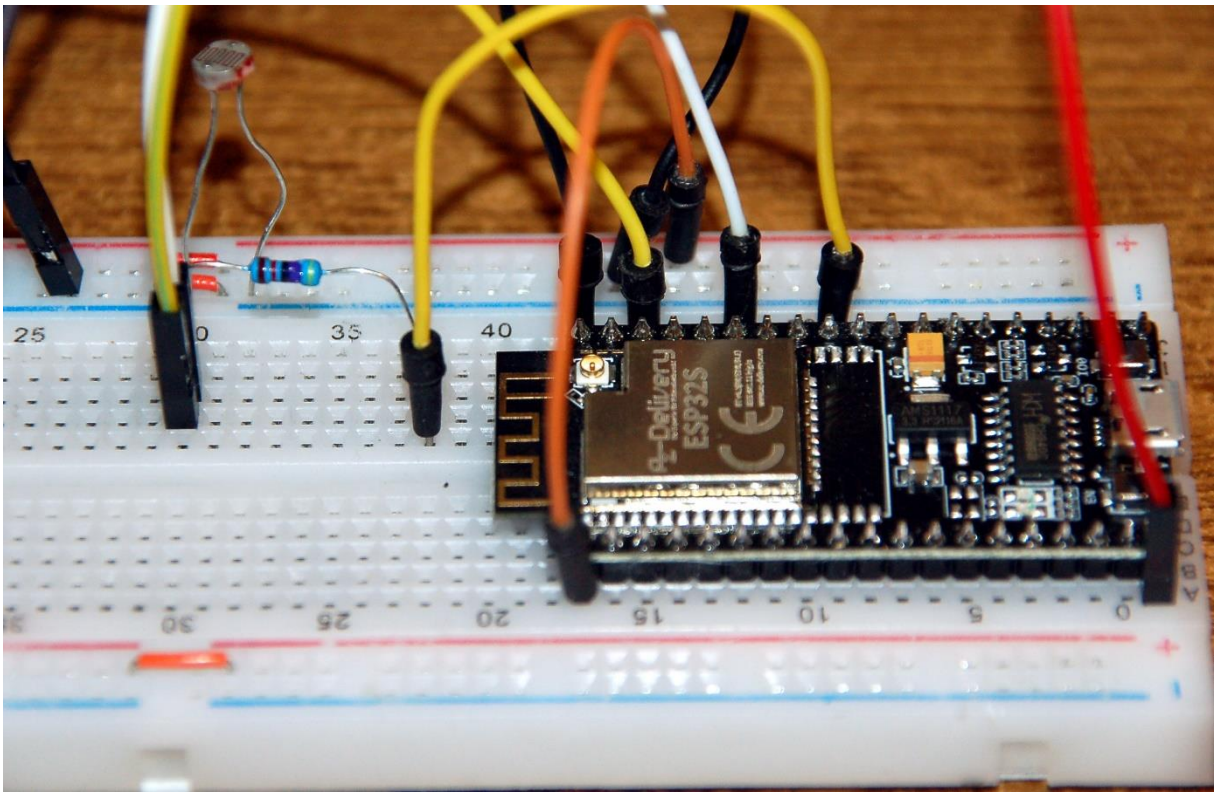

ESP32 - Magic Music Synthesizer

Diese Projektbeschreibung gibt es auch als PDF-Dokument.

Kennen Sie ein Musikinstrument, das mit den Händen gespielt wird, ohne dass die Hände, oder ein anderes Körperteil das Instrument berühren? So etwas gibt es seit den 20-er Jahren des letzten Jahrhunderts. Sein Name ist Theremin.

Das ursprüngliche Instrument bestand aus zwei Hochfrequenzsendern mit Antennen. Durch Annähern der Hände an diese Antennen und der daraus resultierenden Verstimmung der Sendefrequenz wurde letztlich die Lautstärke und die Tonhöhe des Instruments beeinflusst.

Dass es auch ohne Hochfrequenz und ohne Antennen geht, das zeige Ich Ihnen heute. Damit willkommen zu

MicroPython auf dem ESP32 und ESP8266

heute

Der ESP32 als Musiker

Beim Experimentieren mit einem phantastischen Modul, das eigentlich aus einer ganz anderen Anwendungsecke kommt, als aus dem Musikbereich, reifte nach und nach die Idee zur Verwendung dieses ToF-Moduls zur Erzeugung von Tönen.

ToF ist das Akronym für Time of Flight und das Modul **VL53L0X Time-of-Flight (ToF) Laser Abstandssensor** macht nichts anderes als die Flugzeit eines von ihm ausgesandten Laserimpulses bis zur Reflexion an einem Hindernis und zurück zu messen.

Messergebnisse können über den I2C-Bus abgerufen werden. Die Zahlenwerte bewegen sich im zwei- bis vierstelligen Bereich. Das ist ein zu eins in die Frequenz von Tonsignalen umzusetzen. Und dafür ist die Ausgabe eines PWM-Signals an einem beliebigen esp-GPIO-Pin einsetzbar.

Zwei Probleme gab es aber für die Lautstärkesteuerung zu lösen. Gleich zu Beginn stellte sich heraus, dass das oben genannte Modul eine feste Hardwareadresse hat und somit am gleichen Bus nur ein Modul verwendbar ist. Zwar gibt es I2C-Multiplexer, die als Schalter dienen, und somit zwischen zwei Modulen mit gleicher Adresse umschalten können. Mein ursprünglicher Ansatz war, das zweite Modul auch über ein PWM-Signal mit variablem Duty Cycle eine LED ansteuern zu lassen, die über einen Spannungsteiler mit einem LDR die Lautstärke hätte steuern sollen.

Aber manchmal sieht man den Wald vor lauter Bäumen nicht. Ohne LDR fand ich keine Lösung, der musste also in jedem Fall als Lautstärkeregler dienen. Aber dann kann man ihn doch einfach gleich mit Umgebungslicht beleuchten und mit der Hand abschatten um seinen Widerstandwert zu verändern.

Aus diesen Überlegungen entstand schließlich der Schaltplan für das ESP32-Theremin.

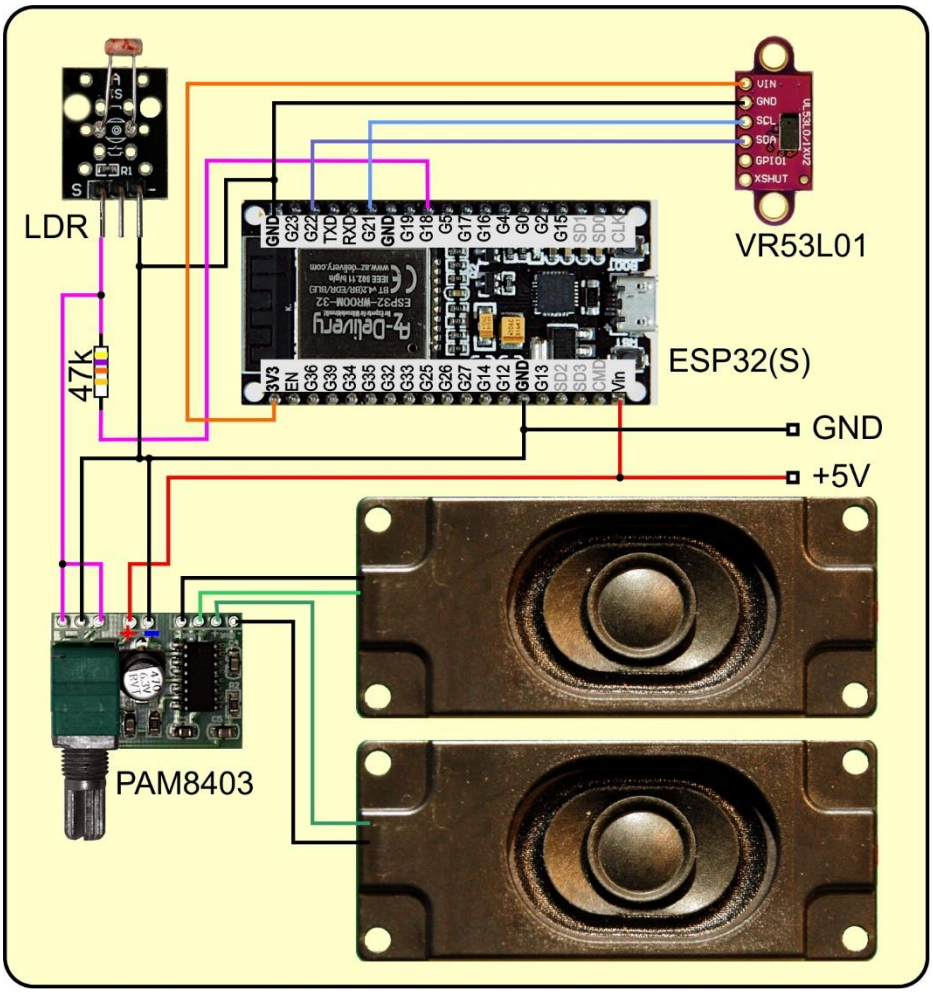

Abbildung 1: Magic Music - Schaltung

Hardware

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	VL53L0X Time-of-Flight (ToF) Laser Abstandssensor
1	PAM8403 digitaler Mini Audio Verstärker 2x 3 Watt DC 5V
1	2 Stück DFplayer Mini 3 Watt 8 Ohm Mini-Lautsprecher
1	KY-018 Foto LDR Widerstand oder Fotowiderstand Photo Resistor Dioden Set 150V 5mm LDR5528
1	Widerstand 47kΩ
diverse	Jumperkabel
1	Minibreadboard oder Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins

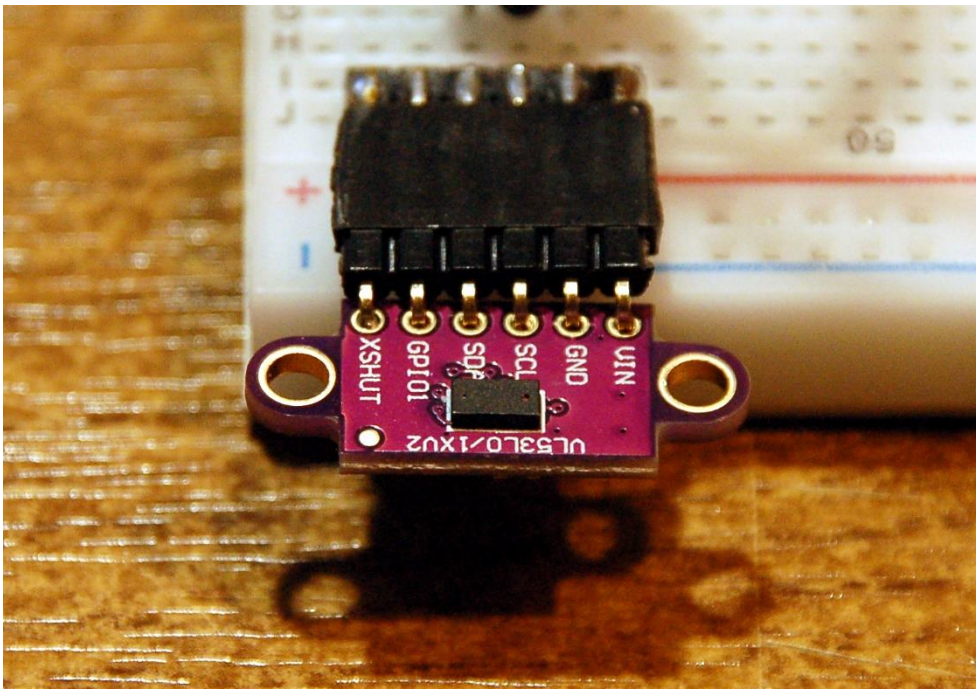

Abbildung 2: VL53L0X - Laser-Entfernungsmesser

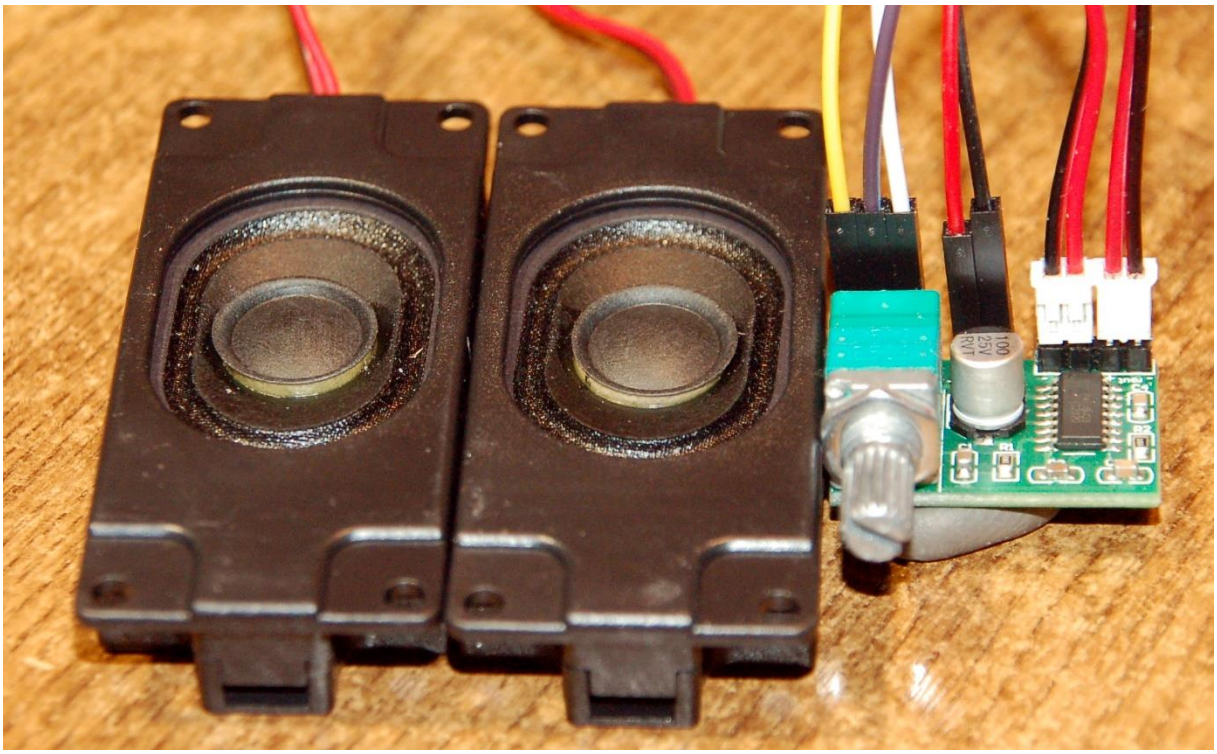
Abbildung 3: Minilautsprecher und Verstärker

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP32:

MicropythonFirmware
Bitte unbedingt die hier angegebene Version flashen, weil sonst die PWM-Steuerung nicht korrekt funktioniert. Bei neueren Versionen ist die minimale PWM-Frequenz 700Hz aufwärts! Warum, das weiß der Geier!
ESP32 mit 4MB Version 1.15 Stand 18.04.2021

Die MicroPython-Programme zum Projekt:

VL53L0X.py modifiziertes Treibermodul für den ToF-Sensor auf ESP32(S) adaptiert
magicmusic.py ausbaufähige Software zum ESP32-Theremin
Original Software auf github für wipy made by pycom
Beispieldateien:
magic1.py
magic2.py
magic3.py
magic4.py
magic5.py

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 Treibermodul für den VR53L01

Das Datenblatt für den VR53L01 liefert nicht, wie üblich, eine Register-Map und eine Beschreibung für die zu setzenden Werte. Stattdessen wird eine API beschrieben, die als Interface zum Baustein dient. Ellenlange Bezeichner in einer bunten Vielfalt prasseln auf den Leser herunter. Das hat meinen Eifer, ein Modul zu schreiben, sehr rasch auf null heruntergeschraubt. Glücklicherweise bin ich, nach ein bisschen Suchen, auf Github fündig geworden. Dort gibt es ein Paket mit einer Readme-Datei, einem Beispiel zur Anwendung und mit dem sehr umfangreichen Modul VL53L0X.py (648 Zeilen). Der erste Start der Beispieldatei **main**.py brachte eine Ernüchterung, obwohl ich das Modul zum ESP32S hochgeladen und die Pins für den ESP32 umgeschrieben hatte.

Das Original

```
from machine import I2C
i2c = I2C(0)
i2c = I2C(0, I2C.MASTER)
i2c = I2C(0, pins=('P10','P9'))
```

wird zu

```
from machine import SoftI2C
i2c = I2C(0,scl=Pin(21),sda=Pin(22))
```

Der MicroPython-Interpreter meckerte ein fehlendes Chrono-Objekt an, im Modul VL53L0X.VL53L0X in Zeile 639. Nachdem das Modul für einen anderen Port geschrieben ist, **wipy devices made by Pycom**, konnte es gut sein, dass noch mehr

derartige Fehler auftauchen würden. Andere Firmware, andere Bezeichner für GPIO-Pins, andere Einbindung von Hardware-Modulen des ESP32…

Nachdem aber außer den I2C-Pins keine weitere Verbindung zwischen ESP32 und VR53L01 bei dem Beispiel gebraucht wird, stufte ich die Wahrscheinlichkeit, weiterer Problemstellen als gering ein. Ich nahm mir also die Methode **perform_single_ref_calibration**() im Modul VL53L0X.VL53L0X im Editor vor. Der Name Chrono deutete auf ein Zeit-Objekt hin.

Und tatsächlich, in der Firmware des WiPy ist die Benutzung der Hardware-Timer anders gelöst wie beim nativen ESP32. Es geht aber im Prinzip nur um die Initialisierung und Überprüfung eines Timeouts.

```
from machine import Timer
...
...
    def perform_single_ref_calibration(self, vhv_init_byte):
        chrono = Timer.Chrono()
        self._register(SYSRANGE_START, 0x01|vhv_init_byte)
        chrono.start()
        while self._register((RESULT_INTERRUPT_STATUS & 0x07)
= 0):
            time_elapsed = chrono.read_ms()
            if time_elapsed > _IO_TIMEOUT:
                return False
        self._register(SYSTEM_INTERRUPT_CLEAR, 0x01)
        self._register(SYSRANGE_START, 0x00)
        return True
```

Dafür habe ich aber meine eigene Softwarelösung. Eine Funktion, Verzeihung, eine Methode, wir bewegen uns ja in der Definition einer Klasse, also eine Methode **TimeOut**(), die eine Zeitdauer in Millisekunden nimmt und die Referenz auf eine Funktion in ihrem Inneren zurückgibt. So ein Konstrukt nennt man Closure. Damit habe ich nun einfach die Zeitsteuerung ersetzt, kürzer aber genauso effektiv.

```
    def TimeOut(self,t):
        start=time.ticks_ms()
        def compare():
            return int(time.ticks_ms()-start) >= t
        return compare
```

```
    def perform_single_ref_calibration(self, vhv_init_byte):
        self._register(SYSRANGE_START, 0x01|vhv_init_byte)
        chrono = self.TimeOut(_IO_TIMEOUT)
        while self._register((RESULT_INTERRUPT_STATUS & 0x07)
== 0):
            if chrono() :
                return False
        self._register(SYSTEM_INTERRUPT_CLEAR, 0x01)
        self._register(SYSRANGE_START, 0x00)
```

```
        return True
```

Außerdem machte sich Erleichterung breit, als sich beim erneuten Start des Demoprogramms keine weitere Fehlermeldung mehr zeigte. Im Gegenteil, im Terminal wurden lauter nette Entfernungswerte in mm ausgegeben.

```
import sys,os
import time
from machine import Pin,PWM
from machine import I2C
import VL53L0X

i2c = I2C(0,scl=Pin(21),sda=Pin(22))

# Create a VL53L0X object
tof = VL53L0X.VL53L0X(i2c)
sound=PWM(Pin(18), freq=20, duty=512)

tof.set_Vcsel_pulse_period(tof.vcsel_period_type[0], 18)

tof.set_Vcsel_pulse_period(tof.vcsel_period_type[1], 14)

tof.start()

while True:
# Start ranging
    tof.read()
    print(tof.read())
```

Die Werte lagen direkt im brauchbaren Hörfrequenzbereich. Ich brauchte also die Werte nur statt der Ausgabe ins Terminal an die PWM-Frequenzeinstellung übergeben.

```
import sys,os
import time
from machine import Pin,PWM
from machine import SoftI2C
import VL53L0X

i2c = SoftI2C(scl=Pin(21),sda=Pin(22))

# Create a VL53L0X object
tof = VL53L0X.VL53L0X(i2c)
sound=PWM(Pin(18), freq=20, duty=512)

tof.set_Vcsel_pulse_period(tof.vcsel_period_type[0], 18)

tof.set_Vcsel_pulse_period(tof.vcsel_period_type[1], 14)

tof.start()
print("started")
```

```
while True:
    sound.freq(tof.read())
```

Haben sie alles aufgebaut? Wohnt das Modul VR53L0X.py schon im Flash des ESP32? Ist der Verstärker eingeschaltet? Dann starten Sie doch einmal **magic1**.py in einem Editorfenster. Und wenn jetzt aus den Lautsprechern ein fröhliches Gedudel erklingt und sich die Tonfrequenz durch heben und senken der Handfläche über dem VR53L01 verändert, dann haben Sie gewonnen.

Die Lautstärke lässt sich auf die gute alte analoge Weise über den Spannungsteiler aus 47kΩ-Widerstand und LDR stufenlos verstellen, indem Sie den LDR abschatten – lauter, oder belichten – leiser. Der Wert des Widerstands sollte so gewählt werden, dass bei normaler Belichtung die Lautstärke nahezu auf Null ist. Der zu wählende Widerstandswert hängt damit von der herrschenden Umgebungshelligkeit ab.

Je nachdem, was man mit den eingelesenen Werten vom VR53L01 anstellt, kann man verschiedene Klangergebnisse erzielen. Im Folgenden stelle ich einfach mal einige Programmschnippsel vor. Sie finden sie in den Beispieldateien magicX.py, mit X in range (1,6), um es im MicroPython-Jargon auszudrücken.

```
tof.start()
f1=500
print("started")

while True:
    f2=(tof.read())
    df=(f2-f1)
    if df != 0:
        s= df//10 if df >= 50 else  df//abs(df)
    if s != 0:
        for f in range(f1,f2,s):
            sound.freq(f)
            time.sleep_ms(10)
    f1=f2
```

```
tof.start()
print("started")
f1=500
f2=(tof.read())
while True:
    f2=(tof.read())
    f=(f2+f1*os.urandom(1)[0]*2)//2
    sound.freq(f)
    f1=f2
```

```
tof.start()
```

```
f1=500
print("started")
f2=(tof.read())
while True:
    f2=(tof.read())*2
    df=(f2-f1)
    s= df//40
    if s != 0:
        for f in range(f1,f2,s):
            sound.freq(f)
            time.sleep_ms(10)
    f1=f2
```

```
tof.start()
f1=500
print("started")
f2=(tof.read())
while True:
    f2=((tof.read())+7*f1)//8
    sound.freq(f2)
    time.sleep_ms(20)
    f1=f2
```

Die Beispiele zeigen, dass der Sound durch folgende Maßnahmen beeinflusst werden kann. Kombinationen daraus sind denkbar

Frequenz f2 und Verarbeitung mit f1

- Rückkopplung
- Addition, Subtraktion
- Multiplikation mit einer Zufallszahl
- …

Einfügen einer for-Schleife für weiche Übergänge

Vorgabe der Spieldauer je Frequenz

Periodische Lautstärkeveränderung durch Fingerfächer oder Flackerlicht

Noch ein Tip zum Schluss:

Offene Zuleitungen zum Verstärker können Störstrahlung aus der Umgebung einfangen, z.B. 50Hz-Brumm. Der kann aber auch von künstlicher Beleuchtung durch Leuchtstofflampen aber auch von Netzteilen von LED-Lampen stammen und über den LDR aufgenommen werden.

Und jetzt, viel Spaß beim Experimentieren!