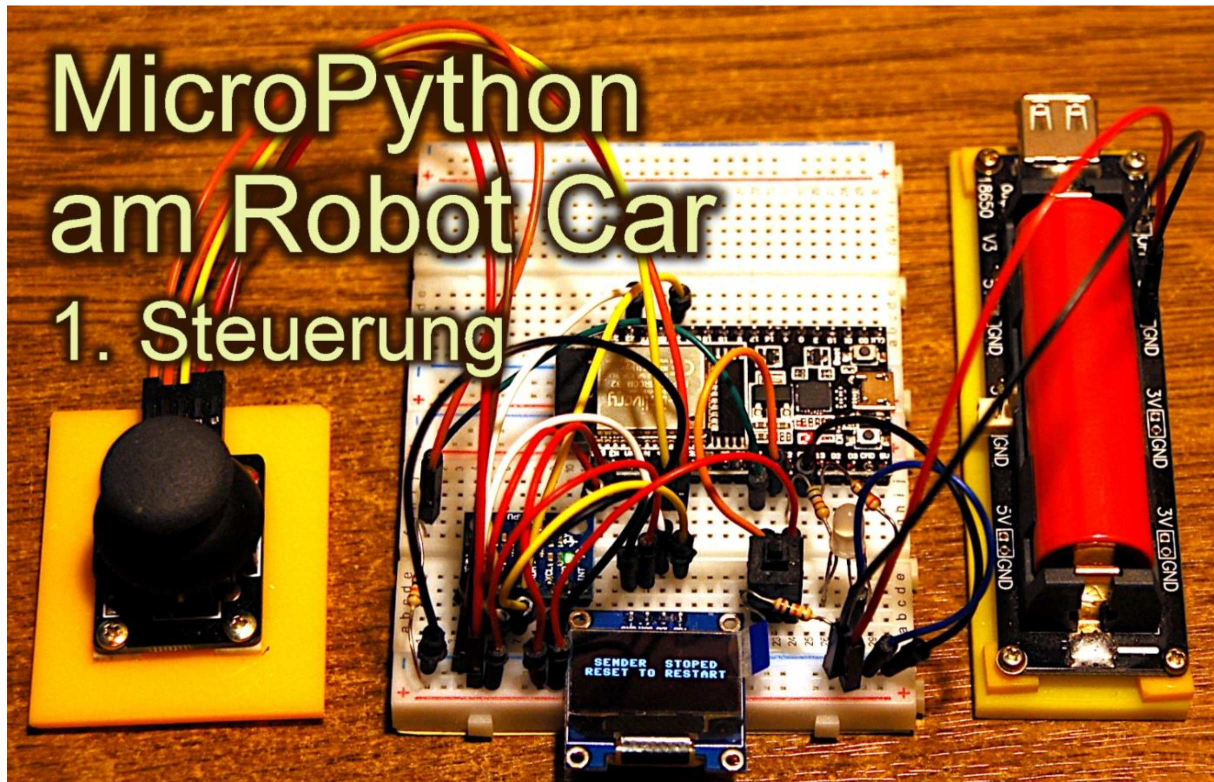


Ein Freund fragte mich vor einiger Zeit, ob ich nicht in MicroPython die Steuerung für ein Robot Car entwickeln könnte. Klar, meinte ich, müsste gehen. Eine Fernsteuerung auf WiFi-Basis, da reicht vermutlich ein ESP8266 und auf dem Fahrzeug wird man einen ESP32 brauchen wegen der zahlreichen Steuerleitungen. Fahrzeug bauen, Batteriekasten drauf, eine Motorensteuerung, dafür gibt's spezielle Chips und ein paar Sensoren für Abstandsmessung und Bahnführung, das geht rucki zucki!

Dachte ich. –

Vor ein paar Wochen…

Aber – God is a girl und der Teufel ist ein Eichhörnchen und der steckt wie man weiß im Detail. Aber jetzt ist es tatsächlich so weit, ich kann Ihnen heute schon einmal die Steuerung für das Projekt vorstellen. Damit willkommen zum Making-Of von

MicroPython am Robot Car
1.Teil – Die Steuerung

Die Teile waren schnell ausgesucht. Für die Steuerung hatte ich folgendes Material besorgt. Das Fahrzeug selbst kommt in Teil 2. Und in Teil 3 gibt es dann eine Überraschung – die etwas andere Robotersteuerung.

1	ESP32 Dev Kit C V4
1	GY-521 MPU-6050 3-Achsen-Gyroskop und Beschleunigungssensor für Arduino - GY-521
1	1-Relais 5V KY-019 Modul High-Level-Trigger für Arduino - 1x Modul
1	KY-023 Joystick Modul für Arduino UNO R3 - 1x Modul
1	ADS1115 ADC Modul 16bit 4 Kanäle für Arduino und Raspberry Pi
1	KY-011 Bi-Color LED Modul 5mm für Arduino
1	KY-004 Taster Modul Sensor Taste Kopf Schalter Schlüsselschalter für Arduino
1	Widerstand 330 Ohm für rote LED
1	Widerstand 680 Ohm für grüne LED
1	Widerstand 10k für Joystick Modul (Taster)
2	MB-102 Breadboard Steckbrett mit 830 Kontakten für Arduino oder die Miniausführung AZDelivery 3 x Mini Breadboard 400 Pin mit 4 Stromschienen kompatibel mit Jumper Wire Kabeln und kompatibel mit Arduino
1	AZDelivery 18650 Battery Expansion Shield V3 Micro-USB-Anschluss inkl. USB Kabel kompatibel mit Arduino inklusive E-Book
1	0,96 Zoll OLED I2C Display 128 x 64 Pixel für Arduino und Raspberry Pi
diverse	Jumperkabel

Verwendete Software:
Fürs Flashen und die Programmierung des ESP:
Thonny oder
µPyCraft

Für das Testen der Funktion des Senders:
ncat

Fürs Testen des Servers auf dem Fahrzeug
packetsender

Ein paar Gedanken zu MicroPython

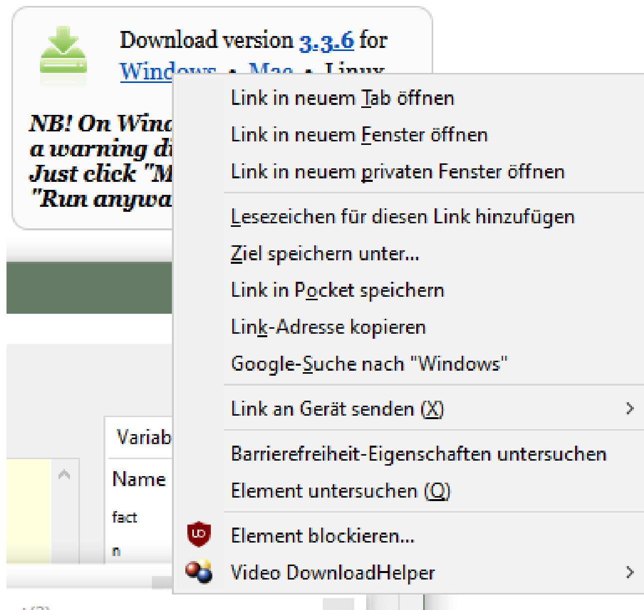
In diesem Projekt wird die Interpretersprache MicroPython benutzt. Der Hauptunterschied zur Arduino-IDE ist, dass Sie die MicroPython-Firmware auf den ESP32 flashen müssen, bevor der Controller MicroPython-Anweisungen versteht. Wenn das geschehen ist, können Sie sich aber zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle senden und sofort die Antwort sehen, ohne vorher ein ganzes Programm compilieren zu müssen. Welche Schritte dazu nötig sind, sagt Ihnen der folgende Abschnitt.

Die Entwicklungsumgebung - Thonny

Thonny ist unter MicroPython das Gegenstück zur Arduino-IDE. In Thonny sind ein Programmeditor und ein Terminal sowie weitere interessante Entwicklungstools in einer Oberfläche vereint. So haben sie das Arbeitsverzeichnis auf dem PC, das Dateisystem auf

dem ESP32, Ihre Programme im Editor, die Terminalconsole und zum Beispiel den Object inspector in einem Fenster übersichtlich im Zugriff.

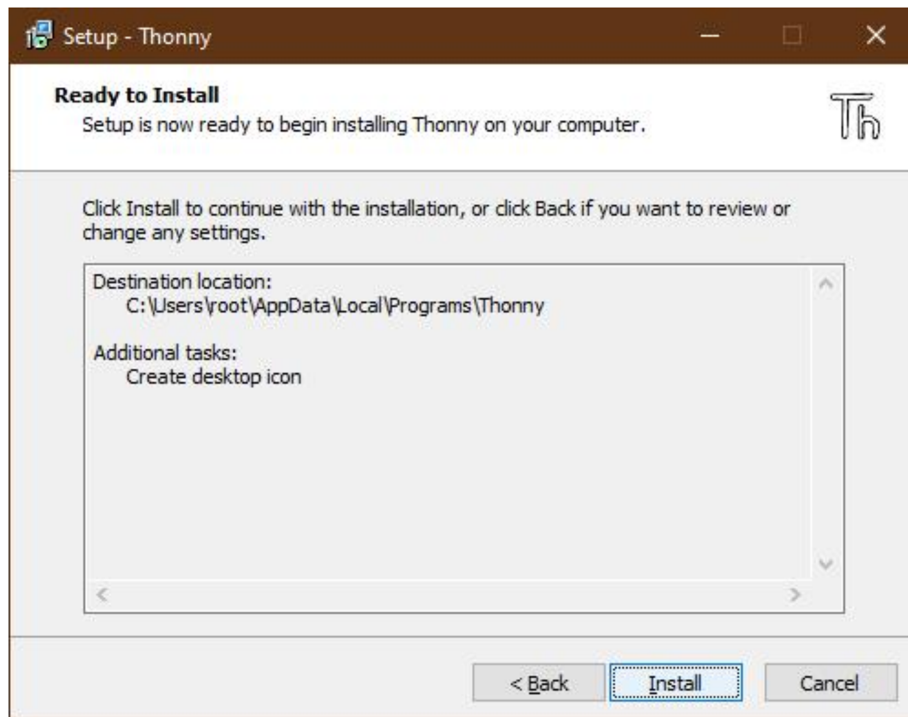
Die Resource zu Thonny ist die Datei thonny-3.3.x.exe , deren neuste Version direkt von der Produktseite heruntergeladen werden kann. Dort kann man sich auch einen ersten Überblick über die Eigenschaften des Programms verschaffen.



Mit Rechtsklick auf **Windows** und **Ziel speichern unter** laden Sie die Datei in ein beliebiges Verzeichnis Ihrer Wahl herunter. Alternativ können Sie auch diesem Direktlink folgen.
Im Bundle von **Thonny** sind neben der IDE selbst auch **Python 3.7** für Windows und **esptool.py** enthalten. Python 3.7 (oder höher) ist die Grundlage für Thonny und esptool.py. Beide Programme sind in Python geschrieben. **esptool.py** dient unter anderem auch in der Arduino-IDE als Werkzeug, um Software auf den ESP32 (und andere Controller) zu transferieren.

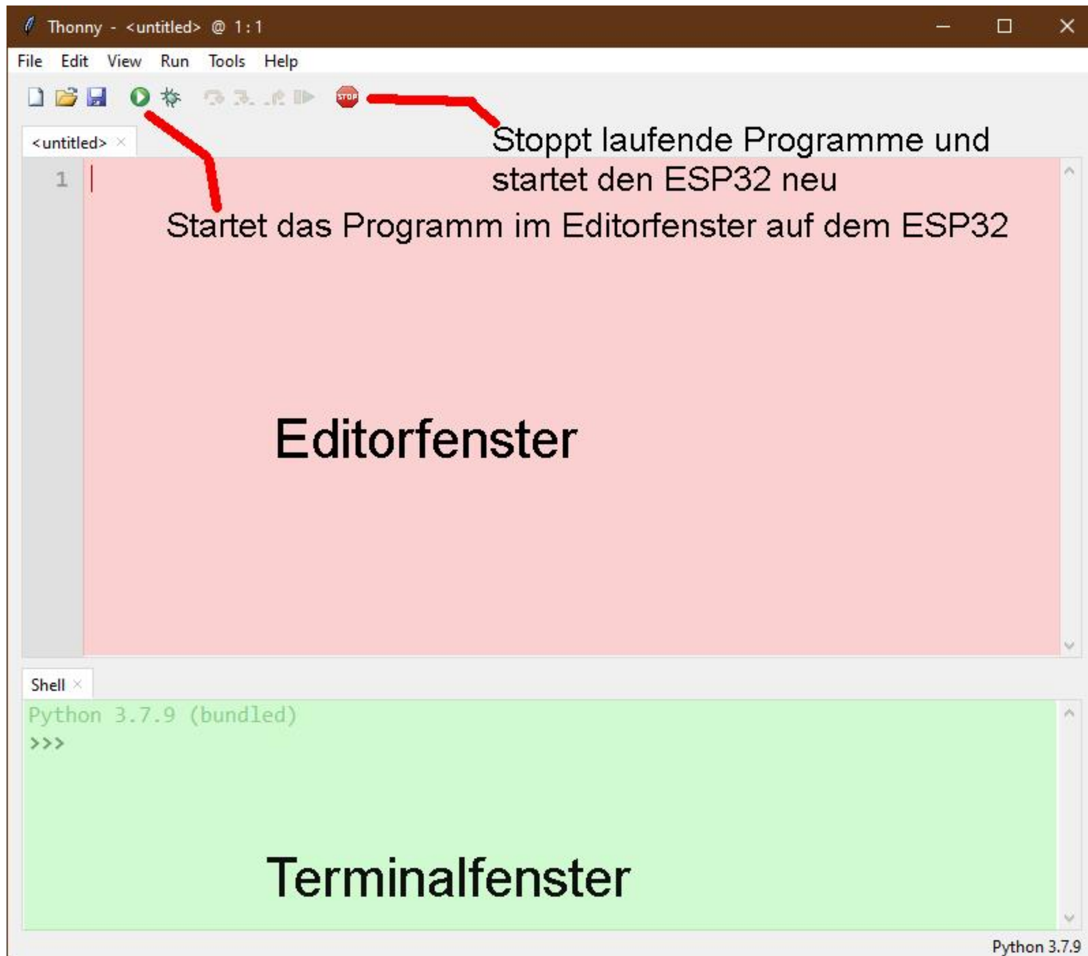
Starten Sie jetzt die Installation von Thonny durch Doppelklick auf ihre heruntergeladene Datei, wenn Sie die Software nur für sich selbst nutzen möchten. Wenn Thonny & Co. allen Usern zur Verfügung stehen soll, müssen Sie die exe-Datei als Administrator ausführen. In diesem Fall klicken Sie rechts auf den Dateieintrag im Explorer und wählen **Als Administrator ausführen**.

Sehr wahrscheinlich meldet sich der Windows Defender (oder Ihre Antivirensoftware). Klicken Sie auf **weitere Informationen** und im folgenden Fenster auf **Trotzdem ausführen**. Folgen Sie jetzt einfach der Benutzerführung mit **Next**.

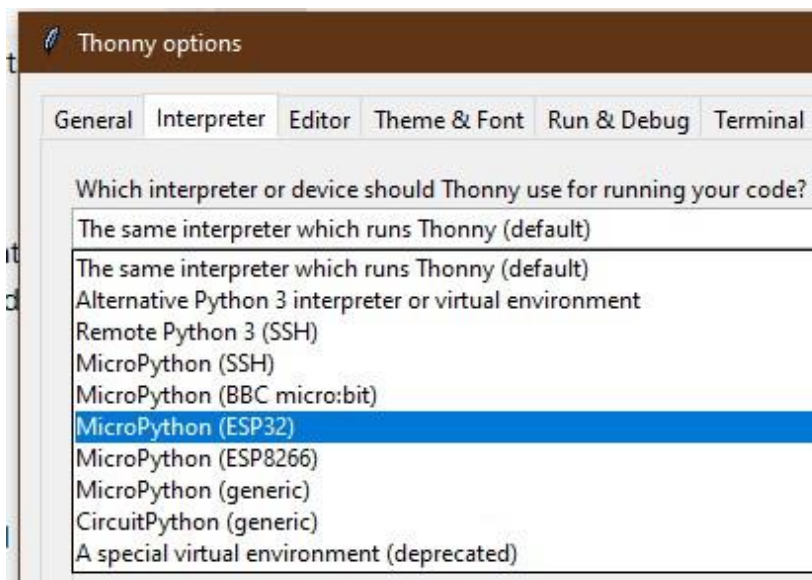


Mit Klick auf **Install** startet der Installationsprozess.

Beim ersten Start geben Sie die Sprache an, dann wird das Editorfenster zusammen mit dem Terminalbereich angezeigt.

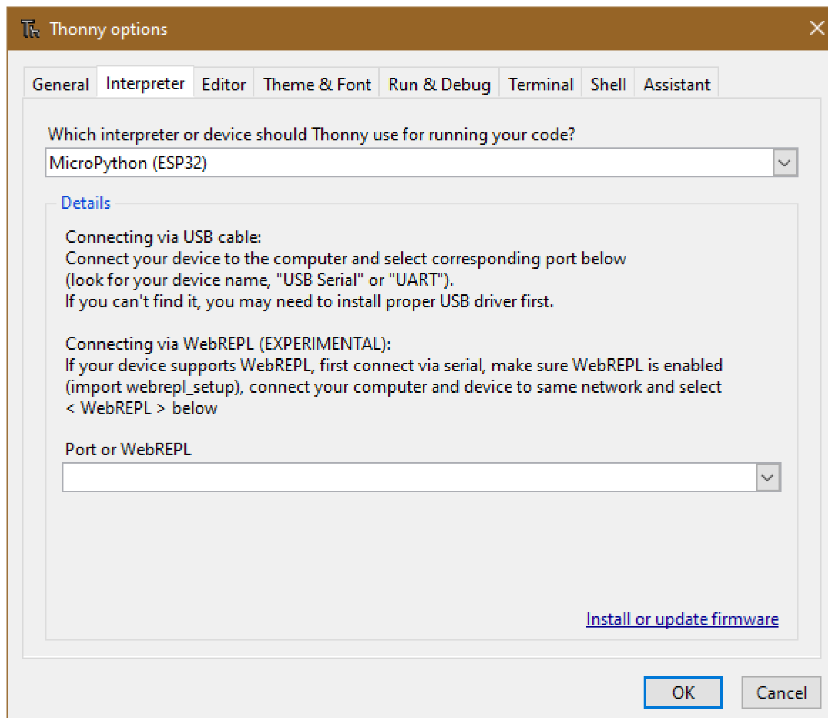


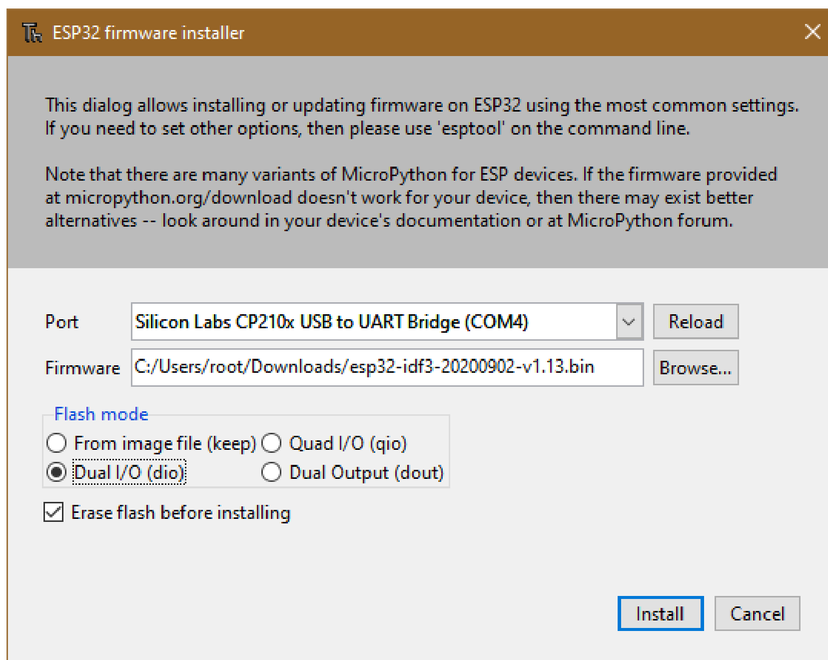
Stellen Sie als erste Aktion den verwendeten Controllertyp ein. Mit **Run – Select Interpreter …** landen Sie in den Optionen. Für dieses Projekt stellen Sie bitte Micropython(ESP32) ein.



Laden Sie jetzt die Firmware Micropython für den ESP32 herunter und speichern Sie diese Datei in einem Verzeichnis Ihrer Wahl. Die bin-Datei muss als erstes auf den ESP32 transferiert werden. Das geschieht auch mit Thonny. Rufen Sie wieder mit

Run – Select Interpreter … Thonny Options auf. Rechts unten klicken Sie auf **Install or update Firmware.**





Wählen Sie den seriellen Port zum ESP32 und die heruntergeladene Firmwaredatei aus. Mit **Install** starten Sie den Prozess. Nach kurzer Zeit befindet sich die MicroPython-Firmware auf dem Controller und Sie können die ersten Befehle über REPL, die MicroPython-Kommandozeile, an den Controller senden. Geben Sie im Terminalfenster zum Beispiel folgenden Befehl ein.

```
print("Hallo Welt")
```

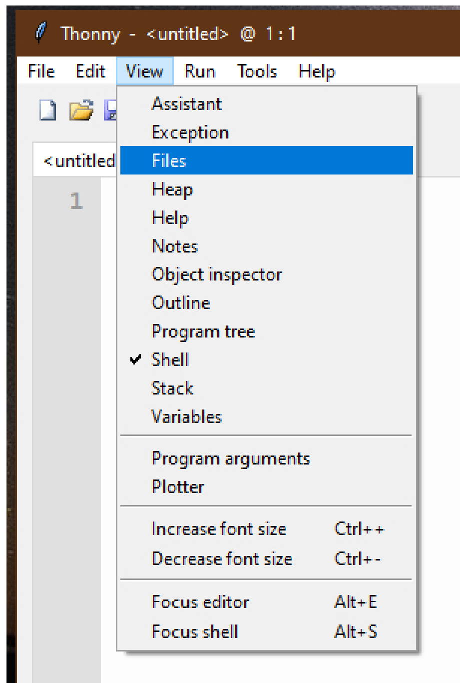
```
Shell × Program tree ×
I (602) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (608) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (614) heap_init: At 4009DE28 len 000021D8 (8 KiB): IRAM
I (621) cpu_start: Pro cpu start user code
I (304) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
MicroPython v1.13 on 2020-09-02; ESP32 module with ESP32
Type "help()" for more information.
>>> print("Hallo Welt")
Hallo Welt
>>>
```

Anders als in der Arduino-IDE können Sie einzelne Befehle an den ESP32 senden und er wird, so es MicroPython-Anweisungen sind, brav antworten. Senden Sie dagegen einen für den MicroPython-Interpreter unverständlichen Text, wird er sie mit einer Fehlermeldung darauf aufmerksam machen.

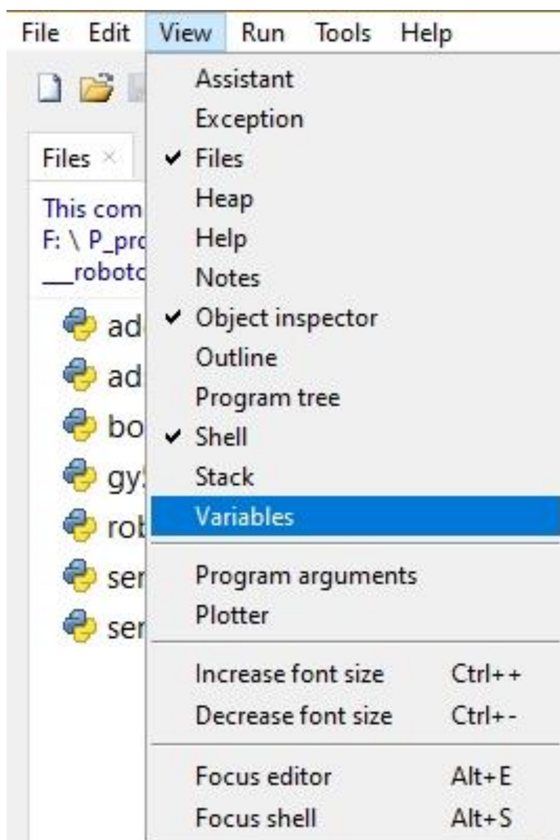
```
>>> print"hallo nochmal"

SyntaxError: invalid syntax
Traceback (most recent call last):
  File "<stdin>", line 1
SyntaxError: invalid syntax
```

Zum Arbeiten fehlt jetzt aber noch die Übersicht über den Workspace und das Device Directory. Der Workspace ist ein Verzeichnis auf dem PC, in dem sich alle für ein Projekt wichtigen Dateien befinden. In Thonny ist sein Name **This Computer**. Das Device Directory ist dazu das Gegenstück auf dem ESP32. In Thonny heißt es **MicroPython device.** Sie bringen es folgendermaßen zur Anzeige.

Klicken Sie auf **View** und dann auf **Files**



Jetzt werden beide Bereiche, oben der Workspace und unten das Device Directory, angezeigt. Weitere Tools blenden Sie über das Menü **View** ein bzw. aus.



Die Handsteuerung zum Robot Car

Die grundlegenden Kenntnisse zum Fahrzeug wurden bereits in vorangegangenen Beiträgen zum Thema Robot Car vermittelt, zum Beispiel in "Fernbedienungen mit Joystick". Auch eine Einführung in den Umgang mit der Sprache MicroPython hat es schon gegeben, Projekte mit MicroPython und dem ESP8266/ESP32 - Teil 1 bis 5. Aber es ist interessant, beide Welten miteinander zu vereinen. In diesem Projekt treten wieder die Vorteile von MicroPython als Interpreterlösung zum einen und der große Flash- und RAM-Speicher des ESP32 zusammen mit der WiFi-Option zum anderen stark in den Vordergrund. Bei der Erforschung der Geheimnisse von Hard- und Software kommt mir aber doch immer wieder ein Satz aus einem Lied von Kate Wolf in den Sinn: "In China and a womans heart there are places no one knows". Das will heißen, es gibt immer etwas Neues in den Gefilden von MicroPython zu entdecken. Auf dem Weg dazu waren sehr viele kleine Schritte nötig, die meistens über REPL führten und sehr schnell umgesetzt und überprüft werden konnten, ohne dass ich dazu ein ganzes Programm erstellen und compilieren musste, wie das in der Arduino-IDE der Fall ist.

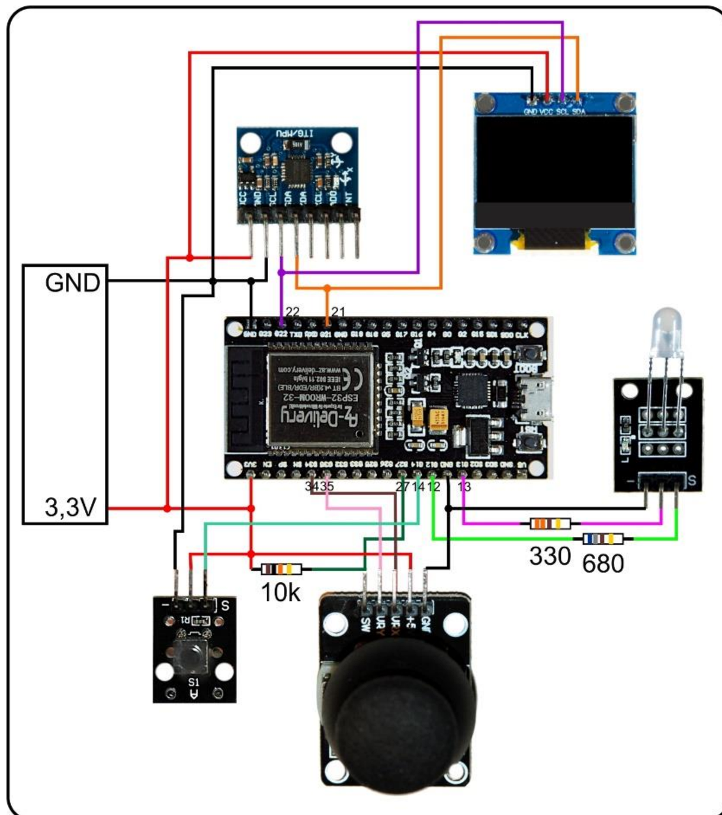
Die Lösung des gesamten Jobs,

- Erzeugung der Fahrstufen
- Normierung der Fahrstufen auf der Basis von verschiedenen Hardwarekomponenten wie ADC-Wandlung oder Beschleunigungsmessung
- Kommunikation mit dem Programm über möglichst wenige Tasten
- optische Aufbereitung der Rückmeldungen und
- die Funkübertragung zum Mobil

machten eine Aufteilung des gesamten Steuerprogramms in einzelne Module und Programmteile nötig. So entstanden aus Vorhandenem durch Verfeinerung neue, bessere Klassen, die allesamt einzeln getestet werden konnten, vielfach direkt über REPL, die Kommandozeile von MicroPython am PC. Die Module erfüllen die Aufgabe von Libraries in der Arduino-IDE.

Ursprünglich dachte ich, mit einer zweifarbigen LED als Signal für Rückmeldungen auszukommen. Eine RGB-LED schied aus, weil am ESP8266, den ich zu Beginn für die Steuerung vorgesehen hatte, kein Anschluss für die dritte LED mehr frei war. Also wurde ein OLED-Display am I2C-Bus in Dienst gestellt, das Rückmeldungen im Klartext ermöglicht – welch ein Luxus! Aber es lief prima. – Bis Murphy zuschlug.

Als die einzelnen Module und die beiden Programmteile fertig waren und für sich gesehen funktionierten, musste ich enttäuscht feststellen, dass der SRAM des ESP8266 bei weitem nicht ausreichte. Also blieb nur der Umstieg auf den größeren Bruder ESP32. Die Schaltung entstand auf zwei gekoppelten Minibreadboards, auf die auch alle anderen Teile außer Batterie und Joystick passten. Sie können den Verdrahtungsplan des Senders auch als PDF downloaden.



Folgende Module haben spezielle Aufgaben im Steuerprogramm, das aus den beiden Dateien boot_sender.py und sender.py besteht. Der Inhalt von boot_sender.py wir, wenn alles funktioniert, in die Datei boot.py kopiert, damit der ESP32 autonom starten kann.

adcrc.py	Liefert die Rohdaten eines Joysticmoduls, das an zwei konfigurierbare Analogpins des ESP32 angeschlossen ist.
ads1115rc.py	Liefert die Rohdaten eines Joystickmoduls, das an ein ADS1115-Modul via I2C angeschlossen ist.
gy521rc.py	Das Accelerometer-Modul liefert an den ESP32/ESP8266 via I2C Neigungsdaten
beep.py	Das Modul ist zuständig für akustische und visuelle LED-Signale als Rückmeldung von Systemprozessen. Außerdem kann es einen zeitlich verzögerten IRQ-Prozess starten.
button.py	Das ist der Nachfolger der Module touch.py, touchx.py und touch8266.py Das Modul stellt die überarbeiteten Dienste für die Bedienung von Tasten bereit. Es enthält die Klassen BUTTONS, BUTTON32 und BUTTON8266.
oled.py	Stellt Methoden für die positionierte Ausgabe von Daten auf einem OLED-Display zur Verfügung. Die Eingaben unterliegen einer Plasibilitätskontrolle bezüglich Position und Textlänge.
ssd1306.py	Enthält die Basisumgebung für Module mit dem SSD1306-I2C-Processor.
robotcar.py	Normiert die Rohdaten von den Eingabemodulen adcrc.py, ads1115rc.py und gy521rc.py auf Fahrstufen bezüglich des Mittelwerts und der sich daraus ergebenden Bereichswerte. Die Anzahl der Fahrstufen sowie die Ausblendung um die Fahrstufe 0 sind durch Parameter variabel.

sender.py	Übernimmt die Ergebnisse der Verbindungsaufnahme von boot_sender.py während der Testphase. Im Produktionsbetrieb erfolgt der Aufruf aus der Datei boot.py.
boot_sender.py	Enthält alle Einstellungen und Vorgaben zum Aufbau einer WiFi-Verbindung zu einem Accesspoint im lokalen Netz oder zur direkten Verbindung zum Accesspoint des Servers auf dem Robot Car. Der Inhalt von boot_sender.py muss für den autonomen Start des Senders in dessenDatei boot.py hineinkopiert werden.

Den wesentlichen Teil der drei Klassen adcrc.ADC32, ads1115rc.ADS1115 und gy521rc.GY521 habe ich im Folgenden für die Klasse ADC32 aufgelistet.

Download adcrc.py

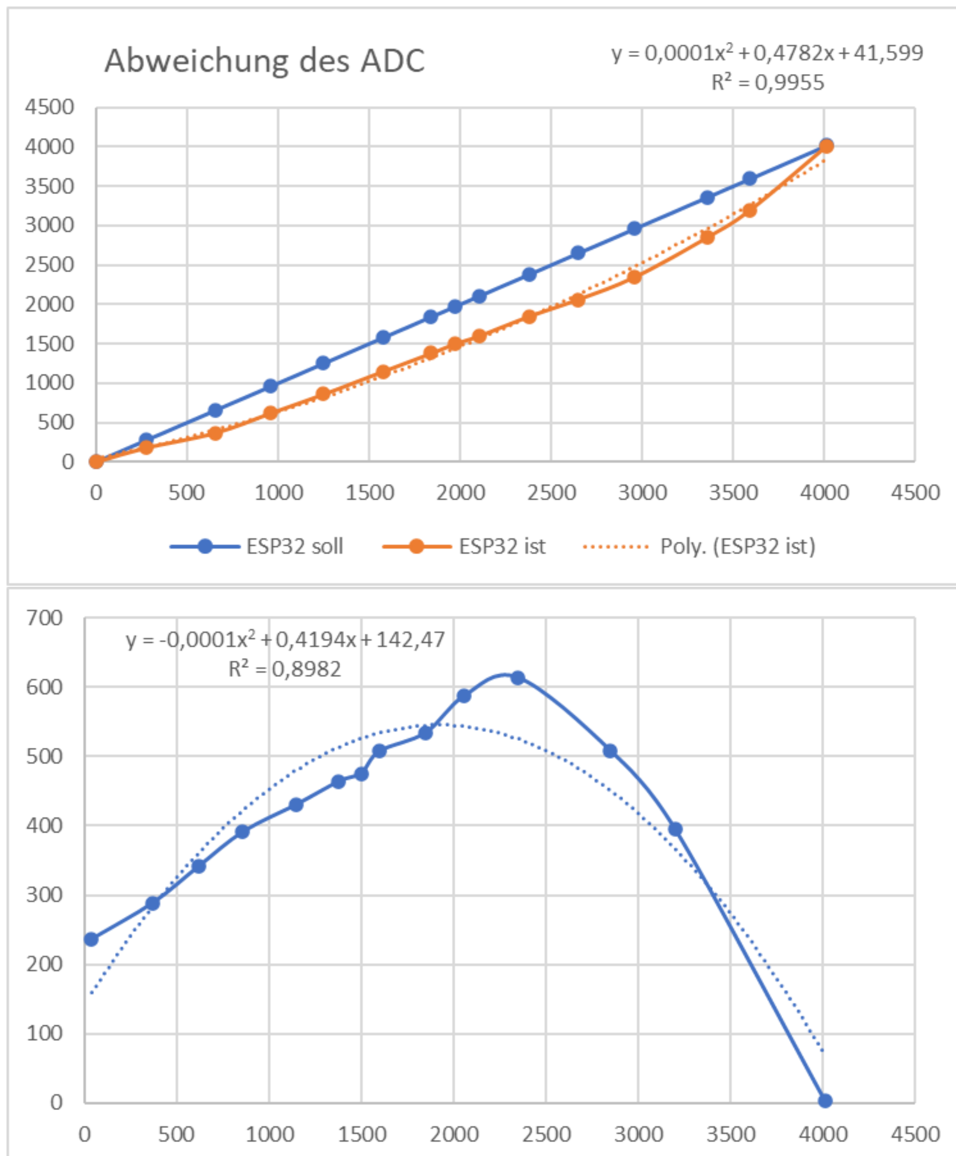
```
class ADC32:

    def __init__(self,i2c,xChannel=34,yChannel=35):
        self.cx=ADC(Pin(xChannel))
        self.cx.atten(ADC.ATTN_11DB)
        self.cx.width(ADC.WIDTH_12BIT)
        self.cy=ADC(Pin(yChannel))
        self.cy.atten(ADC.ATTN_11DB)
        self.cy.width(ADC.WIDTH_12BIT)
        self.FILE="adc.ini"
        print("Construktor: ADC intern ESP32")
        print("FILE=",self.FILE)

    def getXY(self):
        a=-0.000185
        b=1.7
        wert=[0,0]
        x=0
        for i in range(4): x+=self.cx.read()
        x=x//3
        y=0
        for i in range(4): y+=self.cy.read()
        y=y//3
        wert[0]=int(a*x*x+b*x)
        wert[1]=-int(a*y*y+b*y)
        return wert
```

Der Constructor braucht nicht wirklich einen I2C-Bus. Die Angabe ist lediglich aus Kompatibilitätsgründen zu den anderen Quellen der Richtungserfassung notwendig. Der Rest der Constructoranweisungen befasst sich nur mit dem ESP32-eigenen ADC.

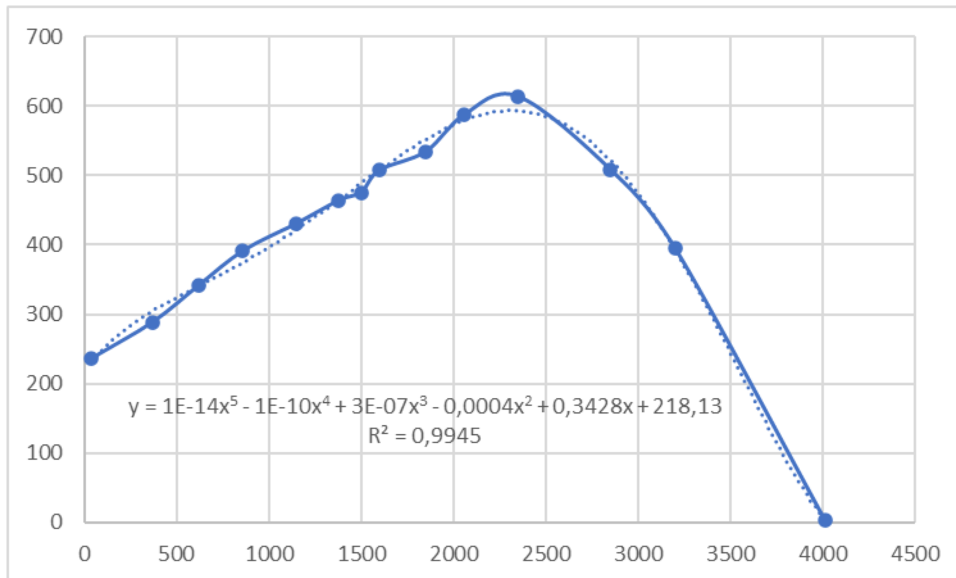
Wichtig ist die Methode **getXY**(), die keinen Parameter nimmt. Sie ist in allen drei Klassen mit gleicher API enthalten. Daher können die drei Module/Klassen gegeneinander problemlos ausgetauscht werden. Was sonst innerhalb der Klassen passiert, ist für den Datenexport uninteressant.

Für den ESP32 muss man etwas ausholen. Der ADC des ESP32 ist nicht gerade mit Genauigkeit gesegnet. Die Kennlinie "Counts gegen Eingangsspannung" ist alles andere als linear. Ganz grob angenähert, kann man eine quadratische Abweichung des Messfehlers annehmen. Besser käme man mit einer Korrekturfunktion 5.

Ordnung hin, aber das sprengt die zeitlichen Grenzen der Anwendbarkeit für dieses Projekt. Die Abweichung (orange) vom Sollwert (blau) ist über den gesamten Messbereich erheblich. Für die Korrektur werden die Differenzen Soll – Ist berechnet. und grafisch dargestellt. Man passt eine (möglichst einfache) Trendlinie an und lässt die Formel angeben. Mit Hilfe der Formel kann man dann zu beliebigen Messwerten die Abweichung berechnen und zum Messwert addieren.



In dieser Grafik sind die Differenzwerte in Abhängigkeit der Rohwerte vom ADC aufgetragen. Die Trendlinie 2. Ordnung ist nicht optimal, lindert aber dennoch die Messfehler des ADC. Besser wäre die folgende Anpassung, aber werfen Sie mal einen Blick auf denFunktionsterm! Der Korrelationskoeffizient R^2 ist natürlich mit 0,9945 schon fast ideal. Dennoch, wir bleiben bei der quadratischen Funktion.



Die Spannungswerte des Joysticks werden erst einmal durch die Mittelwertberechnung aus 4 Messungen geglättet. Daraufhin werden die Rohwerte durch die Korrekturfunktion zweiter Ordnung geschickt, deren Parameter a und b durch ein Kalkulationsblatt ermittelt wurden. Diese Werte sind exemplarbedingt von Modul zu Modul abweichend und sollten daher jeweils selbst ermittelt werden. Außerdem ist es sinnvoll die Parameter a und b manuell noch etwas zu modellieren, bis der Ist-Graph dem Soll-Graphen am besten entspricht.

Wenn eine höhere Genauigkeit gewünscht ist, empfehle ich die Verwendung des ADC-Moduls ADS1115. Die Auflösung ist 16-Bit gegen 12-Bit beim ESP32. Das Modul wäre auch am ESP8266 einsetzbar. Angesprochen wird das Modul über I2C. Nachdem der I2C-Bus ebenfalls die Module OLED und GY521 bedienen muss, ist der ADS1115 also eine gute Option.

Den I2C-Bus verwenden bis zu drei Module im Projekt. Daher ist es sinnvoll, die Definition für den Bus nur einmal im Startprogramm (letztlich boot.py) vorzunehmen, statt dezentral in den einzelnen Modulen. Die Module bekommen das I2C-Objekt vom Hauptteil als Parameter oder in der globalen Umgebung überreicht (von boot.py an server.py). Analog ist die Verwendung von BEEP und OLED geregelt, einmal declarieren und mehrfach verwenden.

Die Klassen ADS1115 und GY521 haben zwar aufgrund der internen Struktur einen teils erheblichen Overhead an Systemkonstanten und Methoden, bieten aber einen normierten Zugriff auf die Rohdaten für die Positionsauswertung, eben über die Methode getXY().

Die Klasse BUTTONS stellt Methoden für die Behandlung von Tasten-Instanzen zur Verfügung, die mittels BUTON32 und BUTTON8266 erzeugt werden können. (Für Touchpad-Objekte wird in Kürze eine vergleichbare Klasse zur Verfügung stehen.)

BEEP und OLED sind Klassen, die Informationen vom System an den Benutzer weitergeben. Die Dokumentation in den Modulen informiert über deren Verwendung.

Die Klasse robotcar.RC erbt von der jeweils entkommentierten Klasse im Kopfbereich von robotcar.py. Sie ist das Kernstück der Steuerung und dafür

zuständig, dass aus den Rohdaten der Sensoren brauchbare Fahrstufenwerte werden.

Der erste Schritt dazu ist die Ermittlung der horizontalen Ruhewerte des Sensors in x-(vor-zurück)- und y-(rechts-links)-Richtung. Dann können die Maximal- und Minimalwerte in den Hauptrichtungen ermittelt werden und schließlich die sich ergebenden Bereiche von der Ruhestellung bis Maximum oder Minimum. Das alles passiert während der Kalibrierung der Steuerung, die einmal beim allerersten Start oder wiederholt auf Tastendruck bei erneuten Starts durchgeführt wird. Die Kalibrierungswerte sind bauteilbedingt, hängen aber auch von der Betriebsspannung ab, die sich im Lauf der Benutzung durch die Entladung der Batterien verändern kann. Nach erfolgter Kalibrierung werden die Werte in einer ini-Datei im Dateisystem des ESP32 gespeichert, deren Name von der Art des Sensors abhängt. Beim Neustart wird versucht, diese Datei zu lesen. Gelingt das ohne Fehler, dann werden die enthaltenen Werte für die folgende "Sitzung" benutzt. Andernfalls muss neu kalibriert werden. Während der Kalibrierung wird über das OLED angezeigt, wie der Sensor zu neigen ist. (Luxus ist so toll!)

Die Rohwerte von den Sensoren schwanken teils erheblich. Durch die Einführung von Fahrstufen wird dieser Effekt unterschiedlich gut gemildert. Dennoch werden auch die Werte der Fahrstufen dadurch beeinflusst. Damit im Bereich um die Mittelwerte kein Flattern des Fahrzeugs entsteht, kann der Ruhebereich durch die Angabe der Parameter der Methode getSpeedLevel() eingestellt werden. Ein größerer Wert sorgt für mehr Ruhe um die Nulllage. Auch die Anzahl der Fahrstufen kann im Programm eingestellt werden. Probieren Sie aus, was in Ihrem Fall optimal ist.

```
# File: robotcar.py
# Rev:  1.1  - simplified API
# Date: 06-03-2021
# Author: J. Grzesina
# **************************************************************
from time import time,sleep
#from ads1115rc import ADS1115 as RC32
#from gy521rc import GY521 as RC32
from adcrc import ADC32 as RC32

class RC(RC32):
    def __init__(self,i2c,d=None):
        self.i2c=i2c
        self.mx=0
        self.my=0
        self.minX=0
        self.maxX=0
        self.minY=0
        self.maxY=0
        self.divXback=1
        self.divXahead=1
        self.divYleft=1
        self.divYright=1
        self.d=d
        super().__init__(self.i2c)
```

```
    def getMeanValues(self,delay):
        #print("horizontale Position")
        if self.d:
            self.d.clearAll()
            self.d.writeAt("HORIZONTAL",0,0)
        sx=0; sy=0
        start=time()
        current=start
        end=start+delay
        n=0
        while current<=end:
            x,y=self.getXY()
            n+=1
            sx+=x; sy+=y
            sleep(0.1)
            current=time()
        return(sx//n,sy//n)

    def getMaxX(self,delay):
        #print("x zeigt nach oben")
        if self.d:
            self.d.clearAll()
            self.d.writeAt("HINTEN AB",0,0)
        start=time()
        current=start
        end=start+delay
        m=0
        while current<=end:
            a=self.getXY()[0]
            if a>m: m=a
            current=time()
        return m

    def getMinX(self,delay):
        #print("x zeigt nach unten")
        if self.d:
            self.d.clearAll()
            self.d.writeAt("VORNE AB",0,0)
        start=time()
        current=start
        end=start+delay
        m=0
        while current<=end:
            a=self.getXY()[0]
            if a<m: m=a
            current=time()
        return m

    def getMaxY(self,delay):
        #print("y zeigt nach oben (rechts ab)")
        if self.d:
```

```
            self.d.clearAll()
            self.d.writeAt("RECHTS AB",0,0)
        start=time()
        current=start
        end=start+delay
        m=0
        while current<=end:
            a=self.getXY()[1]
            if a>m: m=a
            current=time()
        return m

    def getMinY(self,delay):
        #print("y zeigt nach unten (links ab)")
        if self.d:
            self.d.clearAll()
            self.d.writeAt("LINKS AB",0,0)
        start=time()
        current=start
        end=start+delay
        m=0
        while current<=end:
            a=self.getXY()[1]
            if a<m: m=a
            current=time()
        return m

    def calibrate(self,duration):
        if self.d:
            self.d.clearAll()
            self.d.writeAt("CALIBRATE NOW",0,0)
            self.d.writeAt("DURATION {}s \
            EACH".format(duration),0,1)
            sleep(3)
        self.mx,self.my=self.getMeanValues(duration)
        self.maxX=self.getMaxX(duration)
        self.minX=self.getMinX(duration)
        self.maxY=self.getMaxY(duration)
        self.minY=self.getMinY(duration)
        D=open(self.FILE,"wt")
        D.write(str(self.mx)+"\n")
        D.write(str(self.my)+"\n")
        D.write(str(self.minX)+"\n")
        D.write(str(self.maxX)+"\n")
        D.write(str(self.minY)+"\n")
        D.write(str(self.maxY)+"\n")
        D.close()
        if self.d:
            self.d.writeAt("CALIBR. DONE",0,0)
            sleep(3)
        return(self.mx,self.my,self.minX,\
        self.maxX,self.minY,self.maxY)
```

```
    def readCalibration(self):
        D=open(self.FILE,"rt")
        self.mx=int(D.readline())
        self.my=int(D.readline())
        self.minX=int(D.readline()) # ahead
        self.maxX=int(D.readline()) # back
        # y-values are brought inverted by getXY()
        #to meet x-values behavior
        self.minY=int(D.readline()) # left
        self.maxY=int(D.readline()) # right
        D.close()
        if self.d:
            self.d.writeAt("CALIBRBRATION",0,2)
            self.d.writeAt("READ FROM FILE",0,3)
            sleep(3)
        return(self.mx,self.my,self.minX,\
        self.maxX,self.minY,self.maxY)

    def calculateAreas(self,fahrstufen=15):
        self.divXback=int(((self.maxX-self.mx)//fahrstufen)+1)
        self.divXahead=int(((self.mx-self.minX)/fahrstufen)+1)
        self.divYleft=int(((self.my-self.minY)/fahrstufen)+1)
        self.divYright=int(((self.maxY-self.my)/fahrstufen)+1)
        return (self.divXback,self.divXahead,\
        self.divYleft,self.divYright)

    def getSpeedLevel(self,excludeX=3,excludeY=3):
        x,y=(self.getXY())
        x=(x if x<=self.maxX else self.maxX)
        x=(x if x>=self.minX else self.minX)
        y=(y if y<=self.maxY else self.maxY)
        y=(y if y>=self.minY else self.minY)
        dx=x-self.mx; dy=y-self.my
        fx=(-dx//self.divXback if dx >= 0 \
        else -dx//self.divXahead)
        fy=(dy//self.divYright if dy >= 0 \
        else dy//self.divYleft)
        fx=(fx if abs(fx)>=excludeX else 0)
        fy=(fy if abs(fy)>=excludeY else 0)
        return (fx,fy)
```

Das Programm boot_sender.py richtet die Basis für den darauf aufbauenden Teil server.py ein. Nach der Definition verschiedener Variablen und Objekte wird versucht, entweder eine Verbindung mit dem lokalen WLAN-Router oder direkt mit dem Accesspoint des Servers auf dem Robot Car aufzubauen. Die Auswahl findet über eine der Tasten statt. Das OLED informiert über die Anforderung.

In diesem Bereich müssen Sie die Zugangsdaten für Ihre private Umgebung verfügbar haben. Der WLAN-Router wird SSID und Passwort verlangen, beim

Accesspoint des ESP32 genügt die SSID, ein Passwort wird hier bei der Anmeldung nicht erwartet.

Ist die Verbindung aufgebaut, teilt das OLED-Display die Verbindungsdaten mit. Am Ende des Boot-Teils besteht die Möglichkeit, das Programm abzubrechen. Diese "Sollbruchstelle" hat sich während der Programmentwicklung als sehr nützlich erwiesen. Auch die Zweiteilung der gesamten Anwendung ist taktisch geschickt, weil man bei Manipulationen am Server-Teil nicht jedes Mal den Boot-Teil mit Anmeldung etc. durchlaufen muss. Außerdem lässt sich der Boot-Teil auch auf andere WiFi-Anwendungen portieren. Der Boot-Teil ist also quasi ein Modul ohne eigene Klasse.

```
# File: boot.py
# Purpose: booting robot car sender/server
# Author:  J. Grzesina
#
#*************** Beginn Bootsequenz ***********************
# Dieser Teil geht an den Anfang von boot.py
#***************** Importgeschaeft ************************
# Dieser Teil wird beim Einsatz von boot.py erledigt.
import os,sys
from time import time,sleep, sleep_ms, ticks_ms

from machine import Pin,I2C

import esp
esp.osdebug(None)

import gc           # Platz fuer Variablen schaffen
gc.collect()
#
# *****************  wifi_connect   \*********************
# Dieser Teil verbindet mit einem WLAN-Accesspoint
#
# File: wifi_connect.py
# Rev.: robot car 1.1
# Date: 2021-03-10
# Author: Jürgen Grzesina (krs@grzesina.eu)
#
#**************Variablen deklarieren *********************
# Die Dictionarystruktur (dict) erlaubt spaeter die
Klartextausgabe
# des Verbindungsstatus anstelle der Zahlencodes
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "GOT_IP"
    }

#*************** Funktionen deklarieren ******************
```

```
def hexMac(byteMac):
  """
  Die Funktion hexMAC nimmt die MAC-Adresse
  im Bytecode entgegen und
  bildet daraus einen String fuer die Rueckgabe
  """
  macString =""
  for i in range(0,len(byteMac)):     # Fuer alle Bytewerte
    # vom String ab Position 2 bis Ende
    macString += hex(byteMac[i])[2:]
    # Trennzeichen bis auf das letzte Byte
    if i <len(byteMac)-1 :
      macString +="-"
  return macString

# ---------------------------------------------------------
# *********** create essential objects ******************
# ---------------------------------------------------------
#
# Pintranslator für ESP8266-Boards
# LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
# ESP8266 Pins 16  5  4  0  2 14 12 13 15
#                 SC SD  FL L
#
# ----------
SD =    21
SC =    22
i2c=I2C(-1, scl=Pin(SC), sda=Pin(SD))

rot=13
gruen=12
blau=None
from beep import BEEP
b=BEEP(None,rot,gruen,blau,200)

from button import BUTTONS,BUTTON32
LichtPin=14
MotorPin=27
t=BUTTONS()
licht=BUTTON32(LichtPin,True,"LICHT")
motor=BUTTON32(MotorPin,True,"MOTOR")
abbruchtaste=licht
Ltimeout=100
from oled import OLED
d=OLED(i2c,128,64)
#from display import LCD
# d=LCD(i2c)
# *************** Get connected ************************
# Netzwerk-Interface-Instanz erzeugen und ESP32-Stationmodus
aktivieren;
# moeglich sind network.STA_IF und network.AP_IF beide
gleichzeitig,
```

```
# wie in LUA oder AT-based ist in MicroPython nicht moeglich
# Create network interface instance and activate station mode;
# network.STA_IF and network.AP_IF,both at the same time,
# as in LUA or AT-based or Adruino-IDE is not possible in
MicroPython
import ubinascii
import network

#request = bytearray(100)
#act=bytearray(10)
nic = network.WLAN(network.STA_IF)  # erzeuge WiFi-Objekt nic
nic.active(True)  # Objekt nic einschalten
#
MAC = nic.config('mac')    # binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)          # in eine Hexziffernfolge umgewandelt
print("STATION MAC: \t"+myMac+"\n") # ausgeben
# Verbindung mit AP aufnehmen, falls noch nicht verbunden
# connect to WLAN-AP or robot car directly
e=t.jaNein(motor,licht,"CONNECT over WLAN?",d=d,b=b,laufZeit=4)
d.clearAll()
ct=("10.0.1.199","255.255.255.0","10.0.1.20","10.0.1.100")
# Geben Sie hier Ihre eigenen Zugangsdaten an
mySid = "YOUR SIID"; myPass = "YOUR PASSWORD"
if e==t.JA:
    targetIP="10.0.1.101"
    targetPort=9000
elif e==t.NEIN:
    targetIP="10.0.2.101"
    targetPort=9000
    ct=("10.0.2.199","255.255.255.0","10.0.2.20","10.0.2.100")
    mySid = 'robotcar'; myPass = "NOT NEEDED"
else:
    targetIP="10.0.1.10"
    targetPort=9000
print(targetIP)
if not nic.isconnected():
  # Zum AP im lokalen Netz verbinden und Status anzeigen
  nic.connect(mySid, myPass)
  # warten bis die Verbindung zum Accesspoint steht
  #print("connection status: ", nic.isconnected())
  d.clearAll()
  d.writeAt("CONNECTING TO AP",0,0)
  d.writeAt(mySid,0,1)
  while not nic.isconnected():
    #pass
    print("{}.".format(nic.status()),end='')
    #sleep(1)
    if b: b.blink(1,0,0,500,anzahl=1) # while not connected
# Wenn bereits verbunden,
# zeige Verbindungsstatus und Config-Daten
#print("\nconnected: ",nic.isconnected())
#print("\nVerbindungsstatus: ",connectStatus[nic.status()])
```

```
nic.ifconfig(ct)
STAconf = nic.ifconfig()
print("STA-IP:\t\t",STAconf[0],"\nSTA-
NETMASK:\t",STAconf[1],"\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
#
# Write connection data to OLED-Display
d.clearAll()
d.writeAt("CONNECTED AS:",0,0)
d.writeAt(STAconf[0],0,1)
d.writeAt(STAconf[1],0,2)
d.writeAt(STAconf[2],0,3)
sleep(3)
#
# Write connection data to LCD
# ***************************************************************
# ****************** Abbruchoption **************************
# Falls gewuenscht Abbruch durch Tastendruck
if t.jaNein(motor,licht,meldung1="ABBRUCH?",d=d,b=b,laufZeit=5)
!= t.JA :
  # tpNein touched between 5 sec or
  # untouched at all start server
  if d: d.clearAll()
  exec(open('sender.py').read(),globals())
  #exec(open('sender1.py').read(),globals())
else: # falls das Pad an tpJa beruehrt wurde
  print("Die Bootsequenz wurde abgebrochen!")
  if d:
      d.clearAll()
      d.writeAt("ABGEBROCHEN",0,0)
```

Bisher ging es um notwendiges Zubrot für das Projekt Robot Car,. mit Ausnahme dessen, was ich oben als Kern bezeichnet habe, die Klasse robotcar.RC. Dieser Kern wird von der Datei sender.py ummantelt.

Wir importieren, wie immer, wenn es um Server oder Clients geht, die Klasse socket. Als nächstes folgt der Import der Klasse RC, benannt nicht nach **R**emote **C**ontrol sondern nach **R**obot **C**ar, obwohl Ersteres in diesem Fall auch zutreffend ist. Dem Objekt a, das wir von RC ableiten, geben wir das I2C-Objekt i2c und auch noch das OLED-Objekt d mit auf den Weg.

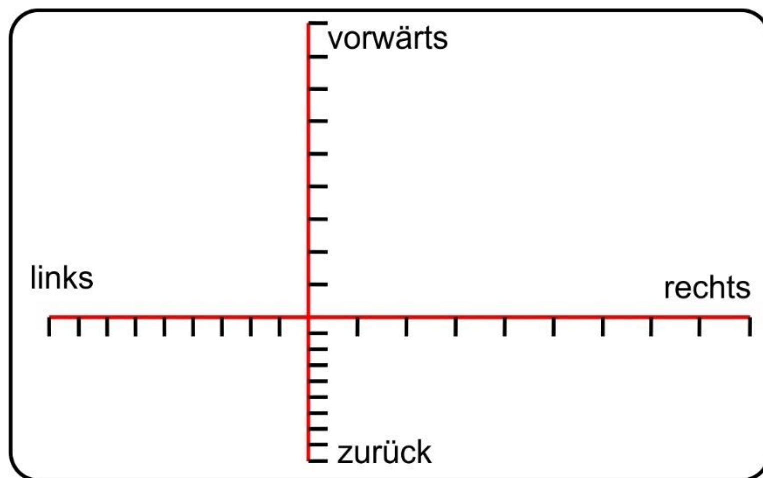
```
from robotcar import RC
# In robotcar.py you can decide which of the following modules to use
# Joystick @ ADS1115 more precise
# Joystick @ ESP32-ADC
# Accelerometer GY521
a=RC(i2c,d=d) # already declared in boot-section
```

Das Programm fragt, ob es eine erneute Kalibrierung durchführen soll. Wir beantworten die Frage mit Hilfe der linken Taste, die später das Licht am Fahrzeug schalten soll. Wird die Taste nicht gedrückt, versucht das Programm die Datei adc.ini zu lesen. Wenn das Programm das allererste Mal gestartet wird, tritt hier mit Sicherheit ein Fehler auf, die Datei existiert ja noch nicht. Der Fehler wird von der

except-Anweisung aufgefangen, der Fehlertext wird an der Konsole ausgegeben und die Kalibrierung wird aufgerufen. Was Sie dabei tun müssen, sagt Ihnen das OLED-Display. Nach jeder Kalibrierung wird das Ergebnis in die ini-Datei geschrieben. Für welche Sensorklasse das passieren soll, erfährt RC.calibrate() von dem Instanzattribut self.FILE, das von der Sensorklasse (hier adcrc.ADC32) an robotcar.RC vererbt wird. Klingt aufwendig, ist aber sehr praktisch. Außerdem wird das Ergebnis in den Instanzattributen gespeichert und als Tupel an das aufrufende Programm zurückgegeben.

```
return(self.mx,self.my,self.minX,self.maxX,self.minY,self.maxY)
```

Nun legen wir die ungefähre Anzahl von Fahrstufen pro Richtung fest und lassen die Teiler für die verschiedenen Wertebereiche berechnen. Weil die Ruhelage meistens nicht die Mitte des gesamten Wertebereichs darstellt, fallen die Bereiche (rote Linien) nicht gleich groß aus. Will man gleich viele Unterteilungen haben, müssen die Teiler angepasst werden. Weil später am Fahrzeug die PWM-Werte mit der Fahrstufe als Index aus einer Liste geholt werden, darf der Index=Fahrstufenwert nicht größer als die Anzahl der Listeneinträge sein. Daher wird der Teiler als Integerwert aufgerundet. Das kann aber auch zur Folge haben, dass die Anzahl von Fahrstufen leicht unterhalb des Sollwerts liegt. Auch die Teiler werden wie die Maxima, Minima und Mittelwerte in Instanzattributen gespeichert und eigentlich nur zur Fehlersuche an das aufrufende Programm zurückgegeben. Die Zuweisungen im Hauptprogramm könnte man also getrost entfernen.



Die Entwicklung der diesbezüglichen Methoden hat sich wegen der Einbindung verschiedener Sensoren mit am längsten hingezogen.

Jetzt sind wir fast am Ende des Sender-Teils angekommen. Wir erzeugen eine Socket-Instanz s, binden diese an die von boot.py mitgebrachte IP-Adresse und weisen eine Portnummer zu. Dazu einige Worte zum Sendeprotokollstapel.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Der Sender verwendet das UDP-Protokoll (siehe: ***socket.SOCK_DGRAM)*** anstelle des sonst üblichen TCP-Protokolls (***socket.SOCK_STREAM***), das steht codiert im fett formatierten Text. Ich habe mich in diesem Fall für UDP entschieden, weil die Übertragung viel schneller läuft. Der Sender gibt einfach seine Botschaft an den Server am Robot Car weiter, ohne vorher eine Verbindung auszuhandeln. Die Verbindung ist daher auch ungesichert. Das bedeutet zweierlei, erstens ist nicht

sichergestellt, dass das Paket überhaupt beim Server ankommt und zweitens ist nicht sichergestellt, dass das Paket unversehrt ankommt. Das Erstere ist nicht schlimm, weil genügend weitere Pakete mit der gleichen Art von Inhalt folgen und das Zweite ist auch nicht schlimm, weil ein unverständliches Paket vom Server aussortiert wird. UDP ist also vergleichbar mit einer seriellen Schnittstelle, die Daten sendet, obwohl sie nicht weiß, ob überhaupt eine Gegenstation zuhört.

Im OLED-Display erscheinen die Verbindungsdaten, die grüne LED zeigt an, dass jetzt gefahren werden kann. Die Taste am Joystick schaltet das Motorrelais am Robot Car, die andere Taste das Licht mit jedem Drücken an oder aus. Wird die Motortaste etwa 5 Sekunden gedrückt, schaltet sich der Sender aus. Zum erneuten Start muss man die RST-Taste am ESP32 drücken oder ihn aus- und einschalten.

Wenn Sie das Modul gy521 mit dem Neigungssensor MPU6050 zur Steuerung verwenden, können Sie das Robot Car durch Neigen in die entsprechende Richtung beschleunigen, zurücksetzen oder nach links und rechts lenken. Wenn kein Joystick angeschlossen ist, müssen Sie dessen Taster durch ein Buttonmodul ersetzen, sonst können Sie den Motor nicht schalten. die Programmierung der Motortaste ist wegen der Doppelfunktion der aufwendigste Teil der Senderschleife. Neben den Tasten wird fortlaufend die Fahrstufe abgefragt. die beiden Parameter bestimmen, bis zu welchem Intervall statt der Fahrstufe eine Null zurückgegeben wird. Wenn man es genau betrachtet, besteht der Sender eigentlich nur aus den vier Zeilen, die mit s.sendto beginnen.

Das Übertragungsprotokoll ist sehr einfacher Natur.
v:5 bedeutet Fahrstufe 5 vorwärts, v steht für **v**elocity.
v:-4 heißt dann, mit Fahrstufe 4 rückwärts
d:6 wir fahren mit Stufe 6 nach rechts
d:-6 dasselbe nach links
l:1 (kleines L) licht an, aus, an, aus…
m:1 Motoren an, aus, an, aus …

Das lange Drücken der Motortaste wird nicht an das Robot Car übertragen, sondern manövriert nur die Steuerung in eine Endlosschleife. Das heißt, der Sender braucht auch noch einen Ein-Ausschalter zum Abklemmen von der Batterie. Der kann dann auch zum erneuten Kaltstart der Schaltung dienen.

Hier das Listing von sender.py:

```
# **************** Sender department ***********************
# DER PREPARE BLOCK WIRD NACH ABSCHLUSS DER TESTS
AUSKOMMENTIERT
# UND GEGF. NACH boot_sender.py KOPIERT UND/ODER GECANCELT
# ---------------    Prepare Test   -----------------------
"""
from machine import Pin,I2C
SD =    21
SC =    22
i2c=I2C(-1, scl=Pin(SC), sda=Pin(SD))

rot=13
```

```
gruen=12
blau=None
from beep import BEEP
b=BEEP(None,rot,gruen,blau,200)

from button import BUTTONS,BUTTON32
LichtPin=14
MotorPin=27
t=BUTTONS() # Abbruchtaste, keine zweite Taste, BEEP-Pbj., kein
Display
licht=BUTTON32(LichtPin,True,"LICHT")
motor=BUTTON32(MotorPin,True,"MOTOR")
abbruchtaste=licht
Ltimeout=100
from oled import OLED
d=OLED(i2c,128,64)
#from display import LCD
# d=LCD(i2c)
"""
# ----------------    Prepare block end   -------------------
# --------------------    Objects    ------------------------
try:
  import usocket as socket
except:
  import socket

from robotcar import RC
# In robotcar.py you can decide which
# of the following modules to use
# Joystick @ ADS1115 more precise
# Joystick @ ESP32-ADC
# Accelerometer GY521
a=RC(i2c,d=d) # already declared in boot-section

b.ledOn(1,0,0)
d.writeAt("<<<<RECALIBRATE?",0,1)
c=t.waitForTouch(licht,3)
print("taste",c)
b.ledOn(1,1,0)
if c:
    mx,my,minX,maxX,minY,maxY=a.calibrate(3)
else:
    try:
        mx,my,minX,maxX,minY,maxY=a.readCalibration()
    except Exception as e:
        print("Fehler:",e.args)
        mx,my,minX,maxX,minY,maxY=a.calibrate(3)
b.ledOff()

fahrstufen=25
divXback,divXahead,divYleft,divYright= \
a.calculateAreas(fahrstufen)
```

```
#print("X:{} bis {}; y:{} bis {}".format(minX,maxX,minY,maxY))
#print("meanX:{}; meanY:{}".format(mx,my))
#print("ahead:{}; back:{}; right:{};
left:{}".format(divXback,divXahead,divYright,divYleft))
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', 9181)) # IP comes from boot-section
d.clearAll()
d.writeAt("Sock established",0,2)
d.writeAt("TARGET IS AT:",0,3)
d.writeAt("{}:{}".format(targetIP,targetPort),0,4)
print("Socket estblished, waiting...")
receiver=(targetIP,targetPort) # Address has to be a tuple
t.waitForTouch(abbruchtaste,2)

b.ledOn(0,1,0)
#sys.exit()
Mstate=0 #
Lstate=0 # Reset lightstatus
senderStop=0
gc.collect()
while 1:
    if Mstate>0:
        Mstate -= 1
    else:
        if t.getTouch(motor):
            Mstate=Ltimeout
            if senderStop==0:
                s.sendto("m:1\n",receiver)
            senderStop +=1
            d.writeAt("SHUT DOWN @ 5:
{}".format(senderStop),0,5)
            if senderStop==5:
                if b: b.ledOff()
                if d:
                    d.clearAll()
                    d.writeAt(" SENDER  STOPED",0,2)
                    d.writeAt("RESET TO RESTART",0,3)
                    s.close()
                while 1:
                    pass
        else:
            senderStop=0
    #fx,fy=a.getXY()
    fx,fy=a.getSpeedLevel(2,2)
    x="v:"+str(fx)+"\n"
    y="d:"+str(fy)+"\n"
    #print(x,y)
    #gc.collect()
    s.sendto(x,receiver)
    s.sendto(y,receiver)
    gc.collect()
```

```
    if Lstate >0:
        Lstate -=1
    else:
        if t.getTouch(licht):
            Lstate=Ltimeout
            #print("l:1")
            s.sendto("l:1\n",receiver)
    #sleep(0.2)
```

Woher wissen Sie denn jetzt, nachdem Sie alles hübsch aufgebaut und programmiert haben, dass die Sause auch wirklich ab geht? Methode1 ist, ersetzen Sie die Sendebefehle durch print-Anweisungen etwa in der Form:

```
#s.sendto(x,receiver)
#s.sendto(y,receiver)
print(x,y)
```

Dann wird all das, was sonst gesendet wird, auf der Konsole ausgegeben. Der ESP32 muss dazu natürlich über das USB-Kabel mit dem PC verbunden sein. Erscheint der erwartete Text, können Sie schon mal sicher sein, dass die Sensoren und die Datenaufbereitung korrekt arbeiten. Aber die Information über den Übertragungsweg schlummert noch im Verborgenen.

Machen Sie zum Aufwecken ihren PC zum UDP-Server. Mit ncat geht das. Laden Sie die Freeware herunter und installieren Sie diese. Im Installationsverzeichnis nmap finden Sie die Datei ncat.exe. Öffnen Sie eine Powershell, wechseln Sie in das Installationsverzeichnis und rufen Sie die Datei ncat folgendermaßen auf:

ncat -vv -l 10.0.1.10 9000 -u

Starten Sie jetzt den Sender im gleichen Teilnetz und mit dem hier angegebenen Zielport (9000). Wenn die Ausgabe im ncat-Fenster genauso aussieht wie zuvor im Terminal haben Sie gesiegt.

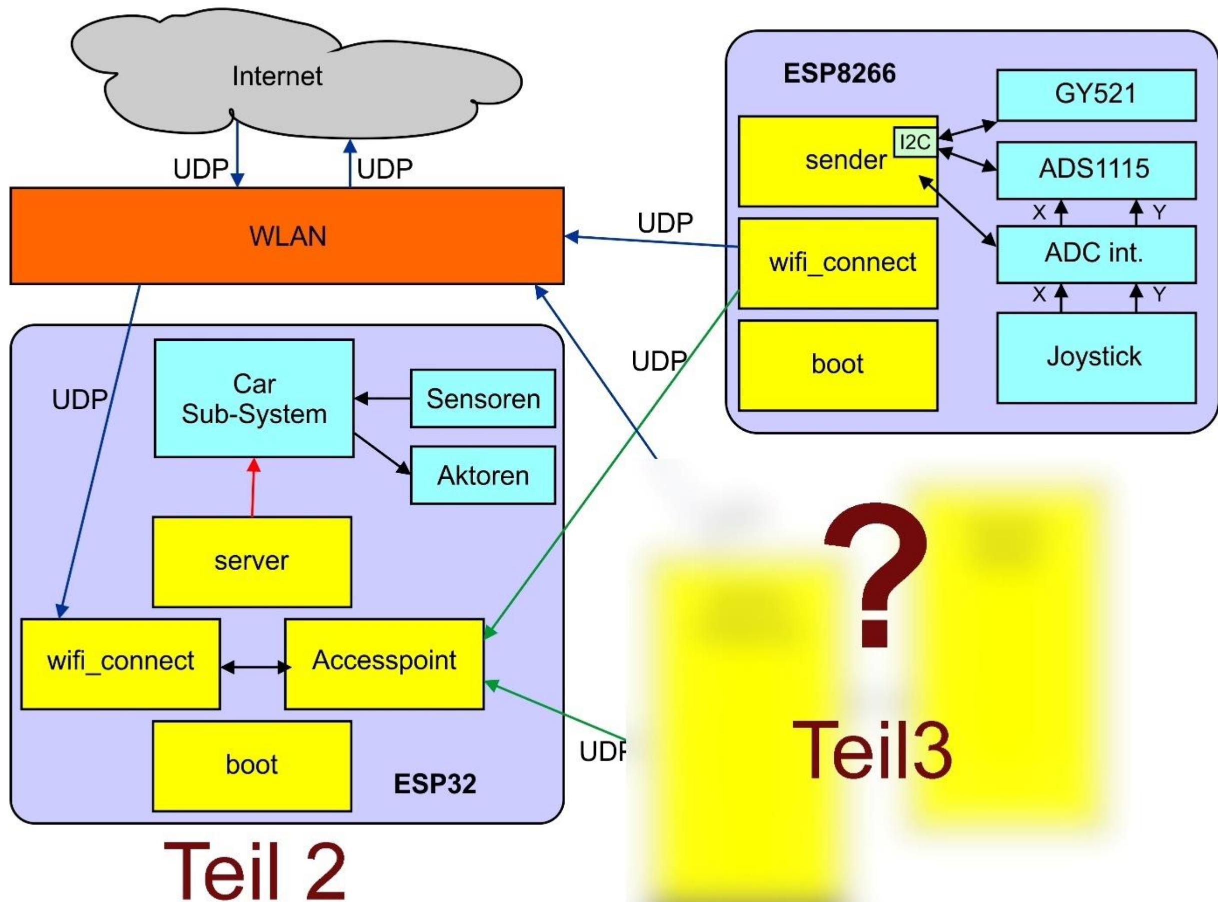
```
C:\WINDOWS\system32\cmd.exe

F:\P_programmieren\az-blog\___robotcar>c:

C:\>cd C:\"Program Files (x86)"\Nmap

C:\Program Files (x86)\Nmap>ncat -vv -l 10.0.1.10 9000 -u
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on 10.0.1.10:9000
Ncat: Connection from 10.0.1.199.
d:-3
v:-3
d:-3
v:-3
d:-3
v:-3
d:-3
v:-3
d:-3
```

Tipp:
Um schnell eine Powershell im Installationsverzeichnis zu öffnen, suchen Sie im Explorer mit ein paar Klicks das Verzeichnis auf. Rechtsklicken Sie jetzt mit gedrückter Shift-Taste auf den Verzeichniseintrag und wählen Sie aus dem Kontextmenü "Powershell-Fenster hier öffnen".

Wenn Sie das Programm öfter einsetzen wollen, empfiehlt sich die Erstellung einer Batchdatei mit dem Pfadwechsel und dem Inhalt des Aufrufs in einem Editor Ihrer Wahl, zum Beispiel mit Thonny. Geben Sie den Text ein und speichern Sie mit einem beliebigen Namen, Ergänzung ".bat" in ein beliebiges Verzeichnis. Ein Doppelklick auf die Datei startet ncat in einem DOS-Fenster.

```
c:
cd C:\"Program Files (x86)"\Nmap
ncat -vv -l 10.0.1.10 9000 -u
```

So, als Vorschau auf den nächsten Teil habe ich zum Schluss eine Grafik, die das Baukastensystem dieses Projekts aufzeigt.



Im Kasten zum ersten Teil, rechts oben, ist noch der ESP8266 angegeben. Schade, dass das mit dem nicht funktioniert hat. Freuen Sie sich trotzdem auf den 2. Teil. Sie können sich ja schon mal ein Fahrzeug aussuchen. Bei meinem fehlt noch ein Abstandssensor und natürlich die Verdrahtung von ESP32 und Motortreiberboard.