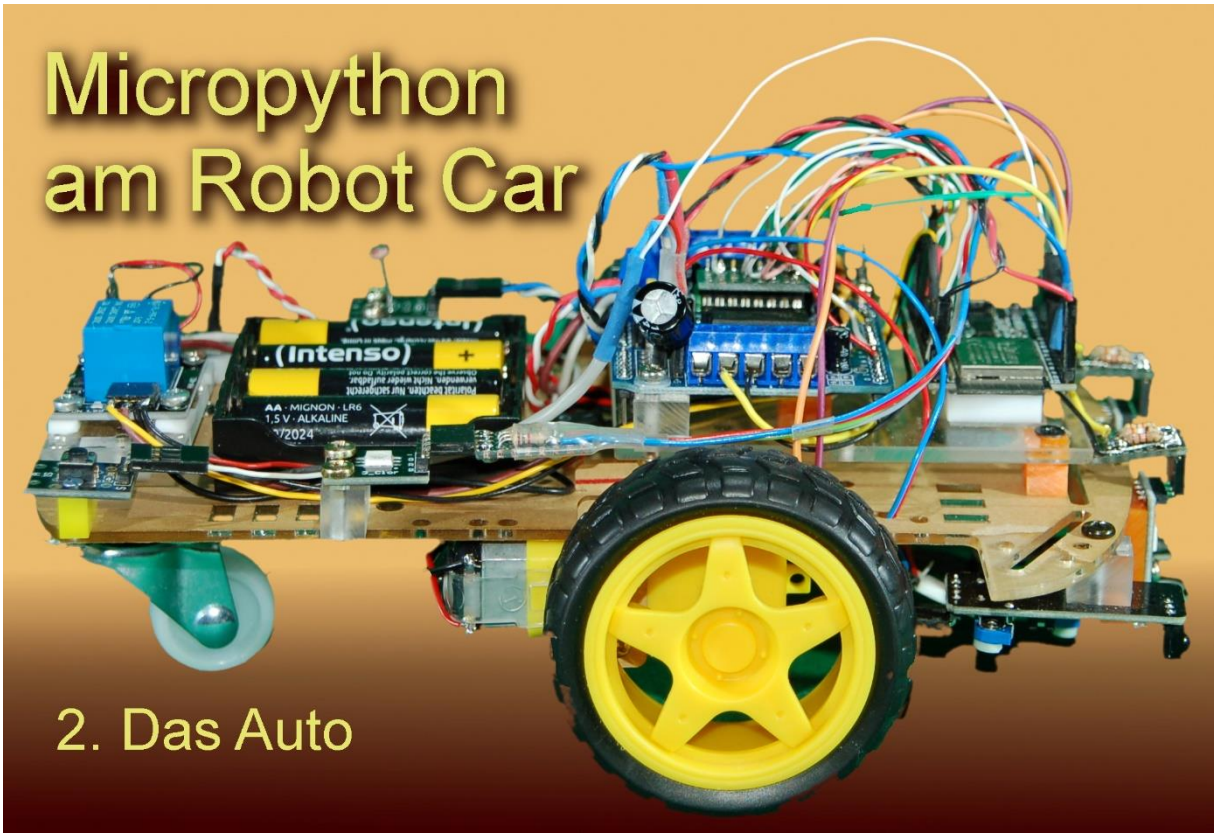


Jetzt machen wir Einen auf mobil – das E-Auto kommt. Gut bestückt mit diversen Sensoren und Aktoren bietet es ausgedehnte Möglichkeiten zur Entfaltung der Phantasie zum Thema Elektromobilität. Damit herzlich willkommen zum zweiten Teil von

MicroPython am Robot Car

Wie beim Sender im ersten Teil, besteht die Software für die Steuerung wieder aus zwei Hauptteilen. Eine speziell auf das Robot Car zugeschnittene Funktion stellt die Ansteuerung der Motorsteuerplatine zur Verfügung. Daneben kommen wieder die Tastensteuerung und das Signalmodul zum Einsatz. Eine OLED- oder LCD-Anzeige habe ich weggelassen, weil man die Zeichen auf die große Entfernung, gar nicht lesen kann. Für die Rückmeldung gibt es eine RGB-LED, welche die Zustände der Schaltung durch Farbsignale anzeigt. Aber alles der Reihe nach.

Das Titelbild zeigt viele der Ausstattungsteile, eine Liste derselben ist aber allemal übersichtlicher.

Materialliste

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

Werkzeug:
Lötkolben/Lötstation + Lötdraht
kleine Spitzzange
kleiner Seitenschneider
Schraubendreher, Kreuzschlitz + Flachschlitz
falls vorhanden, ist ein Gewindeschneider M3 hilfreich

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

Für das Testen der Funktion des Senders:
ncat im Paket nmap

Fürs Testen des Servers auf dem Fahrzeug
packetsender für Windows

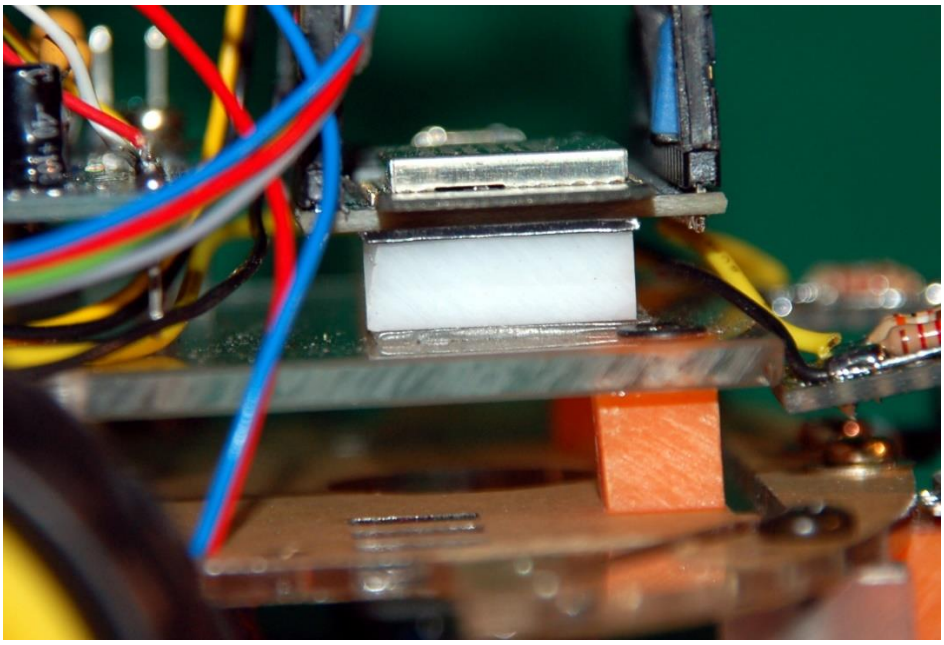
Aufbau der Mechanik und Montage der elektronischen Bauteile

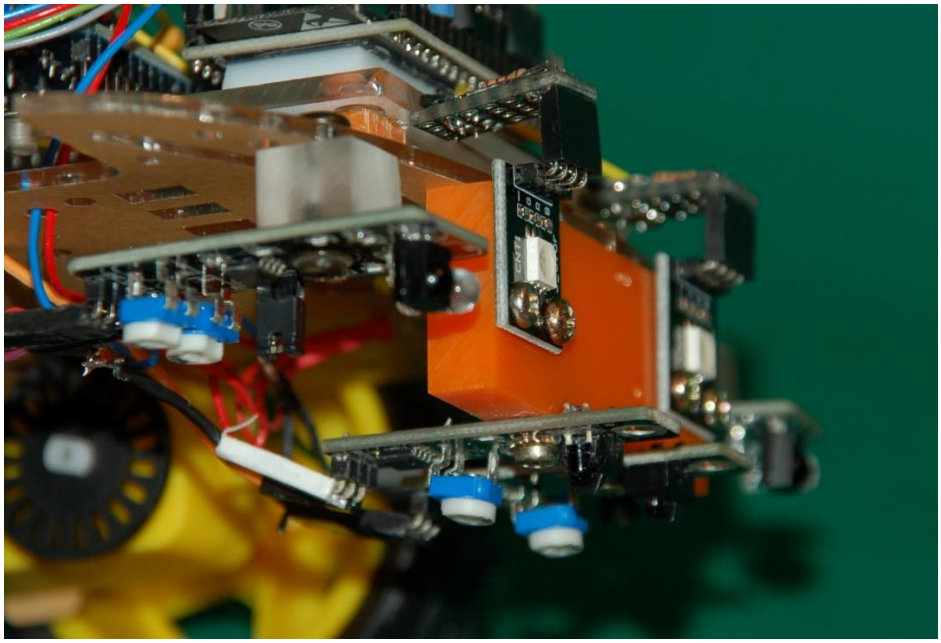
Der Aufbau beginnt mit der Montage der Motoren, des Heckrads und des Batteriekastens. Dazu braucht man keine großartige Beschreibung, es gibt sehr wenig Chancen etwas falsch zu machen.

Im Übrigen wurde bei diesem Projekt alles, was nicht gepasst hat passend gemacht. Das betrifft vor allem die Montage der diversen Hardwarekomponenten, die ausnahmslos auf Podesten oder zumindest mit Abstandshaltern montiert werden mussten. Entweder standen Anschlussdrähte unten aus der Platine wie bei den Sensoren oder es mussten ganze Stiftreihen abgestützt werden wie beim

Motortreiberboard für den Arduino, das ich für mein Projekt erfolgreich zweckentfremdet habe.. Durch diese Herangehensweise mussten auch keine neuen Löcher in die Chassis-Platte des Robot Car gebohrt werden, sondern nur in meine Montageplatten. Das Chassis ist mit seinen 2mm Materialdicke nicht gerade sehr robust.

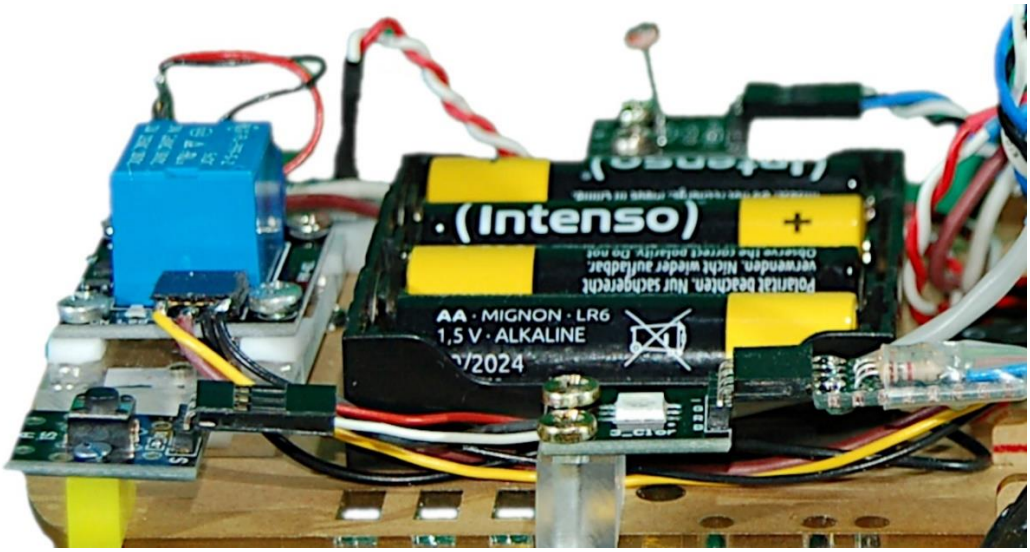
Beim ESP32-Board ist es besser, wenn Sie die unverlötete Version bekommen können. Löten Sie die Stiftreihen dann so ein, dass die Stifte nach oben zeigen. Die Unterseite des Boards ist dann relativ glatt und lässt sich sehr gut mit einem (wiederablösbaren) zweiseitigen Klebeband auf dem Untergrund befestigen. Als Abstandhalter habe ich dafür ein Stück Acrylglas hergenommen. Die Verdrahtung erfolgt dann mit Buchsenleisten oder den Femail-Enden der Jumperkabel.



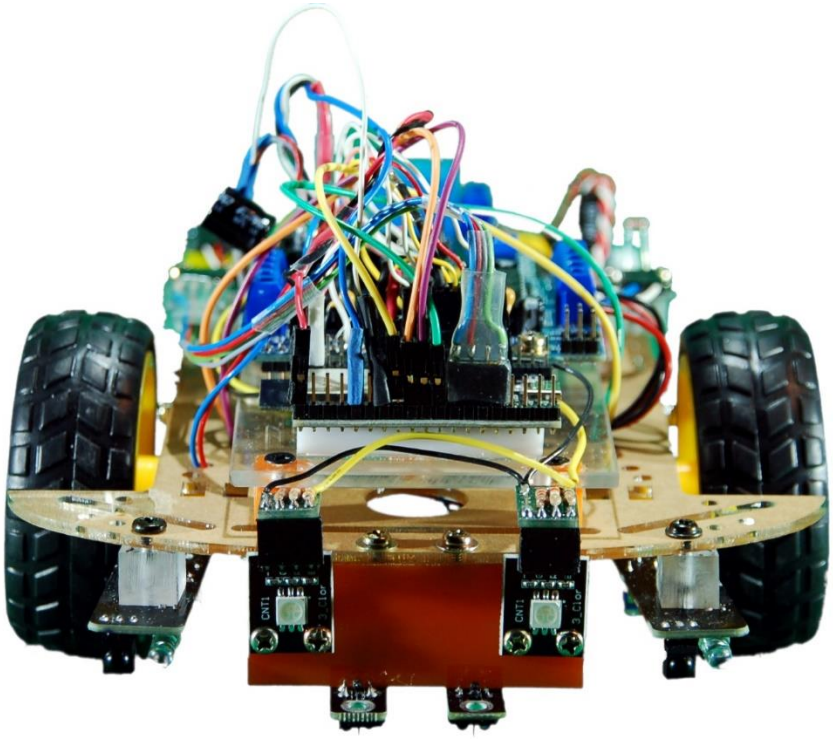


Die Motoren sind 6V-Typen und können direkt aus den 6V des Batteriekastens versorgt werden. Damit der "Kraftteil" vom ESP32 aus abgeschaltet werden kann,

habe ich ein Relais vorgesehen. Es ist highlevel-getriggert und wird von einem GPIO-Pin des ESP32 angesteuert.



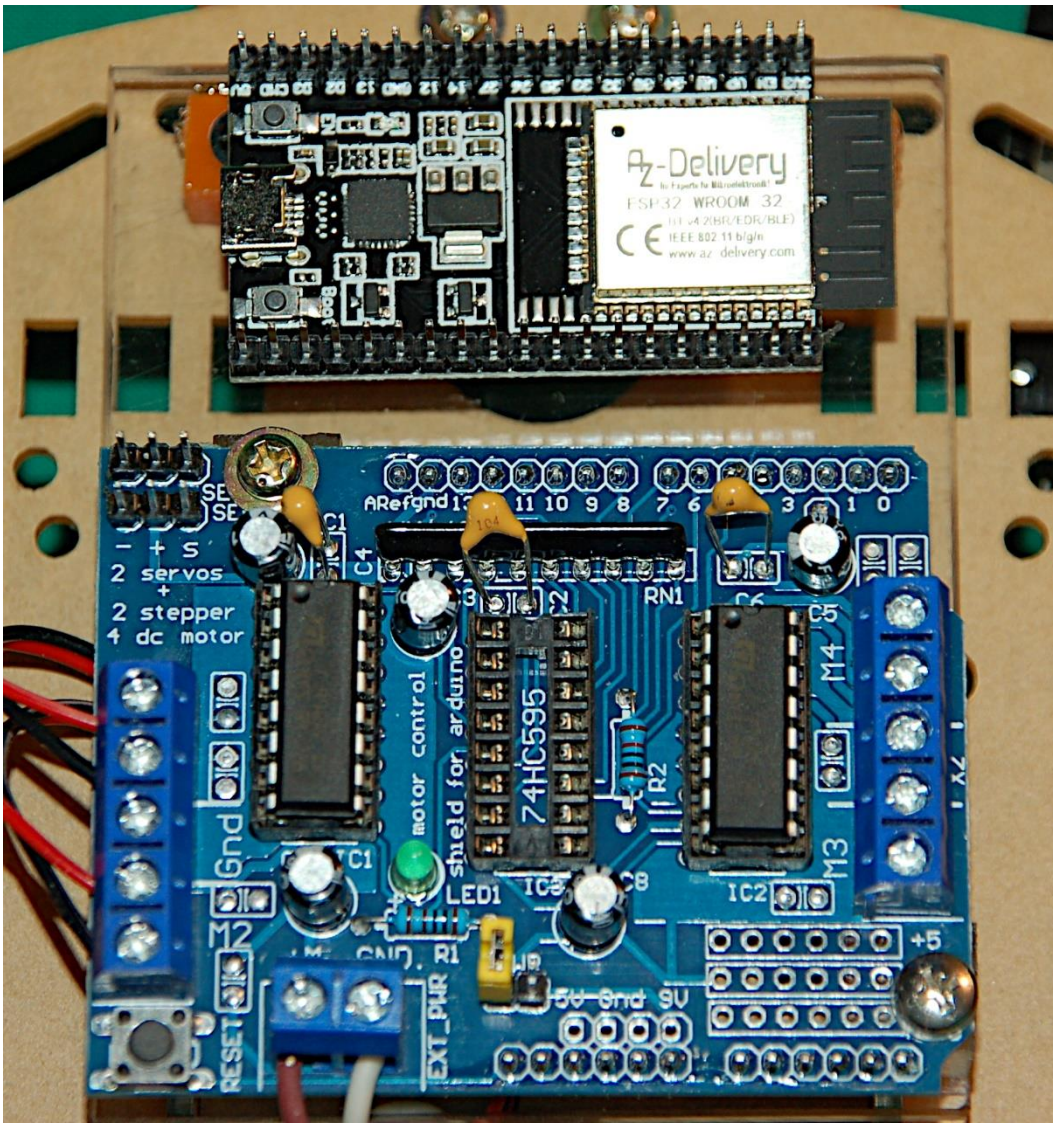
Links vorn im Bild ist die Entwicklertaste auf einem Abstandshalter montiert. Sie dient während der Entwicklung dem gezielten Programmabbruch an wichtigen Punkten. Im Produktionssystem kann der Programmablauf damit gesteuert werden. Ich mache in diesem Projekt bei der Auswahl des Startmodus Gebrauch. Vorn in der Bildmitte sitzt die Status RGB-LED. Das rot-weiße Kabel im Hintergrund kommt von der 3,3V-ReglerPlatine, rechts daneben spitzt unscharf der LDR nach oben heraus.



Die Ansicht von vorne offenbart die Positionen der Abstandssensoren (ganz außen), der Scheinwerfer und, ganz unten, der beiden Sensoren für eine potentielle Liniensteuerung.

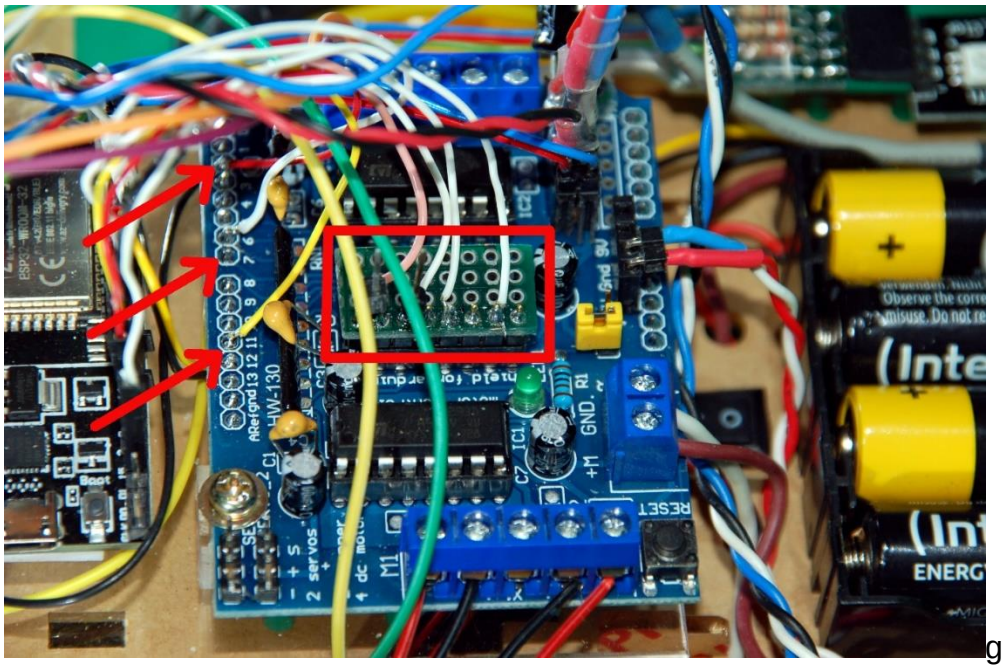
Zu guter Letzt fehlt nur noch die Motortreiberplatine mit den beiden L293 ICs. Den 74HCT595 habe ich entfernt, um über dessen Sockel besser und schneller an die

Anschlüsse der Motortreiber zu kommen. Da ich nur zwei Räder anzusteuern habe, dient einer der beiden L293 als Schalter für die Scheinwerfer und, falls Sie das möchten, für Rücklichter, die automatisch zugeschaltet werden können, sobald der "Rückwärtsgang" eingeschaltet wird. In der vorliegenden Ausbaustufe ist dieses Feature allerdings (noch) nicht enthalten.



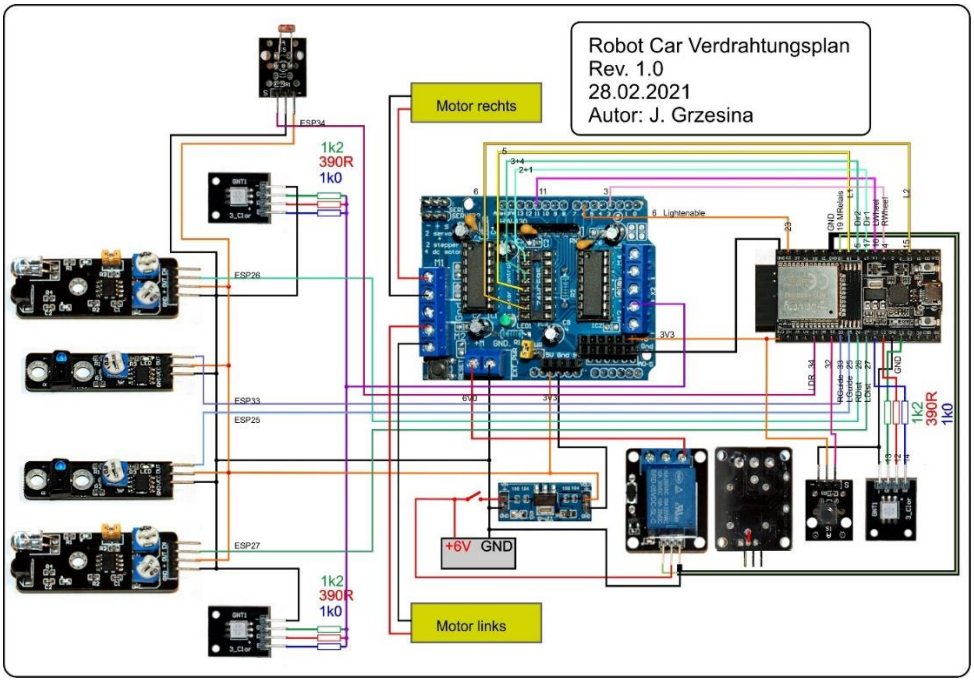
Die Verdrahtung des Shields passiert über Schraubklemmen. Links werden die beiden Motoren angeschlossen, rot, schwarz, schwarz, rot, die mittlere Klemme liegt auf GND-Level. Gegebenenfalls müssen die Anschlüsse vertauscht werden, falls die Fahrtrichtung nicht stimmt oder das Auto im Kreis fährt statt geradeaus. Unten im Bild kommt die Spannungsversorgung vom Relais. Die gesamte Platine kann über das Relais von der Batterie getrennt werden.

Das nächste Bild zeigt mit den Pfeilen die drei Leitungen, die direkt angelötet werden. im roten Kasten sieht man die Verdrahtung zum 74HCT595-Sockel, die mittels eines Stücks Lochrasterplatine umgesetzt ist. Drahtstücke von 0,6mm Durchmesser schauen unten heraus und dienen als Stecker. Wo diese Leitungen am ESP32 angeschlossen werden müssen, ist aus dem Schaltplan zu ersehen, der weiter unten folgt.



g

Nach der Mechanik wird es jetzt Zeit, nahtlos zur elektrischen Seite der Anlage überzugehen. Dafür habe ich einen Schaltplan, den Sie besser als PDF in DIN A4 herunterladen.



Die Frontsensoren habe ich mit ins Projekt aufgenommen, weil sie für ein autonomes Fahren des Autos unumgänglich sind. Vor allem die beiden Antikollisionssensoren (aka obstacle avoidance sensor) sind aber auch schon gut für den ferngesteuerten Betrieb brauchbar. Im Programm wird die schnelle Reaktion durch Interruptprogrammierung möglich. Wird die einstellbare Distanz zu einem Hindernis unterschritten, dann löst das einen Pegelwechsel am entsprechenden GPIO-Pin aus. Die dadurch hervorgerufene Programmunterbrechung (aka Interrupt) schaltet sofort die Motoren aus.

Im Schaltplan folgen nach rechts die beiden RGB-LEDs. Die habe ich gewählt, weil sie sehr helles Licht abgeben, dessen Farbton man über die verwendeten Widerstände einstellen kann. Wie man sehen kann, sind die LEDs nicht direkt mit einem GPIO-Pin verbunden, sondern werden über einen Zweig der Motorsteuerung direkt aus den 6V der Batterie versorgt. Das liegt noch gut innerhalb der Stromstärke-Toleranzen der LEDs und ergibt mehr Helligkeit.

Der LDR macht die Schaltung des Fahrlichts wie bei richtigen Autos in Abhängigkeit von der Umgebungshelligkeit möglich. Die Schaltschwelle lässt sich im Programm festlegen. Natürlich kann man das Licht auch von Hand mittels Fernsteuerung einschalten – kleiner Schnick-Schnack, der Freude macht.

Die 6V-Batterie versorgt zum einen den 3,3V-Regler, an dem die ganze Steuerelektronik dranhängt, ESP32, Sensoren und Signal-LED und zum anderen den "Kraft"-Teil der Motorsteuerplatine und die Relaisspule. Die Betriebsspannung für die beiden L293D beträgt 2,8 bis 5V und wird auch durch den 3,3V-Regler bereitgestellt. Die Steuereingänge des Motorshields kommen gut mit den 3,3V-Signalen vom ESP32 zurecht. Dadurch kann man sich diverse Pegelwandler sparen. Das Shield ist mit den weiten Bereichen der Spannungspegel an den Eingängen, von der Anordnung der Anschlüsse mal abgesehen, ohne Probleme auch unmittelbar für die ESP-Familie brauchbar.

Nach reiflicher Überlegung habe ich beschlossen, den 74HCT595 (Schieberegister mit Ausgangs-Latch und Freigabe) zu entfernen, um auf die Eingänge der L293D-Treiber direkt Zugriff zu haben. Das geht viel schneller und vor allem einfacher. Beim seriellen Betrieb müsste jeder Pegelwechsel an den 8 Ausgängen des Schieberegisters eigens durch eine Treiberroutine hineingeklopft werden. Ich kann jetzt jeden Pegel vom ESP32 aus einzeln in einem Befehl über ein GPIO-Pin ändern. Sollte das Shield später einmal anders verwendet werden, kann man den 74HCT595 ja wieder einsetzen.

Am Relaismodul muss auf der Platinenunterseite eine Verbindung vom +6V-Anschluss der Versorgung zum Schaltkontakt des Relais gelegt werden, dann reicht es, den Arbeitskontakt als Ausgang zum Motorshield zu legen.

Die "Entwicklertaste" am Eingang GPIO32 kann verschiedenen Zwecken dienen. Ich verwende sie hier wie bei allen bisherigen Projekten als Abbruchtaste an wichtigen Stellen im Programm. So dient sie hier auch wie im ersten Teil als Abbruchtaste zwischen dem Boot-Teil des Programms, in welchem die Verbindung zum WLAN-Router herstellt oder, wie in diesem Fall, auch ein eigener Accesspoint eingerichtet werden kann, und dem eigentlichen Server-Teil. Denkbar wäre in diesem Projekt auch der frühzeitige Einsatz beim Programmstart, um zwischen WLAN-Anschluss und eigenem Accesspointaufbau zu wählen. Dazu müsste ein entsprechendes optisches Signal programmiert werden, damit man weiß, wann die Taste zu drücken ist. Über die RGB-Signal-LED ist das ohne Weiteres möglich.

Optische Signale im vorliegenden Programm sind:

- Warten auf die Verbindung zum WLAN-Router (rotes Blinken)
- Abbruchoption (gelbes Dauerlicht)
- Fahrbereitschaft (blaues Dauerlicht)

- linker Liniensensor auf schwarzer Linie (rotes DauerLicht)
- rechter Liniensensor auf schwarzer Linie(grünes Dauerlicht)
- Funkverbindung steht (Fahrlicht 3x kurz blinken)

Lichtsignale können über die Klasse beep.BEEP programmiert werden. Die Klasse button.BUTTONS stellt Tastenaktionen bereit. Die Tastenobjekte werden mittels button.BUTTON32 instanziert.

Leider gab es auch bei den Vorbereitungen zu dieser Folge eine unliebsame Überraschung. Die Suche nach der Ursache hat fast einen ganzen Tag gedauert, weil ich nicht wahrhaben wollte, was sich schließlich aber als offensichtlich herausstellte. Nach fast zwei Wochen klagloser Funktion gab die Funkeinheit des ESP32 Dev KitC V4 den Geist auf, von heute auf morgen. Ich suchte nach Fehlern in meinem Programm, oft schleicht sich ein hartnäckiger Tippfehler ein, den man lange nicht entdeckt, aber nein, das war es nicht. Der WLAN-Router arbeitete einwandfrei, trotzdem bekam ich keinen Kontakt vom ESP32 zu ihm. Die Erkenntnis brachte schließlich der Einsatz eines anderen Moduls (ESP32 mini D1), das mit dem gleichen Programm einwandfrei arbeitete. Damit war die Sache nach ein paar weiteren Tests klar. Falls Ihnen also auch irgendwann etwas ähnliches widerfährt, zweifeln Sie nicht gleich an sich selbst, sondern stellen Sie ruhig auch mal die ordnungsgemäße Funktion eines Bauteils in Frage. Das kann ein Transistor, ein Elko, ein IC oder wie hier das Sendemodul eines ESP32 sein.

Kommen wir jetzt zur Programmierung. Das gesamte Programm besteht wie bereits erwähnt aus zwei Hauptteilen. Die Verbindungsaufnahme beziehungsweise die Bereitstellung eines Accesspoints stellt den Bootanteil dar. Wenn der abgearbeitet ist, lädt das Programm den eigentlichen Serverteil nach, sofern kein Abbruch durch die "Entwicklertaste" erfolgt.

Der Bootanteil selbst besteht aus der essentiellen Bootsequenz boot_essential.py, dem Import und der Initialisierung weiterer Module für die Kommunikation, interfaces.py und dem eigentlichen Teil für die Verbindungsaufnahme, wifi_connect_server.py oder accesspoint.py. Im Anschluss folgt die "Entwicklersequenz" break_section.py, die einen gezielten Programmabbruch an dieser Stelle erlaubt. Nach einem Abbruch stehen alle, bis hierher erfolgten Deklarationen, zum Experimentieren über REPL bereit.

Wird nicht abgebrochen, startet die exec-Anweisung den Serverteil server.py. Dieser Teil beinhaltet die Vorbereitung für den Empfang von UDP-Datagrammen von der Senderseite und die Routinen für die Decodierung und Abarbeitung von empfangenen Anweisungen. Der eigentliche Serverteil ist ähnlich wie der Senderteil aus der letzten Folge sehr kurz und umfasst im Wesentlichen drei Zeilen.

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

Zu Beginn müssen die Pin-Belegungen deklariert werden. Danach folgt die Festlegung der Geschwindigkeitsstufen mit deren Zuordnung zu den PWM-Werten, die von 0 bis 1023 reichen dürfen. Die Motoren fangen erst bei einem Wert von ca. 500 an, sich zu bewegen. Das hängt von der eingestellten PWM-Frequenz und natürlich von den Motordaten selbst ab. Experimentieren ist hier angesagt. Die

Umsetzung im Programm ist einfach gehalten, man gibt den maximalen und minimalen PWM-Wert sowie eine minimale und maximale Fahrstufe an. Das Programm erzeugt daraufhin eine Liste von PWM-Werten, die einer bestimmten Fahrstufe entsprechen. Der Inhalt dieser Liste wird am Terminal ausgegeben und kann dort kontrolliert werden. Analog werden die Einstellungen für die Lenkung festgelegt. Im Programm wird bei der Berechnung überwacht, dass der maximale PWM-Wert unter Berücksichtigung der Lenkung 1023 nicht übersteigen kann. Die Lenkung erfolgt durch Addition des PWM-Geschwindigkeitswerts mit dem PWM-Wert der Lenkung auf dem Außenrad der Kurvenbahn. Am Innenrad erfolgt Differenzbildung. Das heißt, dass das linke Rad bei einer Rechtskurve schneller dreht als das rechte und umgekehrt. Welches Verhältnis zwischen Geschwindigkeits- und Lenkwerten optimal ist, ermitteln Sie am besten durch Versuche. Denkbar sind für die Lenkung drei Ansätze. Ich habe hier die mittlere Variante gewählt.

1. Nur das Außenrad dreht schneller
2. Das Außenrad dreht schneller, das Innenrad langsamer, minimal gar nicht
3. Das Außenrad dreht schneller, das Innenrad kann auch in die Gegenrichtung drehen

Hier der entsprechende Schnipsel aus der service()-Funktion des Serverteils.

```
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
```

Zur Vermeidung von Fehlfunktionen und Programmabstürzen sind zwei Sicherungen eingebaut. Der vom Sender übermittelte Wert in value wird notfalls auf Sdmax zusammengestutzt. Die beiden fett formatierten Zeilen sorgen dafür, dass der PWM-Wert am Innenrad kleinsten falls Null wird. Wenn Sie die beiden Zeilen weglassen, landen Sie bei der Option 1. Der Kurvenradius ist bei der Option 2 kleiner als bei der Option 1.

Die Anzahl der im Serverteil deklarierten Fahrstufen muss nicht mit der Anzahl im Senderteil übereinstimmen. Für den Anfang ist es vielleicht besser, die Fahrstufenanzahl im Sender niedriger und im Server=Fahrzeug höher zu halten. Das zwingt zu moderaterer Fahrweise. Der umgekehrte Fall wird vom Fahrzeug ausgebremst, weil bei zu hoher Fahrstufe vom Sender ein Fehler am Fahrzeug die Steuerung durch einen Abbruch des Programms unmöglich machen würde. Wie viele Fahrstufen Sie einprogrammieren bleibt wieder Ihrem persönlichen Empfinden überlassen.

Ich habe während der Entwicklung die gesamte Prüfung der Funktion beider Programmteile, so weit möglich, einzeln vorgenommen. Die nächste Teststufe ist nach den ersten Programmtests das "Trockendock". Damit das Fahrzeug nicht abhaut, wenn man die Motoren aktiviert, hatte ich das Gefährt hochgelegt, damit die

Räder keinen Bodenkontakt hatten. Als auch das alles wunschgemäß lief, wurde der Bootteil boot_server_wlan.py als boot.py auf den ESP32 geschickt, ebenso der Serverteil server.py, zusammen mit den nötigen Modulen, beep.py und button.py. beim nächsten Neustart des ESP32 fuhr das System autonom hoch. Die Steuerung funktionierte wie erwartet.

Zum Test der einzelnen Funktionen auf dem Fahrzeug habe ich die Freeware packetsender verwendet. Man kann hier Steuerbefehle von Hand eingeben, via UDP an den Server auf dem Robot Car schicken und die Reaktion von Programm und Maschine dort testen. Danach übernimmt die Handsteuerung das Kommando, wenn alles perfekt funktioniert.

Im Folgenden finden Sie die Listings der beiden wesentlichen Programmteile boot.py und server.py zur genauen Durchsicht. Vergessen Sie nicht, die SSID Ihres WLAN-Routers und das zugehörige Passwort sowie eine IP, die Maske, das Gateway und den DNS Ihres lokalen Netzwerks einzutragen. Ich habe es hier vorgezogen die IP fest am ESP32 einzustellen und nicht vom DHCP-Dienst des Routers zu beziehen. Für Server ist das so üblich.

boot.py (WLAN-Verbindung herstellen)

```
# 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*******************
```

Wenn der ESP32 mit eigenem Accesspoint laufen soll, ersetzen Sie einfach den fett formatierten Teil durch folgende Sequenz. Der Sender muss dann auch für diese Betriebsart gestartet werden.

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)
```

Nachdem eine Verbindung bereitsteht, kann der Server gestartet werden. Das passiert durch die exec-Anweisung am Ende des Bootteils. Die Datei server.py muss sich auf dem ESP32 im Rootverzeichnis \ befinden. Die ersten beiden Importe ermöglichen den isolierten händischen Test der service()-Routine des Serverteils ohne dass eine Netzverbindung existieren muss. Im Produktionssystem werden diese Imports einfach übergangen und durch solche im Boot-Teil ersetzt.

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)
```

Die Funktion **service(action)** bekommt von der UDP-Empfangsschleife den wesentlichen Teil der vom Sender abgesetzten Codes. Das sind das Befehls-Prefix und der Wert nach dem Doppelpunkt incl. Vorzeichen, der damit maximal 3-stellig sein kann, also insgesamt bis zu 5 Zeichen. Diese Informationen werden in einer Folge von if-Konstrukten decodiert und in Aktionen umgesetzt. Besonders dringende Reaktionen, die durch Pegelwechsel der Frontsensoren angefordert werden, werden durch Programmunterbrechungen und die zugeordneten IRQ-Serviceroutinen asynchron bedient.

Der Server versteht zum Beispiel die folgenden Kommandos vom Sender.

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 steht für **d**irection
d:-6 dasselbe nach links
l:1 (kleines L) licht an, aus, an, aus…
m:1 Motoren an, aus, an, aus …

Wenn die Abstandssensoren ansprechen, wird das Motorrelais ausgeschaltet. Das erledigt die Interruptserviceroutine **stopMotor**().

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

Eine Referenz auf die ISR wird bei der Definition der Pin-Objekte **LDist** und **RDist** als IRQ-Quelle dem benannten Parameter **handler** als Callback-Funktion übergeben. Die Programmunterbrechung wird hier angefordert, wenn an den Pins der Pegel von 1 auf 0 fällt. Für steigende Flanken würde Pin.IRQ_RISING angegeben. Beide Flanken werden durch Oderieren der Konstanten erkannt.

Falls während des Betriebs die Umgebungshelligkeit abnimmt, schaltet sich automatisch das Fahrlicht ein, unabhängig davon ob vorher das Kommando "Licht

an" von der Handsteuerung aus erfolgt ist oder nicht. Wichtig ist nur, dass die Handsteuerung in Betrieb ist. Dann werden nämlich ständig v- und d-Kommandos gesendet, die den Server in die Decoderschleife zwingen. In deren Verlauf wird auch der LDR am Fahrzeug abgefragt.

Für das autonome Fahren müssen die Funktionen der Interrupt-Service-Routinen (aka ISR) der Linienfolger noch mit Code bestückt werden, der darauf abzielt, eine automatische Lenkbewegung hervorzurufen. Hier sind Sie im Einzelnen zum Forschen aufgefordert. Wie Lenkbewegungen am Fahrzeug hervorgerufen werden, wissen Sie ja bereits.

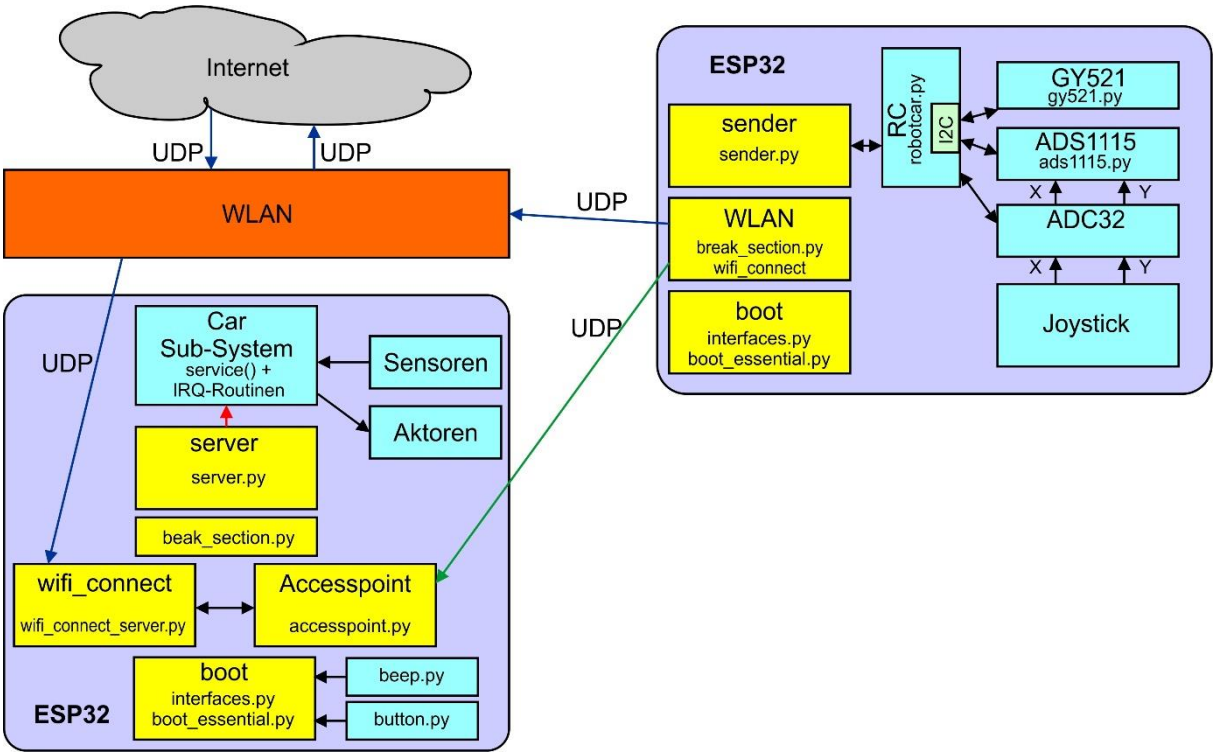
Probleme in der Behandlung bereiten die Abstandssensoren, weil die beide keine eindeutigen Flanken liefern, weder beim Auftreten einer zu großen Annäherung noch beim Entfernen vom Hindernis. Messungen mit dem DSO haben gezeigt, dass stets eine Impulsfolge mit wechselndem Impuls-Pause-Verhältnis auftritt (mittlere Periodendauer ca. 1ms). Diese Impulsfolgen liegen je nach Abstand zum Hindernis zwischen ca. 50ms und 280ms auseinander, dazwischen habe ich GND-Potential gemessen. Die Ruhelage der Abstandssensoren ist der HIGH-Pegel. Es muss geprüft werden, ob hier nicht ein Ultraschall-Abstands-Sensor oder ein Time-Of-Flight-Sensor besser geeignet ist.

Diese Frage wird vielleicht im nächsten Beitrag beantwortet, in dem es auf jeden Fall um eine alternative Steuerungsmöglichkeit des Robot Car gehen wird. Bis dahin wünsche ich viel Spaß beim Bau und den ersten Fahrversuchen mit dem MicroPython-gesteuerten Robot Car.

Hier finden Sie noch eine Liste mit allen verwendeten Programmteilen. Die Grafik verdeutlicht deren Einsatz.

Datei	Klasse	Beschreibung
beep.py	BEEP	Methoden für optische und akustische Signale
button.py	BUTTONS, BUTTON32 BUTTON8266	Methoden zu Einlesen von Tasten
boot_essential.py		grundlegende Importe zur Systemsteuerung, Zeitsteuerung und GPIO-Behandlung
interfaces.py		Importe für Signalsteuerung, Tastenbedienung und Displaysteuerung
break_section.py		Abbruchoption über eine Taste während optischem Signal
wifi_connect_server.py		allgemein eine Verbindung zu einem WLAN-Accesspoint als Server herstellen
accesspoint.py		startet den ESP32 als Accesspoint
server.py		Serverprogrammteil mit UDP-Empfang, Decodierung der Anweisungen und Umsetzen in Steueraktionen für das Fahrzeug. Abtasten der Sensoren und Reaktion auf Distanzereignisse durch IRQ-Routinen.

boot_server_wlan.py		enthält alle Programmteile, die zum Booten des Systems und zum Einrichten der WLAN-Verbindung nötig sind, incl. Abbruchbedingung. Der Inhalt kann komplett in die Datei boot.py für einen autonomen Start übernommen werden.



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