`rp2` — functionality specific to the RP2040

The `rp2` module contains functions and classes specific to the RP2040, as used in the Raspberry Pi Pico.

See the RP2040 Python datasheet for more information, and pico-micropython-examples for example code.

PIO related functions

The `rp2` module includes functions for assembling PIO programs.

For running PIO programs, see `rp2.StateMachine`.

rp2.asm_pio(**, out_init=None, set_init=None, sideset_init=None, in_shiftdir=0, out_shiftdir=0, autopush=False, autopull=False, push_thresh=32, pull_thresh=32, fifo_join=PIO.JOIN_NONE*)

Assemble a PIO program.

The following parameters control the initial state of the GPIO pins, as one of `PIO.IN_LOW`, `PIO.IN_HIGH`, `PIO.OUT_LOW` or `PIO.OUT_HIGH`. If the program uses more than one pin, provide a tuple, e.g. `out_init=(PIO.OUT_LOW, PIO.OUT_LOW)`.

- *out_init* configures the pins used for `out()` instructions.
- *set_init* configures the pins used for `set()` instructions. There can be at most 5.
- *sideset_init* configures the pins used side-setting. There can be at most 5.

The following parameters are used by default, but can be overridden in `StateMachine.init()`:

- *in_shiftdir* is the default direction the ISR will shift, either `PIO.SHIFT_LEFT` or `PIO.SHIFT_RIGHT`.
- *out_shiftdir* is the default direction the OSR will shift, either `PIO.SHIFT_LEFT` or `PIO.SHIFT_RIGHT`.
- *push_thresh* is the threshold in bits before auto-push or conditional re-pushing is triggered.
- *pull_thresh* is the threshold in bits before auto-pull or conditional re-pulling is triggered.

The remaining parameters are:

- *autopush* configures whether auto-push is enabled.
- *autopull* configures whether auto-pull is enabled.
- *fifo_join* configures whether the 4-word TX and RX FIFOs should be combined into a single 8-word FIFO for one direction only. The options are `PIO.JOIN_NONE`, `PIO.JOIN_RX` and `PIO.JOIN_TX`.

rp2.asm_pio_encode(*instr, sideset_count, sideset_opt=False*)

Assemble a single PIO instruction. You usually want to use `asm_pio()` instead.

```
>>> rp2.asm_pio_encode("set(0, 1)", 0)
57345
```

rp2.bootsel_button()

Temporarily turns the QSPI_SS pin into an input and reads its value, returning 1 for low and 0 for high. On a typical RP2040 board with a BOOTSEL button, a return value of 1 indicates that the button is pressed.

Since this function temporarily disables access to the external flash memory, it also temporarily disables interrupts and the other core to prevent them from trying to execute code from flash.

class **rp2.PIOASMError**

This exception is raised from `asm_pio()` or `asm_pio_encode()` if there is an error assembling a PIO program.

PIO assembly language instructions

PIO state machines are programmed in a custom assembly language with nine core PIO-machine instructions. In MicroPython, PIO assembly routines are written as a Python function with the decorator `@rp2.asm_pio()`, and they use Python syntax. Such routines support standard Python variables and arithmetic, as well as the following custom functions that encode PIO instructions and direct the assembler. See sec 3.4 of the RP2040 datasheet for further details.

wrap_target()

Specify the location where execution continues after program wrapping. By default this is the start of the PIO routine.

wrap()

Specify the location where the program finishes and wraps around. If this directive is not used then it is added automatically at the end of the PIO routine. Wrapping does not cost any execution cycles.

label(label)

Define a label called *label* at the current location. *label* can be a string or integer.

word(instr, label=None)

Insert an arbitrary 16-bit word in the assembled output.

- *instr*: the 16-bit value
- *label*: if given, look up the label and logical-or the label's value with *instr*

jmp(...)

This instruction takes two forms:

jmp(label)

- *label*: label to jump to unconditionally

jmp(cond, label)

- *cond*: the condition to check, one of:
  - `not_x`, `not_y`: true if register is zero
  - `x_dec`, `y_dec`: true if register is non-zero, and do post decrement
  - `x_not_y`: true if X is not equal to Y
  - `pin`: true if the input pin is set
  - `not_osre`: true if OSR is not empty (hasn't reached its threshold)
- *label*: label to jump to if condition is true

wait(polarity, src, index)

Block, waiting for high/low on a pin or IRQ line.

- *polarity*: 0 or 1, whether to wait for a low or high value
- *src*: one of: `gpio` (absolute pin), `pin` (pin relative to StateMachine's `in_base` argument), `irq`
- *index*: 0-31, the index for *src*

in_(src, bit_count)

Shift data in from *src* to ISR.

- *src*: one of: `pins`, `x`, `y`, `null`, `isr`, `osr`
- *bit_count*: number of bits to shift in (1-32)

out(dest, bit_count)

Shift data out from OSR to *dest*.

- *dest*: one of: `pins`, `x`, `y`, `pindirs`, `pc`, `isr`, `exec`
- *bit_count*: number of bits to shift out (1-32)

push(...)

Push ISR to the RX FIFO, then clear ISR to zero. This instruction takes the following forms:

- push()
- push(block)
- push(noblock)
- push(iffull)
- push(iffull, block)
- push(iffull, noblock)

If `block` is used then the instruction stalls if the RX FIFO is full. The default is to block. If `iffull` is used then it only pushes if the input shift count has reached its threshold.

pull(...)

Pull from the TX FIFO into OSR. This instruction takes the following forms:

- pull()
- pull(block)
- pull(noblock)
- pull(ifempty)
- pull(ifempty, block)
- pull(ifempty, noblock)

If `block` is used then the instruction stalls if the TX FIFO is empty. The default is to block. If `ifempty` is used then it only pulls if the output shift count has reached its threshold.

mov(dest, src)

Move into *dest* the value from *src*.

- *dest*: one of: `pins`, `x`, `y`, `exec`, `pc`, `isr`, `osr`
- *src*: one of: `pins`, `x`, `y`, `null`, `status`, `isr`, `osr`; this argument can be optionally modified by wrapping it in `invert()` or `reverse()` (but not both together)

irq(...)

Set or clear an IRQ flag. This instruction takes two forms:

irq(index)

- *index*: 0-7, or `rel(0)` to `rel(7)`

irq(mode, index)

- *mode*: one of: `block`, `clear`
- *index*: 0-7, or `rel(0)` to `rel(7)`

If `block` is used then the instruction stalls until the flag is cleared by another entity. If `clear` is used then the flag is cleared instead of being set. Relative IRQ indices add the state machine ID to the IRQ index with modulo-4 addition. IRQs 0-3 are visible from to the processor, 4-7 are internal to the state machines.

set(dest, data)

Set *dest* with the value *data*.

- *dest*: `pins`, `x`, `y`, `pindirs`
- *data*: value (0-31)

nop()

This is a pseudoinstruction that assembles to `mov(y, y)` and has no side effect.

.side(value)

This is a modifier which can be applied to any instruction, and is used to control side-set pin values.

- *value*: the value (bits) to output on the side-set pins

.delay(value)

This is a modifier which can be applied to any instruction, and specifies how many cycles to delay for after the instruction executes.

- *value*: cycles to delay, 0-31 (maximum value reduced if side-set pins are used)

[value]

This is a modifier and is equivalent to `.delay(value)`.

Classes

- class DMA – access to the RP2040's DMA controller
- class Flash – access to built-in flash storage
- class PIO – advanced PIO usage
- class StateMachine – access to the RP2040's programmable I/O interface