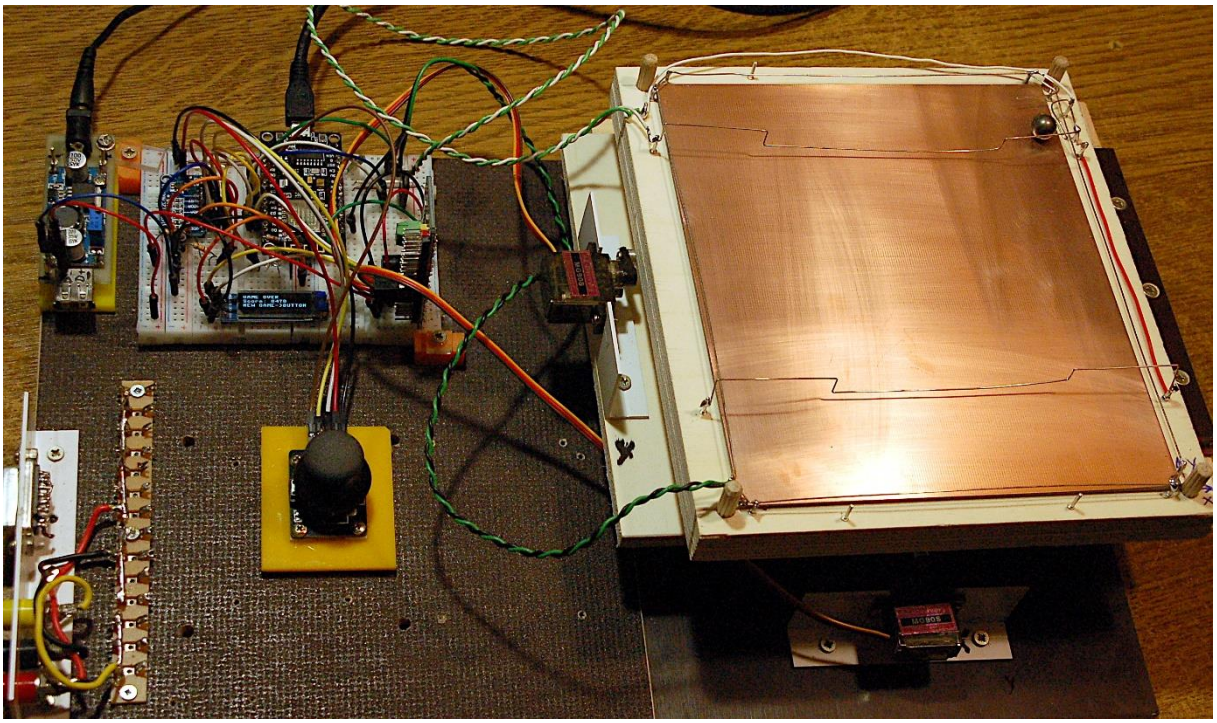
Gesamtaufbau

Dieser Blogpost ist auch als PDF-Dokument verfügbar.

Die Kugel rollt, wenn Sie den Joystick schwingen. Wutschen und Wedeln ist angesagt beim Joy Ball Wizzard. Ein Spiel, bei dem es auf Geschicklichkeit, Zeit und Reaktion ankommt. Die Stahlkugel rollt auf einer Leiterplatte und muss vom Start durch diverse Hindernisse, möglichst ohne Berührung mit diesen oder mit der Umrandung, ins Ziel bugsiert werden. Wie so etwas gebaut werden kann, das verrate ich dieser neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der Joy Ball Wizzard

Joy, ist klar, denn der Steuerknüppel ist ein Mini-Joystick, Ball oder Kugel, dafür nehme ich eine Kugellagerkugel mit 10mm Durchmesser und Wizzard – na ja, ein wenig zaubern muss man schon, bis die Mechanik aufgebaut und die Schaltung hergestellt ist. Beim Spiel ist es aber auch nicht ganz so einfach, die Kugel nur durch Neigen der Sperrholzplatten zu lenken. Die federnden Hindernisse lenken die Kugel schon gerne mal wo anders hin als man das haben wollte. In dieser Episode geht es um die Mechanik und um erste Tests der Hardware. In der nächsten Folge bauen wir dann das Programm zum Spiel.

Die Elektro-Mechanik

Schauen wir uns als Erstes die Mechanik an. Für den Aufbau brauchen wir:

Material

4	Pappelsperrholz von 8mm; 180 x210mm
2	Holzleiste 15x19mm; 180mm als Auflager
4	Sperrholzleisten 8mm; 15mmx180mm einseitig abgeschrägt
2	Lederstreifen ca. 1mm stark; 30x180mm als Scharnier
4	Holzdübel 6mm Ø
1	**ebene** Leiterplatte 150x180mm
2	Kunststoff- oder Aluwinkel 15x25mm; ca 100mm lang
4	Spax-Schrauben 3x10mm
12	Spax-Schrauben 3x20mm
1	Stahlkugel 8..10mm
diverse	Lötnägel 8mm und 10mm überstehend
ca.2m	versilberter Kupferdraht
etwas	Kontaktkleber und Holzleim
2	2-adrige Kabel ca. 30cm
2	2-polige Stiftleisten
2	MG90S Micro Servomotor kompatibel mit Arduino
2	Schrauben oder Stifte optional: mit Kugellager

Werkzeug:
Schraubendreher, Stechahle, Laubsäge, Bohrer 2mm Ø + 6mm Ø, kleine Flachzange, Lötgerät,

Der Aufbau erfolgt von unten nach oben.

1. An die erste Sperrholzplatte an der hinteren Schmalseite eine der Holzleisten hochkant anschrauben 3x 3x20.

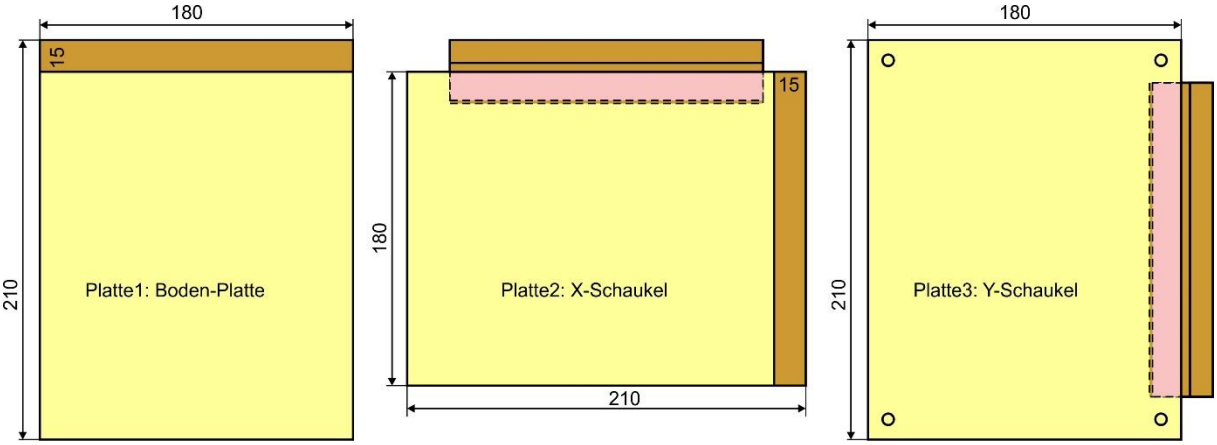

Abbildung 1: Platten und Scharniere

2. Auf die zweite Sperrholzplatte in der Mitte der hinteren Längsseite einen der Lederstreifen von unten aufkleben, so dass er 15mm übersteht. Das Leder mit einer abgeschrägten Sperrholzleiste bündig mit der Kante der Sperrholzplatte mit 3 Schrauben 3x20 fixieren.

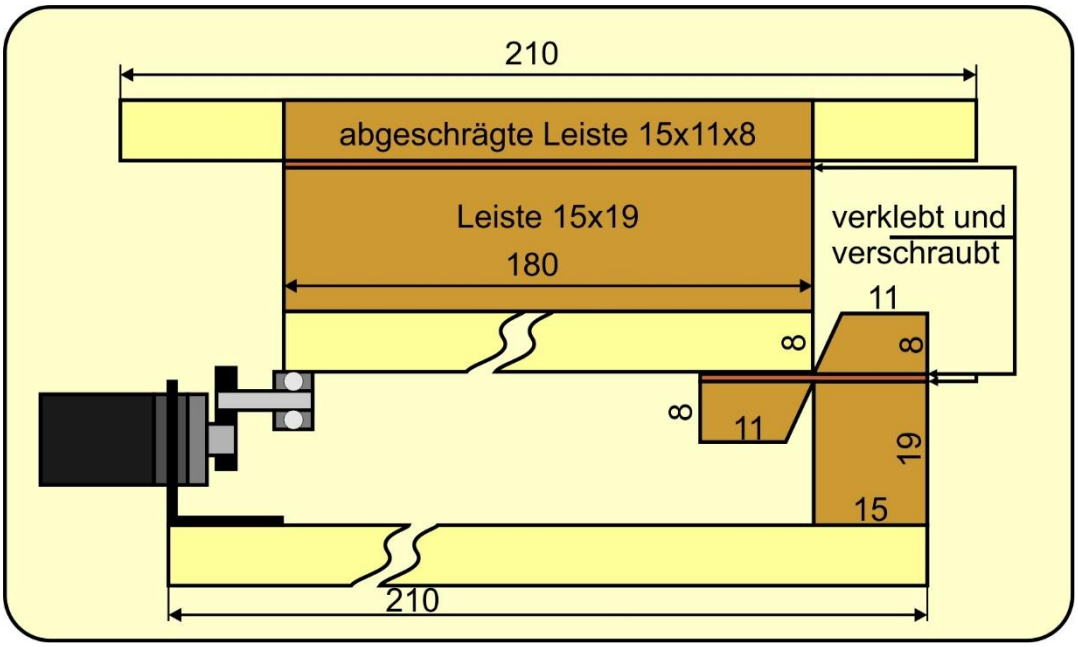

Abbildung 2: Schaukel, erste und zweite Etage (Seitenansicht von links)

3. Auf die Oberseite von Platte 2 an der rechten Schmalseite die zweite Holzleiste 15x19mm oben hochkant aufschrauben.

4. Die zweite Platte möglichst waagrecht lagern und das Leder bündig auf die Holzleiste von Platte 1 kleben wie in Abbildung 3 dargestellt. Mit abgeschrägter Holzleiste und drei Schrauben 3x20mm fixieren.


Abbildung 3: Leder-Scharnier

5. Aus beiden KS-Winkeln mittig einen Ausschnitt herstellen, sodass der Servomotor seitlich knapp Platz hat. Der Ausschnitt soll gut tiefer sein als die Dicke des Motors beträgt. Winkel bündig mit der Vorderkante der Platte 1 montieren (Abbildung 2).

6. Am Hebel den Stift/(Kugellager) montieren und auf dem Motor befestigen. Den Motor in der Höhe so ausrichten, dass Stift/Lager beim Absenken nicht den Boden berühren. Hebel gegebenenfalls kürzen. Außerdem …

Abbildung 4: Servo mit Kugellager aus einer alten Festplatte am Hebel

7. Motor möglichst so ausrichten, dass der Stift bei mittiger Position des Hebels die Platte waagrecht hält. Dann die Position der Befestigungslöcher auf der Leiste markieren. Motor entfernen, Winkel vorbohren (2mm). Motor einbauen, verschrauben.

8. Platte 3 und 4 deckungsgleich fixieren. 6mm -Löcher an den Ecken gemäß Abbildung 5 bohren, Platte 3 nicht ganz durchbohren. Die Lage der Platten an den Sägekanten markieren, damit man sie später wieder genauso zusammenfügen kann.

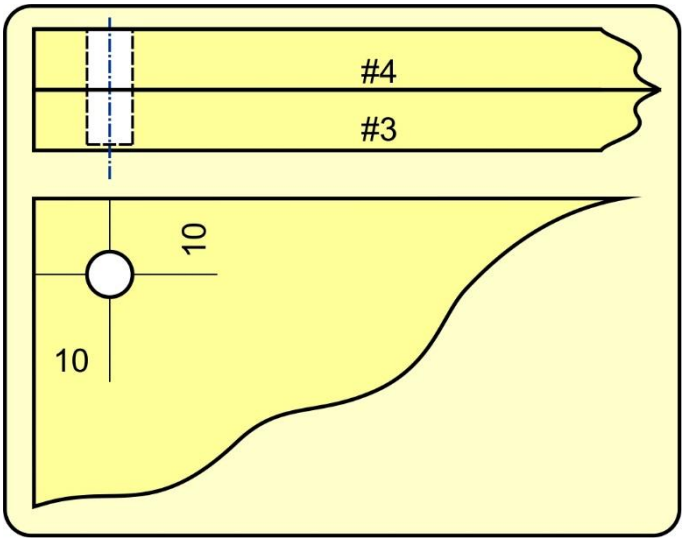

Abbildung 5: Bohrungen Platte 3 und 4

9. In die Platte 3 die vier Holzdübel einkleben. Die Leiterplatte als Abstandshalter auflegen und Platte 4 so aufsetzen, dass die Markierungen übereinstimmen. Kleber aushärten lassen.

10. Den zweiten Lederstreifen an der Platte 3 ebenso anbringen wie in 2. beschrieben.

11. Die Platte 3 wie in 4. beschrieben auf der Platte 3 befestigen.

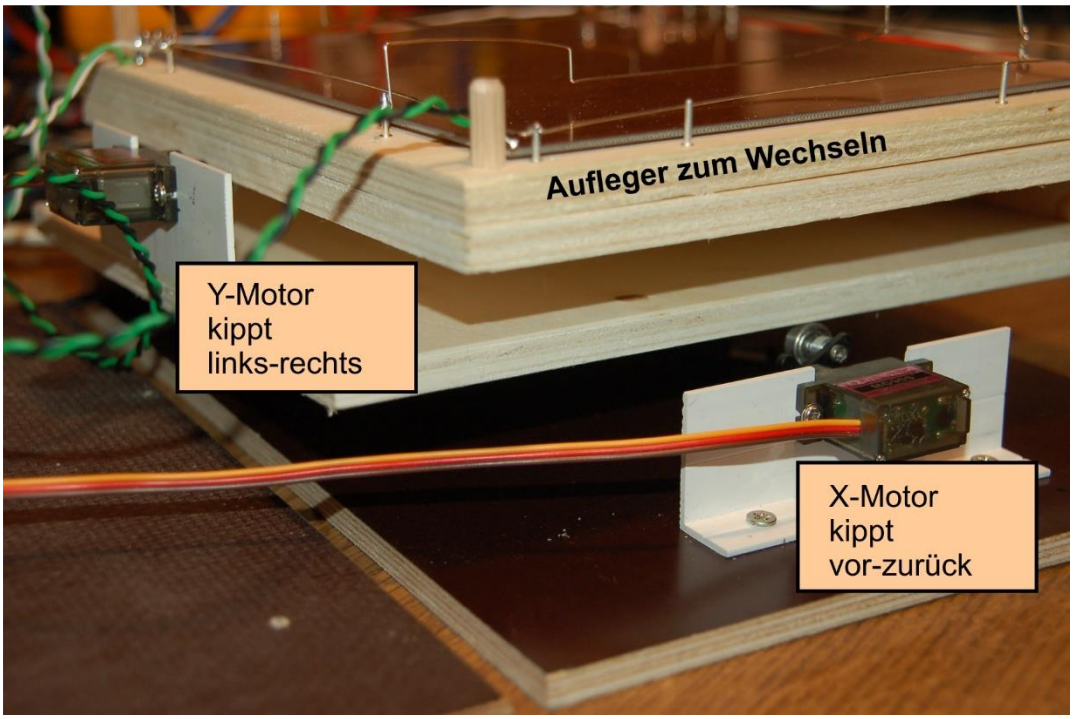

Abbildung 6: Plattenaufbau mit Servos

12. Winkel mit Servomotor auf Platte 2 befestigen wie in 5., 6. und 7. beschrieben.

13. Auf den Aufleger mit doppelseitigem Klebeband die Leiterplatte aufkleben und mit einem seitlichen Abstand von 1 bis 2mm Nägel/Lötnägel mit einem Überstand von ca. 8mm einschlagen und den blanken Kupferdraht als Begrenzung dranlöten.

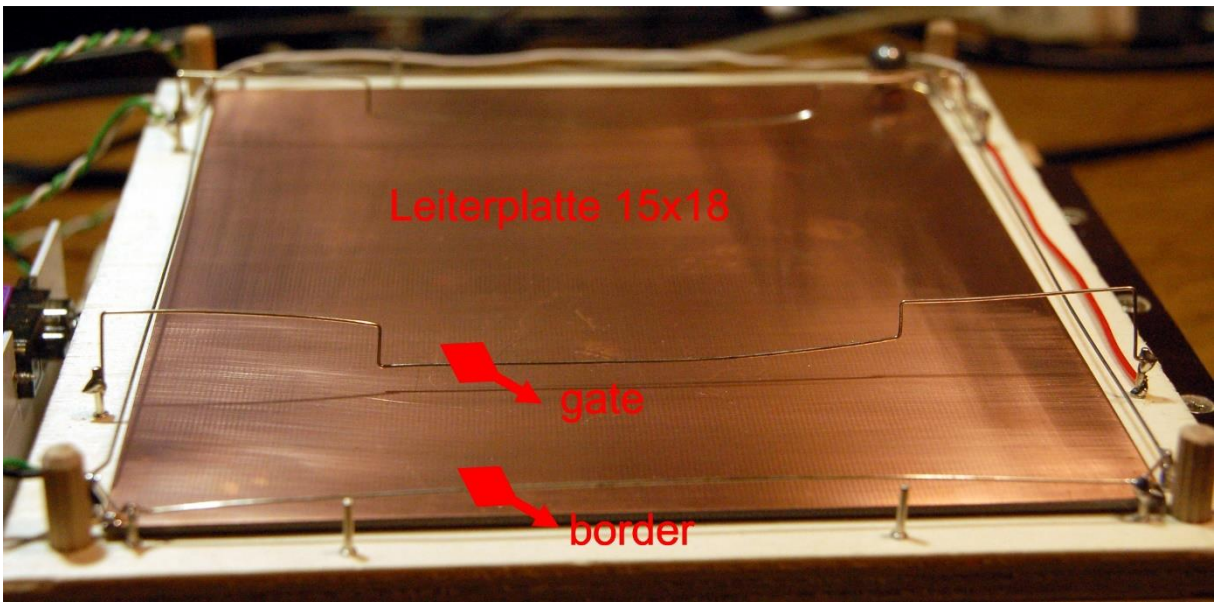

Abbildung 7: Spielfeld mit Gate und Border

14. An weiteren Nägeln die Gates und das Target befestigen.

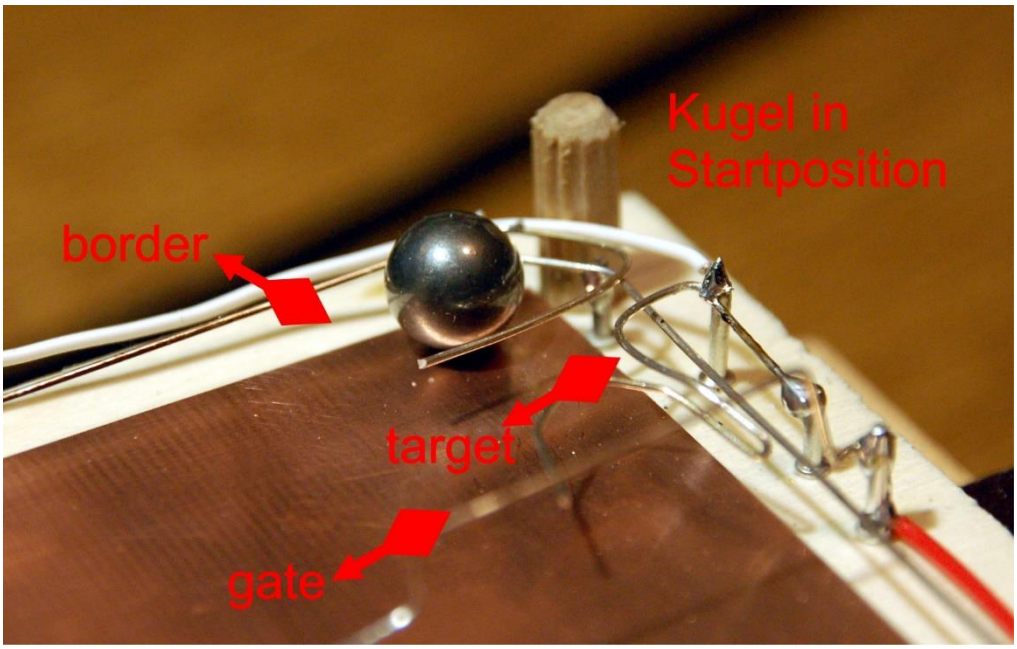

Abbildung 8: Gate- Target und Border-Anschlüsse

Die Kabel mit der Leiterplatte (GND) und den Drahtelementen verlöten. Ans andere Kabelende je eine der Steckleisten löten.

Hardware

1	NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI WLAN unverlötet mit CP2102 oder NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	KY-023 Joystick-Modul oder PS2 Joystick Shield Game Pad Keypad V2.0
1	ADS1115 ADC Modul 16bit 4 Kanäle für Raspberry Pi
1	Servotreiber-Modul PCA9685
1	LM2596S Step-down DC-DC Buck Converter mit 3-stelliger Digitalanzeige oder LM2596S DC-DC Netzteil Adapter Step down Modul
2	Widerstand 10kΩ
diverse	Jumperkabel
2	Minibreadboard oder Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins

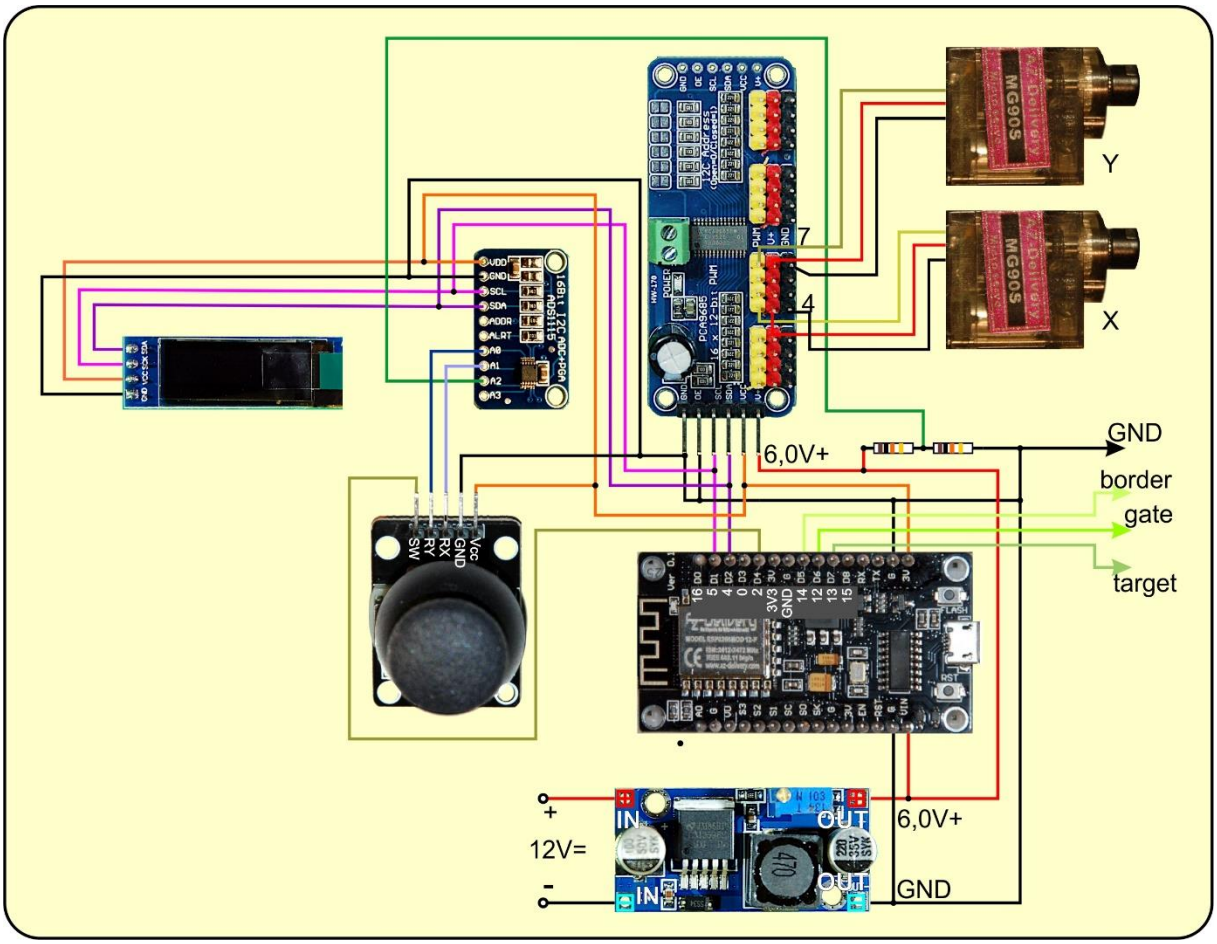

Abbildung 9: Joy Ball Wizzard-Schaltung

Eine größere Darstellung gibt es hier als PDF-Dokument.

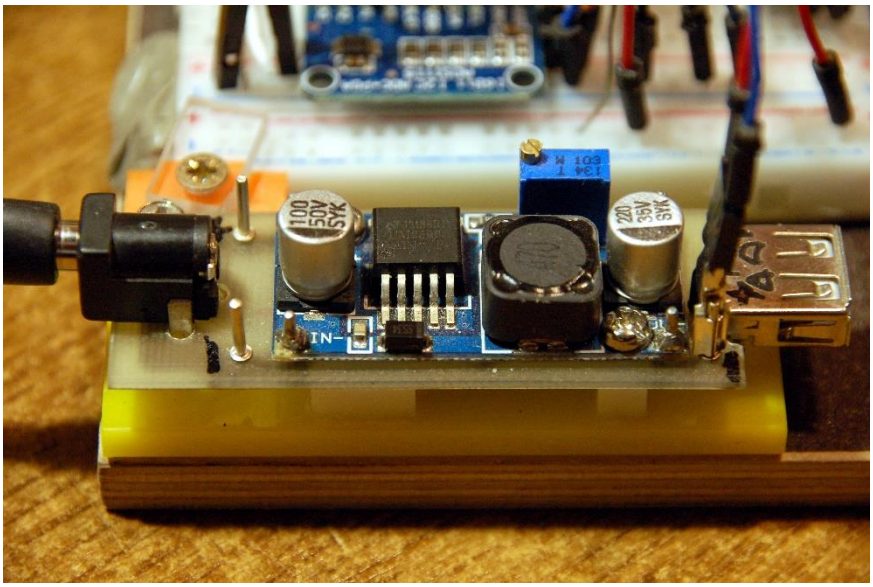

Abbildung 10: Spannungsversorgung mit Buck-Konvertrer

Zur Versorgung der Servos ist eine Spannung von maximal 6V zulässig. Die wird von dem Buck-Konverter geliefert, der seinerseits durch ein 12V-Steckernetzteil versorgt wird. Der ESP8266 bekommt ebenfalls die 6V an seinem Anschluss VIN. Das geht, weil der ESP8266 Node-MCU V3 ebenso wie der Amica einen Eingangs-

Spannungsregler AMS1117 3V3 an Bord haben, der Spannungen bis maximal 20V verträgt. Dieser Spannungsregler versorgt denn auch den Rest der Schaltung am 3,3V-Ausgang des ESP8266 mit dieser Spannung.

Der PCA9685 kann bis zu 16 Servos oder LEDs mit PWM-Signalen versorgen. Die Frequenz für alle Anschlüsse ist gleich und kann zwischen 40Hz und 1kHz eingestellt werden. Die Kommunikation zwischen PCA9685 und dem ESP8266 erfolgt über den I2C-Bus. Das Modul pca9685.py enthält die Klasse SERVO. Es stellt alle Methoden zur Verfügung, die Zur Steuerung der PWM-Ausgänge nötig sind.

Die Servos und eventuelle LEDs werden direkt an die 16 Ausgänge gesteckt. Damit Rückwirkungen auf den ESP8266 weitestgehend unterdrückt werden, sollte das PCA9685-Modul eine eigene 6V- und GND Zuleitung vom Spannungswandler bekommen. Keine schlechte Idee ist auch ein Elko von 470µF/ 16V an der Steckleiste oder an der grünen Lüsterklemme auf dem Board.

Auch die OLED-Anzeige liegt am I2C-Bus. Sie informiert über anstehende User-Aktionen und den Spielstand.

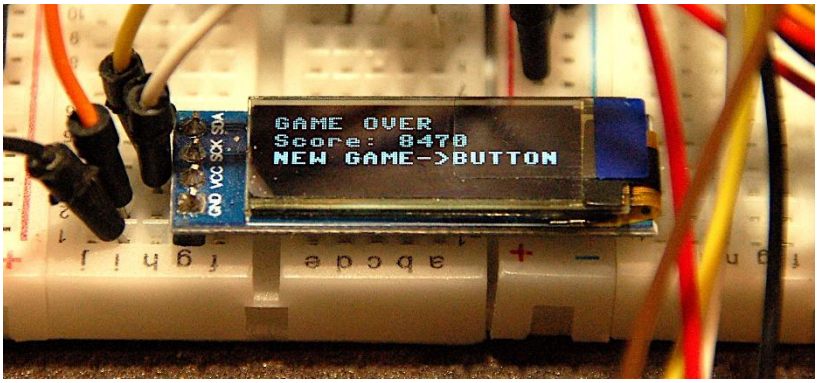

Abbildung 11: Spielstandanzeige

Die, für unseren Zweck verwendbaren, digitalen Eingänge des ESP8266 sind mit der Taste des Joysticks (GPIO0=D3) und mit den Sensoreingängen vom Spielbrett verbunden, Umgrenzung (border GPIO12 =D6), Hindernisse (gate GPIO14=D5) und Zielklammer (target GPIO13=D7). D4=GPIO2 ist für das Freigeben der Ausgänge des pca9685 reserviert. Der Ausgang bedient gleichzeitig die LED auf dem ESP8266-Board. Die leuchtet also, wenn die Servos freigegeben sind, weil es eine LOW-aktive LED ist. Es kann optional an D4 auch noch eine externe LED mit passendem Vorwiderstand (ca. 1kΩ) gegen 3,3V angeschlossen werden (im Schaltplan nicht abgebildet). D0 sowie D8 scheiden wegen systembedingter Besonderheiten aus.

Um die Spannungspegel vom Joystick zum ESP8266 zu bringen verwende ich ein AD-Wandlermodul vom Typ ADS1115. Es liefert bis zu vier 16-Bit-ADC-Werte von guter Genauigkeit und Linearität. Angesprochen wird es ebenfalls über I2C. Zwei Eingänge, A0 und A1, werden für den Joystick benötigt, A2 überwacht den Pegel der 6V-Spannung.

Als Joystick kommen zwei Varianten in Frage, das einfache Modul KY-023 oder das komfortablere Arduino-Shield mit 6 Buttons.

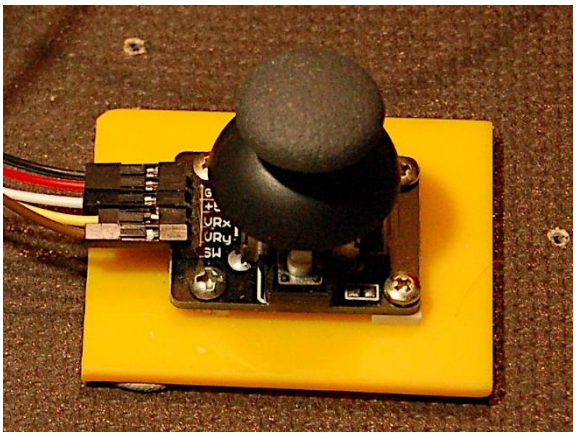

Abbildung 12: Joystick auf Bodenplatte montiert

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
ads1115.py: Modul ads1115; ADS1115-Treiber
ssd1306.py: OledTreiber
oled.py: OLED-Klasse
servo_Test.py: Servo Testprogramm
test_Joy.py: Joystick Testprogramm
test_Oled.py: OLED-Testprogramm

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.

Die Testprogramme

Als Erstes sollten wir, nach dem Flashen der Firmware, dem ESP8266 abgewöhnen, nach einem Accesspoint zu suchen, weil das nach meinen Erfahrungen verschiedentlich zu merkwürdig eigensinnigem Verhalten des Moduls führt.

Auf der Kommandozeile von Thonny geben wir folgenden Befehl und dann "d" für disable ein. Diese Aktion sollte jedes Mal (nur einmal) nach dem Flashen der Firmware erfolgen.

```
>>> import webrepl_setup
  WebREPL daemon auto-start status: disabled

  Would you like to (E)nable or (D)isable it running on boot?
  (Empty line to quit)
  > d
  No further action required
```

Für die Tests der Mechanik und Hardware müssen zunächst die Treiber für das PWM- und das ADC-Modul, der OLED-Treiber ssd1306 und das oled-Modul in den Flash des ESP8266 hochgeladen werden.

pca9685.py: Modul mit der Klasse SERVO, PCA9685-Treiber
ads1115.py: Modul ads1115; ADS1115-Treiber
ssd1306.py: OledTreiber
oled.py: OLED-Klasse

Laden Sie bitte die Dateien in Ihr Arbeitsverzeichnis (aka workspace) herunter. Navigieren Sie in Thonny dorthin. Ein Rechtsklick auf die Datei öffnet das Kontextmenü. Mit **Upload to /** starten Sie den Vorgang.

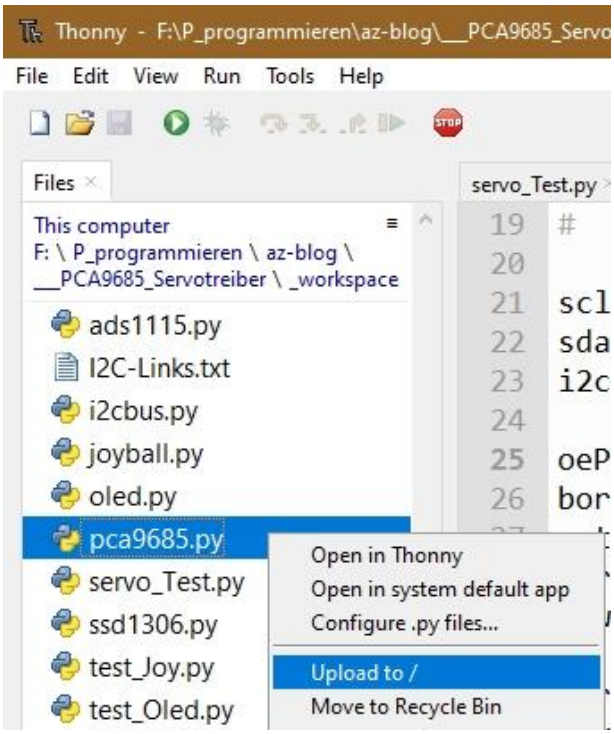

Abbildung 13: Upload starten

Wiederholen Sie den Vorgang für die drei weiteren Dateien.

OLED-Test

Beginnen wir mit dem Test des OLED-Displays. Das passiert mit Hilfe der Datei test_Oled.py. Wir starten mit einigen Importen. Das Modul oled.py importiert seinerseits ssd1306.py. Weil dieses für keine weiteren Aktionen direkt im Hauptprogramm gebraucht wird, ist das in Ordnung so.

Den I2C-Bus initialisieren wir bereits im Hauptprogramm, weil wir ihn für andere Zwecke auch noch brauchen. Der Pintranslator setzt als Kommentar die Arduino-Benennungen zu denen von Espressif in Beziehung. Letztere werden von MicroPython benutzt.

```
#pca_first.py
# import webrepl_setup
# > d fuer disable
# Dann RST; Neustart!
import os,sys       # System- und Dateianweisungen
from machine import Pin, reset, I2C
from time import sleep, ticks_ms
from oled import OLED
import esp
esp.osdebug(None)
import gc
gc.collect()

# Pintranslator fuer 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

sclPin=Pin(5)
sdaPin=Pin(4)
i2c=I2C(scl=sclPin,sda=sdaPin,freq=400000)

d=OLED(i2c,128,32)
d.writeAt("Hello world!",0,0)
sleep(2)
d.writeAt("Ola muchachos!",0,1)
d.hline(0,24,127,1)
d.show()
sleep(2)
d.clearAll()

d.writeAt("ABER...",0,1)
sleep(2)
d.hline(0,24,127,1)
d.writeAt("Hello world!",0,0,show=False)
sleep(2)
d.writeAt("Ola muchachos!",0,1,show=True)
sleep(2)
d.clearAll()
```

Die für SCL und SDA deklarierten Pins übergeben wir dem Konstruktor der I2C-Klasse zusammen mit dem Wert der Betriebsfrequenz von 400kHz. Über das Protokoll auf dem I2C-Bus gibt es in der nächsten Folge noch genaue Informationen.

Mit dem I2C-Objekt instanziieren wir das Display-Objekt **d**. Die Klasse OLED erbt den Namensraum der Klasse **SSD1306_I2C**, die ihrerseits vom Modul **framebuf** die

Klasse **FrameBuffer** geerbt hat. Unter dem Strich bedeutet das, dass alle Methoden der drei Klassen gleichberechtigt unter dem Scope von **d** verfügbar sind. Allerdings gibt es einen kleinen Unterschied zwischen den Methoden der Klasse OLED und den Methoden der anderen beiden Klassen **SSD1306_I2C** und **FrameBuffer**.

Die schreibenden Methoden von **OLED** besitzen einen optionalen Parameter **show**, der per Default auf True gesetzt ist. Das bedeutet, dass jeder schreibende Zugriff auf das Display sofort sichtbar wird. Das kostet Zeit, denn damit wird der gesamte Framebufferinhalt zum Display geschaufelt. Und deshalb ist es in zeitkritischen Situationen besser mit **show=False** zuerst den Buffer mit allen Schreibaktionen zu füllen und erst zum Schluss den Transfer des Buffers zum Display durchzuführen.

Beim ersten Teil der Ausgabe erscheinen die Texte und die Linie sofort nach dem Befehl. Die Linie, als Methodenaufruf aus FrameBuffer, erscheint erst nach dem Aufruf von **show**().

Beim zweiten Teil wird durch die drei Befehle zuerst nur der Framebuffer gefüllt. Erst durch das **show=True** erfolgt der Übertrag zum Display und die Anzeige.

Testen Sie das jetzt einfach einmal selbst. Laden Sie das Programm test_Oled.py in Thonny durch Doppelklick aus dem Workspace in ein Editorfenster und starten Sie es mit der Funktionstaste F5.

Am Ende habe ich noch zwei Sequenzen eingefügt, die einen markanten Unterschied in der Laufzeit aufzeigen. Im ersten Teil kommt nach jedem Ausgabebefehl ein show()-Aufruf, beim zweiten Teil erst zum Schluss.

```
start=ticks_ms()
d.writeAt("Immediate show...",0,1)
d.hline(0,24,127,1)
d.show()
d.writeAt("Hello world!",0,0)
d.writeAt("Ola muchachos!",0,1)
d.clearAll()
runTime=ticks_ms()-start
print("Runtime immediate:",runTime,"ms")


start=ticks_ms()
d.writeAt("ABER...",0,1,False)
d.hline(0,24,127,1)
d.writeAt("Hello world!",0,0,show=False)
d.writeAt("Ola muchachos!",0,1,show=True)
d.clearAll()
runTime=ticks_ms()-start
print("Runtime once:",runTime,"ms")
```

Die Ausgabe im Terminal bedarf keines Kommentars.

this is the constructor of OLED class
Size:128x32
Runtime immediate: 193 ms

Runtime once: 79 ms

Joystick-Test

Im Joystick sind zwei einstellbare Widerstände verbaut, deren Enden an +Vcc=3,3V und an GND liegen. Der Schleifkontakt greift daher als Spannungsteiler Spannungen zwischen 0V und 3,3V ab. Diese Spannungen führen wir dem Analog-Digital-Wandler (aka ADC) auf Kanal 0 und 1 zu.

Der ADC liefert uns aber keinen Spannungswert, sondern einen Zahlenwert zwischen 0 und 32767 (wir nutzen nur den positiven Bereich). Der ergibt sich durch einen Zähler, der mitläuft, während im ADC-Modul eine Vergleichsspannung sukzessive erhöht wird. Der Zähler stoppt, wenn Vergleichsspannung und externe Spannung gleich sind.

Nun ergeben sich durch Rauschen auf der Signalleitung und Toleranzen bei der Wandlung stets kleine Abweichungen von einem Messwert zum nächsten. Das Flattern der Messwerte stört uns, weil es zu Zitterbewegungen des Servos führt. Deshalb nutzen wir von den effektiven 15 Bits des positiven Spannungsbereichs des ADS1115 nur die obersten 12, mit denen auch der PCA9685 etwas anfangen kann.

Eine Möglichkeit dazu wäre, den Messwert einfach um 3 Bit-Positionen nach rechts zu schieben, dann fallen die untersten 3 Bits ins Nirvana und mit dem Rest füttern wir den Servo-Treiber. Wir gehen aber noch einen Schritt weiter. Dazu mehr nach dem Test.

Laden Sie dafür das Programm test_Joy.py in ein Editor Fenster und starten Sie es. Den Vorspann kennen wir schon. Unter Verwendung des I2C-Objekts i2c instanziieren wir das ADS1115-Objekt **joy**. Das ADS1115-Modul liegt auf I2C-Hardware-Adresse 0x48. Den **Switch** des Joysticks legen wir auf den Pin **GPIO0**, den wir als Eingang mit Pullup-Widerstand definieren.

```
#pca_first.py
# import webrepl_setup
# > d fuer disable
# Dann RST; Neustart!
import os,sys       # System- und Dateianweisungen
from machine import Pin, reset, I2C
from time import sleep, ticks_ms
from ads1115 import ADS1115
import esp
esp.osdebug(None)
import gc
gc.collect()

# Pintranslator fuer 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

sclPin=Pin(5)
```

```
sdaPin=Pin(4)
i2c=I2C(scl=sclPin,sda=sdaPin,freq=400000)

joy=ADS1115(i2c,0x48)
sw=Pin(0,Pin.IN,Pin.PULL_UP)

while 1:
    joy.setChannel(0) #
    x=joy.getConvResult()
    joy.setChannel(1) #
    y=joy.getConvResult()
    print(x,y)
    sleep(1)
    if sw.value()==0:
        break
while 1:
    joy.setChannel(0)
    x=joy.transform((-joy.getConvResult()),-28200,0,0,4096)
    joy.setChannel(1)
    y=joy.transform((-joy.getConvResult()),-28200,0,0,4096)
    print(x,y)
    sleep(1)
    if sw.value()==0:
        break
```

In der ersten while-Schleife stellen wir den Eingang des ADS1115 auf Kanal0 und rufen das Konversionsergebnis ab. Dasselbe machen wir mit Kanal1. Wir lassen die Werte im Terminal ausgeben und warten eine Sekunde.

Die gemessenen Werte sind positiv, das war wegen der Basiseinstellungen des ADS1115 (unipolar bis 4,096V) während der Instanziierung nicht anders zu erwarten. Aber die kleinen Werte kommen, wenn der Steuerknüppel nach rechts oder von uns weg, also nach hinten, gedrückt wird. Das ist ungünstig, umgekehrt wäre es besser.

Das regeln wir mit der Methode **transform**() aus der Klasse ADS1115. Zuerst machen wir den Messwert negativ. Damit spiegeln wir quasi den Steuerknüppel. Aber negative Werte können wir gar nicht brauchen und deshalb lassen wir uns durch die Methode **transform**() den Wertebereich von -28000 bis 0 auf den Bereich 0 bis 4096 abbilden. Eine Methode dieser Art gibt es zwar in LUA-NodeMCU (match) aber nicht in MicroPython. Also habe ich mir das selber gebastelt.

Ein Druck auf den Steuerknüppel beendet den ersten Teil. In der zweiten while-Schleife bekommen wir also Werte im brauchbaren Bereich und in der gewünschten Zuordnung zur Bewegung des Steuerknüppels. Durch Verändern der Transformations-Zielwerte können wir außerdem sogar den Ergebnis-Bereich anpassen. Probieren Sie doch einmal folgende Einstellungen aus:

```
x=joy.transform((-joy.getConvResult()),-26500,0,0,4096)
y=joy.transform((-joy.getConvResult()),-26500,0,0,4096)
```

oder

x=joy.transform((-joy.getConvResult()),-26500,0,260,2350)
y=joy.transform((-joy.getConvResult()),-26500,0,500,2100)

Der Servo-Test

Das Testprogramm für die Servos braucht nur wenig mehr als das Programm **test_Joy**.py. Wir instanziieren das Servo-Objekt **servo** durch Übergabe des I2C-Objekts **i2c** und des optionalen Parameters **oe**, der den GPIO2 als Steuerausgang zur Freischaltung der Ausgänge des PCA9685 definiert. Der optionale Parameter **freq** ist nicht aufgeführt, weshalb der Defaultwert 50 wirksam wird.

```
#pca_first.py
# import webrepl_setup
# > d fuer disable
# Dann RST; Neustart!
import os,sys       # System- und Dateianweisungen
from machine import Pin, reset, I2C
from time import sleep, ticks_ms
from pca9685 import SERVO
from ads1115 import ADS1115
import esp
esp.osdebug(None)
import gc
gc.collect()

# Pintranslator fuer 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

sclPin=Pin(5)
sdaPin=Pin(4)
i2c=I2C(scl=sclPin,sda=sdaPin,freq=400000)

oePin=Pin(14,Pin.OUT)

servo=SERVO(i2c,oe=2)
joy=ADS1115(i2c,0x48)

while 1:
    joy.setChannel(0)
    x=max(min(joy.getConvResult()>>3,3400),0)
    pulseX=joy.transform(-x,-3400,0,300,2100)
    joy.setChannel(1)
    y=max(min(joy.getConvResult()>>3,3400),0)
    pulseY=joy.transform(y,0,3400,350,2350)
    servo.writePulseTimeSlice(4,0,pulseX)
    servo.writePulseTimeSlice(7,0,pulseY)
    print(x,y,"--",pulseX,pulseY)
```

Zum Einlesen der Kanäle und der Transformation der Werte folgt in der while-Schleife statt der Ausgabe der Werte, deren Weiterleitung an die Servos. - Es ist schon ein Erlebnis, wenn die Bühnen am Spieltisch plötzlich anfangen zu wippen und zu schaukeln.

Üblicherweise gibt es auf den analogen Eingangsleitungen einen Rauschpegel, den der ESP8266 selbst erzeugt oder der vom Elektrosmog aus der Umgebung stammt. Nach der AD-Wandlung äußert sich das in vogelwildem Wechsel der niederwertigen Stellen der Wandlerwerte. Das gilt es zu unterdrücken. Aus diesem Grund habe ich auch einen Wandler mit einer Auflösung von 16 Bit gewählt. Positive Signale bis 4,096V werden mit 15-Bit Breite auf Werte von 0x0000 bis 0x7FFF umgesetzt. Wenn ich nun die unteren 3 Bits ins Nirvana schicke, indem ich den Wandlerwert um 3 Binärstellen nach rechts schiebe, habe ich immer noch 12 Bit Auflösung übrig und das Rauschen ist ausgefiltert.

Die Wandlerwerte grenze ich nun auf 0 bis 3400 ein, das ist der Bereich, den ich mit dem Joystick vorgeben kann. Dann setze ich die Werte mit **transform**() um und zwar so, dass die Grenzwerte einen Mittelwert ergeben, bei dem der jeweilige Boden waagrecht liegt und die Kugel in Ruhe liegen bleibt.

Wie mit all diesen Vorbereitungen nun ein Spiel programmiert werden kann, das sehen Sie in der nächsten Folge. Dann werden wir uns auch die Klasse PCA9685 näher anschauen und daran das Verhalten des PCA9685-Chips studieren. Jetzt wünsche ich schon mal viel Vergnügen beim Ausprobieren der Infrastruktur und beim Feintuning. Bis dann!