reihe von globalen Variablen. Dann betreten wir die Main-Loop.

```
void loop() {
  if (nachricht) {
    command=received[0];
    if (command==0){
      writePrt(received[1],received[2]);
    }
    if (command==1){
      writeDDR(received[1],received[2]);
      }
    if (command==2){
      readPrt(received[1],received[2]);
      }
    if (command==3){
      readDDR(received[1]);
      }
    nachricht=false;
  }
}
```

Die Variable **nachricht** wird **true**, wenn ein Schreibbefehl vom ESP8266-01 eingetroffen ist. Das erste Byte (in **received**[0]) enthält das Kommando mit dem wir dem Arduino sagen, was zu tun ist. Bei den Werten 0 und 1 handelt es sich um Schreibbefehle. Wir übergeben die Port-Nummer, die über das zweite Byte kommt und den zu schreibenden Wert aus Byte Nummer drei an die entsprechenden Proceduren.

Mit der Kommandonummer 2 fordert der ESP8266-01 ein Byte von einem Port an, der durch das Byte in **received**[1] spezifiziert ist. Der Parameter in **received**[2] sagt uns, ob das PORT-Register (Ausgabe-Puffer) oder das PIN-Register (Eingabe-Puffer) gelesen werden soll. Die Routine **readPrt**() bedient diesen Auftrag.

Zum Auslesen des DDR muss nur die Portgruppe bekannt sein, sie kommt wie üblich im zweiten Byte.

Nach dem Abarbeiten des Auftrags wird **nachrichten** auf **false** gesetzt und **loop**() lauert auf die nächste Anfrage. Durch das Auslagern der ganzen Befehle, die zum Ausführen der Anfragen nötig sind, in einzelne Prozeduren bleibt die Main-Loop klar und übersichtlich.

Kommen wir zu den Event-Handlern. Diese Prozeduren sollen als Interrupt Service Routinen (aka ISR) möglichst kurz sein. Außerdem geben sie keine Werte zurück, an wen denn auch. Schließlich werden sie nicht von einer stets gleichen Stelle des übergeordneten Prozesses aus aufgerufen, sondern sporadisch an zufälligen und wechselnden Positionen in der Hauptschleife gestartet.

```
void auftrag(int anzahl) {
  nob=0;
  while(Wire.available()){
    received[nob]=Wire.read();
    nob += 1;
  }
  nachricht=true;
}
```

In **anzahl** soll die Anzahl eingetroffener Zeichen an **auftrag**()übergeben werden. Wir zählen aber selber mit, **nob** (Number of Bytes) ist der Zähler, den wir zu Beginn auf 0 setzen und in der Schleife als Index in das Array **received**[] benutzen. Weil wir keinen Wert zurückgeben können, benutzen wir die globale Variable **nachricht** als Flag für angekommene Zeichen. Alles andere erledigt die Hauptschleife mit ihren Vasallen.

```
void senden(void){
  Wire.write(inhalt);
}
```

Die kürzeste Prozedur des Programms sendet nur den Inhalt der globalen Variablen **inhalt** über den Bus. Für den Inhalt der Variablen sind die Dienstroutinen zuständig, die auf Lesebefehle vom ESP8266-01 reagieren.

Zum Schreiben in eines der Portregister dient die Arbeits-Biene **writePrt**(). Sie erhält die Portbezeichnung und den Wert. Aufgrund der von uns vergebenen Portnummer erledigen die Schreibbefehle in der switch-Struktur schließlich den Rest.

```
void writePrt(uint8_t port, uint8_t val){
  switch (port){
    case 0:
      PORTB=val;
      break;
    case 1:
      PORTC=val;
      break;
    case 2:
      PORTD=val;
      break;
    default:
      break;
  }
}
```

In ähnlicher Weise arbeitet die Prozedur **writeDDR**().

```
void writeDDR(uint8_t port, uint8_t val){
  switch (port){
    case 0:
      DDRB=val;
      break;
    case 1:
      DDRC=val;
      break;
    case 2:
      DDRD=val;
      break;
    default:
      break;
  }
}
```

Dieses Projekt zeigt in beiden Teilen, ESP8266-01- und MicroPython-Part, verschiedene Ansätze der programmtechnischen Umsetzung auf. Vielleicht haben Sie sich gerade gedacht, die beiden Prozeduren gleichen sich doch wirklich wie ein Ei dem anderen. Die müsste man doch zusammenfassen können. Das geht in der Arduino-IDE aber nicht, denn DDRB + 1 ist eben nicht DDRC. In Assembler wäre aber genau das durch die Verwendung der AVR-internen Speicheradressen machbar, in der Arduino-IDE müsste man dafür einige Klimmzüge machen, drum lassen wir es lieber.

Wenn man neben unseren PORT-Nummern im MicroPython-Programm (B=0, C=1 und D=2) auch Nummern für die DDRs einführt, wäre eine Zusammenführung in einer Prozedur möglich. Dennoch müssten wir innerhalb der Routine wieder zwischen PORT und DDR differenzieren.

Eine Prozedur, die einen kleinen Schritt in Richtung Integration geht, ist **readPrt**(). Hier berücksichtigen wir, dass für eine Portgruppe sowohl das Ausgaberegister PORT als auch das Eingangsregister PIN ausgelesen werden muss. Möglich wird das in einer Prozedur durch die Einführung eines weiteren Parameters **dir**, mit dem wir die Richtung angeben, 0 für Eingang, 1 für Ausgang. Natürlich wird innerhalb der Routine wieder die Unterscheidung zwischen PIN und PORT erforderlich. Die Konzentration in weniger Routinen müsste man also mit mehr Parametern beim Prozeduraufruf und vor allem mit komplexerem Code im Prozedurkörper bezahlen. Um das Programm übersichtlicher und pflegeleichter zu gestalten haben wir ja gerade die Arbeitstiere aus der Main-Loop in einzelne Routinen ausgelagert.

Von der Philosophie zurück zum wirklichen Leben, hier ist die Prozedur **readPrt**(). Ich denke, zum Code selber muss ich jetzt nichts mehr weiter ausführen, die Struktur kennen wir ja schon.

```
void readPrt(uint8_t port, uint8_t dir) {
  if (dir==0){
    switch (port){
      case 0:
        inhalt=PINB;
        break;
      case 1:
        inhalt=PINC;
        break;
      case 2:
        inhalt=PIND;
        break;
      default:
        break;
    }
  }
  else{
    switch (port){
      case 0:
        inhalt=PORTB;
        break;
      case 1:
        inhalt=PORTC;
        break;
      case 2:
        inhalt=PORTD;
        break;
      default:
        break;
    }
  }
}
```

Auch die Routine zum Auslesen der DDRs bietet nichts neues.

```
void readDDR(uint8_t port) {
  switch (port){
    case 0:
      inhalt=DDRB;
      break;
    case 1:
      inhalt=DDRC;
      break;
    case 2:
      inhalt=DDRD;
      break;
    default:
      break;
  }
}
```

Test der Verbindung

Den Sketch arduino_als_slave.ino gibt es zum Download, folgen Sie einfach dem Link. Zum Testen schließen wir beide, den Arduino und den ESP8266-01, mit je einem USB-Kabel am PC an.

Zuerst laden wir den Sketch auf den Arduino hoch. Dazu stellen wir das richtige Board und die richtige COM-Schnittstelle ein. Bei mir sind das der Arduino nano und COM3.

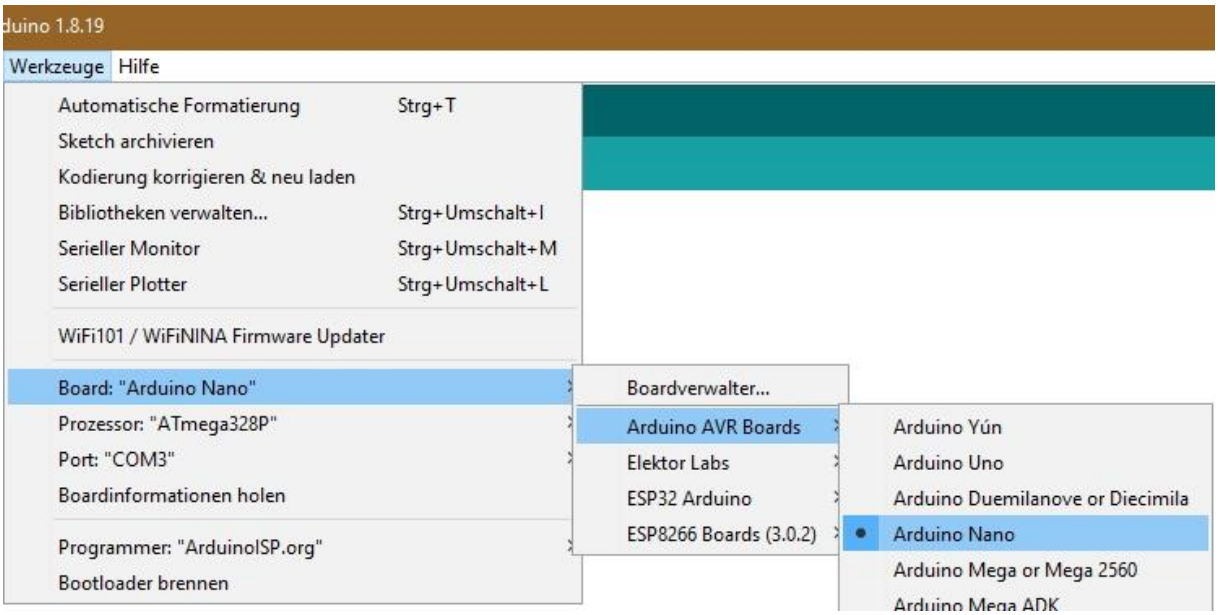

Abbildung 7: Boardauswahl

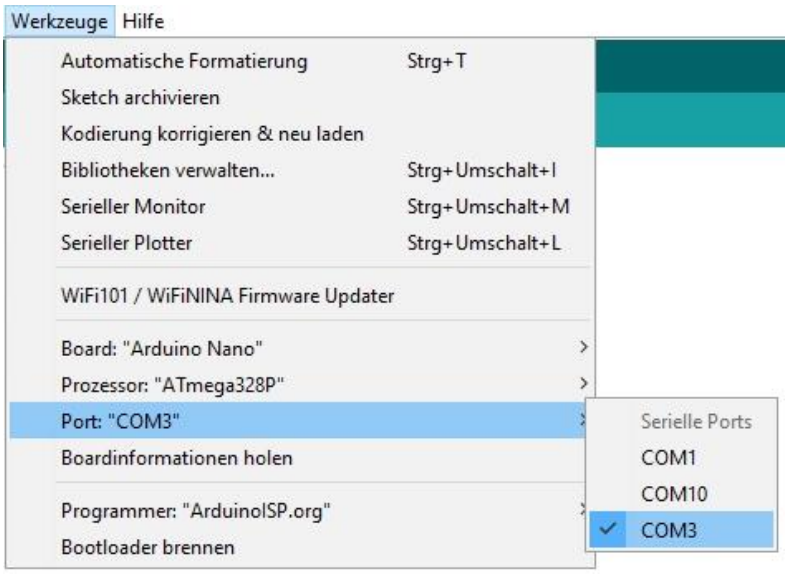

Abbildung 8: Serielle Schnittstelle auswählen

Ein Klick auf den Pfeilbutton startet den Upload des Sketches. Das dauert einige Zeit.

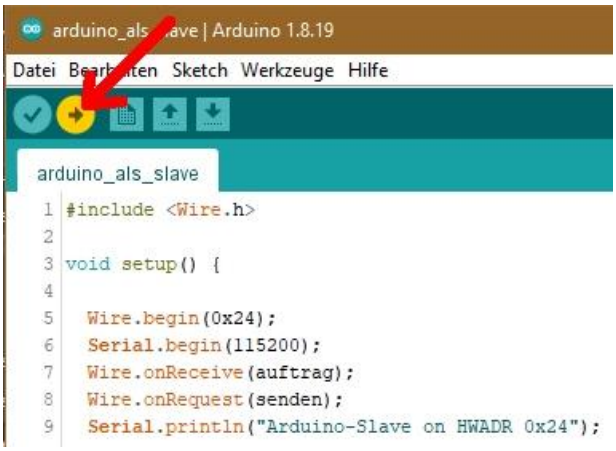

Abbildung 9: Upload starten

Mit der folgenden Meldung sagt uns die Arduino-IDE, dass die Kompilation und der Upload erfolgreich verlaufen sind.

Abbildung 10: Upload erfolgreich abgeschlossen

Dann starten wir Thonny, laden die Datei esp_i2c_master.py in ein Editorfenster und starten es mit der Funktionstaste F5. Das Programm verrät uns die Hardware-Adresse des Arduinos, initialisiert die I2C-Instanz und registriert unsere Funktionen. Jede von ihnen können wir jetzt von Hand testen, und wenn es Not tut, natürlich auch sofort nachbessern und erneut testen. Erkennen Sie den Unterschied zum Arbeiten mit der Arduino-IDE?

In der nächsten Folge werden wir die Funktionssammlung in einem Modul in einer Klasse integrieren und damit ein Programm stricken. Aber jetzt zum Test.

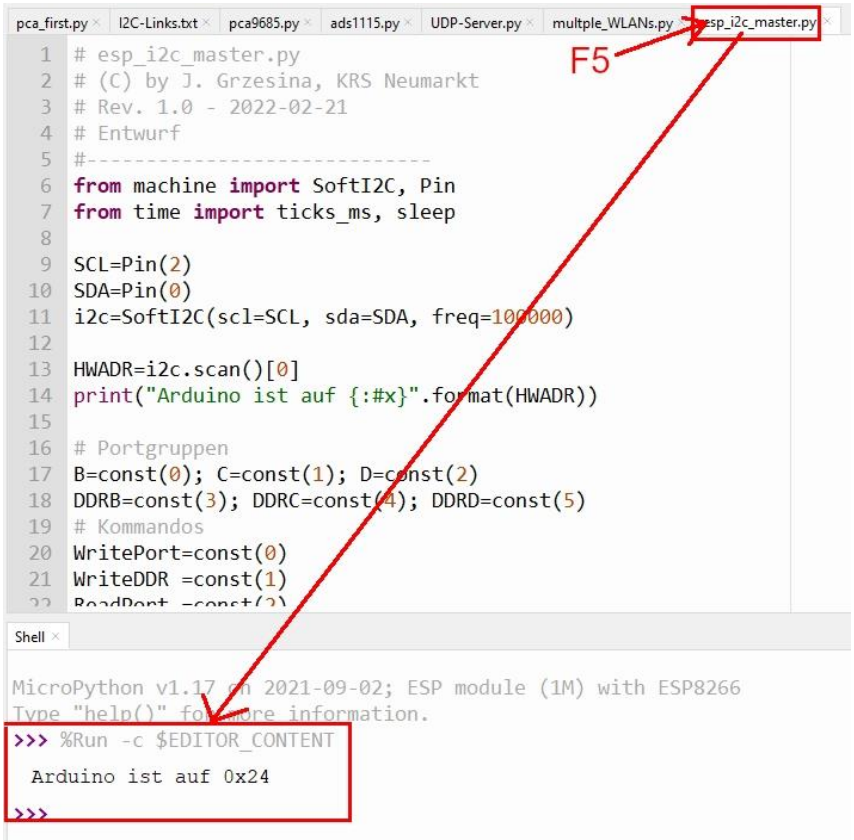

Abbildung 11: MicroPython-Programm starten

Wir haben die RGB-LED an den Pins D2, D3 und D4 liegen. Die gehören zum Port PORTD und haben die gleichen Nummern. Im DDR müssen diese Pins als Ausgang geschaltet werden. Bit2, Bit3 und Bit4 werden auf 1 gesetzt.

>>> writeDDR(D,0b00011100)

3

Es wurden 3 Bits über den Bus geschickt. Ob die auch richtig angekommen sind und verarbeitet wurden, wird sich gleich nach dem nächsten Aufruf zeigen.

>>> writePort(D,0b00011100)

3

Und sieh da – die LED gibt weißes Licht ab, es müssen also alle drei Kanäle eingeschaltet sein. Gegenprobe – die rote LED liegt bei mir an PD3 – also mal eben kurz wutschen und wedeln … lumos!

>>> writePort(D,0b00001000)

3

Ich habe soeben den grünen (PD3) und den blauen (PD4) Kanal ausgeschaltet, rotes Licht bleibt übrig.

Lassen Sie uns zum Abschluss dieses Posts noch eine kleine Lichtorgel basteln.

Geben Sie die folgenden Zeilen einfach ganz am Ende der Datei esp_i2c_master.py ein oder entfernen Sie aus dem Download die Kommentarzeichen. Nach dem Abspeichern (Strg + S) starten Sie wieder mit F5.

```
writeDDR (D,0b00011100) # PD2,PD3,PD4 auf Ausgang
writePort(D,0)          # alle Ausgangs-Leitungen auf 0
while 1:                # Dauerlauf
for i in range(8):      # von 0 bis 7 laufen lassen
    writePort(D,i<<2)   # die Bits von i um 2 Pos. nach links
    sleep(1)            # kleine Pause
```

Viel Vergnügen beim weiteren Experimentieren! Die bisher erschienen Posts zum Thema MicroPython finden Sie übrigens hier.