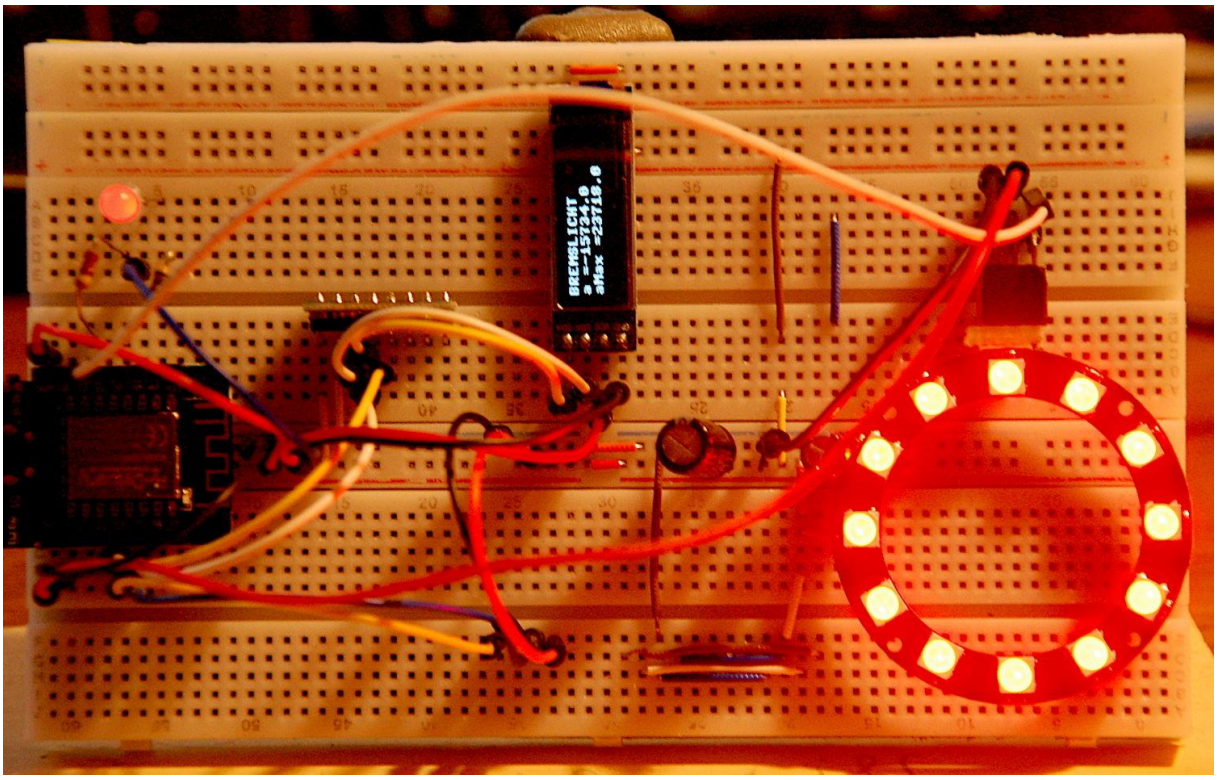

*Fahrradbremslicht - Aufbau*

Dieser Beitrag ist auch als PDF-Dokument verfügbar.

Wenn gerade kein Erdbeben in Sicht ist, was man vorab natürlich nie wissen kann, dann lässt sich unser Seismometer natürlich noch zu ganz anderen Dingen verwenden, zum Beispiel um Erschütterungen der besonderen Art zu vermeiden. Da habe ich kürzlich mit einem Bekannten eine Fahrradtour gemacht. Ich hielt unvermittelt an, um zwei Leute etwas zu fragen, da hat es auch schon gescheppert. Der hinter mir fahrende Bekannte hat zu spät registriert, dass ich angehalten hatte. Die Erschütterungen bei der Kollision waren deutlich bei beiden von uns zu spüren - körperlich und zwischenmenschlich.

Die Erkenntnis aus diesem Vorfall: mit Bremsleuchte am Fahrrad wäre das vielleicht nicht passiert. Deshalb ich mich an die Konstruktion einer Fahrradbremsleuchte gemacht. Während der Programmierung des GY-521-Treibers fiel mir dann auch noch eine zweite Anwendung zum Accelerometer ein. Wie wär's mit einem Schrittzähler und etwas pythonischer Zauberei – pythonic magic? Na dann, willkommen zu einer weiteren Folge aus der Reihe

# MicroPython auf dem ESP32 und ESP8266

heute

## Eine Bremsleuchte für das Fahrrad

Als Sensoren habe ich dieses Mal Accelerometer mit I2C-Interface genommen. Da ist zunächst einmal der LIS3DH. Er arbeitet mit maximal 12-Bit Auflösung und liefert

über den Bus Werte im Zweierkomplement-Format. Drei Auflösungen in 8-,10- und 12-Bit sind verfügbar, aber kein Filter zum Glätten der Messwerte. Das hat mich bewogen, zum GY-521-Modul mit einem MPU6050 zu greifen. Der arbeitet mit 16-Bit Auflösung und, wie der LIS3DH ebenfalls, in den Bereichen 2g, 4g, 8g und16g (Erdbeschleunigung g=9,81m/s²). Das Tiefpassfilter glättet das nervöse Zucken der ADCs in fünf programmierbaren Stufen und kann auch ausgeschaltet werden. Neben dem Drei-Achsen-Accelerometer besitzt der MPU6050 auch noch ein Drei-Achsen-Gyroskop, mit dem Winkel gemessen werden können. Dieses Feature wird hier aber nicht verwendet, wir wollen mit dem Fahrrad ja keine Saltos vollführen oder Loopings drehen.

## Hardware

Als Controller habe ich einen ESP8266 gewählt, weil der als D1 mini Ausführung kleine Abmessungen hat. Grundsätzlich wäre auch ein anderes ESP8266-Modell oder gar ein ESP32 brauchbar.

Die ersten drei ESP8266-Modelle in der Teileliste haben den Vorteil, dass am Anschluss VIN / 5V die Spannung des USB-Anschlusses verfügbar ist. Das spart während der Entwicklung ein zusätzliches Netzteil. Für den späteren Betrieb kann ein Li-Akku vom Typ 18650 in Verbindung mit einem passenden Batteriehalter die Spannungsversorgung übernehmen. Das Battery Expansion Shield liefert dann die notwendigen 5V für den Neopixel-Ring, eine Versorgung mit drei AA-Alkalizellen wurde auch erfolgreich getestet. Damit die Leuchte auch richtig viel Licht abgibt, setze ich einen Neopixel-Ring mit 12 LEDs ein.

| | |
|---|---|
| 1 | D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder<br>D1 Mini V3 NodeMCU mit ESP8266-12F oder<br>NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI oder<br>NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI |
| 1 | 0,91 Zoll OLED I2C Display 128 x 32 Pixel |
| 1 | GY-521 MPU-6050 3-Achsen-Gyroskop und Beschleunigungssensor |
| 1 | LED Ring 5V RGB WS2812B 12-Bit 50mm |
| 1 | Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set |
| 1 | Battery Expansion Shield 18650 V3 inkl. USB Kabel |
| 1 | Li-Akku oder 4,5V-Batterie |

Und hier sind Aufbau und Schaltung.

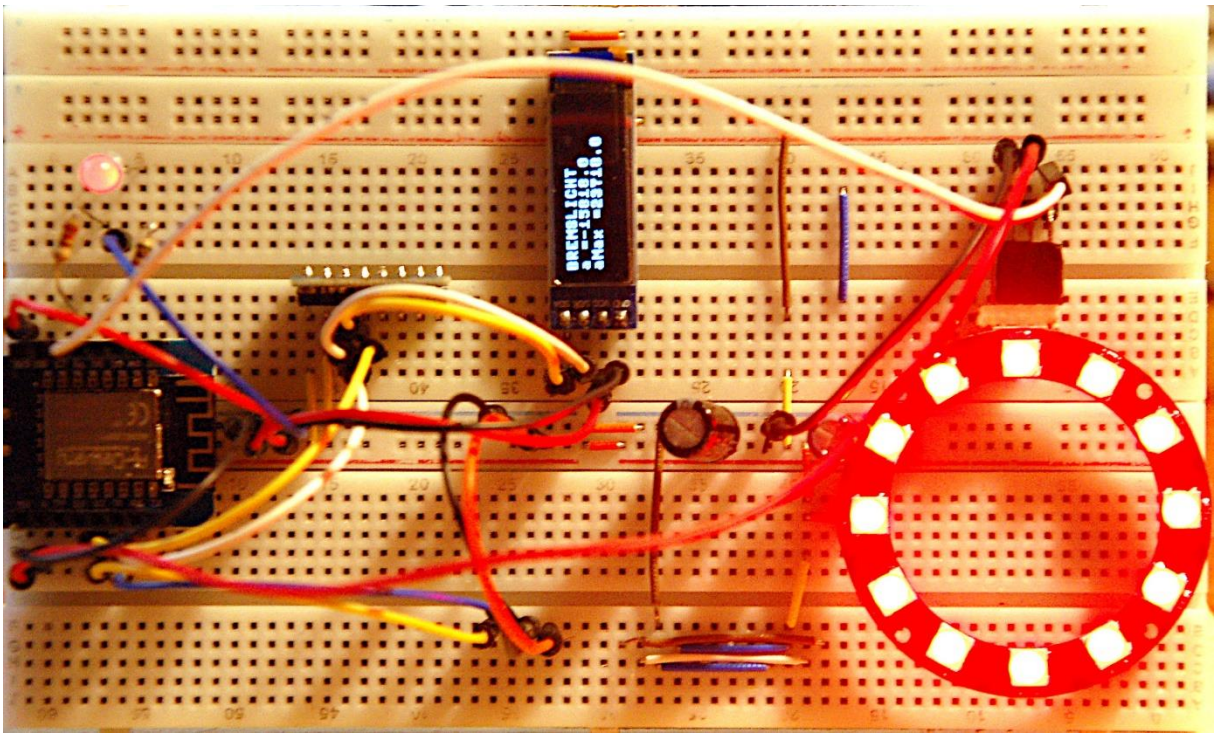

*Abbildung 1: Bremslicht - Aufbau*

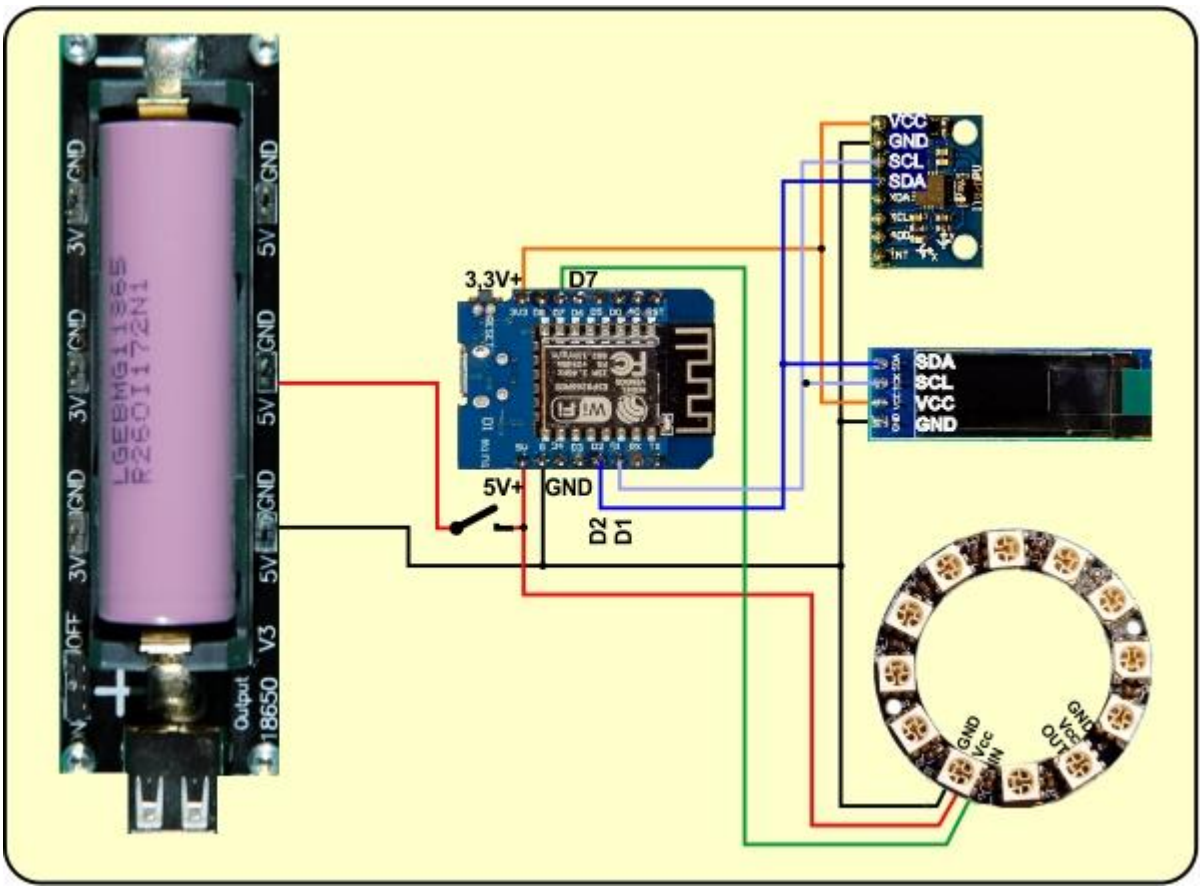


*Abbildung 2: Bremslicht - Schaltung*

# Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

## Verwendete Firmware für den ESP8266:

v1.19.1 (2022-06-18) .bin

## Verwendete Firmware für den ESP32:

v1.19.1 (2022-06-18) .bin

## Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für das OLED-Display
gy521.py Treiber für den GY-521
bremslicht.py Betriebsprogramm für das Bremslicht
stepcounter.py Anwendung Schrittzähler
ringcounter.py Programm der Nobelversion

# MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 18.06.2022) 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.

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.

## Der Treiber für das GY-521-Modul

Der Kern der Anwendung ist das Treibermodul für den GY521. Über eine ganze Reihe von Registern lässt sich dessen Verhalten steuern. Vor einigen Jahren hatte ich bereits einen Treiber in AVR-Assembler geschrieben, den ich in MicroPython umcodiert und mit dieser Version etwas erweitert habe. Grundlage war und ist das Datenblatt von INVENSENSE.

Wir starten mit einigen Importen. Für kurze Pausen nehme ich die Funktion **sleep**() aus dem Modul **time**. Längere Verzögerungen werden mit der Methode **TimeOut**() aus dem Modul **timeout** umgesetzt. **TimeOut**() liefert eine Referenz auf die Closure **compare**() als Rückgabewert. Durch Abfrage dieser Funktion realisiere ich einen nichtblockierenden Timer. Das heißt, dass der Controller während der Ablaufzeit andere Dinge erledigen kann, was beim Einsatz von **sleep**() nicht möglich ist. Wir kommen bei der Besprechung der Methode **getMeanValues**() darauf zurück. Zum einfachen Umwandeln von **bytes**-Objekten in vorzeichenbehaftete16-Bit-Ganzzahlen verwende ich die Funktion **unpack**().

```
from time import sleep
from timeout import *
from struct import unpack
```

Die Klasse **GY521_Error** wird von der Klasse Exception abgeleitet und dient als Container für die Fehlerbehandlungen in der Klasse GY521. Findet der Controller auf dem I2C-Bus keinen MPU6050, dann wird eine **Device_Error**-Exception geworfen.

Diesen Ausnahmefehler fangen wir im Hauptprogramm mit einem try-except-Konstrukt ab.

```
class GY521_Error(Exception):
    pass

class Device_Error(GY521_Error):
    def __init__(self):
        super().__init__("Device error","No MPU6050 found")
```

Es folgt eine ganze Latte von Konstanten- und Variablendeklarationen. Hier erscheinen vor allem die für uns interessanten Registeradressen. In Listen werden die Korrekturwerte für die Beschleunigungs- und Gyroskop-Werte vorgehalten. Diese Werte habe ich in Einzelmessungen ermittelt.

```
class GY521():
    AddressLow  = const(0x68) # 7-Bit default
    AddressHigh = const(0x69) # 7-Bit alternative

    Config      = const(0x1a)
    GyroConfig  = const(0x1b)
    AccelConfig = const(0x1c)
    IntPinConfig= const(0x37)
    IntEnable   = const(0x38)
    IntStatus   = const(0x3A)

    AccelXoutH  = const(0x3B)
    AccelXoutL  = const(0x3C)
    AccelYoutH  = const(0x3D)
    AccelYoutL  = const(0x3E)
    AccelZoutH  = const(0x3F)
    AccelZoutL  = const(0x40)
    TempOutH    = const(0x41)
    TempOutL    = const(0x42)
    GyroXoutH  = const(0x43)
    GyroXoutL  = const(0x44)
    GyroYoutH  = const(0x45)
    GyroYoutL  = const(0x46)
    GyroZoutH  = const(0x47)
    GyroZoutL  = const(0x48)

    SignalPathReset = const(0x68) # 0x07 to reset sig paths
    UsrCtrl     = const(0x6A)
    PwrMgmt1    = const(0x6B)
    PwrMgmt2    = const(0x6C)
    WhoAmI      = const(0x75)

    Arange2     = const(0)
    Arange4     = const(1)
    Arange8     = const(2)
    Arange16    = const(3)
```

```
    Achse=["x","y","z"]
    ATeiler=[16454,16392,16561]
    AOffset=[301,8,802]
    GTeiler=[65.536,65.536,65.536]
    GOffset=[610,175,170]
    g=9.81
```

Dem Konstruktor der Klasse GY521, der Methode **__init__**(), wird ein I2C-Bus-Objekt übergeben, das im Hauptprogramm definiert wird. Das hat den Vorteil, dass dasselbe Objekt auch anderen Modulen, hier dem OLED-Display, zur Verfügung steht. Die Standard-Hardwareadresse des MPU6050 ist 0x68. Sie wird im optionalen Parameter **addr** als Defaultwert übergeben. Durch die Attribute **self.i2c** und **self.hwadr** werden die Objekte für die Methoden der Klasse verfügbar gemacht, die auf den Bus zugreifen müssen.

```
    def __init__(self, i2c, addr=AddressLow):
        self.i2c = i2c
        self.hwadr = addr
        if not self.devPresent():
            raise Device_Error
        self.accelScale(0)
        self.gyroScale(0)
        self.ascale=0
        self.gscale=0
        self.axes(7) # enable x=4,y=2,z=1
        self.writeByteToReg(PwrMgmt1, 0x80) # mpu6050 reset
        self.writeByteToReg(PwrMgmt1, 0x00) # free running
        print("GY-521 (MPU6050) is  @",hex(addr))
```

Dann prüfen wir, ob auch wirklich ein MPU6050 auf dem Bus vorhanden ist. Das macht die Methode **devPresent**(), die zu diesem Zweck das Register **WhoAmI** auszulesen versucht. Das Byte muss laut Datenblatt den Wert **0x68** haben.

Accelerometer und Gyroskop werden auf den empfindlichsten Messbereich eingestellt, und es werden alle drei Raumachsen aktiviert. Nach einem Reset des Chips starten wir im Freilauf-Modus.

**readBytesFromReg**() liest eine beliebige Anzahl Bytes über den I2C-Bus ein, nachdem die Adresse des ersten Registers gesendet wurde. Der Defaultwert für die Anzahl an Bytes ist 1. Wird nur ein Byte geholt, muss beim Aufruf die 1 nicht angegeben werden.

```
    def readBytesFromReg(self,reg,num=1):
        buf=bytearray(num)
        try:
            buf[0]=reg
            self.i2c.writeto(self.hwadr,buf[:1])
            self.i2c.readfrom_into(self.hwadr,buf)
            return buf
        except OSError:
            return buf
```

**buf** ist ein Bytearray mit n Elementen. Es ist nötig, weil die zu senden Daten dem Buffer-Protokoll folgen müssen. Bytearrays und bytes-Objekte tun das, Zahlen, Listen etc. erfüllen die Vorgabe nicht.

**>>> i2c.writeto(0x68,0x3b)**
*Traceback (most recent call last):*
*  File "<stdin>", line 1, in <module>*
*TypeError: object with buffer protocol required*

Deshalb muss die Register-Adresse in das erste Byte des Bytearrays gesteckt werden. Anschließend senden wir nur die Scheibe mit diesem Element. Bitte beachten Sie, dass der Inhalt des ersten Elements des Arrays im folgenden Beispiel etwas anderes darstellt als die Scheibe (Slice) des Arrays mit diesem Element.

**>>> buf=bytearray(2)**
**>>> buf[0]=0x3C**
**>>> buf[1]=0xB2**
**>>> buf**
*bytearray(b'<\xb2')*
**>>> buf[0]**
*60*
**>>> buf[:1]**
*bytearray(b'<')*

Nach dem Setzen der Registeradresse lesen wir n Bytes vom GY521 in das Bytearray ein, halt so viele, wie das Array lang ist. Der Buffer wird in jedem Fall zurückgegeben. Falls ein Übertragungsfehler aufgetreten ist, sind das n Nullbytes.

Dass mehrere Bytes gelesen oder geschrieben werden können, ohne vorher immer die nächste Registeradresse zu senden, ist nicht selbstverständlich, sondern hängt vom Hersteller des Chips ab. Der MPU6050 beherrscht dieses Verfahren, das unter dem Begriff Autoinkrement läuft.

```
    def writeByteToReg(self,reg,data):
        buf=bytearray(2)
        buf[0]=reg
        buf[1]=data
        self.i2c.writeto(self.hwadr,buf)
```

Mit **writeByteToReg**() senden wir ein Byte zum MPU6050. Registernummer und Datenbyte sind Ganzzahlen und werden wieder in die Elemente eines Bytearrays gesteckt, um die Werte übertragen zu können.

**burstWrite**() kann mehrere Datenbytes mit Hilfe von Autoinkrement in einem Durchgang senden. Die Daten müssen hier allerdings bereits in Form eines Bytearrays vorliegen. Das intern erzeugte Bytearray **buf** muss für die Registeradresse um ein Element größer sein als das Array in **data**. Auch die Übergabe von **data** an **buf** gelingt mit der Notierung als Slice.

```
    def burstWrite(self,reg,data):
        buf=bytearray(len(data)+1)
        buf[0]=reg
        buf[1:]=data
        self.i2c.writeto(self.hwaddr, buf)
```

Zum Slicing will ich es nicht versäumen, Sie auf ein interessantes Verhalten von MicroPython aufmerksam zu machen. Listen und Bytearrays weisen hier Ähnlichkeiten auf.

**>>> a=[1,2,3,4,5]**
**>>> a[:3]**
*[1, 2, 3]*
**>>> a[3:]**
*[4, 5]*

**>>> b=a**
**>>> b[3]=7**
**>>> b**
*[1, 2, 3, 7, 5]*

Soweit werden unsere Erwartungen erfüllt, denn wir wissen, dass die Indizierung von Listenelementen mit der 0 beginnt und dass die Obergrenze ausgeschlossen wird. Ebenso ist bekannt, dass Listen mutabel sind, Werte in der Liste also geändert werden können. Aber jetzt wird es interessant.

**>>> a**
*[1, 2, 3, 7, 5]*

Beim Kopieren von a nach b wurde nicht nur in b aus der 4 eine 7 sondern auch in a. Das liegt daran, dass nicht wirklich eine neue Liste b angelegt wurde, sondern b nur ein Alias für a ist. Beide Bezeichner referenzieren dasselbe Objekt im Speicher. Der Vergleich mit **is** macht das ebenso deutlich, wie die Untersuchung der Identitäten von a und b.

**>>> a is b**
*True*

**>>> id(a)**
*1073685808*

**>>> id(b)**
*1073685808*

Wenn wir wirklich eine echte Kopie der Liste b herstellen wollen, dann gelingt das wieder mit Hilfe der Slice-Notierung.

**>>> c=b[:]**
**>>> c[3]=0**
**>>> c**
*[1, 2, 3, 0, 5]*
**>>> b**
*[1, 2, 3, 7, 5]*

```
    def devPresent(self):
        return self.readBytesFromReg(WhoAmI,1)[0]==0x68

    def status(self):
        return self.readBytesFromReg(IntStatus,1)[0] & 0x01
```

Wenn ein Registerinhalt eingelesen wird, geschieht das in Form eines bytes-Objektes, wie oben beschrieben. An den Inhalt des ersten Elements, den Bytewert, kommen wir durch die Indizierung mit 0. Den Zahlenwert kann man zum arithmetischen Vergleich benutzen oder ihn undieren, um einzelne Bits zu maskieren.

Die nächsten sieben Methoden arbeiten alle nach demselben Schema. Wird kein Argument beim Aufruf übergeben, dient die Methode als Getter, liest also ein Register aus und gibt den Wert zurück. Der Parameter ist optional und mit None als Defaultwert vorbelegt.

Wird eine Zahl als Argument übergeben, dann spielt die Methode einen Setter und wir prüfen den Wert auf Gültigkeit und schreiben ihn schließlich in das entsprechende Register. Damit nur ganz bestimmte Bits gesetzt werden, ohne den Wert der anderen im selben Register zu verändern, müssen wir das Register erst einmal auslesen, die entsprechenden Bits durch Undieren auf 0 setzen und danach durch Oderieren die gewünschten Positionen, gemäß dem übergebenen Argument, zu setzen. Ich erläutere das am Beispiel der Methode **cycle**().

```
    def cycle(self, state=None):
        st=self.readBytesFromReg(PwrMgmt1,1)[0] & 0x20
        if state is None:
            return st >> 5
        else:
            assert state in (0,1)
            st &= 0xDF
            st |= state << 5
            self.writeByteToReg(PwrMgmt1,st)
```

Das **CYCLE**-Bit liegt im Register **PwrMgmt1** an Bit- Position 5 und schaltet den Freilaufmodus ein (1) und aus (0). Das Register muss in jedem Fall eingelesen

werden, deshalb steht die Anweisung außerhalb des if-Konstrukts. Das CYCLE-Bit wird durch Undieren maskiert.

Wird kein Argument beim Aufruf übergeben, hat **state** den Wert None. Wir schieben das **CYCLE**-Bit um 5 Positionen nach rechts. Das zurückgegebene Byte hat jetzt den Wert 0 oder 1.

Wenn das CYCLE-Bit verändert werden soll, übergeben wir den Wert 1 oder 0 an **cycle**(). **assert** überprüft die Gültigkeit und wirft eine **AssertationError**-Exception, wenn **state** weder 0 noch 1 ist.

Durch Undieren löschen wir das Bit auf Position 5, 0xDF = 0b11011111. Den Wert in state schieben wir an die 5. Bit-Position und oderieren mit dem Wert in st bevor wir st in das Register zurückschreiben. Dieses Verfahren gelingt mit einzelnen Bits aber auch mit Bit-Gruppen wie in **axes**() oder **accelScale**()

```
    def getRawAccel(self):
        self.writeByteToReg(PwrMgmt1,0)
        accel = self.readBytesFromReg(AccelXoutH, 6)
        accel = list(unpack(">hhh",accel))
        for i in range(3):
            accel[i]=int(accel[i] - GY521.AOffset[i]/ \
                         (2**self.ascale))
        return accel
```

Zum Abholen der rohen Messwerte lesen wir die 6 Register ab der Adresse des HIGH-Bytes des x-Werts **AccelXoutH** = 0x3B ein. Jeder Achsenwert besteht aus zwei Bytes in Big-Endian-Notierung, das HIGH-Byte kommt also vor dem LOW-Byte bei uns. **accel** nimmt das **bytes**-Objekt auf, das **readBytesFromReg**()zurückgibt. Die Funktion **unpack**() aus dem Modul **struct** ist der Renner. Der Format-String ">hhh" macht aus dem **bytes**-Objekt drei vorzeichenbehaftete 16-Bit Ganzzahlen. Das ">"-Zeichen steht für Big Endianess. **unpack**() liefert ein Tupel zurück, das wir mit **list** in eine Liste verwandeln, schließlich wollen wir die Messwerte mit den Offset-Werten korrigieren, das geht bei einem Tupel nicht, Tupel sind immutabel, das heißt die Werte der Elemente können nicht verändert werden. Den Offset für die verschiedenen Messbereiche berechnen wir als Quotient aus dem Grundwert für den Bereich 2g und der Zweierpotenz des Skalencodes.

Zum Kalibrieren der Achsen starten wir später das fertige Programm, und brechen mit Strg + C ab. Wir setzen die Werte in der Liste AOffset auf 0 und messen für jede Raumrichtung einmal den Wert für die nach oben zeigende Achse, dann stellen wir den Wert fest, wenn die Achse nach unten in Richtung Erdmittelpunkt zeigt. Den Vorgang wiederholen wir für die verbleibenden Achsen.

**>>> AOffset=[0,0,0]**
**>>> %Run -c $EDITOR_CONTENT**
*this is the constructor of OLED class*
*Size:128x32*
*GY-521 (MPU6050) is  @ 0x68*
*Traceback (most recent call last):*

*File "<stdin>", line 55, in <module>*
*File "gy521.py", line 192, in getRawAccel*
*KeyboardInterrupt:*

**>>> mpu.getRawAccel()**
*[*16457*, 122, 72]*
**>>> mpu.getRawAccel()**
*[*-16361*, 524, -454]*

Oben = 16457
Unten = -16361

Offset = (16457 – 16361) / 2 = 96 / 2 = 48
Teiler = (16457 + 16361) / 2 = 32818 / 2 = 16409

Mit den berechneten Werten füttern wir die Listen **ATeiler** und **AOffset**.

```
    ATeiler=[16409,16392,16561]
    AOffset=[48,8,802]
```

In genau derselben Weise arbeitet **getRawGyro**().

Die Rohwerte sind reine Zählwerte der ADCs. Für verschiedene Zwecke reicht das, auch für die Bremsleuchte wird es genügen. Wenn man allerdings wirklich Beschleunigungen in m/s² messen möchte, muss man die Achsen kalibrieren. Das gelingt mit Hilfe der Erdbeschleunigung 9,81 m/s². Der Messbereich 2g muss dann, wenn eine der Achsen vertikal nach oben zeigt, für diese Richtung den Wert 1g =9,81m/s² liefern.

```
    def acceleration(self, val=None):
        if val is None:
            accel=self.getRawAccel()
        else:
            accel=val
        scale=self.accelScale()
        g=GY521.g
        for i in range(3):
            accel[i]=accel[i]*g/(GY521.ATeiler[i]/ \n
                      (2**self.ascale))
        return accel # m/s^2
```

Die Liste, die **getRawAccel**() liefert, übernehmen wir in die Variable accel, falls nicht ein 3-Achsenwert in Form einer Liste oder eines Tupels als Argument an den Parameter **val** übergeben wurde. Dann holen wir uns den Skalierungs-Code. Hiermit picken wir uns den Teiler aus der Liste **ATeiler**. Die Konstante für den Ortsfaktor holen wir in die lokale Variable g. Die Schreibweise der nächsten Zeile wird dadurch verkürzt. 9,81 kommt dann bei einem der Rückgabewerte heraus, wenn

accel[i]*g/(GY521.ATeiler[i] / (2**self.ascale))= 1

wird.

Beispiel:  16463 * 9,81 / (16457 / 2^0)
= 16463 * 9,81 / (16457 * 1)
= 16463 /16457 * 9.81
=  1,0003 * 9,81 = 9,81

Die anderen beiden Achsenwerte sollten einen Wert nahe 0 liefern, weil diese Achsen dann eine horizontale Ausrichtung haben. Zurückgegeben wird die Liste mit den kalibrierten Werten.

Als Dreingabe liefert uns der MPU6050 auch noch die Temperatur auf dem Chip. Den Rohwert holen wir uns wieder mit **unpack**() aus dem bytes-Paket. Im Datenblatt steht auf Seite 30, wie der Rohwert gegart werden kann. Damit man bei den Accelerometer-, Gyroskop- und Temperatur-Registern Werte vom gleichen Sample-Ereignis bekommt, müssen die Register in einem Zug gelesen werden, nicht byteweise, denn während der Bus inaktiv ist, werden die intern ermittelten Messwerte an die Register übertragen, auf die der Lesezugriff des I2C-Busses erfolgt.

```
    def getTemp(self):  # bytes: getAccel(byte: n)
        temp = self.readBytesFromReg(TempOutH, 2)
        temp=unpack(">h",temp)[0]
        temp=temp / 340.00 + 36.53
        return temp
```

Zur Kalibrierung der Accelerometer-Achsen habe ich noch drei weitere Methoden integriert, **getMin**() und **getMax**() tun das, was ihr Name schon vermuten lässt, sie ermitteln über einen Zeitraum, der in Millisekunden an den Parameter **delay** zu übergeben ist, den minimalen und maximalen Wert in der jeweiligen Achsenrichtung im Parameter **direction**. Der optionale Parameter **pause** sorgt für einen entspannten Wechsel der Achsenausrichtung, wenn die Methoden von **calibrate**() aus aufgerufen werden.

```
    def getMax(self,direction,delay,pause=5000):
        print(GY521.Achse[direction],"-Achse zeigt nach oben")
        fertig=TimeOut(pause)
        while not fertig(): pass
        print("running")
        fertig=TimeOut(delay)
        m=0
        while not fertig():
            a=self.getRawAccel()[direction]
            if a>m: m=a
        return m
```

Im Terminal wird die Achse und deren Ausrichtung angegeben, dann ist erst einmal Pause für die mechanische Umsetzung. Hier könnte man getrost auch **sleep**() anstelle der while-Schleife verwenden. "running" kündigt den Messvorgang an. Der Timer wird gestellt, fertig nimmt die Referenz auf die Closure **compare**() auf, die von **TimeOut**() zurückgegeben wird. **m** wird den maximalen oder minimalen Messwert aufnehmen und wird zu Beginn auf 0 gesetzt. Das folgende while fragt den Ablauf

des Timers ab. Während die Schleife läuft, wird ein Wert von der Achse geholt, deren code in **direction** steht, 0 für x, 1 für y und 2 für z. Ist der aktuelle Wert größer oder kleiner als der bisherige Extremwert, dann wird letzterer durch den aktuellen Wert ersetzt.

```
    def calibrate(self,duration, pause=5000):
        self.maxX=self.getMax(0,duration,pause)
        self.minX=self.getMin(0,duration,pause)
        self.maxY=self.getMax(1,duration,pause)
        self.minY=self.getMin(1,duration,pause)
        self.maxZ=self.getMax(2,duration,pause)
        self.minZ=self.getMin(2,duration,pause)
        fx=(self.maxX-self.minX)//2
        fy=(self.maxY-self.minY)//2
        fz=(self.maxZ-self.minZ)//2
        ox=(self.maxX+self.minX)//2
        oy=(self.maxY+self.minY)//2
        oz=(self.maxZ+self.minZ)//2
        print("Faktor",fx,fy,fz)
        print("Offset",ox,oy,oz)
        return (fx,fy,fz),(ox,oy,oz)
```

**calibrate**() setzt die oben angegebenen Formeln für die Bestimmung der Teiler- und Offset-Werte programmtechnisch um. Die Ausgabe der Richtungsanweisungen könnte man auch an das OLED-Display weiterleiten und die Werte anschließend in einer Datei im Flash des Controllers speichern. Von dort könnten sie beim Programmstart dann wieder eingelesen werden. Auf diese Weise würde eine mobile Neukalibrierung möglich gemacht. Wie so etwas aussehen kann, ist im Projekt zum CO2-Meter beschrieben.

## Das Hauptprogramm

Selbiges ist recht kurz, weil die Klasse GY521 die Hauptarbeit erledigt. Wenn wir noch die Kommentar- und Leerzeilen weglassen, kommen gerade mal 48 Code-Zeilen zusammen. Wir beginnen wie immer mit dem Importgeschäft. Von gy521 übernehmen wir mit dem "*" alles in den Namensraum des Hauptprogramms, also nicht nur die Klasse GY521 sondern auch die Klassen GY521_Error und Device_Error sowie die Importe.

```
from machine import SoftI2C, Pin
from oled import OLED
from gy521 import *
import sys
from neopixel import NeoPixel
import gc
```

Es folgt die controllerabhängige Deklaration des I2C-Busses. Die Variable **sys.platform** verrät den Typ.

```
if sys.platform == "esp8266":
    i2c=SoftI2C(scl=Pin(5),sda=Pin(4),freq=400000)
elif sys.platform == "esp32":
    i2c=SoftI2C(scl=Pin(22),sda=Pin(21),freq=400000)
else:
    raise RuntimeError("Unknown Port")
```

Wir deklarieren den GPIO-Ausgang für die Neopixelsteuerung und instanziieren das Neopixel-Objekt.

```
np=Pin(13,Pin.OUT)
neo=NeoPixel(np,12)

d=OLED(i2c, heightw=32)
d.clearAll()
d.contrast(255)
d.writeAt("BREMSLICHT",0,0)
```

Gleich danach wird das OLED-Display-Objekt erzeugt, das Display gelöscht, der Kontrast hochgedreht und die Überschrift ausgegeben.

Mit **try** leiten wir den Anweisungsblock für die Vorbereitungen des Hauptprogramms ein. Ist kein MPU6050 am Bus, greift der **except**-Block am Ende des Listings. Sonst definieren wir die Funktion **warn**(), die den Neopixelring ansteuert. Mit 1 als optionales Argument werden alle 12 LEDs auf hellstes rot programmiert. Eine 0 als Argument löscht alle LEDs. Erst der Aufruf von **neo.write**() schickt den ESP-internen Puffer an den Ring.

```
try:
    mpu=GY521(i2c)

    def warn(state=1):
        if state==1:
            for i in range(12):
                neo[i]=(255,0,0)
        else:
            for i in range(12):
                neo[i]=(0,0,0)
        neo.write()
```

Um die maximale Bremsbeschleunigung während einer Fahrt zu ermitteln, setzen wir den Merker **aMax** auf 0 und steigen dann in die Hauptschleife ein. In meinem Fall zeigt die z-Achse des Sensors in Fahrtrichtung. Bremsbeschleunigungen erzeugen daher negative Werte des Sensor-z-Werts. Die Anzeige der Beschleunigungswerte vermittelt ein Feeling dafür, welche Werte überhaupt auftreten können. Danach richtet sich dann der Grenzwert, ab dem die LEDs aufleuchten sollen.

```
    aMax=0
    mpu.lpFilter(3)
```

```
    while 1:
        ar=mpu.getRawAccel()
        ac=mpu.acceleration(ar)[2]
        a=ar[2] # 2, wenn z-Achse in Fahrtrichtung zeigt
        if a < -800:
            warn()
            d.writeAt("a ={}     ".format(abs(ac)),0,1,False)
            if aMax > abs(ac):
                aMax=abs(ac)
            d.writeAt("aMax ={}     ".format(aMax),0,2)
        else:
            warn(0)
```

Wir holen also zuerst die Rohwerte und übergeben sie an **acceleration**(). Mit -800 reagiert die Anzeige schon sehr empfindlich auf Bremsvorgänge. Das entspricht etwa 0,5m/s² und bedeutet, dass sich die Geschwindigkeit pro Sekunde um 0,5 m/s verringert. Bei einer Fahrt mit 20km/h legt man in einer Sekunde 5,6m zurück. Mit der angegebenen Bremsbeschleunigung beträgt in diesem Fall die Zeit bis zum Stillstand gute 11 Sekunden in denen ca. 30m zurückgelegt werden, ein ganz gemächlicher Bremsvorgang also.

Die except-Sequenz schließt das Programm ab, wird aber nur durchlaufen, wenn kein MPU6050 am Bus liegt. Der Fehler wird gezielt abgefangen.

```
except Device_Error as e:
    print("Fehler:",e)
    d.writeAt("Fehler",0,1)
    d.writeAt("Kein MPU 6050!      ",0,2)
```

Zum Testen der Anlage montiert man sie am besten erst einmal am Lenker und sucht sich einen Platz fernab von jeglichem Verkehrsgetümmel, um sich und andere durch die Bremsvorgänge nicht zu gefährden. Die Aufmerksamkeit ist jetzt wohl auch, statt auf das Geschehen um uns herum, auf die Anzeige gerichtet. Seien Sie also bitte vorsichtig!

Hier ist noch einmal ein Link zum fertigen Programm.

Das Bremslicht wird zum Schrittzähler, wenn man die Hauptschleife ein bisschen abändert. Beim Gehen bewegt sich der Oberkörper leicht auf und ab. Das hat mit Richtungsänderungen und somit auch mit Beschleunigungsvorgängen zu tun. Die vom MPU6050 gemessenen Werte pendeln um die Ruhelage bei 16000 Counts. Als Grenzwerte habe ich empirisch 18000 und 15000 ermittelt. Der Zähler wird beim Überschreiten des oberen Grenzwerts um 1 erhöht. Dann warten wir, bis der Messwert unter 15000 gefallen ist.

```
cnt=0
aMax=0
while 1:
    a=mpu.getRawAccel()[0]
    d.writeAt("a= {}".format(a),0,2)
```

```
    if a > 18000:
        cnt+=1
        warn()
        d.writeAt("cnt = {}     ".format(cnt),0,1)
        while a >= 15000:
            a=mpu.getRawAccel()[0]
            pass
    elif a < 15000:
        warn(0)

    if taste.value()==0:
        sys.exit()
```

Auch den Schrittzähler gibt es fertig zum Download.

Jedes der beiden Programme muss für den autonomen Betrieb ohne USB-Kabel und PC natürlich in den Flash-Speicher des Controllers hochgeladen werden. Wählen Sie das entsprechende Editorfenster aus und drücken Sie **Shift+Strg+S**. Im folgenden Dialog wählen Sie **MicroPython Device** und vergeben den Namen **main.py**. Dann können Sie den Aufbau vom USB-Bus abkoppeln. Mit der RST-Taste startet das Programm, sobald der Aufbau von einer Batterie versorgt wird.

Das Programm kann man aber auch noch kannibalisch aufmuffen, indem man den Neopixelring bei jedem Schritt mit hochzählen lässt. Let's do magic now!

Die Funktion **warn**() ersetzen wir durch die Closure **ring**(). Closure deshalb, weil wir damit einen autonomen Ringzähler Modulo 13 erhalten, ohne dass sich die Mainloop auch noch darum kümmern muss. Wie funktioniert das?

```
def counter(n):
    def count():
        nonlocal n
        n+=1
        n%=13
        if n==12: # alle aus
            for i in range(12):
                neo[i]=(0,0,0)
        else:
            for i in range(n+1):
                neo[i]=(64,0,0)
        neo.write()
    print(id(count))
    return count
```

An **counter**() übergeben wir den um 1 verringerten Startwert der Zählung. Der Parameter n wird in der eingeschlossenen Funktion als nichtlokal deklariert: **nonlocal n**. Damit kann die in **counter**() deklarierte Funktion **count**() den Wert des für sie externen Variablen n ändern. Wir erhöhen den Zählerstand und ermitteln den ganzzahligen Teilungsrest Modulo 13. Der Zähler läuft also von 0 bis 12.

Weil der für **count**() externe Parameter **n** innerhalb der Funktion als nichtlokale Variable referenziert wird, wird die Funktion **count**() zu einer Closure. Die innere Funktion kapselt die für sie nichtlokale, externe Variable in ihrem Scope, schließt sie ein. Das erklärt den Namen Closure, von close schließen, einschließen. Dadurch wird es möglich, dass nach dem Beenden der umgebenden Funktion noch auf n zugegriffen werden kann. Das geschieht über die Referenz auf **count**(), die von **counter**() zurückgegeben wird. Diese Referenz weisen wir der Variablen **ring** zu. Wie Sie sehen, referenzieren **ring** und **count** dasselbe Objekt, die IDs sind identisch.

```
ring = counter(-1)
```

**>>> ring=counter(-1)**
*1073688704*
**>>> ring**
*<closure>*
**>>> id(ring)**
*1073688704*
**>>> ring()**
*0*
**>>> ring()**
*1*

Mit jedem Aufruf von **ring**() wird von 0 aus hochgezählt bis 12. Mit Erreichen dieses Werts werden die LEDs alle gelöscht und die Zählung beginnt wieder bei 0. Am Programm müssen jetzt nur noch eine Zeile ergänzt, eine geändert und zwei gelöscht werden.

```
ring=counter(11)
cnt=0
aMax=0
while 1:
    a=mpu.getRawAccel()[0]
    d.writeAt("a= {}".format(a),0,2)
    if a > 18000:
        cnt+=1
        ring()
        d.writeAt("cnt = {}     ".format(cnt),0,1)
        while a >= 15000:
            a=mpu.getRawAccel()[0]

    if taste.value()==0:
        sys.exit()
```

Hier geht es zum fertigen Programm ringcounter.py.