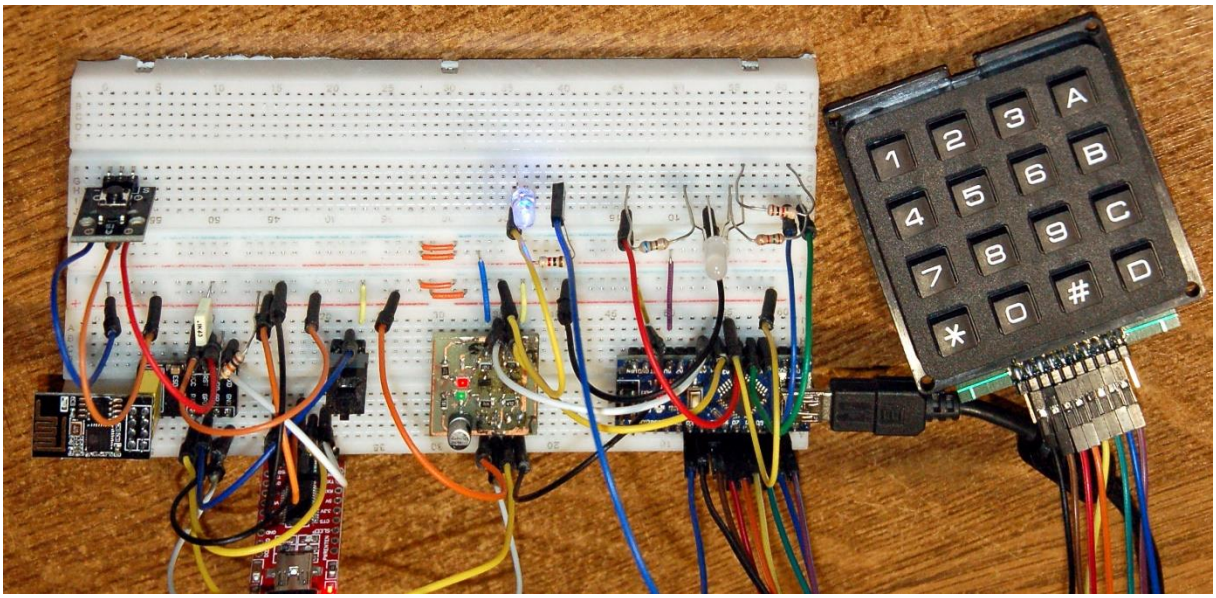

*Abbildung 1: Universal-Sensor-Aktor-Porterweiterung*

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

Kennen Sie einen Sensor, der an 4 Ports analoge Spannungen messen kann und auf 6 Leitungen PWM-Signale absetzen und 18 digitale IO-Leitungen bedienen? Und das mit einem einzigen Baustein! Außerdem ist das kleine Ding über I2C-zu kontakten. Und man kann damit eine "intelligente" Tastatur realisieren, die eine Tastenabfrage mit Timeout bietet. Und sicher noch eine ganze Menge mehr!

Das gibt es nicht, meinen Sie? Na, dann lade ich Sie herzlich ein, mich auf dieser Tour durch den Blogpost aus der Reihe

# MicroPython auf dem ESP32 und ESP8266

zu begleiten. Abbildung 2 und der Untertitel lassen Sie vielleicht schon erahnen, was Sie erwartet.

## Arduino goes ESP8266/32 via I2C (Teil 4)

Das kleine Ding rechts unten auf dem Breadboard sieht nicht nur aus wie ein Arduino nano, es ist auch einer. Und genau das ist der Star des heutigen Beitrags, neben dem ESP8266-01 in der linken unteren Ecke das Entwicklungsboards.

In der vorangegangenen Episode hatte ich die Hardware und deren Verknüpfung zum I2C-System behandelt. An den Teilen hat sich bis auf den Zuwachs der 4x4-Matrix-Tastatur, einer LED und dem zugehörigen Widerstand nichts geändert, an der Verdrahtung schon.

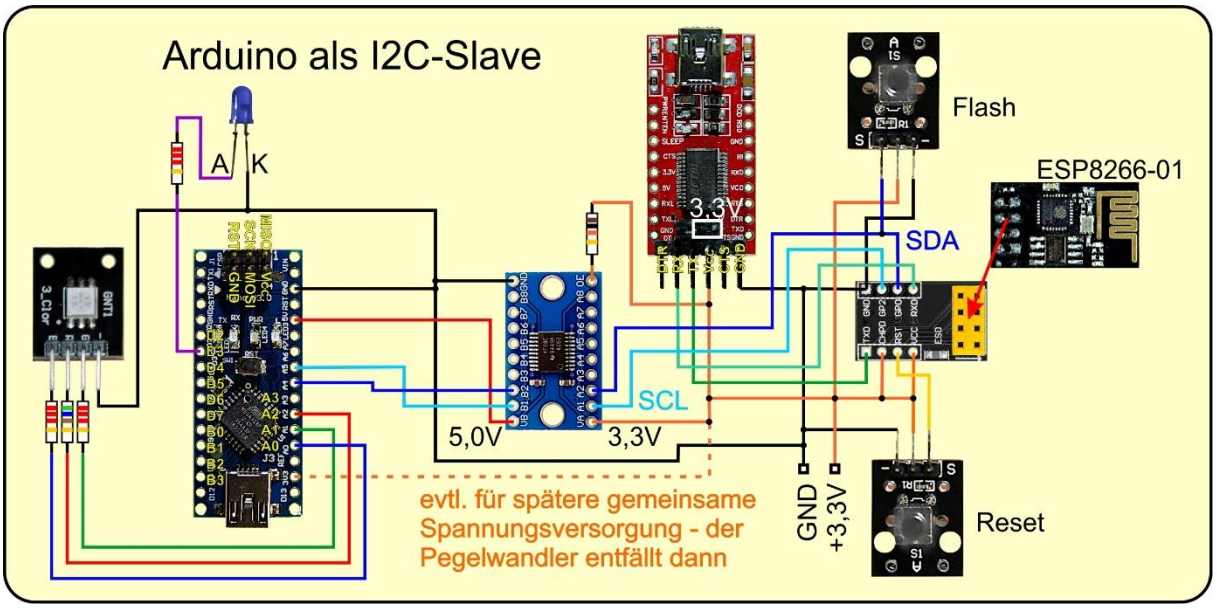


*Abbildung 2: IO-Ports und PWM-LED-Steuerung*

## 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 |
| 1 | 4x4 Matrix Array Keypad Tastenfeld Tastatur oder<br>4x4 Matrix Keypad Tastatur |
| 1 | LED (blau oder grün, passend zum Widerstand 2,2kΩ |
| 3 | 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

In der letzten Folge hatte ich schon darauf hingewiesen, dass, falls Ihnen das Leben Ihres ESP8266-01 lieb ist, Sie dafür Spannungen von mehr als 3.6V von seinen Tentakeln fernhalten müssen. Das betrifft die Versorgungsspannung aus dem FTDI232-Adapter, aber auch die I2C-Leitungen zum Arduino. Deshalb befindet sich dort auch ein Level-Shifter als Dolmetscher, der die Signale auf dem Bus zwischen 5V und 3,3V übersetzt. Falls Sie kein solches Modul zur Hand haben, hilft auch ein Aufbau mit diskreten Bauteilen. Den Zusammenhang zeigt das folgende Schaltbild.

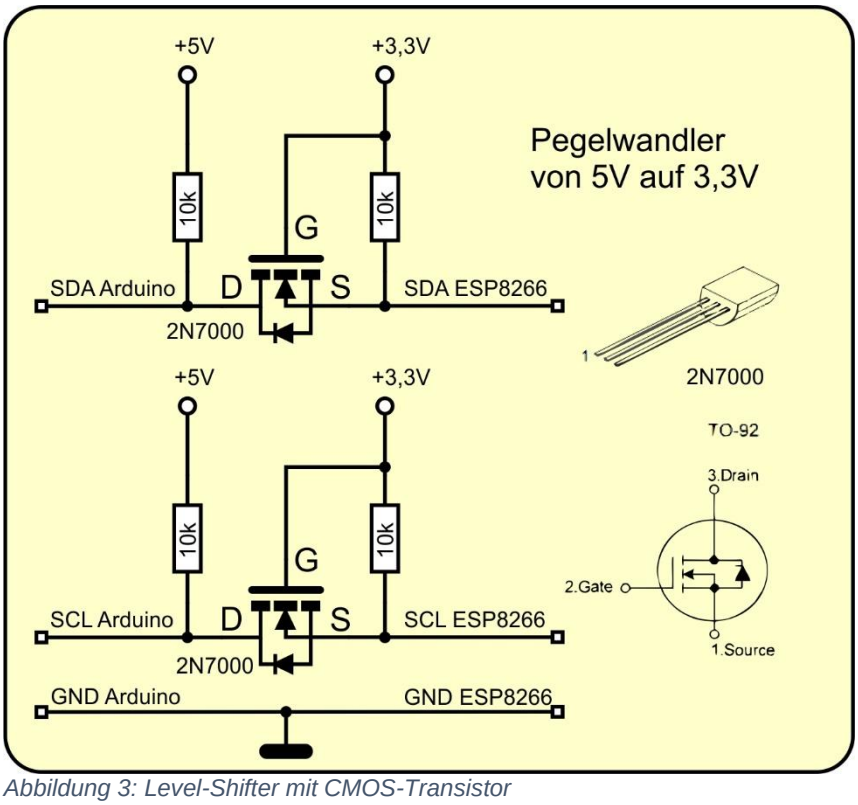


*Abbildung 3: Level-Shifter mit CMOS-Transistor*

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, wie die 3,3V der Versorgungsspannung. Wichtig: der Source-Anschluss des Transistors muss auf der Seite mit der niedrigeren Versorgungsspannung liegen.

Der AVR-Controller ATMega328 auf dem 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-Maker. 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. Wir sprechen hier ganze Ports an und nicht einzelne Leitungen, das hat einige Vorteile.

| AVR Port | PD0<br>RXD | PD1<br>TXD | PD2 | PD3 | PD4 | PD5 | PD6 | PD7 |
|---|---|---|---|---|---|---|---|---|
| Arduino | D0<br>RXD | D1<br>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 beschreiben und abzufragen. Ebenfalls sollen einzelne,

gezielte Bitoperationen möglich sein. Kommandos und Daten wandern dabei über den I2C-Bus, der ESP8266-01 ist der Auftraggeber.

Die Gruppe C tanzt etwas aus der Reihe. Die C-Leitungen dienen neben ihrer Funktion als digitale Ports auch als Eingänge des analogen Mutiplexers, der seinerseits die Signale dem ADC (Analog-Digital-Converter) zuführt. Den Eingang A3 werden wir heute in diesem Sinn einsetzen. A4 und A5, alias PC4 und PC5, stellen den Anschluss zur I2C-Hardware her. Auf dem Arduino sind die Funktionen dieser Schnittstelle durch Hardware realisiert.

Damit wir die Steuerung der Ports überprüfen können, schließen wir eine RGB-LED über drei Widerstände an bestimmte IO-Leitungen an. Dafür dienen die Ports PC0 (blau), PC1 (grün) und PC2 (rot).

## 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
tastatur.ino Erweiterter Sketch

### Verwendete Firmware für den ESP8266/ESP32:

MicropythonFirmware
Bitte eine Stable-Version aussuchen

### Die MicroPython-Programme zum Projekt:

arduino_i2c.py: Zur Kommunikation mit dem Arduino
ardutest.py: Testprogramm für das Modul arduino_i2c.py

## 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. Die folgende Liste ist ein Auszug aus der Beschreibung für den Flashvorgang:

a) In Thonny die Vorbereitungen erledigen
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.

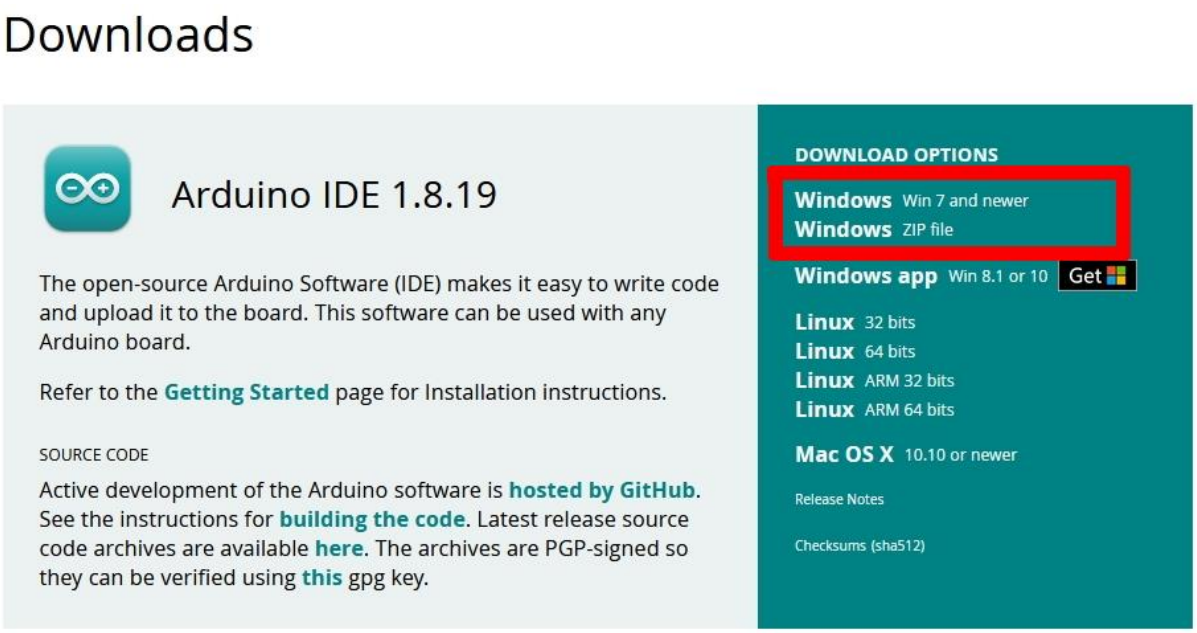


*Abbildung 4: Arduino-IDE herunterladen*

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

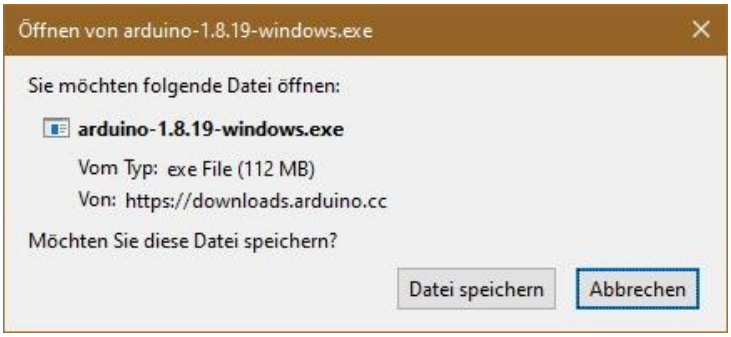


*Abbildung 5: Installationsdatei speichern*

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 und besprechen die Änderungen, die seit der Vorversion stattgefunden haben.

## Der ESP8266-01 mit den Änderungen und Erweiterungen

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 (nur einmal) 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
```

In der vorigen Folge hatte ich kurz Gedanken zu der Art vorgestellt, wie ein Programm gestaltet sein sollte. Dass ein Entwurf, in welchem alle Strukturen und Befehle in der Hauptschleife untergebracht werden, sehr unübersichtlich und daher schwer zu lesen und zu pflegen ist, leuchtet ein.

Ebenso unübersichtlich ist eine ellenlange Liste von Sprüngen aus der Main-Loop zur ebenso langen Folge von Mini-Prozeduren, welche einzelne Jobs erledigen. Jobs zusammenzufassen und dafür in den Prozeduren zu verzweigen, um alle Fälle abzuarbeiten ist ebenfalls kontraproduktiv, weil das die Prozeduren aufbläht. Lange Parameterlisten in den Aufrufen und schwer zu pflegende Funktionskörper sind also auch nicht das gelbe vom Ei.

Für dieses Projekt habe ich mich, bei zunehmender Komplexität der Aufgaben, bemüht, einen Kompromiss zu suchen. Und ich meine, eine brauchbare Lösung gefunden zu haben, die noch dazu die Assemblerschreibweise beim Portzugriff widerspiegelt. Assemblerbefehle mit Register- oder Speicherzugriffen werden ebenso gelesen wie Zuweisungen in den Hochsprachen, von rechts nach links. Aber, sehen Sie selbst.

Die Funktionen für den portweiten Zugriff auf den IO-Bereich habe ich in jeweils einer einzigen Methode vereint. Methode, nicht Prozedur oder Funktion? Pardon, ja, beziehungsweise nein. Ich habe nämlich alle Funktionen und Konstanten aus der letzten Folge inzwischen in ein Modul mit dem Namen arduino_i2c.py verpackt. Und

im Zusammenhang mit der darin befindlichen Klasse ARDUINO, werden Funktionen als Methoden bezeichnet. Die Klasse erlaubt es mir, Werkzeuge in beliebig vielen Projekten einzusetzen. Trotzdem muss ich die Datei nur an einer Stelle pflegen.

Die Syntax (aka Schreibweise) der Methodenaufrufe lehnt sich also an die Assemblerschreibweise an.

Assembler:
out PORTC, 0x07  # schreibe 0x07 in das PORT-Register der Gruppe PortC

Methode aus der Klasse ARDUINO
ARDUINO.writeIO(PORT,C,0x07)

Und was der Arduino nach dem Empfang des Kommandos tut sieht so aus.

PORTC=0x07

Sie erkennen sicher, den Vorteil dieses Zugriffs gegenüber der üblichen Vorgehensweise, einzelne Pins separat zu bedienen. PORTC=0x07 ist übrigens ein Befehl, der standardmäßig in der Arduino-IDE verfügbar ist, ohne Import irgendwelcher Libraries. Aber das finde ich nur leider nicht direkt in der Sprachreferenz der Arduino-Crew. Seltsam ist nur, weshalb das offiziell verschleiert wird. Mecker beiseite, was gibt es Neues?

Wie entsteht aus einer lockeren Folge von Konstanten-Definitionen und Funktionen ein Modul mit einer Klasse?

Die Konstanten haben Zuwachs bei den Kommandos und bei der Datenrichtung bekommen. Die Portgruppe wurde um die DDRs abgespeckt. Zum Vergleich bitte ich, die Datei esp_i2c_master.py herunterzuladen. In U0 steht der Wert der Versorgungsspannung am 5V-Pin des Arduino, der mit einem DVM (aka Digitales-Volt-Meter) bestimmt wird und in der Regel ca. 5V betragen sollte

```
from time import sleep
HWADR=const(0x24)
# Portgruppen
B=const(0); C=const(1); D=const(2)
# Kommandos
WRITEPORT=const(0)
WRITEDDR =const(1)
READPORT =const(2)
READDDR  =const(3)
GETKEY   =const(4)
ANALOGOUT=const(5)
ANALOGIN =const(6)
# Daten-Richtung
PIN =const(0)
PORT=const(1)
DDR =const(2)
# Betriebsspannung des Arduino
U0  =4.8
```

Als Nächstes folgen die Klassen-Definition und die Codierung des Konstruktors **__init__**(). Im Programm wird er später über den Namen der Klasse aufgerufen. Der Konstruktor ist die Routine, die ein Objekt, auch Instanz genannt, dieser Klasse erzeugt. Hier werden Anfangswerte gesetzt, und Instanzvariablen erzeugt, die von den Methoden (=Funktionen) der Klasse genutzt werden können. Unsere Klasse heißt also ARDUINO. Ähnlich wie bei der while-Schleife oder der Methoden-Deklaration, wird nach der Deklaration der Klasse eine Stufe eingerückt.

```
class ARDUINO:

    def __init__(self,i2c,hwadr=None,u0=None):
        self.i2c=i2c
        if u0 is not None:
            self.U0 = u0
        else:
            self.U0 = U0
        if hwadr is None:
            try:
                self.hwadr==i2c.scan()[0]
            except:
                self.hwadr=HWADR
        else:
            self.hwadr=hwadr
        print("Constructor of Arduino-Interface")
        print("Arduino @ {:#x}".format(self.hwadr))
```

Bei der Instanziierung eines ARDUINO-Objekts wird das I2C-Objekt, welches im Hauptprogramm zu erstellen ist, übergeben. Beim Aufruf des Konstruktors kann die Hardware-Adresse des Arduino-Slaves übergeben werden. Wird der Parameter **hwadr** beim Aufruf nicht angegeben, dann sucht **i2c.scan**() nach dem Ziel oder es wird der Default-Wert **0x24** verwendet, wenn der Scan fehlschlägt. Das behandeln wir mit **try – except**. In ähnlicher Weise wird der Wert für die Betriebsspannung des Arduino gesetzt. Die Ergebnisse werden uns durch die print-Befehle mitgeteilt.

Sicher ist Ihnen der Parameter self in der Parameter-Liste aufgefallen. Der ist so zu verstehen, dass damit eine Referenz (aka Bezug, Adressierung) auf das erzeugte Objekt übergeben wird. Alle Variablen und Methoden innerhalb der erzeugten Klasseninstanz erhalten damit eine Bindung und Kapselung auf dieses Objekt. Das grenzt sie von etwaigen weiteren Instanzen derselben Klasse ab.

**writeReg**() und **readReg**() heißen jetzt also Methoden. Bei beiden bleibt der Code bis auf das zusätzliche **self** unverändert. Durch den Bezug auf das Objekt selbst durch **self**, wird die i2c-Instanz an beliebigen Stellen in den Methoden der Klasse verfügbar, ohne dass sie in einer Parameterliste übergeben werden muss.

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

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

Wir können die beiden Methoden als Bindeglied zwischen der I2C-Schnittstelle mit ihren Methoden und den Methoden unserer Anwendung werten. Beide Methoden arbeiten bei der Übertragung mit Datenstrukturen, die auf dem Bytes-Protokoll aufsetzen, nehmen aber Integerwerte an, die sie für die Sendung in ein Bytearray umsetzen.

Interessant wird es jetzt bei den Methoden für den Zugriff auf die Ports.

```
    def writeIO(self,reg,port,value):
        assert port in range(3)
        assert value in range(256)
        assert reg in [PORT,DDR]
        try:
            if reg==PORT:
                written=self.writeReg(WRITEPORT,port,value)
            else:
                written=self.writeReg(WRITEDDR,port,value)
        except:
            pass
        return written
```

Die Methode **writeIO**() fasst die Schreibbefehle auf Port- und DDR-Register zusammen. Im Vergleich zu den Vorgängerversionen, die noch zwischen PORT und DDR unterschieden haben, wurde Speicherplatz gespart, weil weniger Programmtext anfällt. Erkauft wurde das durch den Parameter **reg**, der zusätzlich in die Liste aufgenommen wurde. Das ist gut vertretbar, weil damit einige Programmzeilen gespart werden konnten. Im gleichen Zug wurde deren Plausibilitätsprüfung vereinfacht. Die try-except-Struktur wurde nur geringfügig um die if-Struktur ausgeweitet, die jetzt PORTs und DDRs bedient. Dennoch erspart dieses Vorgehen die Definition einer kompletten zweiten Routine.

Ähnlich verhält es sich mit den Methoden **readIO**(), **setBit**() und **getBit**().

```
    def readIO(self,reg,port):
        assert port in range(3)
        assert reg in range(3)
        inhalt=None
        try:
            if reg==PIN:
                inhalt=self.readReg(READPIN,port,PIN)
            elif reg==PORT:
                inhalt=self.readReg(READPORT,port,PORT)
```

```
            else:
                inhalt=self.readReg(READDDR,port)
        except:
            pass
        return inhalt
```

Die Methode **setBit**() benötigt neben **self** vier Argumente. Weil sie nicht nur Bits auf 1, sondern auch auf 0 setzen kann, muss außer der Bitposition auch dieser Wert angegeben werden. Durch die Verwendung des Parameters **reg** muss innerhalb **setBit**() nicht mehr zwischen PORT und DDR unterschieden werden. Das überlassen wir den Methoden **readIO**() und **writeIO**(). Der Programmtext wird dadurch von 15 auf 10 Zeilen verkürzt.

```
    def setBit(self,reg,port,pos,val):
        assert val in range(2)
        assert reg in [PORT,DDR]
        assert port in range(3)
        cont=self.readIO(reg,port)[0]
        if val==1:
            cont |= 1<<pos
        else:
            cont &= 255-(1<<pos)
        self.writeIO(reg,port,cont)
```

Auch **getBit**() wurde verbessert. Statt mit 8 Zeilen nur PORT und DDR zubedienen, schaffen es die verbliebenen 6 Zeilen auch noch PIN mit einzubeziehen.

```
    def getBit(self,reg,port,pos):
        assert reg in [PIN,PORT,DDR]
        assert port in range(3)
        cont=self.readIO(reg,port)[0]
        cont &= 1<<pos
        return cont>>pos
```

Soviel zum Umbau des bestehenden Programms zu Modul und Klasse. Es gibt aber auch noch einige Erweiterungen. Da sind zum Beispiel die Methoden **voltage**(), **readAnalog**() und **writeAnalog**.

**voltage**() dient dazu, den mit einem Voltmeter gemessenen Wert der Betriebsspannung des Arduino dem Programm mitzuteilen. Er ist wichtig für die korrekte Berechnung der Spannungen, die mit den analogen Eingängen erfasst werden. Ohne Argument aufgerufen, gibt die Methode den Wert im Speicher zurück. Wird beim Aufruf ein Spannungswert übergeben, schreibt ihn die Methode in die Instanz-Variable U0. Beispiel:

```
>>> a.voltage()
4.73
```

```
    def voltage(self,value=None):
        if value is not None:
            self.U0=value
        else:
            return self.U0
```

Der Name von **readAnalog**() ist dem Kommando **analogRead**() der Arduino-IDE nachempfunden wurde aber bewusst verändert, damit die Umgebung eindeutig zugeordnet werden kann. In **line** wird die Nummer des Analogeingangs am Arduino angegeben. Hat der optionale Parameter **voltage** den Wert **True**, dann wird ein Spannungswert in Volt zurückgegeben. True ist der Defaultwert, mit dem gearbeitet wird, wenn man das Argument beim Aufruf weglässt. Der Parameter **digits** ist ebenso optional und standardmäßig mit 2 Nachkommstellen vorbelegt, wenn dieser Parameter beim Aufruf nicht angegeben wird.

```
    def readAnalog(self,line,voltage=True,digits=2):
        buf=bytearray(2)
        buf[0]=ANALOGIN
        buf[1]=line
        written=self.i2c.writeto(self.hwadr,buf)
        sleep(0.1)
        low,high=self.i2c.readfrom(self.hwadr,2)
        counts=low+256*high
        if voltage:
            pot=10 ** digits
            return int(self.U0*counts/1023*pot) / pot
        else:
            return counts
```

Mit der Übermittlung von Aufträgen an den Arduino sind wir mittlerweile schon vertraut. Wir füllen den Sendepuffer mit den Werten der Argumente, schicken die Post ab und holen nach einer kurzen Verschnaufpause den Messwert ab. Der 16-Bit-Wert kommt in Form von zwei Bytes im Format Little-Endian. Das heißt, das niederwertige Byte LSB kommt vor dem höherwertigen MSB an. In **counts** werden die beiden wieder zusammengesetzt.

Falls **voltage** True ist, berechnen wir aus dem Zählwert vom ADC und dem Wert der Betriebsspannung in **self.U0** den Wert der gemessenen Spannung, der dann final auf die gewünschte Anzahl von Nachkommastellen (ab-)gerundet wird. Ist **voltage** False, wird einfach nur der Zählwert des ADC zurückgegeben.

Eine programmierte, glatte Gleichspannung kann der ATMega328 nicht abgeben, denn er besitzt keinen DAC (Digital-Analog-Converter). Aber er kann über die PWM-Kanäle 3, 5, 6, 9, 10 und 11 Rechteck-Pulse verschiedener Länge aber fester Frequenz abgeben. Das läuft unter dem Namen **P**uls-**W**eiten-**M**odulation. LEDs oder Gleichstrommotoren, die man mit so einem Signal ansteuert reagieren mit unterschiedlich heller Lichtabgabe oder veränderter Drehzahl. Beispiele folgen später.

```
    def writeAnalog(self, line, value):
```

```
        buf=bytearray(4)
        buf[0]=ANALOGOUT
        buf[1]=line
        buf[2]=value & 0xFF
        buf[3]=(value >> 8) & 0xFF
        written=self.i2c.writeto(self.hwadr,buf)
        return written
```

Die Werte für **value** können standardmäßig für alle Kanäle von 0 bis 255 gehen. Man kann aber diese Auflösung steigern, wenn man an den Timer-Registern des Arduino ein bisschen rumfummelt. Es lässt sich dann auch die Taktfrequenz des PWM-Signals verändern. Das erlauben, leider nur für die PWM-Ausgänge D9 und D10, die Methoden **resolution**() und **prescaler**().

Die Auflösung an D9 und D10 kann 8, 9, oder 10 Bit betragen. Diese Werte übergeben wir dem Methodenaufruf. Die Methode prüft zunächst die Einhaltung des Bereichs, füttert dann das Array mit dem Befehl **RESOLV** und dem Auflösungs-Wert in **val** und schickt das Array an den Arduino. Was der damit tut, das schauen wir uns später an.

```
    def resolution(self, val):
        assert val in range(8,11)
        buf=bytearray(2)
        buf[0]=RESOLV
        buf[1]=val
        written=self.i2c.writeto(self.hwadr,buf)
        return written
```

Ähnlich verhält sich die Methode **prescaler**(). Sie nimmt einen der Werte aus der Tabelle und sendet den Index dieses Werts. Ungültige Parameterwerte werden aussortiert.

```
    def prescaler(self,val):
        werte=[0,1,8,64,256,1024]
        try:
            psFaktor=werte.index(val)
            buf=bytearray(2)
            buf[0]=PRESCALER
            buf[1]=psFaktor
            written=self.i2c.writeto(self.hwadr,buf)
            return written
        except:
            print("falscher Wert")
            return None
```

Den Zusammenhang der PWM-Frequenz mit den anderen Parametern Taktfrequenz des ATMega328, Auflösung und Vorteiler des Timers1 kann man der folgenden Abbildung 6 entnehmen. Das entsprechende Kalkulationsblatt liegt zum Download bereit.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | PWM-Frequenzen beim Arduino | | | | |
| 2 | | | | | |
| 3 | Taktfrequenz | 8 | MHz | | |
| 4 | | | | | |
| 5 | | | | | |
| 6 | PWM-Frequenz = | Taktfrequenz / (Vorteilerwert * Auflösung in Stufen) | | | |
| 7 | | | | | |
| 8 | | | | | |
| 9 | Vorteiler | | Auflösung | | |
| 10 | | 8 | 9 | 10 | Bit |
| 11 | | 256 | 512 | 1024 | Stufen |
| 12 | 1 | 31250,0 | 15625,0 | 7812,5 | |
| 13 | 8 | 3906,3 | 1953,1 | 976,6 | |
| 14 | 64 | 488,3 | 244,1 | 122,1 | |
| 15 | 256 | 122,1 | 61,0 | 30,5 | |
| 16 | 1024 | 30,5 | 15,3 | 7,6 | |

*Abbildung 6: PWM-Frequenzen beim Arduino*

Durch Messungen am DSO (Digitales Speicher Oszilloskop) und das Auslesen der Register TCCR1A und TTTR1B des Timers 1 konnte ich verblüfft feststellen, dass mein Arduino nano mit 8HMz getaktet wird statt mit 16MHz.

Kommen wir zur letzten neuen Methode.

```
    def getKey(self,timeout=0):
        buf=bytearray(3)
        buf[0]=GETKEY
        buf[1]=timeout
        written=self.i2c.writeto(self.hwadr,buf)
        key=0x7f
        while key==0x7f:
            key=self.i2c.readfrom(self.hwadr,1)[0]
        return chr(key)
```

**getKey**() gibt mit dem Kommando **GETKEY** und dem **timeout**-Wert in Sekunden das Einlesen einer Taste von einem 4x4-Matrixblock am Arduino in Auftrag. Wird in der Einlese-Schleife der Wert 0x7F gelesen, dann bedeutet das, dass innerhalb des Zeitfensters noch keine Taste gedrückt wurde. Der Wert 0xFF würde uns sagen, dass die Wartezeit überschritten wurde. Der Arduino weiß, welchen ASCII-Code er uns beim Drücken einer Taste schicken muss. Die Tastenwerte sind in der Regel kleiner als 127 aber letztlich (fast) frei belegbar.

Damit wird es Zeit, uns mit dem Sketch für den Arduino zu beschäftigen.

## Arduino – der multifunktionale Sensor.

Wenn wir schon dabei sind, bleiben wir doch gleich bei der Tastenabfrage. Die MicroPython Methode **getKey**() greift zweimal auf den Arduino zu. Ein Schreibbefehl gibt die Abfrage einer Taste in Auftrag. Weil MicroPython aber nicht wissen kann, wann wir geneigt sind, eine Taste zu drücken, wird der Lesebefehl nicht nach einer

bestimmten Verzögerung, wie beim Einlesen von Port-Registern erteilt, sondern mit ständiger Wiederholung in einer Schleife. Was tut der Arduino zwischendurch?

Die Funktion **key**() erledigt die eigentliche Abfrage. Weil wir mit ganzen Ports und nicht nur mit Einzelleitungen arbeiten können, nutzen wir die Reversal-Technik. Deren Ablauf stelle ich jetzt vor

Die Tastatur besteht aus zweimal 4 Leitungen. Die Zeilen R0 bis R3 kreuzen sich mit den Spalten S0 bis S3. Durch Drücken an den Kreuzungspunkten wird ein Kontakt hergestellt.

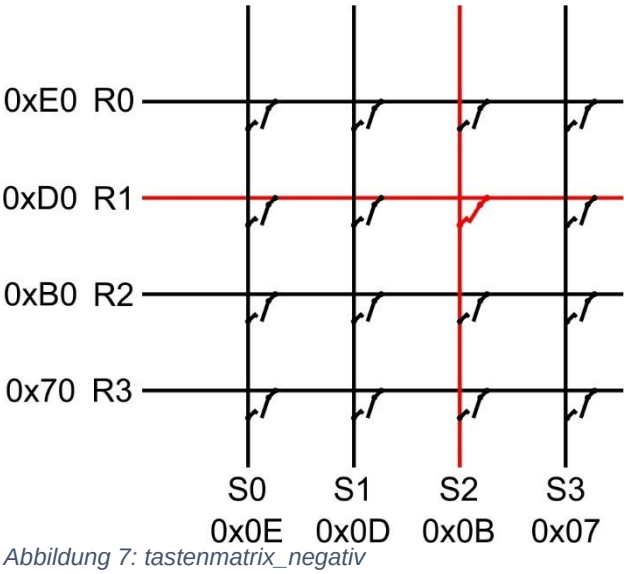


*Abbildung 7: tastenmatrix_negativ*

Wir verbinden jetzt die Reihen mit den Portleitungen D4 bis D7 und die Spalten mit den Portleitungen B0 (D8) bis B3 (D11). Einen vergrößerten Ausschnitt aus dem Schaltbild aus Abbildung 8 gibt es hier.

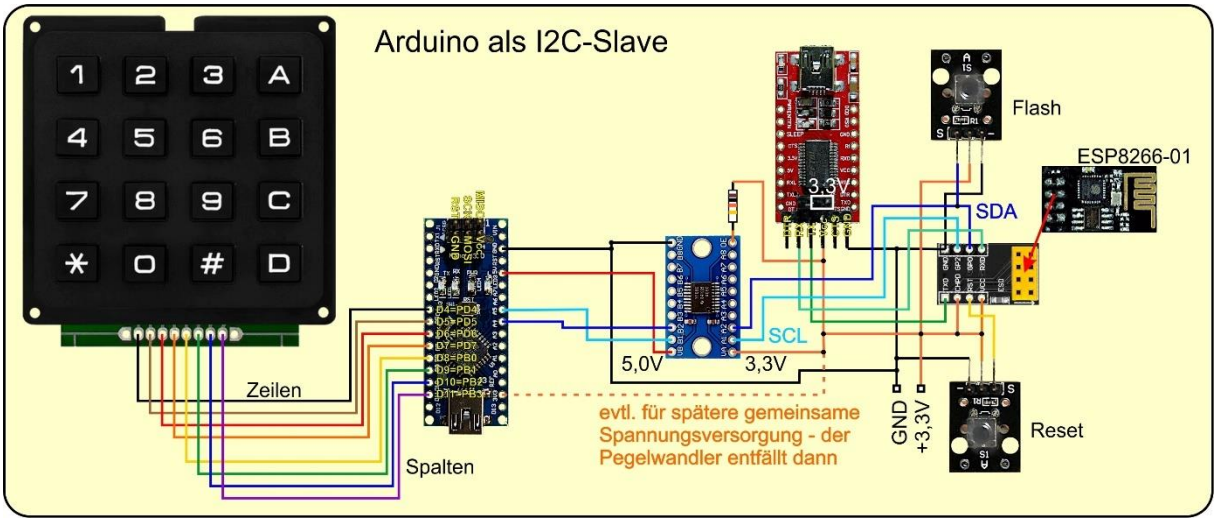


*Abbildung 8: I2C-Tastatur*

Jetzt zum Programm. Wir definieren die Maske **mask**, deren High-Nibbel lauter Einsen und deren Low-Nibbel lauter Nullen enthält.

```
char key(bool ascii) {
  uint8_t const mask = 0b11110000; // D4..7 ist anfangs Output
  uint8_t const reverse = byte(~mask);
  uint8_t const keynumber[]={0x7D,0xEE,0xED,0xEB,
                      0xDE,0xDD,0xDB,0xBE,
                      0xBD,0xBB,0xE7,0xD7,
                      0xB7,0x77,0x7E,0x7B};
  char const
asciicode[]={'0','1','2','3','4','5','6','7','8','9','A','B','
C','\x0A','*','+'};
  uint8_t iod;
  uint8_t code;
  uint8_t number;
  code=0xFF;
  number=0xFF;
```

```
  DDRD |= mask;     // Zeilen Output D4..7
  DDRB &= mask;     // Spalten Input B0..3
  PORTB |= reverse; // mit Pullup
  PORTD &= reverse; // Zeilen auf 0; D4..7
  delayMicroseconds(50);
```

Mit DDRD oderiert, machen die Einsen PORTD.4 bis PORTD.7 zu Ausgängen. Dieselbe Maske mit DDRB undiert, lässt PINPB.0 bis PINPB.3 als Eingänge wirken. Die Variable **reverse** enthält den negierten Wert von **mask**, Einsen und Nullen haben also den Platz getauscht. Wenn wir das Low-Nibbel von PORTB jetzt mit Einsen beschreiben, schalten wir die zugehörigen Pullup-Widerstände der Eingangsleitungen ein. Das spart uns externe Widerstände an den Portleitungen. Jetzt setzen wir das High-Nibbel von PORTD durch Undieren mit **reverse** auf 0000. Wird die Taste 6 gedrückt, dann landet die 0 von R1 über S2 am Pin PINB.2. Die anderen Pins werden durch Pullups auf 1 gezogen. Kurze Wartezeit.

```
  iod=PINB & reverse;// Spalten einlesen B0..3 => Lownibble
  DDRD &= reverse;  // Zeilen Input D4..7
  PORTD |= mask;    // Pullup an D4..7
  DDRB |= reverse;  // Spalten Ausgang B0..3
  uint8_t out = PORTB;
  out &= mask;  // B0..3 auf 0
  out |= iod;
  PORTB =out;   // Lownibble auf Port
  delayMicroseconds(50);
```

Den Zustand am Eingang B lesen wir in die Variable **iod** ein und isolieren die unteren 4 Bits, indem wir das High-Nibbel auf 0 undieren. Nun drehen wir den Spieß um. Das Low-Nibbel von PORTB wird zum Ausgang, das High-Nibbel von Port D zum Eingang mit aktivierten Pullups.

PIND wird eingelesen, das Low-Nibbel gelöscht, dann durch das Low-Nibbel von iod ersetzt und das Ganze Byte wieder in den PORTB oderiert. S2 ist jetzt auf 0 gesetzt,

die restlichen Spalten sind auf 1 und das High-Nibbel von PORTB wurde nicht verändert, man weiß ja nie was diese Pins grade steuern. Kurzes Warten.

```
  code = (PIND & mask)>>4;  // Highnibble isolieren
  code = code | (iod<<4);   // Tastencode kombinieren
  for (uint8_t i=0;i<16;i++){
    if (code == keynumber[i]){
      number=i;
      break;
    }
  }
```

Weil ich die Tabelle **keynumber**[ ] berreits vorab definiert hatte und nicht mehr ändern wollte, tausche ich High-Nibbel und Low-Nibbel kurzerhand aus. Ich hätte auch die Beschaltung ändern können, aber dann hätte die Zeichnung in Abbildung 8 nicht mehr gestimmt. Jedenfalls enthält **number** jetzt den Tastencode 0xDB.

Mit der for-Schleife ermittle ich den Index des Tastencodes in der Tabelle **keynumber**[ ]. Das geht in der Arduino-IDE leider nicht so komfortabel wie in MicroPython, dort würde das ein Befehl machen: keynumber.index(code).

```
  char taste = '\x7F'; // 0x7F --> keine Taste
  if (number < 16) {
    taste=asciicode[number];
  }
  if (ascii) {
    return taste;
  }
  else {
    return number;
  }

}
```

Schließlich belegen wir **taste** mit **0x7F** ("unberührt") vor. Ist der Tastencode in **number** kleiner 16, dann wurde eine Taste gedrückt und ich ermittle schon einmal deren ASCII-Code mit Hilfe der Tabelle **asciicode**[ ].

Abschließend gebe ich in Abhängigkeit von der boolschen Variablen **ascii** entweder den ASCII-Code oder die Tastennummer (0..15) zurück. Wurde keine Taste gedrückt, dann enthalten taste und number beide den Wert 0xFF.

Die Funktion **wait**() bringt uns die Funktionalität des Wartens auf eine Taste mit oder ohne Timeout.

```
void wait(uint8_t timeout){
  //inhalt=0x7F; // bisher keine Taste
  tosend[0]=0x7F;
  cnt=1;
```

```
  long jetzt=millis();
  long ende=jetzt+timeout*1000;
  while (true) {
    uint8_t k=key(true);
    if (k < 0x7F) {
      // Taste abzuholen
      tosend[0]=k;
      return;
    } // if
    if (timeout==0){
      ende=millis()+10000;
    } // if
    if (millis()>ende) {
      // Timeout
      tosend[0]=0xFF;
      return;
    } // if
  } // while 1
}
```

Der Sendepuffer wird mit 0x7F (noch kein Tastendruck) vorbelegt. Wir ermitteln die Startzeit und berechnen die Schlusszeit in Millisekunden.

In der while-Schleife schauen wir nach, ob eine Taste gedrückt wurde. Ist der von **key**() zurückgegebene Wert kleiner 127, dann schreiben wir ihn in den Sendepuffer. Der Timeout ist ausgeschaltet, wenn für dessen Wert eine 0 an **wait**() übergeben wurde. Wir erhöhen dann die Ablaufzeit vom aktuellen Stand in **millis**() um 10 Sekunden. Dadurch wird die nachfolgende Abbruchbedingung niemals wahr. Andernfalls wird **millis**() nach der vorgegebenen Zeit einen höheren Wert als **ende** liefern und wir schreiben den Code einer nicht gedrückten Taste, also 0xFF in den Sendepuffer, was für Timeout steht.

Nach den beiden Mammut-Funktionen kommen jetzt noch die 4 mickrigen Dinger für die Analogwertbehandlung.

**analogInput**() ist das Backend der MicroPython-Methode **readAnalog**(). Die Funktion nimmt die Nummer des analogen Eingangs und sendet den ausgelesenen 16-Bit-Wert in Form von zwei Bytes zurück. Die Byteanordnung ist Little-Endian, also Low-Byte zuerst.

```
void analogInput(uint8_t line){
  cnt=1;
  uint16_t counts=analogRead(line);
  tosend[0]=counts & 0xFF;
  tosend[1]=counts>>8;
  cnt=2;
}
```

**analogOutput**() nimmt drei Parameter, die Nummer des PWM-Pins in Arduino-Notation sowie Low- und High-Byte des PWM-Werts. Der Wert wird wieder

zusammengesetzt und dann ganz normal mit dem **analogWrite**()-Befehl ausgegeben.

```
void analogOutput(uint8_t line, uint8_t low, uint8_t high) {
  uint16_t value = low+(high<<8);
  analogWrite(line,value);
}
```

Der ATMega328 besitzt drei Hardwaretimer. TCNT0 und TCNT2 sind 8-Bit-Zähler, TCNT1 ist ein 16-Bit-Zähler. Die Counter TCNT0 und TCNT1 bekommen ihr Taktsignal von einem Prescaler (aka Vorteiler), der vom IO-Systemtakt abgeleitet wird. Die Bits 0 bis 3 im Timer-Counter-Controll-Register1B wählen von 5 möglichen Werten einen aus.

| TCCR1B.2 | TCCR1B.1 | TCCR1B.0 | Teiler |
|---|---|---|---|
| 0 | 0 | 0 | Takt aus |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 8 |
| 0 | 1 | 1 | 64 |
| 1 | 0 | 0 | 256 |
| 1 | 0 | 1 | 1024 |

Der Teiler wird eingestellt, indem man im TCCR1B die Bit-Werte entsprechend setzt. **val** bringt die vom ESP8266-01 gesendete Konfiguration mit. TCCR1B wird ausgelesen, die drei untersten Bits durch Undieren gelöscht, mit **val** oderiert und Das Byte nach TTCR1B zurückgeschrieben.

**Warnung:**
**Die Veränderung von Prescaler-Wert und Auflösung kann andere Prozesse, die ebenfalls den Timer TCNT1 nutzen beeinflussen!**

```
void setPrescaler(uint8_t val){
  TCCR1B = (TCCR1B & 0xF8) | val;
}
```

Je größer die Auflösung, also die Bitbreite eines PWM-Werts ist, um so feinere Abstufungen sind möglich. Der TCNT1 bietet mit seinem 16-Bit breiten Zählerregister die Möglichkeit, zwischen einer 8, 9 und 10 Bit breiten Auflösung. Eingestellt wird das mit den Bits 1 und 0 im TCCR1A-Register. Der ESP8266-01 übermittelt die gewünschte Bitbreite. Wenn ich von diesem Wert in **val** 7 subtrahiere, erhalte ich die drei verschiedenen Bitwerte, die ich in das Register TCCR1A oderieren muss.

| TCCR1A.1 | TCCR1A.0 | Auflösung | Anzahl der Stufen |
|---|---|---|---|
| 0 | 1 | 8 | 255 |
| 1 | 0 | 9 | 511 |
| 1 | 1 | 10 | 1023 |

```
void aufloesung(uint8_t val){
  TCCR1A = (TCCR1A & 0xFC) | (val-7);
```

```
  }
```

Die Formel, nach der die PWM-Frequenz zu berechnen ist, habe ich in Abbildung 7 schon einmal angegeben.

PWM-Frequenz = IO-System-Takt / (Vorteiler * Anzahl der Stufen)

Bleibt noch ein Blick auf die erweiterte Main-Loop. Hinzugekommen sind die fett dargestellten Zeilen. Natürlich könnte man das Aufdröseln der Kommandos eleganter mit einer switch – case – Struktur lösen.

```
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]);
      }
    if (command==4){
      wait(received[1]);
      }
    if (command==5){
      analogOutput(received[1],received[2],received[3]);
      }
    if (command==6){
      analogInput(received[1]);
      }
    if (command==7){
      aufloesung(received[1]);
      }
    if (command==8){
      setPrescaler(received[1]);
      }
    nachricht=false;
  }
}
```

## Testphase

Wie in der letzten Folge flashen wir zunächst den Arduino. Sie können den Sketch tastatur.ino hier komplett herunterladen.

Für den ESP8266-01 brauchen wir zwei Dateien. Das Modul arduino_i2c.py können Sie ebenfalls herunterladen. Kopieren Sie die Datei in ihr Arbeitsverzeichnis um es im Workspace von Thonny anzuzeigen. Mit einem Rechtsklick auf die Datei öffnen Sie das Kontextmenü. Laden Sie nun die Datei auf den ESP8266-01 hoch. Als nächstes brauchen Sie die Datei ardutest.py, die Sie auch herunterladen oder den Text selbst in einem neuen Editorfenster eingeben können.

```
from machine import SoftI2C, Pin
from arduino_i2c import *
# from time import ticks_ms, sleep

SCL=Pin(2)
SDA=Pin(0)
i2c=SoftI2C(scl=SCL, sda=SDA, freq=100000)
a=ARDUINO(i2c,u0=4.73)
```

Messen Sie jetzt die Betriebsspannung Ihres Arduino am 5V-Pin mit einem DVM und tragen Sie den Messwert statt 4.73 in der letzten Zeile ein.

Danach starten Sie das Programm mit F5.

```
>>> %Run -c $EDITOR_CONTENT
  Constructor of Arduino-Interface
  Arduino @ 0x24
```

Jetzt können alle Methoden der Klasse ARDUINO im Terminalbereich von Thonny von Hand aufgerufen werden. Dazu benutzen wir die erzeugte Instanz a. Beachten Sie bitte die Groß-Klein-Schreibung!

Beginnen wir mit der RGB-LED.

```
a.writeIO(DDR,C,7)          # Bit 0 bis 2 von PORTC auf Ausgang
  3

a.writeIO(PORT,C,1)         # Bit 0 auf 1
  3
```

Und die blaue LED leuchtet.

```
a.writeIO(PORT,C,2)         # Bit 1 auf 1, Bit 0 auf 0
  3
```

Aus blau wird grün.

```
a.writeIO(PORT,C,4)         # Bit 2 auf 1, Bit 1 auf 0
  3
```

Wechselt grün zu rot.
oder

```
a.writeIO(PORT,C,0b110)   # Bits 2 und 1 auf 1, Bit 0 auf 0
  3
```
Das ergibt gelb, wenn die Widerstände auf die LEDs abgestimmt sind.

```
a.writeIO(PORT,C,0b101)   # Bits 2 und 0 auf 1, Bit 1 auf 0
  3
```
Rot und blau ergibt magenta.

```
a.setBit(DDR,D,3,1)       # Bit 3 von PORTD auf Ausgang
a.setBit(PORT,D,3,1)      # LED leuchtet
a.setBit(PORT,D,3,0)      # LED erlischt
```

Verbinden Sie jetzt Eingang A3 mit der 3,3V-Versorgung.

```
a.readAnalog(3)
3.13
```
Die Spannung auf der 3,3V-Leitung ist 3,13V. Wiederholte Aufrufe bringen bei 2 Nachkommastellen meist (nahezu) identische Werte.

```
>>> a.readAnalog(3,False)
679
```
Das ist der Rohwert vom ADC. Raten Sie mal, was bei dieser Rechnung herauskommt?

Ganzzahl(679 *4,73 /1023 * $10^2$) / $10^2$ = ?

```
>>> a.readAnalog(3,digits=4)
3.1348
```
Der optionale Parameter **voltage** wird per default mit True belegt aber der Parameter **digits** muss explizit genannt werden.

```
>>> a.writeAnalog(3,127)
4
>>> a.writeAnalog(3,64)
4
>>> a.writeAnalog(3,32)
4
>>> a.writeAnalog(3,16)
4
```
Die LED an D3 wird zunehmend dunkler.

```
>>> a.writeAnalog(3,127)
4
>>> a.writeAnalog(3,255)
4
>>> a.writeAnalog(3,192)
4
```
Die Helligkeitsunterschiede in der oberen Hälfte sind weit weniger markant.

Am DSO (aka Digitales Speicher Oszilloskop) sieht das so aus:

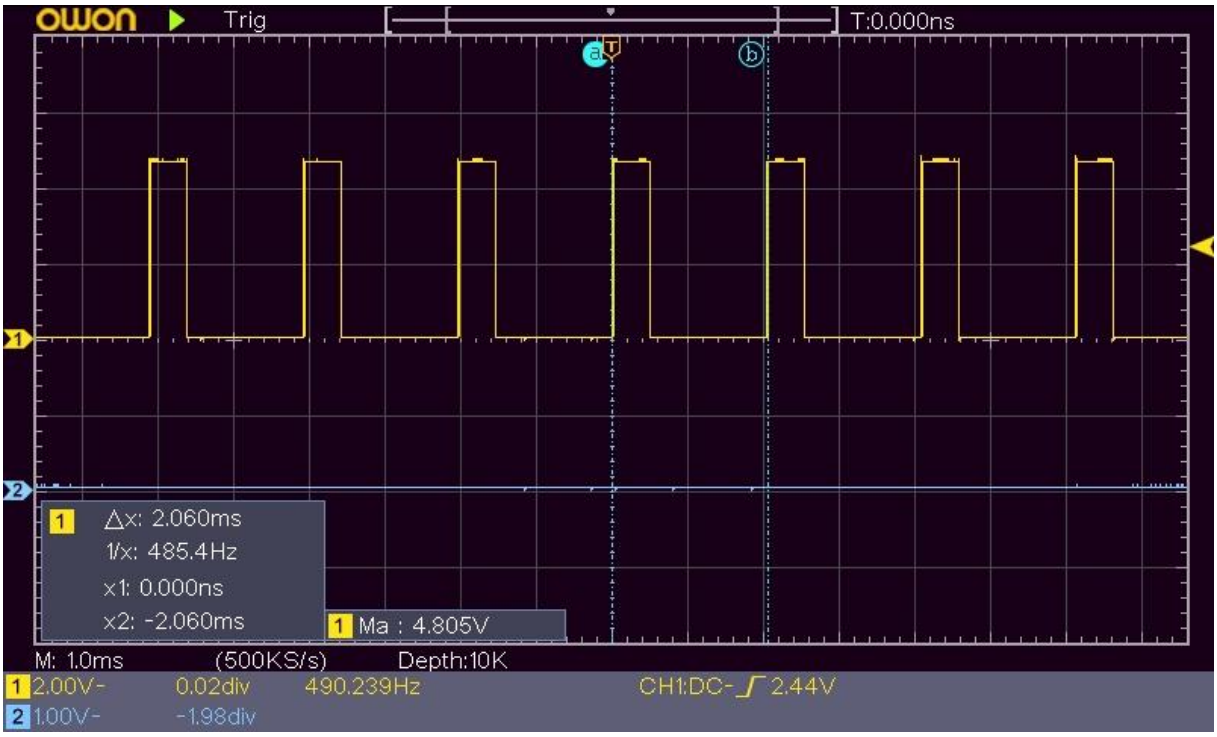


*Abbildung 9: PWM mit D=25% (63 von 255)*

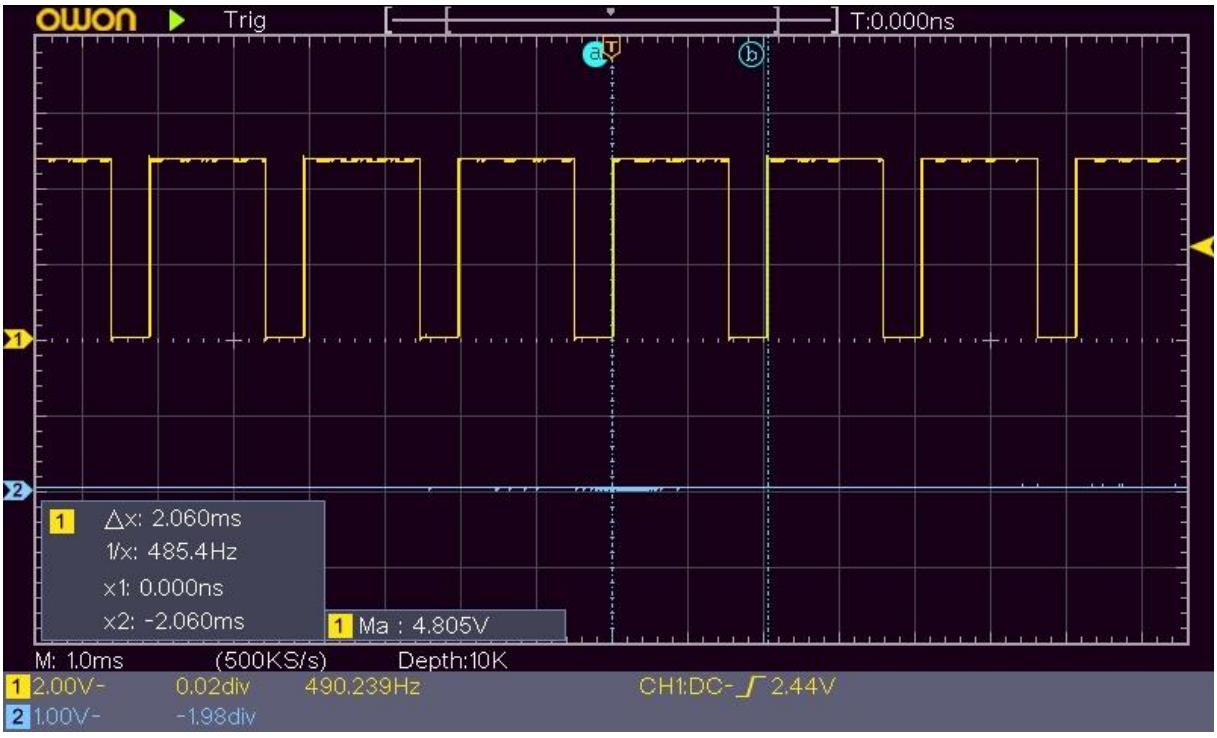


*Abbildung 10: PWM mit D=75% (191 von 255)*

Mit dem Arduino verfügen Sie jetzt, neben den anderen Features, über eine "intelligente" 16er-Tastatur, die sie über den I2C-Bus ansteuern können. Damit lässt sich auch ganz gut in MicroPython eine Input-Funktion zum Eingeben von Ziffern realisieren. Acht GPIO-Pins am ESP8266/ESP32 werden dadurch frei und dazu einiges an Speicherplatz. Ein Display wird dafür aber dringendst empfohlen. Wie so etwas zu realisieren ist, das finden Sie hier. Links zu weiteren meiner Projekte gibt es hier.

Ich wünsche viel Vergnügen bei Ihren Experimenten mit Ihrer Arduino-eierlegenden-Wollmilchsau am I2C-Bus! In meinem nächsten Beitrag werde ich den Arduino über den ESP8266-01 mit dem WLAN verkuppeln. Ein TCP-Webserver oder ein UDP-Server auf dem ESP8266-01 werden die Möglichkeit schaffen, den Arduino über einen Webbrowser oder eine Handy-App zu kontrollieren. Bis dann!