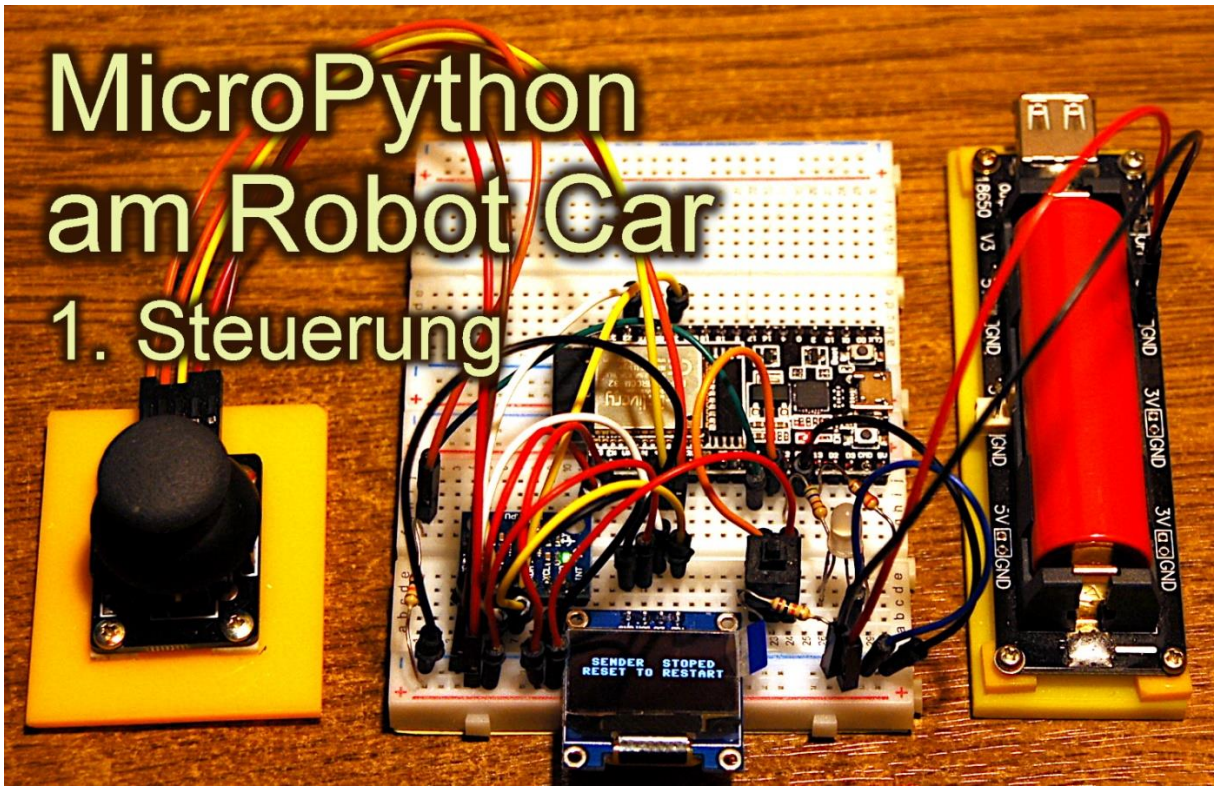


Some time ago a friend asked me if I could develop the control for a robot car in MicroPython. Sure, I said, I have to go. A WiFi-based remote control, an ESP8266 is probably sufficient and an ESP32 will be needed on the vehicle because of the numerous control lines. Build a vehicle, put a battery box on it, a motor control, for this there are special chips and a few sensors for distance measurement and path guidance, that is quite easy!

I thought. -

A few weeks ago…

But - God is a girl and the devil is a squirrel and, as you know, it's in the details. But now the time has come, I can already introduce you to the control of the project. So welcome to the making-of of

MicroPython on the Robot Car

Part 1 - The controls

The parts were quickly selected. I had the following material for the controls. The vehicle itself comes in part 2. And in part 3 there will be a surprise - the somewhat different robot control.

1	NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI Wifi Development Board mit CH340 - 1x Lolin V3
1	oder besser gleich (siehe Beschreibung) NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI Wifi Development Board mit CH340 - 1x Lolin V3
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

Used Software:
For flashing and programming the ESP:
Thonny oder
µPyCraft

For testing the functionality of the transmitter:
ncat

For testing the server on the vehicle
packetsender

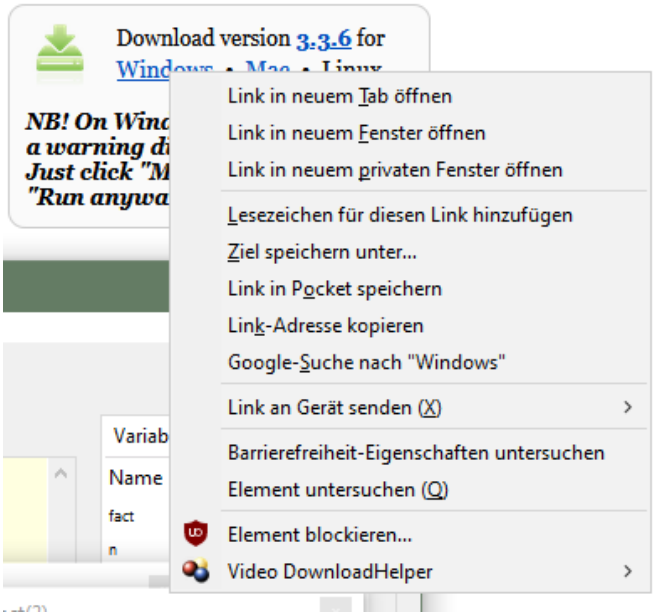
A few thoughts on MicroPython

The interpreter language MicroPython is used in this project. The main difference to the Arduino IDE is that you have to flash the MicroPython firmware onto the ESP32 before the controller understands MicroPython instructions. When that is done, you can have a casual conversation with your controller, send individual commands and immediately see the answer without having to compile an entire program beforehand. The following section tells you which steps are necessary to do this.

The development environment - Thonny

Thonny is the counterpart to the Arduino IDE under MicroPython. In Thonny, a program editor and a terminal as well as other interesting development tools are combined in one interface. So you have the working directory on the PC, the file system on the ESP32, your programs in the editor, the terminal console and, for example, the object inspector in one window.

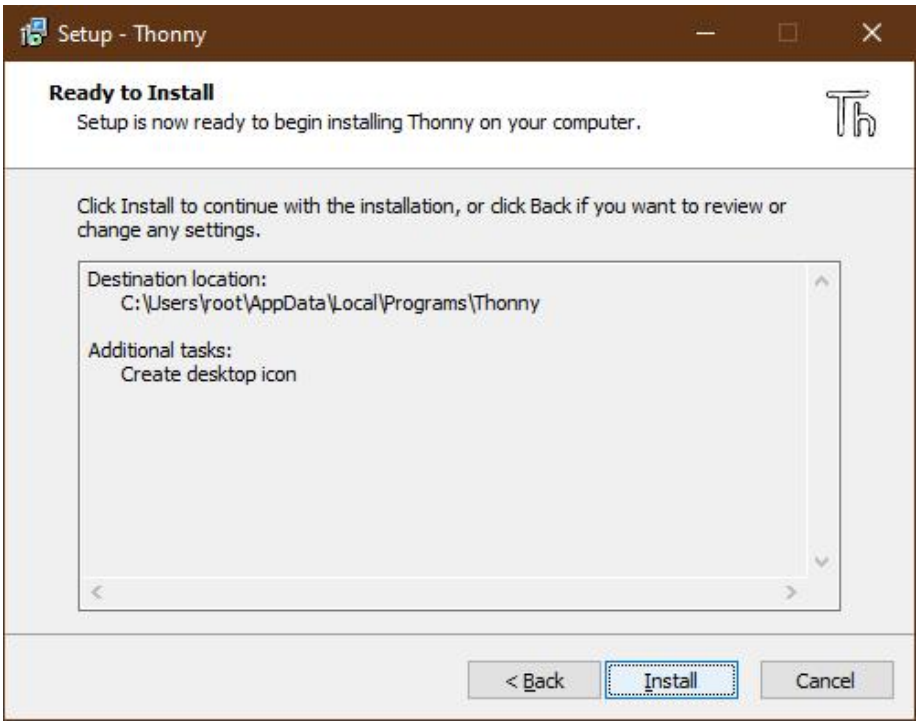
The resource for Thonny is the file thonny-3.3.x.exe, the latest version of which can be downloaded directly from the product page. There you can also get an initial overview of the features of the program.



Right-click on Windows and save target as to download the file to any directory of your choice. Alternatively, you can also follow this direct link.
In addition to the IDE itself, the Thonny bundle also includes Python 3.7 for Windows and esptool.py. Python 3.7 (or higher) is the basis for Thonny and esptool.py. Both programs are written in Python. esptool.py also serves as a tool in the Arduino IDE to transfer software to the ESP32 (and other controllers).

Now start the installation of Thonny by double-clicking on your downloaded file, if you only want to use the software for yourself. If Thonny & Co. is to be available to all users, you must run the exe file as an administrator. In this case, right click on the file entry in Explorer and select Run as administrator.

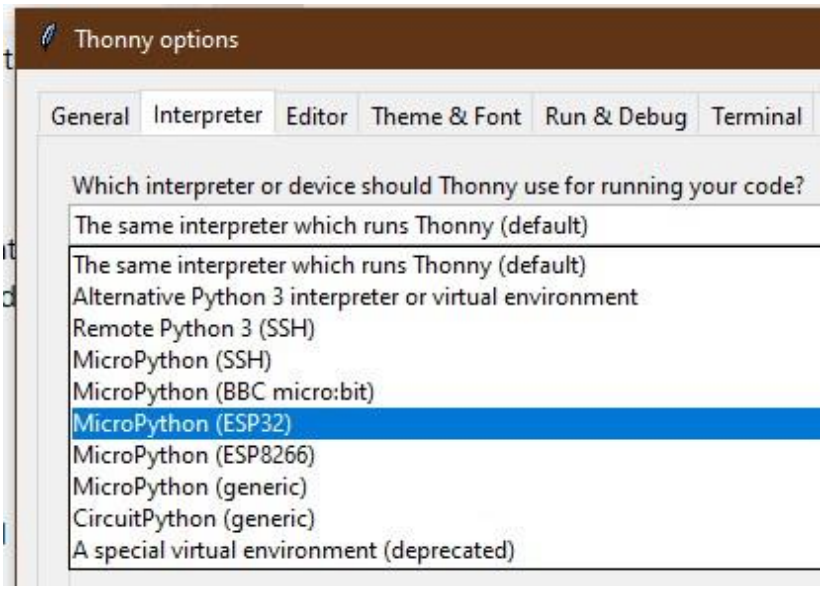
Most likely, Windows Defender (or your antivirus software) will answer you. Click on more information and, in the window that opens, click on Run anyway. Now just follow the user guidance with Next.



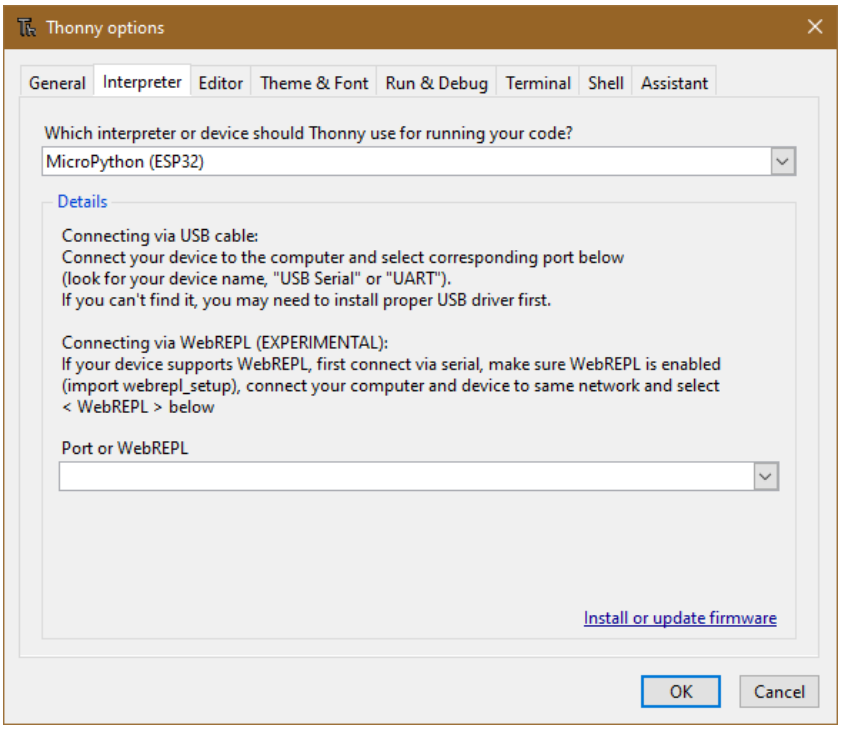
Click on Install to start the installation process.

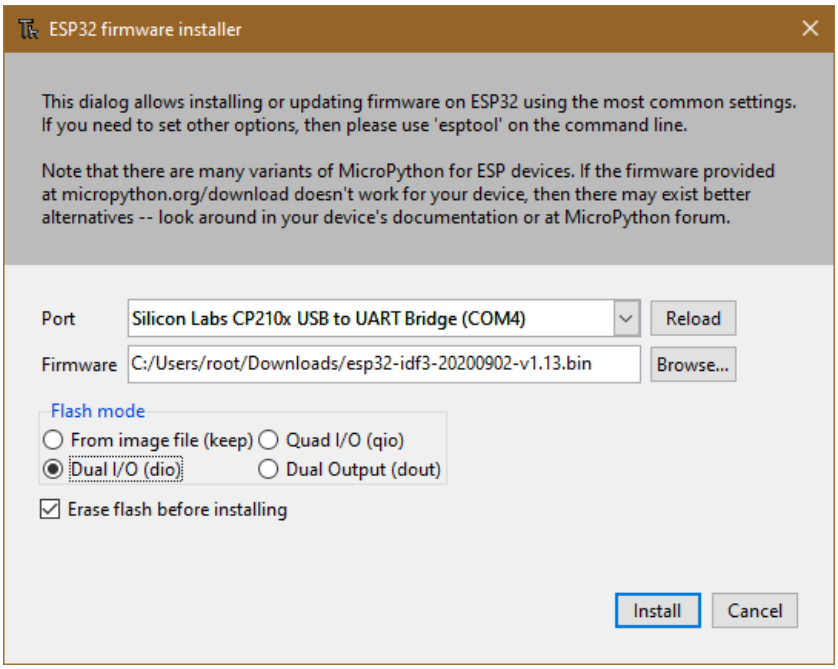
When you start the program for the first time, you specify the language, then the editor window is displayed together with the terminal area.

As the first action, set the type of controller used. With Run - Select Interpreter… you are landing in the options window. For this project, please set Micropython (ESP32).



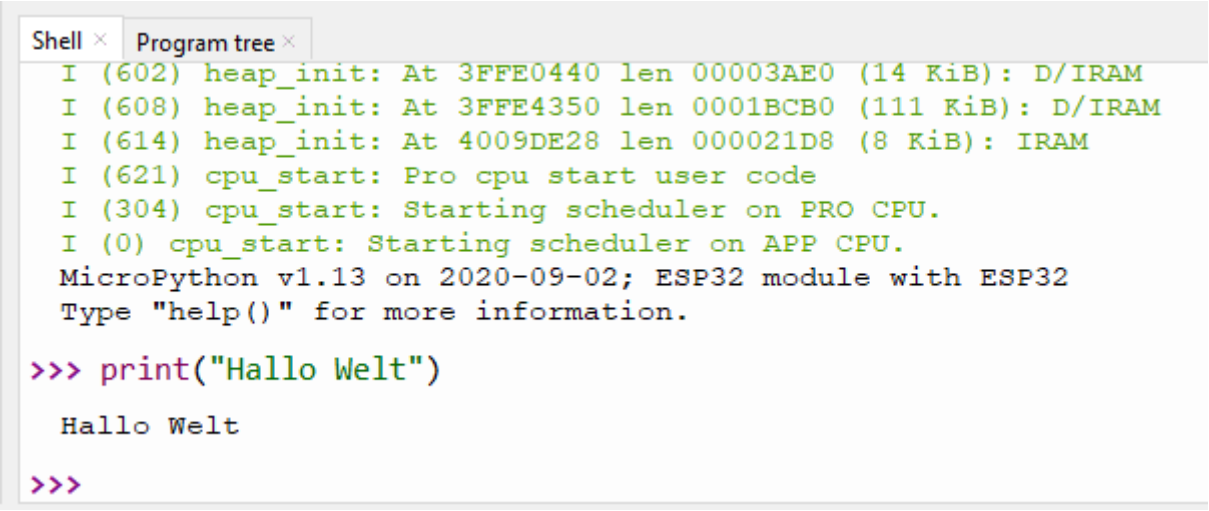
Now download the Micropython firmware for the ESP32 and save this file in a directory of your choice. The bin file must first be transferred to the ESP32. This also happens to Thonny. Call up Thonny Options again with Run - Select Interpreter…. At the bottom right click on Install or update Firmware.





Select the serial port to the ESP32 and the downloaded firmware file. Start the process with Install. After a short time, the MicroPython firmware is on the controller and you can send the first commands to the controller via REPL, the MicroPython command line. For example, enter the following command in the Terminal window.

print("Hallo Welt")



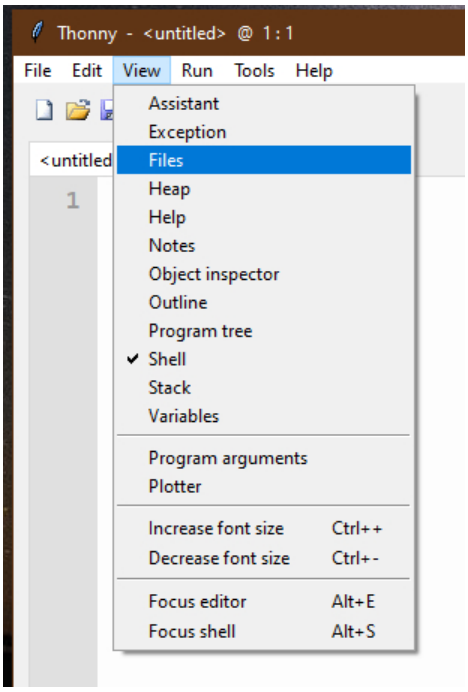
In contrast to the Arduino IDE, you can send individual commands to the ESP32 and, if it is MicroPython instructions, it will respond well. On the other hand, if you send a text that is incomprehensible to the MicroPython interpreter, it will alert you to this with an error message.

>>> print "hello again"

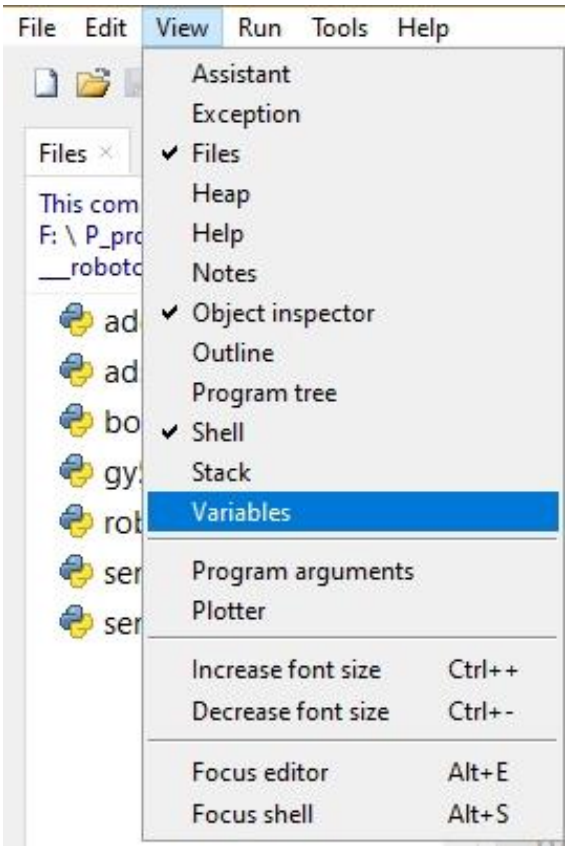
SyntaxError: invalid syntax
Traceback (most recent call last):
  File "<stdin>", line 1
SyntaxError: invalid syntax

To work, however, the overview of the workspace and the device directory is still missing. The workspace is a directory on the PC in which all files important for a project are located. In Thonny his name is This Computer. The device directory is the counterpart on the ESP32. In Thonny it is called MicroPython device. You can display it as follows.

Click on View and then click on Files.



Now both areas, the workspace at the top and the device directory at the bottom, are displayed. You can show or hide additional tools via the View menu.



The hand control for the robot car

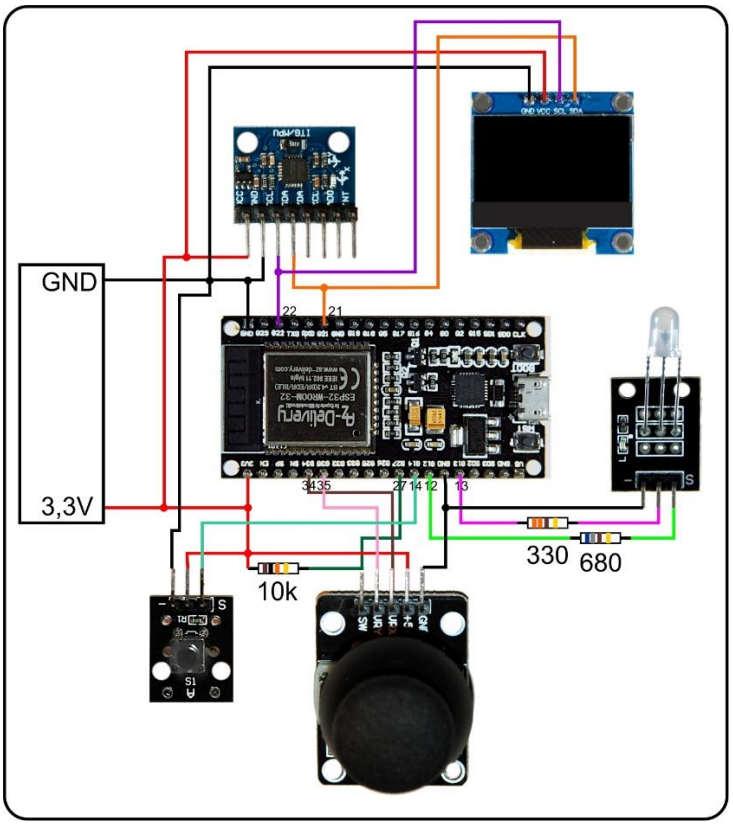
Basic knowledge of the vehicle has already been conveyed in previous articles on the subject of robot cars, for example in "Remote controls with joystick". There has also already been an introduction to the use of the MicroPython language, Projekte mit MicroPython und dem ESP8266/ESP32 - Teil 1, 2, 3, 4, 5 und 6. But it is interesting to combine both worlds. In this project, the advantages of MicroPython as an interpreter solution on the one hand and the large flash and RAM memory of the ESP32 together with the WiFi option on the other hand come to the fore. While researching the secrets of hardware and software, a sentence from a song by Kate Wolf always comes to mind: "In China and a woman's heart there are places no one knows". That means there is always something new to discover in the realm of MicroPython. On the way to this, a lot of small steps were necessary, most of which led via REPL and could be implemented and checked very quickly without having to create and compile an entire program, as is the case in the Arduino IDE.
The solution to the whole job,

- Generation of the speed steps
- Standardization of the speed steps on the basis of various hardware components such as ADC conversion or acceleration measurement
- Communication with the program using as few buttons as possible
- Optical processing of the feedback and
- the radio transmission to the mobile

made it necessary to divide the entire control program into individual modules and program parts. In this way, new, better classes were created from the existing through refinement, all of which could be tested individually, often directly via REPL, the MicroPython command line on the PC. The modules perform the task of libraries in the Arduino IDE.

Originally I thought I could get by with a two-color LED as a signal for feedback. An RGB LED was ruled out because there was no longer a free connection for the third LED on the ESP8266, which I had planned for the controller at the beginning. So an OLED display was put into service on the I2C bus, which enables feedback in plain text - what a luxury! But it went fine. - Until Murphy struck.

When the individual modules and the two program parts were finished and functioned independently, I was disappointed to find that the SRAM of the ESP8266 was nowhere near enough. So the only option was to switch to the bigger brother ESP32. The circuit was created on two coupled mini breadboards, on which all other parts except for the battery and joystick fit. You can also download the transmitter wiring diagram as a PDF



The following modules have special tasks in the control program, which consists of the two files boot_sender.py and sender.py. If everything works, the contents of boot_sender.py are copied to the boot.py file so that the ESP32 can start autonomously.

adcrc.py	Supplies the raw data of a joystic module that is connected to two configurable analog pins of the ESP32.
ads1115rc.py	Provides the raw data of a joystick module that is connected to an ADS1115 module via I2C.
gy521rc.py	The accelerometer module supplies inclination data to the ESP32 / ESP8266 via I2C
beep.py	The module is responsible for acoustic and visual LED signals as feedback from system processes. It can also start a delayed IRQ process.
button.py	This is the successor of the modules touch.py, touchx.py and touch8266.py The module provides the revised services for the operation of keys. It contains the BUTTONS, BUTTON32 and BUTTON8266 classes.
oled.py	Provides methods for the positioned output of data on an OLED display. The entries are subject to a plasibility check with regard to position and text length.
ssd1306.py	Contains the basic environment for modules with the SSD1306-I2C processor.
robotcar.py	Normalizes the raw data from the input modules adcrc.py, ads1115rc.py and gy521rc.py to speed steps with regard to the mean value and the resulting range values. The number of speed steps as well as the fade-out around speed step 0 are variable by parameters.

sender.py	Adopts the results of the connection establishment from boot_sender.py during the test phase. In production, it is called from the boot.py file.
boot_sender.py	Contains all settings and specifications for establishing a WiFi connection to an access point in the local network or for a direct connection to the access point of the server on the Robot Car. The contents of boot_sender.py must be copied into its boot.py file for the transmitter to start autonomously.

The essential part of the three classes adcrc.ADC32, ads1115rc.ADS1115 and I have listed gy521rc.GY521 below for the ADC32 class.

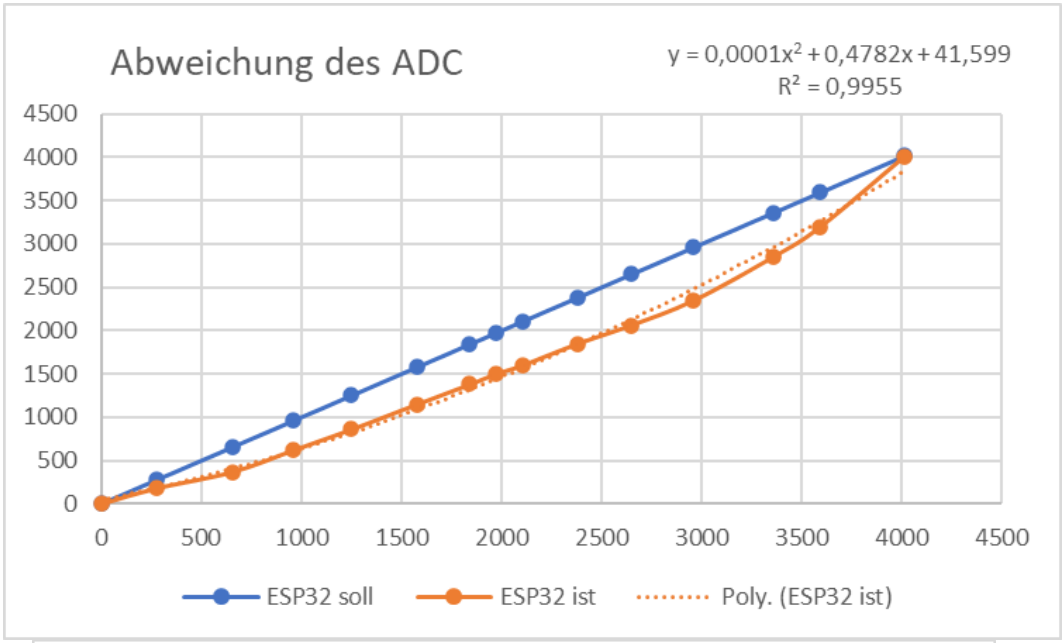
Download **adcrc.py**

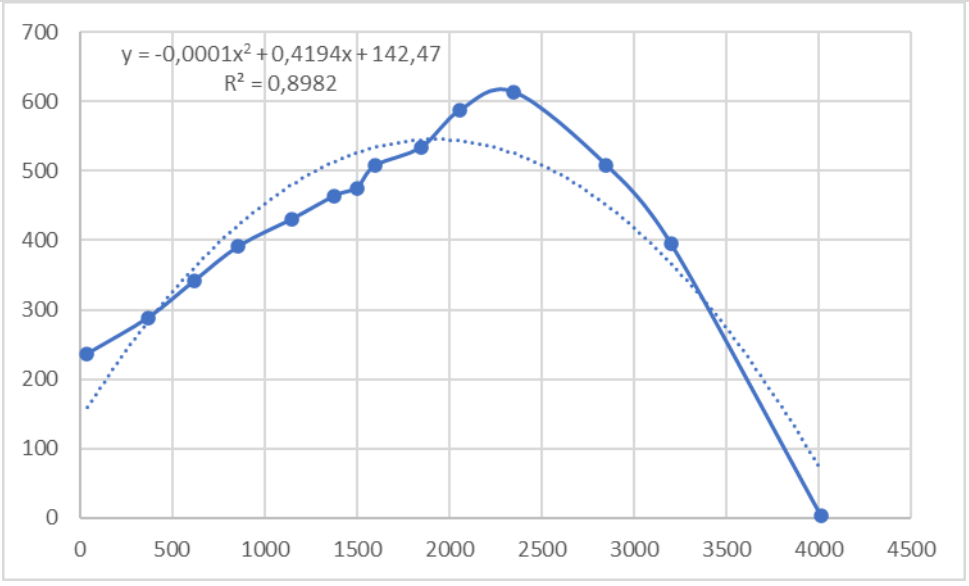
```
class ADC32:

    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
```

The constructor doesn't really need an I2C bus. This information is only necessary for reasons of compatibility with the other sources of direction detection. The rest of the constructor instructions only deal with the ESP32's own ADC.

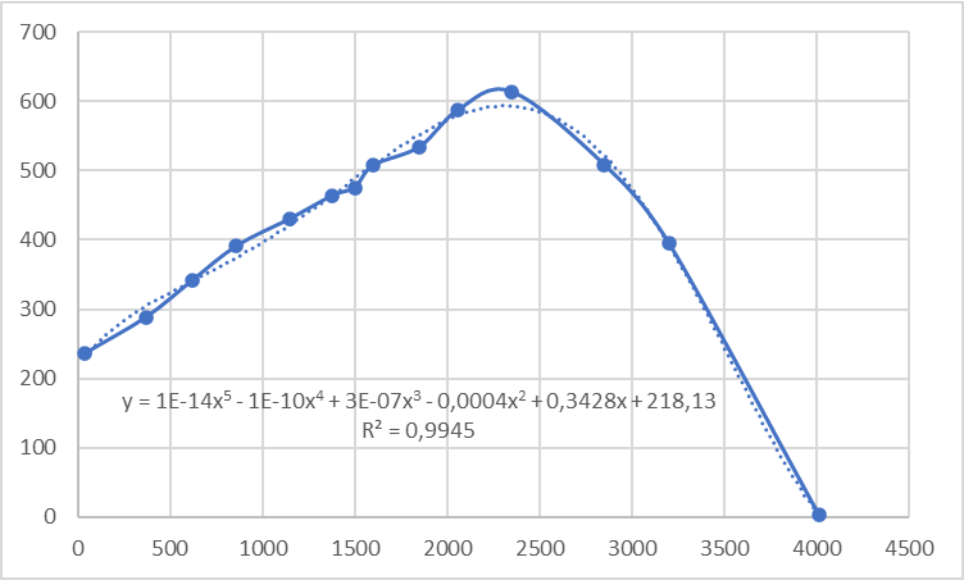
The getXY () method, which takes no parameters, is important. It is included in all three classes with the same API. Therefore the three modules / classes can be exchanged for each other without any problems. What else happens within the classes is of no interest for the data export.

For the ESP32 you have to go back a bit. The ESP32's ADC isn't exactly blessed with accuracy. The "counts versus input voltage" curve is anything but linear. As a very rough approximation, one can assume a quadratic deviation of the measurement error. It would be better to use a 5th order correction function, but that goes beyond the time limits of applicability for this project. The deviation (orange) from the target value (blue) is considerable over the entire measuring range. The target / actual differences are calculated for the correction. and shown graphically. Adjust a trend line (as simple as possible) and let the formula be specified. With the help of the formula you can then calculate the deviation for any measured value and add it to the measured value.





In this graphic, the difference values are plotted as a function of the raw values from the ADC. The 2nd order trend line is not optimal, but it still alleviates the measurement errors of the ADC. The following adjustment would be better, but take a look at the function term! The correlation coefficient R2 is of course almost ideal at 0.9945. Still, we stick with the quadratic function.



The voltage values of the joystick are first smoothed by calculating the mean value from 4 measurements. The raw values are then sent through the second-order correction function, the parameters a and b of which were determined using a spreadsheet. These values differ from module to module depending on the example and should therefore be determined by yourself. In addition, it makes sense to manually model the parameters a and b until the actual graph corresponds best to the target graph.

If higher accuracy is required, I recommend using the ADS1115 ADC module. The resolution is 16-bit against 12-bit with the ESP32. The module could also be used on the ESP8266. The module is addressed via I2C. Since the I2C bus also has to serve the OLED and GY521 modules, the ADS1115 is a good option.

Up to three modules use the I2C bus in the project. It therefore makes sense to define the bus only once in the start program (ultimately boot.py) instead of decentrally in the individual modules. The modules receive the I2C object from the main part as a parameter or in the global environment (from boot.py to server.py). The use of BEEP and OLED is regulated analogously, declaring it once and using it multiple times.

The ADS1115 and GY521 classes have a sometimes considerable overhead of system constants and methods due to their internal structure, but offer standardized access to the raw data for position evaluation, using the getXY () method.

The BUTTONS class provides methods for handling key instances that can be generated using BUTON32 and BUTTON8266. (A comparable class will soon be available for touchpad objects.)

BEEP and OLED are classes that pass information from the system to the user. The documentation in the modules provides information about their use.

The class robotcar.RC inherits from the uncommented class in the header area of robotcar.py. It is the core of the control system and is responsible for turning the raw data from the sensors into usable speed step values.

The first step is to determine the horizontal rest values of the sensor in the x (front-back) and y (right-left) directions. Then the maximum and minimum values in the main directions can be determined and finally the resulting ranges from the rest position to the maximum or minimum. All of this happens during the calibration of the controller, which is carried out once at the very first start or repeatedly at the push of a button on new starts. The calibration values depend on the component, but also depend on the operating voltage, which can change in the course of use due to the discharge of the batteries. After calibration, the values are saved in an ini file in the file system of the ESP32, the name of which depends on the type of sensor. When restarting, an attempt is made to read this file. If this succeeds without errors, the values contained are used for the following "session". Otherwise it has to be recalibrated. During the calibration, the OLED shows how the sensor should be tilted. (Luxury is so great!)

The raw values from the sensors sometimes fluctuate considerably. The introduction of speed steps mitigates this effect to different degrees. Nevertheless, the values of the speed steps are also influenced by it. So that the vehicle does not flutter in the area around the mean values, the rest area can be set by specifying the parameters of the getSpeedLevel () method. A larger value ensures more calm around the zero position. The number of speed steps can also be set in the program. Try out what is best in your case.

```
# 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)
```

The boot_sender.py program sets up the basis for the server.py part that is based on it. After defining various variables and objects, an attempt is made to either establish a connection with the local WLAN router or directly with the access point of the server on the Robot Car. The selection is made using one of the buttons. The OLED informs about the requirement.

In this area you must have the access data for your private environment available. The WLAN router will require the SSID and password, the SSID is sufficient for the access point of the ESP32, a password is not expected here when logging in.

Once the connection has been established, the OLED display shows the connection data. At the end of the boot part you have the option of canceling the program. This "breaking point" has proven to be very useful during program development. The division of the entire application into two is also tactically clever, because you do not have to go through the boot part with login etc. every time you manipulate the server part. The boot part can also be ported to other WiFi applications. The boot part is a kind of module without a class of its own.

```
# 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)
```

So far it has been about the necessary extra income for the Robot Car project. with the exception of what I called the core above, the robotcar.RC class. This core is encased in the sender.py file.

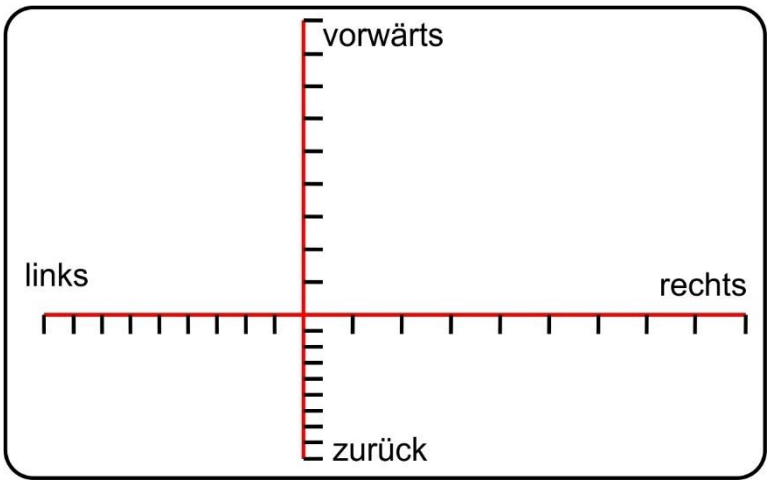
As always when it comes to servers or clients, we import the socket class. Next comes the import of the RC class, named not after Remote Control but after Robot Car, although the former is also applicable in this case. We give the I2C object i2c and also the OLED object d to the object a, which we derive from RC.

```
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
```

The program asks whether it should recalibrate. We answer the question with the help of the left button, which will later switch the lights on the vehicle. If the key is not pressed, the program tries to read the adc.ini file. When the program is started for the very first time, an error will certainly occur here, the file does not yet exist. The error is caught by the except statement, the error text is output on the console and the calibration is called. The OLED display tells you what you have to do. After each calibration, the result is written to the ini file. For which sensor class this should happen, RC.calibrate () learns from the instance attribute self.FILE, which is inherited from the sensor class (here adcrc.ADC32) to robotcar.RC. Sounds like a lot of work, but it's very practical. In addition, the result is saved in the instance attributes and returned to the calling program as a tuple.

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

Now we determine the approximate number of speed steps per direction and have the divisors calculated for the different value ranges. Because the rest position usually does not represent the middle of the entire value range, the areas (red lines) are not the same size. If you want to have the same number of subdivisions, the dividers must be adapted. Because the PWM values with the speed step as an index are fetched from a list later on the vehicle, the index = speed step value must not be greater than the number of list entries. Therefore, the divisor is rounded up as an integer value. However, this can also mean that the number of speed steps is slightly below the setpoint. The divisors, like the maxima, minima and mean values, are also stored in instance attributes and are actually only returned to the calling program for troubleshooting. You could safely remove the assignments in the main program.



The development of the relevant methods has taken the longest because of the integration of various sensors.

Now we are almost at the end of the transmitter part. We create a socket instance s, bind it to the IP address brought with us by boot.py and assign a port number. A few words about the protocol stack for sending.

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

The sender uses the UDP protocol (see: socket.SOCK_DGRAM) instead of the usual TCP protocol (socket.SOCK_STREAM), which is encoded in bold text. In this case I chose UDP because the transmission is much faster. The sender simply forwards its message to the server on the robot car without negotiating a connection beforehand. The connection is therefore also unsecured. This means two things, firstly it is not guaranteed that the package will arrive at the server at all and secondly it is not guaranteed that the package will arrive intact. The former is not bad because there are enough other packets with the same kind of content to follow and the second is not bad either because an incomprehensible packet is sorted out by the server. UDP is therefore comparable to a serial interface that sends data although it does not know whether a remote station is listening at all.

The connection data appear in the OLED display, the green LED indicates that it is now possible to drive. The button on the joystick switches the motor relay on the robot car, the other button switches the light on or off with each press. If the motor button is pressed for about 5 seconds, the transmitter switches off. To start again you have to press the RST button on the ESP32 or switch it off and on.

If you use the module gy521 with the tilt sensor MPU6050 for control, you can accelerate the Robot Car by tilting it in the appropriate direction, reverse it or steer it to the left and right. If no joystick is connected, you have to replace its button with a button module, otherwise you cannot switch the motor. programming the motor button is the most complex part of the transmitter loop because of its dual function. In addition to the buttons, the speed level is continuously queried. the two parameters determine the interval up to which a zero is returned instead of the speed step. If you look closely, the sender actually only consists of the four lines that begin with s.sendto.

The transmission protocol is very simple in nature.

v: 5 means speed step 5 forward, v stands for velocity.
v: -4 then means with speed step 4 backwards
d: 6 we go to level 6 to the right
d: -6 the same to the left
l: 1 (small L) light on, off, on, off ...
m: 1 motors on, off, on, off ...

The long press of the motor button is not transmitted to the Robot Car, but only maneuvers the controller into an endless loop. This means that the transmitter also needs an on / off switch to disconnect it from the battery. This can then also be used to cold start the circuit again.

Here ist he listing of 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)
```

How do you know now, after you have nicely set up and programmed everything, that the party is really going to happen? Method 1 is to replace the send commands with print statements of the form:

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

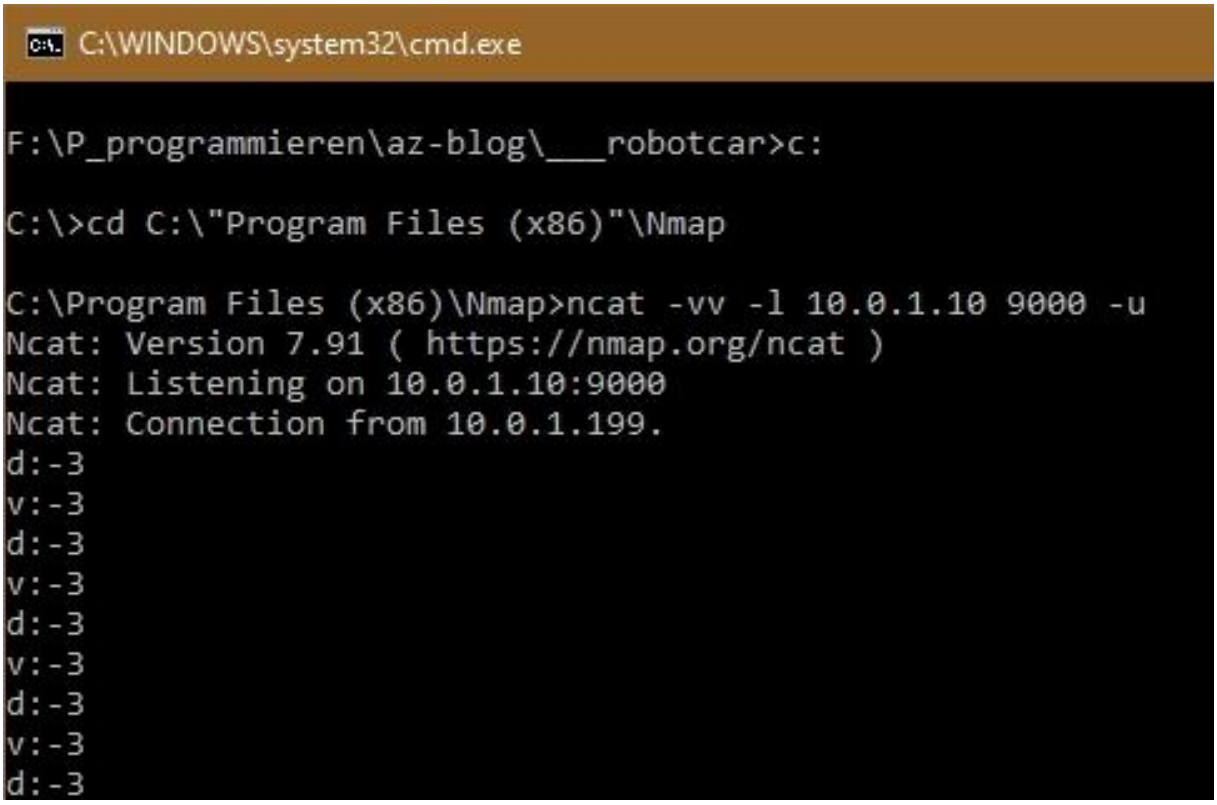
Then everything else that is broadcast is output on the console. The ESP32 must of course be connected to the PC via the USB cable. If the expected text appears, you can be sure that the sensors and data processing are working correctly. But the information about the transmission path is still hidden.

To wake up, make your PC a UDP server. You can do this with ncat. Download and install the freeware. The ncat.exe file can be found in the nmap installation directory.

Open a Powershell, change to the installation directory and call up the ncat file as follows:

ncat -vv -l 10.0.1.10 9000 -u

Now start the transmitter in the same subnet and with the destination port specified here (9000). If the output in the ncat window looks the same as it did in the Terminal before, you are victorious.

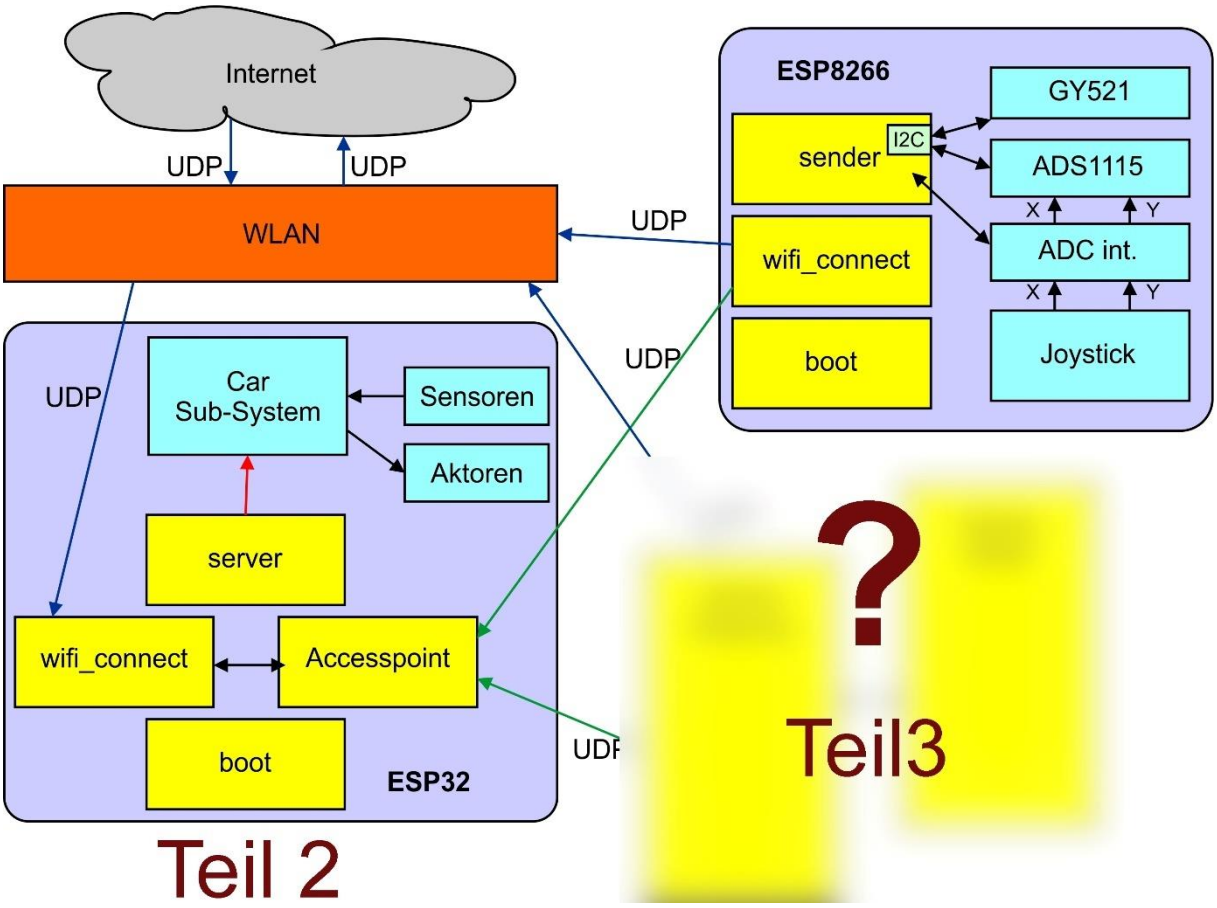


Tipp:
To quickly open a Powershell in the installation directory, look for the directory in Explorer with a few clicks. Now right-click the directory entry while holding down the Shift key and select "Open Powershell window here" from the context menu.

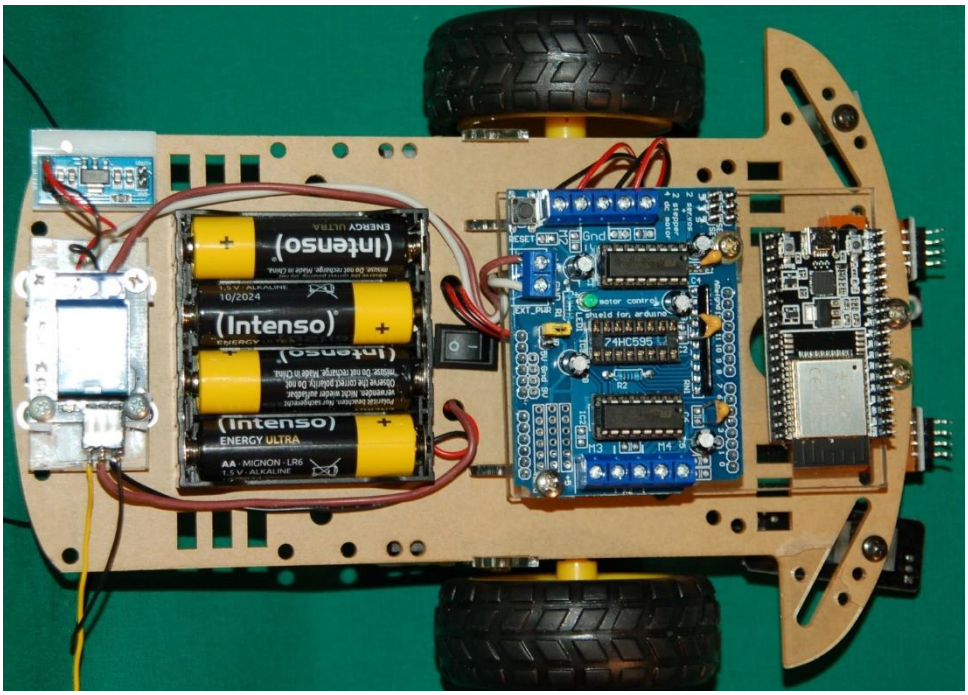
If you want to use the program more often, we recommend creating a batch file with the path change and the content of the call in an editor of your choice, for example with Thonny. Enter the text and save it with a name of your choice, add ".bat" in any directory. A double click on the file starts ncat in a DOS window.

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

So, as a preview of the next part, I have a graphic at the end that shows the modular system of this project.



In the box for the first part, top right, the ESP8266 is also given. It's a shame that didn't work with that. Nevertheless, you can look forward to the second part. You can already choose a vehicle. Mine is still missing a distance sensor and of course the wiring of the ESP32 and motor driver board.



Linkliste:
deutsches PDF
englisches PDF