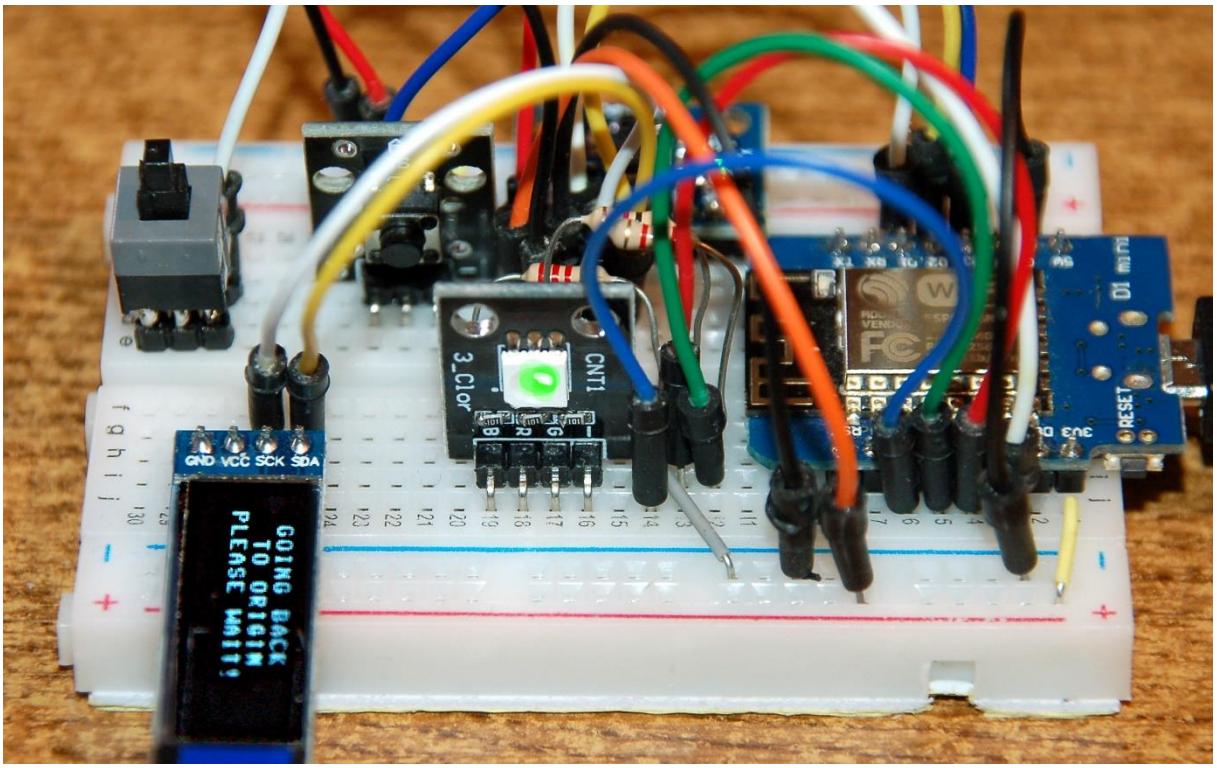

Fernsteuerung

Dieser Beitrag ist auch als PDF-Dokument erhältlich.

Nach dem Aufbau der Mechanik zum Joyball-Wizzard in Folge 1 und der Programmierung des Spiels in Folge 2 werden wir heute ein Feature des ESP8266 integrieren, das bisher keine technische Bedeutung für das Spiel hatte. Aber der Ansatz, den ich darstellen werde, hat nicht nur spielerischen Nutzen, sondern lässt sich für vielerlei weitere Projekte einsetzen, wenn es darum geht, Daten bidirektional per Funk auszutauschen. Die Rede ist nicht von einer RS232-Verbindung, obwohl das Ganze ähnlich abläuft. Kurzum, wir ersetzen das Kabel der RS232-Connection durch die Funkverbindung über WLAN und lassen das Protokoll UDP sprechen. Damit willkommen zu einer weiteren Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute Teil 3

Joy Ball Wizzard goes WLAN

Genau genommen müsste das Spiel umgetauft werden, denn den Joystick habe ich in Urlaub geschickt und dafür einen MPU6050 engagiert. Der MPU6050 ist ein Chip, der Beschleunigungen messen kann, also ein Accelerometer (von engl. accelerate = beschleunigen). Das Modul GY-521 enthält einen solchen Chip, der über den I2C-Bus angesprochen werden kann.

Das Accelerometer GY-521 als Bewegungssensor

Beschleunigungswerte geben an, um welche Geschwindigkeit eines Körpers in Meter pro Sekunde während einer Sekunde zunimmt. Die Einheit ist (1m/s)/s also 1m/s². Im Schwerefeld der Erde wird ein Körper im freien Fall mit g = 9,81m/s² beschleunigt. Wenn ich einen Stein von einem hohen Turm fallen ließe, bitte beachten Sie den Konjunktiv, dann hat er, der Stein natürlich, nicht der Turm, nach einer Sekunde eine Geschwindigkeit von 9,81m/s, nach einer weiteren Sekunde 19,62m/s und so weiter. Exakt betrachtet stimmt diese Aussage natürlich nur, wenn der Stein im Fallturm zu Bremen im Vakuum fällt, weil ihn der Luftwiderstand sonst umso stärker abbremst, je schneller er wird. Beachten Sie bitte ferner, dass der fallende Stein kriminalpolizeiliche Untersuchungen zur Folge haben könnte, vor allem dann, wenn der Stein unterhalb des Turms versehentlich durch den Kopf einer Person abrupt abgebremst würde.

Was hat das alles mit dem Spiel zu tun? Nun, seit Newton wissen wir, dass Kraft gleich Masse mal Beschleunigung ist, als Formel: F=m • a. Das heißt, dass eine Beschleunigung a zu einer Kraft F auf die Masse m führt. Genau das ist das Messprinzip im MPU6050. Eine Masse m, die am Ende einer "Blattfeder" befestigt ist, widersetzt sich der Bewegungsänderung, die durch die Beschleunigung hervorgerufen wird. Die Trägheit der Masse führt zu einer verformenden Kraft auf die Feder. Aber nicht nur eine Bewegungsänderung kann eine Verformung der Feder bewirken.

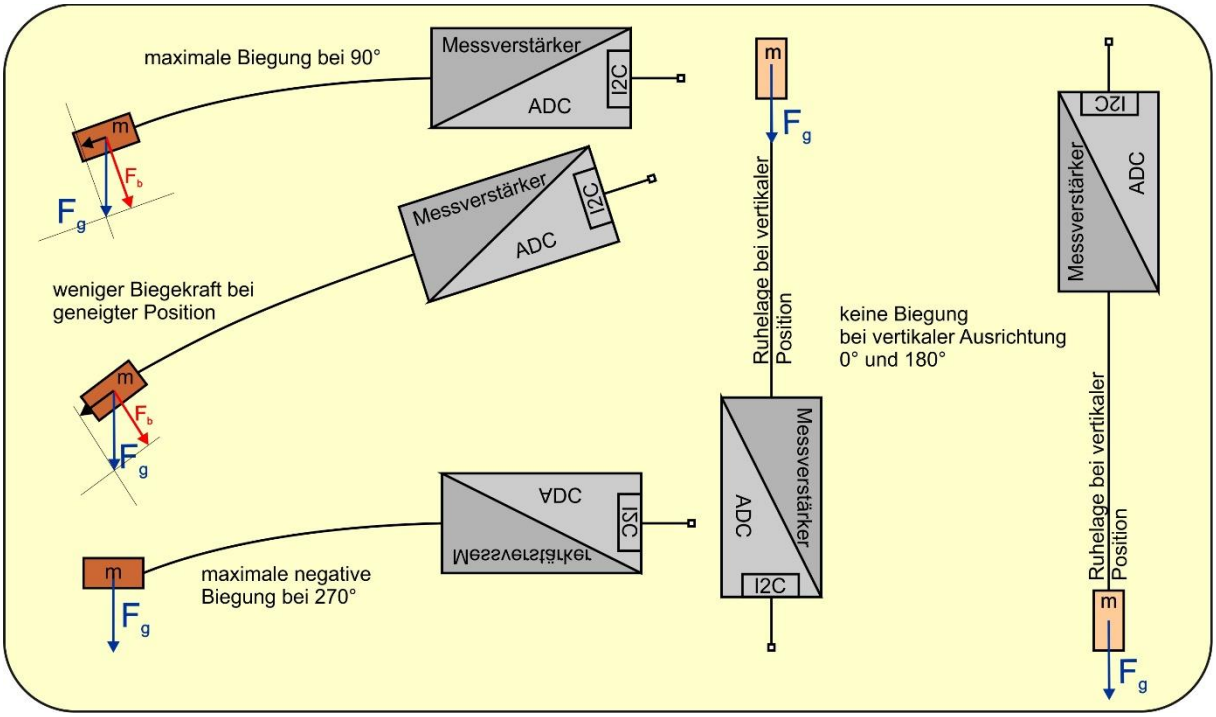

Abbildung 1: Beschleunigungsmesser mit Piezoelement

Durch die Gewichtskraft F_g, die aus der Erdbeschleunigung g = 9,81m/s² resultiert, wir die Masse m, die an einem federnden Piezoelement befestigt ist, aus ihrer Ruhelage ausgelenkt. Durch die Verbiegung des Piezoelements entsteht an dessen Oberfläche eine Spannung, die verstärkt, digitalisiert und über das I2C-Interface verfügbar gemacht wird. Von diesen Einheiten enthält der MPU drei Stück, die nach den Raumrichtungen ausgerichtet sind. Wir nutzen davon die x-Richtung für seitliche Neigung (Rollen = Roll) und die y-Richtung für vor und zurück (Nicken = Pitch).

Durch die Neigung aus der vertikalen Ebene ändert sich der Betrag der Kraft, die senkrecht zum Arm des Piezoelements wirkt und dieses verbiegt, in die eine oder in die andere Richtung. Die Messwerte schwanken dadurch zwischen ca. +/- 16000. Wie diese Werte dazu benutzt werden können, unser Spiel zu steuern, das schauen wir uns gleich nach der Besprechung der Hardware für die Fernsteuerung an. Die Mechanik und Hardware sowie die grundlegende Softwarebeschreibung für das Spiel selbst ist in den vorangegangenen Folgen (1 und 2) umfassend beschrieben.

Hardware

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	GY-521 MPU-6050 3-Achsen-Gyroskop und Beschleunigungssensor
2	KY-004 Taster Modul
1	KY-009 RGB LED SMD Modul
1	MT3608 DC-DC Netzteil Adapter Step up Modul
2	Widerstand 2,2kΩ
1	Widerstand 560Ω
1	Handy-Akku, Li 3,7V 600mAh
1	TP4056 Micro USB 5V 1A Laderegler Lithium Li - Ion Batterie Charger Modul
diverse	Jumperkabel
1	Minibreadboard oder Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins

Alle Teile passen auf ein Mini-Breadboard. Als OLED-Display kann gegebenenfalls das aus Folge 2 verwendet werden, weil die Anzeigefunktionen alle per UDP-Verbindung zur Fernbedienung gesendet werden. Die hier aufgeführten Teile beziehen sich ansonsten alle nur auf die Fernsteuerung.

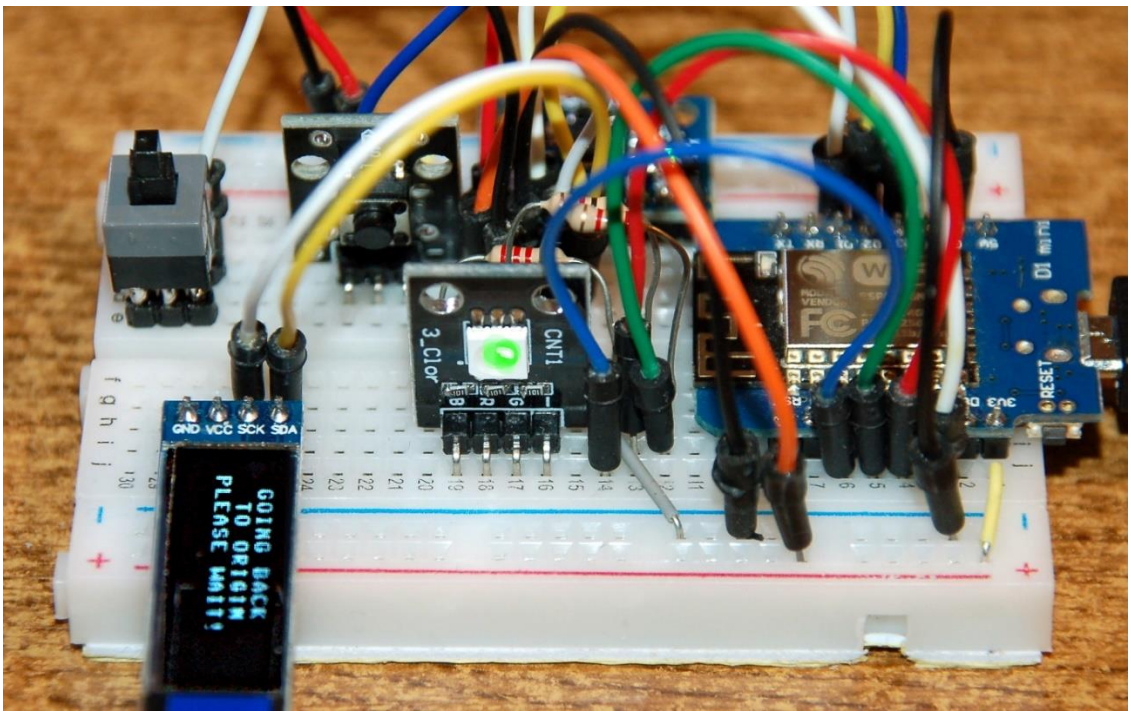

Abbildung 2: Fernsteuerung

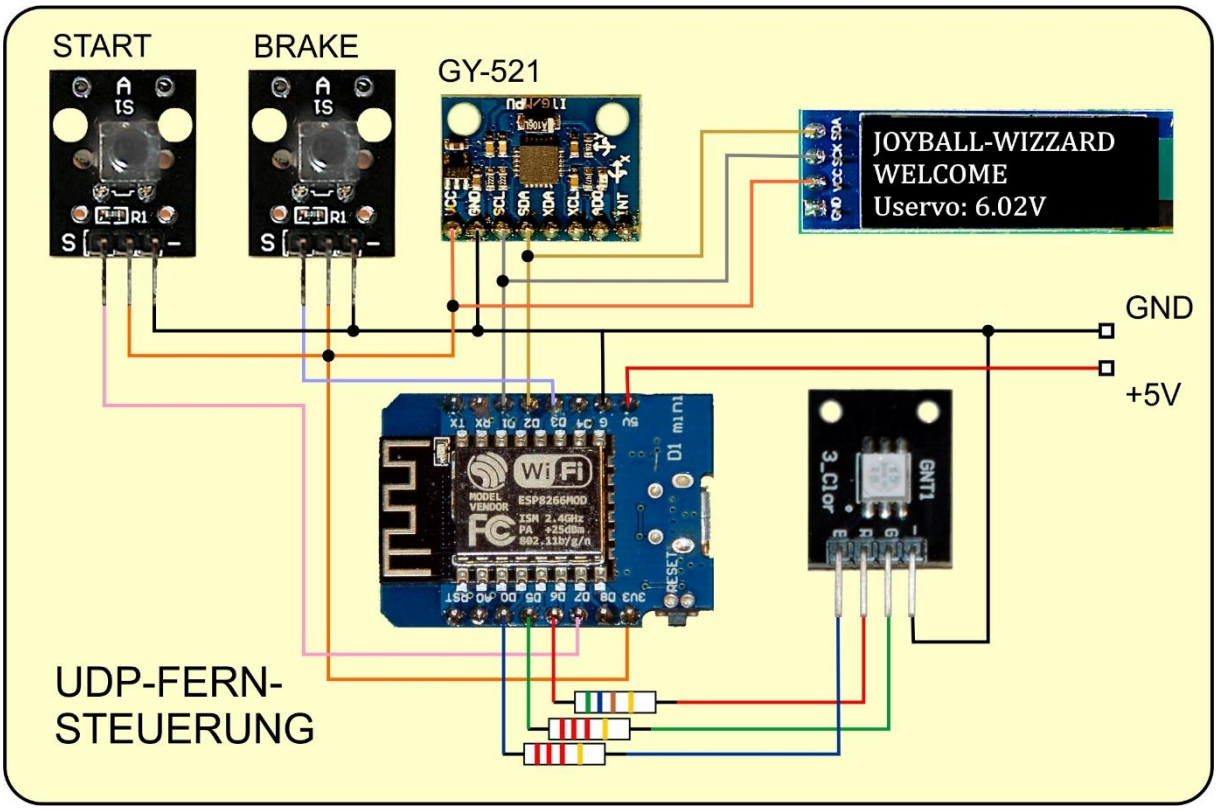

Abbildung 3: Fernsteuerung - Schaltung

Abbildung3 zeigt den grundsätzlichen Schaltungsaufbau. Zur Energieversorgung hatte ich vor, den Lithium-Akku aus einem alten Handy zu verwenden. Das ursprünglich geplante Upcycling in Verbindung mit einem Akku-Lademodul entwickelte sich aber zu einem Irrweg.

Manche Ladeteile haben sogar einen integrierten Step-Up-Wandler, der die 3,7V des Akkus auf konstante 5V anhebt. Es gibt dabei nur ein Problem. Die Dinger schalten nach ca. 30 Sekunden den 5V-Ausgang aus unersichtlichen Gründen ab. Ein versuchtes Workaround mit einem Hallo-Wach-Puls vom ESP8266 brachte letztlich doch keine Lösung. Deshalb habe ich in meinem Ansatz neben dem einfacheren Laderegler mit dem TP4056 einen separaten Boost-Converter eingesetzt, der direkt am Akku angeschlossen ist und die 5V-Versorgung des ESP8266 sicherstellt. Als Ladekabel kann ein USB-Kabel mit Micro-B-Stecker dienen, das an ein 5V-Handy-Ladeteil angeschlossen wird. Dadurch wird der Akku auf eine Spannung von ca. 4,2V gebracht. Das Ladeteil schaltet dann selbsttätig ab. Sinnvoll, ist auch ein kleiner Schalter in der 5V-Leitung zwischen Akku und Fernsteuerung, so wie es die Abbildung 4 zeigt.

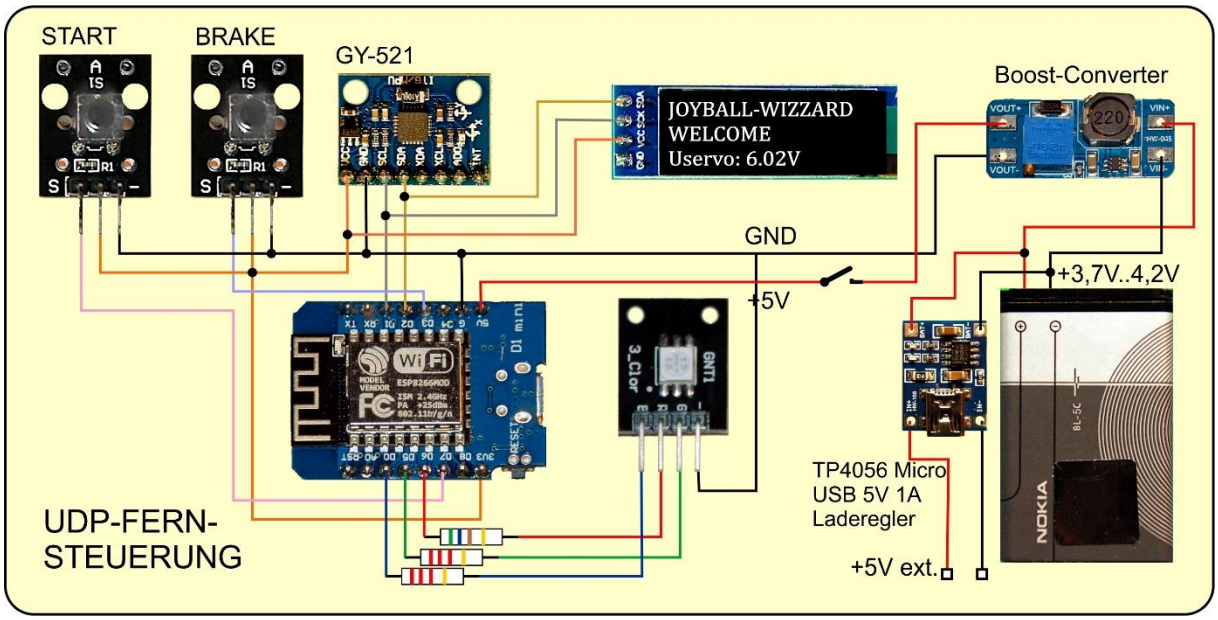

Abbildung 4: Fernsteuerung - Schaltung mit Akku-Laderegler und Step-Up-Converter

Zur Ergänzung kommt hier noch die abgespeckte Schaltung für das Spiel

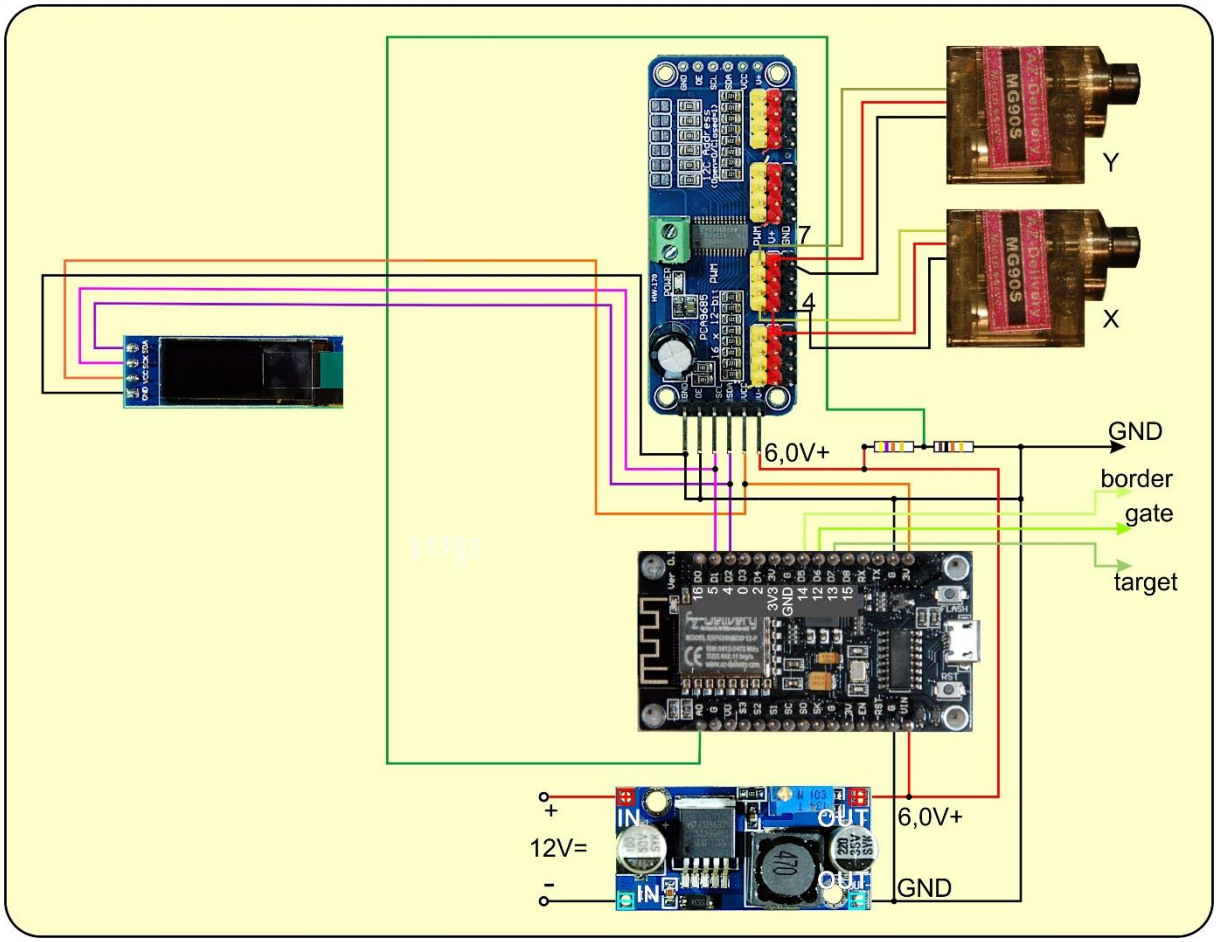

Abbildung 5: Schaltung für Remote-Zugriff

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP8266/ESP32:

MicropythonFirmware
Bitte eine Stable-Version aussuchen (ESP8266 mit 1MB Version 1.18 Stand: 05.03.2022)

Die MicroPython-Programme zum Projekt:

pca9685.py: Modul mit der Klasse SERVO, PCA9685-Treiber
ssd1306.py: OledTreiber
oled.py: OLED-Klasse
gy521_level.py: Treiber für das Accelrometer-Modul GY-521
joyball_funk.py: Software für die Spieleinheit
joyball_sensor.py: Software für die Fernsteuerung

Sonstige Software:

Packet Sender Downloadseite
Packet Sender Windows Install Version
Packet Sender Windows Portable
Packet Sender Linux

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 05.02.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro

fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Das Programm für das Spiel

Das Betriebsprogramm joyball.py für das Spiel aus Folge 2 erhält ein paar Anpassungen und einen größeren Zusatz für die WLAN-Verbindung und die UDP-Einheit. Ich sage hier bewusst nicht UDP-Server, denn die Grenzen zwischen Server und Client sind bei UDP verwaschen. Jeder Server kann auch Client spielen und umgekehrt. Aber dazu kommen wir später. Gehen wir zunächst durch das Programm und schauen uns die markanten Stellen näher an.

```
import os,sys
from machine import Pin, reset, I2C, ADC
from time import sleep, ticks_ms
from pca9685 import SERVO
# from ads1115 import ADS1115
# from oled import OLED
import esp
esp.osdebug(None)
import gc
gc.collect()
import network
import socket
import struct
```

Bei den Importen fallen die auskommentierten Zeilen weg, dafür kommen drei neue hinzu, **network** für die WLAN-Verbindung oder den ESP8266-eigenen Accesspoint, **socket** für das UDP-Protokoll und **struct** fürs Encoding und Decoding von Zahlenwerten. Letzteres spart beim Transfer Übertragungszeit, weil statt der bis zu 5 ASCII-Zeichen der Zahlen nur HIGH-Byte und LOW-Byte des Werts gesendet werden müssen.

Die Instanziierung des I2C-Objekts erfährt keine Änderung, auch nicht der nachfolgende Abschnitt mit der Deklaration der GPIO-Pins.

Weiter können die Instanzen **d** für das OLED-Display und **joy** für den ADC wegfallen. Das OLED-Display wird in der Fernsteuerung eingesetzt. Für die Betriebsspannungsmessung setzen wir den ESP8266-internen ADC ein. Der Eingang A0 verträgt Spannungen bis 3,3V. Der Spannungsteiler aus den beiden 10kΩ-Widerständen kann also beibehalten werden, um die 6V-Versorgungsspannung auf 3V zu reduzieren.

```
servo=SERVO(i2c)
# d=OLED(i2c,128,32)
# joy=ADS1115(i2c,0x48)
adc=ADC(0)
adc.read()
```

Die Methoden **bump**() und **blink**() bleiben unverändert, während **goToOrigin**() an den Stellen eine Änderung erfährt, an denen Ausgaben auf das OLED-Display erfolgen sollen. Anstatt die Daten direkt zum Display zu schicken, gehen diese via Netzwerkverbindung an die Fernsteuerung, wo sie angezeigt werden. Um auch hier Übertragungszeit zu sparen, und zwar im großen Stil, geht nur ein ASCII-Zeichen als Kenner gefolgt von einem ":" und eventuell einem Zahlenwert über den Äther. In der Vorbereitung werden alle Daten als Bytefolge codiert. Die Ausgabebefehle, die bisher an diesen Stellen standen, wurden in die Betriebssoftware der Fernsteuerung verpflanzt.

```
def goToOrigin():
    global trigger, gameOver,start
    oePin.value(0)
    gate.irq(handler=None)
    border.irq(handler=None)

    # Warten, Kugel geht zurueck an den Start
    text="R:0".encode()
    s.sendto(text,address)
    getData()
    servo.writePulseTimeSlice(7,0,1800)
    sleep(1.2)
    servo.writePulseTimeSlice(4,0,350)
    sleep(1.2)
    servo.writePulseTimeSlice(7,0,850)
    sleep(1.2)
    servo.writePulseTimeSlice(4,0,1500)
    sleep(1.2)
```

```
    trigger=True
    gameOver=False

    # Warten auf Tastendruck zum Spielstart
    text="G:0".encode()
    s.sendto(text,address)
    waitForButtonIs0()
    points=vorgabe

    # Meldung des Spielbeginns
    text="S:".encode()
    text=text+struct.pack("H",points)
    s.sendto(text,address)

    gate.irq(handler=bump,trigger=Pin.IRQ_FALLING)
    border.irq(handler=bump,trigger=Pin.IRQ_FALLING)
    start=ticks_ms()
    score=0
```

Bemerkenswert ist der Einsatz der Methode **pack()** aus dem Modul **struct**. Der Parameter "H" ist eine Art Formatanweisung. Er besagt, dass der Integer-Wert **points** als Folge von zwei Bytes codiert werden soll. Folgende Zeilen im Terminal von Thonny demonstrieren das. Die Byteanordnung ist per Default **little endian**, das LSB kommt also zuerst.

```
>>> import struct
>>> struct.pack("H",1035)
b'\x0b\x04'
```

Die Methode gewinnt durch die ausgewanderten Textausgaben etwas an Klarheit. Gleichzeitig geht aber der Nebenbei-Effekt der Texte als Erläuterung zum Programm verloren. Deshalb kann es sinnvoll sein, ein paar Kommentarzeilen einzufügen.

Vorbereitend, für die Abwicklung des Datentransfers, wurden drei neue Funktionen erstellt.

```
def getData():
    try:
        rec,adr=s.recvfrom(6)
        if rec is not None:
            resp=struct.unpack("BHH",rec)
            rec=""
            return resp
    except:
        #print(".",end="")
        return None
```

getData() holt bis zu 6 Bytes aus dem Empfangspuffer des UDP-Sockets **s**. Dessen Deklaration erfolgt weiter unten. Dieses Auslesen ist mit einem Timeout verbunden. Wenn innerhalb von 100ms kein Zeichen im Puffer bereitsteht, wird eine **OSError-Exception** geworfen, die wir mit **try** abfangen müssen, um einem Programmabbruch vorzubeugen. Liegt eine Nachricht von der Fernsteuerung vor, dann landet sie in der

Variablen **rec** als Bytes-Objekt, während **adr** die Socketadresse des Absenders wie üblich als Tuple aufnimmt.

Zur Sicherheit stellen wir noch fest, dass **rec** nicht eine Referenz auf das große Nichts enthält und entpacken dann, was uns die Fernsteuerung geschickt hat - ein Byte für den Tastencode und zweimal zwei Bytes für den x- und y-Wert. Die Funktion gibt die Referenz auf das Tuple der entpackten 3 Zahlenwerte zurück. Im except-Zweig könnte auch einfach nur **pass** stehen. Um aber zu erkennen, was sich tatsächlich abspielt, geben wir einen Punkt aus. Das **end=""** unterdrückt den Zeilenvorschub. Der Rückgabewert in diesem Fall ist **None**, eben nichts.

```
def getButton():
    request=getData()
    if request is not None:
        button,*_=request
        return button
    else:
        return None
```

getButton() schnappt sich einen Datensatz, isoliert daraus die Tasteninformation durch Entpacken des Tuples, das **getData**() zurückgibt. Interessant ist die Schreibweise **button, *_= request**. Hier wird das **Tuple-Unpacking** angewandt. Dabei wandert die Button-Information nach button und die x und y-Werte landen in der temporären Variablen _. Der "*" sorgt dafür, dass alles, was auf das erste Tuple-Element folgt in _ landet. _ kann als eine unbedeutende, temporäre Variable aufgefasst werden. Liegt eine Tasteninformation vor, wird diese zurückgegeben, sonst None.

```
>>> t=(1,234,-345)
>>> button, *_ = t
>>> _
[234, -345]
>>> button
1
```

Bei der direkten Eingabe hat _ eine zusätzliche Bedeutung. In _ wird die letzte Ausgabe im Terminal gespeichert. Deshalb muss _ in dem Beispiel vor **button** abgefragt werden.

```
def waitForButtonIs0():
    sw=getButton()
    while sw is not 0:
        sw=getButton()
    sw=getButton()
    while sw is 0:
        sw=getButton()
```

An verschiedenen Stellen müssen wir am Spielbrett auf die Betätigung einer Taste warten. Der lange Funk-Zeigefinger zum Spielbrett besteht aus einem Tastendruck an der Fernsteuerung und der Weitergabe des entsprechenden Codes, 0 oder 1. Der ESP8266 muss also zuerst auf das Eintreffen einer 0 warten und darf erst weitermachen, wenn er danach die erste 1 empfangen hat.

Der danach folgende Abschnitt

```
# ***************** Connect to WLAN **********************
```

ist im Programm ausreichend kommentiert, weshalb ich hier nicht näher darauf eingehe. Wichtig ist nur, dass Sie für mySSID und myPass Ihre Credentials angeben.

```
mySSID = 'Here_goes_your_SSID'
myPass = 'Here_goes_your_Password'
```

Die zweite kritische Stelle ist die Angabe der Netzwerkdaten, IP, Netzwerk-Maske, Gateway und DNS-Server, die Sie an Ihr Netzwerk anpassen müssen.

```
# Aufbau der Verbindung zum WLAN-Accesspoint
# Wir setzen eine statische IP-Adresse
nic.ifconfig(("10.2.1.95","255.255.255.0","10.2.1.20","10.2.1.
100"))
```

Diese Angaben sollten von Hand vorgegeben werden, weil es, ebenso wie bei der Fernsteuerung, um Netzwerkteilnehmer geht, die sich stets ohne Probleme wieder zusammenfinden müssen. Da sind dynamische Adressen vom DHCP-Server des WLAN-Accesspoints sehr kontraproduktiv.

Eines müssen Sie im Zusammenhang mit dem Accesspoint natürlich auch noch machen, die MAC-Adressen von Spiel und Fernsteuerung am Router als zulässig eintragen. Ziehen Sie bitte dafür Ihr Router-Handbuch zu Rate.

Die Netzwerkverbindung stellt für die Funkverbindung das zur Verfügung, was Sie beim Einrichten einer seriellen Verbindung über USB zum ESP8266 durch Einstecken des Kabels erledigen.

Wie bei der seriellen Übertragung müssen wir jetzt aber noch ein Interface öffnen. Bei der RS232 würden wir ein UART-Objekt (Universal Asynchronous Receiver / Transmitter) instanziieren. Auf Funkebene erzeugen wir ein Socket-Objekt **s**.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', 9009))
print("waiting on port 9009...")
s.settimeout(0.1)
```

Mit **socket.SOCK_DGRAM** vereinbaren wir UDP als Protokoll, das Pakete (Datagramme) transportiert, im Gegensatz zu TCP, das auf Streaming (socket.SOCK_STREAM) setzt. Die nächste Zeile erlaubt uns im Fall eines Neustarts, dass wir dieselbe Socket-Adresse wiederverwenden können. In der dritten Zeile binden wir das Interface an die im WLAN-Teil angegebene IP-Adresse und den Port 9009. Durch das Setzen eines Timeouts von 0,1 Sekunden schaffen wir es, dass die Hauptschleife und die anderen Stellen im Programm, an denen es (hauptsächlich) um den Empfang von Nachrichten geht, den Programmfluss nicht

blockieren. Das ist überall dort der Fall, wo **getData**() direkt oder implizit (zum Beispiel **waitForButtonIs0**()) aufgerufen wird.

Nach all den Vorbereitungen im eigenen Stall, nimmt das Programm im nächsten Abschnitt die Arbeit mit der Gegenstelle, der Fernsteuerung, auf.

```
uServo=adc.read()/158.569
text="W:".encode()
text=text+struct.pack("f",uServo)
s.sendto(text,address)
sleep(5)

goToOrigin()
```

Wir messen die Betriebsspannung auf der 6V-Leitung und schicken das Ergebnis an die Fernsteuerung. Auch die Meldungen, die **goToOrigin**() in Auftrag gibt, werden gesendet. Dazwischen wird die Kugel zurück zum Start geschleust und die globalen Variablen erhalten ihre Ausgangswerte.

Die Hauptschleife kümmert sich um drei Jobs. Da ist zunächst das Abholen von Neigungsdaten. War ein Eingang zu verzeichnen, dann wird das von **getData**() zurückgegebene Tuple auf x- und y-Wert geparst, und die Werte gehen an den PCA9685 weiter. In diesem Fall wird die Tasteninformation in _ entsorgt.

```
while 1:
    response=getData()
    if response is not None:
        _,x,y=response
        servo.writePulseTimeSlice(4,0,x)
        servo.writePulseTimeSlice(7,0,y)
#     d.writeAt("score:{}       ".format(points),0,2)
    if target.value()==0:
        gameover=True
        dauer=int((ticks_ms() -start)/1000)
        timePoints=(15-dauer)*zeitfaktor
        score=points+timePoints
        points=max(0,points)
        print(points, timePoints, score)
        text="O:".encode()
        text=text+struct.pack("H",score)
        s.sendto(text,address)

        waitForButtonIs0()
        goToOrigin()

    if getButton()==0:
        text="C:".encode()
        text=text+struct.pack("H",points)
        s.sendto(text,address)
        sys.exit()
        #reset()
```

Wenn der Eingang **target** auf 0 gegangen ist, befindet sich die Kugel im Ziel. Die Spieldauer wird berechnet und damit auch die Punkte aus der Spielzeit. Wir setzen das zum Score zusammen und senden die encodierte Byte-Folge an die Fernsteuerung zur Anzeige. Dann warten wir auf die Tastenbetätigung an der Fernsteuerung und setzen das Spiel zurück für die nächste Runde.

Wird die Taste an der Fernsteuerung während einer Spielrunde gedrückt, dann wird das als Abbruchsignal gewertet und das Programm auf die Console verlassen. Diese Option ist eigentlich nur während der Entwicklungsphase von Bedeutung und kann entfernt werden, wenn alles funktioniert. Alternativ kann man den Befehl **reset**() entkommentieren und **sys.exit**() in einem Kommentar verstecken. Dann macht der ESP8266 einen kompletten Kaltstart.

Das gesamte Programm joyball_funk.py steht zum Download bereit.

Die Fernsteuerung

Die Teile für die Fernsteuerung haben bis auf die Spannungsversorgung Platz auf einem Mini Breadboard (Abbildung 2).

Das Programm joyball_sensor.py ist von der Netznutzung her ähnlich aufgebaut wie joyball_funk.py. Bei den Importen sind also die Module **network**, **socket** und **struct** wieder mit von der Partie. Für die Abfrage des MPU6050 ist die Klasse **GY521** im Modul **gy521_level**.py zuständig. Wir erzeugen eine I2C-Instanz, das Accelerator-Objekt **ac** und ein OLED-Objekt **d**.

```
# joyball_sensor.py
# Accelerator-Einheit fuer den joyball-wizzard
#
# import webrepl_setup
# > d fuer disable
# Dann RST; Neustart!
import sys,os
from machine import Pin,reset,I2C
from gy521_level import GY521
from oled import OLED
import struct
import esp
esp.osdebug(None)
import gc
gc.collect()
from time import sleep, ticks_ms
import network
import socket
#
SCL=Pin(5)
SDA=Pin(4)
i2c=I2C(-1,SCL,SDA,freq=400000)

ac=GY521(i2c)
d=OLED(i2c,128,32)
```

taste und **button** sind die beiden Tastenobjekte für die Notbremse und den Game-Button. Die verschiedenen LED-Objekte dienen der Darstellung der Verbindungszustände zum WLAN-Router, blau für die Funkverbindung, grün für den Heartbeat, das ist Funktion der Hauptschleife und rot für Fehlerkonditionen.

```
taste=Pin(0,Pin.IN,Pin.PULL_UP) # D3
button=Pin(13,Pin.IN,Pin.PULL_UP) # D7
bluePin,greenPin,redPin=16,14,12
onAir=Pin(bluePin,Pin.OUT,value=0) #D16
heartbeat=Pin(greenPin,Pin.OUT,value=0) #14
error=Pin(redPin,Pin.OUT,value=0) # 12
blue,green,red=0,1,2
led=[onAir,heartbeat,error]

target=("10.0.1.95",9009) # Socket der Spieleinheit
myAddr=("10.0.1.94",9009) # Socket der Fernsteuerung
```

Der MPU6050 sollte vor dem ersten Spielbeginn einmal kalibriert werden, damit der ESP8266 weiß, welche Werte auftreten können und wo der Mittelwert liegt. Wurde die Kalibrierung durchgeführt, dann schreibt die Methode **calibrate**() die Daten in die Datei **accel.ini** im root-Filesystem des ESP8266. Existiert diese Datei, ist der Leseversuch erfolgreich. Andernfalls wird die Kalibrierung aufgerufen.

Dazu sind fünf Schritte notwendig. Was jeweils zu tun ist sagt die Terminalausgabe von Thonny. Die x- und y-Richtungspfeile findet man auf dem GY-521-Platinchen.

```
try:
    ac.readCalibration()
except:
    ac.calibrate(5)
```

Die Funktion parse() kümmert sich um die Entzifferung der Nachrichten vom Spielprogramm. Das Bytes-Objekt wird nach einem ":" durchsucht, der den Kommando-Teil von den Daten trennt. Alles bis zur Stelle **pos** (ausschließlich) kommt nach **cmd**, alles danach nach **val**. In den nachfolgenden if – elif – Strukturen passieren die Ausgaben am OLED-Display. Das folgende Listing zeigt als Ausschnitt den Anfang der Funktion.

```
def parse(code):
    pos=code.find(b':')
    cmd=code[:pos].decode()
    val=code[pos+1:]
    print("cmd,val",cmd,val)
    if cmd == "R":
        d.clearAll()
        d.writeAt("GOING BACK",2,0,False)
        d.writeAt("TO ORIGIN",3,1,False)
        d.writeAt("PLEASE WAIT!",1,2)
    elif cmd == "G":
```

```
        d.clearAll()
        d.writeAt("CLICK BUTTON",1,0,False)
        d.writeAt("to START game",1,1)
    elif cmd == "S":
        points=struct.unpack("H",val)
        d.clearAll()
        d.writeAt("GAME",3,0,False)
        d.writeAt("STARTED",4,1,False)
        d.writeAt("SCORE:{}       ".format(points[0]),0,2)
```

Der anschließende Abschnitt zur Verbindungsaufnahme mit dem WALN-Accesspoint ist bis auf die Zeile

```
nic.ifconfig(("10.2.1.94","255.255.255.0","10.2.1.20","10.2.1.
100"))
```

identisch mit Programmteil in joyball_funk.py. Auch der Aufbau des Socket-Objekts läuft analog. Dabei benutzen wir zweckentfremdet die Funktion parse zur Ausgabe einer Startmeldung. Zwei Sekunden zum Lesen und dann ab in die Jobschleife.

```
# Serverschleife
while 1:
    switch=button.value()
    try: # send command
        x,y=ac.getXY()
        #print(x,y)
        x=max(min(x,ac.maxX),ac.minX)
        y=max(min(y,ac.maxY),ac.minY)
        x=ac.transform(-x,-ac.maxX,-ac.minX,350,2300)
        y=ac.transform(y,ac.minY,ac.maxY,350,2300)
        command=struct.pack("BHH",switch,x,y)
        s.sendto(command,target)
    except:
        pass
    try:
        # receive response
        rec,adr=s.recvfrom(10)
        print("roh:",rec)
        # parse it
        parse(rec)
        rec=""
    except:
        pass # timeout uebergehen
    if taste.value()==0:
        sys.exit()
    blink(0.005,0.03,col,cnt=1)
```

Vier Dinge sind zu erledigen: Abfrage des Spielbuttons, einholen der Neigungsdaten vom GY-521, nachschauen, ob uns der ESP8266 der Spieleinheit eine Meldung

geschickt hat und schließlich prüfen der Abbruchtaste. Abschließend ein kurzes Blinken für den Heartbeat, grün für normale Funktion, rot bei einem Kommunikationsfehler.

Nach dem Einlesen der Neigungswerte werden sie erst einmal in das Fenster eingepasst, das durch die Kalibrierungs-Eckwerte vorgegeben ist. Für das Rollen (seitliche Neigung) müssen wir die Werte spiegeln, damit das Spielbrett den Bewegungen des Fernsteuerungs-Sensors korrekt folgt. danach wandeln wir die Zahlen in ein Bytes-Objekt um und senden es an das Spiel. Tritt bei all dem ein Fehler auf, dann kümmert uns das nicht, **pass**, es folgt ja in Kürze ein weiteres Paket.

Der Empfang von Nachrichten läuft zunächst analog zum Programm **joyball_funk.py**. Da die Aufschlüsselung recht umfangreich ist, wird diese Aufgabe der Funktion **parse**() übergeben, der wir das empfangene Bytes-Objekt weiterreichen. Falls keine Nachricht eingetroffen ist, wird der Timeout mit **try** abgefangen. except hat nicht zu tun, also **pass**.

Ein extrem kurzes Blinken schließt die Schleifenjobs ab.

Damit sind wir vorerst am Ende der Joyball-Sequenz angekommen. Das Programm für die Fernsteuerung gibt es natürlich auch zum Download. In einer der nächsten Folgen zur MicroPython-Blog-Reihe werde ich eine große Anzeige aus 8x8-LED-Martix-Feldern vorstellen, die über Funk oder I2C oder RS232 angesteuert werden kann. Damit lassen sich dann nicht nur Spielergebnisse im Großformat darstellen, sondern auch Messwerte von Sensoren, etc.

Bis dann.