pyb — functions related to the board

The pyb module contains specific functions related to the board.

Time related functions

pyb.delay(*ms*)

Delay for the given number of milliseconds.

pyb.udelay(*us*)

Delay for the given number of microseconds.

pyb.millis()

Returns the number of milliseconds since the board was last reset.

The result is always a MicroPython smallint (31-bit signed number), so after 2^30 milliseconds (about 12.4 days) this will start to return negative numbers.

Note that if `pyb.stop()` is issued the hardware counter supporting this function will pause for the duration of the "sleeping" state. This will affect the outcome of `pyb.elapsed_millis()`.

pyb.micros()

Returns the number of microseconds since the board was last reset.

The result is always a MicroPython smallint (31-bit signed number), so after 2^30 microseconds (about 17.8 minutes) this will start to return negative numbers.

Note that if `pyb.stop()` is issued the hardware counter supporting this function will pause for the duration of the "sleeping" state. This will affect the outcome of `pyb.elapsed_micros()`.

pyb.elapsed_millis(*start*)

Returns the number of milliseconds which have elapsed since `start`.

This function takes care of counter wrap, and always returns a positive number. This means it can be used to measure periods up to about 12.4 days.

Example:

```
start = pyb.millis()
while pyb.elapsed_millis(start) < 1000:
    # Perform some operation
```

pyb.elapsed_micros(*start*)

Returns the number of microseconds which have elapsed since `start`.

This function takes care of counter wrap, and always returns a positive number. This means it can be used to measure periods up to about 17.8 minutes.

Example:

```
start = pyb.micros()
while pyb.elapsed_micros(start) < 1000:
    # Perform some operation
    pass
```

Reset related functions

pyb.hard_reset()

Resets the pyboard in a manner similar to pushing the external RESET button.

pyb.bootloader()

Activate the bootloader without BOOT* pins.

pyb.fault_debug(*value*)

Enable or disable hard-fault debugging. A hard-fault is when there is a fatal error in the underlying system, like an invalid memory access.

If the *value* argument is `False` then the board will automatically reset if there is a hard fault.

If *value* is `True` then, when the board has a hard fault, it will print the registers and the stack trace, and then cycle the LEDs indefinitely.

The default value is disabled, i.e. to automatically reset.

Interrupt related functions

pyb.disable_irq()

Disable interrupt requests. Returns the previous IRQ state: `False` / `True` for disabled/enabled IRQs respectively. This return value can be passed to enable_irq to restore the IRQ to its original state.

pyb.enable_irq(*state=True*)

Enable interrupt requests. If `state` is `True` (the default value) then IRQs are enabled. If `state` is `False` then IRQs are disabled. The most common use of this function is to pass it the value returned by `disable_irq` to exit a critical section.

Power related functions

pyb.freq([*sysclk* [, *hclk* [, *pclk1* [, *pclk2*]]]])

If given no arguments, returns a tuple of clock frequencies: (sysclk, hclk, pclk1, pclk2). These correspond to:

- sysclk: frequency of the CPU
- hclk: frequency of the AHB bus, core memory and DMA
- pclk1: frequency of the APB1 bus
- pclk2: frequency of the APB2 bus

If given any arguments then the function sets the frequency of the CPU, and the busses if additional arguments are given. Frequencies are given in Hz. Eg freq(120000000) sets sysclk (the CPU frequency) to 120MHz. Note that not all values are supported and the largest supported frequency not greater than the given value will be selected.

Supported sysclk frequencies are (in MHz): 8, 16, 24, 30, 32, 36, 40, 42, 48, 54, 56, 60, 64, 72, 84, 96, 108, 120, 144, 168.

The maximum frequency of hclk is 168MHz, of pclk1 is 42MHz, and of pclk2 is 84MHz. Be sure not to set frequencies above these values.

The hclk, pclk1 and pclk2 frequencies are derived from the sysclk frequency using a prescaler (divider). Supported prescalers for hclk are: 1, 2, 4, 8, 16, 64, 128, 256, 512. Supported prescalers for pclk1 and pclk2 are: 1, 2, 4, 8. A prescaler will be chosen to best match the requested frequency.

A sysclk frequency of 8MHz uses the HSE (external crystal) directly and 16MHz uses the HSI (internal oscillator) directly. The higher frequencies use the HSE to drive the PLL (phase locked loop), and then use the output of the PLL.

Note that if you change the frequency while the USB is enabled then the USB may become unreliable. It is best to change the frequency in boot.py, before the USB peripheral is started. Also note that sysclk frequencies below 36MHz do not allow the USB to function correctly.

pyb.wfi()

Wait for an internal or external interrupt.

This executes a `wfi` instruction which reduces power consumption of the MCU until any interrupt occurs (be it internal or external), at which point execution continues. Note that the system-tick interrupt occurs once every millisecond (1000Hz) so this function will block for at most 1ms.

pyb.stop()

Put the pyboard in a "sleeping" state.

This reduces power consumption to less than 500 uA. To wake from this sleep state requires an external interrupt or a real-time-clock event. Upon waking execution continues where it left off.

See `rtc.wakeup()` to configure a real-time-clock wakeup event.

pyb.standby()

Put the pyboard into a "deep sleep" state.

This reduces power consumption to less than 50 uA. To wake from this sleep state requires a real-time-clock event, or an external interrupt on X1 (PA0=WKUP) or X18 (PC13=TAMP1). Upon waking the system undergoes a hard reset.

See `rtc.wakeup()` to configure a real-time-clock wakeup event.

Miscellaneous functions

pyb.have_cdc()

Return True if USB is connected as a serial device, False otherwise.

Note

This function is deprecated. Use pyb.USB_VCP().isconnected() instead.

pyb.hid(*(buttons, x, y, z)*)

Takes a 4-tuple (or list) and sends it to the USB host (the PC) to signal a HID mouse-motion event.

Note

This function is deprecated. Use `pyb.USB_HID.send()` instead.

pyb.info([*dump_alloc_table*])

Print out lots of information about the board.

pyb.main(*filename*)

Set the filename of the main script to run after boot.py is finished. If this function is not called then the default file main.py will be executed.

It only makes sense to call this function from within boot.py.

pyb.mount(*device, mountpoint, *, readonly=False, mkfs=False*)

> **Note**
>
> This function is deprecated. Mounting and unmounting devices should be performed by `uos.mount()` and `uos.umount()` instead.

Mount a block device and make it available as part of the filesystem. `device` must be an object that provides the block protocol. (The following is also deprecated. See `uos.AbstractBlockDev` for the correct way to create a block device.)

- `readblocks(self, blocknum, buf)`
- `writeblocks(self, blocknum, buf)` (optional)
- `count(self)`
- `sync(self)` (optional)

`readblocks` and `writeblocks` should copy data between `buf` and the block device, starting from block number `blocknum` on the device. `buf` will be a bytearray with length a multiple of 512. If `writeblocks` is not defined then the device is mounted read-only. The return value of these two functions is ignored.

`count` should return the number of blocks available on the device. `sync`, if implemented, should sync the data on the device.

The parameter `mountpoint` is the location in the root of the filesystem to mount the device. It must begin with a forward-slash.

If `readonly` is `True`, then the device is mounted read-only, otherwise it is mounted read-write.

If `mkfs` is `True`, then a new filesystem is created if one does not already exist.

pyb.repl_uart(*uart*)

Get or set the UART object where the REPL is repeated on.

pyb.rng()

Return a 30-bit hardware generated random number.

pyb.sync()

Sync all file systems.

pyb.unique_id()

Returns a string of 12 bytes (96 bits), which is the unique ID of the MCU.

pyb.usb_mode([*modestr,*] *vid=0xf055, pid=0x9801, hid=pyb.hid_mouse*)

If called with no arguments, return the current USB mode as a string.

If called with `modestr` provided, attempts to set USB mode. This can only be done when called from `boot.py` before `pyb.main()` has been called. The following values of `modestr` are understood:

- `None`: disables USB
- `'VCP'`: enable with VCP (Virtual COM Port) interface
- `'MSC'`: enable with MSC (mass storage device class) interface
- `'VCP+MSC'`: enable with VCP and MSC
- `'VCP+HID'`: enable with VCP and HID (human interface device)

For backwards compatibility, `'CDC'` is understood to mean `'VCP'` (and similarly for `'CDC+MSC'` and `'CDC+HID'`).

The `vid` and `pid` parameters allow you to specify the VID (vendor id) and PID (product id).

If enabling HID mode, you may also specify the HID details by passing the `hid` keyword parameter. It takes a tuple of (subclass, protocol, max packet length, polling interval, report descriptor). By default it will set appropriate values for a USB mouse. There is also a `pyb.hid_keyboard` constant, which is an appropriate tuple for a USB keyboard.

Classes

- class Accel – accelerometer control
- class ADC – analog to digital conversion
- class CAN – controller area network communication bus
- class DAC – digital to analog conversion
- class ExtInt – configure I/O pins to interrupt on external events
- class I2C – a two-wire serial protocol
- class LCD – LCD control for the LCD touch-sensor pyskin
- class LED – LED object
- class Pin – control I/O pins
- class PinAF – Pin Alternate Functions
- class RTC – real time clock
- class Servo – 3-wire hobby servo driver
- class SPI – a master-driven serial protocol

- class Switch – switch object
- class Timer – control internal timers
- class TimerChannel — setup a channel for a timer
- class UART – duplex serial communication bus
- class USB_HID – USB Human Interface Device (HID)
- class USB_VCP – USB virtual comm port