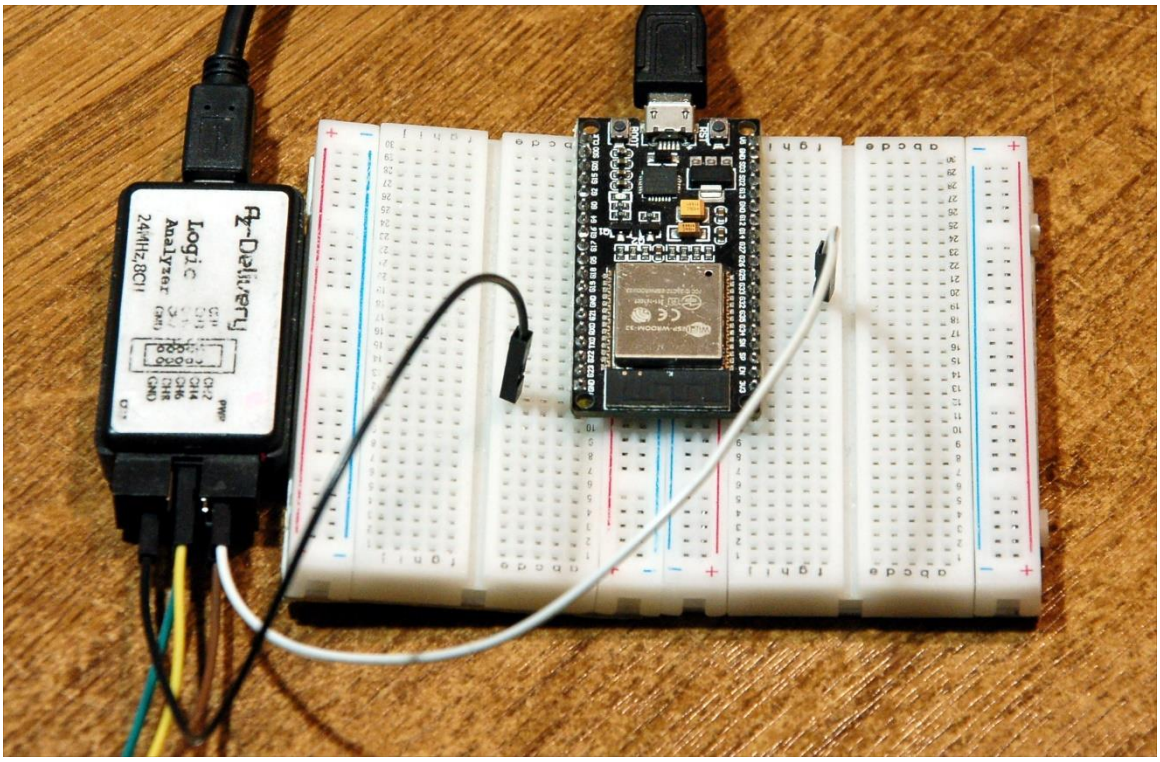

Logic Analyzer am ESP32

Diesen Beitrag gibt es auch als PDF-Dokument.

Beim Theremin wurden Töne über einen PWM-Ausgang erzeugt. Das kann der ESP32 recht gut. Allerdings klingen Rechtecksingale nicht berauschend. Für eine Signalverfolgung in Audioschaltungen mögen sie auch leidlich taugen. Andere Signalformen mit höheren Frequenzen lassen sich aber via PWM nicht erzeugen.

Aber der ESP32 kann ja noch mehr. Schließlich gibt es zwei DAC-Kanäle, die digitale Werte in analoge Spannungen umwandeln. Das sollte doch reichen, um damit zum Beispiel Sinus-, Dreieck- und Sägezahn-Signale zu erzeugen. Den Weg vom ersten Versuch bis zum fertigen Gerät beschreibe ich dieser und den folgenden Beiträgen aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der ESP32 versucht DDS

DDS ist das Akronym für Direkte Digitale Synthese. Und die könnte im einfachsten Fall dadurch erreicht werden, dass einzelne, aufeinander folgende Signalpegel so berechnet werden, dass im zeitlichen Verlauf eine bestimmte Signalform herauskommt. Diese Zahlenwerte müssen natürlich in eine Folge von Spannungswerten übersetzt werden, das erledigt ein DAC (Digital-Analog-Converter).

Eines wollen wir aber gleich zu Beginn festhalten: Ein echtes Analogsignal wird auf diese Weise nicht herauskommen, denn der DAC akzeptiert nur Ganzzahlen und die auch nur im Bereich von 0 bis 255. Dem entsprechend sind auch nur 256 verschiedene Spannungswerte zwischen 0 und 3,3V möglich. Das entspricht einem Zuwachs von 3,3V / 256 = 0,0129V als LSB (Least Significant Bit hier: kleinster Zuwachs für eine Zählstufe). Trotzdem lassen sich damit brauchbare Signalformen erzeugen, vor allem, wenn man zum Glätten der Stufen ein Tiefpassfilter einsetzt.

Die Hardware

1	ESP32 Dev Kit C unverlötet oder ESP32 Dev Kit C V4 unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder NodeMCU-ESP-32S-Kit oder ESP32 Lolin LOLIN32 WiFi Bluetooth Dev Kit
1	Breadboard Steckbrett mit 400 Kontakten 3-er Set
1	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F
1	Logic Analyzer

Mit Ausnahme des NodeMCU-ESP-32S-Kit müssen zwei Breadboards an den Längsseiten über zwei Stromschienen zusammengesteckt werden, damit der Controller mit allen Beinchen Platz hat und außerdem auch noch freie Steckplätze für die Jumperkabel bleiben.

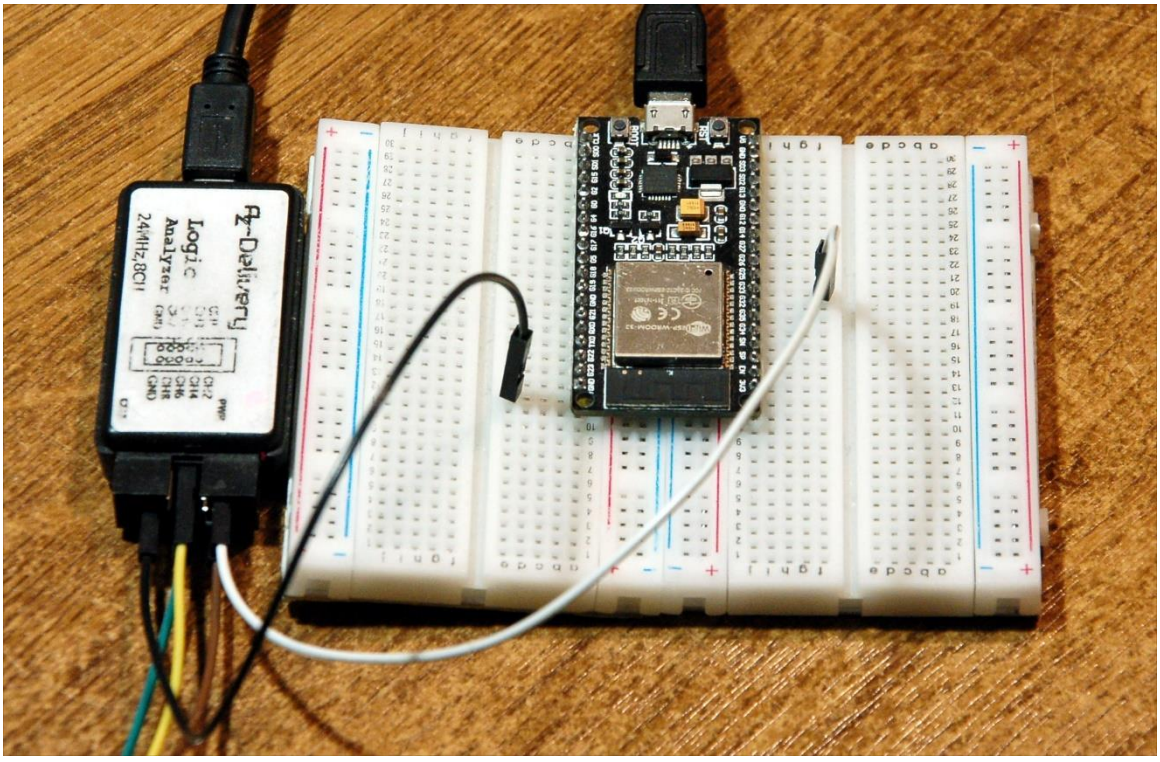

Abbildung 1: Logic Analyzer am ESP32

Zur Vermessung der erzeugten Frequenzen habe ich einen Logic Analyzer eingesetzt. Mit der kostenlosen Software Logic2 von Saleae kann man sehr genau die Periodendauer von Rechtecksignalen bestimmen. Durch Kehrwertbildung erhält man aus der Periodendauer die Frequenz und umgekehrt.

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft

Signalverfolgung:

Saleae Logic 2

Verwendete Firmware für einen ESP32:

MicropythonFirmware Download
v1.19.1 (2022-06-18) .bin

Die MicroPython-Programme zum Projekt:

dac.py erstes Testprogramm
build_tables.py Tabellen-Generator für die DDSs-Werte
dds_generator.py Demoprogramm mit Listen der DDSs-Werte
dds_generator_irq.py Betriebsprogramm
funktionstabellen.py Modul mit Ausgabewerten für den DAC
build_tables.py Programm zum Erzeugen der Funktionstabellen

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 18.06.2022) auf den ESP-Chip gebrannt wird. Wie Sie den **Raspberry Pi Pico** einsatzbereit kriegen, finden Sie hier.

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.

Ein erster Versuch

Wenn nur 256 mögliche Spannungswerte für eine Periode möglich sind, reichen auch 256 Phasenpunkte für eine Periodendauer des Signals Im Fall eines Sägezahnsignals sieht das so aus, dass wir jedem der 256 Phasenpunkte seine Nummer als Wert zuweisen. Die Funktion **sawtooth**() macht genau das.

```
def sawtooth(i):
    return i
```

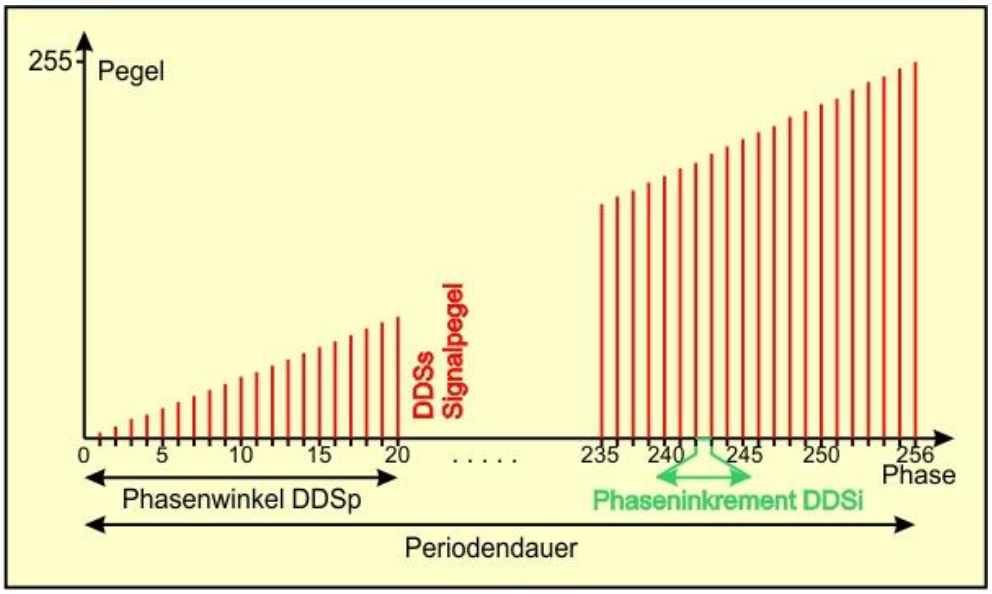

Abbildung 2: Entstehung eines Sägezahnsignals

Ähnlich können wir ein Dreiecksignal erzeugen, wenn wir für die erste Hälfte der Periode die Ausgabewerte linear ansteigen und in der zweiten Hälfte linear abfallen lassen.

```
def triangle(i):
    return i*2 if i <=127 else 511-i*2
```

Ein Rechtecksignal entsteht, wenn für die Hälfte der Periodendauer der DAC-Wert auf 0 und für den Rest der Zeit auf 255 gesetzt wird.

```
def square(i):
    return 0 if i <= 127 else 255
```

Für einen Sinusverlauf benutzen wir die Sinusfunktion aus dem Modul **math**. Dabei müssen wir berücksichtigen, dass MicroPython für Winkelgrößen das Bogenmaß verwendet. Um das Inkrement für die Winkelargumente zu erhalten, teilen wir den Vollwinkel 2π (= 360°) in 256 Teile und erhalten 0,02454. Diesen Wert multiplizieren wir mit den Phasenpunkten 0 bis 255 und geben das Resultat an die sin-Funktion weiter, die uns Werte zwischen -1 und 1 liefert. Wir multiplizieren mit 127 und heben die Nulllinie auf 128 an.

```
def sine(i):
    return int((128+127*sin(dPhi*i))+0.5) # aufrunden
```

Wir bauen die vier Funktionen, deren Rückgabewert der Signalpegel DDSs ist, nun in ein erstes Testprogrämmchen ein. Als Argument übergeben wir die Nummer der Phase.

Nach dem Importgeschäft berechnen wir das Winkelinkrement für die Sinusfunktion. Als DAC-Ausgang nehmen wir GPIO25 den wir dem Konstruktor des DAC-Objekts übergeben.

```
# dac.py
#
from math import sin, pi
from machine import Pin, DAC
from time import sleep_us
from sys import exit

dPhi=2*pi/255
dacPin=25
dac=DAC(Pin(dacPin))

def sine(i):
    return int((128+127*sin(dPhi*i))+0.5)

def square(i):
    return 0 if i <= 127 else 255

def sawtooth(i):
    return i
```

```
def triangle(i):
    return i*2 if i <=127 else 511-i*2

while 1:
    for phase in range(256): #DDSi = 1
        dac.write(square(phase))
```

Die Hauptschleife muss nichts anderes tun, als endlos die Phasennummern durchzuzählen und an die ausgewählte Funktion weiterzugeben. Den berechneten Pegelwert schicken wir an den DAC-Wandler.

Am DSO (digitales Speicher Oszilloskop) erscheint das Rechtecksignal mit einer Frequenz von 143Hz. Auch der Logic Analyzer liefert ähnliche Werte.

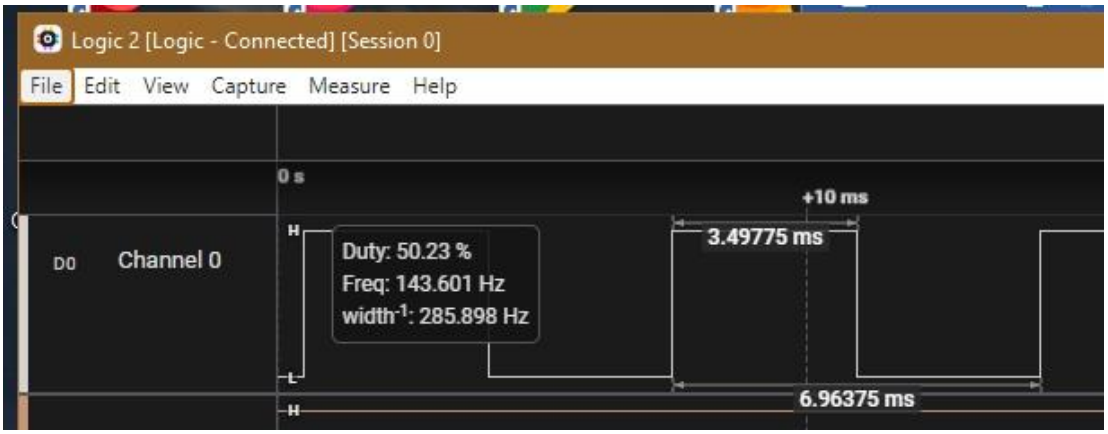

Abbildung 3: Das Rechecksignal bringt es auf 143 Hz

Beim Sägezahn bekommen wir 153Hz, beim Dreieck 136Hz und beim Sinus grade mal um die 60 Hz. – Nicht gerade berauschend, daran müssen wir arbeiten. Dazu kommt auch noch die schadhafte Wellenform des Sinussignals, das wir so nicht lassen können. Die anderen drei Signale waren OK.

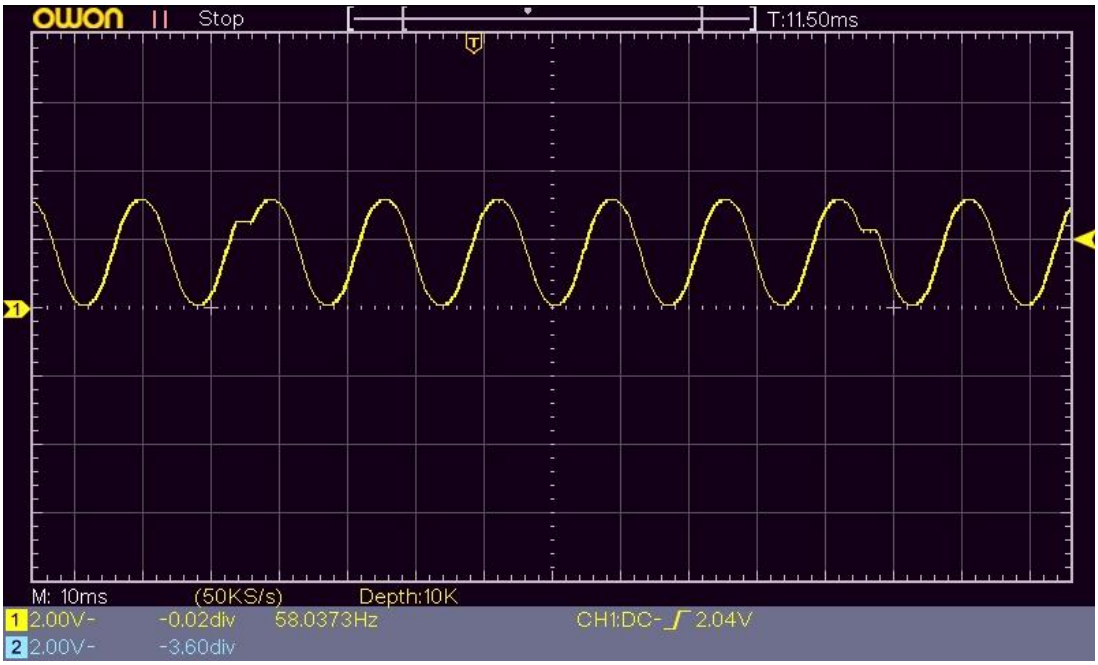

Abbildung 4: Sinuskurve mit Störungen

Die unterschiedlichen Frequenzwerte resultieren erst einmal aus der unterschiedlichen Laufzeit der Signalpegelberechnung in den Funktionen. Bei der aufwendigen Berechnung der Sinuswerte funken wohl zusätzlich die Hintergrundprozesse ESP32 mit dazwischen. Wenn gerade eine Unterbrechungsanforderung (IRQ=Interrupt Request) eintrifft, unterbricht die ISR (Interrupt Service Routine) den Hauptprogrammlauf, was dazu führt, dass der in diesem Moment eingestellte Signalpegel länger als zuträglich gehalten wird. Das passiert offenbar circa alle 90ms.

Das Zuweisen der Signalwerte an den DAC muss also beim Sinus schneller gehen und vor allem für alle Wellenformen gleich schnell. Wo ist denn eben mal der Zauberstab? Hokus – Pokus – Wir brauchen eine Lösung! Und die lautet: vorausberechnen der Werte. In der Hauptschleife müssen die einfach nur noch abgerufen werden. Pro Wellenform stellen wir einen Satz von 256 Werten her und legen diese in einer Liste ab. Wenn wir die vier Blöcke dann auch noch in eine Wellenformdatei schreiben, können wir diese als Modul importieren. Natürlich berechnen wir die Werte nicht selbst und wir tippen sie auch nicht in die Datei ein. Das lassen wir mal schön dem ESP32 erledigen. Wir müssen ihm nur sagen, wie er das machen soll. Schauen wir uns am Beispiel Sinus an, wie so ein Werteblock aussehen muss. Eingeleitet wird er durch die Definition einer Liste. Dann folgen 8-er-Gruppen mit den Signalwerten.

```
Sinus= [
128, 131, 134, 137, 140, 144, 147, 150,
153, 156, 159, 162, 165, 168, 171, 174,
177, 180, 182, 185, 188, 191, 194, 196,
199, 201, 204, 206, 209, 211, 214, 216,
218, 220, 222, 224, 226, 228, 230, 232,
234, 236, 237, 239, 240, 242, 243, 244,
246, 247, 248, 249, 250, 251, 251, 252,
253, 253, 254, 254, 254, 255, 255, 255,
255, 255, 255, 255, 254, 254, 253, 253,
252, 252, 251, 250, 249, 248, 247, 246,
245, 244, 242, 241, 240, 238, 236, 235,
233, 231, 229, 227, 225, 223, 221, 219,
217, 215, 212, 210, 208, 205, 203, 200,
197, 195, 192, 189, 187, 184, 181, 178,
175, 172, 169, 167, 164, 160, 157, 154,
151, 148, 145, 142, 139, 136, 133, 130,
126, 123, 120, 117, 114, 111, 108, 105,
102, 99, 96, 92, 89, 87, 84, 81,
78, 75, 72, 69, 67, 64, 61, 59,
56, 53, 51, 48, 46, 44, 41, 39,
37, 35, 33, 31, 29, 27, 25, 23,
21, 20, 18, 16, 15, 14, 12, 11,
10, 9, 8, 7, 6, 5, 4, 4,
3, 3, 2, 2, 1, 1, 1, 1,
1, 1, 1, 2, 2, 2, 3, 3,
4, 5, 5, 6, 7, 8, 9, 10,
12, 13, 14, 16, 17, 19, 20, 22,
24, 26, 28, 30, 32, 34, 36, 38,
40, 42, 45, 47, 50, 52, 55, 57,
```

```
60, 62, 65, 68, 71, 74, 76, 79,
82, 85, 88, 91, 94, 97, 100, 103,
106, 109, 112, 116, 119, 122, 125, 128,
]
```

Zum Schluss schließen wir die Listenklammer.

Das Programm **build_tables.py** erledigt den Auftrag für die vier Wellenformen in etwa einer Sekunde. Das entstehende Modul **funktionstabellen.py** wird auch gleich in den Flash des Controllers geschrieben, wir müssen sie also nicht extra dorthin verfrachten.

```
# build_tables.py
#
# Wertetabellen fuer DDS generieren
#
from math import sin, pi

dPhi=2*pi/255
def sinus(i):
    return int((128+127*sin(dPhi*i))+0.5)

def square(i):
    return 0 if i <= 127 else 255

def sawtooth(i):
    return i

def triangle(i):
    return i*2 if i <=127 else 511-i*2
```

Bis hierher kennen wir die Sache schon. Jetzt erzeuge ich zwei Listen. **functions** enthält die Bezeichner der vier Funktionen, damit ich sie in einer Schleife automatisch referenzieren kann. **functions_text** beinhaltet die Bezeichner der zu erzeugenden Listen.

Die **with**-Anweisung erzeugt ein Handle **f** auf die Datei **funktionstabellen.py** zum (Über-)Schreiben. Die erste der drei **for**-Schleifen durchläuft über den Index **fi** die Liste der Funktionsblöcke. Die Ausgabe erfolgt über das Handle **f** und ebenfalls in REPL zur direkten Kontrolle. Als Erstes wird die Listendefinition geschrieben, dann holen wir die Referenz auf die Funktion in die Variable **func**.

```
functions=[sinus, square, sawtooth, triangle]
functions_text=["Sinus", "Square", "Sawtooth", "Triangle"]

with open("funktionstabellen.py","w") as f:

    for fi in range(len(functions)):
        print(functions_text[fi],"= [")
        f.write(functions_text[fi]+"= [\n")
        func=functions[fi]
```

```
        for i in range(32):
            for j in range(8):
                k= i*8 + j
                print(func(k),", ",end="")
                f.write(str(func(k))+", ")
            f.write("\n")
            print()
        f.write("]\n\n")
        print("]\n")
```

Um 256 Werte in 8-er-Gruppen abzulegen brauchen wir 32 Zeilen. Die erzeugen wir durch die **i-Schleife**. Die **j-Schleife** schreibt die einzelnen Zeilen. Aus i und j berechnen wir die Phasennummer k, die wir an die Funktion in **func**() übergeben. Zurück kommt eine Ganzzahl, die wir in REPL mit einem nachfolgenden Komma ausgeben. Damit das in einer Zeile erfolgt, geben wir als Zeilenende-Zeichen den leeren String **""** an. Für die Ausgabe in die Datei muss die Ganzzahl in einen String umgebaut werden. Nach dem achten Wert muss ein Zeilenende ausgegeben werden. In die Datei schreiben wir dafür ein **New Line = "\n",** in REPL erledigt das die Funktion **print**(). Am Blockende schließen wir die Liste und geben als Trenner zwei Leerzeilen aus. Damit die neue Datei im Flash in der Dateiliste angezeigt wird, klicken wir in Thonny auf die drei Linien (roter Rahmen) und dann auf Refresh.

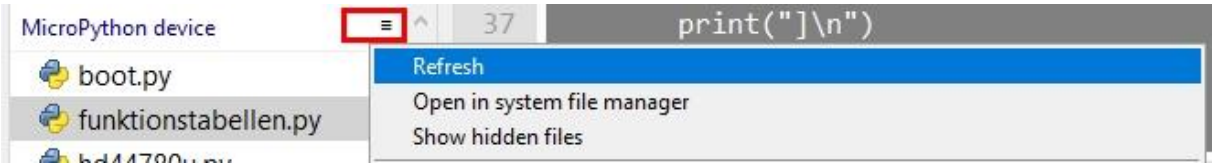

Abbildung 5: Neue Datei anzeigen lassen

Dann schauen wir doch mal, was die Aktion an Geschwindigkeitsgewinn gebracht hat. Anstatt die Samplewerte DDSs zu berechnen, lesen wir sie jetzt einfach aus den importierten Listen, das strafft den Umfang des Listings. Die auszugebende Wellenform legen wir durch den Index **wForm** in die Liste **functions_values** fest.

```
# dds_generator.py
#
from machine import Pin, DAC
from funktionstabellen import *
from time import sleep_us
from sys import exit
#
dac=DAC(Pin(25))
functions_values=[Sinus, Square, Sawtooth, Triangle]
functions_text=["Sinus", "Square", "Sawtooth", "Triangle"]
wForm=0
func=functions_values[wForm]

while 1:
    for val in func:
        dac.write(val)
```

In der Hauptschleife müssen jetzt nur noch die Werte aus der jeweiligen Liste geholt und über den DAC ausgegeben werden. Der Gewinn an Geschwindigkeit geht beim Sinus an die 500%, bei den anderen Wellenformen liegt er auch bei gut 200%. Was nicht verwundert, ist die Tatsache, dass jetzt für alle Wellenformen dieselbe Ausgabefrequenz erreicht wird, 304Hz. Allerdings ist bei allen Wellenformen ein leichter Jitter am DSO zu verzeichnen, der immer noch den Hintergrundaktionen des ESP32 zu verdanken ist, die Frequenz schwankt also immer noch um einen Mittelwert. Das wird sich auf dem ESP32 auch nicht abstellen lassen. Aber wir haben noch eine Trumpfkarte in der Hinterhand. Immerhin ist die Signalform jetzt für alle Wellenformen astrein.

Das nächste zu lösende Problem ist die Vorgabe und Einhaltung einer bestimmten Frequenz. Wie können wir niedrigere und vor allem höhere Frequenzen mit unserem Generator erreichen. Nun das DDS-Prinzip ist mit der bisherigen Programmierung noch nicht ganz erreicht – warum?

Niedrigere Frequenzen könnten wir erreichen, wenn wir in die Hauptschleife kurze Schlafzyklen der folgenden Art einbauen würden.

```
while 1:
    for val in func:
        dac.write(val)
        sleep_us(1)
```

Jede µs erhöht die Durchlaufzeit der Hauptschleife und damit steigt die Periodendauer des Ausgangssignals um einen noch nicht wirklich bekannten absoluten Wert. Hier helfen nur Messungen der Periodendauer mit dem Logic Analyzer oder dem DSO weiter. Ich habe daher die Periodendauer für einige Sleep-Werte mit dem Logic Analyzer bestimmt, weil er präzisere Werte. Für genaue Messwerte brauchen wir die steilen Flanken des Rechtecksignals.

Periodendauer gegen Sleep-Wert

Sleepwert in µs	-	0	1	5	10	20	45
Periodendauer in ms	3,2835	4,16	4,191	4,644	5,682	8,24475	14,71275
	3,29325	4,17	4,201	4,63675	5,6815	8,2465	14,71375
	3,29425	4,16025	4,1905	4,6525	5,6805	8,265	14,6945
Periode Mittelwert	3,2903	4,1634	4,1942	4,6444	5,6813	8,2521	14,7070
Zuwachs		0,873	0,031	0,450	1,037	2,571	6,455
Zuwachs / µs			0,031	0,113	0,207	0,257	0,258
Frequenz	303,9	240,2	238,4	215,3	176,0	121,2	68,0
Abnahme / µs				4,62	7,86	5,48	2,13

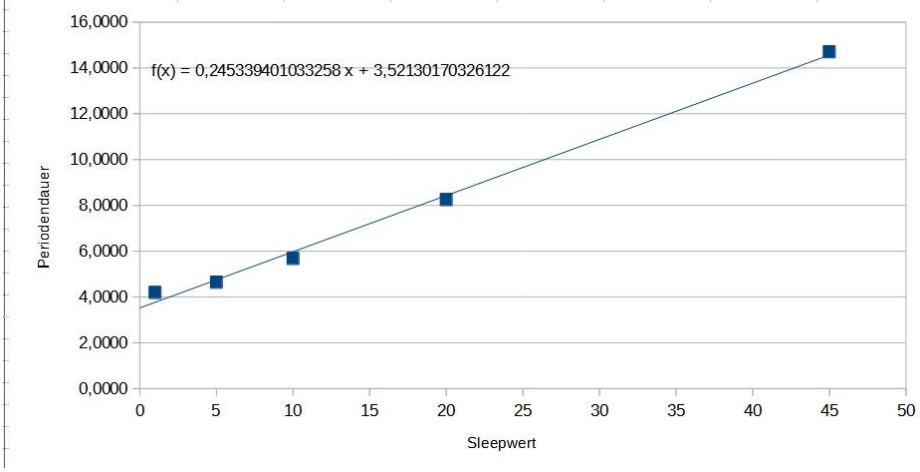

Abbildung 6: Frequenzänderung durch die Sleep-Funktion - suboptimal!

Ohne den Sleep-Befehl haben wir eine Schleifen-Laufzeit und damit eine Peroidendauer von 3,29ms. Alleine das Einfügen der sleep_us-Anweisung ohne Verzögerung, **sleep_us(0)**, bringt eine Verlängerung um 0,87ms. Aber auch beim Schritt von 0µs auf 1µs nimmt die Periodendauer nicht um erwartete 256µs sondern um 31µs zu. Dieser Wert steigert sich bis 258µs / µs und wird auch erst bei 20µs aufwärts relativ konstant. Die Frequenzänderung passiert, besonders am Anfang, in großen Sprüngen – das ist also keine Lösung. Aber es ist auch immer noch nicht wirklich DDS.

Immerhin kommen wir jetzt schon auf ca. 300Hz mit einer Periodendauer von 3,29ms. Der nächste Ansatz nähert sich DDS und verwendet einen Hardwaretimer des ESP32, um die 256 Signalwerte der Tabelle auszugeben. Das Timing ist damit fix und sollte Berechnungen zuverlässig(er) machen. Wie können wir aber mit einem festen Timing variable Frequenzen erhalten?

Die Frequenz wäre besser einstellbar, wenn wir mehr Phasenstufen hätten. Wie wäre es mit 65536 oder 1677216? Dann bräuchten wir aber auch ebenso viele Signalwerte in den Listen, meinen Sie? Nicht unbedingt! Bleiben wir bei Werten von 0 bis 65535 = 0xFFFF. Wir können doch jetzt den DDSp-Wert um beliebige Inkremente DDSd aus diesem Wertebereich erhöhen. Damit erreichen wir mehr oder weniger schnell den Maximalwert 0xFFFF oder überschreiten ihn. Dann beginnt die Zählung automatisch wieder von vorne. Addieren wir nur ein kleines Inkrement, dann zögern wir den Overflow des Phasenzeigers hinaus, die Frequenz sinkt.

Wenn wir jetzt als Index in die Tabelle der Signalwerte aber nur das MSB (Most Significant Byte = Höher wertiges Byte) des Phasenzeigers benutzen, können wir die Frequenz in kleinen Stufen ändern, ohne mehr als 256 Signalwerte zu benötigen.

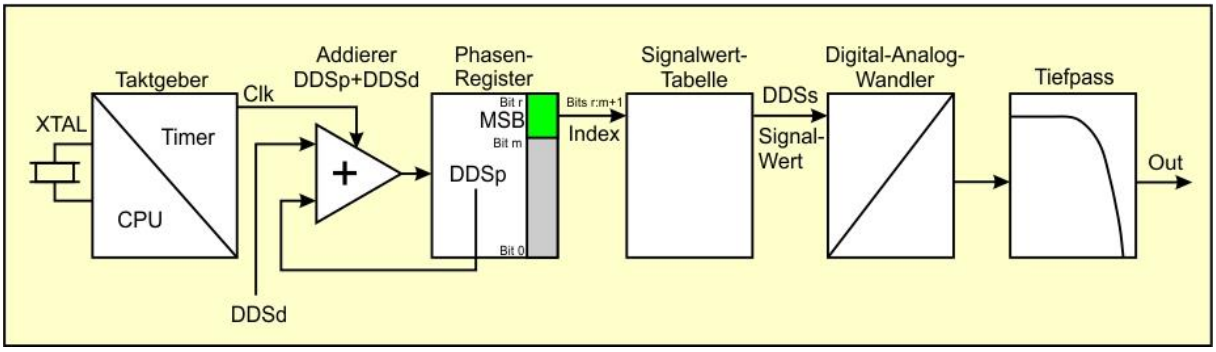

Abbildung 7: Schema eines DDS-Systems

Wie die zeitliche Abfolge von Hauptprogramm und ISR (Interrupt Service Routine) des Timers ist, erfahren wir durch einen Trick vom Programm selbst. Die Hauptschleife ist jetzt eigentlich arbeitslos. Sie hat nur den Pin **loop**=Pin(12) ständig ein- und auszuschalten. Der Kanal 1 des DSO zeigt uns, wann die Main Loop aktiv ist. Beim Betreten der **timerISR**() lassen wir den Pin **irq**=Pin(12) einschalten und nach getaner Arbeit beim Verlassen wieder ausschalten (Kanal 2). Während die ISR das Kommando hat, schweigt das Umschalten der Hauptschleife. Das sieht dann folgendermaßen aus.

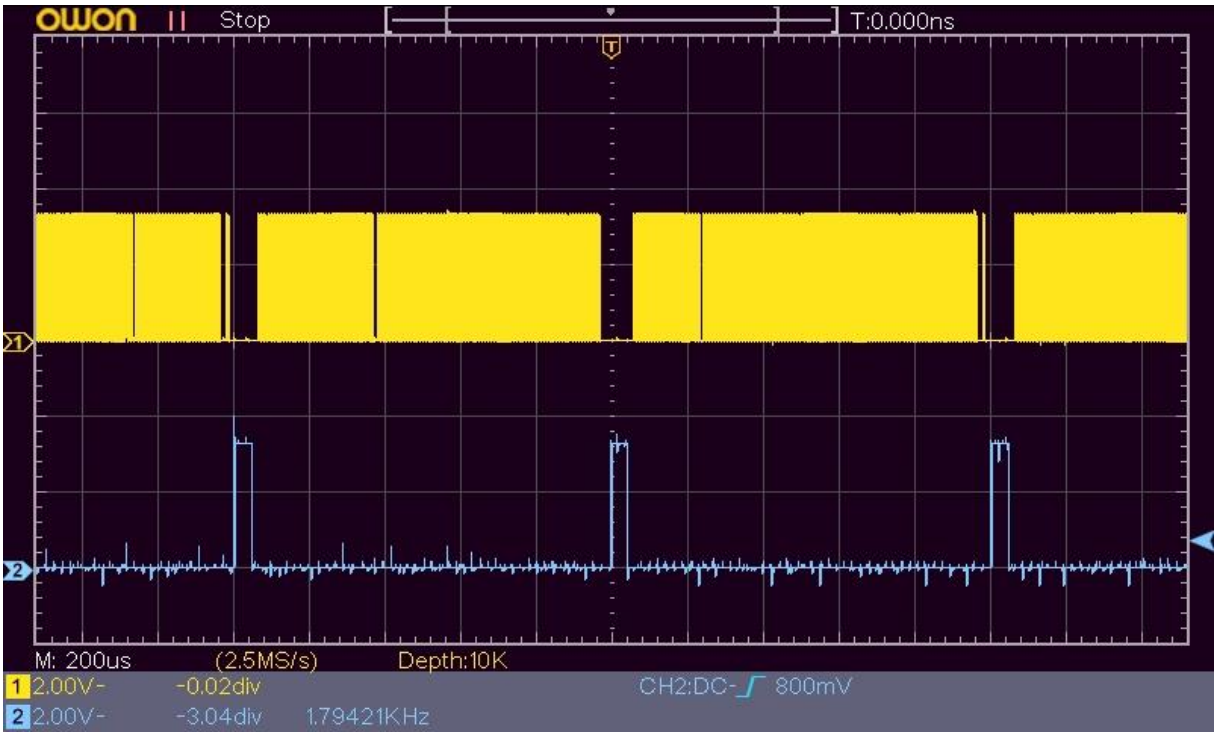

Abbildung 8: IRQ-Timing

Wir erkennen sehr gut die im Programm vorgegebene Folgezeit des Timer-Interrupts von 1ms. Dazwischen gibt es unzählige Impulse von der Hauptschleife. Betrachten wir doch das Umfeld eines Interrupts einmal genauer.

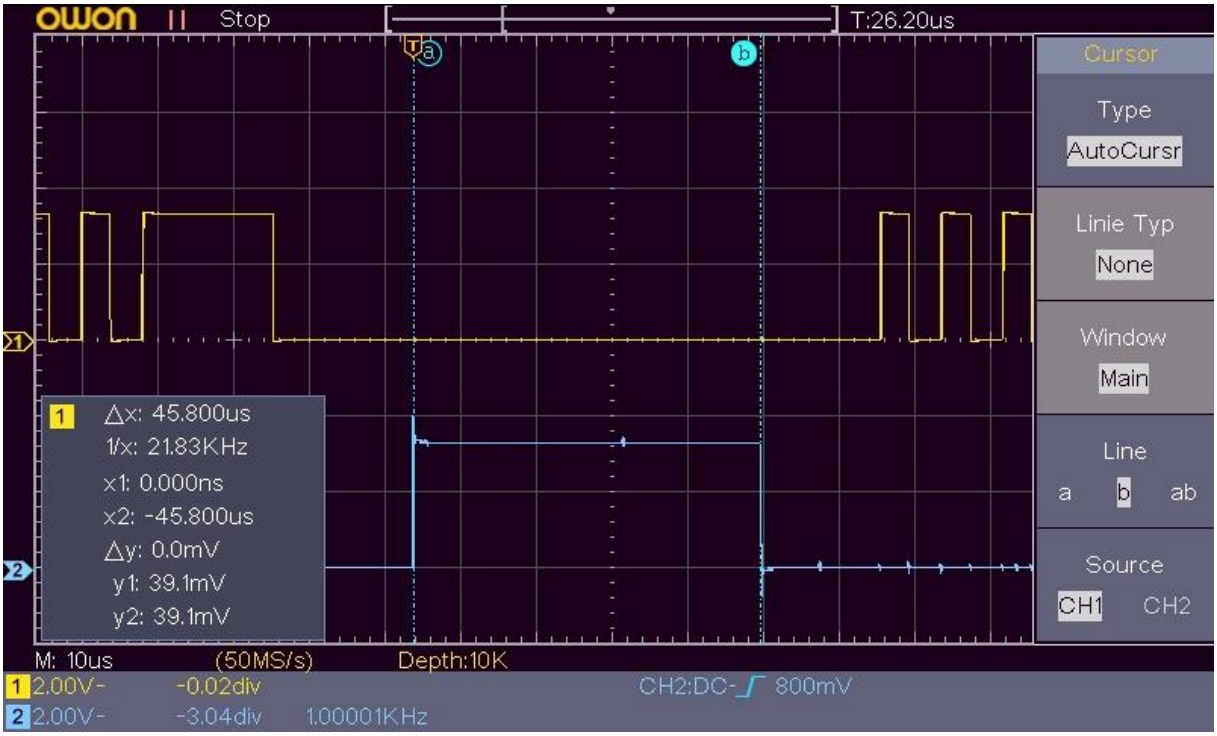

Abbildung 9: IRQ-Timing Detail

Nach der letzten Flanke des Main-Loop-Takts vergehen 18,4µs bis sich die ISR mit der steigenden Flanke meldet. Die Operationen der ISR brauchen 45,8µs. Erst 16µs nach dem Ende der ISR setzt der Takt der Hauptschleife wieder ein. Ein IRQ-Event nimmt also insgesamt ca. 70µs in Anspruch. Wenn wir eine IRQ-Folgezeit von 80µs

(mit Puffer zu 70µs) ansetzen könnten, kämen wir auf eine Ausgabefrequenz unseres DDS-Signals von grade mal 48Hz. Das Ausgabesignal der vorliegenden Anordnung liegt vor allem wegen der langen IRQ-Folgezeiten von 1ms bei weniger als 4Hz – ja richtig gelesen vier Hertz! Operatives Hecheln – dann war das alles für die Katz? **Ja** – mit dem ESP32 alleine wohl schon. **Aber** es muss eine Lösung geben, denn sonst hätte ich diesen Artikel nicht geschrieben.

Es gibt sogar zwei Lösungsansätze. Wir könnten einen DDS-Chip kaufen oder - selbst einen programmieren. Ich habe mich für letzteres entschieden, weil das kostengünstiger und vor allem für einen Elektroniker spannender und interessanter ist. Die Grundlagen von DDS haben wir ja gerade erarbeitet. Was wir brauchen ist eine drastische Geschwindigkeitssteigerung. Die Erkenntnis, dass der ESP32 alleine unter Verwendung von MicroPython nicht unmittelbar für diesen Zweck geeignet ist, führt noch lange nicht zum Abbruch des Projekts. Angepeilt ist eine Arbeitsteilung. Der ESP32 wird die Steuerungsrolle bekommen und ein anderer Microcontroller wird die Signale erzeugen. Wie geht es also weiter?

Was wir brauchen ist ein Microcontroller, der unter Assembler maschinennah programmiert werden kann. Wir erreichen damit zwei Verbesserungen:

- Assembler ist eine maschinennahe Programmierung und liefert sehr schnelle Programme. Durch die Auswahl der verwendeten Befehle und Strukturen lässt sich das Programm auf Geschwindigkeit optimieren. Assembler ist in diesem Fall auch einer Programmierung in C vorzuziehen.
- Wer die CPU wann unterbricht, entscheiden nur wir und nicht irgend ein Betriebssystem. Dadurch erhalten wir perfekte Signalkurven.

Ich habe da in der Bastelkiste gekramt und einen ATMEL ATTiny2313 gefunden. Der wird zwar nur mit 16 oder maximal 20MHz getaktet, liefert aber DDS-Signale bis 20kHz ab und, das sei schon mal verraten, er hat neben dem Programm selbst noch Platz für bis zu 6 Wellenformtabellen, trotz der nur 2 kByte Programmspeicher.

Das Programm wird in AVR-Assembler geschrieben. Dabei wird uns das kostenlose ATMEL-Studio 7.0 helfen. Mit dem ebenfalls kostenlosen Tool AVRDUDE werden wir das Programm auf dem Tiny 2313 einquartieren. Als Programmer habe ich einen MK II von ATMEL zur Verfügung. Man kann aber ohne weiteres auch einen herumliegenden Arduino für diesen Zweck einrichten.

Mit diesen Themen werden wir uns in den nächsten Folgen näher beschäftigen. Bleiben Sie also dran!