MicroPython on the RP2040

Tobias Gresch

Swiss Python Summit, 22.09.2022

Outline

- Introduction
  - About Tweax Sàrl
  - MicroPython
  - RP2040
- MicroPython on RP2040
  - Primer
  - Bi-Core Processing
  - PIO
  - DMA
- Examples and Discussion
  - PWM to Analog Converter
  - MicroPython vs. C/C++

tweax sàrl

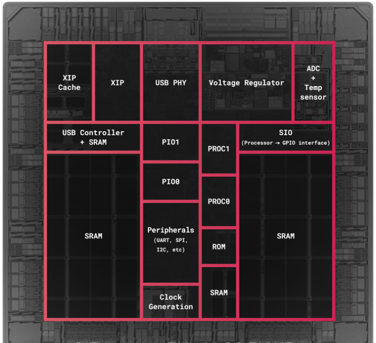
- Founded 1 year ago by myself
- Operating from Lausanne, Switzerland
- Aim is to provide custom IT solutions based on open source hard and software
- Need some help? Do not hesitate to contact us:
  - +41 (0)21 652 9749
  - info@tweax.ch

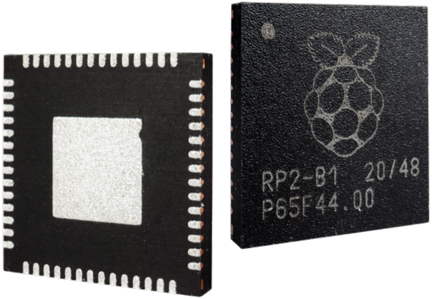
MicroPython Trivia

- Project initiated by Damien George and open-sourced after successful Kickstarter campaign in 2013
- Implementation of Python 3 programming language, optimized to run on **microprocessors** and in **constrained environments**
- Includes subset of Python standard library
- Hardware support through specific modules (machine, ...)
- Ported to many different microprocessors / architectures

RP2040 Trivia

- Designed by Raspberry Pi
- Announced Jan 2021
- Low-cost, 40 nm process, $7 \times 7\mathrm{mm}^2$ QFN-56 package
- Fast, flexible, innovating, easy to use





RP2040 Architecture

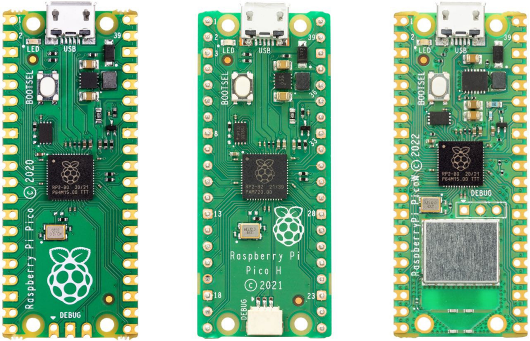
- Dual-core ARM Cortex-M0+ processor
- 264 kB on-chip SRAM in six independent banks
- Supports $\leq$16 MB off-chip Flash via QSPI
- DMA controller
- Fully connected AHB crossbar
- On-chip programmable LDO and PLLs
- Accelerated integer and floating-point libraries on-chip

RP2040 Peripherals

- 2×UART, 2×SPI controllers, 2×I2C controllers, 16×PWM channels
- 1×USB 1.1 controller and PHY, with host and device support
- 8×Programmable I/O (PIO) state machines for custom peripheral support
- 30 GPIO pins (4 can be used as analogue inputs)
- Temperature sensor

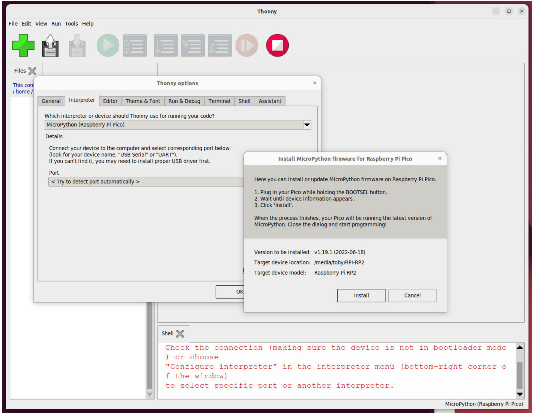
Raspberry Pi Pico

- Plug n’ play module based on rp2040
- Low-cost
- Optionally with headers (H) or WiFi (W)
- Plenty of other modules and hats from third parties available



Integrated Development Environment

- Use your favourite IDE with MicroPython plug-in (PyCharm, VS Code, platform IO, …)
- Raspberry Pi Pico Python SDK mentions **Thonny**



- Supports installing latest MicroPython firmware
- Smooth and painless hardware integration

Primer

- GPIO / Pins

```python
from machine import Pin, ADC, PWM

led = Pin(25, Pin.OUT, value=0)
led.value(1) # equivalent to led.on()
led.value(0) # equivalent to led.off()
```

```python
inp = Pin(0, Pin.IN, Pin.PULL_UP)
print(inp.value())

def irq_handler(pin):
    led.value(led.value() ^ 1)  # toggle LED
inp.irq(irq_handler, Pin.IRQ_FALLING)
```

```python
adc = ADC(Pin(26))      # create ADC object
print(adc.read_u16())  # print value, 0-65535
```

```python
pwm = PWM(Pin(25))  # create PWM object from a pin
pwm.freq(10)        # set frequency 7Hz .. 125MHz
pwm.duty_u16(32768) # set duty cycle, range 0-65535
pwm.deinit()        # continue until deinit() is called!
```

- Delays

```python
import time

time.sleep(1)             # sleep for 1 second
time.sleep_ms(500)        # sleep for 500 milliseconds
time.sleep_us(10)         # sleep for 10 microseconds
start = time.ticks_ms()  # get millisecond counter

delta = time.ticks_diff( # compute time difference
  time.ticks_ms(), start)
```

- Timers

```python
from machine import Timer
# software timers using global microsecond
# timebase

tim = Timer(
    period=1000,              # period in ms
    mode=Timer.PERIODIC,
    callback=lambda t:print('Hello, World!')
)
```

Overclocking the RP2040

- Default system clock frequency: 125 MHz
- Can be adjusted dynamically up to $\approx$ 270 MHz

```
>>> import machine
>>> machine.freq()
125000000
>>> machine.freq(270_000_000)
>>> machine.freq()
270000000
```

- Overclocking possible $>$ 400 MHz
- Requires tweaking of:
  - $0.85 \leq \mathrm{VREG} \leq 1.3$ VDC
  - Throttling QSPI clock (PICO_FLASH_SPI_CLKDIV=4 in second-stage bootloader)

Bi-Core Support

- Main thread runs on *core0*
- Run optional thread on *core1* (just one!)
- Code will run concurrently, no GIL!
- Thread synchronization and locks

```
import _thread, machine, time
LED = machine.Pin(25, machine.Pin.OUT)
def task(n, delay):
    for i in range(n):
        LED.high()
        time.sleep(delay)
        LED.low()
        time.sleep(delay)
    print('done')

_thread.start_new_thread(task, (10, 0.5))
```

```
>>> %Run -c $EDITOR_CONTENT
>>> done
```

Offloading Occasional Tasks

- Use uasyncio and regular Python together

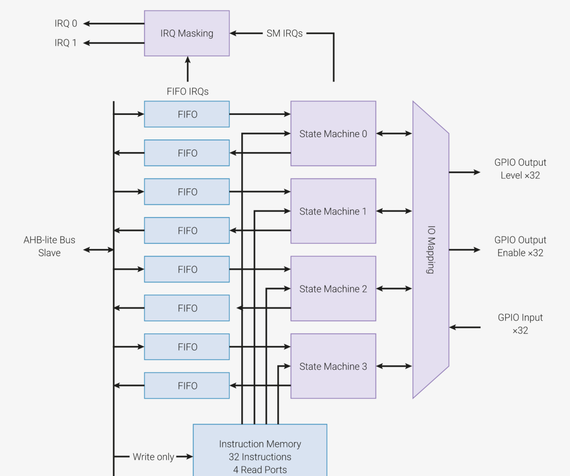
```
import uasyncio, time, _thread

LED = machine.Pin(25, machine.Pin.OUT)
LED_LOCK = uasyncio.Lock()

async def flash(led, dur_ms):
    async with LED_LOCK:
        print('flash')
        led.on()
        await uasyncio.sleep_ms(dur_ms)
        led.off()

async def say_after(text, delay_ms):
    await uasyncio.sleep_ms(delay_ms)
    print(text)

def other():
    while True:
        print('doing someting on core1')
        task = uasyncio.create_task(
            flash(LED, 200))
        time.sleep(1)
```

```
async def main():
    _thread.start_new_thread(other, ())
    task1 = uasyncio.create_task(
            say_after('Hello', 1000))
    task2 = uasyncio.create_task(
            say_after('World!', 2000))
    while True:
        await uasyncio.sleep_ms(10)
uasyncio.run(main())
```

```
>>> %Run -c $EDITOR_CONTENT
doing someting on core1
flash
doing someting on core1
Hello
flash
doing someting on core1
World!
flash...
```

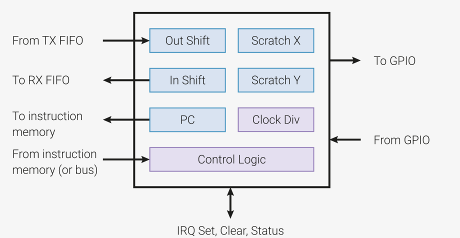
PIO (Programmable IO)

- Two PIO blocks with 4 state machines each
- Co-processors dedicated to I/O



PIO State Machine

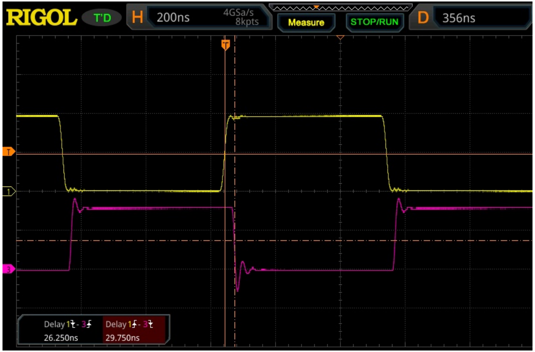
- Each state machine equipped with:
  - 2× 32-bit shift registers (ISR, OSR)
  - 2× 32-bit scratch registers (X, Y)
  - RX/TX DMA capable FIFOs — 4 (8) words deep
  - Fractional clock divider
  - IRQ flag (set/clear/status)
  - flexible GPIO mapping for up to 30 GPIOs
- Share instruction memory (32 instructions)



PIO Assembly

- 9 instructions: JMP, WAIT, IN, OUT, PUSH, PULL, MOV, IRQ and SET
- 1 instruction per cyle
- No arithmetic! (reverse, bitwise compl. in MOV)
- rp2 module in MicroPython for in-line PIO assembly

```
from machine import Pin
import rp2

@rp2.asm_pio(set_init=rp2.PIO.OUT_HIGH)
def not_gate():
    wrap_target()           # default
    label("check")
    jmp(pin, "set LOW")
    set(pins, 1)
    jmp("check")
    label("set LOW")
    set(pins, 0) [1]        # set HIGH
    wrap()                  # default

IN_A = Pin(0, Pin.IN); OUT = Pin(2, Pin.OUT)
sm = rp2.StateMachine(0, not_gate, freq=125_000_000,
     set_base=OUT, jmp_pin=IN_A)
sm.active(1)
```



Gate Logic

- Single-gate logic with PIO

```
@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW,
    sideset_init=rp2.PIO.OUT_HIGH)
def nand2_gate():
    set(y, 0b11)
    wrap_target()
    label("get input")
    mov(isr, null)
    in_(pins, 2)
    mov(x, isr)
    jmp(x_not_y, "set LOW")
    set(pins, 1).side(0)
    jmp("get input")
    label("set LOW")
    set(pins, 0).side(1)

_OUT = Pin(2, Pin.OUT)
sm = rp2.StateMachine(0, nand2_gate,
    freq=125_000_000,
    in_base=IN_A,
    set_base=OUT,
    sideset_base_pin=_OUT)
```

```
@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def or2_gate():
    label("get input")
    mov(isr, null)
    in_(pins, 2)
    mov(x, isr)
    jmp(not_x, "set LOW")
    set(pins, 1)
    jmp("get input")
    label("set LOW")
    set(pins, 0)
```

```
@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def xor2_gate():
    set(y, 0b11)
    wrap_target()
    label("get input")
    mov(isr, null)
    in_(pins, 2)
    mov(x, isr)
    jmp(not_x, "set LOW")
    jmp(x_not_y, "set HIGH")
    label("set LOW")
    set(pins, 0)
    jmp("get input")
    label("set HIGH")
    set(pins, 1)
```

DMA

- 12 independent DMA channels
- One read and one write access of up to 32 bits per cycle
- Very flexible, i.e. config by channel or chaining
- Use *uctypes* for definition of hardware registers

```
# See https://iosoft.blog/pico-adc-dma for description
# Apache V2.0
from uctypes import BF_POS, BF_LEN, UINT32, BFUINT32,
        struct

DMA_CHAN_REGS = {
    "READ_ADDR_REG":        0x00|UINT32,
    "WRITE_ADDR_REG":       0x04|UINT32,
    "TRANS_COUNT_REG":      0x08|UINT32,
    "CTRL_TRIG_REG":        0x0c|UINT32,
    "CTRL_TRIG":           (0x0c,DMA_CTRL_TRIG_FIELDS)
}
```

```
DMA_CTRL_TRIG_FIELDS = {
    "AHB_ERROR":   31<<BF_POS | 1<<BF_LEN | BFUINT32,
    "READ_ERROR":  30<<BF_POS | 1<<BF_LEN | BFUINT32,
    "WRITE_ERROR": 29<<BF_POS | 1<<BF_LEN | BFUINT32,
    "BUSY":        24<<BF_POS | 1<<BF_LEN | BFUINT32,
    "SNIFF_EN":    23<<BF_POS | 1<<BF_LEN | BFUINT32,
    "BSWAP":       22<<BF_POS | 1<<BF_LEN | BFUINT32,
    "IRQ_QUIET":   21<<BF_POS | 1<<BF_LEN | BFUINT32,
    "TREQ_SEL":    15<<BF_POS | 6<<BF_LEN | BFUINT32,
    "CHAIN_TO":    11<<BF_POS | 4<<BF_LEN | BFUINT32,
    ...
}
```

- Instantiate specific structure at given memory address using *uctypes.struct()*
- Access the values using dot-notation

PWM to Analogue Converter

LTC2644

Dual 12-/10-/8-Bit PWM to V_{OUT} DACs with 10ppm/°C Reference

Production

Overview | Evaluation Kits | Documentation | Product Recommendations | Reference Materials | Design Resources | Support & Discussions | Sample & Buy

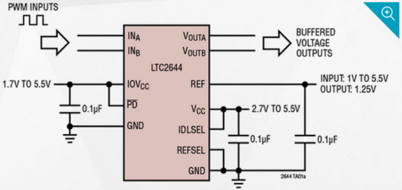
View All Data Sheets (1)

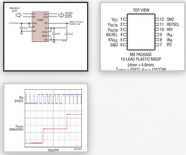
Data Sheet Rev. B

JP Rev. 0

User Guides

1 View All





BACKORDERED!

Overview

Features and Benefits | Product Details

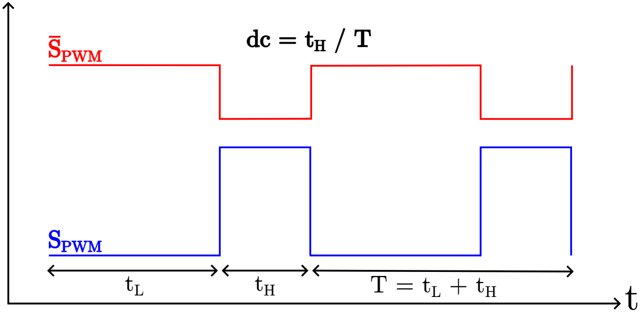
- No Latency PWM-to-Voltage Conversion
- Voltage Output Updates and Settles within 8µs
- 100kHz to 30Hz PWM Input Frequency
- ±2.5LSB Max INL; ±1LSB Max DNL (LTC2644-12)
- Guaranteed Monotonic
- Pin-Selectable Internal or External Reference

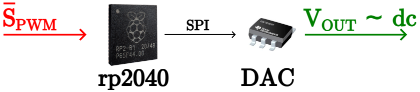
Product Categories

D/A Converters (DAC)

- PWM to Voltage Output D/A Converters
- Single-Supply Voltage Output D/A Converters
- Multichannel Voltage Output D/A Converters

PWM to Analogue Converter





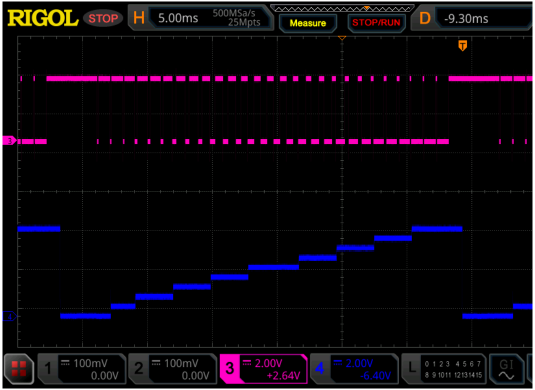
PWM to Analogue Converter

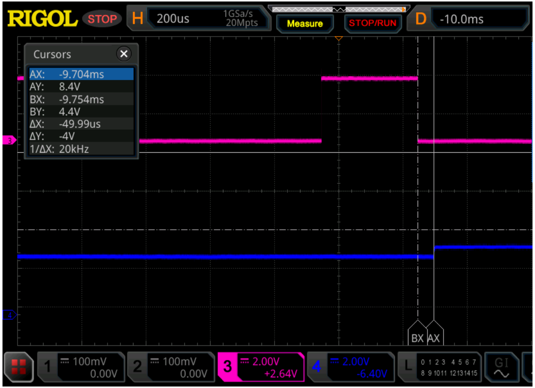
- Implementation for fixed PWM frequency and no DMA

```python
from machine import Pin, SPI
from micropython import const
import rp2
import struct

const _DAC_RES = 2**12
IN = Pin(0, Pin.IN, pull=Pin.PULL_UP)
CS = Pin(17, mode=Pin.OUT, value=1)
spi =  SPI(0, baudrate=10_000_000, bits=8,
        polarity=0, phase=1, firstbit=SPI.MSB,
        sck=Pin(18), mosi=Pin(19), miso=Pin(16)
)
write_buf = bytearray(2)
sm.put(_DAC_RES)        # 12-bit DAC
while True:
    val = sm.get() # get value from RX FIFO (blocking)
    write_buf = struct.pack(">H", val)
    try:
        CS.value(0)
        spi.write(write_buf)
    finally:
         CS.value(1)
```

```python
@rp2.asm_pio()
def pwm_cnt():
    pull(noblock)       # get DAC resolution from main
    mov(x, osr)         # save value for reuse
    mov(y, osr)
    wait(0, pin, 0)     # wait 0 -> 1
    wait(1, pin, 0)     # (inverted signal)
    label("start loop")
    jmp(pin, "decrement counter")
    jmp("get out")
    label("decrement counter")
    jmp(y_dec, "start loop")
    mov(y, null)        # underrun, set to 0
    label("get out")
    mov(isr, y)
    push()              # counter value to FIFO -> main
const _PWM_FREQ = 100
const _FREQ = 2 * _DAC_RES *_PWM_FREQ
sm = rp2.StateMachine(0, pwm_cnt, freq=_FREQ,
                      in_base=IN, jmp_pin=IN)
sm.active(1)
```

PWM to Analogue Converter





- Settling time $\approx$7 times slower than LTC2644 (50 vs. 7μs)
  - Ensure that CS signal transitions immediately after data has been transferred
  - Use DMA for speedup and running in background

MicroPython vs. C/C++

- Use whatever you are comfortable with
- C/C++ is in case you are lacking RAM or need more efficient processing
- Optimize your application in MicroPython:
  - Native Code Emitter: Produces machine instructions
  - Viper Code Emitter: Further optimizations, especially for integer arithmetic and bit manipulations
  - String interning: Store strings in Flash after identification with *micropython.qstr_info(1)*
  - Use third-party modules like ulab (numpy-alike array manipulation)
  - Write C/C++ modules

Summary / Questions

- Summary
  - rp2040 is an interesting microprocessor with some unique features
  - MicroPython is a neat way to discover features of the rp2040 and experiment with them
  - Find excellent projects using rp2040 on the web
    - Logic analyzers
    - Oscilloscopes
    - …
- Questions?