PIO State Machines

ECE 476 Advanced Embedded Systems

Jake Glower - Lecture 29

Please visit Bison Academy for corresponding
lecture notes, homework sets, and solutions

Introduction

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

PIO State Machines are a semi-unique feature of Pi-Picos

- These are independent processors which are able to drive the I/O pins.
- These are described on page 34 of the PI-Pico data sheets:

> *The PIO subsystem on RP-series microcontrollers allows you to write small, simple programs for what are called PIO state machines, of which RP2040 has eight split across two PIO instances,*

The intent is to make nonstandard communications more efficient

- NeoPixels
- HT11, HT22 TH sensors
- Essentially replacing bit-banging.

State Machines

Four state machines are available

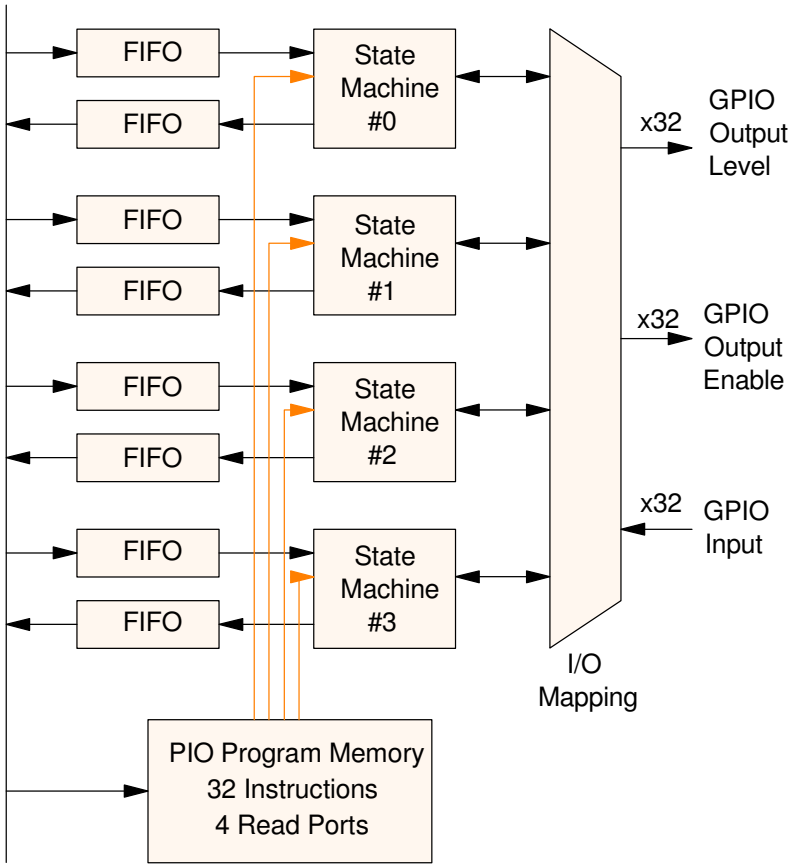
- Each can run a short program
- Limit = 32 instructions
- Reduced assembler instruction set

Clock range

- min = 2kHz
- max = 125MHz

Each PIO State Machines can

- Read from I/O pins
- Write to I/O pins
- Pass data to/from the Pico via FIFO buffers



State Machine Instructions

https://www.seeedstudio.com/blog/2021/01/25/programmable-io-with-raspberry-pi-pico/

The PIO State Machines use a limited set of assembler instructions.

```
IN – Shifts in one word of 32 bits into the ISR
OUT – Shifts out one word of 32 bits from the OSR to another location
PUSH – Sends data to the RX (input) FIFO
PULL – Gets data from the TX (output) FIFO
MOV – Moves data from one location to another
IRQ – Sets or clears interrupt flag
SET – Writes data to destination
WAIT – Pauses until a defined action happens
JMP – Jumps to a different point in the code
```

This doesn't seem like a lot,

- It's enough to program some fairly elaborate I/O functions
  - CAN
  - NeoPixel
  - HT11
  - etc

Topics for This Lecture:

- Output a 1kHz square wave
  - basic program
- Output a 1Hz square wave
  - looping
- Output a 2kHz square wave with variable duty cycle
  - multiple PIO functions
- Generate a pulse with N bounces on the rising edge
  - passing data to a PIO function
- Driving a NeoPixel
  - generating nonstandard output signals
  - with precise timing

Output a 1kHz Square Wave

https://dev.to/blues/a-practical-look-at-pio-on-the-raspberry-pi-pico-50j8

Starting out, let's make a light blink at 1kHz:

```
import time
import rp2
from machine import Pin

def blink():
    set(pins, 1)
    set(pins, 0)
    wrap()

@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
sm = rp2.StateMachine(0, blink, freq=2000, set_base=Pin(16))

sm.active(1)
time.sleep(3)
sm.active(0)
```

Explaining this program:

```
def blink():
    set(pins, 1)
    set(pins, 0)
    wrap()
```

Assembler subroutine which runs on the state-machine.

This program

- Sets the I/O pin
- Clears the I/O pin, then
- The program repeats (wrap())

Set up the PIO State Machine

```
@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
sm = rp2.StateMachine(0, blink, freq=2000, set_base=Pin(16))
```

This defines state machine #0

- The routine called is *blink*
- The state machine operates with a 2000Hz clock frequency
  - range is 2kHz to 125MHz (!)
- The output pin used is GP16

When initializing the state-machine, you can also specify input pins, input shift direction, output shift direction, and other parameters. Please visit MicroPython for a more detailed explanation

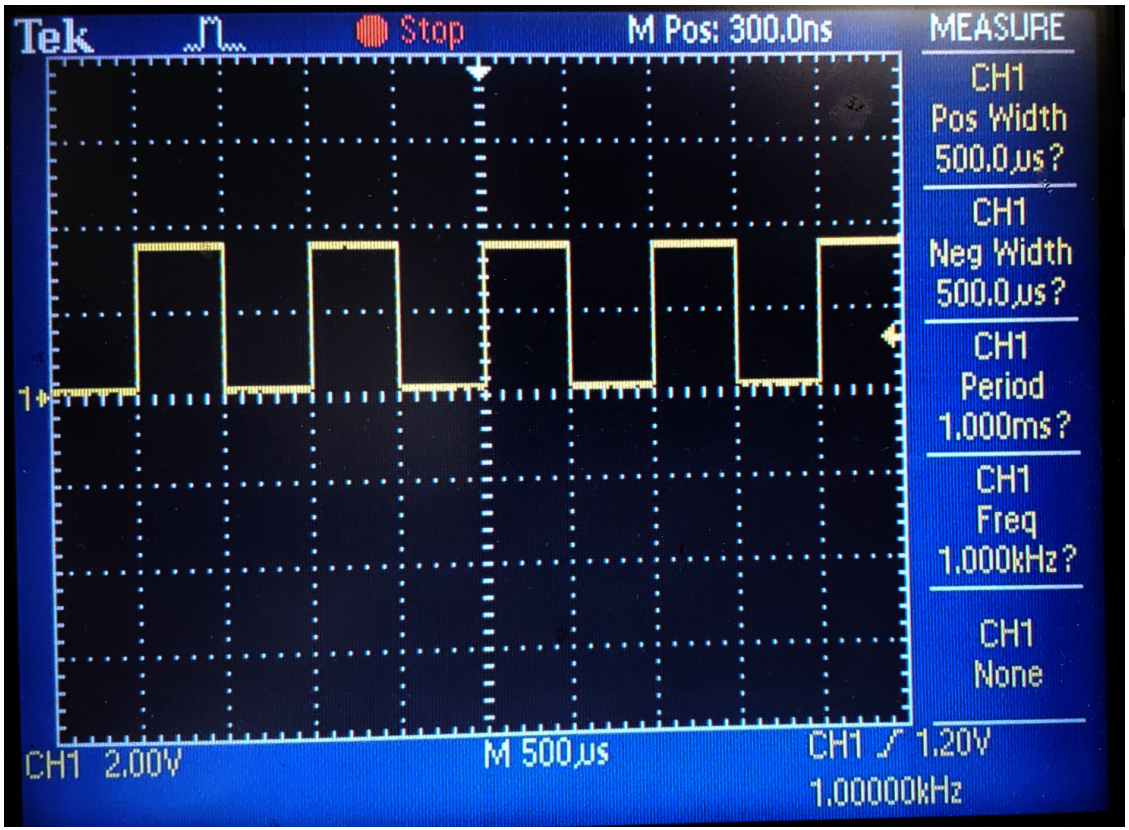
- https://docs.micropython.org/en/latest/library/rp2.StateMachine.html

The last set of commands:

```
sm.active(1)
time.sleep(3)
sm.active(0)
```

turns on (activates) the state-machine for three seconds.

- The result is a 1kHz square wave



A slower square wave can be produced by adding wait states:

- [31] adds 31 wait states (essentially nop() commands)
- range is [1] to [31]
  - 6 bits

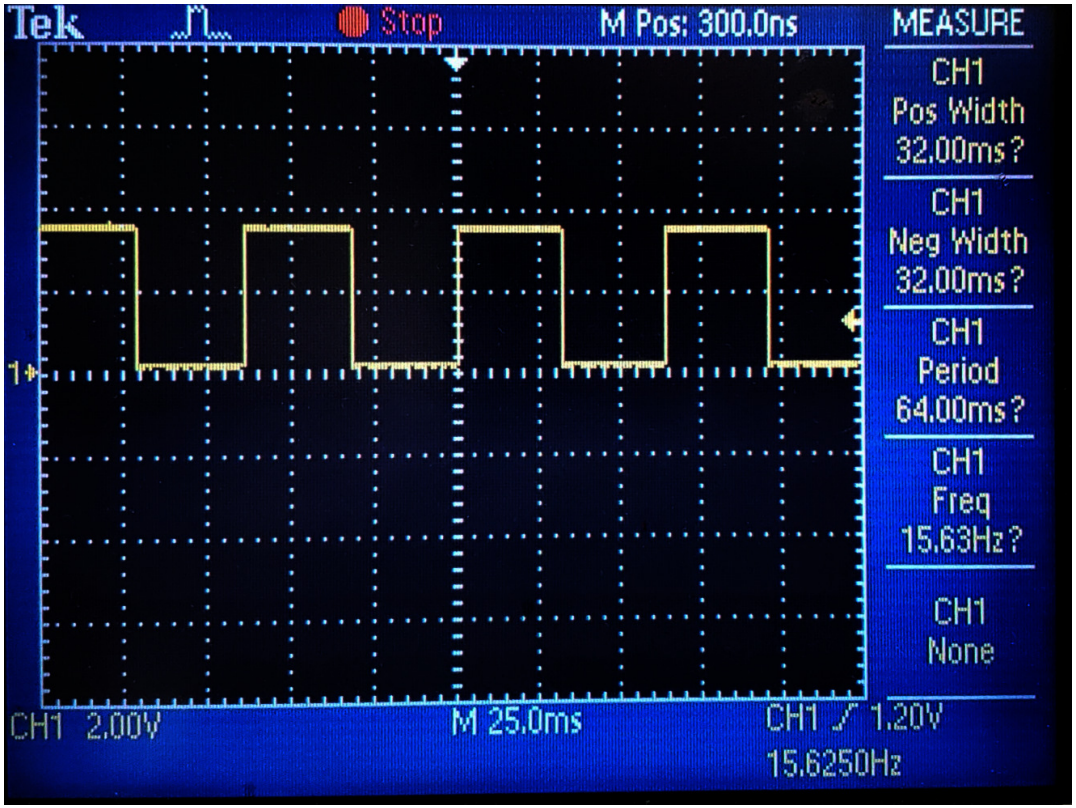
```
import time
import rp2
from machine import Pin


def blink():
    set(pins, 1)   [31]
    nop()          [31]
    set(pins, 0)   [31]
    nop()          [31]
    wrap()

@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
sm = rp2.StateMachine(0, blink, freq=2000, set_base=Pin(16))

sm.active(1)
time.sleep(3)
sm.active(0)
```

Result is a 64ms square wave

- Each loop has four instructions,
- Plus 31x4 nops inserted

The period is thus

$$T = 32 \cdot 4 \cdot 0.5ms = 64ms$$



Output a 1Hz Square Wave

Looping can be accomplished by adding counters and labels.

set output pin
move 10 to register x

move 10 to regiser y

wait seven clocks
decrement y. jump to "loop_1" if not zero
decrement x. jump to "loop_0" if not zero

clear output pin
move 10 to register x

move 10 to regiser y

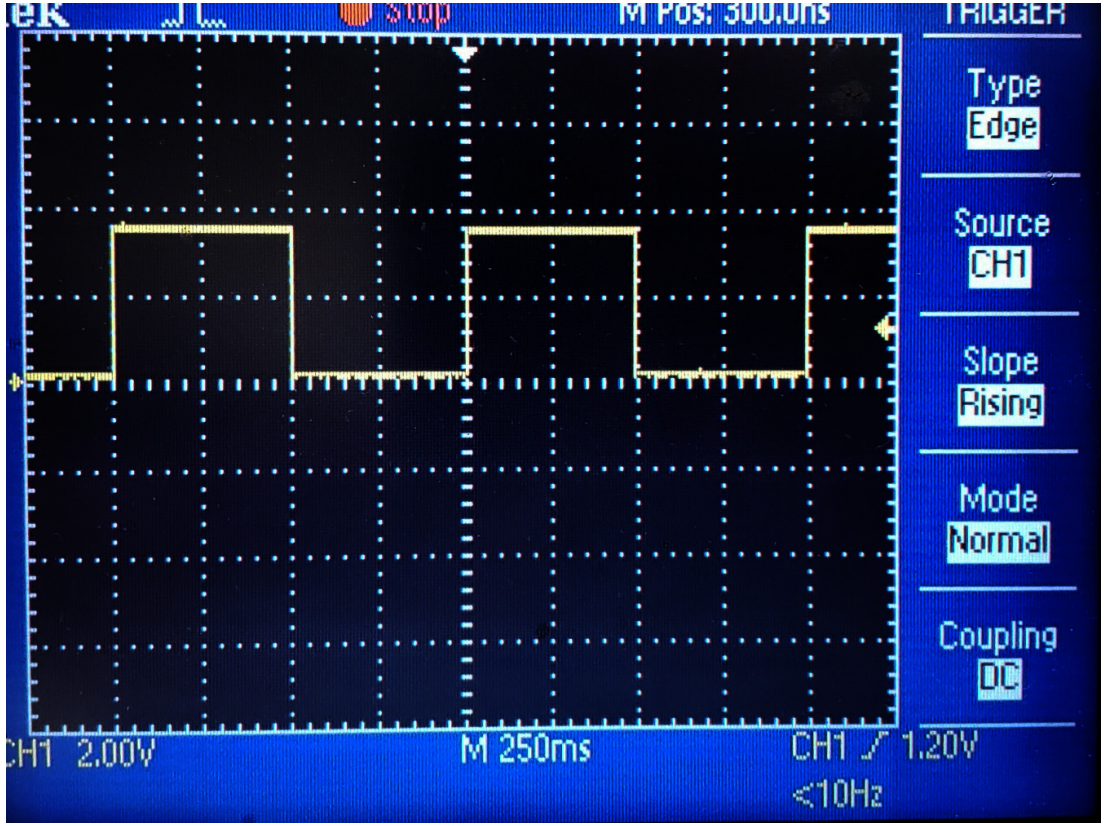
wait seven clocks
decrement y. jump to "loop_2" if not zero
decrement x. jump to "loop_3" if not zero

repeat

```
@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink_1Hz():
    set(pins, 1)
    set(x, 10)

    label("loop_0")
    set(y, 10)
    label("loop_1")
    nop().delay(6)
    jmp(y_dec, "loop_1")
    jmp(x_dec, "loop_0")

    set(pins, 0)
    set(x, 10)

    label("loop_2")
    set(y, 10)
    label("loop_3")
    nop().delay(6)
    jmp(y_dec, "loop_3")
    jmp(x_dec, "loop_2")

    wrap()
```

With this program

- You loop 100 times (x=10, y=10, keep looping)
- Each loop takes ten clocks
  - nop.delay(6) takes seven clocks
  - plus three for the jump instruction and label
- For a total of 1000 clocks (500ms) high, 1000 clocks (500ms) low



Aliasing & Two PIO State Machines

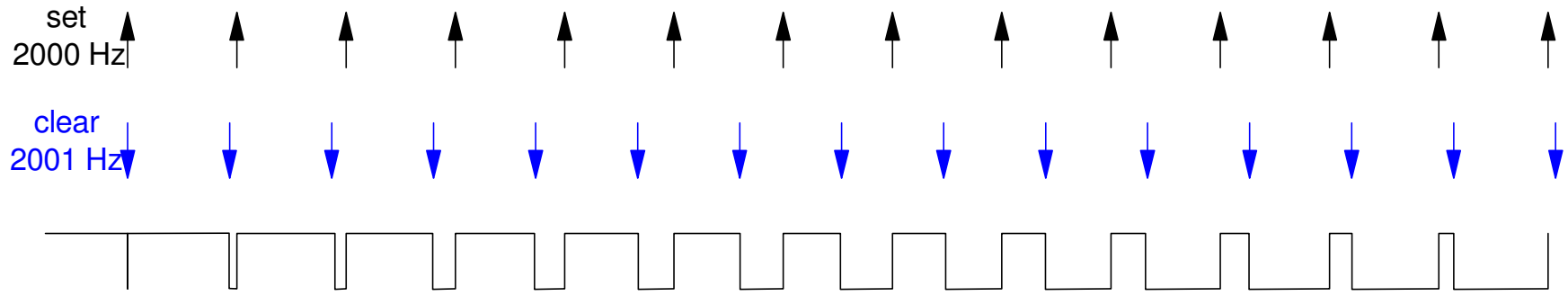
The Pico has four PIO State Machines

- These can run separate programs
- These can run with different clocks

Example: Turn on two PIO state machines

- led_off(): Called every 2000 Hz
- led_on(): Called every 2001 Hz

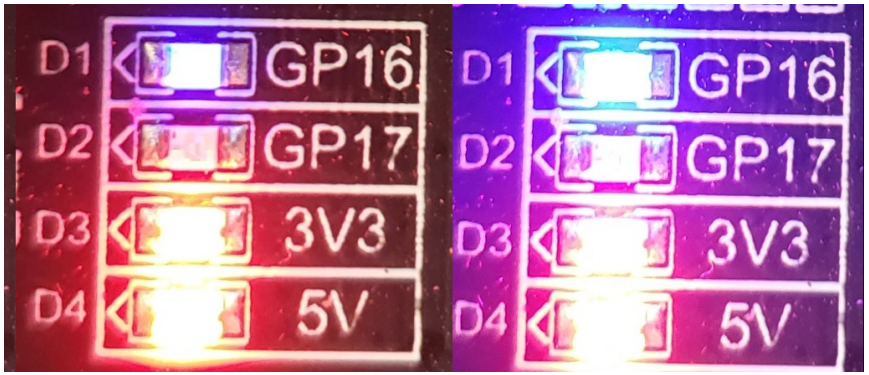
This produces a variable duty cycle (beat frequency = 1Hz)



Code:

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

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

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

sm1 = StateMachine(1, led_off, freq=2000, set_base=Pin(16))
sm2 = StateMachine(2, led_on, freq=2001, set_base=Pin(16))

sm1.active(1)
sm2.active(1)
```

https://www.seeedstudio.com/blog/2021/01/25/programmable-io-with-raspberry-pi-pico/



One-Time Programs - BlinkN

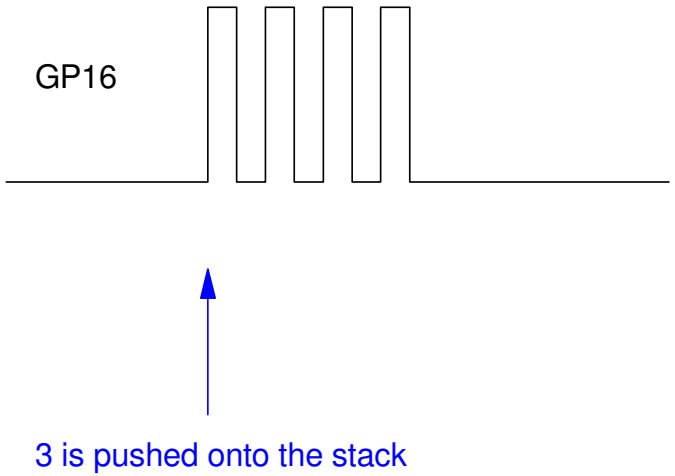
https://dev.to/blues/a-practical-look-at-pio-on-the-raspberry-pi-pico-50j8

PIO programs can be triggered by the main routine

- Triggered when data is pushed onto the stack
  - *put()* executed
- Also allows data to be passed to the PIO program

Example: BlinkN

- Output N+1 pulses when triggered
- Each pulse is 1ms
- *put(3)* generates four pulses



Code:

Program starts with *sm.put(3)*

- Puts the number 3 on the stack
- Triggers the PIO code

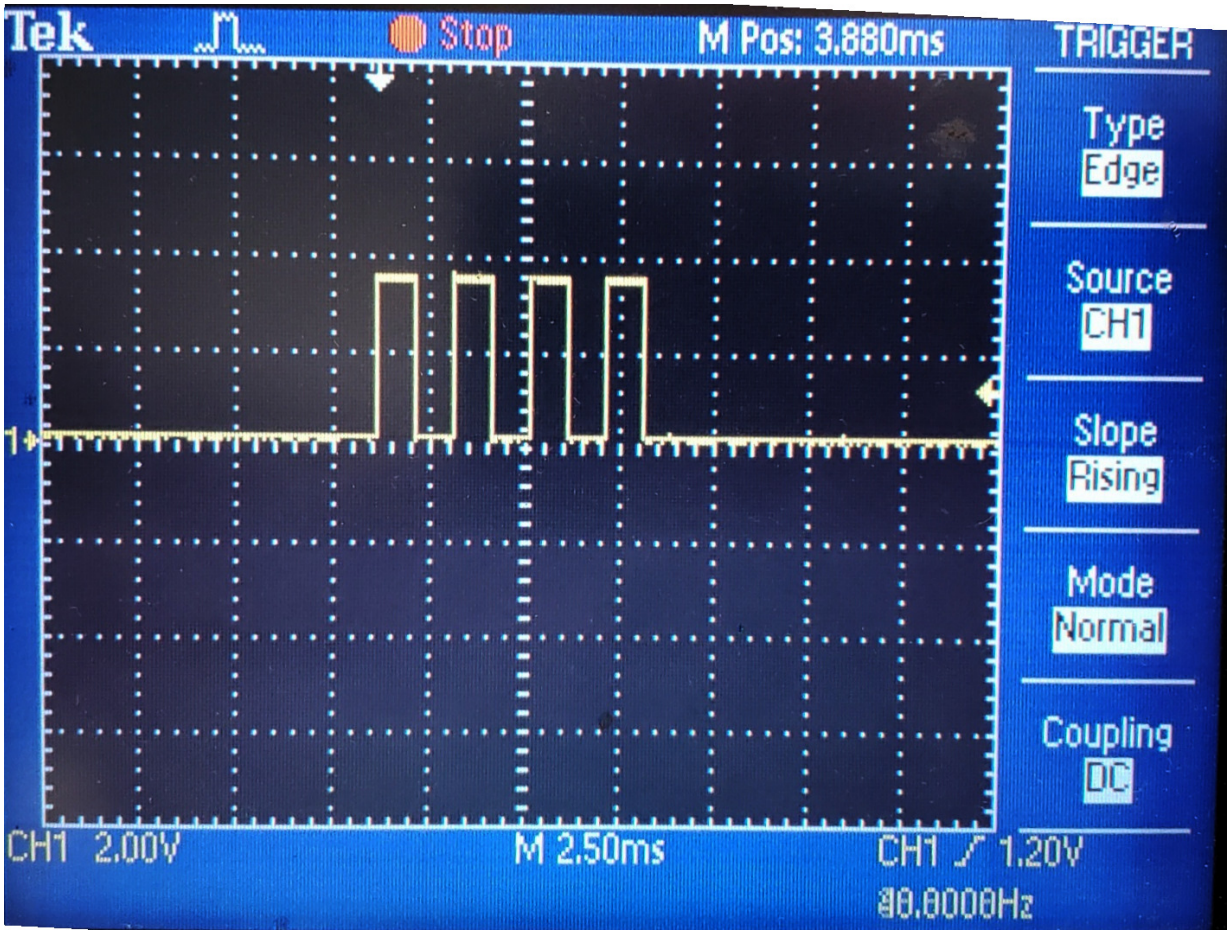
BlinkN():

- Pulls number off the stack
  - Placed in osr
- Moves *osr* to register x
  - Counter for following loop
- Pulses GP16 high then low
  - 2 clocks = 1ms
- Decrements x
  - Loop back if x >= 0
  - Exit if x < 0

```
import time
import rp2
from machine import Pin

@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)

def BlinkN():
    pull()
    mov(x, osr)
    jmp(not x, "loop_end")
    label("loop_2")
    set(pins, 1).delay(1)
    set(pins, 0)
    jmp(x_dec, "loop_2")
    label("loop_end")

sm = rp2.StateMachine(0, BlinkN,
freq=2_000, set_base=Pin(16))

sm.active(1)

while(1):
    sm.put(3)
    time.sleep(0.1)
```

Result: Blink(N)

- *sm.put(3)* command
- Creates four pulses
- Each 1ms wide (2 clocks)



One-Time Program: Bouncing

https://dev.to/blues/a-practical-look-at-pio-on-the-raspberry-pi-pico-50j8

Multiple PIO State Machines are available

- Pico has four

Each can run a separate program

Example: Create an output with N bounces on the rising edge

Use two PIO State Machines

- sm0: Sets GP16 after N+1 bounces
- sm1: Clears GP16

Calling Sequence for 5+1 bounces:

```
sm0.put(5)
sm1.put(1)
```

State Machine 1 Code:

State-machine 1 is fairly simple:

- It pulls the data off the state to clear the stack, and then
- Clears GP16

```
@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def clear_pin():
    pull()
    set(pins, 0)
```

State Machine 0 Code

State-machine 0 is a little more complicated:

- It first pulls the number pushed off the stack, storing it in the osr register
- This value is then moved to register x, telling the state-machine how many times to bounce
- The pins are then set and cleared (one bounce)
- Counter x is then decremented, and
- The bouncing continues until x is decremented past zero
- Once bouncing is completed, the GPIO pin is set

```
@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def set_pin():
    pull()
    mov(x, osr)
    label("loop")
    set(pins, 1).delay(1)
    set(pins, 0)
    jmp(x_dec, "loop")
    set(pins, 1)
```

Overall Code

set_pin():

- Pulls number off the stack
- Bounces N+1 times
- Ends with GP16 set

clear_pin():

- Pulls data off the stack
  - clear the stack
- Sets GP16

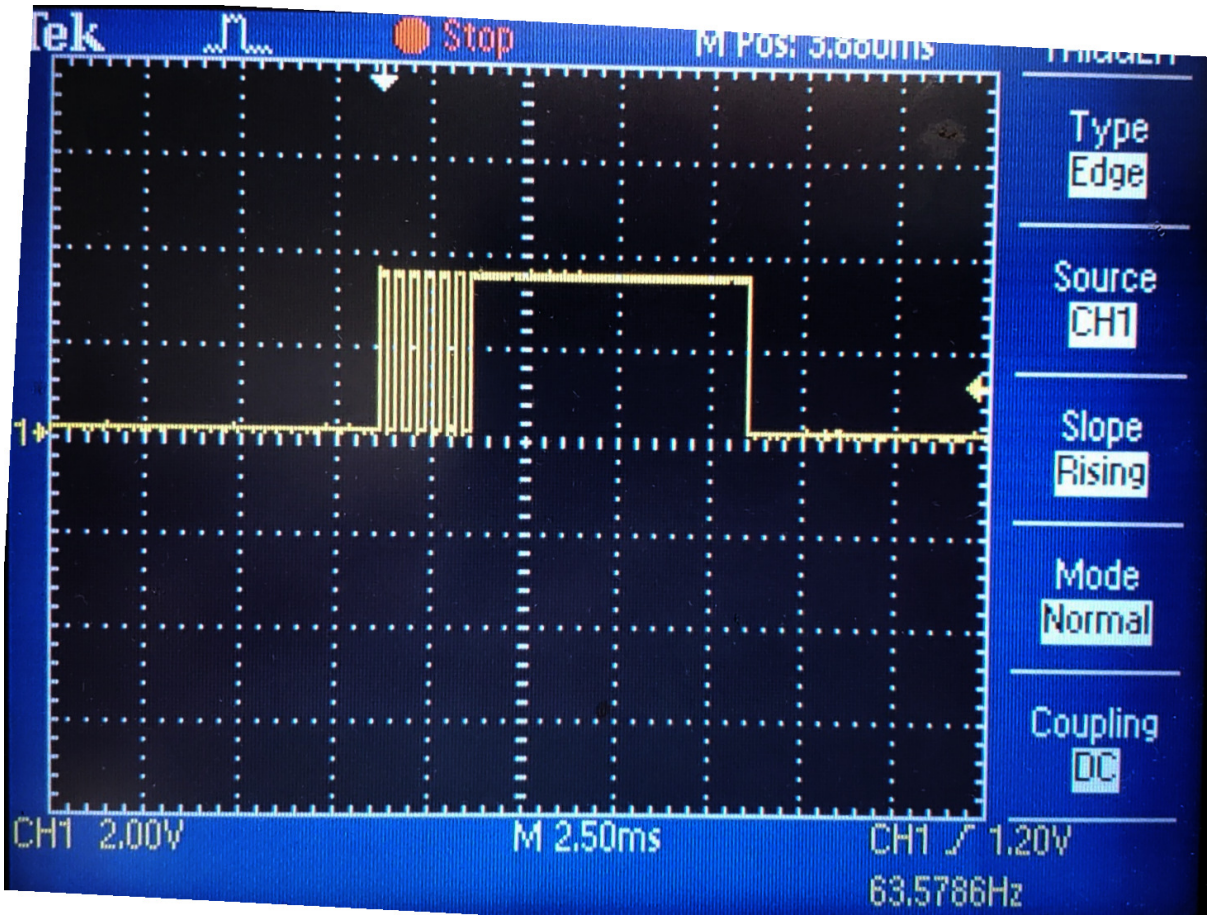
Main Loop

- Set with 5+1 bounces
- Wait 10ms
- Then clear

```
import time
import rp2
from machine import Pin

@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def set_pin():
    pull()
    mov(x, osr)
    label("loop")
    set(pins, 1).delay(1)
    set(pins, 0)
    jmp(x_dec, "loop")
    set(pins, 1)

@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def clear_pin():
    pull()
    set(pins, 0)

sm0 = rp2.StateMachine(0, set_pin,
freq=10_000, set_base=Pin(16))
sm1 = rp2.StateMachine(1, clear_pin,
freq=10_000, set_base=Pin(16))

sm0.active(1)
sm1.active(1)

while(1):
    sm0.put(5)
    time.sleep(0.01)
    sm1.put(1)
    time.sleep(0.1)
```

Bouncing Results

- Bounces N+1 times (6)
- Total time = 10ms
  - Main routine isn' t locked up while the PIO code runs



NeoPixels & State Machines

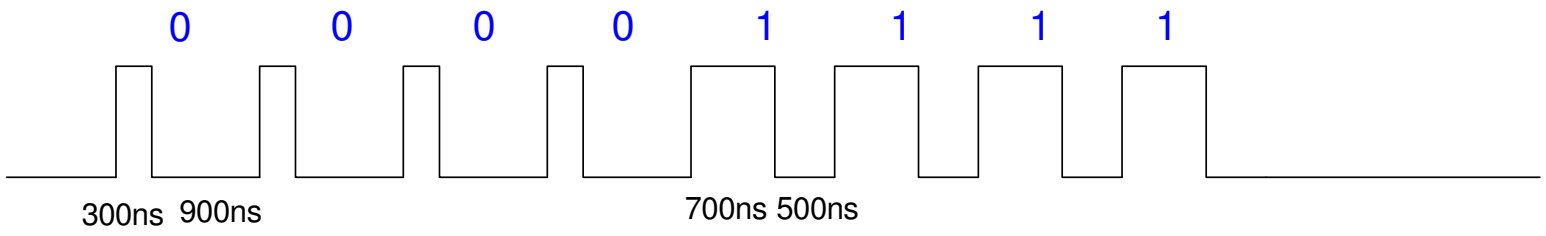
https://www.instructables.com/Respberry-Pi-Pico-W-NeoPixels-Experiments-With-Pro/

Finally, let's drive a NeoPixel using a state-machine.

- Really what PIO State Machines are designed for
- Provide non-standard I/O with critical timing
- Without tying up the CPU with bit-banging

NeoPixel Timing:

- Logic 1: 700ns pulse +/- 120ns
- Logic 0: 300ns pulse +/- 120ns
- Each Bit: 1200ns +/- 120ns



Step 1: Send GRB data to the PIO routine

- Send a single GRB message
- Total length = 24 bits
- Pass as a 32-bit number
  - Left justified
  - Shift left to get each following bit

Code:

```
g = 50
r = 100
b = 150
grb = (g << 16) + (r << 8) + b
sm.put(grb << 8)
```

Step 2: State-Machine takes over

- 32-bit word is pulled
  - stored in osr
- x is set to 23
  - count 24 bits
- move msb of osr to y
  - shift left
- Output a 1 or 0
  - 1 = 700ms high, 500ms low
  - 0 = 300ms high, 900ms low
- Decrement x
  - go onto the next bit

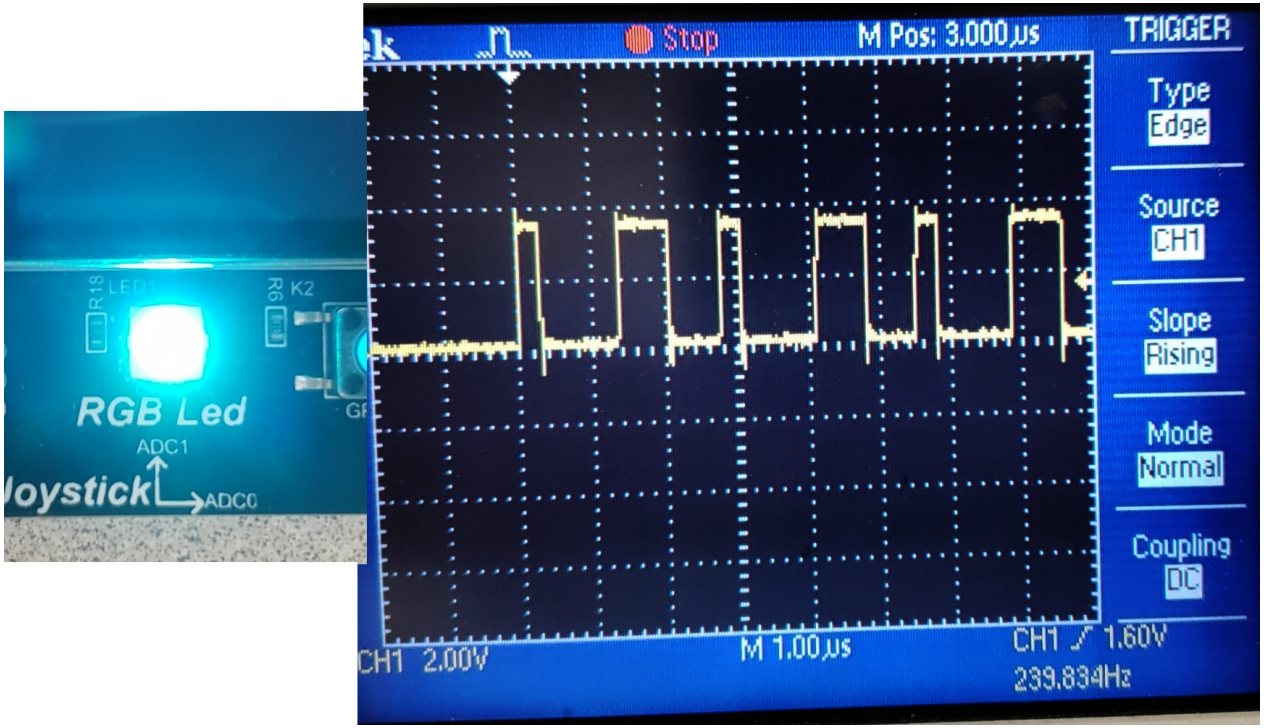
```
def neo_prog():
    pull()
    set(x, 23)

    label("loop_pixel_bit")

    out(y, 1)

    jmp(not_y, "bit_0")
    set(pins, 1).delay(13)
    set(pins, 0).delay(9)
    jmp("bit_end")
    label("bit_0")
    set(pins, 1).delay(5)
    set(pins, 0).delay(17)

    label("bit_end")
    jmp(x_dec, "loop_pixel_bit")
```

https://www.instructables.com/Respberry-Pi-Pico-W-NeoPixels-Experiments-With-Pro/

Calling Routine

- Single NeoPixel
- Teal light

```
sm = rp2.StateMachine(0, neo_prog,
freq=20_000_000, set_base=Pin(12))
sm.active(1)

g = 0x55
r = 0x0F
b = 0x1F
grb = (g << 16) + (r << 8) + b
while(1):
    sm.put(grb << 8)
    time.sleep(0.1)
```



Talking to N NeoPixels

https://www.instructables.com/Respberry-Pi-Pico-W-NeoPixels-Experiments-With-Pro/

Calling Sequence

- Push the number of NeoPixels on the stack, then
- Push the GRB values of each NeoPixel onto the stack

Example: 16 NeoPixels

```
x = 0
N = 16
while(1):
    x = (x + 1) % 256
    sm.put(N-1)
    for i in range(0,N):
        g = 0
        r = i*10
        b = 160 - r
        grb = (g << 16) + (r << 8) + b
        sm.put(grb << 8)
    time.sleep(0.05)
```

State-Machine:

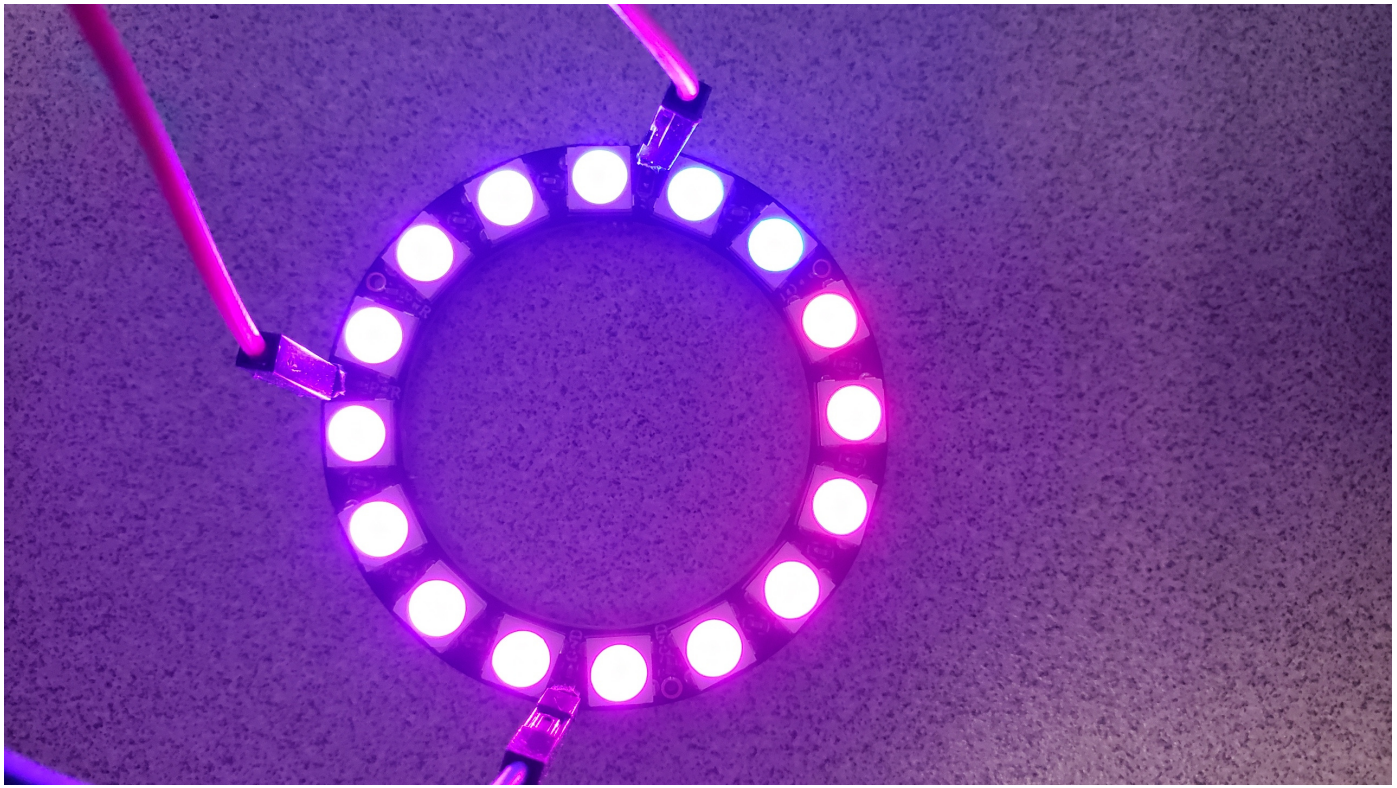
https://www.instructables.com/Respberry-Pi-Pico-W-NeoPixels-Experiments-With-Pro/

- *neo_prog():*
- State machine that drives N NeoPixels

```
def neo_prog():
    pull()                          # osr <= number of pixels - 1
    mov(y, osr)                     # y <= number of pixels - 1
    label("loop_pixel")
    mov(isr, y)                     # isr (pixel counter) <= y
    pull()                          # osr <= 24 bits GRB
    set(x, 23)                      # x (bit counter) <= 23
    label("loop_pixel_bit")
    out(y, 1)                       # y <= left-most 1 bit of osr
    jmp(not_y, "bit_0")
    set(pins, 1).delay(13)          # 1: high (7 cycles)
    set(pins, 0).delay(9)           # 1: low (5 cycles)
    jmp("bit_end")
    label("bit_0")
    set(pins, 1).delay(5)           # 0: high (3 cycles)
    set(pins, 0).delay(17)          # 0: low (9 cycles)
    label("bit_end")
    jmp(x_dec, "loop_pixel_bit") # x is bit counter
    mov(y, isr)                     # y <= isr (pixel counter)
    jmp(y_dec, "loop_pixel")     # y is pixel counter
```

Result:

- Drives all 16 NeoPixels
- Each can be controlled independently
- Updated each time you push data on the sm0 stack



Summary

- PIO State Machines are a fairly unique feature of the Raspberry Pi Pico

With state machines

- You are able to drive devices which use nonstandard interfaces
- Without having to resort to bit-banging.
- While driving the device, the main routine is free to do other stuff

This can improve the efficiency of code running on a Pi-Pico.

References

- https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf
- https://www.seeedstudio.com/blog/2021/01/25/programmable-io-with-raspberry-pi-pico/
- https://dev.to/blues/a-practical-look-at-pio-on-the-raspberry-pi-pico-50j8
- https://www.instructables.com/Respberry-Pi-Pico-W-NeoPixels-Experiments-With-Pro/