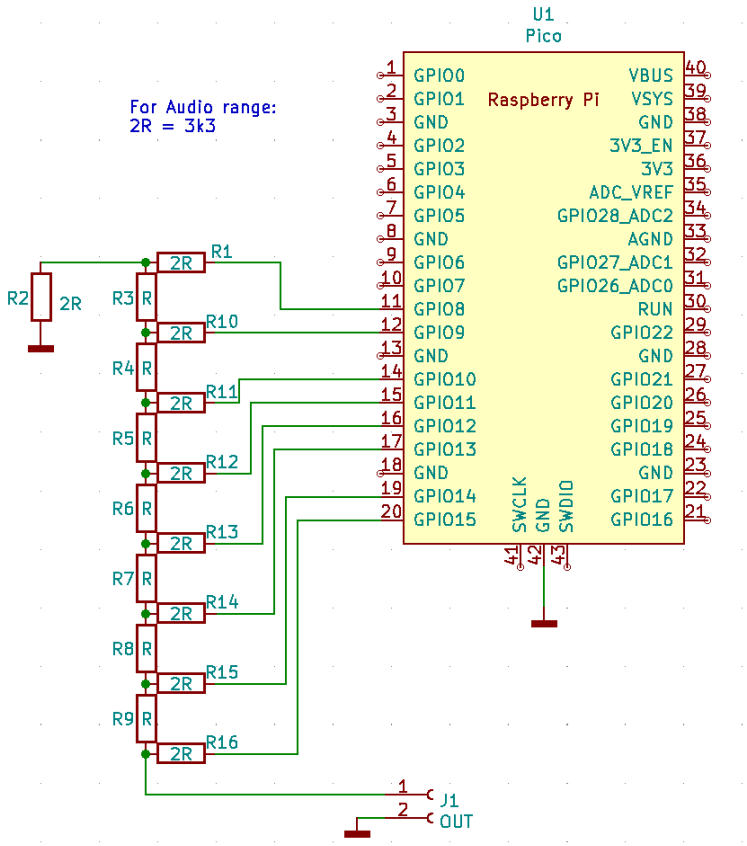
DDS Generator mit PIO und DMA

jean-claude.feltes@education.lu

Eine weitere Etappe auf meiner Reise durch die Möglichkeiten der PIO-Programmierung des Raspi Pico. Den Beginn schildere ich hier:
https://staff.ltam.lu/feljc/electronics/uPython/PIO_Programmierung.pdf

1. Prinzipschaltung



Der Ausgang sollte mit einem Impedanzwandler gepuffert werden.
Naturgemäss variiert die Ausgangsspannung zwischen 0 und 3.3V, für eine reine Wechselspannung müsste ein Subtrahierer (-3.3V), eventuell mit einem Verstärker kombiniert, nachgeschaltet werden.

Der DAC ist hier diskret aufgebaut, es kann natürlich auch ein fertiges IC wie der ZN426 mit parallelem Eingang verwendet werden.

2. Software

Hierzu gibt es einige sehr gute Ideen die auf Ansätzen von Rolf Odemann und H.J. Berndt basieren.

https://github.com/TFillary/Waveform-Generator

https://www.instructables.com/Arbitrary-Wave-Generator-With-the-Raspberry-Pi-Pic/
https://magpi.raspberrypi.com/articles/arbitrary-waveform-generatorh
https://hackaday.com/2023/11/29/arbitrary-wave-generator-for-the-raspberry-pi-pico/
https://github.com/favict/pico-waveform-generator
https://www.instructables.com/Poor-Mans-Waveform-Generator-Based-on-RP2040-Raspb/

Um einen breiten Frequenzbereich zu überstreichen wird bei diesen die Anzahl der Samplingpunkte dynamisch verändert nachdem die Frequenz festgelegt wurde.

Der Vorteil ist, dass die Frequenz bis weit in den MHz-Bereich getrieben werden kann (solange der DAC dies mitmacht).

Der Nachteil ist, dass die Samples erst beim Festlegen der Frequenz berechnet werden, was bei 4096 Samples ca. 500ms braucht. Dies schliesst eine musikalische Anwendung praktisch aus.

Ich habe mich eine ganze Weile mit dem Code von Odemann beschäftigt und vieles davon verstanden, ohne aber durchzublicken was die Idee hinter dem Algorithmus für die Anzahl der Samples ist.

Da es mir hier mehr um das Verständnis von PIO und DMA geht, habe ich mich schlussendlich dazu durchgerungen, einen einfacheren Code zu schreiben, auch wenn er dann nur im Audiobereich genutzt werden kann. (Prinzipiell wären auch höhere Frequenzen möglich, wenn man wie bei Odemann mehrere Perioden in den Sample Buffer legt).

Die Grundidee ist dabei, eine feste Anzahl von Samples zu benutzen und die Kurve(n) (Sinus, Sägezahn oder was auch immer) beim Programmstart zu berechnen und dann per DMA aus einem Array zu lesen und per PIO auszugeben.

Ein Vorteil dieser Idee ist es, dass nun die Frequenz sehr schnell geändert werden kann, was eine musikalische Anwendung möglich macht.

3. DDS simpel

Die Grundidee ist folgende:

- Samples beim Start berechnen
- Eine PIO State Machine gibt die von DMA-Kanal 0 gelieferten Samples parallel aus, mit einer Frequenz $f_{SM} = f \cdot N$ wobei N die Anzahl der Samples ist, z.B. N = 4096
- Ein zweiter DMA-Kanal ist mit dem ersten verkettet und startet diesen wieder neu am Ende jeder Periode, so dass ein kontinierlicher Datenfluss entsteht.

Ich habe versucht die DMA-Konfiguration etwas “pythonischer” statt über Registerwerte zu machen, damit sie leichter verständlich ist. Leider ging das nicht für alles, deswegen habe ich die Konstanten erst einmal am Programmanfang stehen lassen.

a) Importe und Konstanten:

```
from machine import Pin,mem32
from rp2 import PIO, StateMachine, asm_pio, DMA
from array import array
from math import pi,sin
from uctypes import addressof
import time

PIO0_TXF0       = 0x50200010
PIO0_SM0_CLKDIV = 0x502000c8

DMA0_Read  = 0x50000000; DMA1_Read  = 0x50000040
DMA0_Write = 0x50000004; DMA1_Write = 0x50000044
DMA0_Count = 0x50000008; DMA1_Count = 0x50000048
DMA0_Trig  = 0x5000000C; DMA1_Trig  = 0x5000004C
DMA0_Ctrl  = 0x50000010; DMA1_Ctrl  = 0x50000050
```

Die Registeradressen sind im Datenblatt zu finden.

Wichtige Variablen sind als nächstes N, die Anzahl der Samples und der Datenpuffer buffer als Bytearray:

```
N=4096
buffer = bytearray(N)
```

b) Parallelausgabe

Anschliessend wird die PIO State Machine definiert, die für das parallele Ausgeben der Daten zuständig ist:

```
@asm_pio(out_init=(PIO.OUT_HIGH,)*8, out_shiftdir=PIO.SHIFT_RIGHT,
         autopull=True, pull_thresh=32)
def parallel():
    out(pins,8)

sm=StateMachine(0, parallel, freq= 100_000_000, out_base=Pin(8))
```

Es werden wegen `autopull=True` und `pull_thresh=32` automatisch 32-Bit-Daten in den FIFO geholt. Diese werden dann aber als 4 Byte-Werte nacheinander ausgegeben.

Die Definition von sm an dieser Stelle ist provisorisch damit sm als globale Variable genutzt werden kann, später wird sie umdefiniert mit $f_{SM}=f\cdot N$

Als Basispin dient GP8, das heisst wir nutzen die Pins GP8...GP15, wobei GP8 das niederwertigste Bit ist.

c) DMA

Es folgt die DMA-Definition:

```
dma0 = DMA()
```

```
dma1 = DMA()

def DMA_Stop():
    dma0.ctrl = 0
    dma1.ctrl = 0

def DMA_Start(words):
    # dma0: From buffer to port (to state machine)
    # size = 2 -> 32 bit transfer   (1 -> B2 signal)
    control0 = dma0.pack_ctrl(inc_read=True,
                              inc_write=False,
                              size=2,
                              treq_sel=0,
                              enable = 1,
                              high_pri= 1,
                              chain_to = 1)
    dma0.config(read=buffer,
                write=PIO0_TXF0,
                count=words,
                ctrl=control0)

    # dma1: chain dma0 for next package of data
    control1 = dma1.pack_ctrl(inc_read=False,
                              inc_write=False,
                              size=2,
                              treq_sel=63,
                              high_pri= 1,
                              enable = 1,
                              chain_to = 0)
    dma1.config(read=addressof(array('i',[addressof(buffer)])),
                write=DMA0_Read,
                count=1,
                ctrl=control1,
                trigger = 1)
```

Das Einfachste ist das Stoppen des DMA-Transfers. Dies geschieht durch Nullsetzen des Kontrollregisters.

Das Starten geschieht für 2 Kanäle, einmal **dma0 für den eigentlichen Datentransfer**, dann **dma1 für das Verketten**.
Es wird beidesmal mit 32-Bit-Werten gearbeitet: size = 2 .
Die Anzahl der Transfers wird in der Variablen words übergeben, die später den Wert int(N/4) erhält. Dieser ergibt sich aus N Samples von denen wegen der 32-Bit Transfers aber jeweils 4 miteinander übergeben werden.

Kanal dma0:

Wir lesen aus dem buffer und schreiben in den PIO_TX FIFO: die Adresse soll automatisch inkrementiert werden, aber das Schreiben erfolgt immer an der gleichen Adresse , also `inc_read=True,inc_write=False,`.

Die Übertragung soll jedesmal erfolgen wenn die PIO Daten anfordert: `treq_sel = 0`.
Für diese Einstellung kommt das TREQ Signal (trigger request) von der PIO.

chaint_to = 1 bewirkt die Verkettung, so dass am Ende des Transfers sofort der andere DMA-Kanal übernimmt.

Kanal dma1:

Dieser hat die Aufgabe den ersten Kanal am Ende einer Periode wieder zu triggern, sodass ein glatter Datendurchsatz entsteht.

Hier werden weder Read- noch Write- Adresse inkrementiert, da wir von einer festen Adresse lesen und in eine feste Adresse schreiben.

`chain_to = 0` da hier keine Verkettung nötig ist. Wir wollen ja nur dma0 starten.

`trigger = 1` bewirkt dass der Kanal sofort gestartet wird.

Mit `treq_sel=63` wird erreicht, dass der Kanal "triggered by software" ist, das heisst dass er **nicht auf ein externes Hardware-Signal wartet** wie dma0, sondern sofort starten kann. [1].

Komplizierter wird es bei `read=addressof(array('i',[addressof(buffer)]))` und `write=DMA0_Read`

Der Zweck ist es, dma0 mitzuteilen wo die neue Startadresse zum Lesen ist. Man könnte meinen diese wäre doch schon definiert bei der ersten Konfiguration. Dies gilt aber nicht mehr, wenn dma0 neu gestartet wird. dma1 muss also die korrekte buffer – Adresse an dma0 übertragen.

`write=DMA0_Read` sagt dma1, dass er etwas in das Read Register von dma0 schreiben soll. Was? Nun, die Adresse des buffers.

Man könnte dies probieren:
`dma1.config(read=addressof(buffer), write=DMA0_Read,...`
aber das würde dazu führen, dass dma1 die Daten aus buffer und nicht die Adresse des buffers an dma0 übergeben würde.

Mit der korrekten Methode wird zuerst ein Integer – Array ('i' = Integer) mit der Adresse des buffers erzeugt: `array('i',[addressof(buffer)])` und die Adresse dieses Arrays (mit nur einem Element) von dma1 gelesen.

Zusammengefasst: es wird die Adresse des buffers gelesen und ins DMA0_Read – Register geschrieben, so dass dma0 die werte für eine neue Periode auslesen kann.

d) Füllen des Buffers

Hierzu wird eine Funktion definiert, die zu Beginn aufgerufen wird. Denkbar sind alle möglichen Funktionen wie Sinus, Rechteck, Dreieck, Sägezahn usw.

Da wir 8 Bit ausgeben, müssen die Werte zwischen 0 und 255 liegen, der Buffer wird mit N Werten gefüllt, eine Periode soll also N Werte umfassen.

Für höhere Frequenzen kann man natürlich auch mehrere Perioden auf diese Weise definieren.

Als Beispiel eine Sinusfunktion:

```
def sine():
    global buffer
    for i in range(N):
        buffer[i]=int(127+127*sin(2*pi*i/N))
```

e) Starten und Stoppen des Generators

```
def start(f):
    global sm
    stop()
    sm=StateMachine(0, parallel, freq= int(f*N), out_base=Pin(8))
    DMA_Start(int(N/4))
    sm.active(1)

def stop():
    global sm
    DMA_Stop()
    sm.active(0)
```

Beim Starten wird die State Machine mit einer Frequent f*N betrieben, da eine Periode N Samples hat. Beim DMA-Start haben wir N/4 Transfers, da immer 4 Bytes miteinander übertragen werden.

f) Einfacher Test

```
def test():
    sine()
    start(1000)
    time.sleep(10)
    stop()

test()
```

Hier wird während 10 Sekunden ein 1kHz – Sinussignal ausgegeben.

g) Grenzen des einfachen Entwurfs

Die tiefste Frequenz ist durch $f \cdot N \geq 2\,kHz$ vorgegeben, da die tiefste Frequenz für die State Machine etwa bei 2kHz liegt (genauer 1.908kHz). Bei N = 4096 ergibt sich also ca. 0.5Hz. Mit der zusätzlichen Einschränkung, dass die Frequenz der State Machine Integer sein muss, ist die untere Grenze praktisch 1Hz. Tiefere Werte erreicht man durch einen längeren Buffer (mehr Samples) oder eine Absenkung der Pico-Taktfrequenz.

Die höchste Frequenz ergibt sich aus der maximalen Frequenz der State Machine (und nicht zu vergessen der maximalen Frequenz des DA-Wandlers!):

$f \cdot N \leq 125\,MHz \rightarrow f \leq 125\,\frac{MHz}{N}$, mit N = 4096 also etwa 30.5kHz.

Wir überstreichen also problemlos den ganzen Audio-Bereich.

Da der Inhalt des Buffers zu Beginn berechnet wird, steht das Signal am Ausgang sofort zur Verfügung, im Gegensatz zu den Entwürfen die auf Rolf Odemanns Algorithmus basieren. Diese haben allerdings einen grösseren Frequenzbereich.

Mein Beispiel sollte ohnehin mehr dem Verständnis dienen als ein fertiger Entwurf sein.

h) Erweiterungen

- Verschiedene Buffer für verschiedene Kurvenformen definieren und zwischen diesen umschalten.
- Für höhere Frequenzen mehrere Perioden im Buffer ablegen.
- Ausgänge abschalten oder auf null legen nach dem Stoppen (sonst kann am Ausgang des DA-Wandlers ein beliebiger Wert anliegen).
- Für einen richtigen Funktionsgenerator einen analogen Ausgangsverstärker mit Potentialverschiebung vorsehen.
- Der Pico hat genug Ressourcen für einen zweiten Generator.
- Mit MIDI-Eingang könnte der Generator als Basis für einen modularen Synthesizer dienen. Könnte man über PIO ein I2S-Audiosignal ausgeben? Ich weiss es (noch) nicht.
- Und natürlich: Frequenzeinstellung über Tastenfeld oder Drehgeber.

4. Dokumentation

[1] Informationen die mir ChatGPT geliefert hat.
Erstaunlich und erschreckend wie effizient man damit arbeiten kann. Statt langwieriger Suchvorgänge bekommt man sekundenschnell ein (strukturiertes!) Ergebnis, das man allerding immer (immer!) überprüfen muss. Es ist ein wenig wie eine fachliche Diskusssion mit einem Kollegen.

[2] Arbitrary waveform generator
→ Rolf Odemann

https://github.com/TFillary/Waveform-Generator
https://magpi.raspberrypi.com/articles/arbitrary-waveform-generatorh
https://hackaday.com/2023/11/29/arbitrary-wave-generator-for-the-raspberry-pi-pico/
https://github.com/favict/pico-waveform-generator
https://www.instructables.com/Poor-Mans-Waveform-Generator-Based-on-RP2040-Raspb/