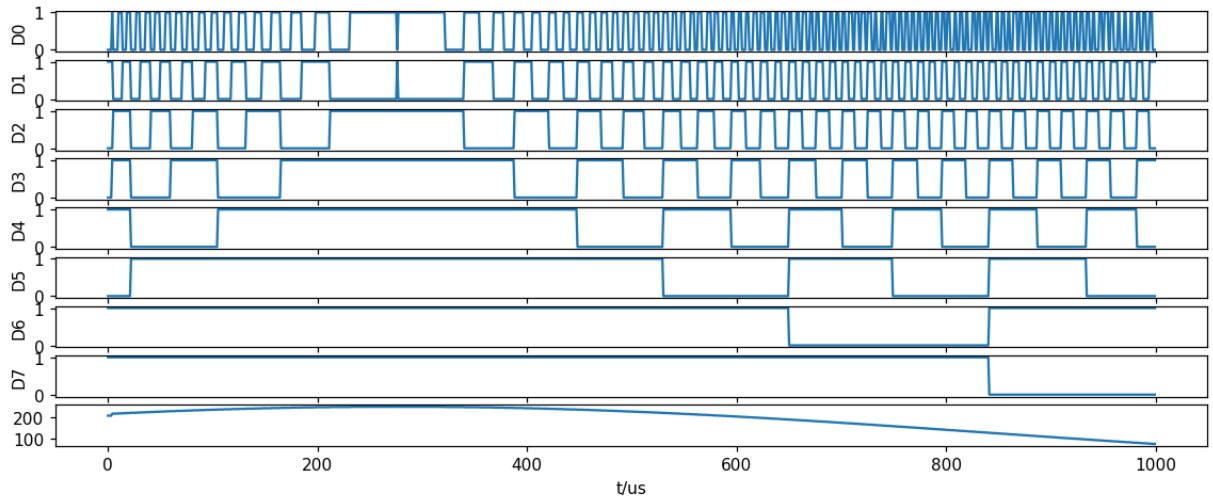
# Einfacher Logic Analyser mit PIO und DMA

jean-claude.feltes@education.lu

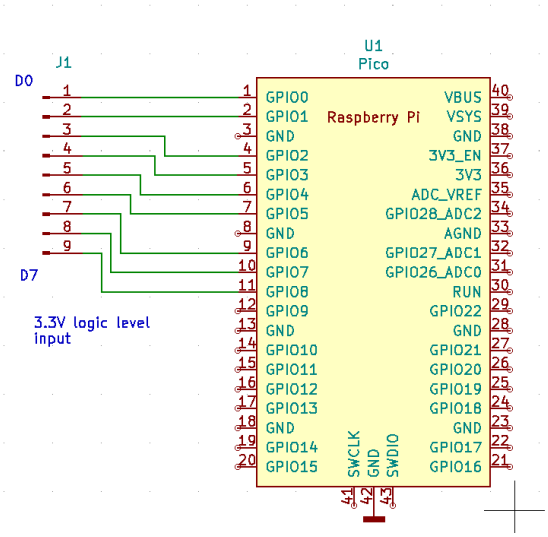
Eine weitere Etappe auf meiner Reise durch die Möglichkeiten der PIO-Programmierung des Raspi Pico. Die Grundlagen schildere ich hier:
https://staff.ltam.lu/feljc/electronics/uPython/PIO_Programmierung.pdf

Das Folgende ist als edukatives Projekt zu verstehen, es reicht aber schon aus, um z.B. die Signale eines anderen Pico zu analysieren. Die Sampling Rate kann dabei theoretisch erstaunlich hoch getrieben werden. In der Praxis kann es hardwarebedingte Einschränkungen durch Reflexionen und Störsignale geben. Ein erster Test mit Flachkabel am Eingang zeigte Übersprechen zwischen den Kanälen, bei freier Verdrahtung mit 10cm langen Breadboard-Drähten gelang mir aber ein Betrieb mit 100MHz Samplerate.
Hier zur Einstimmung ein Diagramm der Ausgangssignale eines DDS-Generators mit f = 440Hz, die übertragenen 8-Bit-Werte zeigen im untersten Diagramm einen Ausschnitt der Sinuskurve.




## 1. Prinzipschaltung

Sie besteht eigentlich nur aus dem Raspi Pico:




Nicht eingezeichnet: USB-Verbindung zu einem PC.

# 2. Übersicht

Das Prinzip ist folgendes:

- Die Zustände von 8 Eingangspins werden als 8-Bit-Daten on einer PIO State Machine parallel in einem Taktzyklus eingelesen.
- Diese Daten werden dann über DMA in einen vorher definierten Pufferspeicher der Grösse N geschrieben.
- Wenn alle N Bytes im Pufferspeicher stehen, werden sie über die USB-Schnittstelle an den PC ausgegeben und / oder auf dem Pico gespeichert.
- Ein Programm auf dem PC muss die Daten einlesen, die einzelnen Bits aus den Byte-Werten herausfiltern und grafisch darstellen.

# 3. Micropython-Software

Die Pico-Software ist inspiriert von Diskussionen im Micropython-Forum, z.B.
https://github.com/orgs/micropython/discussions/15360

**Allgemeines:**

Zunächst werden Module importiert und einige Konstanten gesetzt

```
from machine import Pin
from time import ticks_us, ticks_diff
from rp2 import PIO, DMA, asm_pio, StateMachine

SIZE_BYTE       = const(0)
DREQ_PIO0_RX0   = const(4)
PIO0_BASE   = const(0x50200000)
RXF0        = const(0x020)
```

Die Konstanten beziehen sich auf den DMA.

Es folgt die Definition globaler Variablen:

```
scanfreq = 100_000_000
N = 10_000
resolution_us = (1/scanfreq) * 1E6
values = bytearray(N)
```

`scanfreq` legt die Samplerate fest, hier mit dem höchsten Wert von 100MHz.
`N` ist die Anzahl der abgetasteten Werte

Diese Werte können aber jederzeit mit der Funktion `set_parameters()` neu gesetzt werden.

Anschliessend werden die Eingänge GPIO0 bis GPIO7 als Inputs deklariert:

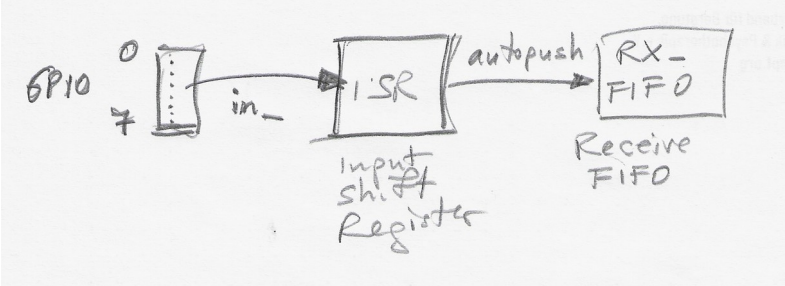
```
def set_inputs():
    # define GPIO0...7 as inputs
    for i in range(0,8):
        Pin(i, Pin.IN)
```

**Einlesen der Pin-Zustände:**

```
@asm_pio(in_shiftdir=PIO.SHIFT_LEFT, autopush=True, push_thresh=8)
def pio_fetchbytes():
    in_  (pins,8)
```

Mit `in_` werden jeweils 8 Bit parallel ins Input Shift Register geladen. (`in_` statt `in` wird benutzt um Verwechslungen mit dem Python-Keyword `in` zu vermeiden.

`autopush=True` sorgt dafür, dass die Daten automatisch vom Input Shift Register (ISR) in den Receive FIFO (RX_FIFO) geschrieben werden, dies sofort wegen `push_thresh=8` , also bei jedem Byte.




**Instanziieren eines State Machine Objekts:**

```
def prepare_sm():
    sm = StateMachine(0,pio_fetchbytes, freq = scanfreq, in_base=0)
    return sm
```

Hier wird festgelegt, dass die Funktion pio_fetchbytes() mit der Frequenz scanfreq ausgeführt wird.

`in_base = 0` legt fest, dass der LSB-Pin an GPIO0 liegt, es werden also die Eingänge GPIO0...7 benutzt.

Der erste Parameter 0 legt fest, dass die State Machine mit dem Index 0 benutzt wird, es könnte genausogut eine ander der 8 SM benutzt werden.

**Vorbereitung des DMA-Objekts:**

```
def prepare_DMA():
    dma = DMA()
    control = dma.pack_ctrl(inc_read=False, inc_write=True,
        size=SIZE_BYTE, treq_sel=DREQ_PIO0_RX0)
    dma.config(read=PIO0_BASE+RXF0, write=values, count=N, ctrl=control)
    return dma
```

Früher musste man zur Konfiguration Registerwerte setzen (siehe Datenblatt), heute (Dezember 2024) geht es einfacher, wenn auch nicht alle Parameter selbsterklärend sind.

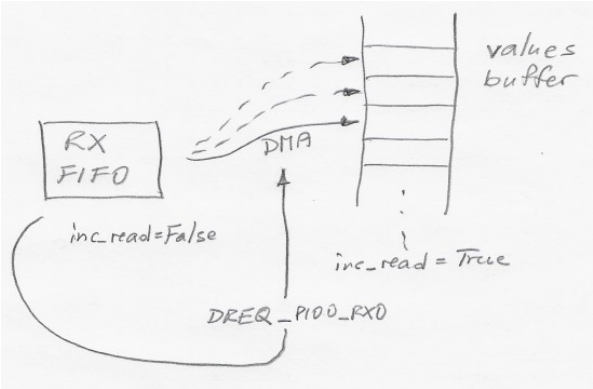
Die Konfiguration erfolgt in 2 Schritten: einige Parameter werden in der Struktur `control` zusammengefasst. Diese wird dann zusammen mit den restlichen Parametern an dma.config übergeben.

Hier die Bedeutung der control - Parameter:

- **`inc_read=False`** : wir lesen die Daten immer an der gleichen Stelle, nämlich vom RX_FIFO der State Machine sm.
- **`inc_write=True :`** wir schreiben die Daten in aufeinanderfolgende Speicheradressen
- **`size=SIZE_BYTE`** : klar, es werden einzelne Bytes entsprechend den eingelesenen 8 Eingängen geschrieben
- **`treq_sel=DREQ_PIO0_RX0 :`** Der DMA-Transfer wird durch ein Transfer Request (TREQ) – Signal getriggert. Dieses kann von 40 verschiedenen Quellen kommen (siehe Datenblatt). In unserem Fall: **`treq_sel=DREQ_PIO0_RX0,`** das heisst es wird getriggert vom Receive FIFO der State Machine 0, also bei jedem eingelesenen Byte.

Zusätzliche Parameter bei **`dma.config`**:

- **`read=PIO0_BASE+RXF0`** gibt die Quelladresse für das Lesen an, hier die Adresse des RX FIFO Registers
- **`write=values`**: geschrieben wird in das Buffer-Array mit dem Namen values.
- **`count=N:`** es werden N "Pakete" von 1 Byte Grösse geschrieben. (Hier wegen size = SIZE_BYTE, es könnten in einem anderen Fall auch 16 Bit oder 32 Bit – "Päckchen" sein.)
- **`ctrl=control`** lädt die oben definierte Kontrollstruktur
- **`trigger = False`** wird hier nicht explizit gesetzt da der Defaultwert False ist.
  In diesem Fall wird der Transfer erst gestartet wenn `trigger = True` gesetzt wird.




**Scannen der Werte:**

```
def scan():
    set_inputs()
    sm = prepare_sm()
    dma = prepare_DMA()
    sm.restart()
    sm.active(True)

    dma.config(trigger=True)
    while dma.count:
        pass
```

Hier wird der trigger-Parameter auf True gesetzt, d.h. der DMA-Transfer kann beginnen.

Wir warten ab bis dma.count = 0 ist, d.h. bis alle N Werte eingelesen sind.

Nun steht das Ergebnis im Buffer values.

Der Rest des Programms dient zum Setzen der Parameter, zum Speichern und Ausgeben der Werte:

```
def set_parameters(n, freq):
    global N, values, scanfreq, resolution_us
    N = n
    values = bytearray(N)
    scanfreq  = freq
    resolution_us = (1/scanfreq) * 1E6

def print_values():
    #print(str(values))
    numbers = list(values)
    print(numbers)

def print_info():
    print('starting')
    print("Resolution: ", resolution_us, "us")
    print("Number of samples: ", N)
    print("time: 0 ... ", N * resolution_us, "us")

def save():
    with  open("la_values.dat", "w") as f:
        f.write(str(list(values)))
        f.write('\n\r')
    f.close()

def test():
    set_parameters(10000, 1_000_000)     # n, freq
    print_info()
    scan()
    print_values()
    save()
```

# 4. PC-Software

Das Einfachste ist natürlich das Auslesen der Werte in der Date values.dat mit Thonny und die anschliessende Verarbeitung. Das ist machbar, aber nicht sehr elegant.

Schöner ist es mit einem eigenen Programm, das auch die grafische Darstellung übernimmt.

Zu Kommunikation mit dem Pico benutze ich ein Modul das eigens dafür überarbeitet wurde:

https://github.com/jean-claudeF/Picoconnect/tree/main

Mit diesem Modul kann ein Pico über seinen Namen gefunden werden, auch wenn mehrere Picos über USB angeschlossen sind.
Über ein Objekt Pico sind Methoden verfügbar, mit denen Funktionen auf dem Pico ausgeführt werden können.

Dadurch wird dein Grund-Programm angenehm einfach und kurz.

**Importe und Parameter:**

```
from picoconnect_pa01 import *
import matplotlib.pyplot as plt
import numpy as np

N = 10_000
scanfreq = 10_000_000
dt_us = 1000_000 / scanfreq
```

**Verbindung zum Pico aufbauen:**

```
dict = create_pico_dictionary()
LA = Pico("LOGICANALYSER")
LA.connect()
```

Zunächst wird ein Dictionary der angeschlossenen Picos erzeugt, welches im Modul `picoconnect_pa01` intern gespeichert wird. Dieses enthät die informationen über USB-Ports mit Picos und Namen (Keywords) der Picos. Voraussetzung ist eine Datei info.txt auf dem Pico, welche hier das Keyword “LOGICANALYSER” enthält. Mit diesem Pico verbinden wir uns.

**Importe und Parameter auf dem Pico:**

```
LA.execute("from logic_analyser_03 import *")

cmd = "set_parameters("+ str(N) + ", " + str(scanfreq) + ")"
LA.execute(cmd)
LA.execute("print_info()")
```

Hier zeigt sich das Schöne an der Micropython-Programmierung! Wir können Funktionen aufrufen, Befehle ausführen und Variablen setzen. Das wäre mit einer kompilierten Sprache wie Arduino-C nicht möglich.

**Pico-Eingänge scannen und Werte speichern:**

```
LA.execute("scan()")
LA.execute("save()")
```

**Werte an den PC schicken und dort empfangen:**

```
t = LA.read_file("la_values.dat")
exec("y = " + t)
```

Die Werte stehen in der Datei values.dat als Liste [val1, val2, ….]

Um sie verarbeiten zu können wird ein exec benutzt. Danach ist y = [val1, val2, ….] und wir können den Vektor y beutzen.

**Verarbeitung der Werte:**

```
t = range(0, N)
t = np.array(t)
t = t*dt_us
y = np.array(y)

# binary values:
b = []
for j in range(0,8):
    b.append((y & 2**j) >> j)
```

Es werden die Numpy-Arrays für die Zeit t und für y erzeugt.

Aus den y-Werten werden dann die Binär-Werte für D0...D7 herausgefiltert.

**Plotten und Schliessen der Verbindung:**

```
fig = plt.figure()
for j in range(0,8):

    fig.add_subplot(9,1,j+1)
    plt.plot(t,b[j])
    plt.ylim(-0.05,1.05)
    plt.ylabel("D"+str(j))

ax8 = fig.add_subplot(9,1,9)
ax8.plot(t, y)
plt.xlabel("t/us")
plt.show()

LA.disconnect()
```

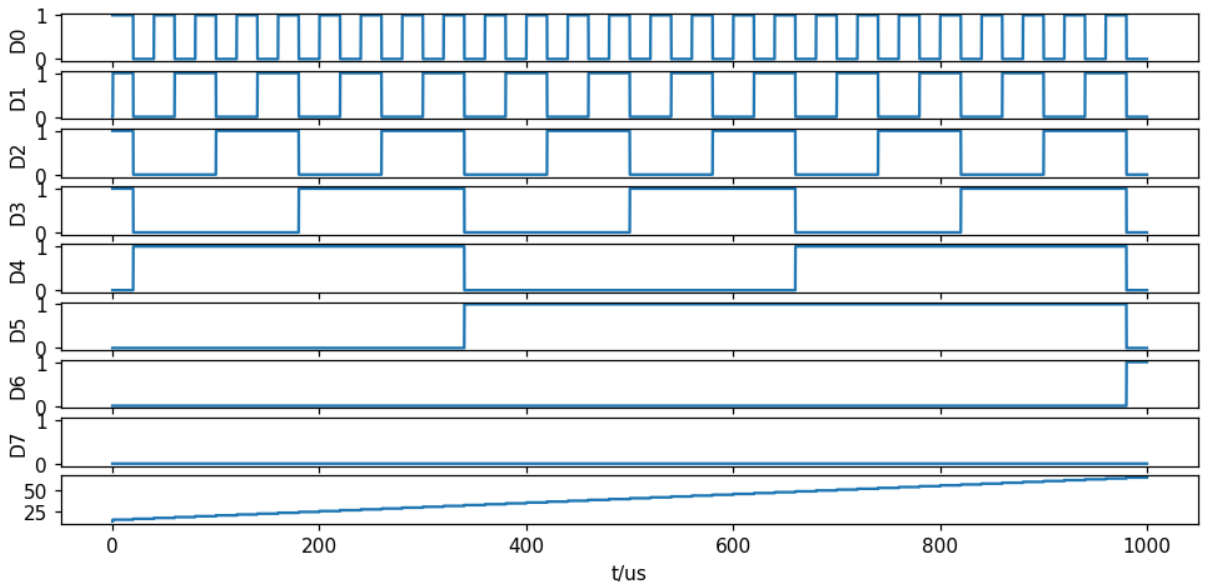
Matplotlib ist einfach und leistungsfähig!

Es werden 9 Subplots erzeugt, 8 davon für D0 bis D7, dann noch eines für die durch die einzelnen Bits dargestellte Zahl.

Der Befehl `fig.add_subplot(9,1,j+1)` erzeugt 9 Subplots in einer Spalte mit j+1 als Nummer die den Subplot positioniert.

# 5. Beispiel einer Messung

Das Ergebnis für die Signale eines Binärzählers sieht dann z.B. so aus:




mit N = 10_000, Samplerate 10MHz

Der Zähler lief dabei auf einem zweiten Pico und hatte dieses Programm:

```
from rp2 import PIO, StateMachine, asm_pio
from machine import Pin

@asm_pio(out_init = (PIO.OUT_HIGH,) * 8, out_shiftdir = PIO.SHIFT_RIGHT)
def parallel_out():
    pull()
    out(pins, 8)

sm = StateMachine(0, parallel_out, freq = 100_000_000, out_base = Pin(8))
sm.active(1)

while True:
    for i in range(255):
        sm.put(i)
```

Hier wird i in einer for-Schleife hochgezählt und über die parallele Schnittstelle ausgegeben, was ein Sägezahnsignal erzeugt. Die Geschwindigkeit des Zählers ist dabei weitgehend unabhängig von der Frequenz der State Machine, solange sie hoch genug ist (z.B. höher als 10MHz). Sie wird durch den Micropython-Teil begrenzt und hat für D0 eine Frequenz von nur 36kHz, mit einem merklichen Jitter, bedingt durch den zeitlich nicht deterministischen Micropython-Teil.

Man kann übrigens mit höherer Abtastfrequenz, z.B. 100MHz versuchen aus dem Diagramm den Jitter zu bestimmen, da man Werte bequem mit der Maus anwählen und die Koordinaten ablesen kann. Das Ergebnis zeigt eine Periodendauer von D0 zwischen 26 und 27.2µs, was durch eine Messung mit dem Oszilloskop bestätigt wird.

# 6. Ausblick

Hier wurde ein Projekt gezeigt das in vielen Hinsichten ausbaufähig ist.
Die Software wurde, um das Verständnis zu erleichtern, möglichst einfach gehalten.

Einige Ideen zum Weitermachen:

- grafische Oberfläche für die PC-Software mit bequemem Einstellen der Parameter
- Puffer an den Pico-Eingängen zum Schutz vor Überspannung beim Auslesen von 5V-Signalen
- Triggermöglichkeit
- Decodierung von Bus-Signalen
- usw.

Viel Spass bei der Erweiterung!

Vollständiges Micropython-Programm logic_analyser_03.py:

```
scanfreq = 100_000_000      # global var, can be set with set_parameters
resolution_us = (1/scanfreq) * 1E6

# Allocate array to store data:
N = 10_000      # Scan N values
values = bytearray(N)      # result array


from machine import Pin
from time import ticks_us, ticks_diff
from rp2 import PIO, DMA, asm_pio, StateMachine

SIZE_BYTE       = const(0)
DREQ_PIO0_RX0   = const(4)

PIO0_BASE   = const(0x50200000)
RXF0        = const(0x020)

def set_inputs():
    # define GPIO0...7 as inputs
    for i in range(0,8):
        Pin(i, Pin.IN)

@asm_pio(in_shiftdir=PIO.SHIFT_LEFT, autopush=True, push_thresh=8)
def pio_fetchbytes():
    in_         (pins,8)


def prepare_sm():
    sm = StateMachine(0,pio_fetchbytes, freq = scanfreq, in_base=0)
    return sm

def prepare_DMA():
    dma = DMA()
    control = dma.pack_ctrl(inc_read=False, inc_write=True, size=SIZE_BYTE,
treq_sel=DREQ_PIO0_RX0)
    dma.config(read=PIO0_BASE+RXF0, write=values, count=N, ctrl=control)
    return dma

def scan():
    set_inputs()
    sm = prepare_sm()
    dma = prepare_DMA()
    sm.restart()
    sm.active(True)

    t0 = ticks_us()
    dma.config(trigger=True)
    while dma.count:
        pass
    t1 = ticks_us()
    #dma.close()
    dt = t1 - t0
    print(f'scanned {N} times in {ticks_diff(t1,t0)} us')

def set_parameters(n, freq):
    global N, values, scanfreq, resolution_us
    N = n
    values = bytearray(N)
```

```
    scanfreq  = freq
    resolution_us = (1/scanfreq) * 1E6

def print_values():
    #print(str(values))
    numbers = list(values)
    print(numbers)

def print_info():
    print('starting')
    print("Resolution: ", resolution_us, "us")
    print("Number of samples: ", N)
    print("time: 0 ... ", N * resolution_us, "us")

def test():
    set_parameters(10000, 1_000_000)     # n, freq
    print_info()
    scan()
    print_values()
    save()

#--------------------------------------------------------------------
import os

def save():
    with  open("la_values.dat", "w") as f:
        f.write(str(list(values)))
        f.write('\n\r')
    f.close()

if __name__ == "__main__":
    test()
```