29 PIO State Machines

Introduction

A semi-unique feature of the Raspberry Pi Pico is the existence of state machines. 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:

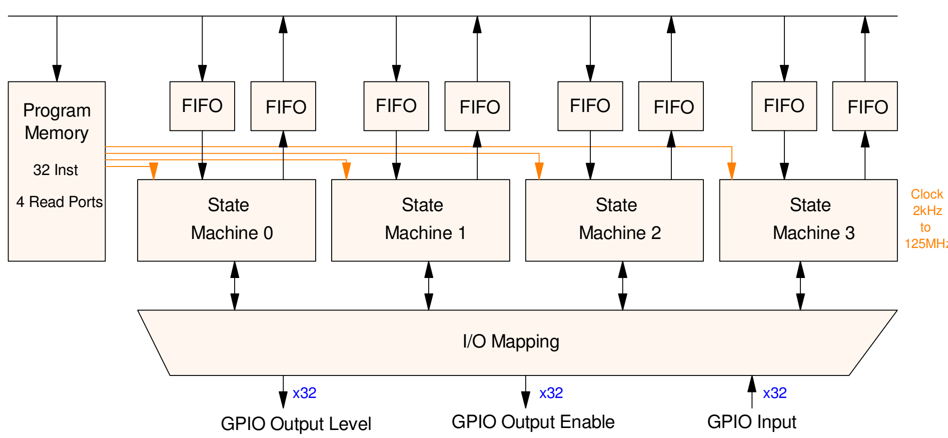
https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf

> *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 of the PIO State Machines is to make nonstandard communications more efficient - essentially replacing bit-banging. Some forms of communications are already supported by the Pi-Pico, including UART, SPI, and I2C. Other forms of communications exist, such as NeoPixels and CAN to name a few. By using PI State Machines, you can create more efficient ways of using these protocols.

State Machines

The PIO State Machines are mini-microcontrollers capable of running a short program autonomously from the rest of the Pi-Pico. Four such state machines are available, each one able to run a short program which is limited to 32 instructions, running with a clock frequency ranging from 2kHz to 125MHz. Each state machine is capable of reading and/or writing to any of the GPIO pins. In addition, each state machine is capable of sending and receiving data from the Pi-Pico through a set of FIFO (first-in, first-out) buffers.



Four GPIO State Machines are available on the Pi-Pico.

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 1 word of 32 bits at a time into the ISR from another location
OUT - Shifts 1 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, but it's enough to program some fairly elaborate I/O functions such as a CAN bus. In this lecture, we'll look at using the PIO State Machines to do a few functions:

- 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), and
- Driving a NeoPixel (generating nonstandard output signals)

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()
```

This is the 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())

```
@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

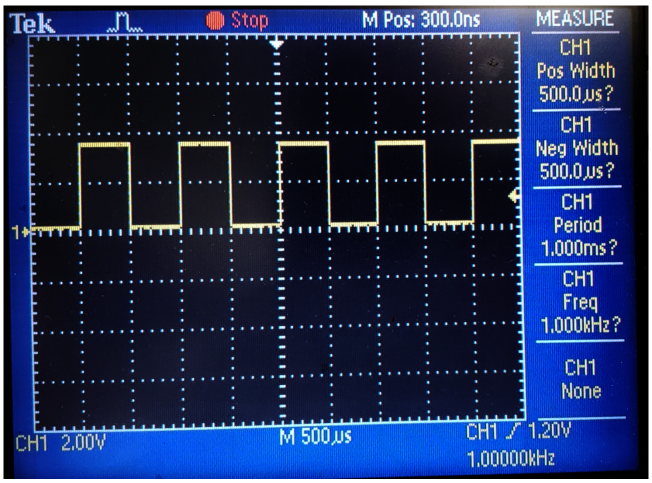
When initializing the state-machine, you can also specify input pins, input shift direction, output shift direction, and other parameters. Please visit MicroPython for a more detailed explanation

- 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



1kHz Square Wave Generated with a State Machine

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

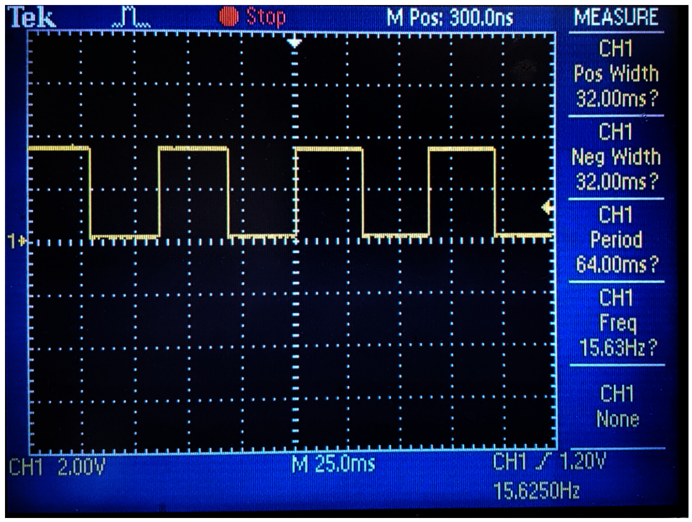
```
import time
import rp2
from machine import Pin


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)
```

In this program, the [31] tells the state-machine to insert 31 nop commands after each instruction (including the nop()). The results is a 15Hz square wave

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

The period is thus

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



Square Wave with a 64ms Period - Generated with a State Machine

Note that the maximum number of nop statements you can insert is 31

- [31] is allowed
- [32] is not (too large)

Output a 1Hz Square Wave

Looping can be accomplished by adding counters and labels.

```
import time
import rp2
from machine import Pin

@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()

sm = rp2.StateMachine(0, blink_1Hz, freq=2_000, set_base=Pin(16))

sm.active(1)
time.sleep(10)
sm.active(0)
```

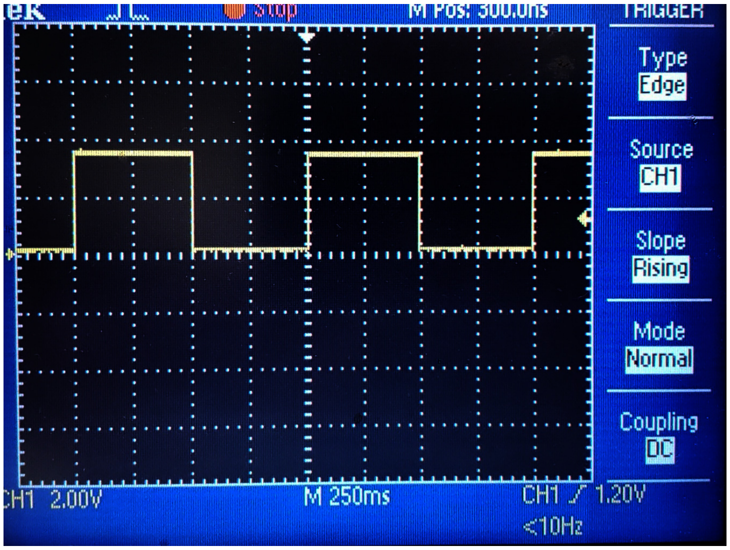
Code for generating a 1Hz square wave

These commands do the following:

```
set(x, 10)          move the number 10 to register x.
                    Valid registers are: x, y, isr, osr
                    Uses 6-bit numbers (range = 0 to 31)

label("loop_0")     Define a label for future jump commands

jmp(y_dec, "loop_1")    decrement y
                        if y >= 0, jump to label "loop_1"
                        otherwise skip the jump and goto the next line
```

With this program

- You set pin 16
- 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



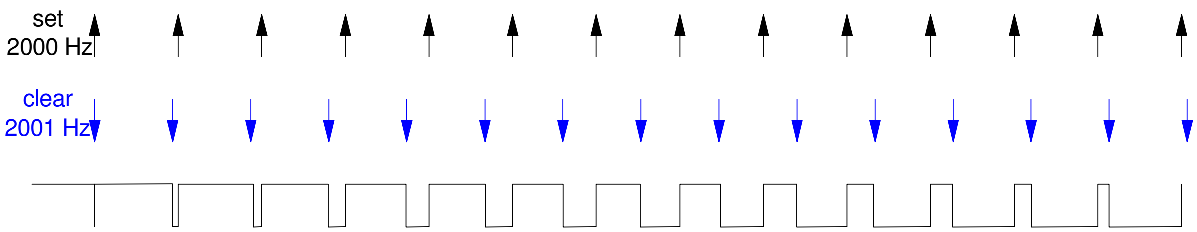
By adding loops, a 1Hz square wave can be produced using State Machines

Aliasing & Two PIO State Machines

Multiple PIO State Machines can be turned on at the same time. For example, the following program turns on two state machines:

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

The result is a variable duty cycle output on GP16 with a period equal to the difference in frequency



A variable duty cycle can be created by using two state machines running ad different clock rates

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/

GP16 gets brighter with a 1Hz beat frequency

One-Time Programs - BlinkN

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

The previous programs ran over and over when the state-machine is activated. In order to run the state machine one time, push data onto the corresponding stack

For example, the following state-machine sends out N+1 pulses where N is the number pushed onto the stack.

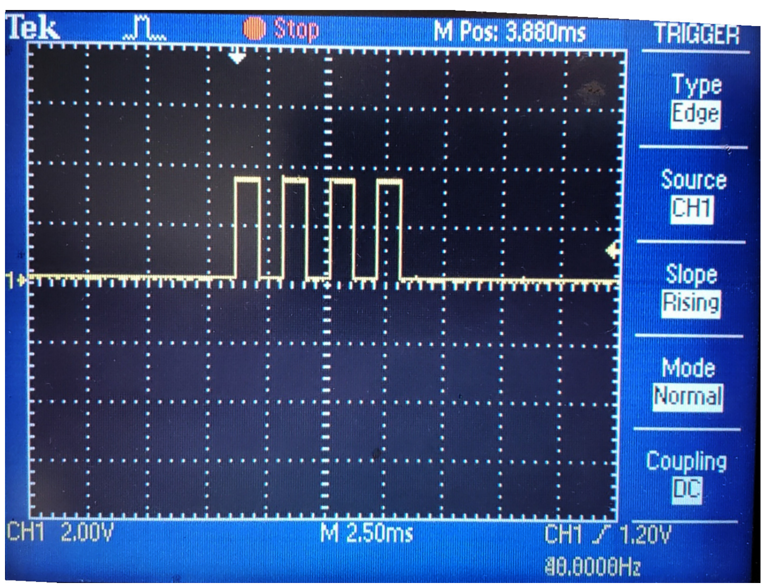
- Data is pulled from the stack using a *pull()* command
- The data is then read from the *osr* register

From that point onward,

- pin is set for two clocks (1ms)
- pin is then cleared for one clock (0.5ms)
- x is decremented and
- the process repeats as long as 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)
```

Blink N+1 times each time sm.put(N) is executed



One-Shot Outputs using State-Machines: four pulses are output each time sm.put(3) is executed

One-Time Program: Bouncing

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

Bouncing is a problem often encountered with mechanical switches. In order to test ways of removing the effects of bouncing, it would help to have a reliable input which bounces a controlled number of times. The following program generates a fixed number of bounces each time a pulse is sent out.

The following program uses two PIO programs:

- sm0: Sets GP16 after N+1 bounces

- sm1: Clears GP16

Instead of each program running over and over as in the previous examples, each program is called only once when a number is pushed onto its corresponding stack

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

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 code for clearing a pin

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)
```

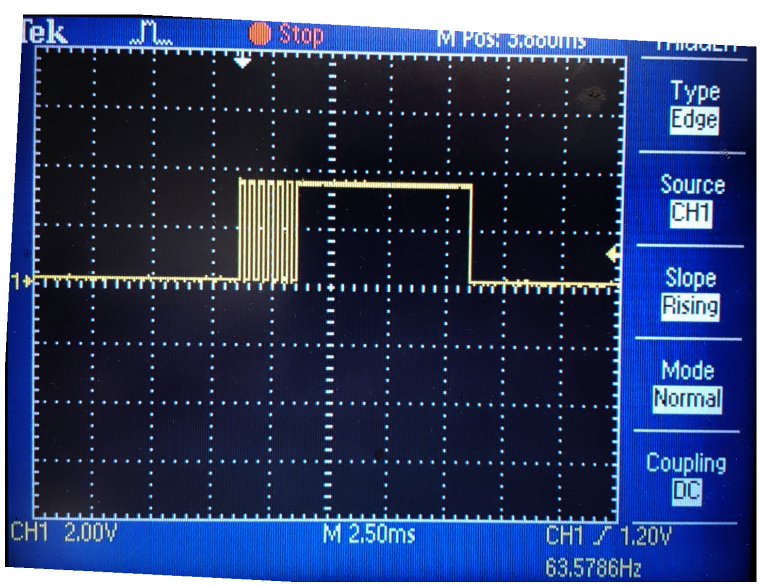
state-machine code for bouncing N+1 times then setting a pin

The overall program is as follows:

```
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)
```

Overall code for bouncing N+1 times when setting a pin

The net result is a pulse with is six bounces every 0.1 second. Note that state-machine(0) starts to execute as soon as you execute the *sm0.put(5) command.* The 10ms delay starts counting from the time that sm0 is called rather than from the time it finishes.



Calling two state-machines: One generates seven bounces then sets, the other clears

NeoPixels & State Machines

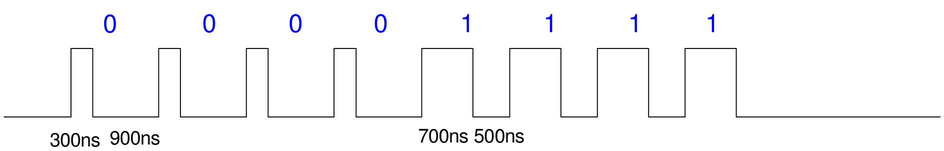
https://www.instructables.com/Respberry-Pi-Pico-W-NeoPixels-Experiments-With-Pro/

Finally, let's drive a NeoPixel using a state-machine. This is really what state-machines are designed for: driving I/O pins for devices with timing-critical nonstandard interfaces.

Recall that the timing for a NeoPixel is a bit unusual. Data is sent to NeoPixels in 24-bit chunks. 1's and 0's are sent as pulse durations:

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

For example, the signal 0x0F would look like the following



Timing for NeoPixels

This is a nonstandard format: it doesn't follow I2C, SPI, or UART protocol. With State-Machines, a Pico can handle this with no problem.

The following program from *instrutables.com* drives the NeoPixel in several steps.

First, drive a single neopixel. The GRB value is pushed onto the stack as a left-justified 32-bit number:

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

Once data is pushed onto the stack, the state-machine takes over and runs at 20MHz

- Logic 0 = 300ns (6 clocks)
- Logic 1 = 700ns (14 clocks)
- Each bit = 1.2ms (24 clocks)

In the program *neo_prog()*

- The 32-bit value of g/r/b is pulled off the stack and stored in the osr (pull())
- A counter, x, is set to 23 to count 24 bits
- The most-significant bit of the osr moved to y
  - If it was a 1, output a 1 for 14 clocks then a 0 for 10 clocks
  - If it was a 0, output a 1 for 6 clocks than a 0 for 18 clocks
- Decrement counter x and repeat 24 times

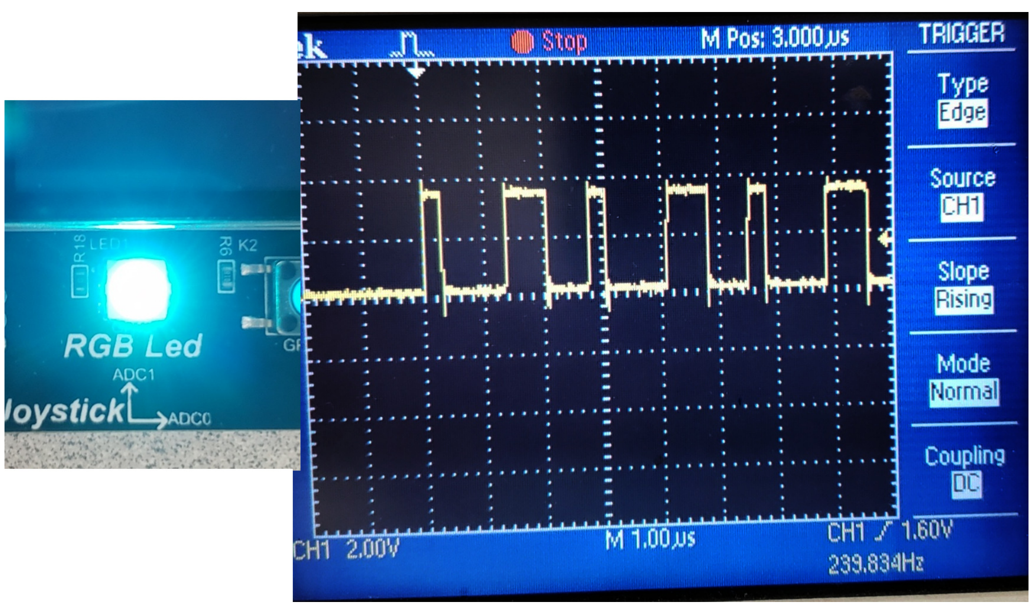
```
def neo_prog():
    pull()                        # osr <= 24 bits GRB
    set(x, 23)                    # x (bit counter) <= 23
    label("loop_pixel_bit")
    out(y, 1)                     # y <= left-most 1 bit of osr
    jmp(not_y, "bit_0")
    set(pins, 1).delay(13)        # 1: high (700ns)
    set(pins, 0).delay(9)         # 1: low (500ns)
    jmp("bit_end")
    label("bit_0")
    set(pins, 1).delay(5)         # 0: high (300ns)
    set(pins, 0).delay(17)        # 0: low (900ns)
    label("bit_end")
    jmp(x_dec, "loop_pixel_bit") # x is bit counter
```

State-machine code for driving a single NeoPixel
https://www.instructables.com/Respberry-Pi-Pico-W-NeoPixels-Experiments-With-Pro/

The calling routine is then:

```
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)
```

Calling sequence for driving a single NeoPixel



NeoPixel output and signals showing 010101... data

A more general routine which can talk to multiple NeoPixels comes from

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

This routine has you

- First push the number of NeoPixels on the stack, then
- Push the GRB values of each NeoPixel onto the stack as a left-justified 32-bit number

For example, if talking to 16 NeoPixels. the main calling routine would be

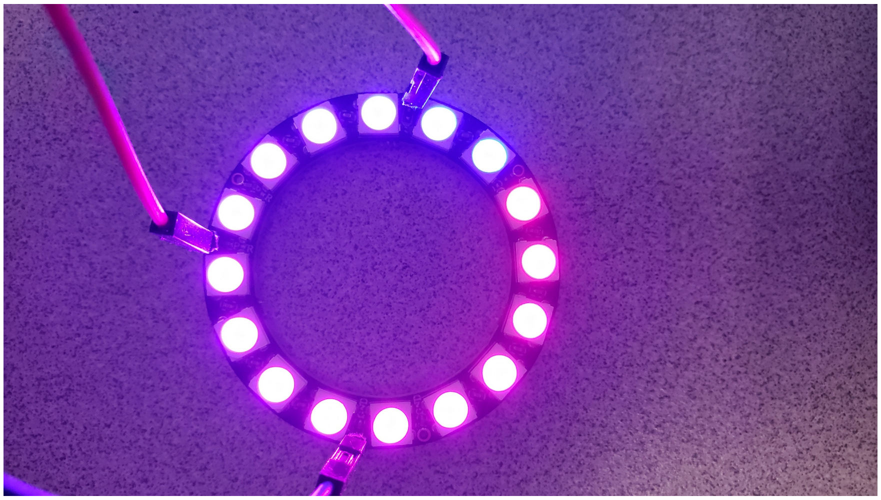
```
x = 0
N = 16
while(1):
    x = (x + 1) % 256
    sm.put(N-1)
    for i in range(0,N):
        g = 0
        r = i*10
        b = 160 - r
        grb = (g << 16) + (r << 8) + b
        sm.put(grb << 8)
    time.sleep(0.05)
```

Calling routine for driving 16 NeoPixels

The NeoPixel driver routine saved as state-machine 0 is then

```
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
```

State-machine code for driving N NeoPixels
https://www.instructables.com/Respberry-Pi-Pico-W-NeoPixels-Experiments-With-Pro/



Using state-machines to drive 16 NeoPixels

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. 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/