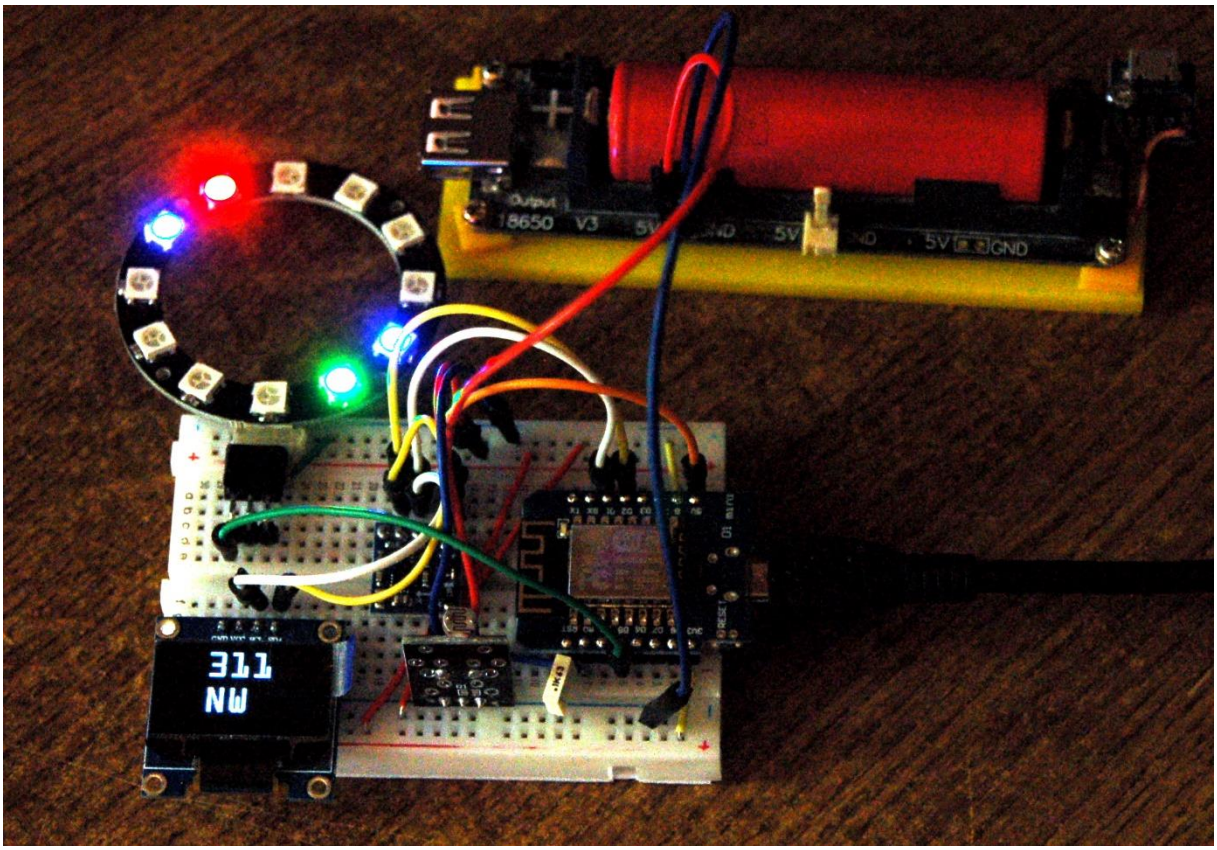

Kompass in Aktion

Diese Anleitung gibt es auch als PDF-Dokument.

Sommer - raus ins Freie! Regentage gibt es aber dennoch hin und wieder. Da könnte man doch das Eine mit dem Anderen verbinden. Ein schnelles Projekt für drinnen und die Nutzung desselben im Freien ist ideal.

Vor ziemlich genau einem Jahr hatte ich mich mit dem GPS-Modul SIM800 beschäftigt. Dazu gibt eine Reihe von Blogbeiträgen, zum Beispiel diesen hier. Ich hatte damals bereits vor, das System durch eine Kompassanwendung zu ergänzen. Genau das soll nun mit diesem Beitrag geschehen. Welche Risiken und Nebenwirkungen dabei auftraten, das verrate ich Ihnen in der aktuellen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der ESP8266-Kompass

Zum Einsatz kommen neben einem ESP8266 D1 mini, den ich wegen seiner geringen Abmessungen gewählt habe, das Magnetometermodul GY-271 mit einem DE5883L als wichtigstem Bauteil, ein OLED-Display, ein LDR-Modul und ein Neopixel-Ring. Als Spannungsversorgung kann zur Not eine Batterieeinheit mit drei AA-Zellen oder besser ein Batteriehalter mit einem 16850-er Li-Akku dienen.

Damit sind wir auch schon bei der Liste mit der nötigen Hardware.

Hardware

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder D1 Mini V3 NodeMCU mit ESP8266-12F WLAN Modul
1	LED Ring 5V RGB WS2812B 12-Bit 37mm oder ähnlich
1	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
1	KY-004 Taster Modul
1	GY-271 Kompassmodul Kompass Magnet Sensor
1	KY-018 Foto LDR Widerstand Diode Photo Resistor Sensor
1	Battery Expansion Shield 18650 V3 inkl. USB Kabel
diverse	Jumperkabel
1	Minibreadboard oder Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins
optional	Logic Analyzer

Die ESP8266-er haben eine unliebsame Eigenschaft, die man aber mit bestimmten Maßnahmen abstellen kann – sie starten alle aus scheinbar unersichtlichen Gründen sporadisch neu durch. Das hat zumindest zwei Gründe. Der hardwarebedingte Grund ist eine gewisse Empfindlichkeit gegen zu niedrige und/oder unsaubere Betriebsspannung. Spannungsschwankungen auf dem USB-Bus können schon ausreichen, dass der Kleine im unregelmäßigen Rhythmus von einigen Sekunden neu bootet. Ein eigenes Netzteil oder eine Batteriebox mit mindestens drei, besser 4 AA-Zellen schafft schon Abhilfe. Hervorgerufen werden solche Neustarts wohl vom AP-Interface, das mit jedem Neustart von der Firmware mit hochgefahren wird, ganz egal, ob es gebraucht wird oder nicht. Durch die höhere Stromaufnahme beim Aktivieren des AP-Interfaces bricht die Spannung kurz ein und löst einen Neustart aus. Schaltet man das Interface aus, kehrt Frieden ein.

>>> import network; network.WLAN(network.AP_IF).active(False)

Ein Kondensator von 0,1µF vom **EN-Pin** auf GND und ein Pullupwiderstand von 4,7KΩ gegen Vcc=3,3V wirken manchmal auch bereits Wunder, ebenfalls wie ein Elektrolytkondensator von 100µF an der Versorgungsspannung.

Der Neopixel-Ring dient als Anzeige der Nord-Süd-Richtung auch bei Dunkelheit. Die Helligkeit der LEDs wird über den LDR (Light-Dependent-Resistor = Fotowiderstand) an die Umgebungshelligkeit angepasst. Auf dem OLED-Display kann man den Peilwinkel auf Grad genau und die Himmelsrichtung im 22,5-Grad-Raster ablesen.

Als Magnetsensor dient ein DE5883L = QMC5883L, der nicht mit dem HMC5883L verwechselt werden darf. Neben abweichenden physikalischen Eigenschaften hat der DE5883L eine komplett andere Registeranordnung wie der HMC5883L. Auch die Hardwareadressen der Bausteine sind unterschiedlich, für den QMC5883L ist es die 7-Bit-Adresse 0x0D. Für den Gebrauch als Kompass muss der Sensor unbedingt kalibriert werden, um die Sensitivität der x- und y-Achse auf gleiches Niveau und

gleiche Nulllinie zu bringen. Erst dann ist eine Umrechnung der Magnetdaten in Winkel möglich.

Die Schaltung ist in Abbildung 1 dargestellt. Der Taster an D3=GPIO0 kann beim Start gedrückt werden, um eine neue Calibrierung des QMC5883L einzuleiten.

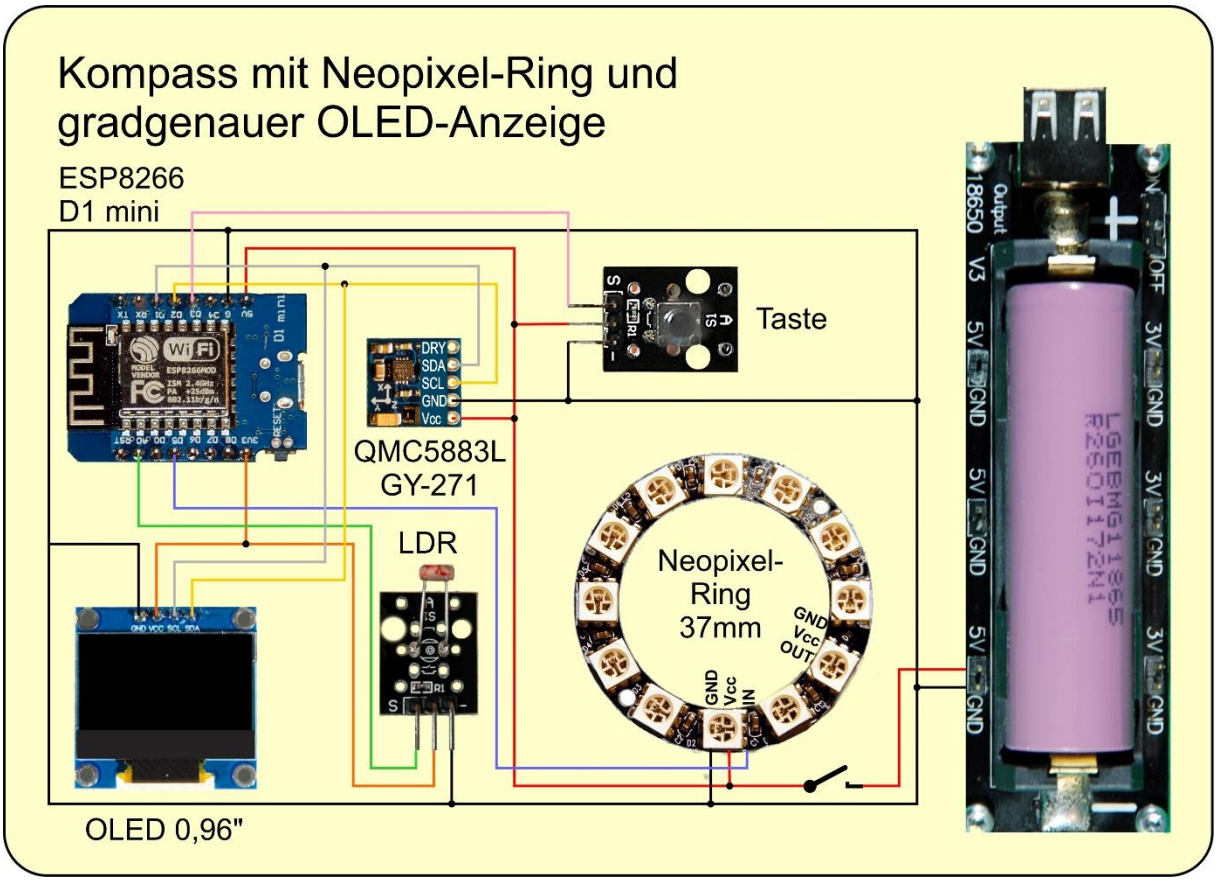

Abbildung 1: Kompass - Schaltung

Damit sind wir auch schon bei der Software für das Projekt.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP8266/ESP32:

MicropythonFirmware
Bitte eine Stable-Version aussuchen
ESP8266 mit 1MB Version 1.18 Stand: 25.03.2022 oder später

Die MicroPython-Programme zum Projekt:

ssd1306.py: Anzeigetreiber für das OLED-Modul
oled.py: normaler Anzeigetreiber für das OLED-Modul
micropython-font-to-py: Paket von Peter Hinch zum Clonen von Windowszeichensätzen darin enthalten ist writer.py, der Treiber für die erzeugten Zeichensätze. Aus dieser Datei habe ich die nicht benötigten Partien entfernt, damit alles in den ESP8266 passt. Daher sollten Sie besser die folgende Datei verwenden.
writer.py: abgespeckter Treiber für die MicroPython-Zeichensätze
ocr20.py: ein Zeichensatz mit 20 Pixel Zeichenhöhe
Datenblatt des QMC5883L
test_calibration.py: Programm zum Aufnehmen der Kalibrierungskurven
kompass.py: Betriebsprogramm zum Kompass.

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 05.02.2022) 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.

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.

Kontaktaufnahme mit (Micro)-Python

Wenn sie bereits mit der Arduino-IDE gearbeitet haben, hatten sie vielleicht hin und wieder das Verlangen einen Befehl auszuprobieren, um dessen Verhalten zu untersuchen. Aber dort ist eine interaktive Arbeit mit dem System nicht vorgesehen, ich habe das weiter oben schon erwähnt.

MicroPython als Abkömmling der Interpretersprache C-Python verhält sich da anders. Sie arbeiten in einer IDE, einer integrierten Entwicklungs- Umgebung, mit direktem Draht zu einem Python-Interpreter. In Python nennt sich dieses Tool **REPL**. Das ist ein Akronym für die Zustände **Read – Evaluate - Print – Loop**. Sie machen am Terminal eine Eingabe, der MicroPython-Interpreter wertet die Eingabe aus, gibt eine Antwort zurück und wartet auf die nächste Eingabe. Die Eingabeaufforderung von Windows, früher MS-DOS, arbeitet nach dem gleichen Schema.

Es gibt eine ganze Reihe von IDEs, Thonny, µPyCraft sind zwei davon, Idle und MU sind zwei weitere. Ich arbeite gerne mit Thonny, weil es außer dem Terminal (REPL) und einem komfortablen Editor zum Erstellen und Testen von Programmen auch noch über 12 Assistenten, einen Plotter und ein Tool zum Flashen der Firmware verfügt. Mit REPL ist es ein Leichtes, jede Anweisung einzeln zu testen. Wir werden das an verschiedenen Stellen in dieser Anleitung ausprobieren. **Eingaben** am REPL-Prompt >>> formatiere ich im Folgenden **fett**, die *Antworten* vom System *kursiv*.

Noch etwas ist bei Python anders als bei C oder anderen Compiler-Sprachen. Es gibt keine Typfestlegung beim Deklarieren von Variablen oder Funktionen. MicroPython findet den Typ selbst heraus.

Ein paar Infos zum I2C-Bus

I2C, eigentlich I²C, also "I quadrat C" kommt von IIC und das ist ein Akronym für Inter-Integrated-Circuit. Damit wird ein Bus mit zwei Leitungen, SDA und SCL, beschrieben, der von Philips entwickelt wurde, um die Kommunikation von Einheiten in einem Gerät zu ermöglichen. Genau zu diesem Zweck werden wir diese Busverbindung heute verwenden. Der ESP8266 soll sich mit dem OLED-Display und dem Magnetometer-Modul unterhalten können. Die schaltungstechnischen Bedingungen für den Betrieb sind durch das Display und das GY271-Modul bereits erfüllt. Es geht um je einen Pullup-Widerstand, der die SCL und die SDA-Leitung gegen Vcc=3,3V zieht. Der HIGH-Pegel ist auch der Leerlauf-Zustand.

Jede Einheit am Bus kann durch ihre Ausgangsstufe den Pegel der Leitung auf GND-Potenzial ziehen. Durch die verschiedenen Zustände auf den Leitungen und deren zeitliche Abfolge lassen sich Signale zusammensetzen, die für die Steuerung der Übertragung wichtig sind: ST = start-condition (Sendungsbeginn), SP = stop-condition (Sendungsende), ACK = acknowledge (Daten angenommen) und NACK = not acknowledge, (Daten abgewiesen, Fehlerzustand).

Der Chef in unserem Projekt ist natürlich der ESP8266. Er gibt als Master den Takt der Übertragung an, 400.000Hz = 400 kHz. In 2,5 µs wird also ein Bit übertragen. In der Regel ist es so, dass mit einer fallenden Flanke auf der Taktleitung SCL (serial clock) das Datenbit, 0 oder 1, auf die Datenleitung SDA (serial Data) geschaltet wird, LOW oder HIGH, 0V oder 3,3V. Zum Zeitpunkt der steigenden Flanke auf SCL wird der Zustand auf SDA als Datenbit vom Empfänger eingelesen. Mit Hilfe eines **Logic Analyzers** kann man die Signale sichtbar machen. Das kann bei der Fehlersuche enorm hilfreich sein. Eine Beschreibung zu dem Gerät finden Sie übrigens in dem Blogpost " Logic Analyzer - Teil 1: I2C-Signale sichtbar machen" von Bernd Albrecht. Dort ist auch beschrieben, wie man den I2C-Bus abtastet. Das kleine Ding wird an den USB-Bus angeschlossen und zeigt mittels einer kostenlosen Software, was auf den Busleitungen los ist. Dort, wo es nicht auf die Form von Impulsen ankommt, sondern lediglich auf deren zeitliche Abfolge ist ein LA Gold wert. Und, während ein DSO (digitale Speicher-Oszilloskop) nur Momentaufnahmen des Kurvenverlaufs liefert, kann man mit dem LA über längere Zeit abtasten und sich dann in die interessanten Stellen hineinzoomen. Das ist in Abbildung 2 für den ersten Teil der Initialsierungssequenz des QMC5883L dargestellt. Die Übertragung beginnt mit der start-condition – SDA wird vom Master auf LOW gezogen, während SCL HIGH ist. Als erstes Byte muss die Hardware-Adresse des Peripherie-Bausteins ausgegeben werden, das ist der 7-Bit-Wert 0x0D. Das zusätzliche LSB (W) ist in diesem Fall 0, weil es sich um einen nachfolgenden Schreibbefehl handelt. Bei einem Lesezugriff wäre das 8. Bit der Hardware-Adresse eine 1. Jedes Bit wird durch einen Punkt gekennzeichnet.

Das 9. Bit ist das ACK-Bit, mit dem der QMC5883L anzeigt, dass er die Adresse erkannt hat. Deshalb zieht der QMC5883L die Datenleitung jetzt kurz auf 0. Im Falle eines NACK bliebe SDA auf 1. Auf die Hardware-Adresse folgt die Nummer des zu beschreibenden Registers, 0x09. Direkt anschließend wird das Datenbyte übertragen, 0x0D. Eine stop-condition schließt die Übertragung ab.

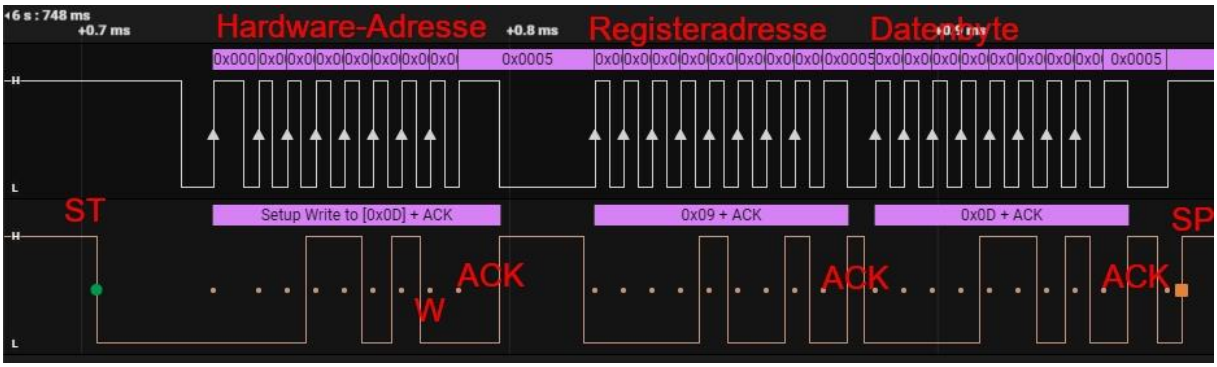

Abbildung 2: Initialisierung des QMC5883L (Ausschnitt)

Wenn die Schaltung schon aufgebaut ist, können wir den ersten Test fahren.

```
>>> from machine import Pin, SoftI2C
>>> SCL=Pin(5)
>>> SDA=Pin(4)
>>> i2c=SoftI2C(SCL,SDA,freq=400000)

>>> i2c.scan()
[13, 60]
```

Der scan-Befehl sagt uns, dass sich auf dem Bus zwei I2C-Geräte befinden. Die Adresse 13=0x0D kennen wir schon, sie gehört dem QMC5883L. Die zweite Adresse 60=0x3C spricht das OLED-Display an.

In MicroPython geht das Testen eben mal kurz direkt über REPL. In der Arduino-IDE müsste ich die paar Zeilen in ein Programm schreiben, lang und breit compilieren und als komplette Firmware zum ESP8266 schicken, um schließlich dieselbe Information zu bekommen. Zeitaufwand zu Nutzen, wenigstens 100 : 1, bei MicroPython habe ich 1:1 mit Sofortwirkung.

Das Magnetometer – QMC5883L

Bevor wir zur Besprechung des Programms kommen, einige Anmerkungen zum Magnetometer-Modul. Wie weiter oben schon erwähnt, gibt es zwei ähnliche ICs, die aber gänzlich unterschiedliche Registernummern und auch abweichende technische Eigenschaften haben. Aus dem Datenblatt von qst habe ich die Informationen für meine Klasse QMC5883L entnommen. Das passt auch zu dem Chip DB5883L, der auf dem Modul GY-271 verbaut ist.

Das Modul verfügt über einen eigenen 3,3V-Regler und außerdem über einen Pegelwandler für die SCL und die SDA-Leitung. Auch die nötigen Pullup-Widerstände sind bereits enthalten. Das Modul ist somit an 5V Betriebsspannung anschließbar und ohne Pegelprobleme mit dem ESP8266 zu verbinden.

Die Anzahl der Register ist überschaubar. Hier die Tabelle 13 von Seite 16 des Datenblatts.

Addr.	7	6	5	4	3	2	1	0	Access
00H	Data Output X LSB Register XOUT[7:0]								Read only
01H	Data Output X MSB Register XOUT[15:8]								Read only
02H	Data Output Y LSB Register YOUT[7:0]								Read only
03H	Data Output Y MSB Register YOUT[15:8]								Read only
04H	Data Output Z LSB Register ZOUT[7:0]								Read only
05H	Data Output Z MSB Register ZOUT[15:8]								Read only
06H						DOR	OVL	DRDY	Read only
07H	TOUT[7:0]								Read only
08H	TOUT[15:8]								Read only
09H	OSR[1:0]		RNG[1:0]		ODR[1:0]		MODE[1:0]		Read/Write
0AH	SOFT_RST	ROL_PNT						INT_ENB	R/W, Read only on blanks
0BH	SET/RESET Period FBR [7:0]								Read/Write
0CH	Reserved								Read only
0DH	Reserved								Read only

Abbildung 3: Registerübersicht des QMC5883L:

Die Datenregister für die drei Raumachsen stehen beim QMC5883L am Anfang der Liste. Die Magnetometerdaten werden als vorzeichenbehaftete 16-Bit-Werte abgelegt. Dementsprechend sind sie einzulesen und zu verarbeiten. Wir benötigen eigentlich nur die x- und y-Richtung. Die Methode **readAxes**() aus der Klasse QMC5883L liest dennoch alle Achsenregister, weil es das Datenblatt so vorschreibt. Dazu komme ich später.

Zwei weitere Datenregister erlauben das Auslesen der Temperatur des Chips und bei entsprechender Zirkulation auch der Umgebung. Laut Datenblatt, Kapitel 9.2.3, Seite 17, ist die relative Genauigkeit mit 100LSB/°C angegeben, das entspricht einer Genauigkeit von 1/100°C. Soll die Temperaturangabe den Umgebungswerten entsprechen, dann muss der Nullpunkt der Skala für jeden Chip separat kalibriert werden. Die Methode **readTemperature**() demonstriert das. Auch dazu später mehr.

Das Register 0x06 kann als Statusregister verstanden werden. Es spiegelt den Zustand der Messeinheit wider. Das Bit DRDY interessiert uns im Zusammenhang mit dem Projekt am meisten, denn es informiert den ESP8266 darüber, dass ein neuer Messwert vorliegt. Es wird vom QMC5883L gesetzt, wenn die Daten aller drei Achsen aus der Messeinheit in das I2C-Interface übertragen wurden. Es wird gelöscht, wenn alle sechs Achsen-Register gelesen wurden. Damit dieser Handshake funktioniert, ist es also wichtig, alle Achsenregister auszulesen und vor dem erneuten Lesen das Bit DRDY zu testen. Das macht die Methode **dataReady**().

Die Bits **OVL** und **DOR** treten in unserem Projekt nicht maßgeblich in Erscheinung.

Die Register 0x09 und 0x0A regeln den Ablauf der Messungen. Die Bits im Controll-Register2 (0x0A) treten im Projekt nicht in Erscheinung. Das Controll-Register1 wird bei der Initialisierung auf den Wert 0x0D gesetzt. Mit einem Oversampling von 512 wird das Rauschverhalten auf ein Minimum reduziert. Für das Erdmagnetfeld sind 2G = 2 Gauss) angemessen. Seit 1970 ist das Gauss in der EU keine gesetzliche Einheit mehr. Sie wurde abgelöst durch die Einheit Tesla (T). 1 Gauss = 1G entspricht der magnetischen Flussdichte von 100µT. Die Flussdichte in horizontaler Richtung beträgt in Deutschland ca. 20 µT, was 0,2G entspricht. Die Ausbeute des Sensorsignals beträgt daher ca. 6400LSB des Sensors als Richtwert.

Die Datenausgaberate wird mit 200 pro Sekunde eingestellt, der Betrieb erfolgt im Dauerlauf. Zur Glättung der Werte wird ein Mittelwert von 100 Einzelmessungen ermittelt, **axesAverage**().

Damit hernach alles korrekt funktioniert muss das Register 0x0B mit einem Wert von 0x01 beschrieben werden. Das Datenblatt gibt dazu leider keine weiteren Hinweise.

Beim ersten Start des Programms wird der Sensor kalibriert. Das OLED-Display informiert darüber, dass die Kalibrierung gestartet wurde und wann sie endet. Während des Vorgangs muss der Aufbau mit dem Sensor mindestens einmal in horizontaler Lage um die z-Achse um mindestens 360 Grad gedreht werden. Als Dauer sind im Programm dafür 20 Sekunden vorgesehen. Zu schnelles Drehen führt zu ungenauen Ergebnissen. Die Kalibrierungsdaten werden automatisch in die Datei **config.txt** auf dem ESP8266 geschrieben und bei den nächsten Starts eingelesen.

Zur Überprüfung dieser Werte kann ein anschließender Test mit Hilfe des Winkelnetzes durchgeführt werden. Das Winkelnetz wird dafür zunächst unter Verwendung eines normalen Magnet-Kompasses genau nach Norden ausgerichtet und auf dem Tisch fixiert. Es sollten sich keine Eisenteile in der unmittelbaren Umgebung befinden. denken Sie auch an Schrauben, Nägel, PC-Gehäuse, etc. im, auf und unter dem Tisch.

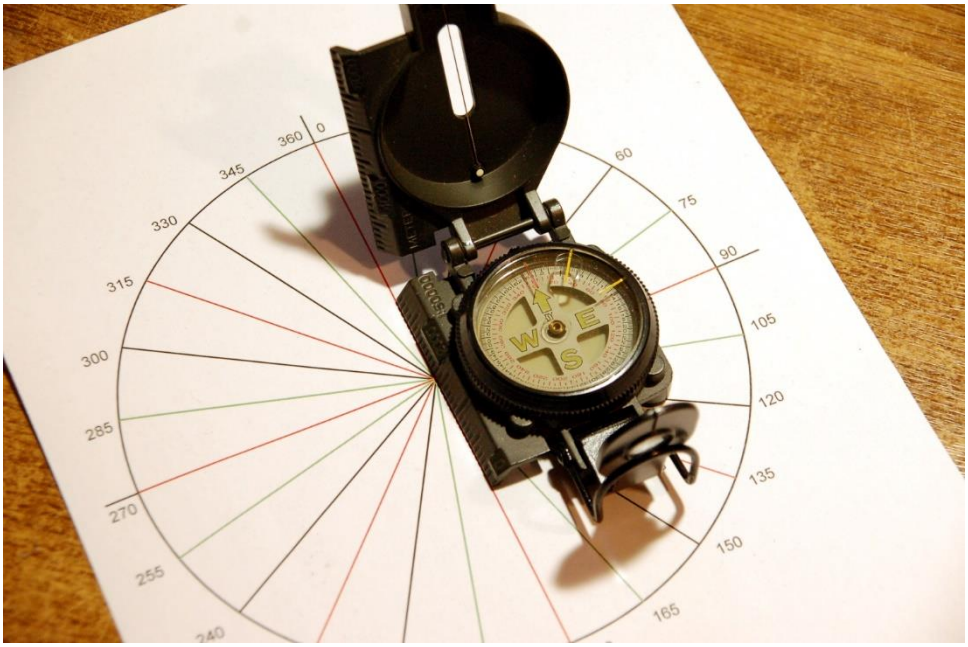

Abbildung 4: Messblatt einnorden

Starten Sie nun einmal das Programm test_calbration.py aus einem Editorfenster mit F5.

```
>>> %Run -c $EDITOR_CONTENT
this is the constructor of OLED class
Size:128x64
QMC5883L is @ 13
```

Damit sind die Methoden der Klasse QMC5883L auf REPL verfügbar und können zur Aufnahme der Kalibriertabelle verwendet werden. Diese Tabelle erstellen wir nun Schritt für Schritt mit unserem Testaufbau.

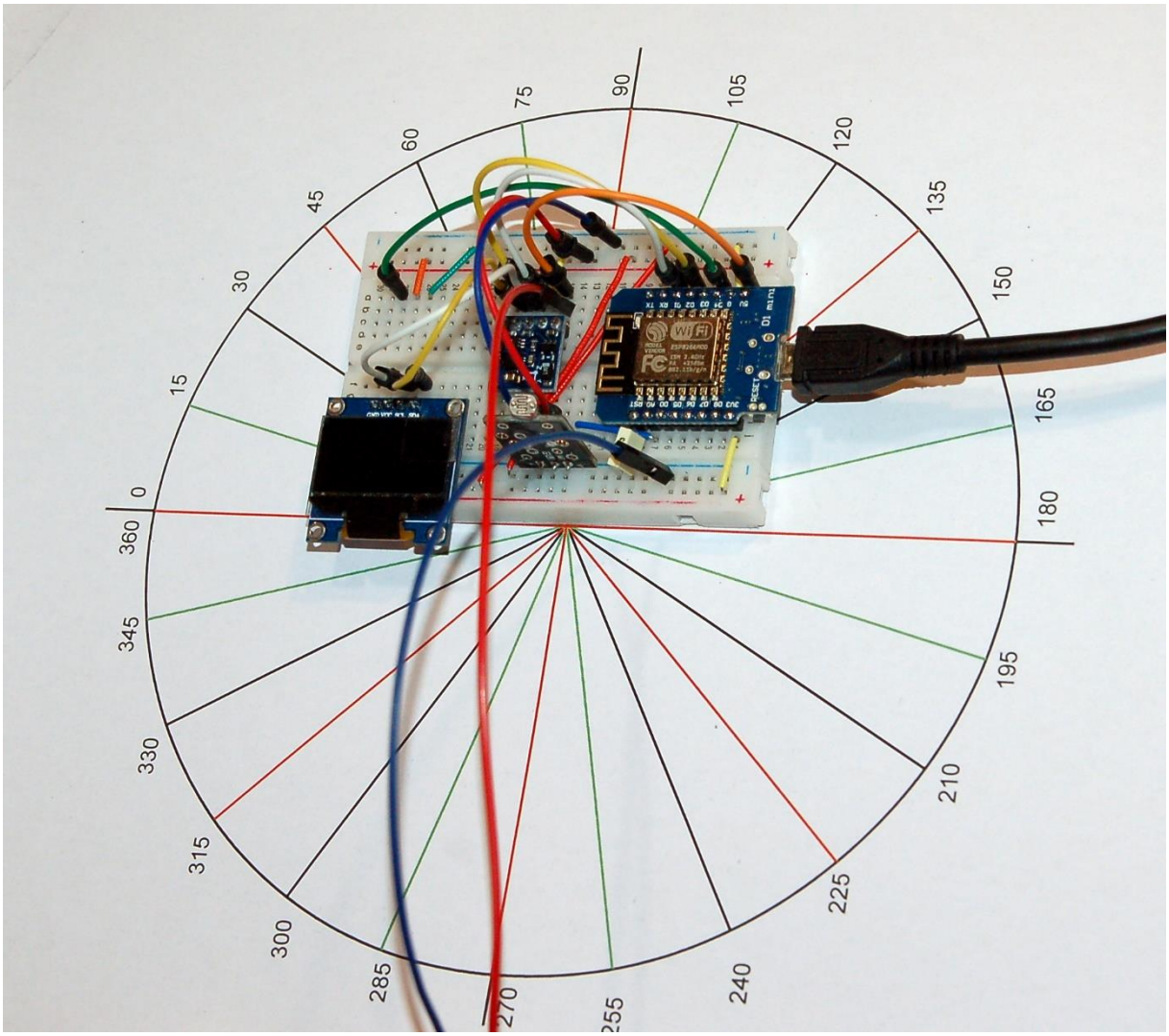

Abbildung 5: Messkurven aufnehmen:

So wie zuvor der Kompass wird das Breadboard an die 0°-Linie angelegt. Die x-Achsenrichtung auf dem GY-271-Modul zeigt dann auch in diese Richtung. Wir rufen die Methode **axesAverage**(100) mit dem Argument 100 auf. Es werden 100 Einzelmessungen durchgeführt, die Ergebnisse gemittelt und der x- und y-Wert ausgegeben. Diese Werte notieren wir zum Winkel 0°.

>>> k.axesAverage(100)
(1091, 68)

Der Prozess wird für jede Markierung des Gradnetzes wiederholt. Zum Schluss tragen wir alle Werte in eine Tabelle eines Kalkulationsblattes ein. Ich habe das in Libre Office getan und daraus ein Diagramm abgeleitet. Die Kalibrierung ist dann gut gelaufen, wenn die beiden Kurven dieselbe Amplitude aufweisen, symmetrisch zur Rechtswertachse liegen und nach Möglichkeit eine vollständige, glatte sin- und cos-Kurve ergeben. In Abbildung 6 muss also noch etwas am y-Offset nachgebessert werden.

Winkel	x	y
0	1091	68
15	1080	291
30	1000	508
45	885	693
60	709	862
75	444	1000
90	230	1050
105	-103	1054
120	-350	970
135	-573	800
150	-730	580
165	-857	300
180	-883	0
195	-826	-290
210	-680	-550
225	-480	-750
240	-220	-870
255	10	-935
270	305	-930
285	550	-829
300	760	-690
315	890	-494
330	990	-308
345	1055	-85
360	1060	135

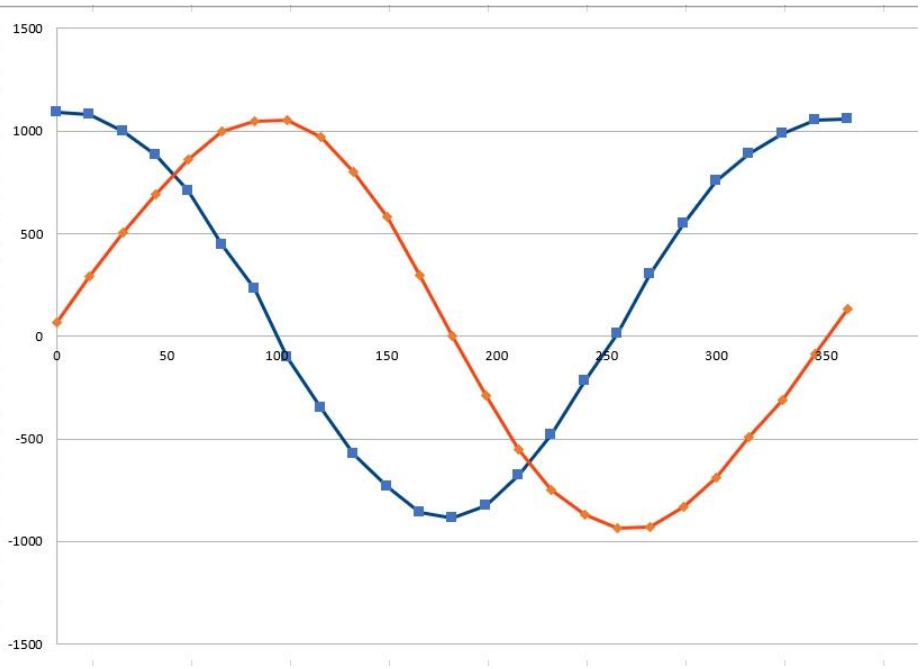

Abbildung 6: Kalibrierchart

Große TTF-Zeichensätze für das OLED-Display

Die Sache mit der Kalibrierung kann man gut mit der normalen 8-Pixel-hohen Standardschrift des OLED-Displays begleiten. Für die Ablesung des Peilwinkels und der groben Himmelsrichtung wäre dann aber schon ein größerer Zeichensatz angebracht. Und das geht unter Einsatz des Pakets micropython-font-to-py von Peter Hinch. Damit lassen sich TTF-Zeichensätze aus Windows in pixel-orientierte Zeichensätze für MicroPython clonen. Laden Sie sich das Paket von GitHub in ein beliebiges Verzeichnis herunter (hier **F:\font2py**) und entpacken Sie es in ein Unterverzeichnis, hier **f2p**.

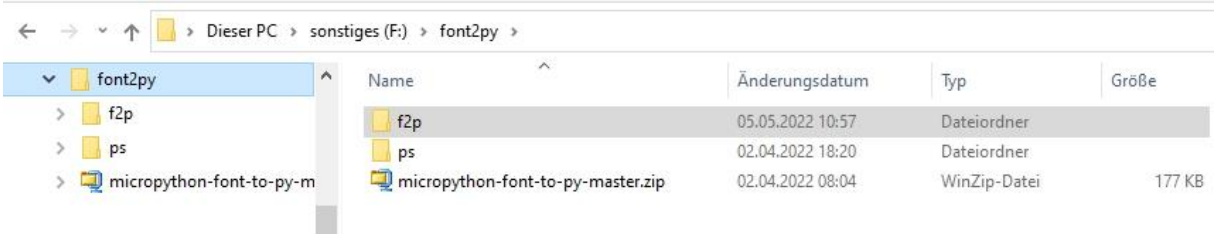

Abbildung 7: Installation des Font_to_py-Pakets

Im Verzeichnis **f2p** habe ich ein Unterverzeichnis **quellen** (roter Punkt) angelegt, in dem ich die umzuwandelnden TTF-Fonts ablege. Aus dem Windows-Fonts-Verzeichnis kopiere ich die Datei **OCRA.ttf** nach **quellen**.

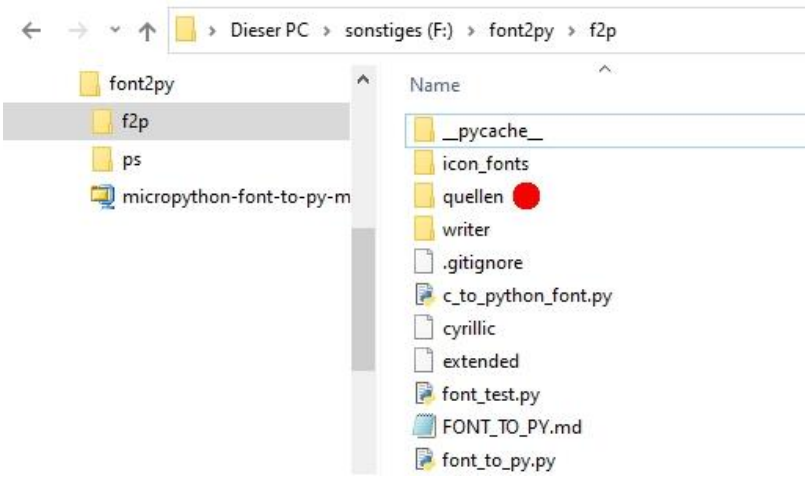

Abbildung 8: Quellverzeichnis für die ausgewählten Fonts

Mit gedrückter Shift-Taste (Umschalten Großschrift) klicke ich mit Rechtsklick auf **f2p** wähle im Kontextmenü PowerShell-Fenster hier öffnen.

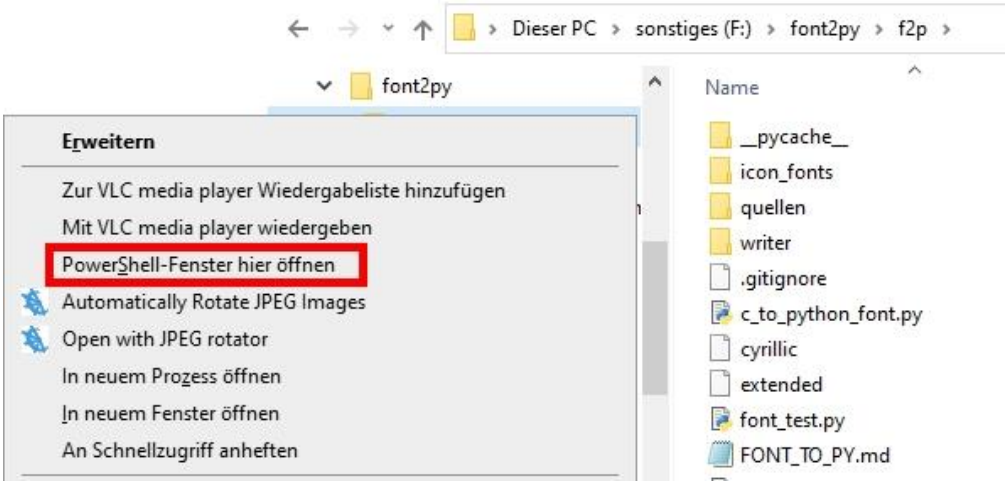

Abbildung 9: PowerShell-Fenster in Zielverzeichnis öffnen

Kopieren sie nun ein Zeichensatzfile aus dem Windowsordner in das Verzeichnis **quellen**.

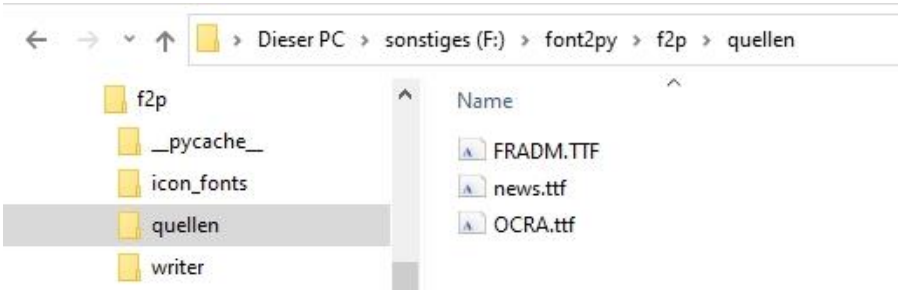

Abbildung 10: TTF-Quell-Dateien

Für die weiteren Aktionen verwende ich die Schrift **OCRA.ttf**. Zwei Schritte stellen die Verwendung der Schriftart OCRA in der Größe 20Pixel zur Verwendung mit unserem MicroPython-Programm als Modul **ocr20.py** bereit. Ich erzeuge, zur

Einsparung von Speicherplatz nur die wirklich benötigten Zeichen: 0123456789NWSO.

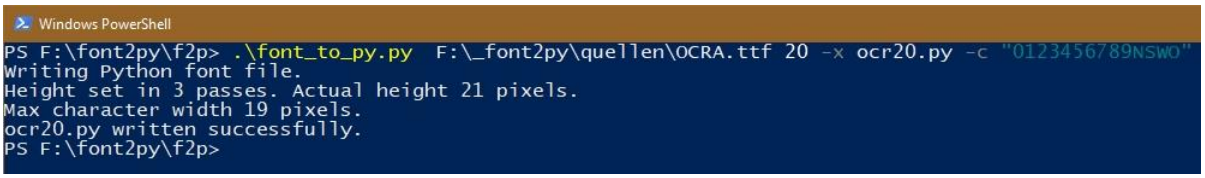

Abbildung 11: Erzeugen der benötigten Zeichen

Die Zeicheninformationen befinden sich jetzt mit allen zusätzlichen Daten in der Datei **ocr20.py**, die ins Arbeitsverzeichnis (workspace) des Projekts kopiert werden muss. Wie der geklonte Zeichensatz aussieht, kann man sich auch im Powershell-Fenster anschauen.

Abbildung 12: Der Zeichensatz OCRA20 für den Einsatz in MicroPython

Außer der Datei **ocr20.py** werden für die Darstellung der großen Zeichen noch die Module **ssd1306.py**, **oled.py** und **writer.py** benötigt. Alle vier müssen in den Flash des ESP8266 hochgeladen werden.

Der Neopixelring

Ein Kompass ohne Peilhilfe nützt wenig. Meine Peilhilfe ist ein Ring aus 12 RGB-LEDs vom Typ WS2812B, die auch unter dem Namen Neopixel-LED bekannt sind. Jede einzelne LED kann über eine allen gemeinsame busähnliche Leitung (IN) angesprochen werden. Drei Bytes legen individuell die Helligkeit der Farben rot, grün und blau fest. Ein Modul im Standard-Kernel der Firmware ermöglich die simple Einrichtung und Ansteuerung.

Die Spannungsversorgung erfolgt parallel. Die Datenleitung führt seriell von einer LED-Einheit zur nächsten und stellt eine besondere Art von Bus dar. Jede Einheit enthält eine RGB-LED und einen Controller, der auf die erste ankommende 24-Bit-Folge der Farbinformation reagiert. Die Signale, mit derselben Periodendauer aber unterschiedlichem Duty Cycle, werden von einem Microcontroller, wie dem ESP8266 erzeugt. Je Neopixel-Einheit werden 24 Bit generiert (jeweils 8 für grün, rot und blau). Die Periodendauer für ein Bit ist 1,25µs +/-0,150µs, die Übertragungsfrequenz beträgt somit ca. 800kHz. Für eine 1 liegt die Leitung 0,8µs auf HIGH und 0,45µs auf LOW, eine 0 wird durch 0,4µs HIGH und 0,85µs LOW codiert. Die ersten ankommenden 24 Bits verarbeitet jede WS2812B-Einheit selbst, ohne sie weiterzugeben. Alle nun folgenden werden verstärkt und an die nächste Einheit

weitergereicht. Die Signalfolge vom Microcontroller wird also von LED zu LED um 24 Bit kürzer. Anders als bei einem üblichen Datenbus erhalten die WS2812B-Einheiten die Daten aber nicht gleichzeitig, sondern zeitversetzt um jeweils die Dauer von 24Bit mal 1,25µs/Bit = 30µs. Diese Signalfolge wird im ESP32/ESP8266 durch die in MicroPython eingebaute Klasse **machine.NeoPixel** erzeugt. Die Ansteuerung der LEDs gestaltet sich dadurch sehr einfach, wodurch sich die Anwendung gerade für Anfänger eignet.

Ein Framebuffer (aka Zwischenspeicher) im RAM-Speicher des ESP-Chips bunkert die Farbwerte (256 hoch 3 = 16,7 Mio.) zwischen, und der Befehl NeoPixel.**write**() schickt die Informationen über den "Bus", der an einem GPIO-Ausgang hängt (bei uns GPIO14 = D5), an den Ring. Das ist auch schon alles. Pro Farbe, rot, grün und blau, lassen sich 256 Helligkeitswerte einstellen.

Mehrere Ringe kann man genau so wie einzelne LEDs cascadieren, indem man den Eingang des nächsten Rings mit dem Ausgang des Vorgängers verbindet. Die Anschlüsse erfolgen rückseitig, am besten mittels dünner Litzen.

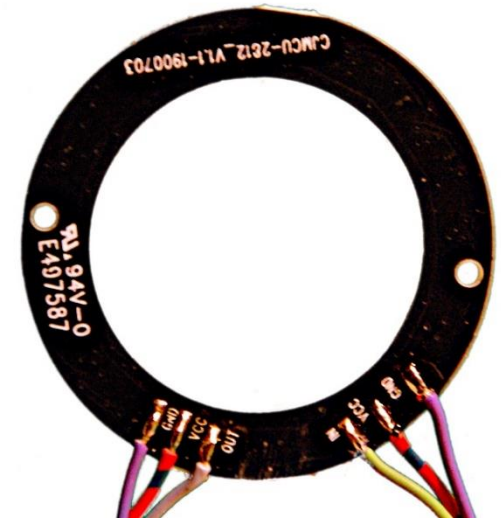

Abbildung 13: LED-Ring_hinten – rechts Zuführung, links Weiterleitung

Abbildung 14: Neopixel-Ring Oberseite

Die Komponenten für Mischfarben ermittelt man am einfachsten experimentell über REPL. Die Helligkeit der einzelnen Teil-LEDs einer Einheit ist recht unterschiedlich. Die RGB-Farbcodes in den Tupeln werden also bei den Mischfarben selten den gleichen Wert haben.

```
>>> from neopixel import NeoPixel
>>> neoPin=Pin(14)
>>> neoCnt=12
>>> np=NeoPixel(neoPin,neoCnt)
>>> np[0]=(32,16,0)
>>> np.write()
```

Zum Abgleich werden die beiden letzten Befehle mit anderem RGB-Code wiederholt, bis die Farbwiedergabe passt. Die hier angegebenen Werte erzeugen gelb als Mischfarbe von rot und grün. Die Ausgabe des Logic-Analyzers zeigt die Codierung der einzelnen Bits. Außerdem verrät der Plot, dass der Grünwert vor dem Rotwert übertragen wird. Die gelbe LED markiert die Position 0 der LED-Nummerierung am Ring.

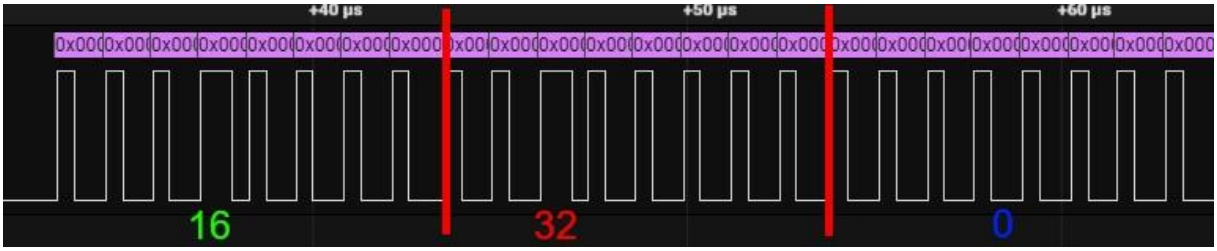

Abbildung 15: Neopixelsignale

Das Kompassprogramm

Die Importabteilung stellt die Zutaten bereit, die wir außer unserem eigenen Code benötigen.

```
import network; network.WLAN(network.AP_IF).active(False)
import gc
from writer import Writer
import sys, os
from machine import Pin, ADC, SoftI2C
from time import sleep,ticks_ms
from neopixel import NeoPixel
from math import atan2, degrees
from oled import OLED
import ocr20 as charSet
```

Für eine störungsfeie Funktion schalten wir das AP-Interface aus. gc steht für Garbage Collection. Kurz vor dem Beginn des Hautprogramms sorgt der Aufruf von gc.collect() für das Aufräumen des RAM-Speichers.

Das Modul Writer arbeitet mit viel RAM-Speicher, weshalb der Import und damit die Declaration der Klasse am Anfang stehen sollte, damit noch genügend zusammenhängender Speicher verfügbar ist.

Von machine brauchen wir die Unterstützung für GPIO-Pins, den analog-Digital-Converter und den I2C-Bus.

Zur Winkelberechnung binden wir die Funktionen **atan2**() und **degrees**() aus dem Modul **math** ein.

Die Klasse **OLED** bildet die Basis für den Treiber der großen Zeichenausgabe. Das zugehörige Modul **ocr20** importieren wir unter dem Alias **charSet**. Beim Ändern des Zeichensatzes muss nur diese eine Zeile angepasst werden. Selbstredend muss die Datei natürlich in den Flash des ESP8266 geladen werden.

Die Klasse QMC5883L

Im Modul struct wohnt die Methode unpack(), die wir in der Klasse QMC5883L benötigen, um die vorzeichenbehafteten 16-Bit-Daten vom Kompassmodul in Zahlen zu verwandeln. Danach lege ich die Registeradressen und die Flags zur Funktionskontrolle als Konstanten fest. Das sorgt dafür, dass die Werte im Flash und nicht im RAM abgelegt werden.

```
import struct
class QMC5883L:
    QMC5883 = const(0x0D) # 7-Bit HWADR
    XRegL = const(0x00)
    XRegH = const(0x01)
    YRegL = const(0x02)
    YRegH = const(0x03)
    ZRegL = const(0x04)
    ZRegH = const(0x05)
    StatusReg = const(0x06)
    TempL = const(0x07)
    TempH = const(0x08)
    CtrlReg1 = const(0x09)
    CtrlReg2 = const(0x0A)
    Period = const(0x0B)

    DOR = const(0x04)
    OVL = const(0x02)
    DRDY= const(0x01)

    ModeMask= const(0xFC)
    Standby = const(0x00)
    Continuous = const(0x01)

    RateMask= const(0xF3)
    ORate10 = const(0x00)
    ORate50 = const(0x04)
    ORate100= const(0x08)
```

```
    ORate200= const(0x0C)

    ScaleMask=const(0xCF)
    FScale2 = const(0x00)
    FScale8 = const(0x10)

    OSRMask= const(0x3F)
    OSR512 = const(0x00)
    OSR256 = const(0x40)
    OSR128 = const(0x80)
    OSR64  = const(0xC0)

    SoftRST= const(0x80)
    RollOnt= const(0x40)
    IntEable=const(0x01)

    BezugsPegel = const(1000)
```

Der Konstruktor der Klasse ist, wie üblich, die Methode **__init__**(). Als Positions-Parameter muss ein I2C-Objekt übergeben werden. Ferner erlauben die optionalen Parameter **ORate**, **FScale** und **OSR** die Einstellung Ausgaberate, der maximal messbaren Flussdichte in Gauss und des Oversamplingwerts. Die Parameter sind mit Default-Werten vorbelegt und werden an Instanzvariablen übergeben. Der Messmodus wird auf kontinuierlich gesetzt. Die Methode **configQMC**() sendet die Werte an den QMC5883L. Wurde bereits mindestens ein Kalibriervorgang durchgeführt, dann findet **readCalibration**() eine Datei. **config.txt** im Flash des ESP8266 und liest von dort die Daten ein. Existiert die Datei nicht, wird eine Kalibrierung durchgeführt.

```
    def __init__(self,i2c,
                 ORate=ORate200,
                 FScale=FScale2,
                 OSR=OSR512,
                 ):
        self.i2c=i2c
        self.mode=Continuous
        self.oRate=ORate
        self.fScale=FScale
        self.osr=OSR
        self.configQMC()
        self.bezugsPegel=BezugsPegel
        self.readCalibration()
        print("QMC5883L is @ {}".format(QMC5883))
```

Die Methode **writeToReg**() nimmt die Registernummer und einen Bytewert. Letzterer wird in ein bytes-Objekt umgewandelt, das die Methode **writeto_mem**() des I2C-Ojekts an den QMC5883L sendet.

```
    def writeToReg(self,reg,val):
        d=bytes([val & 0xFF])
        self.i2c.writeto_mem(QMC5883,reg,d)
```

Mehrere Bytes können als bytes-Objekt oder als bytearray mit der Methode **writeBytesToReg**() gesendet werden.

```
    def writeBytesToReg(self,reg,buf):
        self.i2c.writeto_mem(QMC5883,reg,buf)
```

configQMC() sendet die Konfigurationsdaten an dem QMC5883L. Der Wert für das Register **Period**=0x0B wird vom Hersteller ohne weitere Erläuterung mit 1 vorgegeben.

```
    def configQMC(self):
        c1=self.mode | self.oRate | self.fScale | self.osr
        self.writeToReg(CtrlReg1,c1)
        c2=0
        self.writeToReg(CtrlReg2,c2)
        self.writeToReg(Period,0x01)
```

Hintereinander folgende Register des QMC5883L werden in einem Rutsch durch die Methode **readNbytesFromReg**() gelesen, der neben dem Startregister die Anzahl n zu lesender Bytes übergeben wird.

```
    def readNbytesFromReg(self,reg,n):
        return self.i2c.readfrom_mem(QMC5883,reg,n)
```

Das Bit DRDY im Statusregister signalisiert mit einer 1, dass neue Daten vorliegen. Wir lesen das Statusregister ein und wandeln das bytes-Objekt durch Indizierung der bytes-Folge mit 0 in einen Integerwert um. Wir maskieren das DRDY-Bit sowie das OVL-Bit, das wir zusätzlich in die Position 0 schieben. Die 1 oder 0 in **drdy** und **ovl** kann als True oder False interpretiert werden. Nur wenn drdy True ist und kein Überlauf gemeldet wird (was durch das Erdmagnetfeld alleine sicher nicht vorkommt), können neue Werte abgeholt werden.

```
    def dataReady(self):
        val=self.i2c.readfrom_mem(QMC5883,StatusReg,1)[0]
        drdy = val & DRDY
        ovl = (val & OVL)>>1
        return  drdy & (not ovl)
```

Das geschieht durch **readAxes**(). Wir warten, bis **dataReady**() **True** liefert und lesen dann alle 6 Bytes ab Register 0x00 ein. So schreibt es das Datenblatt vor. Die Bytesfolge in **achsen** stellt die vorzeichenbehafteten 16-Bit-Werte der drei Achsen in Little-Endian-Notierung dar. Das heißt, dass das niederwertige Byte vor dem höherwertigen Byte gesendet wird. Die Methode **unpack**() erhält diese Information

durch das "<"-Zeichen im Formatstring. Die drei "h" sagen, dass es sich um vorzeichenbehaftete 16-Bit-Werte handelt. Die Methode gibt ein Tuple zurück, das wir sofort weiter entpacken und an die Variablen x, y und z für die Rückgabe weiterleiten.

```
    def readAxes(self):
        while not self.dataReady():
            pass
        achsen=self.readNbytesFromReg(0x00,6)
        x,y,z=struct.unpack("<hhh",achsen)
        return x,y,z
```

Mit Hilfe der Kalibrierungswerte, und dem Wert in **BezugsPegel** (=1000) normieren wir die eingelesenen Werte in x- und y-Richtung auf den Bereich -1000 bis +1000 mit Mittelwert 0. Dazu berechnen wir den Abstand der Messwerte vom arithmetischen Mittelwert der Kalibrierung. Anschließend wird auf den Bezugspegel skaliert und auf eine ganze Zahl gerundet.

```
    def normalize(self,x,y):
        x-=k.xmid
        y-=k.ymid
        x=int(x/self.dx*1000+0.5)
        y=int(y/self.dy*1000+0.5)
        return x,y
```

Das Oversampling des QMC5883L beruhigt zwar die Messwerte und verringert das Rauschen, aber glücklich hat mich das nicht gemacht. Deshalb sorgt die Methode axesAverage() für weitere Beruhigung. Für n=100 schwanken die Messwerte nur noch um +/-1 Grad.

```
    def axesAverage(self,n):
        xm,ym=0,0
        for i in range(n):
            x,y,z=k.readAxes()
            x,y=k.normalize(x,y)
            xm+=x
            ym+=y
        xm=int(xm/n)
        ym=int(ym/n)
        return x,y
```

Mit den Werten, die **axesAverage**() zurückgibt, lasse ich den Peilwinkel berechnen. Sonderfälle in Achsennähe werden eigens decodiert. **atan2**() nimmt die Achsenwerte und gibt den Winkel im Bogenmaß zurück. **degrees** rechnet ins Gradmaß um.

```
    def calcAngle(self,x,y):
        angle=0
        if abs(y) < 1:
            if x > 0:
                angle = 0
            if x < 0:
                angle = 180
        else: # |y| > 1
            if abs(x) < 1:
                if y > 0:
                    angle = 90
                if y < 0:
                    angle = 270
            else: # x > 1
                angle = degrees(atan2(y,x))
                if angle < 0:
                    angle+=360
        return angle
```

Die Methode calibrate() erfasst während 20 Sekunden Messdauer fortwährend die Flussdichte in x- und y-Richtung. In dieser Zeit muss der Aufbau, am besten mehrmals, in waagrechter Ausrichtung um die z-Achse um mindestens 360° gedreht werden und das nicht zu schnell. Durch die Drehung wird für jede Achse der minimale und maximale Messwert bestimmt. Daraus berechne ich die Mittelwerte und deren Abweichung zu den Randwerten. Die vier Ergebnisse schreibe ich als Strings in die Datei **config.txt**. Zur sofortigen Kontrolle werden die Werte auch im Terminal ausgegeben. Bei kommerziellen Produkten heißt es in der Anleitung, man soll zur Kalibrierung den Kompass horizontal in Form einer 8 bewegen. Sinn dieser Übung ist, dass er dabei auch zweimal um 360 Grad gedreht wird.

```
    def calibrate(self):
        xmin=32000
        xmax=-32000
        ymin=32000
        ymax=-32000
        finished=self.TimeOut(20000)
        d.clearAll()
        d.writeAt("CALIBRATING",0,0,False)
        d.writeAt("ROTATE DEVICE",0,1)
        sleep(3)
        while not finished():
            x,y,z=self.readAxes()
            xmin=(xmin if x >= xmin else x)
            xmax=(xmax if x <= xmax else x)
            ymin=(ymin if y >= ymin else y)
            ymax=(ymax if y <= ymax else y)
        self.xmid=(xmin+xmax)//2
        self.ymid=(ymin+ymax)//2
        print (xmin,self.xmid, xmax)
        print (ymin,self.ymid, ymax)
        self.dx=(xmax-xmin)//2
```

```
        self.dy=(ymax-ymin)//2
        print (self.dx, self.dy)
        with open("config.txt","w") as f:
            f.write(str(self.xmid)+"\n")
            f.write(str(self.ymid)+"\n")
            f.write(str(self.dx)+"\n")
            f.write(str(self.dy)+"\n")
        d.writeAt("CAL. DONE",0,2)
```

Die **with**-Anweisung sorgt übrigens am Ende der Struktur automatisch für das Schließen der Datei.

readCalibration() ist das Gegenstück zu **calibrate**() und wird nach dem Start des Programms durch den Konstruktor aufgerufen. Stellt try als Exception einen **OSError** fest, dann existiert wohl die Datei **config.txt** nicht und infolge wird eine Konfiguration ausgeführt.

```
    def readCalibration(self):
        try:
            with open("config.txt","r") as f:
                self.xmid=int(f.readline())
                self.ymid=int(f.readline())
                self.dx=int(f.readline())
                self.dy=int(f.readline())
        except OSError:
            self.calibrate()
```

Die Methode **TimeOut**() ist mein nicht blockierender Timer, der in **calibrate**() den Ablauf von 20 Sekunden kontrolliert, während gleichzeitig die Messungen erfolgen. Es ist eine sogenannte Closure. Über diese Art von Funktionen können Sie hier mehr erfahren.

```
    def TimeOut(self,t):
        start=ticks_ms()
        def compare():
            return int(ticks_ms()-start) >= t
        return compare
```

In der Zeile
finished=self.TimeOut(20000)

in **calibrate**() wird der Variablen **finished** eine Referenz auf die Funktion **compare**(), die innerhalb von **TimeOut**() deklariert ist, übergeben. Auf diesem Umweg bleiben die für **compare**() freien Variablen auch nach dem Beenden von **TimeOut**() am Leben und **compare**() ist über den Alias **finished** von außerhalb **TimeOut**() aufrufbar.

while not finished():

finished() liefert dann ein **True** zurück, wenn die 20 Sekunden abgelaufen sind.

Zwei Funktionen arbeiten mit dem LED-Ring zusammen

Die Funktion **alleAus**() löscht alle LEDs im Ring, indem sie in der for-Schleife alle Farbwerte auf 0 setzt. Der Standardparameter **show** ist mit **True** vorebelegt und kann beim Aufruf der Funktion weggelassen werden, wenn nach Ablauf der Schleife die Werte sofort an den Ring weitergegeben werden sollen. False verhindert das und spart Zeit, wenn vorher weitere Farbinformationen gesetzt werden sollen.

Die Funktion **nord**() ist neben der Magnetfeldmessung das Kernstück des Programms. Wie der Name es schon sagt, zeigt die Funktion stets die Nord-Südrichtung durch eine rote und eine grüne LED am Neopixelring an. Das gelingt für die 12 Positionen im 30°-Raster perfekt. Für Zwischenwerte bis 15° um einen 30°-Raster-Wert herum dienen blaue LEDs nahe den Richtungs-LEDs als Hinweis für eine Abweichung. Die Helligkeit der blauen LEDs ist ein Maß für die Größe der Abweichung vom Raster, dunkler weniger, heller mehr. Liegt die Nordrichtung genau zwischen zwei 30°-Rastern, dann leuchten zwei rote und gegenüberliegend zwei grüne LEDs.

```
def nord(alpha,n):
    alleAus(False)
    beta=360-alpha
    step=360//n
    hstep=step//2
    q=(beta//step)%n
    m=beta%step
    hF=(1023-h.read())/1023
    if 1 < m < step//2:
        np[led[q]]=(int(255*hF),0,0)
        np[led[(q+1)%n]]=(0,0,int(64*(m)/hstep*hF))
        np[(led[q]+6)%n]=(0,int(64*hF),0)
        np[(led[(q+1)%n]+6)%n]=(0,0,int(64*(m)/hstep*hF))
    elif m > step//2:
        np[led[(q+1)%n]]=(int(hF*255),0,0)
        np[led[q]]=(0,0,int(64*(step-m)/hstep*hF))
        np[(led[(q+1)%n]+6)%n]=(0,int(hF*64),0)
        np[(led[q]+6)%n]=(0,0,int(64*(step-m)/hstep*hF))
    elif m == step//2:
        np[led[q]]=(int(255*hF),0,0)
        np[led[(q+1)%n]]=(int(255*hF),0,0)
        np[(led[(q+1)%n]+6)%n]=(0,int(hF*64),0)
        np[(led[q]+6)%n]=(0,int(hF*64),0)
    else:
        np[led[q]]=(int(255*hF),0,0)
        np[(led[q]+6)%n]=(0,int(hF*64),0)
    np.write()
```

Die Funktion bekommt den Peilwinkel und die Anzahl der LEDs auf dem Ring mitgeteilt und schaltet zunächst alle Farben auf 0, ohne diesen Zustand an den Ring weiterzugeben. Den Winkel beta der Nordrichtung bekommen wir, indem wir den Peilwinkel alpha der Marschrichtung von 0° oder besser 360° abziehen.

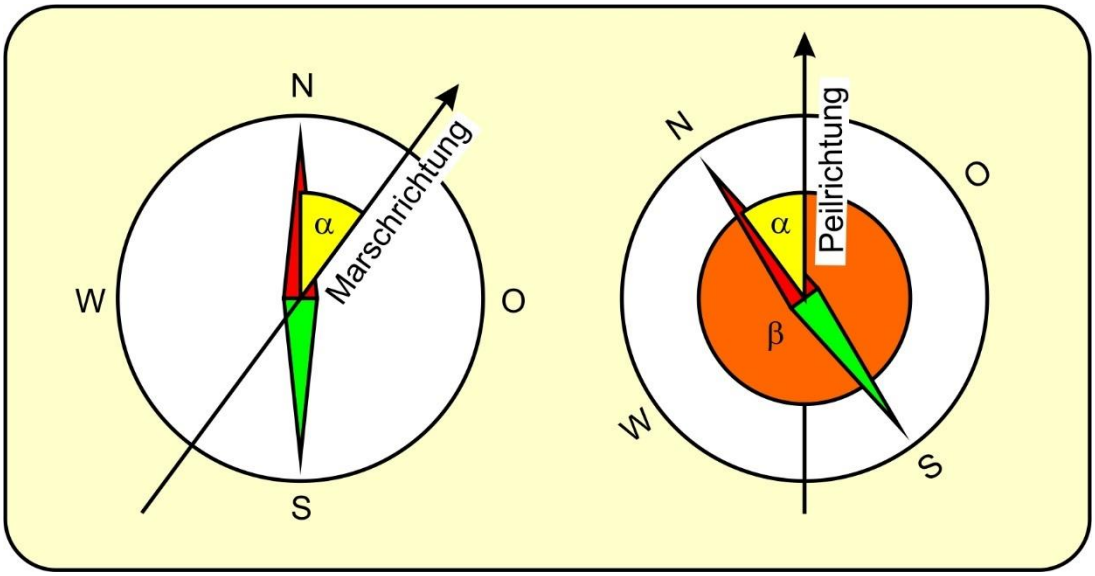

Abbildung 16: Peilwinkel und Nord-Süd-Richtung:

Wir berechnen die Schrittweite des Rasters in Grad und die ganzzahlige Hälfte davon. Der Quotientenwert der Ganzzahldivision des Nordwinkels beta durch Schrittweite liefert die Nummer der LED für die Grobrichtung und der Teilungsrest m sagt uns die Abweichung vom Raster. Die Helligkeit der LEDs wird über den LDR auch von der Helligkeit des Umgebungslichts gesteuert. Das geschieht über den Helligkeitsfaktor **hF**,

Die Nummerierung der LEDs beginnt mit 0 am Anschluss des Rings und setzt sich im Uhrzeigersinn bis 11 fort. Damit die Zählung an einer beliebigen LED beginnen kann, habe ich die Liste **led** definiert. Das Element **led**[0] muss in Peilrichtung liegen, hier ist das also die LED mit der Nummer 3.

led=[3,4,5,6,7,8,9,10,11,0,1,2]

Für das Berechnen des Neopixel-Index ist der Quotientenwert q zuständig. Beim Hochrechnen auf den nächsten Rasterwert oder beim Berechnen der Süd-Richtung (+6) wird der Teilungsrest modulo 12 bestimmt, damit der Index innerhalb der Nummern der LEDs beziehungsweise der gültigen Indizes der Liste led bleibt.

Die Berechnung der RGB-Werte richtet sich nach dem Helligkeitsfaktor und bei den blauen LEDs außerdem nach der Abweichung m vom Rasterwert. Ist m größer als die halbe Rasterweite, dann ist die LED mit der Nummer q+1 der angesagtere Richtungswert und die Abweichung wird auf diese LED bezogen. Sie können sich das am Kompassgradnetz gut verdeutlichen. Nach Abschluss der Berechnungen senden wir die Farbwerte mit **np.write**() an den Ring.

Das Hauptprogramm

Die Vorbereitungen im Hauptprogramm beginnen mit der Auswahl des Controllers und der damit verbundenen Festlegung der Pins für den I2C-Bus.

```
chip=sys.platform
if chip == 'esp8266':
    SCL=Pin(5) # S01: 0
    SDA=Pin(4) # S01: 2
elif chip == 'esp32':
    SCL=Pin(21)
    SDA=Pin(22)
else:
    raise OSError ("Unbekannter Port")
```

Dann räumen wir den Speicher auf, erzeugen die Liste **led** und die Liste **richtungen** mit den groben Richtungsbezeichnungen für die Ausgabe am OLED-Display.

```
gc.collect()
led=[3,4,5,6,7,8,9,10,11,0,1,2]
richtungen=["N","NNO","NO","ONO","O","OSO","SO","SSO","S",
          "SSW","SW","WSW","W","WNW","NW","NNW"]
```

Wir instanziieren ein Neopixel-Objekt für 12 LEDs an GPIO14. Die Rasterweite der Liste **richtungen** ist 22,5 Grad. Der LDR liegt an **A0**. Wir erzeugen eine I2C-Instanz, versorgen damit das OLED-Objekt und mittelbar damit auch das Writer-Objekt. Das Magnetometerobjekt **k** und eine Pin-Instanz **taste** schließen den Reigen der Deklarationen ab.

```
neoPin=Pin(14,Pin.OUT,value=1)
neoCnt=12
np=NeoPixel(neoPin,neoCnt)
delta=22.5
h=ADC(0)
i2c=SoftI2C(SCL,SDA,freq=400000)
d=OLED(i2c)
wr=Writer(d,charSet)
k=QMC5883L(i2c,OSR=OSR512,ORate=ORate200,FScale=FScale2)
taste=Pin(0,Pin.IN,Pin.PULL_UP)
```

Sollte jetzt die Taste gedrückt sein, wird eine Kalibrierung durchgeführt.

```
if taste.value()==0:
    k.calibrate()
```

Sonst geht es in die Hauptschleife.

```
else:
    while 1:
        a=k.axesAverage(100)
        w=int(k.calcAngle(a[0],a[1]))
        print(w)
        nord(w,12)
        richtung=int((w+delta/2)/delta)
        richtung = (richtung if richtung < \
                    len(richtungen) else 0)
        d.fill(0)
        wr.set_textpos(d,0,32)
        wr.printstring(str(int(w+0.5)))
        wr.set_textpos(d,32,32)
        wr.printstring(richtungen[richtung])
        d.show()
```

Wir holen uns die Mittelwerte der normierten Achsenwerte und berechnen aus dem x- und y-Wert den Peilwinkel der Marschrichtung, den wir ans Terminal senden und an die Funktion **nord**() übergeben. Der Index **richtung** in die Liste **richtungen** wird als Ganzzahl aus dem gerundeten Quotienten **w/delta** berechnet und auf den gültigen Bereich eingegrenzt.

Das Display wird gelöscht, nach dem Bestimmen der Ausgabeposition erfolgt die Ausgabe des Winkels und der Richtungsbezeichnung. Die Methode **show**() sendet den bestückten Framebuffer vom ESP8266 an das Display zur Anzeige.

Einsatz

Für den Einsatz im Freien ist es nötig, dass der ESP8266 das Programm autonom starten kann. Dafür speichert man den gesamten Programmtext unter dem Namen boot.py im Workspace ab und lädt diese Datei in den Flash des ESP8266 hoch. Beim nächsten Neustart bootet der ESP8266 mit dem Kompassprogramm auch ohne USB-Verbindung zum PC.

Ich wünsche Ihnen viel Vergnügen beim Bauen und Programmieren und natürlich bei vielen spannenden Abenteuern mit Ihrem neuen Wegbegleiter in Mutter Grün.

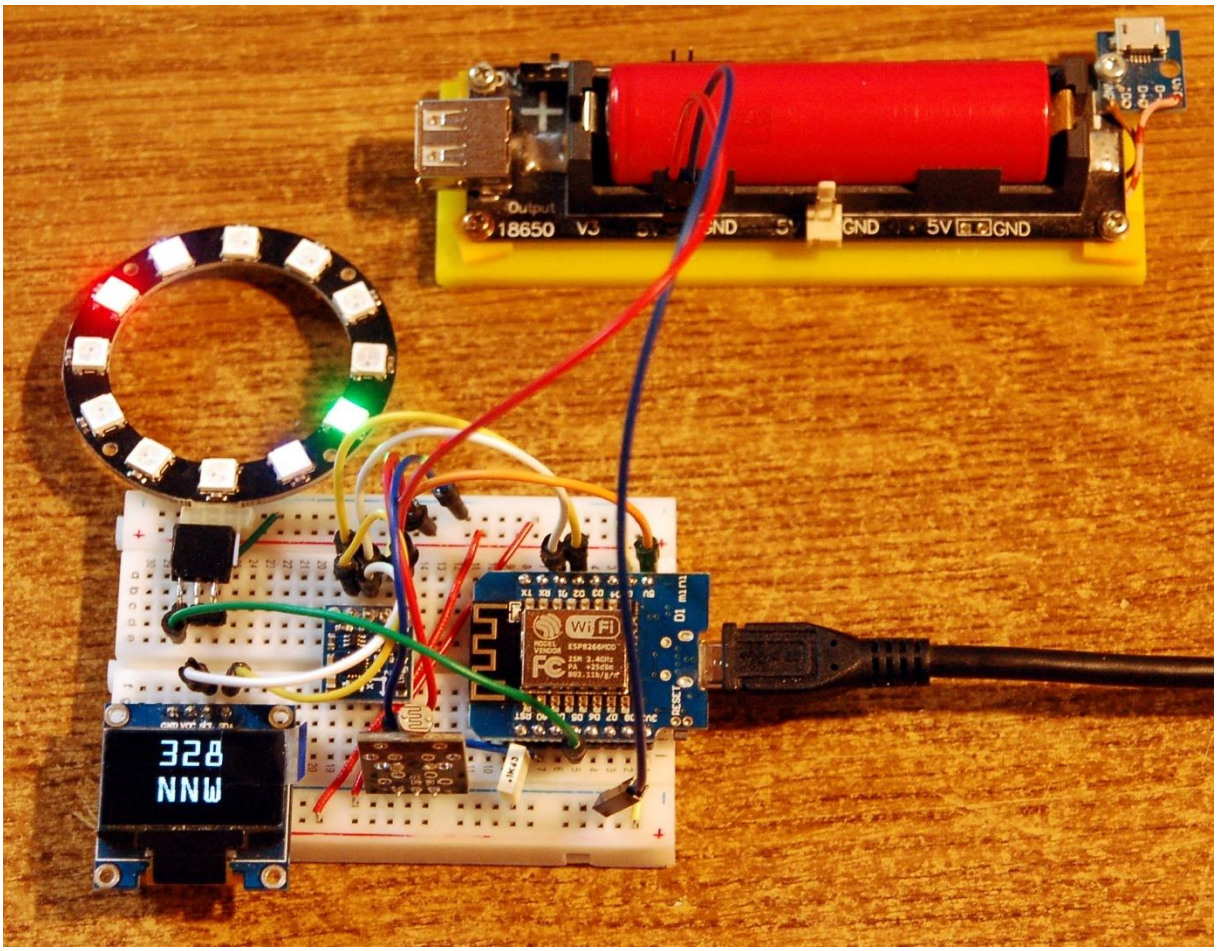

Abbildung 17: Entwicklungsumgebung