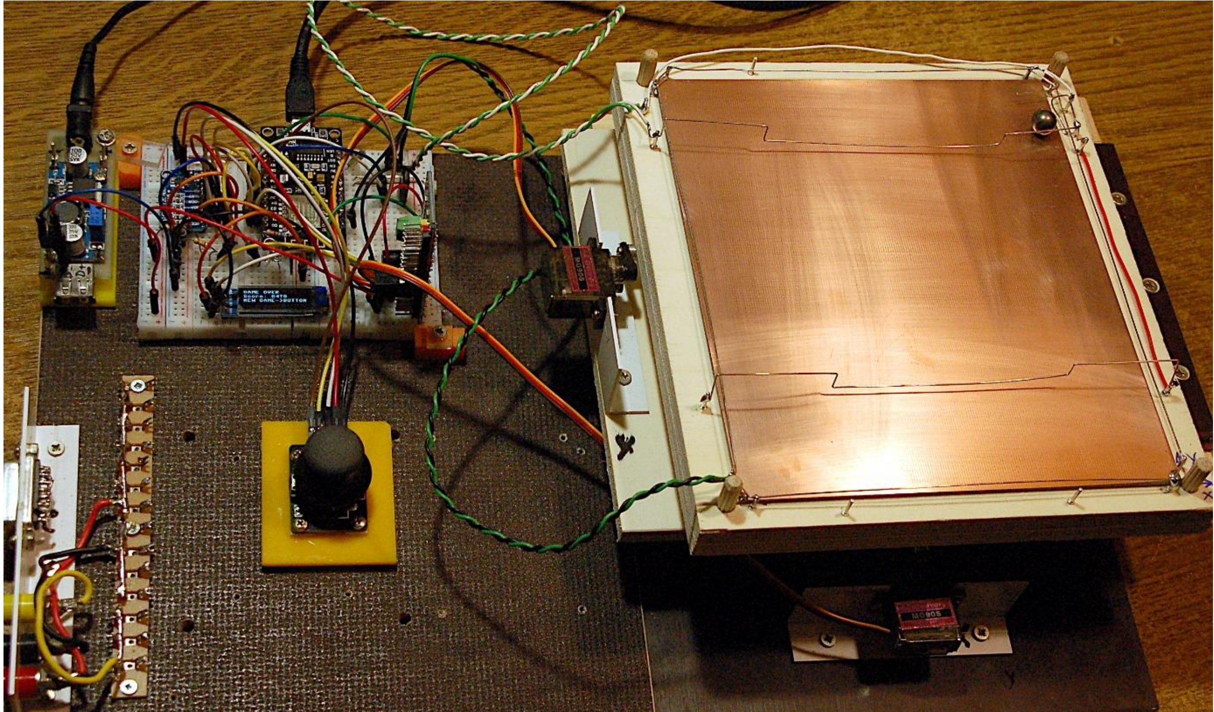
Gesamtaufbau

Dieser Blogpost ist auch als PDF-Dokument verfügbar.

Nach dem Aufbau der Mechanik und den Funktionstests der Hardware in Teil 1 starten wir heute mit der Programmierung einer Spielvariante und einem Blick hinter die Kulissen. Wir schauen uns an, wie ein Servo funktioniert und welchen Beitrag dazu unser Servotreiber-Modul PCA9685 leistet. Dabei erfahren Sie auch, wie man aus den Informationen des Datenblatts ein eigenes MicroPython-Modul baut. Damit willkommen zu einer neuen Episode aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der Joy Ball Wizzard (Teil2)

Die benötigten Teile für die Mechanik und die Hardware habe ich hier noch einmal im Überblick zusammengestellt. Eine detaillierte Beschreibung für den Zusammenbau finden Sie, zusammen mit dem Schaltplan der Elektronik und der Hardware-Beschreibung, im Teil1.

Material für die Mechanik

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,

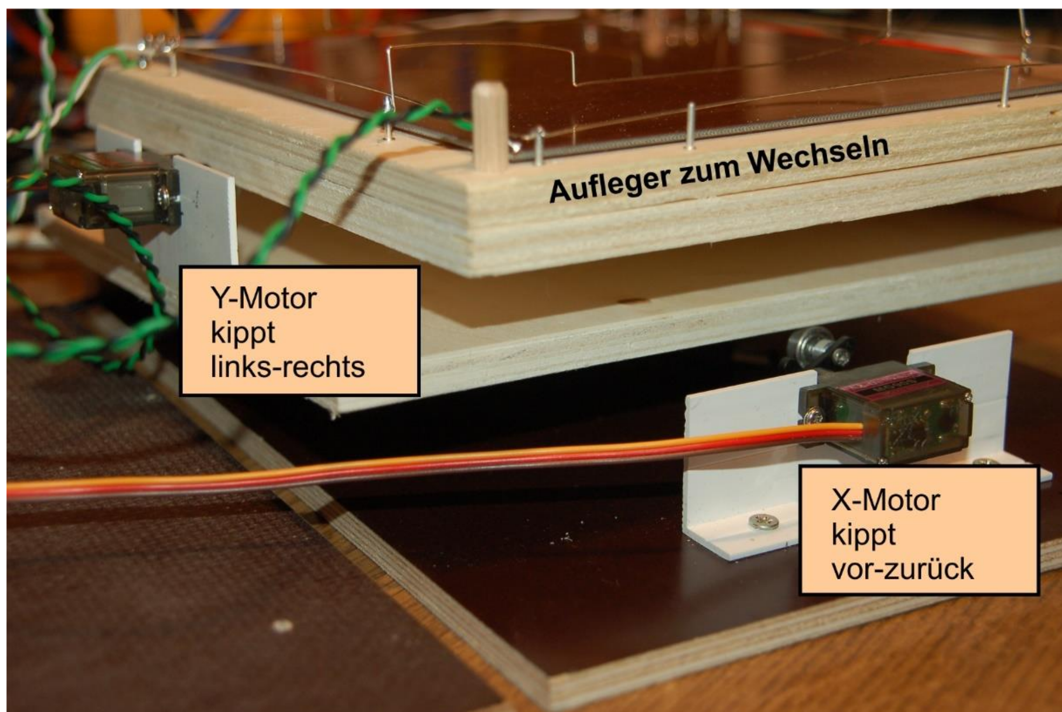

Abbildung 1: Plattenaufbau mit Servos

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]

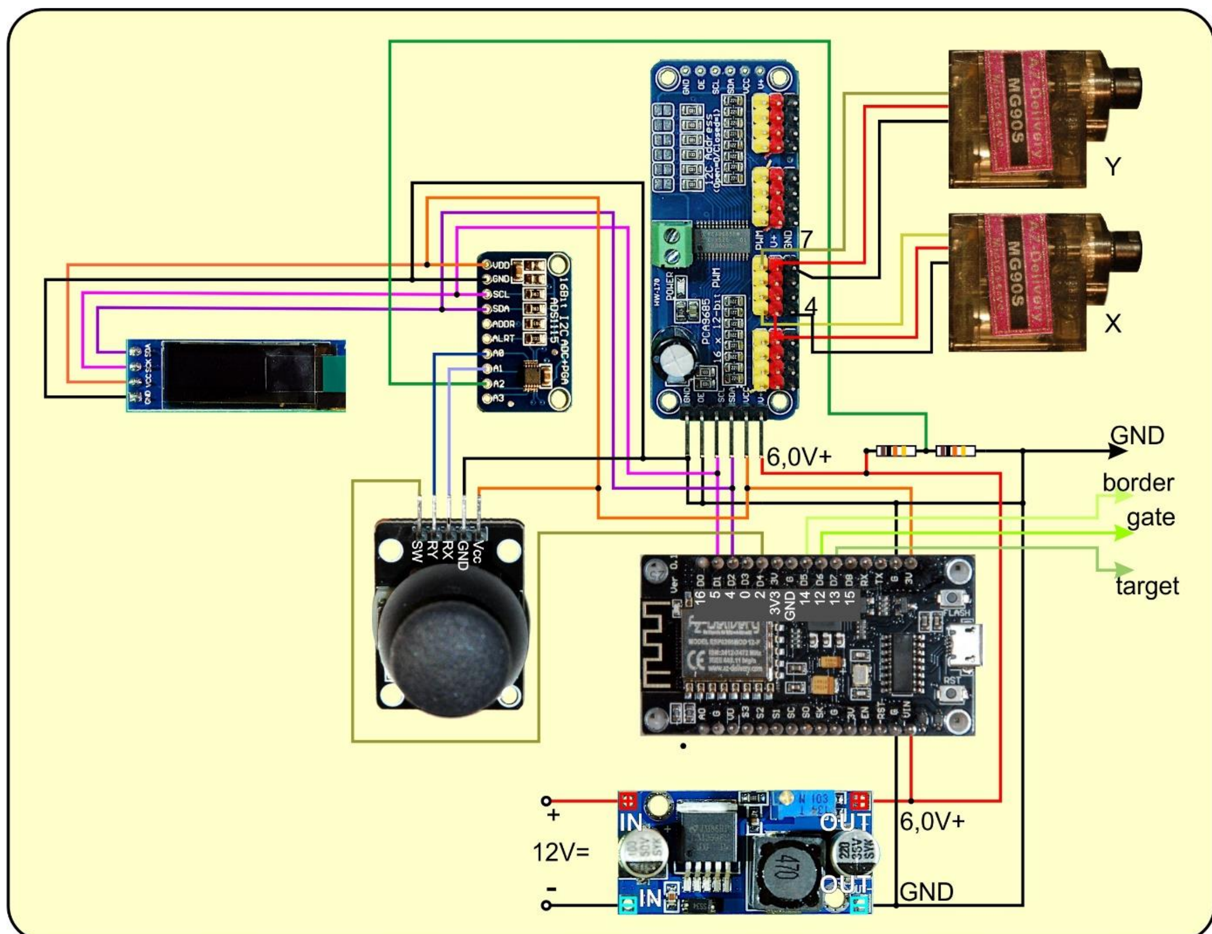

Abbildung 2: Joy Ball Wizzard-Schaltung

Eine größere Darstellung gibt es [hier als PDF-Dokument].

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.

Einen Servo ansteuern, wie geht das?

Andere Fragestellung: Warum bleibt ein Elektromotor an einer bestimmten Drehposition einfach stehen?

Ein Servomotor, kurz Servo, besteht aus einem Elektromotor, der über eine Getriebeuntersetzung einen Hebelarm antreibt. Warum der keine ständige Drehbewegung durchführt, liegt daran, dass an die Drehachse ein Potentiometer angekoppelt ist, also ein einstellbarer Widerstand. Die Drehbewegung stoppt, wenn das Signal von der Servo-Eingangsleitung mit dem vom Poti hervorgerufenen Signal übereinstimmt.

Über die Eingangsleitung (gelb oder orange) des Servos wird aber keine variable Gleichspannung übertragen, sondern Impulse von bestimmter Länge. Die Pulsdauer liegt zwischen 0,3ms (Rechtsanschlag oder 0°) und 2,3ms (Linksanschlag oder 180°) und wird im Rhythmus von 20ms wiederholt. Innerhalb dieser Periodendauer könnte man daher, mit nur einem Zeitgeber durch eine geschickt gestaffelte Zeitsteuerung über Software, an bis zu acht IO-Pins Servomotoren unabhängig voneinander steuern. Das macht der Arduino mit dem ATMega328 genau in dieser Form über eine entsprechende Library. Ich habe so etwas auch schon einmal in AVR-Assembler umgesetzt. Spannende Sache!

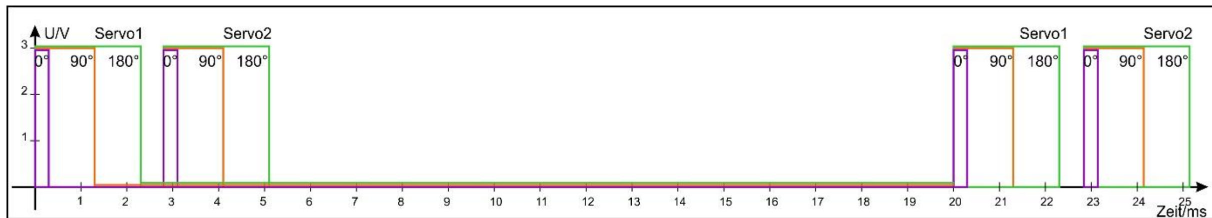

Abbildung 3: Servoimpulsfolge

Unsere Servosteuerung mit dem Chip PCA9685 arbeitet ähnlich, indem je Kanal innerhalb von 4096 möglichen Zeitpunkten die Ausgangsleitung ein- und ausgeschaltet wird. Die Zeitpunkte geben wir vor, indem wir deren Nummern in entsprechende Register schreiben. Den Rest erledigt der PCA9685. Der Baustein kommt eigentlich aus der LED-Ecke, ist aber auch hervorragend für die Steuerung von Servomotoren geeignet. Das Datenblatt bietet NXP zum Download an. Auf dieser Grundlage habe ich ein MicroPython-Modul mit der Klasse PCA9685 programmiert. Damit werden wir uns gleich beschäftigen, nachdem ich noch auf ein paar Einzelheiten des Chips eingegangen bin.

Der PCA9685 wird über I2C-angesteuert und bietet zur Adressierung 6 Adressleitungen A0 bis A5, die durch Lötbrücken gejumpert werden können und so eine sehr flexible Auswahl einer Hardware-Adresse aus 62 möglichen erlauben. Wir benutzen die Basisadresse 0x40. Die Eingänge sind 5V-resistent, daher kann das Modul auch an einem Arduino betrieben werden. Jeder der 16 Ausgänge wird durch zwei mal zwei Register im Zeitverhalten gesteuert. Weil für jeden Schaltvorgang ein Wert von 0 bis 4095 möglich ist, brauchen wir zwei Bytes für dessen Speicherung. Die PWM-Frequenz wird vom internen 25MHz-Takt abgeleitet und kann für alle Ausgänge gemeinsam zwischen 24Hz und 1526Hz eingestellt werden. Innerhalb des durch die Periodendauer festgelegten Zeitfensters, kann in einer Abstufung von 0 (0%) bis 4095 (100%) die Pulsweite des Ausgangssignals (aka Dutycycle) programmiert werden. Weil die Startzeit des Ausgangspulses und seine Abfallzeit unabhängig voneinander gesetzt werden können, ist nicht nur die Pulsdauer mit einer Auflösung von 1 / (PWM_Frequenz * 4096) einstellbar, sondern der Puls ist innerhalb der Periodendauer auch noch frei verschiebbar. Für die potenzielle Erweiterung des Spiels durch zusätzliche Servos ist das Modul daher erste Sahne zumal die GPIO-Pins des ESP8266 anderweitig gebraucht werden.

Eine recht unangenehme Entdeckung habe ich beim ESP32 mit den Firmware-Versionen 1.17 und 1.18 gemacht. Dort funktioniert die PWM-Steuerung aus unersichtlichen Gründen erst ab 700Hz. Das ist für Servos völlig von der Reihe und der Grund, weshalb ich mit für den PCA9685 entschieden habe.

Die Klasse PCA9685

Kommen wir zur Klasse PCA9685. Die Klassendefinition beginnt wie üblich mit dem Schlüsselwort **class** und dem Namen **PCA9685**, der Körper der Klasse wird eingerückt. Es folgen eine ganze Reihe von Konstanten-Definitionen, auf die ich nicht einzeln eingehe. Es handelt sich um die Benennung von Registern und Flags. Alle beginnen mit einem Großbuchstaben, so werden die Bezeichner im Programm als Namen für Konstanten erkennbar.

Jede Klasse braucht einen sogenannten Konstruktor, nach dessen Bauplan Objekte der Klasse abgeleitet werden können. Der Konstruktor in einer MicroPython-Klasse ist eine Methode mit dem festen Namen **__init__().** Später, beim Aufruf, wird dann statt **__init__** der Name der Klasse benutzt, hier also **PCA9685**.

```
    def __init__(self,i2c, hwadr=None, oe=None, freq=None):
        self.i2c=i2c
        if oe is not None:
            self.oe=Pin(oe,Pin.OUT,value=1)
        else:
            self.oe=None
        if hwadr is None:
            self.hwadr=HWADR
        else:
            self.hwadr=hwadr
        self.resetDevice()
        if freq is not None:
            self.freq=freq
        else:
            self.freq=SERVO_FREQ
        self.start(self.freq)
        self.setAutoInc()
        print("Constructor PCA9685, HWADR=",hex(self.hwadr))
```

Der Parameter und das Prefix **self** stellt eine Referenz auf das später erzeugte Objekt dar. Damit die restlichen Parameter **i2c**, **hwadr**, **oe** und **freq** den Methoden der Klasse zur Verfügung stehen, werden die übergebenen Objekte an Instanzvariablen übergeben, die ebenfalls mit dem Vorspann **self** beginnen. Diese Variablen sind innerhalb eines Objekts, das aus der Klasse abgeleitet wird, **global**, aber in Bezug auf mehrere Objekte derselben Klasse **lokal**.

Das **None** referenziert einen einfachen, aber auch interessanten Datentyp für den es nur eine einzige Instanz gibt, das **Nichts**. In unserer Parameterliste heißt das, dass etwa für **hwadr** standardmäßig nichts übergeben wird, wenn der Parameter bei der Instanziierung keinen Wert zugewiesen bekommt. Es wird eben nicht die Referenz auf einen (Zahlen-)Wert zugewiesen, sondern die Referenz auf die Instanz des Nichts. Dieses Vorgehen erlaubt uns, durch die if-Konstrukte flexibel auf bestimmte Situationen zu reagieren. In unserem speziellen Fall muss der Benutzer der Klasse nicht wissen, welche Hardwareadresse der PCA9685 hat, oder welche PWM-Frequenz für ein Servo nötig ist. Das ist durch die Konstanten vorbelegt, und dieser Wert wird im Konstruktor verwendet, wenn keine eigenen Wünsche vorliegen.

Eine Instanz servo der Klasse PCA9586 würde nun am einfachsten folgendermaßen erzeugt. Das setzt natürlich voraus, dass zuvor das I2C-Objekt i2c instanziiert wurde.

```
servo = SERVO(i2c)
```

Die Methode **resetDevice**() tut zweierlei, sie prüft, ob ein PCA9685-Device auf dem Bus liegt und setzt alle derartigen Module zurück. Dabei wird die General Call Address **0x00** als Hardwareadresse benutzt und als zweites Byte der Resetcode **0x06** gesendet. Die Methode sollte 2 als return-Code melden, das ist die Anzahl der auf dem Bus empfangenen ACK-Bits. Beschrieben ist das Vorgehen auf den Seiten 7 (unten) und 8 (oben) des Datenblatts.

```
>>> from machine import Pin, I2C
>>> from pca9685 import SERVO

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

>>> servo=SERVO(i2c,oe=2)
>>> resetDevice()
```

Das sieht auf den Busleitungen SCL (gelb) und SDA (blau) so aus.

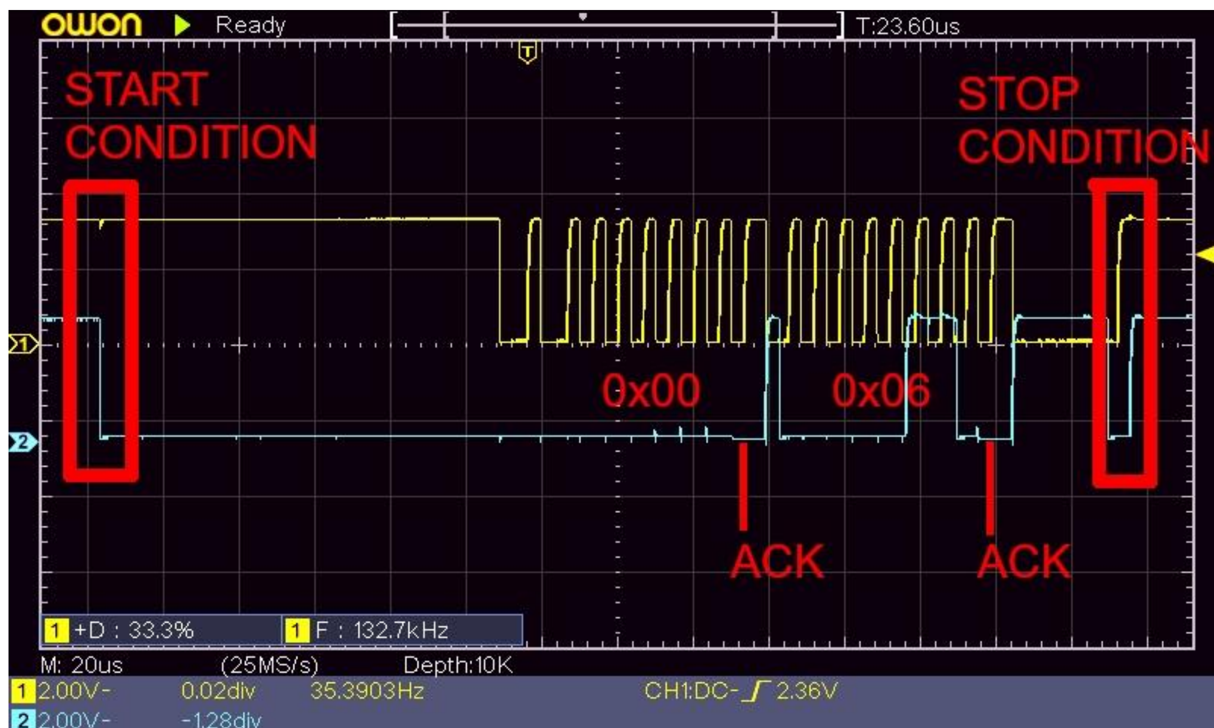

Abbildung 4: Reset-Befehl auf dem I2C-Bus

Die Ruhepegel auf beiden Leitungen sind logisch 1 oder elektrisch 3,3V, hervorgerufen durch die Pullup-Widerstände. Eingeleitet wird ein Transfer durch die Start-Condition. Das ist eine fallende Flanke auf der SDA-Leitung, während SCL auf

HIGH ist. Dann setzt der Master (ESP8266) das MSBit der Hardware-Adresse auf SDA (hier 0) und zieht SCL auf LOW. Mit der folgenden steigenden Flanke auf SCL signalisiert er dem Slave (PCA9685), dass das Datenbit gültig und zu übernehmen ist. Das wiederholt sich mit den weiteren 7 Bits. Während eines neunten Takts kann nun der Slave die SDA-Leitung auf LOW ziehen. Das wird als Acknowledge (ACK) bezeichnet und sagt dem Master, dass das gesendete Byte angenommen wurde. Das ACK ist allerdings keine Bestätigung dafür, dass das Byte unverfälscht angekommen ist. Mit einem NACK, not acknowledged, SDA bleibt HIGH, sagt der Slave, dass er alle erforderlichen Bytes erhalten hat, oder im Moment keine Zeichen empfangen kann. Nach der Hardware-Adresse sendet der Master den Reset-Code 0x06, der vom Slave ebenfalls mit ACK bestätigt wird. Bis die Bytes verarbeitet sind zieht der Slave nun die SCL-Leitung auf LOW. Nachdem die Leitung auf HIGH gegangen ist, teilt der Master dem Slave das Ende der Übertragung durch eine Stop-Condition mit. Das ist die steigende Flanke auf SDA, während SCL HIGH ist. Nun sind beide Leitungen wieder im Ruhezustand.

Für die restlichen Schreib- und Lesebefehle auf dem Bus muss die reguläre Hardware-Adresse des PCA9685 benutzt werden. Wir erzeugen drei Basis-Methoden für diesen Zweck.

```
    def writeReg(self,reg,val):
        buf=bytearray(2)
        buf[0]=reg
        buf[1]=val
        self.i2c.writeto(HWADR,buf)
```

Um einen Wert in ein Register des PCA9685 zu schreiben brauchen wir drei Datenbytes, die Hardware-Adresse **HWADR**, die Registernummer **reg** und natürlich den Datenwert **val**. Die Methode **i2c.writeto**() verlangt die Hardware-Adresse und einen Buffer, der auf dem Bytes-Protokoll fußt. Das trifft für die übergebenen Ganzzahlwerte in **reg** und **val** nicht zu. Wir erzeugen daher ein **bytearray**-Objekt mit zwei Elementen und weisen diesen die Registernummer und den Datenwert zu. Dann schicken wir alles über die Leitung.

Die drei Bytes werden mit ACK (acknowledge = OK) vom PCA9685 quittiert (SDA=0). Die 7-Bit-Hardware-Adresse 0x40 wird um ein Bit nach links geschoben, das write-Bit (SDA=0) als LSBit eingefügt und dann werden die beiden Bytes aus dem Buffer auf die Leitung getaktet. Aus dem Oszillogramm erkennen wir außerdem, dass die effektive Taktrate nicht 400kHz sondern nur 150kHz beträgt (9 Taktimpulse in 60µs).

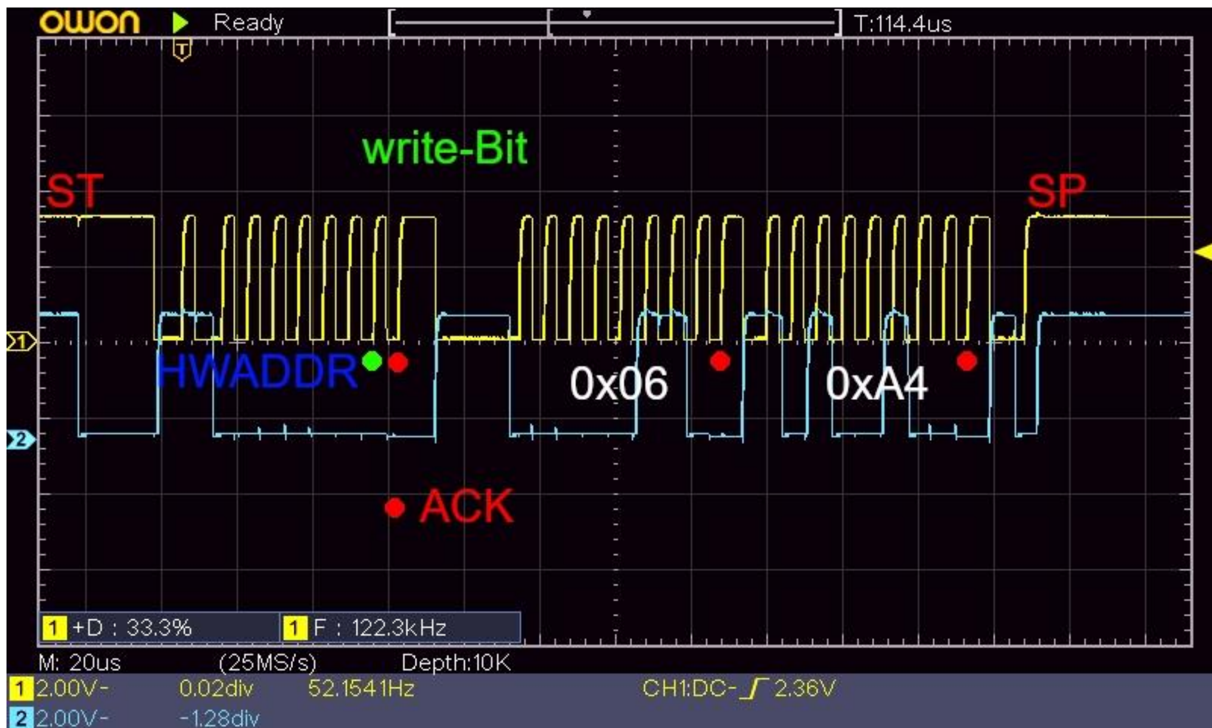

Abbildung 5:Beschreiben des Registers 0x06 mit 0xA4

```
    def readReg(self,reg):
        buf=bytearray(1)
        buf[0]=reg
        self.i2c.writeto(HWADR,buf)
        buf=self.i2c.readfrom(HWADR,1)
        return buf[0]
```

Der Lesebefehl für ein Register erfolgt zweistufig. Zuerst wird mit der Hardware-Adresse die Registeradresse in einem Schreibbefehl gesendet. Danach wird erneut die Hardware-Adresse, dieses Mal mit gesetztem Lese-Bit als LSB, gesendet und infolge davon antwortet der PCA9685 mit dem Inhalt des Registers, das jetzt der ESP8266 mit ACK quittiert. Es weiß, dass nur ein Byte zurückkommt, deshalb ist der Empfangs-Buffer auch nur 1 Byte lang.

```
    def writeToRegs(self,reg,buf):
        buffer=bytearray(1)
        buffer[0]=reg
        buffer+=buf
        noa=self.i2c.writeto(HWADR,buffer)
        return noa # number of ACKs
```

Sollen mehrere Bytes gesendet werden, zum Beispiel, um die Start- und Endzeit für PWM-Pulse zu schreiben, dann setzen wir einfach die Registernummer im Bytearray mit dem übergebenen Buffer zusammen. Der PCA9685 ist so konfiguriert, dass er selbständig die Empfangsregisternummer mit jedem empfangenen Daten-Byte hochzählt (**self.setAutoInc**() im Konstruktor).

Eine besondere Vorgehensweise ist nötig, wenn der PCA9685 im laufenden Betrieb zurückgesetzt werden soll, nachdem wir ihn schlafen gelegt haben. Das macht die Routine **doRestart**() nach den Vorgaben im Datenblatt auf Seite 15.
Zitat:
To restart all of the previously active PWM channels with a few I2C-bus cycles do the following steps:
1. Read MODE1 register.
2. Check that bit 7 (RESTART) is a logic 1. If it is, clear bit 4 (SLEEP). Allow time for oscillator to stabilize (500µs).
3. Write logic 1 to bit 7 of MODE1 register. All PWM channels will restart and the RESTART bit will clear.

```
    def doRestart(self):
        val=self.readReg(MODE1)
        if val & Restart:
            val &= 255-Sleep
            self.writeReg(MODE1,val)
            sleep_us(500)
            val |= Restart
            self.writeReg(MODE1,val)
```

Der Zusammenhang erklärt sich wie folgt. Das Setzen des **Sleep**-Bits in Register **Mode1** schickt den PCA9685 schlafen. Das kann durch die Methode **setSleep**() geschehen, zu der wir später kommen. Gleichzeitig wird auch das Bit **Restart** in **Mode1** gesetzt. Die Positionen der Servos sind jetzt eingefroren und reagieren nicht mehr auf neue Werte, die wir dem PCA9685 senden. Neue Werte werden aber dennoch in die Positionsregister geschrieben. Der Algorithmus in **doRestart**() entriegelt die Sperre, indem er das **Sleep**-Bit löscht, den Oszillator hochfährt und schließlich das **Restart**-Bit durch Schreiben einer 1 löscht. Wurde in der Zwischenzeit keine Änderung der Positionsregister vorgenommen, bleibt der Servo auf seiner Position stehen. Hat man aber eine neue Position gesendet, dann wird diese angefahren.

Unser Ziel ist es, den PCA9685 dazu bringen, dass er Steuerimpulse an die Servos schickt. Das mach die Methode **writePulse**(). Sie nimmt die Nummer des Servos sowie die Nummern der Zeitscheiben für das Einschalten und Ausschalten.

```
    # Zeiten relativ in 0 .. 4096 counts
    def writePulse(self,nbr,on,off):
        assert 0 <= nbr <= 15
        off=min(off, 4095)
        on=min(max(0,on),off)
        buffer=bytearray(5)
        buffer[0]=nbr*4+LEDBASIS
        buffer[1]=on & 0xff
        buffer[2]=(on>>8) & 0xff
        buffer[3]=off & 0xff
        buffer[4]=(off>>8) & 0xff
        noa=self.i2c.writeto(HWADR,buffer)
        return noa
```

Mit **assert** überprüfen wir die Servo-Nummer und grenzen dann die Start- und Ende-Werte auf den Bereich zwischen 0 und 4095 ein. Ein Buffer mit 5 Elementen wird mit der Registeradresse und den in Bytes aufgelösten Positionswerten gefüttert. Die Positions-Register beginnen bei **LEDBASIS**, das ist das Register **0x06**. Jeweils vier Register sind zu beschicken, die Notation ist **Little Endian**, das LSB kommt also zuerst. Dann ist alles bereit, die Hardware-Adresse und der Buffer gehen über die Leitung. Ein Rückgabewert von 5 sagt uns, dass 5 Bytes vom PCA9685 mit ACK quittiert wurden.

Zwei weitere Methoden benutzen **writePulse**(). Die Methode **writePulseFromWidth**() arbeitet auch mit den relativen Zeitscheiben 0 bis 4095. Servo-Nummer und Startzeit werden weitergereicht, der Zeitpunkt zum Ausschalten muss berechnet werden, die Plausibilitätskontrolle übernimmt **writePulse**().

```
    # Zeiten relativ in 0 .. 4096 counts
    def writePulseFromWidth(self, nbr, on, width):
        off=on+width
        noa=self.writePulse(nbr,on,off)
        return noa
```

writePulseTimeSlice() ist die Methode der Wahl für unser Spiel. Mit dem Parameter **run** geben wir die Zeitdauer für den Impuls an, der auf die Servo-Steuerleitung gelegt wird. Er darf 0,3ms bis 2,3ms dauern. Der Startzeitpunkt ist marginal. Aus der eingestellten PWM-Frequenz wird die Periodendauer in Mikrosekunden und damit die Länge einer relativen Zeitscheibe berechnet. Die relativen Werte gehen dann an die Methode **writePulse**().

```
    # Zeiten relativ in µs; Servo-Freq in Hz
    def writePulseTimeSlice(self,nbr,start,run):
        timeLimit=(1/self.freq)*1000000
        timeslice=timeLimit/4096 # µs/cnt
        assert 0 <= start <= timeLimit
        assert run >= 0
        assert start+run <= timeLimit
        von=int(start/timeslice)
        bis=int((start+run)/timeslice)
        noa=self.writePulse(nbr,von,bis)
        return noa
```

Ein paar Service-Routinen runden die Klasse PCA9685 ab.

```
    def setAutoInc(self,ai=True):
        val=self.readReg(MODE1)
        if ai:
            self.writeReg(MODE1,val | AutoInc)
        else:
            self.writeReg(MODE1,val & (~AutoInc) & 0xFF)
```

In MicroPython ist es nur mit Tricks möglich, die bitweise Negierung eines Bytes zu bekommen. Wir müssen das also selbst in die Hand nehmen. Zwei Verfahren führen zum Ziel. **servo**.**AutoInc** hat zum Beispiel hier den Wert 32=0b00100000. Für die Negierung **~servo.AutoInc** erwarten wir den Wert 0b11011111= 223, bekommen aber -33 = -0b100001. Aber **255 – servo.Autoinc** liefert das, was wir möchten und ebenso **~servo.AutoInc & 0xFF**. Letzteres erklärt sich aus der Tatsache, dass -33 = 0xFFFFFFDF. Durch Undieren mit 0xFF erhalten wir 0xDF = 0b11011111.

Wir lesen also das Mode1-Register ein und schreiben es mit gesetztem oder gelöschtem **AutoInc**-Bit wieder zurück.

```
    def setSleep(self,mode=True):
        val=self.readReg(MODE1)
        if mode:
            val = val | Sleep
        else:
            val = val & (~Sleep & 0xff)
        self.writeReg(MODE1,val)
```

Zum Ändern der PWM-Frequenz ist es nötig, das Sleep-Bit vorher zu setzen (default) und danach wieder zu löschen (mit **mode**=**False**). Genau das macht **setSleep**().

start() setzt die PWM-Frequenz für unsere Servos auf den Defaultwert von 50 Hz (ohne Parameter im Aufruf) oder die Frequenz wird auf den übergebenen Wert gesetzt. Der Wert wird dafür in das Prescaler-Register geschrieben. Bis die neue Einstellung stabil arbeitet, warten wir 500µs.

```
    def start(self,freq=50):
        if self.oe is not None:
            self.oe.value(0)
        self.frequency(freq)
        sleep_us(500)
```

Für die Berechnung des Prescaler-Werts offenbart uns die Seite 25 des Datenblatts folgende Formel.

ps = int (PCA_Systemtakt / (4096 * PWM-Frequenz)) - 1

Außerdem erhalten wir den Hinweis:

The PRE_SCALE register can only be set when the SLEEP bit of MODE1 register is set to logic 1.

Genau danach geht die Methode **frequency**() vor. Wenn man sie mit Parameter aufruft, wird dieser Wert, falls zulässig, in das PRE_SCALE-Register **0xFE** geschrieben. Ohne Parameter aufgerufen, erfahren wir den aktuell gesetzten Wert der PWM-Frequenz.

```
    def frequency(self,freq=None):
        if freq is not None:
            assert 24 <= freq <=1526
            self.setSleep()
            self.freq=freq
            ps=(XTAL//(4096*freq))-1
            self.writeReg(PRE_SCALE,ps)
            self.setSleep(False)
        else:
            return XTAL/((self.readReg(PRE_SCALE)+1)*4096)
```

Den gesamten Code des Moduls pca9685.py können Sie, ebenso wie die anderen Module, herunterladen. Folgen Sie dazu auch den Links am Anfang des Kapitels Software.

Das Programm zum Spiel

Beginnen Sie am besten mit dem Upload der vier Module **ads1115**.py, **oled**.py, **ssd11306**.py und **pca9685**.py in den Flash des ESP8266. Dann wollen wir sehen, wie das Programm arbeitet.

Die Importe bringen uns die Methoden aus den Klassen in unseren Namensraum.

```
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
from oled import OLED
import esp
esp.osdebug(None)
import gc
gc.collect()
```

Der I2C-Bus wird dreifach genutzt, deshalb deklarieren wir dessen Instanz im Hauptprogramm. und nicht in jedem einzelnen Modul.

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

Der Ausgang für **Output Enable -OE** am PCA9685 und die Sensoreingänge werden festgelegt.

```
oePin=Pin(2,Pin.OUT) # D4
border=Pin(12,Pin.IN,Pin.PULL_UP) # D6
gate=Pin(14,Pin.IN,Pin.PULL_UP) # D5
target=Pin(13,Pin.IN,Pin.PULL_UP) # D7
newstart=Pin(0,Pin.IN,Pin.PULL_UP) # D3
```

Die Instanzen für die Motorsteuerung, für die AD-Wandlung der Joystickspannungen und für die OLED-Ausgabe werden erzeugt.

```
servo=SERVO(i2c)
joy=ADS1115(i2c,0x48)
d=OLED(i2c,128,32)
```

Auf die Deklaration einiger Variablen folgt die Definition der Interrupt-Service-Routine (aka ISR), **bump**(), welche aufgerufen wird, wenn die Kugel die Sensoren berührt.

```
vorgabe=2000
points=vorgabe
zeitfaktor=50

# globale Variablen deklarieren
trigger=True
gameOver=False
start=0
```

Für die Kontakte zu den Sensoren können verschieden viele Punkte abgezogen werden. Damit nicht sofort wieder ein IRQ (Interrupt Request) ausgelöst werden kann, wird der Handler, also die Routine **bump**(), erst einmal außer Gefecht gesetzt, indem wir den Parameter **handler** auf das große Nichts setzen. **trigger** setzen wir auf **True**, damit die Hauptschleife auf das Event reagieren kann. **trigger** und **points** müssen als global angegeben werden, weil die Änderung der Werte sonst nicht über das Scope der Funktion nach draußen wirken kann.

```
def bump(pin):
    global points,trigger
    if pin==gate:
        points-=10
        gate.irq(handler=None)
    if pin==border:
        points-=10
        border.irq(handler=None)
    trigger=True
```

Die Funktion **goToOrigin**() erledigt alles, was zum Starten eines Spieldurchgangs nötig ist. Zur Rückführung der Kugel an den Start wird die ISR auf das große Nichts gesetzt, es soll ja kein IRQ ausgelöst werden. Das Display informiert uns, dass die Rückführung ein paar Takte dauert. Dann setzen wir den Kantentrigger auf True, geben die Starthilfe aus und warten auf den Klick des Joystick-Buttons.

Ist dieser erfolgt, füllen wir das Punktekonto auf, oder auch nicht, dann geht es mit dem vorherigen Punktestand weiter. Die Startmeldung wird ausgegeben, die Sensoren werden scharfgeschaltet und die Zeitmessung gestartet. Sobald die Kugel den Rand oder ein Gate berührt, geht der Eingang am ESP8266 von 3,3V auf 0V. Die fallende Flanke löst die Unterbrechungsanforderung aus und die Routine **bump**() wird aufgerufen.

Mit dem Start des Hauptprogramms messen wir den Wert der Versorgungsspannung. Das kann wichtig werden, wenn wir als Energiequelle nicht das 12V-Steckernetzteil, sondern einen Akku verwenden. Den Messwert bekommen wir mit der Begrüßungsmeldung am Display. Nach 5 Sekunden sorgt **goToOrigin**() für einen geordneten weiteren Start.

```
joy.setChannel(2)
uServo=2*joy.getVoltage()
d.clearAll()
d.writeAt("JOY-BALL-WIZZARD",0,0,False)
d.writeAt("WELLCOME!",0,1,False)
d.writeAt("U-Servo:{:.2f}V".format(uServo),0,2,True)
sleep(5)

goToOrigin()
```

In der Hauptschleife setzen wir den ADC-Kanal zunächst auf 0, holen den x-Wert vom Joystick, den wir auf 12 Bits reduzieren, gleich eingrenzen und auf den Zeitbereich des Servos transformieren. Dasselbe passiert mit dem Kanal1 für den y-Wert. Danach bekommen die Servos die entsprechenden Impulse über den PCA9685 zugespielt.

```
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)
    d.writeAt("score:{}       ".format(points),0,2)
```

Nach der Ausgabe des Spielstands behandeln wir besondere Ereignisse. **trigger** ist **True**, wenn ein IRQ ausgelöst wurde. In diesem Fall warten wir 50ms und schalten danach die Sensorik wieder scharf.

```
    if trigger:
        sleep(0.05)
        gate.irq(handler=bump,trigger=Pin.IRQ_FALLING)
        border.irq(handler=bump,trigger=Pin.IRQ_FALLING)
```

Wurde während einer Spielrunde der Joystick-Button gedrückt, löst das einen Neustart des ESP8266 aus, und alle Werte werden zurückgesetzt. Allerdings kann das Spiel nur dann wirklich neu starten, wenn das Programm als **boot**.py in den Flash hochgeladen wurde. Hinweise dazu finden Sie im Kapitel Sofware.

Eine Spielrunde ist beendet, wenn die Kugel in der Zielklammer landet und **target** damit auf 0 geht. Wir setzen **gameover** auf **True** und stellen die Spieldauer fest. Waren wir schneller als 15 Sekunden, gibt es die Restsekunden mal 50 als

Pluspunkte. Beim Überschreiten des Zeitlimits gibt es dementsprechende Abzüge. Der Spielstand ergibt sich dann aus der Summe der Punkte aus dem Startdepot und den Zeitpunkten. Der Spielstand wird mitgeteilt, und mit einem Klick des Joystick-Buttons geht es in die nächste Runde.

```
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)
        d.clearAll()
        d.writeAt("GAME OVER",0,0, False)
        pnts=str(score)
        d.writeAt("Score: "+pnts,0,1,False)
        d.writeAt("NEW GAME->BUTTON",0,2)

        while not newstart.value()==0:
            pass

        sleep(0.3)
        goToOrigin()
```

Ausblick

Nun liegt es an Ihnen, was sie aus dem Spiel noch alles machen. Das Spielfeld kann gewechselt werden, indem die oberste Platte mit der Platine ausgetauscht wird. Weitere Gates können eingefügt werden. Sie können auch Experimente mit der Steuerung durchführen, indem Sie die Werte für die Zeitscheiben verändern. Und ein oder mehr weitere Servos können eingesetzt werden, um variable Hindernisse zu schaffen. Lassen Sie Ihrer Phantasie einfach freien Lauf.

Viel Erfolg und jede Menge Spaß dabei, das wünsche ich Ihnen. In der nächsten Folge werden wir die Steuerung vom Joystick auf eine kabellose Schwerkraftsteuerung verlegen. Die Verbindung wird dann über WLAN und das UDP-Protokoll hergestellt. Lassen Sie sich überraschen, bis dann!