This is the documentation for the latest development branch of MicroPython and may refer to features that are not available in released versions.

If you are looking for the documentation for a specific release, use the drop-down menu on the left and select the desired version.

`machine` — functions related to the hardware

The `machine` module contains specific functions related to the hardware on a particular board. Most functions in this module allow to achieve direct and unrestricted access to and control of hardware blocks on a system (like CPU, timers, buses, etc.). Used incorrectly, this can lead to malfunction, lockups, crashes of your board, and in extreme cases, hardware damage.

A note of callbacks used by functions and class methods of `machine` module: all these callbacks should be considered as executing in an interrupt context. This is true for both physical devices with IDs >= 0 and “virtual” devices with negative IDs like -1 (these “virtual” devices are still thin shims on top of real hardware and real hardware interrupts). See Writing interrupt handlers.

Memory access

The module exposes three objects used for raw memory access.

`machine.mem8`

Read/write 8 bits of memory.

`machine.mem16`

Read/write 16 bits of memory.

`machine.mem32`

Read/write 32 bits of memory.

Use subscript notation `[...]` to index these objects with the address of interest. Note that the address is the byte address, regardless of the size of memory being accessed.

Example use (registers are specific to an stm32 microcontroller):

```
import machine
from micropython import const

GPIOA = const(0x48000000)
GPIO_BSRR = const(0x18)
GPIO_IDR = const(0x10)

# set PA2 high
machine.mem32[GPIOA + GPIO_BSRR] = 1 << 2

# read PA3
value = (machine.mem32[GPIOA + GPIO_IDR] >> 3) & 1
```

Reset related functions

`machine.reset()`

Hard resets the device in a manner similar to pushing the external RESET button.

`machine.soft_reset()`

Performs a soft reset of the interpreter, deleting all Python objects and resetting the Python heap.

`machine.reset_cause()`

Get the reset cause. See constants for the possible return values.

`machine.bootloader(` [*value*] `)`

Reset the device and enter its bootloader. This is typically used to put the device into a state where it can be programmed with new firmware.

Some ports support passing in an optional *value* argument which can control which bootloader to enter, what to pass to it, or other things.

Interrupt related functions

The following functions allow control over interrupts. Some systems require interrupts to operate correctly so disabling them for long periods may compromise core functionality, for example watchdog timers may trigger unexpectedly. Interrupts should only be disabled for a minimum amount of time and then re-enabled to their previous state. For example:

```
import machine

# Disable interrupts
state = machine.disable_irq()

# Do a small amount of time-critical work here

# Enable interrupts
machine.enable_irq(state)
```

`machine.disable_irq()`

Disable interrupt requests. Returns the previous IRQ state which should be considered an opaque value. This return value should be passed to the `enable_irq()` function to restore interrupts to their original state, before `disable_irq()` was called.

`machine.enable_irq(`*`state`*`)`

Re-enable interrupt requests. The *state* parameter should be the value that was returned from the most recent call to the `disable_irq()` function.

Power related functions

`machine.freq(`[*`hz`*]`)`

Returns the CPU frequency in hertz.

On some ports this can also be used to set the CPU frequency by passing in *hz*.

`machine.idle()`

Gates the clock to the CPU, useful to reduce power consumption at any time during short or long periods. Peripherals continue working and execution resumes as soon as any interrupt is triggered, or at most one millisecond after the CPU was paused.

It is recommended to call this function inside any tight loop that is continuously checking for an external change (i.e. polling). This will reduce power consumption without significantly impacting performance. To reduce power consumption further then see the `lightsleep()`, `time.sleep()` and `time.sleep_ms()` functions.

`machine.sleep()`

Note

This function is deprecated, use `lightsleep()` instead with no arguments.

`machine.lightsleep(`[*`time_ms`*]`)`

`machine.deepsleep(`[*`time_ms`*]`)`

Stops execution in an attempt to enter a low power state.

If *time_ms* is specified then this will be the maximum time in milliseconds that the sleep will last for. Otherwise the sleep can last indefinitely.

With or without a timeout, execution may resume at any time if there are events that require processing. Such events, or wake sources, should be configured before sleeping, like `Pin` change or `RTC` timeout.

The precise behaviour and power-saving capabilities of lightsleep and deepsleep is highly dependent on the underlying hardware, but the general properties are:

- A lightsleep has full RAM and state retention. Upon wake execution is resumed from the point where the sleep was requested, with all subsystems operational.

- A deepsleep may not retain RAM or any other state of the system (for example peripherals or network interfaces). Upon wake execution is resumed from the main script, similar to a hard or power-on reset. The `reset_cause()` function will return `machine.DEEPSLEEP` and this can be used to distinguish a deepsleep wake from other resets.

`machine.wake_reason()`

Get the wake reason. See constants for the possible return values.

Availability: ESP32, WiPy.

Miscellaneous functions

`machine.unique_id()`

Returns a byte string with a unique identifier of a board/SoC. It will vary from a board/SoC instance to another, if underlying hardware allows. Length varies by hardware (so use substring of a full value if you expect a short ID). In some MicroPython ports, ID corresponds to the network MAC address.

`machine.time_pulse_us(pin, pulse_level, timeout_us=1000000, /)`

Time a pulse on the given *pin*, and return the duration of the pulse in microseconds. The *pulse_level* argument should be 0 to time a low pulse or 1 to time a high pulse.

If the current input value of the pin is different to *pulse_level*, the function first (*) waits until the pin input becomes equal to *pulse_level*, then (**) times the duration that the pin is equal to *pulse_level*. If the pin is already equal to *pulse_level* then timing starts straight away.

The function will return -2 if there was timeout waiting for condition marked (*) above, and -1 if there was timeout during the main measurement, marked (**) above. The timeout is the same for both cases and given by *timeout_us* (which is in microseconds).

`machine.bitstream(pin, encoding, timing, data, /)`

Transmits *data* by bit-banging the specified *pin*. The *encoding* argument specifies how the bits are encoded, and *timing* is an encoding-specific timing specification.

The supported encodings are:

- `0` for "high low" pulse duration modulation. This will transmit 0 and 1 bits as timed pulses, starting with the most significant bit. The *timing* must be a four-tuple of nanoseconds in the format `(high_time_0, low_time_0, high_time_1, low_time_1)`. For example, `(400, 850, 800, 450)` is the timing specification for WS2812 RGB LEDs at 800kHz.

The accuracy of the timing varies between ports. On Cortex M0 at 48MHz, it is at best +/- 120ns, however on faster MCUs (ESP8266, ESP32, STM32, Pyboard), it will be closer to +/-30ns.

Note

For controlling WS2812 / NeoPixel strips, see the `neopixel` module for a higher-level API.

`machine.rng()`

Return a 24-bit software generated random number.

Availability: WiPy.

Constants

`machine.IDLE`

`machine.SLEEP`

`machine.DEEPSLEEP`

IRQ wake values.

`machine.PWRON_RESET`

`machine.HARD_RESET`

`machine.WDT_RESET`

`machine.DEEPSLEEP_RESET`

`machine.SOFT_RESET`

Reset causes.

`machine.WLAN_WAKE`

`machine.PIN_WAKE`

`machine.RTC_WAKE`

Wake-up reasons.

Classes