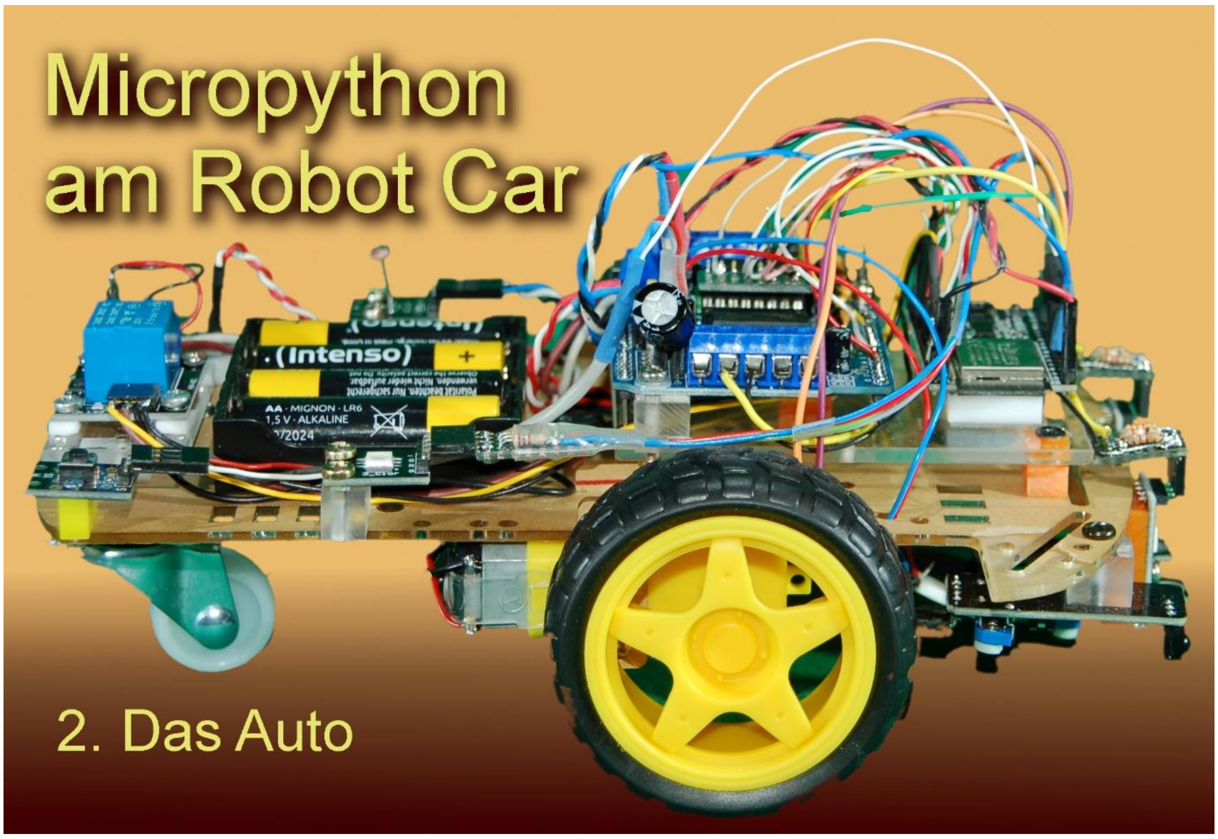


Now we're going to get mobile - the electric car is coming. Well equipped with various sensors and actuators, it offers extensive possibilities for developing your imagination on the subject of electromobility. So welcome to the second part of

MicroPython and the Robot Car

As with the transmitter in the first part, the software for the control again consists of two main parts. A function specially tailored to the robot car is used to control the motor control board. In addition, the button module and the signal module are used again. I have left out an OLED or LCD display because the characters cannot be read at all from a great distance. There is an RGB LED for the feedback, which shows the status of the circuit by means of color signals. But first things first.

The cover picture shows many of the pieces of equipment, but a list of these is always clearer.

Material list

1	ESP32 Dev Kit C V4 unverlötet
1	KY-004 Taster Modul Sensor Taste Kopf Schalter
1	1-Relais 5V KY-019 Modul High-Level-Trigger für Arduino
3	KY-009 RGB LED SMD Modul Sensor für Arduino
1	KY-018 Foto LDR Widerstand Diode Photo Resistor
2	KY-033 Linien Folger Line Tracking Sensor Modul TCRT5000 für Arduino
2	KY-032 IR Hindernis Sensor Modul für Arduino

1	AMS1117 3,3V Stromversorgungsmodul für Arduino Raspberry Pi – 1x AMS1117
1	4-Kanal L293D Motortreiber Shield Schrittmotortreiber für Arduino Mega 2560 und UNO R3, Diecimila, Duemilanove
1	KY-018 Foto LDR Widerstand Diode Photo Resistor Sensor für Arduino
1	Robot Car Chassis mit zwei Motoren, zweirädrig incl. Batteriehalter und Schalter
je 3	Widerstand für RGB-LEDs 1,2kΩ(grün), 390Ω(rot) und 1,0kΩ(blau)
einige	Buchsenleistenabschnitte
einige	Stiftleistenabschnitte
einige	Jumperkabel
etwas	dünne, isolierte Kupferlitze
diverse	M3-Schrauben, Scheiben und Muttern
etwas	doppelseitiges Klebeband (ablösbar)
einige	Acrylglasreste nützlich, falls vorhanden
einige	Stücke Lochrasterplatine etwa PCB Board Set Lochrasterplatte Lochrasterplatine

Tools:
Soldering iron / soldering station + solder wire
small needle-nose pliers
small side cutter
Screwdriver, Phillips + flat slot
If available, an M3 thread cutter is helpful

Used software:
For flashing and programming the ESP:
Thonny or
µPyCraft

For testing the functionality of the transmitter:
ncat im Paket nmap

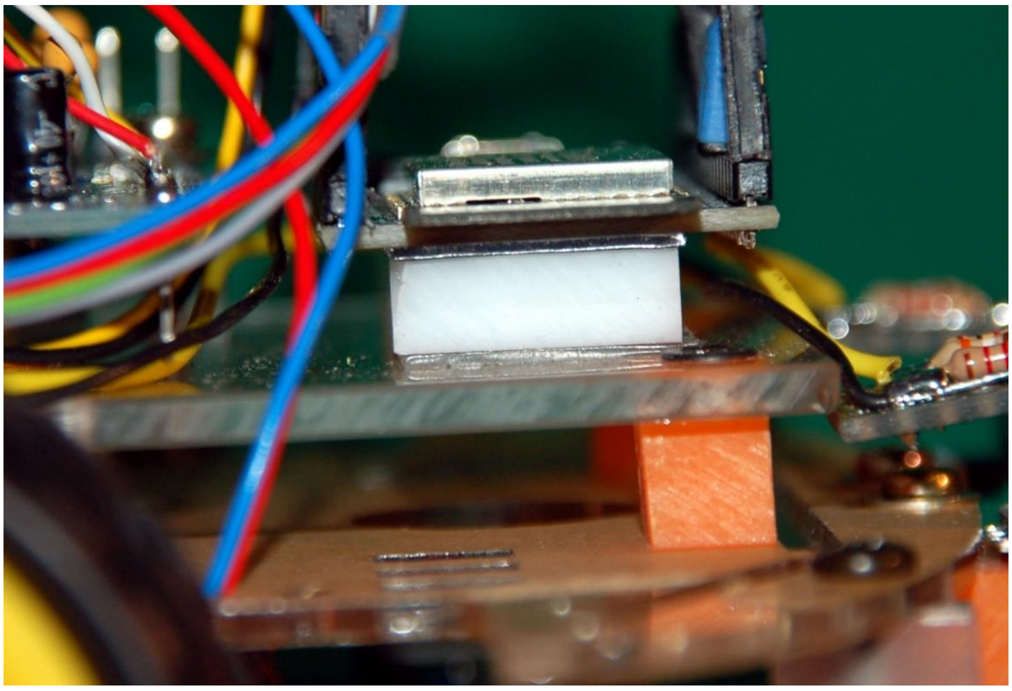
For testing the server on the vehicle
packetsender für Windows

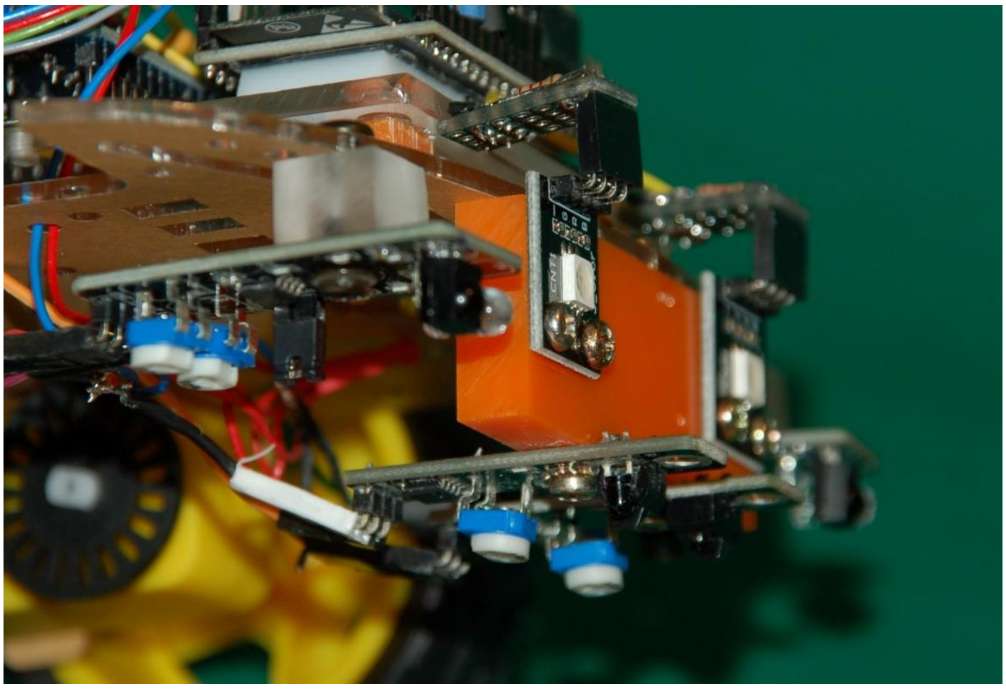
Construction of the mechanics and assembly of the electronic components

Setup begins with assembling the motors, tail wheel, and battery box. You don't need a great description to do this, there is very little chance of going wrong.

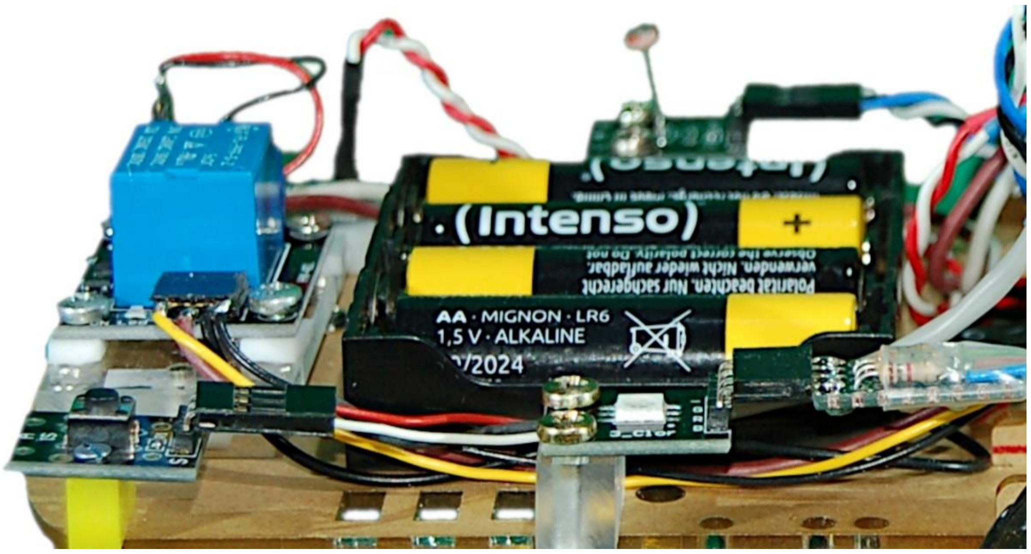
Incidentally, in this project everything that did not fit was made to fit. This applies above all to the assembly of the various hardware components, all of which had to be mounted on pedestals or at least with spacers. Either there were connecting wires from the bottom of the board like with the sensors or whole rows of pins had to be supported like with the motor driver board for the Arduino, which I successfully misused for my project. With this approach no new holes had to be made in the chassis plate of the robot Car can be drilled, but only into my mounting plates. With its 2mm material thickness, the chassis is not really very robust.

With the ESP32 board it is better if you can get the unsoldered version. Then solder the rows of pins so that the pins are facing up. The underside of the board is then relatively smooth and can be easily attached to the surface with (removable) double-sided adhesive tape. I used a piece of acrylic glass as a spacer. The wiring is then done with socket strips or the Femail ends of the jumper cables.

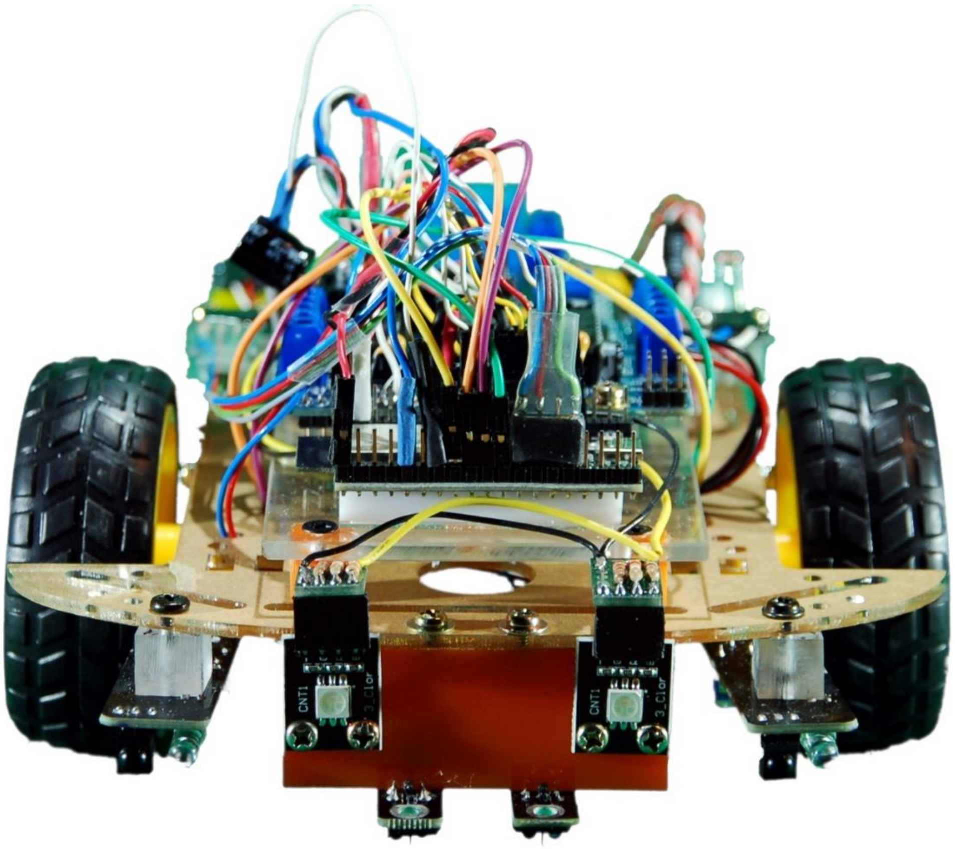




The motors are 6V types and can be supplied directly from the 6V of the battery box. So that the "power part" can be switched off from the ESP32, I have provided a relay. It is high level triggered and is controlled by a GPIO pin of the ESP32.

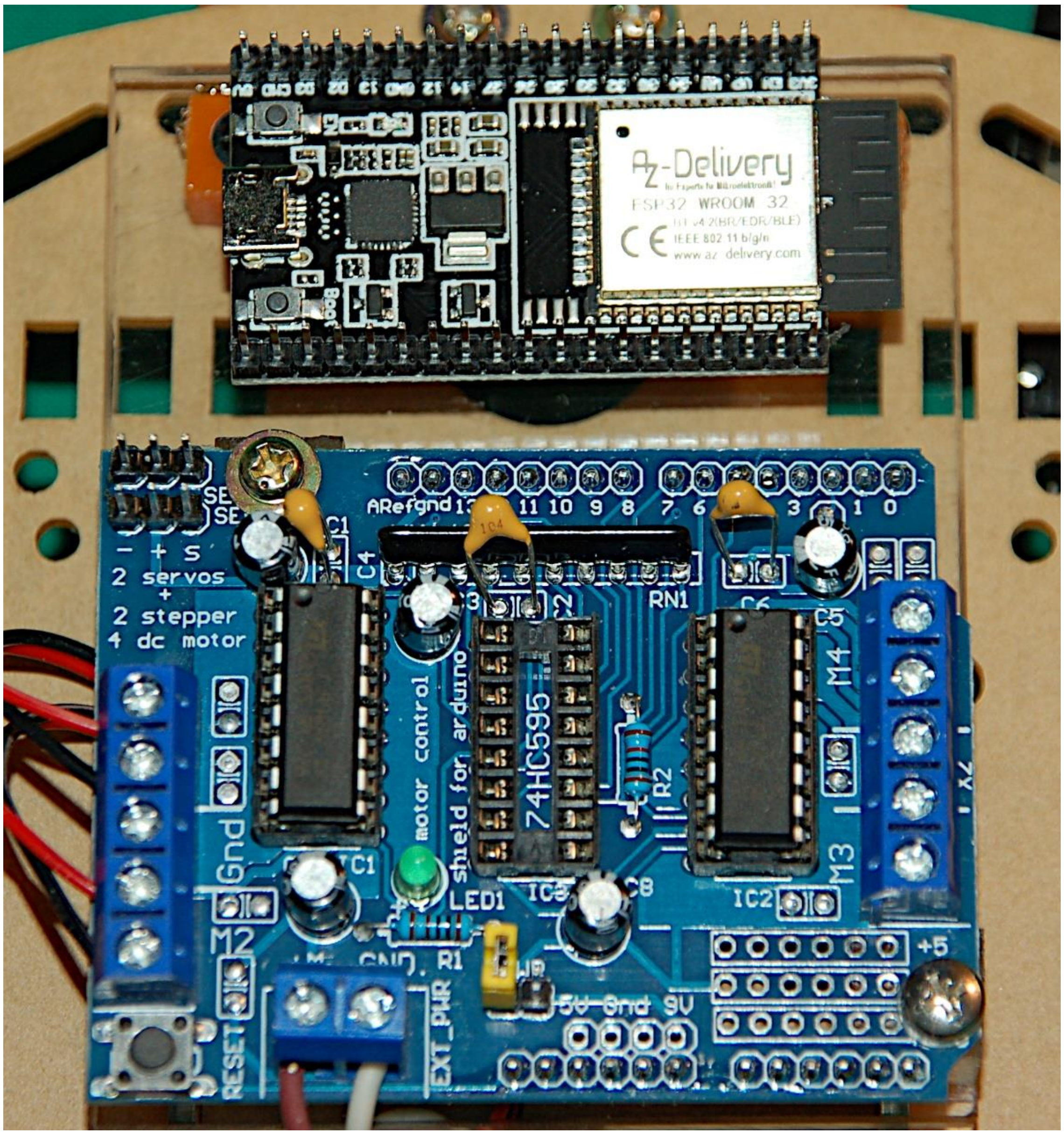


In the front left of the picture, the developer button is mounted on a spacer. It is used during development to specifically terminate the program at important points. This can be used to control the program sequence in the production system. I make use of this in the selection of the start mode in this project. The status RGB LED is located in the center of the picture. The red and white cable in the background comes from the 3.3V controller board, to the right of it the LDR is blurred and pointed upwards.

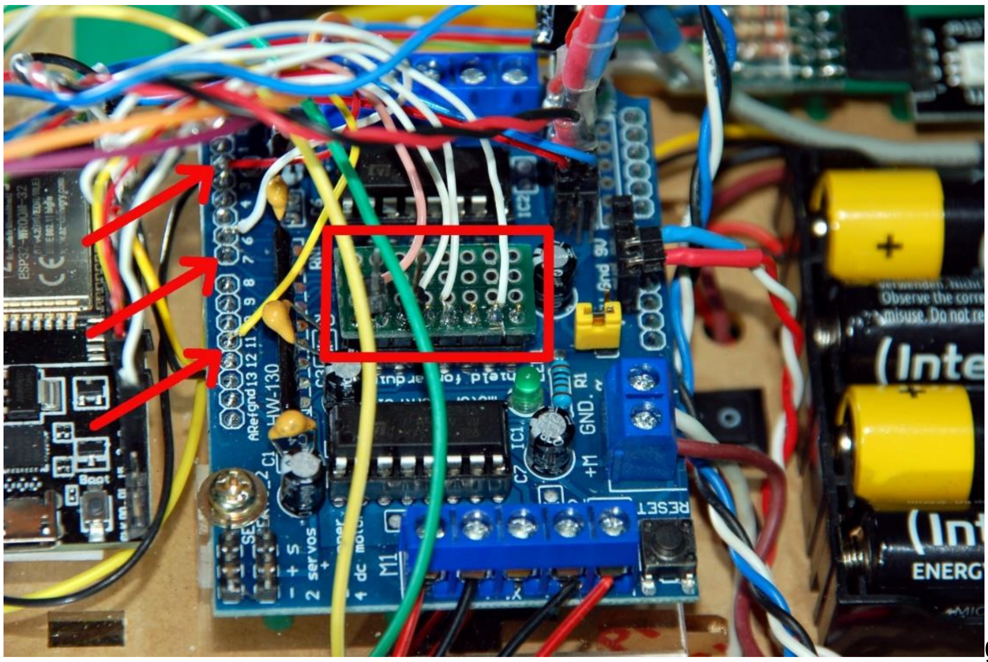


The view from the front reveals the positions of the distance sensors (far outside), the headlights and, at the very bottom, the two sensors for a potential line control.

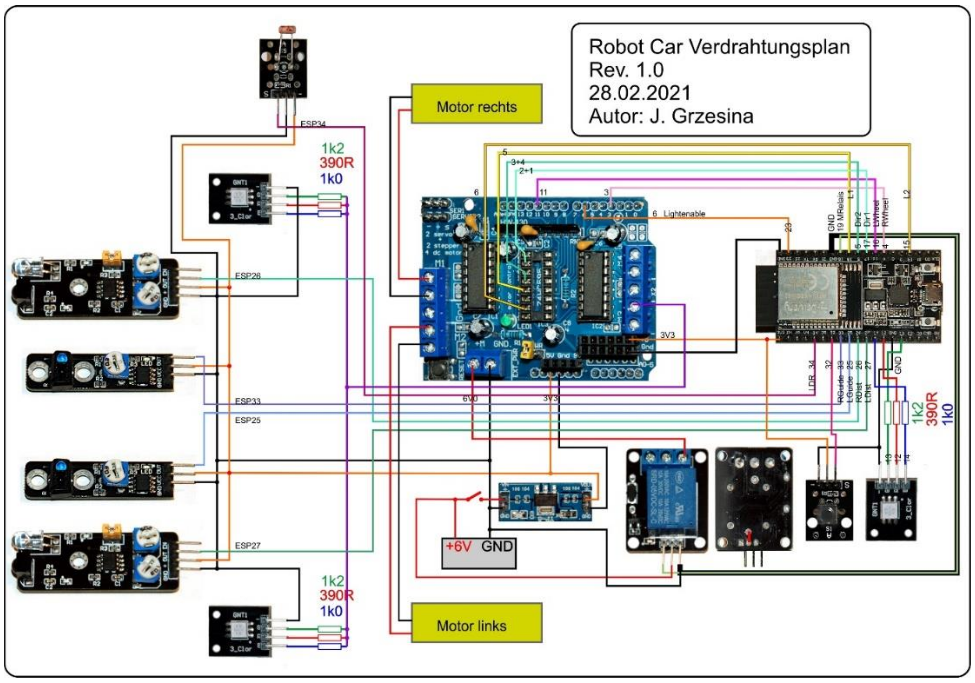
Last but not least, the only thing missing is the motor driver board with the two L293 ICs. I removed the 74HCT595 in order to have better and faster access to the motor driver connections via its socket. Since I only have to control two wheels, one of the two L293s serves as a switch for the headlights and, if you want, for the taillights, which can be switched on automatically as soon as the "reverse gear" is switched on. However, this feature is not (yet) included in the current expansion stage.



The wiring of the shield is done via screw terminals. The two motors are connected on the left, red, black, black, red, the middle terminal is on GND level. The connections may have to be swapped if the direction of travel is incorrect or the car drives in a circle instead of straight ahead. At the bottom of the picture the power supply comes from the relay. The entire circuit board can be disconnected from the battery via the relay.
The next picture shows with the arrows the three lines that are directly soldered on. In the red box you can see the wiring to the 74HCT595 socket, which is implemented using a piece of breadboard. Pieces of wire with a diameter of 0.6mm protrude from the bottom and serve as connectors. The wiring diagram below shows where these lines must be connected to the ESP32.


g

After the mechanics, it is now time to seamlessly move on to the electrical side of the system. I have a circuit diagram for this, which you better [download as a PDF in A4](#).



I included the front sensors in the project because they are essential for autonomous driving of the car. Above all, the two anti-collision sensors (aka obstacle avoidance sensors) can also be used well for remote-controlled operation. Rapid response is possible in the program through interrupt programming. If the adjustable distance to an obstacle is not reached, this triggers a level change at the corresponding GPIO pin. The resulting program interruption (aka interrupt) immediately switches off the motors.

The two RGB LEDs follow to the right in the circuit diagram. I chose them because they emit very bright light, the color of which can be adjusted using the resistors used. As you can see, the LEDs are not directly connected to a GPIO pin, but are supplied directly from the 6V of the battery via a branch of the motor control. This is still well within the amperage tolerances of the LEDs and results in more brightness.

The LDR enables the driving lights to be switched on, just like in real cars, depending on the ambient brightness. The switching threshold can be set in the program. Of course, you can also switch the light on by hand using a remote control - little bells and whistles that make you happy.

The 6V battery supplies on the one hand the 3.3V regulator to which the entire control electronics are attached, ESP32, sensors and signal LED and on the other hand the "power" part of the motor control board and the relay coil. The operating voltage for the two L293D is 2.8 to 5V and is also provided by the 3.3V regulator. The control inputs of the motor shield get along well with the 3.3V signals from the ESP32. This means you can save yourself various level converters. With the wide range of voltage levels at the inputs, apart from the arrangement of the connections, the shield can also be used directly for the ESP family without any problems.

After careful consideration, I decided to remove the 74HCT595 (shift register with output latch and enable) in order to have direct access to the inputs of the L293D drivers. It's much faster and, above all, easier. In serial operation, every level change at the 8 outputs of the shift register would have to be tapped in by a driver routine. I can now change each level individually from the ESP32 in one command via a GPIO pin. If the shield is to be used differently later, the 74HCT595 can be used again.

On the relay module, on the underside of the board, a connection must be made from the + 6V connection of the supply to the switching contact of the relay, then it is sufficient to connect the normally open contact as an output to the motor shield.

The "developer button" at the GPIO32 input can serve various purposes. As with all previous projects, I use it here as a cancel button at important points in the program. As in the first part, it serves as a cancel button between the boot part of the program, in which the connection to the WLAN router is established or, as in this case, a separate access point can be set up, and the actual server part. In this project, it would also be conceivable to use it at an early stage when starting the program in order to choose between WLAN connection and your own access point setup. To do this, a corresponding optical signal would have to be programmed so that you know when to press the button. This is easily possible via the RGB signal LED.

Optical signals in this program are:

- Waiting for the connection to the WLAN router (red flashing)
- Abort option (steady yellow light)
- Ready to drive (steady blue light)
- left line sensor on black line (red continuous light)
- right line sensor on black line (green continuous light)
- Radio connection is established (headlights flash 3x briefly)

Light signals can be programmed using the beep.BEEP class. The button.BUTTONS class provides button actions. The button objects are instantiated using button.BUTTON32.

Unfortunately, there was also an unpleasant surprise during the preparations for this episode. The search for the cause took almost a whole day because I did not want to admit, what ultimately turned out to be obvious. After almost two weeks without problems, the radio unit of the ESP32 Dev KitC V4 said goodbye overnight. I looked for bugs in my program, often a stubborn typo creeps in that you haven't discovered for a long time, but no, it wasn't. The wireless router worked perfectly, but I couldn't get any contact from the ESP32 to it. The realization finally brought the use of another module (ESP32 mini D1), which worked perfectly with the same program. That said, after a few more tests, things were clear. So if something similar happens to you sometimes, don't doubt yourself right away, but also question the proper function of a component. This can be a transistor, an electrolytic capacitor, an IC or, as here, the transmitter module of an ESP32.

Now let's get into programming. As already mentioned, the entire program consists of two main parts. The establishment of a connection or the provision of an access point represents the boot part. When this has been processed, the program reloads the actual server part, provided that the "developer button" is not aborted.

The boot part itself consists of the essential boot sequence boot_essential.py, the import and initialization of further modules for communication, interfaces.py and the actual part for establishing the connection, wifi_connect_server.py or accesspoint.py. This is followed by the "developer sequence" break_section.py, which allows the program to be terminated at this point. After a termination, all declarations made up to this point are available for experimentation via REPL.

If not aborted, the exec statement starts the server part server.py. This part includes the preparation for the reception of UDP datagrams from the sender side and the routines for the decoding and processing of received instructions. The actual server part is very short, similar to the transmitter part from the last episode, and essentially comprises three lines.

```
request, addr = s.recvfrom(1024)
act=request[0:4]
service(act)
```

At the beginning, the pin assignments must be declared. This is followed by the definition of the speed levels with their assignment to the PWM values, which can range from 0 to 1023. The motors only start moving at a value of approx. 500. That depends on the set PWM frequency and of course on the motor data itself. Experimentation is the order of the day. The implementation in the program is kept simple, you specify the maximum and minimum PWM value as well as a minimum and maximum speed level. The program then generates a list of PWM values that correspond to a specific speed level. The content of this list is output at the terminal and can be checked there. The settings for the steering are determined in the same way. During the calculation, the program monitors that the maximum PWM value, taking the steering into account, cannot exceed 1023. The steering takes place by adding the PWM speed value with the PWM value of the steering on the outer wheel of the cam track. Difference is formed on the inner wheel. This means that the left wheel turns faster when turning to the right than the right one and vice versa. The best way to determine the optimal ratio between speed and steering values is through experiments. There are three conceivable approaches to steering. I chose the middle variant here.

1. Only the outer wheel rotates faster
2. The outer wheel turns faster, the inner wheel slower, minimally not at all
3. The outer wheel turns faster, the inner wheel can also turn in the opposite direction

Here is the corresponding snippet from the service () function of the server part.

```
if command=="d":
    D=abs(value)
    D=(D if D <=Sdmax else Sdmax)
    Yaw=Dir[D]
    Direction=(0 if value>=0 else 1) # rechts=0
if Direction == 0:  # nach rechts, links dreht schneller
    PWMLeft=Velocity+Yaw
    PWMRight=(Velocity-Yaw if Velocity>=Yaw else 0)
else:
    PWMLeft=(Velocity-Yaw if Velocity>=Yaw else 0)
    PWMRight=Velocity+Yaw
```

To avoid malfunctions and program crashes, two safeguards are built in. The value transmitted by the sender in value is, if necessary, truncated to Sdmax. The two lines formatted in bold ensure that the PWM value on the inner wheel is the smallest if zero. If you omit the two lines, you end up with option 1. The curve radius is smaller with option 2 than with option 1.

The number of speed steps declared in the server part does not have to match the number in the transmitter part. For a start it might be better to keep the number of speed steps lower in the transmitter and higher in the server = vehicle. This forces you to drive more moderately. The opposite case is slowed down by the vehicle because if the transmitter's gear is too high, an error in the vehicle would make control impossible by aborting the program. How many speed steps you program is up to your personal preferences.

During the development, I carried out the entire test of the function of both program parts individually, as far as possible. The next test level after the first program tests is the "dry dock". To prevent the vehicle from skidding off when the motors were activated, I put the vehicle up so that the wheels did not come into contact with the ground. When everything went as expected, the boot part boot_server_wlan.py was sent to the ESP32 as boot.py, as was the server part server.py, together with the necessary modules, beep.py and button.py. the next time the ESP32 is restarted, the system boots up autonomously. The controls worked as expected.

I used the freeware packetsender to test the individual functions on the vehicle. You can enter control commands here by hand, send them to the server on the Robot Car via UDP and test the reaction of the program and machine there. Then the hand control takes over the command if everything works perfectly.

In the following you will find the listings of the two essential program parts boot.py and server.py for a detailed review. Do not forget to enter the SSID of your WLAN router and the associated password as well as an IP, the mask, the gateway and the DNS of your local network. I preferred to set the IP permanently on the ESP32 and not obtain it from the router's DHCP service. This is common for servers.

boot.py (connect to WLAN)

```
# File: boot.py
# Purpose: booting robot car server
# Author:  J. Grzesina
#
#****************** Beginn Bootsequenz ********************
# Dieser Teil geht an den Anfang von boot.py
#******************** Importgeschaeft **********************
# Dieser Teil wird immer 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()

# Bis hierher allgemeiner Boot-Teil
# -----------------------------------------------------------
# ************** create essential objects *****************
# -----------------------------------------------------------
# ------------- allgemeine Schnittstellen *****************
#
# 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
#
# ----------
# I2C-Bus
SD =    21  # ESP32
SC =    22
i2c=I2C(-1, scl=Pin(SC), sda=Pin(SD))
```

```
# Signal Klasse
from beep import BEEP
rot=12
gruen=13
blau=14
b=BEEP(None,rot,gruen,blau,200)

# Taster
from button import BUTTONS,BUTTON32 #,BUTTON8266
entwicklerPin=32
t=BUTTONS() # stellt Methoden + Klassenattribute bereit
at=BUTTON32(entwicklerPin,invert=True,name="cancel")

# LCD und OLED
#from lcd import LCD  # braucht hd44780u.py
#HWADR=0x20
#CharPerLine=16
#Lines=2
#d=LCD(i2c,HWADR,CharPerLine,Lines)

# from display import OLED # braucht ssd1306.py
# d=OLED(i2c)
# from display import LCD
# d=LCD(i2c)
d=None
# *************** Special boot section end ****************
# *******************  wifi_connect  ***********************
# Dieser Teil verbindet mit einem WLAN-Accesspoint
# erfordert Klasse beep.BEEP, display.OLED | display.LCD
#
# File: wifi.py
# Rev.: robot car 1.1
# Date: 2021-03-01
# Author: Jürgen Grzesina (krs@grzesina.eu)
#
#****************ariablen deklarieren ********************
# Die Dictionarystruktur (dict) erlaubt  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
    macString += hex(byteMac[i])[2:]  # ab Position 2 bis Ende
    if i <len(byteMac)-1 :            # Trennzeichen
      macString +="-"
  return macString


# ******************* Get connected ********************
# Netzwerk-Instanz erzeugen, 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 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')     # # MAC-Adresse abrufen und
myMac=hexMac(MAC)           # in eine Hexziffernfolge
umgewandelt
print("STATION MAC: \t"+myMac+"\n") # ausgeben
# Verbindung mit AP im lokalen Netzwerk aufnehmen,
# falls noch nicht verbunden
# connect to LAN-AP
if not nic.isconnected():
  # Geben Sie hier Ihre eigenen Zugangsdaten an
  mySid = "YOUR_SSID_GOES_HERE"
  myPass = "PUT_YOUR_PASSWORD_HERE"
  # 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())
  while not nic.isconnected():
    #pass
    print("{}.".format(nic.status()),end='')
    sleep(1)
    if b: b.blink(1,0,0,500,anzahl=1) # blink red LED
# Wenn verbunden, zeige Verbindungsstatus & Config-Daten
print("\nconnected: ",nic.isconnected())
print("\nVerbindungsstatus: ",connectStatus[nic.status()])
nic.ifconfig(("10.0.1.101","255.255.255.0","10.0.1.20", \
              "10.0.1.100"))
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
if d:
    d.writeAt(STAconf[0],0,0)
    d.writeAt(STAconf[1],0,1)
    d.writeAt(STAconf[2],0,2)
sleep(3)
#
#
# ******************* Abbruchoption ***********************
# Falls gewuenscht Abbruch durch Tastendruck
# erfordert die Klasse BUTTONS, BUTTTON32
if t.jaNein(tj=at,laufZeit=5,b=b) != t.JA :
  # tpNein touched between 5 sec or untouched at all start
server
  if d: d.clearAll()
  exec(open('server.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)


# *****************   end wifi-connection******************
```

If the ESP32 is to run with its own access point, simply replace the part formatted in bold with the following sequence. The transmitter must then also be started for this operating mode.

accesspoint.py

```
# ***************  Setup accesspoint ********************
#
try:
  import usocket as socket
except:
  import socket
import ubinascii
import network

nic = network.WLAN(network.AP_IF)
nic.active(True)
ssid="robotcar"
passwd="uranium238"

# Start als Accesspoint
nic.ifconfig(("10.0.2.101","255.255.255.0",\
              "10.0.2.101","10.0.2.101"))
```

```
print(nic.ifconfig())

# Authentifizierungsmodi ausser 0 werden nicht unterstützt
nic.config(authmode=0)

MAC=nic.config("mac") # liefert ein Bytes-Objekt
# umwandeln in zweistellige Hexzahlen ohne Prefix und in
String decodieren
MAC=ubinascii.hexlify(MAC,"-").decode("utf-8")
print(MAC)
nic.config(essid=ssid, password=passwd)

while not nic.active():
  if b: b.blink(1,0,0,500,anzahl=1)

print("Server Robot Car ist empfangsbereit")
if b: b.ledOn(0,1,1) # pink
sleep(3)
```

After a connection is available, the server can be started. This happens through the exec statement at the end of the boot part. The server.py file must be located on the ESP32 in the root directory \. The first two imports enable the isolated manual test of the service () routine of the server part without a network connection having to exist. In the production system, these imports are simply skipped and replaced by those in the boot part.

server.py

```
# File: server.py
# Rev.: robot car 1.1
# Date: 2021-03-17
# Author: Jürgen Grzesina
#
# **************** Server department ******************
try:
    sleep(0.1)
except:
    from time import sleep, sleep_ms

if not("sys" in dir()):
    import sys

from machine import PWM ,Pin,ADC

try:
  import usocket as socket
except:
  import socket
# ------------------- Variables  -----------------------
```

```
D1Pin=const(17)      # Richtung
D2Pin=const(5)       # Richtung
MPin=const(19)       # Motor-Relais
LdistPin=const(27)   # LDist
RdistPin=const(26)   # RDist
LfollowPin=const(25)# LGuide
RfollowPin=const(33)# RGuide
Light1Pin=const(18) # L1
Light2Pin=const(15) # L2
LightEnablePin=const(23) # FrontLight enable
LDRPin=const(34)     # Lichtsensor
LWheelPin=const(16) # LWheel
RWheelPin=const(4 ) # RWheel


Bearing=0     # rueckwaerts=1; vorwaerts=0
#BearingOld=0  # vorherige Fahrtrichtung
Velocity=0    # Betrag Geschwindigkeit
PWMFreq=200  # PWM-Frequenz
LWheel=PWM(Pin(LWheelPin),PWMFreq)
RWheel=PWM(Pin(RWheelPin),PWMFreq)
LWheel.duty(0)
RWheel.duty(0)

Dir1=Pin(D1Pin,Pin.OUT)  # Richtungseingang 2+15 am L293D
Dir1.off()
Dir2=Pin(D2Pin,Pin.OUT)  # Richtungseingang 7+10 am L293D
Dir2.off()
Direction=0   # 1=rechts+Yaw>=0, 0=links+Yaw<0
Yaw=0         # Betrag Lenkung PWM-Korrektur
PWMLeft=0     # PWM-Wert linker Motor
PWMRight=0    # PWM-wert rechter Motor

LGuide=Pin(LfollowPin,Pin.IN)
RGuide=Pin(RfollowPin,Pin.IN)

LDist=Pin(LdistPin,Pin.IN)
RDist=Pin(RdistPin,Pin.IN)

Motors=0      # Motorrelais schaltet Vcc2 am L296D
MotorRelais=Pin(MPin,Pin.OUT)
MotorRelais.value(0)

Light=0       # Scheinwerfer 1 oder 0
L1=Pin(Light1Pin,Pin.OUT)
L2=Pin(Light2Pin,Pin.OUT)
LightEnable=Pin(LightEnablePin,Pin.OUT)
LightEnable.value(0)
Enabled=0

LDR=ADC(Pin(LDRPin))
LDR.atten(ADC.ATTN_11DB)
LDR.width(ADC.WIDTH_10BIT)
LightThreshold=512
LDR.read()
Dark=LDR.read()<LightThreshold

# Liste der Geschwingigkeiten erstellen
veloMax=800   # maximaler PWM-Wert fuer Geschwindigkeit
veloMin=490
Svmin=3
Svmax=20
StufeVelo=(veloMax-veloMin)//(Svmax-Svmin)
L=[veloMin+x*StufeVelo for x in range(Svmax-Svmin+1)]
Velo=[]
for i in range(Svmin):
    Velo.append(0)
Velo.extend(L)
Sdmin=2
Sdmax=11
dirMax=1023-veloMax
dirMin=dirMax//Sdmax
StufeDir=(dirMax-dirMin)//((Sdmax-Sdmin))
L=[dirMin+x*StufeDir for x in range(Sdmax-Sdmin+1)]
Dir=[]
for i in range(Sdmin):
    Dir.append(0)
Dir.extend(L)

print(StufeVelo,Velo,"\n",StufeDir,Dir)

# ---------------     Functions     -----------------
# IRQ-Service Routines
def stopMotor(pin):
    #global Motors, MotorRelais
    #Motors=0
    #MotorRelais(Motors)
    LWheel.duty(0)
    RWheel.duty(0)
    #print("Motorstopp")

# ---------------------------
def deviation(pin):
    if pin==LGuide:
        print(pin, "links lenken")
        b.ledOff()
        b.ledOn(1,0,0)
    else:
        print(pin, "rechts lenken")
        b.ledOff()
        b.ledOn(0,1,0)

# ---------------------------
# functions controlled by buttons
```

```
def mainLight(state):
    global Enabled
    Enabled=state
    if state:
        L1.on()
        L2.off()
        LightEnable.on() # Lightenable-Pin auf 1
    else:
        LightEnable.off()

# ---------------------------
def mainOn():
    L1.on()
    L2.off()
    Enabled=1
    LightEnable.on()

# ---------------------------
def backLight(state):
    if state:
        L1.off()
        L2.on()
        LightEnable.on()
    else:
        LightEnable.off()

# ---------------------------
def blinkFront(pulse,pause,cnt):
    h_enabled=Enabled
    if Enabled:
        LightEnable.off()
        sleep_ms(pause)
    for i in range(cnt+1):
        mainOn()
        sleep_ms(pulse)
        LightEnable.off()
        sleep_ms(pause)
    if h_enabled:
        mainOn()

# ---------------------------
def service(action):
  global Bearing, Yaw, Velocity
  global Motors, Light
  global Direction
  global PWMLeft, PWMRight
  #global BearingOld#,Dir1,Dir2
  #act=action.decode("utf-8")
  act=action.decode("utf8")
  command=act[0]
  value=int(act[2:])
  #print("Kommando: {}   Wert: {}".format(command, value))
  if command=="m":
      Motors=(0 if Motors else 1) # Motorrelais umschalten
      MotorRelais.value(Motors)
      LWheel.duty(0)
      RWheel.duty(0)
  if command=="l":
      Light=(0 if Light else 1)   # Licht an/aus
      mainLight(Light)
  if command=="v":
      V=abs(value)
      V=(V if V <=Svmax else Svmax)
      Velocity=Velo[V]
      Bearing=(1 if value>0 else 0) # vor=0
  if command=="d":
      D=abs(value)
      D=(D if D <=Sdmax else Sdmax)
      Yaw=Dir[D]
      Direction=(0 if value>=0 else 1) # rechts=0
  if Direction == 0:  # nach rechts, links dreht schneller
      PWMLeft=Velocity+Yaw
      PWMRight=(Velocity-Yaw if Velocity>=Yaw else 0)
  else:
      PWMLeft=(Velocity-Yaw if Velocity>=Yaw else 0)
      PWMRight=Velocity+Yaw
  if Bearing:
      Dir1.off()
      Dir2.on()
  else:
      Dir1.on()
      Dir2.off()
  #print("v={}; b={}".format(Velocity,Bearing))
  #print("Y={}; D={}".format(Yaw, Direction))
  #print("PWML={}; PWMR={}".format(PWMLeft,PWMRight))
  LWheel.duty(PWMLeft)
  RWheel.duty(PWMRight)
  lw=LDR.read()
  Dark=(1 if lw > LightThreshold else 0)
  #print("LDR", lw, Dark)
  #backLight(Velocity and (not Bearing))
  mainLight(Dark or Light)

# ------------------ Vorbereitungen -------------------
LDist.irq(trigger=Pin.IRQ_FALLING, handler=stopMotor)
RDist.irq(trigger=Pin.IRQ_FALLING, handler=stopMotor)
#LGuide.irq(trigger=Pin.IRQ_FALLING,handler=deviation)
#RGuide.irq(trigger=Pin.IRQ_FALLING,handler=deviation)
Motors=1
MotorRelais(Motors)
blinkFront(500,500,1)
#sys.exit()---
# ----------------- Server starten --------------------
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', 9000))
print("Socket estblished, waiting...")
blinkFront(200,300,3)
b.ledOff()
b.ledOn(0,0,1)

# -------------- Serverschleife -----------------------
while True:
  request, addr = s.recvfrom(1024)
  act=request[0:5]
  #print('from {}\nContent = {}'.format(addr,str(request)))
  service(act)
```

The service (action) function receives most of the codes sent by the sender from the UDP receiving loop. These are the command prefix and the value after the colon including the sign, which can therefore have a maximum of 3 digits, i.e. a total of up to 5 characters. This information is decoded in a sequence of if constructs and converted into actions. Particularly urgent reactions that are requested by level changes in the front sensors are handled asynchronously by program interruptions and the assigned IRQ service routines.

The server understands, for example, the following commands from the sender.

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 the right at level 6, d stands for direction
d: -6 the same to the left
l: 1 (small L) light on, off, on, off ...
m: 1 motors on, off, on, off ...

When the distance sensors respond, the motor relay is switched off. That'll do it Interruptserviceroutine **stopMotor**().

```
LDist.irq(trigger=Pin.IRQ_FALLING, handler=stopMotor)
RDist.irq(trigger=Pin.IRQ_FALLING, handler=stopMotor)
```

A reference to the ISR is passed to the named parameter handler as a callback function when the pin objects LDist and RDist are defined as IRQ sources. The program interruption is requested here when the level on the pins drops from 1 to 0. Pin.IRQ_RISING would be specified for rising edges. Both edges are recognized by oring the constants.

If the ambient brightness decreases during operation, the headlights are switched on automatically, regardless of whether or not the "lights on" command was issued by the hand control beforehand. It is only important that the hand control is in operation. Then v and d commands are constantly sent, which force the server into the decoder loop. In the course of this, the LDR on the vehicle is also queried.

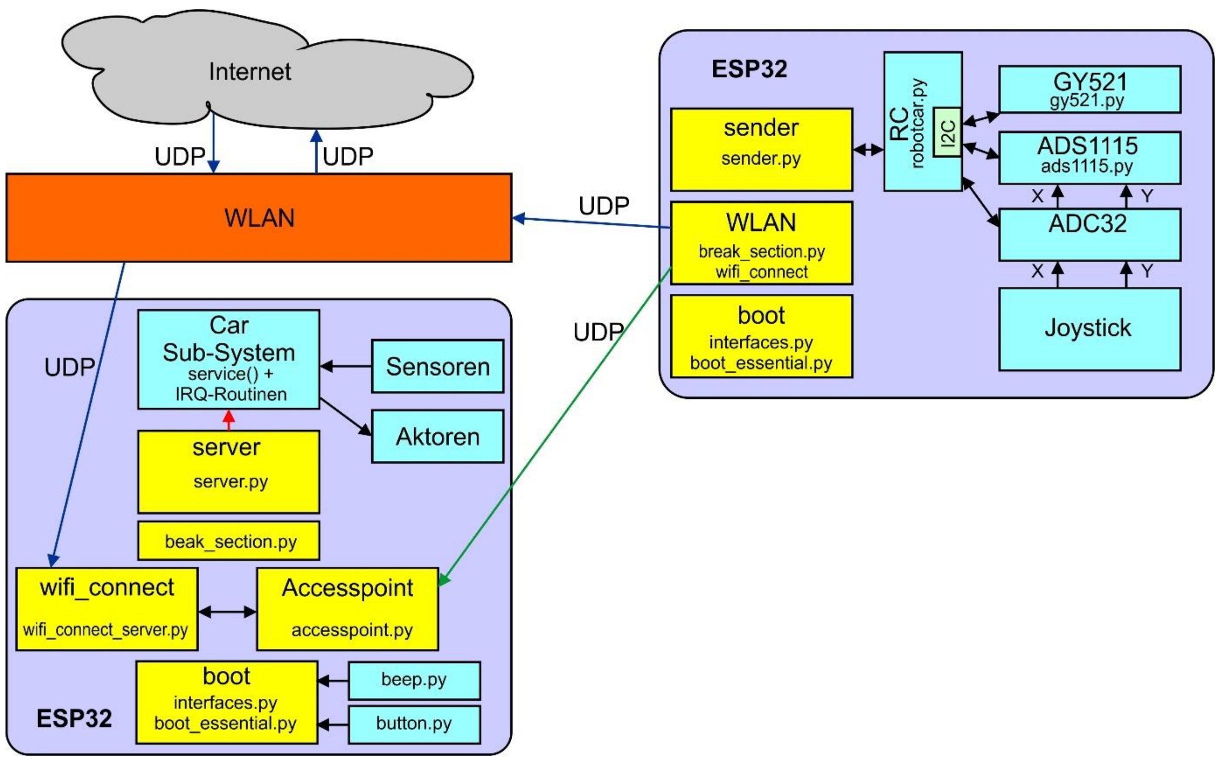
For autonomous driving, the functions of the interrupt service routines (aka ISR) of the line followers still have to be equipped with code aimed at triggering an automatic steering movement. Here you are called upon to research in detail. You already know how steering movements are caused on the vehicle.

The distance sensors cause problems in treatment, because neither of them provide clear edges, neither when approaching too much nor when moving away from the obstacle. Measurements with the DSO have shown that there is always a pulse train with a changing pulse-pause ratio (mean period approx. 1 ms). Depending on the distance to the obstacle, these pulse sequences are between approx. 50ms and 280ms apart, in between I measured GND potential. The rest position of the distance sensors is the HIGH level. It must be checked whether an ultrasonic distance sensor or a time-of-flight sensor is more suitable.

This question may be answered in the next post, which will definitely be about an alternative control option for the robot car. Until then, I hope you enjoy building it and making your first driving tests with the MicroPython-controlled Robot Car.

Here you will find a list of all the program parts used. The graphic illustrates their use.

Datei	Klasse	description
beep.py	BEEP	Methods for optical and acoustic signals
button.py	BUTTONS, BUTTON32 BUTTON8266	Methods for reading keys
boot_essential.py		basic imports for system control, timing and GPIO handling
interfaces.py		Imports for signal control, key operation and display control
break_section.py		Abort option via a button during an optical signal
wifi_connect_server.py		generally establish a connection to a WLAN access point as a server
accesspoint.py		starts the ESP32 as an access point
server.py		Server program part with UDP reception, decoding of instructions and implementation in control actions for the vehicle. Scanning the sensors and reacting to distance events using IRQ routines.
boot_server_wlan.py		Contains all program parts that are necessary to boot the system and set up the WLAN connection, including termination conditions. The entire content can be transferred to the boot.py file for an autonomous start.



Links zum Teil 1
Link zur deutschen PDF-Fassung
Link zur englischen PDF-Fassung

Links zum Teil 2
Link zur deutschen PDF-Fassung
Link zur englischen PDF-Fassung