PIO Programmierung beim Raspi Pico

Von jean-claude.feltes@education.lu

Doku:
https://docs.micropython.org/en/latest/library/rp2.StateMachine.html#rp2.StateMachine

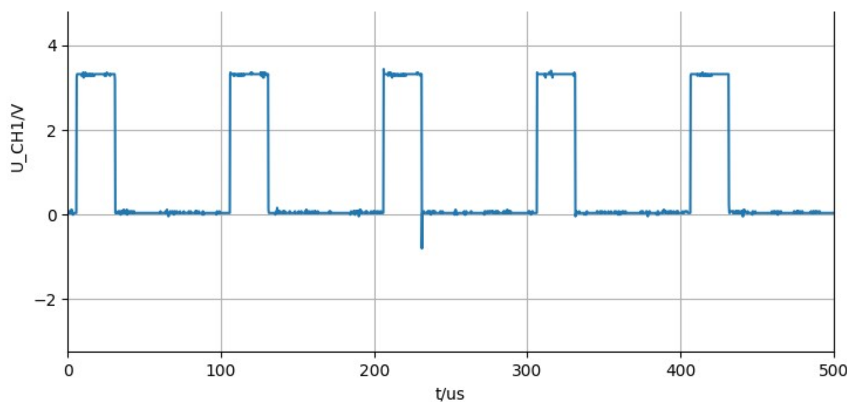
Beispiele:
https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio

Youtube:
https://www.youtube.com/watch?v=YafifJLNr6I&list=PLiRALtgGsxmZs_LXGkh09Zr2NUmk_mtEI&index=1&pp=iAQB

https://www.youtube.com/watch?v=yYnQYF_Xa8g
https://youtu.be/wet9CYpKZOQ (in C)

C Dokumentation:
https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf

1. Beispiel: 10 kHz PWM – Signal mit 25% Tastgrad




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


@asm_pio(set_init = PIO.OUT_LOW)
def pwm1():
    set(pins, 0)   [2]
    set(pins, 1)


sm = StateMachine(1, pwm1, freq= 40000, set_base = Pin(15))

while True:
    sm.active(1)
```

```
    time.sleep(1)
    sm.active(0)
    time.sleep(1)
```

`@asm_pio`

Die folgenden Instruktionen beziehen sich auf PIO – Assembler, die definieren eine State Machine. Mit set_init kann deren Initialzustand festgelegt werden.

`set(pins, 1)`

setzt ein Pin auf 1 (oder 0)

`[2]`

Jede Instruktion dauert genau 1 Taktzyklus. Mit einem Wert von 1 bis 31 (hier 2) kann ein Delay von 1 bis 31 Taktzyklen eingefügt werden.
Das Unterprogramm pwm1 dauert also 1 + 2 +1 = 4 Taktzyklen.

Für längere Delays müssen `nop() [31]` – Instruktionen eingefügt werden. (nop = No Operation).

2

`sm = StateMachine(1, pwm1, freq= 40000, set_base = Pin(15))`

Hiermit wird das Unterprogramm pwm1 in die Statemachine sm geladen, hier die mit der Nummer 1. Es gibt 8 State Machines, nummeriert von 0 bis 7.
Die Frequenz wird auf 40kHz festgelegt, damit sich ein PWM – Signal von 40kHz / 4 = 10kHz ergibt.
Der Wert freq kann zwischen 1.908kHz und 125MHz (!) liegen.
Es gibt 8 unabhängige State Machines.

Mit set_base = Pin(15) wird der Pin für die set – Instruktion festgelegt.

In der Hauptschleife wird die State Machine jede Sekunde einmal ein- bzw. ausgeschaltet.

Das Oszilloskop bestätigt, dass auch sehr hohe Frequenzen möglich sind, und dass es unter ca. 2kHz nicht mehr funktioniert.

2. Rechteckgenerator 1kHz bis 62MHz

Fast noch einfacher lässt sich ein Hochfrequenz – Rechteckgenerator programmieren:

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

pin_out = Pin(16)

@asm_pio(set_init = PIO.OUT_LOW)
def rectgen():
    set(pins, 0)
    set(pins, 1)

while True:
    f = input("Frequency/kHz (1 ... 62250): ")
    freq = int(float(f) * 1000)
    print(freq, "kHz")
```

```
    sm = StateMachine(0, rectgen, freq= 2* freq, set_base = pin_out)
    sm.active(1)
```

3. Beispiel: Blinken mit 10Hz

Originalbeispiel aus der Doku: (Schnelles Blinken während 2 Sekunden, f = 6.25Hz):

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

led = Pin(25, Pin.OUT)

@asm_pio(set_init = PIO.OUT_LOW)
def blink():
    wrap_target()
    set(pins, 1)    [31]
    nop()           [31]
    nop()           [31]
    nop()           [31]
    nop()           [31]
    set(pins, 0)    [31]
    nop()           [31]
    nop()           [31]
    nop()           [31]
    nop()           [31]
    wrap()

sm = StateMachine(1, blink, freq= 2000, set_base = led)

sm.active(1)
time.sleep(2.00)
sm.active(0)
```

wrap_target() und wrap() definieren eine Schleife die solange ausgeführt wird bis die State Machine gestoppt wird. Dabei ist wrap_taget() der Anfang und wrap() der Sprungbefehl zu wrap_target().
Der Code zwischen beiden wird sozusagen “ge-wrapped”, also eingepackt.
Sind diese Befehle notwendig? Eigentlich nicht, denn die State Machine wird sowieso in einer Schleife fortwährend ausgeführt. Sie waren im offiziellen Beispiel vorhanden, an kann sieaber auch weglassen.
Hier https://www.youtube.com/watch?v=wet9CYpKZOQ&t=806s wird erklärt warum man sie eventuell doch benötigt: wenn es innerhalb des Programms eine zusätzliche Schleife geben soll. Wrap spart hier einen Jump-Befehl.

Die Zahlen in eckigen Klammern [n] bezeichnen eine Verzögerung um n Taktzyklen (1...31).
[31] → Verzögerung um 31 Zyklen, zusammen mit 1 Zyklus für jeden Befehl also 32.

Die SM (State Machine) wird wegen der niedrigen Frequenz mit der kleinstmöglichen Taktfrequenz betrieben. Die Taktperiodendauer ist hier $\frac{1}{2\,kHz}=500\,\mu s$.

Jede Zeil dauert also $32 \cdot 0.5\,ms = 16\,ms$, im Beispielprogramm also insgesamt also für eine halbe Periode $5 \cdot 16\,ms = 80\,ms$, für eine Periode also 160ms, entsprechend einer Frequenz von 6.25Hz.

Will man eine Frequenz von 10Hz, muss das Beispiel modifiziert werden. Die Periodendauer ist 100ms, eine halbe Periode dauert also 50ms.
Das mögliche Raster für die Periode ist die Taktperiode, also 0,5ms.
Davon braucht man für die halbe Periode 50ms / 0.5ms = 100.
Die maximale Verzögerung einer Zeile ist 31 + 1 = 32.
Von diesen braucht man also 100 / 32 = 3.125
Also 3 Zeilen mit [31], dann hat man 96 Taktzyklen, es fehlen noch 4, also braucht man noch eine Zeile

```
nop()     [3]
```

Das modifizierte Beispiel sieht dann so aus:

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

led = Pin(25, Pin.OUT)

@asm_pio(set_init = PIO.OUT_LOW)
def blink():
    set(pins, 1)    [31]
    nop()           [31]
    nop()           [31]
    nop()           [3]
    set(pins, 0)    [31]
    nop()           [31]
    nop()           [31]
    nop()           [3]

sm = StateMachine(1, blink, freq= 2000, set_base = led)

sm.active(1)
time.sleep(2.00)
sm.active(0)

led = Pin(25, Pin.OUT)
led.value(0)
```

Beim Testen des Beispiels fiel mir auf, dass je nach der im Befehl time.sleep(2) eingestellten Zeit die LED nicht ausgeschaltet wurde, wie es eigentlich der Fall sein sollte. Eine Neuinitialisierung löste das Problem:

```
led = Pin(25, Pin.OUT)
led.value(0)
```

4. Niederfrequenz - Rechteckgenerator

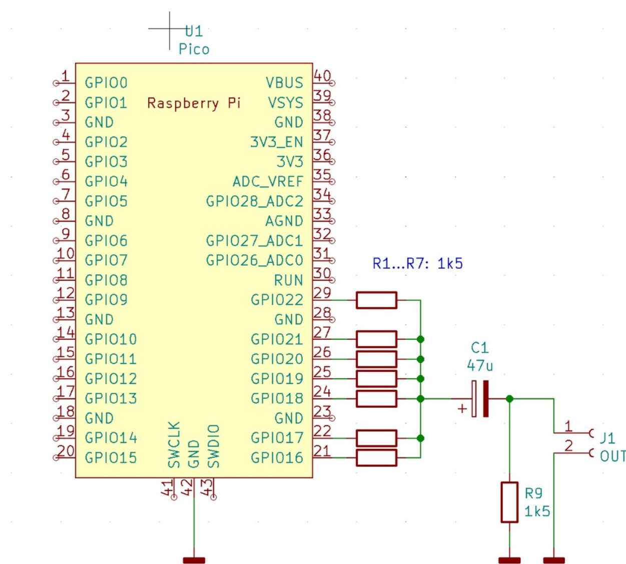
Das erste Rechteckgenerator – Beispiel erlaubte nur recht hohe Frequenzen.
Mit unserer “Verzögerungstaktik” kann eine Periode auf 100 Zyklen gedehnt werden, so dass die tiefste Frequenz 2kHz / 100 = 20Hz beträgt:

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

pin_out = Pin(16)

@asm_pio(set_init = PIO.OUT_LOW)
def rectgen():                                # 1 period = 100 cycles
    set(pins, 0)   [29]     # 30 cycles
    nop()          [19]     # 20 cycles
    set(pins, 1)   [29]     # 30 cycles
    nop()          [19]     # 20 cycles

# start with f = 1kHz
sm = StateMachine(0, rectgen, freq= 100000, set_base = pin_out)
sm.active(1)


while True:
    fs = input("Frequency / Hz (20 ... 1.25M): ")
    f = int(float(fs))
    if f <= 20:
        sm.active(0)
        print("Stopped")
    else:
        print(f, "Hz")
        sm = StateMachine(0, rectgen, freq= 100* f, set_base = pin_out)
        sm.active(1)
```

5. Exkurs: Generator für Synthesizer?

Mit den 8 State Machines könnte man doch 8 Töne gleichzeitig erzeugen! Ein Rechtecksignal ist musikalisch nicht sooo interessant, es hat nur ungeradzahlige Oberwellen. Aber mehrere Rechtecksignale addiert, das könnte schon interessanter klingen, vorausgesetzt es gibt eine leichte Differenz in der Frequenz, so dass eine Schwebung entsteht. Ausserdem kann man Harmonische und Subharmonische zur Grundfrequenz addieren, was einen Akkord ergibt.

Für ein erstes Experiment benutzte ich diese Schaltung:



und diesen Code:

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

pin_out = [Pin(16), Pin(17), Pin(18), Pin(19), Pin(20), Pin(21), Pin(22),
Pin(26)]

@asm_pio(set_init = PIO.OUT_LOW)
def rectgen():                                # 1 period = 100 cycles
    set(pins, 0)    [29]     # 30 cycles
    nop()           [19]     # 20 cycles
    set(pins, 1)    [29]     # 30 cycles
    nop()           [19]     # 20 cycles


def tone(f, index):
    print(f, "Hz")
    sm = StateMachine(index,rectgen,freq= int(100* f),set_base = pin_out[index])
    sm.active(1)

tone(880, 0)
tone(440.1, 1)
tone(439.8, 2)
tone(439.5, 3)
tone(220.2, 4)
tone(330.5, 5)
tone(111, 6)
tone(110, 7)
```

Das Programm erzeugt einen Ton der Note A in verschiedenen Oktaven mit beigemischter Quinte E und leichter Schwebung.

6. Blinken mit 1Hz

Für so langsame Frequenzen müsste man unzählige nop()[31] einfügen.
Hier kommt der zählende Jump-Befehl ins Spiel, der einer FOR – NEXT – Schleife entspricht.

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

led = Pin(25, Pin.OUT)
#led = Pin(15, Pin.OUT)

@asm_pio(set_init = PIO.OUT_LOW)
def blink():

    # half period with 1          Cycles:
    set(pins, 1)                  # 1
    set(x, 31)       [6]          # 1 + 6

    label("delay_1")
    nop()            [29]         # (1+29)
    jmp(x_dec, "delay_1")         #  1      = 31, * (31+1)
```

```
    # half period with 0
    set(pins, 0)

    set(x, 31)       [6]
    label("delay_0")
    nop()            [29]
    jmp(x_dec, "delay_0")


sm = StateMachine(1, blink, freq= 2000, set_base = led)
sm.active(1)
```

Zunächst wird mit set(x, 31) das X-Register gesetzt. Es gibt 2 “Scratch”-Register X und Y.

Der Befehl jmp(x_dec, label) springt zum vorher festgelegten Label, solange x nicht null ist. Bei jedem Sprung wird x automatisch decrementiert. Der Startwert im X-Register gibt also an, wie oft die Schleife ausgeführt wird.

Wenn man versucht längere Verzögerungen durch Werte > 31 im X-Register zu erreichen, wird man sofort mit den Einschränkungen des PIO-Assemblers konfrontiert. Am besten versteht man diese indem man einen Blick (na ja, einer wird wohl nicht genügen!) in das C-Handbuch wirft:

https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf

Hier sind im Kapitel “PIO Instruction set reference” die Befehle erklärt.

Hier der Aufbau des set- Befehls:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SET	1	1	1	Delay/side-set					Destination			Data				

Man sieht, dass für die Daten nur 5 Bit zur Verfügung stehen, das reicht gerade mal für die Zahlen 0...31. Wenn man jetzt aufgrund dieser Beschränkungen Zweifel am Sinn des PIO-Systems hat, sollte man sich daran erinnern dass sein Zweck nicht Datenverarbeitung, sondern die Entwicklung von peripheren Interfaces ist, und darin ist PIO echt stark!

Unsere Schleife wird also hier mit X = 31 gerade 32 mal ausgeführt. Im Innern der Schleife haben wir eine Verzögerung von 29 Zyklen, dazu addiert sich 1 Zyklus für nop() und 1 Zyklus für den Jump-Befehl, insgesamt also 31. Die gesamte Dauer für eine halbe Periode ist also 32 * 31 + 8 = 1000 Zyklen. Bei einer Taktfrequenz von 2kHz (Periodendauer 0.5ms) ergibt das eine Dauer von 500ms, also genau richtig.

7. Verwirrung beim Timing?

Bei den Timing – Überlegungen ist man (bzw. bin ich) schnell verwirrt.
Es drängen sich Fragen auf, z.B.:

Frage	Antwort
Wieviel Zyklen dauert `nop()` allein ? (Macht das überhaupt Sinn?)	1
Wie lange dauert z.B. `nop() [5]`	1 + 5 = 6
Wievielmal wird eine Schleife mit `jmp(x_dec, ..)` durchlaufen, wenn vorher `set(x, 3)` gesetzt wurde.	3 + 1 = 4
Wieviele Zyklen braucht `label( “...”)`	0

Um mir darüber klar zu werden half ein kleines Testprogramm und das Oszilloskop:

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

@asm_pio(set_init = PIO.OUT_LOW)
def pwm1():
                               # Cycles:     Output
    set(pins, 0)               # 1            0   during 2 / 3 / 4 / 5 cycles
    #nop()                      # 1
    #nop()    [1]               # 2

    set(x, 2)                   # 1
                                #---------------
    label("start_loop")          # 0
    set(pins, 1)                 # 1           1  during (x+1) * 2 cycles
    jmp(x_dec, "start_loop")     # 1


sm = StateMachine(1, pwm1, freq= 10000, set_base = Pin(15))
sm.active(1)
```

Je nachdem ob und wie man die nop()-Zeilen mitnimmt, ergeben sich für die 0-Phase 2 – 5 Zyklen.

Die 1-Phase ist immer (2+1)*2 = 6 Zyklen lang.

8. Interrupt jede Sekunde

Hierzu muss das vorige Blinkprogramm nur leicht modifiziert werden:

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

led = Pin(25, Pin.OUT)

@asm_pio(set_init = PIO.OUT_LOW)
```

```
def blink():

    # half period with 1          Cycles:
    set(pins, 1)                  # 1
    irq(rel(0))                   # 1
    set(x, 31)       [5]          # 1 + 5

    label("delay_1")
    nop()            [29]         # (31+1) * (29 + 1 + 1)
    jmp(x_dec, "delay_1")

    # half period with 0
    set(pins, 0)

    set(x, 31)       [6]
    label("delay_0")
    nop()            [29]
    jmp(x_dec, "delay_0")

def int_handler(sm):
    print(time.time())

sm = StateMachine(0, blink, freq= 2000, set_base = led)
sm.irq(int_handler)
sm.active(1)

time.sleep(5)
sm.active(0)
```

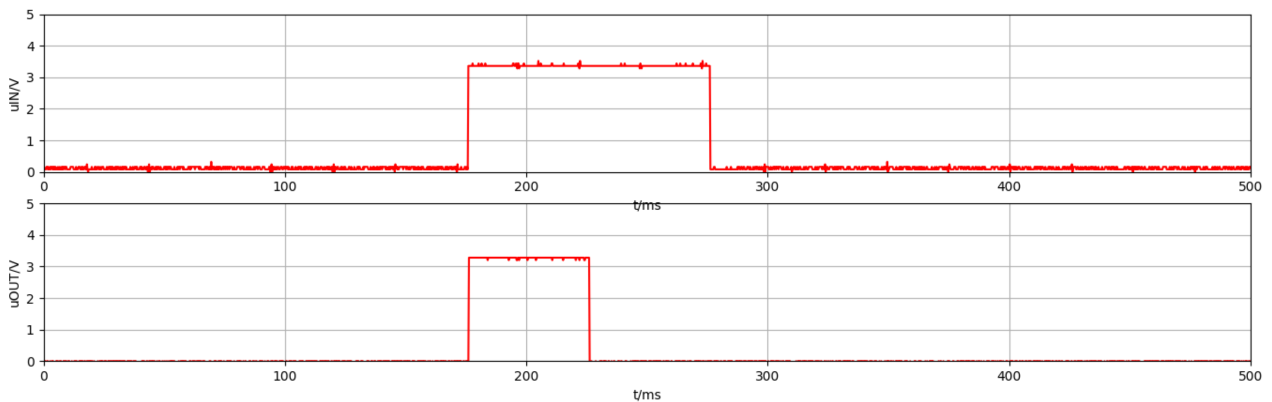
Hier ist zunächst die Instruktion `irq(rel(0))` hinzugekommen, die einen Interrupt aufruft.
Das rel(0) bedeutet dass der Interrupt sofort ausgeführt wird. Dagegen würde z.B. `irq(rel(4))` bedeuten, dass der interrupt erst nach 4 Taktzyklen ausgeführt werden soll.

Es muss noch eine Interrupt-Handler-Funktion definiert werden.
Achtung: Dem Interrupt-Handler muss bei der Definition eine Referenz auf die State Machine übergeben werden (hier sm), auch wenn diese nicht benutzt wird. Man könnte sie aber benutzen um mit der state Machine zu interagieren. Ausserdem muss im Fall mehrerer State Machines ja festgelegt werden, welcher Handler zu welcher Maschine gehört.

Da in unserer ersten halben Periode ein Befehl hinzugekommen ist, muss die Verzögerung adaptiert werden: `set(x, 31)       [5]`

9. Interrupt durch externes Signal

Ein Eingangssignal soll einen Interrupt auslösen.
Im Besispiel wird daraufhin ein Ausgangssignal mit bestimmter Länge erzeugt:



oben: Eingangssignal an Pin 16, unten: Ausgangssignal an Pin 15

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

pin16 = Pin(16, Pin.IN, Pin.PULL_UP)
pin15 = Pin(15, Pin.OUT)

@asm_pio()
def wait_pin():
    wait(1, pin, 0)
    irq(block, rel(0))
    wait(0, pin, 0)

def int_handler(sm):
    print(time.time(), sm)
    pin15.value(1)
    time.sleep(0.05)
    pin15.value(0)

sm = StateMachine(0, wait_pin, in_base = pin16)
sm.irq(int_handler)
sm.active(1)

while True:
    time.sleep(0.01)
```

In diesem Beispiel wird ein Interrupthandler in Micropython aufgerufen, als Demo. Er benutzt einen Printbefehl und setzt einen Pin für 50µs auf H. Auf diese Weise kann eine Art Monoflop-Verhalten simuliert werden.

Wollte man dies tun um gezielt Impulse bestimmter Länge zu erzeugen, wäre es aber günstiger die Impulse innerhalb der PIO-Funktion zu erzeugen, dann ist das Verhalten deterministisch.

10. Parallelausgabe

Standardmässig gibt es (im Gegensatz zu BASCOM oder Arduino C) in Micropython keine Möglichkeit, einen 8-Bit-Port als Ganzes zu setzen. Man muss die einzelnen Pins der Reihe nach setzen oder rücksetzen. Dabei entstehen kurzzeitig ungewollte Zustände.

Ein kurzes PIO-Programm kan helfen:

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

@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 = 10000000, out_base = Pin(0))
sm.active(1)

while True:
    for i in range(255):
        sm.put(i)
        time.sleep_us(10)
```

`out_init = (PIO.OUT_HIGH,) * 8`
definiert 8 konsekutive Output Pins die im Ruhezustand High sind.

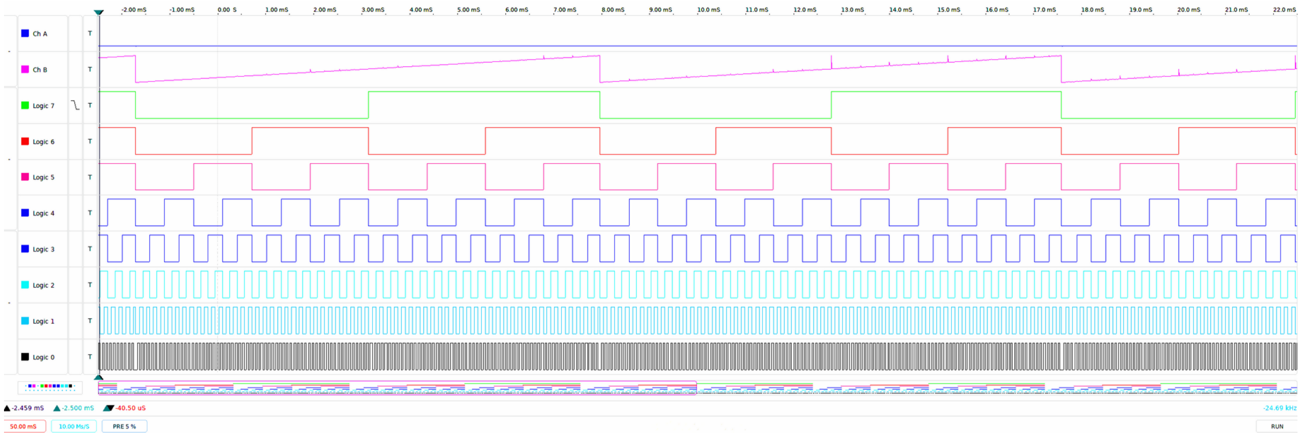
`out_shiftdir = PIO.SHIFT_RIGHT`
definiert die Schieberichtung so, dass das MSB (Most Significant Bit) beim Pin mit der höchsten Nummer liegt (hier GPIO7). Anders ausgedrückt liegt das LSB (Least Significant Bit) an dem Pin der mit `out_base = Pin(0)` als GPIO0 festgelegt wird bei der Instantiierung der State Machine. Genaueres siehe weiter unten.

Der `pull()`- Befehl holt Daten aus dem TX FIFO ab. Defaultmässig wartet die Maschine bis Daten vorhanden sind.

Mit `out(pins, 8)` werden die 8 Datenbits dann am GPIO-Port ausgegeben.

Die State Machine wird mit einer recht hohen Frequenz initialisiert damit die Verzögerungen klein bleiben (hier nicht besonders wichtig).

Als Demo läuft ein Zähler 0...255 in einer Schleife. Mit `sm.put(i)` wird der Zählwert an die State Machine übergeben.



11. Ein genauerer Blick auf die OUT Instruktion

Da mir die Details der Sache noch nicht ganz klar war, habe ich einen Blick in das C SDK Dokument und ins RP2040 Datenblatt geworfen.

Dort ist die Struktur des OUT-Befehls erklärt:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUT	0	1	1	Delay/side-set					Destination			Bit count				

Die 3 höchstwertigsten Bits codieren den Befehl: OUT.
Die Bits 8 bis 12 stehen für Delay [n] und Sideset zur Verfügung. Hieraus ergibt sich auch die sonst schwer zu verstehende Einschränkung, dass für die beiden zusammen nur 32 Möglichkeiten zur Verfügung stehen, z.B. ergibt sich daraus:

- Delay [1] bis [31] ohne Sideset
- Sideset bis zu 5 GPIOs ohne Delay

Wie wird der Unterschied zwischen beiden gemacht ? Dies geschieht bei der Definition des PIO-Programms, z.B.

```
@rp2.asm_pio(sideset_init=PIO.OUT_LOW, sideset_count=3)
def my_program():
      . . .
```

Hier werden 3 Bits für sideset reserviert (3 GPIOs können gleichzeitig mit der OUT-Instruktion geschaltet werden), es bleiben also nur noch 2 für das Delay, also wären [1] , [2] oder [3] möglich, nicht aber [4] bis [31].

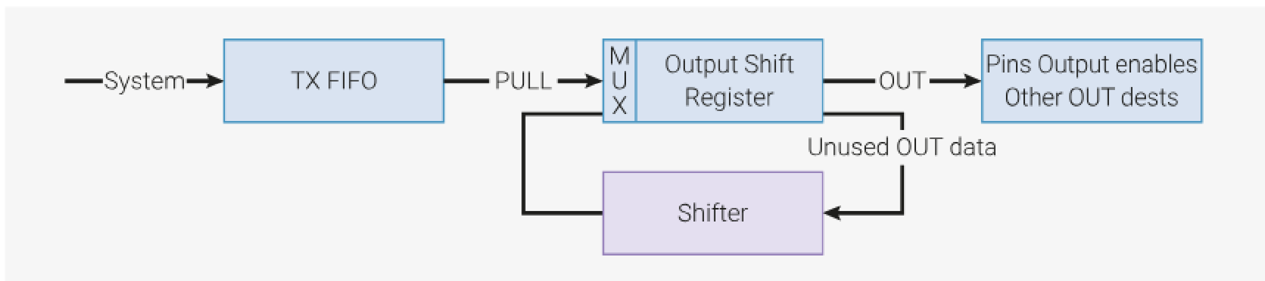
Diese Überlegungen gelten natürlich auch für andere Befehle.

Weiter geht es mit den Destination Bits. Dafür gibt es bei 3 Bits 8 verschiedeme Möglichkeiten, in unserem Fall PINS = 000:

`out(pins, 8)`
Die letzten 5 Bits sind für die Anzahl der auszugebenden Bits reserviert, hier 8 = 00100b.
Mit 5 Bits ergeben sich 32 Möglichkeiten, es können also Daten mit einer maximalen Breite von 32 Bit ausgegeben werden.

Wo kommen diese Datenbits her?

Im RP2040 Datenblatt findet man ein genaueres Blockschaltbild:



Mit `sm.put(i)` wird im Micropython-Teil des vorigen Beispiels ein Wert in den 32 Bit breiten TX FIFO geschoben.

Von dort wird er im PIO-Assembler-Teil mit dem Befehl `pull()` abgeholt und ins Output Shift Register OSR gesetzt.

Der Befehl `out(pins, 8)` schiebt hier 8 Bits aus dem Output Shift Register zur Destination, in unserem Fall pins. Die Destination wird bei der Instanziierung der State Machine festgelegt:

```
sm = StateMachine(0, parallel_out, freq = 10000000, out_base = Pin(0))
```
hier beginnend mit GPIO0.

In einem komplexeren Programm könnte eine weitere State Machine mit einem anderen Basispin festgelegt werden, die unabhängig von der ersten funktionieren würde. So würde man ein Verhalten ermöglichen wie es von anderen Mikrocontrollern bekannt ist bei denen man 8 Bit breite Ports mit einem Befehl als Ganzes setzen kann.

Die Anzahl der ausgegebenen Bits kann zwischen 1 und 32 liegen.
In unserem Beispiel wurde mit `out_shiftdir = PIO.SHIFT_RIGHT` die Schieberichtung von links nach rechts festgelegt. Dies bedeutet dass das rechts stehende Bit 0 als erstes (zusammen mit den anderen 7) zu den Pins herausgeschoben wird. Der Base Pin (hier GPIO0) erhält also Bit0, was Sinn macht.

In unserem Beispiel werden nur die 8 niederwertigsten der 32 Bit des OSR genutzt, die anderen sind 0. Es würde keinen Sinn machen, die Schieberichtung umzukehren, denn in diesem Fall würde man immer die führenden Null-Bits ausgeben.

Statt die Daten explizit mit `pull()` abzuholen, könnte auch autopull aktiviert werden, so dass die Daten automatisch abgeholt werden.

12. Parallelausgabe mit Data Ready Signal

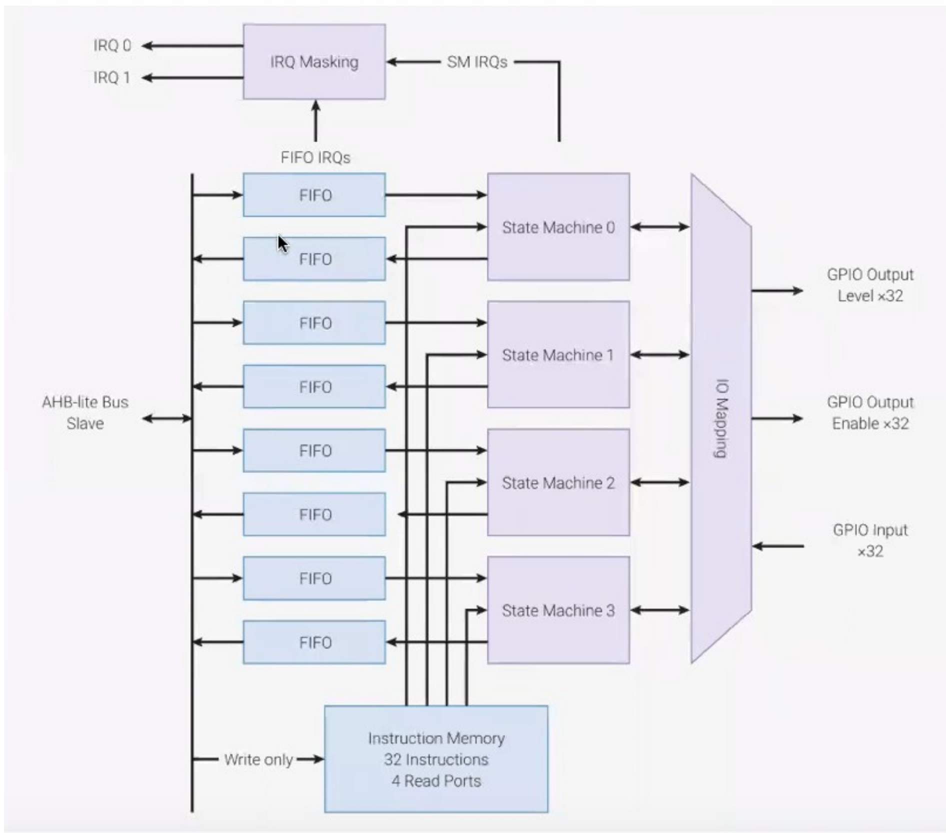
Für die Kommunikation mit einem anderen Controller benötigt man ein Signal welches mitteilt, wann die Daten zur Verfügung stehen. Mit der Sideset-Instruktruktion kann dieses gleichzeitig mit den Daten ausgegeben werden:

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

@asm_pio(sideset_init=PIO.OUT_LOW, out_init=(rp2.PIO.OUT_HIGH,) * 8,
out_shiftdir=PIO.SHIFT_RIGHT)
def parall():
    pull()          #.side(0)
    out(pins, 8)    .side(1)             # Set output pins and DR pin simultanuously
    nop()           .side(0)      [1]    # reset DR

sm = StateMachine(0, paral, freq=100000, sideset_base=Pin(16),out_base=Pin(0))
paral_sm.active(1)

while True:
    for i in range(500):
        paral_sm.put(i)
```

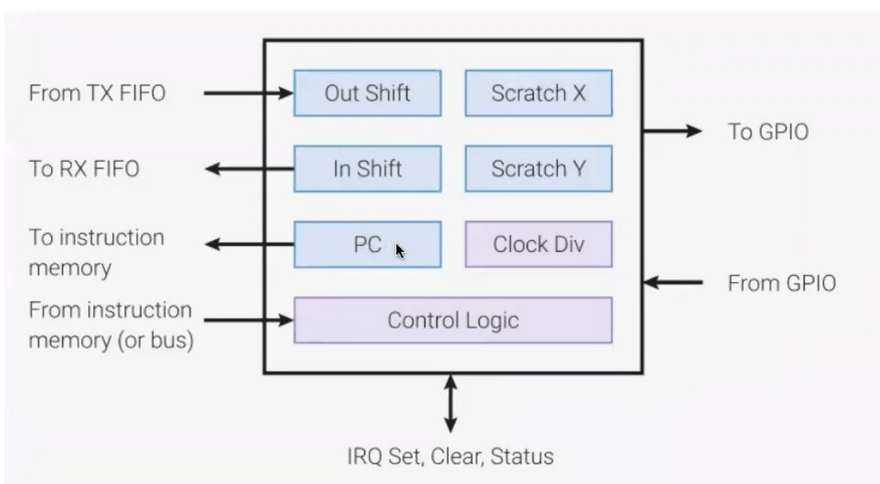
13. Anhang: Aufbau der PIO

2 Blöcke mit je 4 State Machines → 8 State Machines

1 Block:



1 State Machine:



Instruction set:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Delay/side-set					Condition			Address				
WAIT	0	0	1	Delay/side-set					Pol	Source		Index				
IN	0	1	0	Delay/side-set					Source			Bit count				
OUT	0	1	1	Delay/side-set					Destination			Bit count				
PUSH	1	0	0	Delay/side-set					0	IfF	Blk	0	0	0	0	0
PULL	1	0	0	Delay/side-set					1	IfE	Blk	0	0	0	0	0
MOV	1	0	1	Delay/side-set					Destination			Op		Source		
IRQ	1	1	0	Delay/side-set					0	Clr	Wait	Index				
SET	1	1	1	Delay/side-set					Destination			Data				