class Pin – control I/O pins

A pin object is used to control I/O pins (also known as GPIO - general-purpose input/output). Pin objects are commonly associated with a physical pin that can drive an output voltage and read input voltages. The pin class has methods to set the mode of the pin (IN, OUT, etc) and methods to get and set the digital logic level. For analog control of a pin, see the `ADC` class.

A pin object is constructed by using an identifier which unambiguously specifies a certain I/O pin. The allowed forms of the identifier and the physical pin that the identifier maps to are port-specific. Possibilities for the identifier are an integer, a string or a tuple with port and pin number.

Usage Model:

```
from machine import Pin

# create an output pin on pin #0
p0 = Pin(0, Pin.OUT)

# set the value low then high
p0.value(0)
p0.value(1)

# create an input pin on pin #2, with a pull up resistor
p2 = Pin(2, Pin.IN, Pin.PULL_UP)

# read and print the pin value
print(p2.value())

# reconfigure pin #0 in input mode
p0.mode(p0.IN)

# configure an irq callback
p0.irq(lambda p:print(p))
```

Constructors

class **machine.Pin**(*id, mode=-1, pull=-1, *, value, drive, alt*)

Access the pin peripheral (GPIO pin) associated with the given `id`. If additional arguments are given in the constructor then they are used to initialise the pin. Any settings that are not specified will remain in their previous state.

The arguments are:

- `id` is mandatory and can be an arbitrary object. Among possible value types are: int (an internal Pin identifier), str (a Pin name), and tuple (pair of [port, pin]).
- `mode` specifies the pin mode, which can be one of:
  - `Pin.IN` - Pin is configured for input. If viewed as an output the pin is in high-impedance state.
  - `Pin.OUT` - Pin is configured for (normal) output.
  - `Pin.OPEN_DRAIN` - Pin is configured for open-drain output. Open-drain output works in the following way: if the output value is set to 0 the pin is active at a low level; if the output value is 1 the pin is in a high-impedance state. Not all ports implement this mode, or some might only on certain pins.
  - `Pin.ALT` - Pin is configured to perform an alternative function, which is port specific. For a pin configured in such a way any other Pin methods (except `Pin.init()`) are not applicable (calling them will lead to undefined, or a hardware-specific, result). Not all ports implement this mode.
  - `Pin.ALT_OPEN_DRAIN` - The Same as `Pin.ALT`, but the pin is configured as open-drain. Not all ports implement this mode.
- `pull` specifies if the pin has a (weak) pull resistor attached, and can be one of:
  - `None` - No pull up or down resistor.
  - `Pin.PULL_UP` - Pull up resistor enabled.
  - `Pin.PULL_DOWN` - Pull down resistor enabled.
- `value` is valid only for Pin.OUT and Pin.OPEN_DRAIN modes and specifies initial output pin value if given, otherwise the state of the pin peripheral remains unchanged.
- `drive` specifies the output power of the pin and can be one of: `Pin.LOW_POWER`, `Pin.MED_POWER` or `Pin.HIGH_POWER`. The actual current driving capabilities are port dependent. Not all ports implement this argument.
- `alt` specifies an alternate function for the pin and the values it can take are port dependent. This argument is valid only for `Pin.ALT` and `Pin.ALT_OPEN_DRAIN` modes. It may be used when a pin supports more than one alternate function. If only one pin alternate function is supported the this argument is not required. Not all ports implement this argument.

As specified above, the Pin class allows to set an alternate function for a particular pin, but it does not specify any further operations on such a pin. Pins configured in alternate-function mode are usually not used as GPIO but are instead driven by other hardware peripherals. The only operation supported on such a pin is re-initialising, by calling the constructor or `Pin.init()` method. If a pin that is configured in alternate-function mode is re-initialised with `Pin.IN`, `Pin.OUT`, or `Pin.OPEN_DRAIN`, the alternate function will be removed from the pin.

Methods

Pin.init(*mode=-1, pull=-1, *, value, drive, alt*)

Re-initialise the pin using the given parameters. Only those arguments that are specified will be set. The rest of the pin peripheral state will remain unchanged. See the constructor documentation for details of the arguments.

Returns `None`.

`Pin.value([x])`

This method allows to set and get the value of the pin, depending on whether the argument `x` is supplied or not.

If the argument is omitted then this method gets the digital logic level of the pin, returning 0 or 1 corresponding to low and high voltage signals respectively. The behaviour of this method depends on the mode of the pin:

- `Pin.IN` - The method returns the actual input value currently present on the pin.
- `Pin.OUT` - The behaviour and return value of the method is undefined.
- `Pin.OPEN_DRAIN` - If the pin is in state '0' then the behaviour and return value of the method is undefined. Otherwise, if the pin is in state '1', the method returns the actual input value currently present on the pin.

If the argument is supplied then this method sets the digital logic level of the pin. The argument `x` can be anything that converts to a boolean. If it converts to `True`, the pin is set to state '1', otherwise it is set to state '0'. The behaviour of this method depends on the mode of the pin:

- `Pin.IN` - The value is stored in the output buffer for the pin. The pin state does not change, it remains in the high-impedance state. The stored value will become active on the pin as soon as it is changed to `Pin.OUT` or `Pin.OPEN_DRAIN` mode.
- `Pin.OUT` - The output buffer is set to the given value immediately.
- `Pin.OPEN_DRAIN` - If the value is '0' the pin is set to a low voltage state. Otherwise the pin is set to high-impedance state.

When setting the value this method returns `None`.

`Pin.__call__([x])`

Pin objects are callable. The call method provides a (fast) shortcut to set and get the value of the pin. It is equivalent to Pin.value([x]). See `Pin.value()` for more details.

`Pin.on()`

Set pin to "1" output level.

`Pin.off()`

Set pin to "0" output level.

Pin.mode([*mode*])

Get or set the pin mode. See the constructor documentation for details of the `mode` argument.

Pin.pull([*pull*])

Get or set the pin pull state. See the constructor documentation for details of the `pull` argument.

Pin.drive([*drive*])

Get or set the pin drive strength. See the constructor documentation for details of the `drive` argument.

Not all ports implement this method.

Availability: WiPy.

Pin.irq(*handler=None, trigger=(Pin.IRQ_FALLING | Pin.IRQ_RISING), *, priority=1, wake=None, hard=False*)

Configure an interrupt handler to be called when the trigger source of the pin is active. If the pin mode is `Pin.IN` then the trigger source is the external value on the pin. If the pin mode is `Pin.OUT` then the trigger source is the output buffer of the pin. Otherwise, if the pin mode is `Pin.OPEN_DRAIN` then the trigger source is the output buffer for state '0' and the external pin value for state '1'.

The arguments are:

- `handler` is an optional function to be called when the interrupt triggers. The handler must take exactly one argument which is the `Pin` instance.
- `trigger` configures the event which can generate an interrupt. Possible values are:
  - `Pin.IRQ_FALLING` interrupt on falling edge.
  - `Pin.IRQ_RISING` interrupt on rising edge.
  - `Pin.IRQ_LOW_LEVEL` interrupt on low level.
  - `Pin.IRQ_HIGH_LEVEL` interrupt on high level.

  These values can be OR'ed together to trigger on multiple events.
- `priority` sets the priority level of the interrupt. The values it can take are port-specific, but higher values always represent higher priorities.
- `wake` selects the power mode in which this interrupt can wake up the system. It can be `machine.IDLE`, `machine.SLEEP` or `machine.DEEPSLEEP`. These values can also be OR'ed together to make a pin generate interrupts in more than one power mode.
- `hard` if true a hardware interrupt is used. This reduces the delay between the pin change and the handler being called. Hard interrupt handlers may not allocate memory; see Writing interrupt handlers.

This method returns a callback object.

Constants

The following constants are used to configure the pin objects. Note that not all constants are available on all ports.

Pin.IN

Pin.OUT

Pin.OPEN_DRAIN

Pin.ALT

Pin.ALT_OPEN_DRAIN

Selects the pin mode.

Pin.PULL_UP

Pin.PULL_DOWN

Pin.PULL_HOLD

Selects whether there is a pull up/down resistor. Use the value `None` for no pull.

Pin.LOW_POWER

Pin.MED_POWER

Pin.HIGH_POWER

Selects the pin drive strength.

Pin.IRQ_FALLING

Pin.IRQ_RISING

Pin.IRQ_LOW_LEVEL

Pin.IRQ_HIGH_LEVEL

Selects the IRQ trigger type.

class Signal – control and sense external I/O devices

The Signal class is a simple extension of the `Pin` class. Unlike Pin, which can be only in "absolute" 0 and 1 states, a Signal can be in "asserted" (on) or "deasserted" (off) states, while being inverted (active-low) or not. In other words, it adds logical inversion support to Pin functionality. While this may seem a simple addition, it is exactly what is needed to support wide array of simple digital devices in a way portable across different boards, which is one of the major MicroPython goals. Regardless of whether different users have an active-high or active-low LED, a normally open or normally closed relay - you can develop a single, nicely looking application which works with each of them, and capture hardware configuration differences in few lines in the config file of your app.

Example:

```
from machine import Pin, Signal

# Suppose you have an active-high LED on pin 0
led1_pin = Pin(0, Pin.OUT)
# ... and active-low LED on pin 1
led2_pin = Pin(1, Pin.OUT)

# Now to light up both of them using Pin class, you'll need to set
# them to different values
led1_pin.value(1)
led2_pin.value(0)

# Signal class allows to abstract away active-high/active-low
# difference
led1 = Signal(led1_pin, invert=False)
led2 = Signal(led2_pin, invert=True)

# Now lighting up them looks the same
led1.value(1)
led2.value(1)

# Even better:
led1.on()
led2.on()
```

Following is the guide when Signal vs Pin should be used:

- Use Signal: If you want to control a simple on/off (including software PWM!) devices like LEDs, multi-segment indicators, relays, buzzers, or read simple binary sensors, like normally open or normally closed buttons, pulled high or low, Reed switches,

moisture/flame detectors, etc. etc. Summing up, if you have a real physical device/sensor requiring GPIO access, you likely should use a Signal.
- Use Pin: If you implement a higher-level protocol or bus to communicate with more complex devices.

The split between Pin and Signal come from the usecases above and the architecture of MicroPython: Pin offers the lowest overhead, which may be important when bit-banging protocols. But Signal adds additional flexibility on top of Pin, at the cost of minor overhead (much smaller than if you implemented active-high vs active-low device differences in Python manually!). Also, Pin is a low-level object which needs to be implemented for each support board, while Signal is a high-level object which comes for free once Pin is implemented.

If in doubt, give the Signal a try! Once again, it is offered to save developers from the need to handle unexciting differences like active-low vs active-high signals, and allow other users to share and enjoy your application, instead of being frustrated by the fact that it doesn't work for them simply because their LEDs or relays are wired in a slightly different way.

Constructors

class **machine.Signal**(*pin_obj, invert=False*)

class **machine.Signal**(*pin_arguments..., *, invert=False*)

Create a Signal object. There're two ways to create it:

- By wrapping existing Pin object - universal method which works for any board.
- By passing required Pin parameters directly to Signal constructor, skipping the need to create intermediate Pin object. Available on many, but not all boards.

The arguments are:

- `pin_obj` is existing Pin object.
- `pin_arguments` are the same arguments as can be passed to Pin constructor.
- `invert` - if True, the signal will be inverted (active low).

Methods

Signal.value([*x*])

This method allows to set and get the value of the signal, depending on whether the argument `x` is supplied or not.

If the argument is omitted then this method gets the signal level, 1 meaning signal is asserted (active) and 0 - signal inactive.

If the argument is supplied then this method sets the signal level. The argument `x` can be anything that converts to a boolean. If it converts to `True`, the signal is active, otherwise it is inactive.

Correspondence between signal being active and actual logic level on the underlying pin depends on whether signal is inverted (active-low) or not. For non-inverted signal, active status corresponds to logical 1, inactive - to logical 0. For inverted/active-low signal, active status corresponds to logical 0, while inactive - to logical 1.

`Signal.on()`

Activate signal.

`Signal.off()`

Deactivate signal.

class ADC – analog to digital conversion

Usage:

```
import machine

adc = machine.ADC()             # create an ADC object
apin = adc.channel(pin='GP3')   # create an analog pin on GP3
val = apin()                    # read an analog value
```

Constructors

class **machine.ADC**(*id=0, *, bits=12*)

Create an ADC object associated with the given pin. This allows you to then read analog values on that pin. For more info check the pinout and alternate functions table.

Warning

ADC pin input range is 0-1.4V (being 1.8V the absolute maximum that it can withstand). When GP2, GP3, GP4 or GP5 are remapped to the ADC block, 1.8 V is the maximum. If these pins are used in digital mode, then the maximum allowed input is 3.6V.

Methods

ADC.channel(*id, *, pin*)

Create an analog pin. If only channel ID is given, the correct pin will be selected. Alternatively, only the pin can be passed and the correct channel will be selected. Examples:

```
# all of these are equivalent and enable ADC channel 1 on GP3
apin = adc.channel(1)
apin = adc.channel(pin='GP3')
apin = adc.channel(id=1, pin='GP3')
```

ADC.init()

Enable the ADC block.

`ADC.deinit()`

Disable the ADC block.

class ADCChannel – read analog values from internal or external sources

ADC channels can be connected to internal points of the MCU or to GPIO pins. ADC channels are created using the ADC.channel method.

`machine.adcchannel()`

Fast method to read the channel value.

`adcchannel.value()`

Read the channel value.

`adcchannel.init()`

Re-init (and effectively enable) the ADC channel.

`adcchannel.deinit()`

Disable the ADC channel.

class ADC – analog to digital conversion

Usage:

```
import machine

adc = machine.ADC()             # create an ADC object
apin = adc.channel(pin='GP3')   # create an analog pin on GP3
val = apin()                    # read an analog value
```

Constructors

class **machine.ADC**(*id=0, *, bits=12*)

Create an ADC object associated with the given pin. This allows you to then read analog values on that pin. For more info check the pinout and alternate functions table.

Warning

ADC pin input range is 0-1.4V (being 1.8V the absolute maximum that it can withstand). When GP2, GP3, GP4 or GP5 are remapped to the ADC block, 1.8 V is the maximum. If these pins are used in digital mode, then the maximum allowed input is 3.6V.

Methods

ADC.channel(*id, *, pin*)

Create an analog pin. If only channel ID is given, the correct pin will be selected. Alternatively, only the pin can be passed and the correct channel will be selected. Examples:

```
# all of these are equivalent and enable ADC channel 1 on GP3
apin = adc.channel(1)
apin = adc.channel(pin='GP3')
apin = adc.channel(id=1, pin='GP3')
```

ADC.init()

Enable the ADC block.

ADC.deinit()

Disable the ADC block.

class ADCChannel – read analog values from internal or external sources

ADC channels can be connected to internal points of the MCU or to GPIO pins. ADC channels are created using the ADC.channel method.

machine.adcchannel()

Fast method to read the channel value.

adcchannel.value()

Read the channel value.

adcchannel.init()

Re-init (and effectively enable) the ADC channel.

adcchannel.deinit()

Disable the ADC channel.

class UART – duplex serial communication bus

UART implements the standard UART/USART duplex serial communications protocol. At the physical level it consists of 2 lines: RX and TX. The unit of communication is a character (not to be confused with a string character) which can be 8 or 9 bits wide.

UART objects can be created and initialised using:

```
from machine import UART

uart = UART(1, 9600)                         # init with given baudrate
uart.init(9600, bits=8, parity=None, stop=1) # init with given parameters
```

Supported parameters differ on a board:

Pyboard: Bits can be 7, 8 or 9. Stop can be 1 or 2. With *parity=None*, only 8 and 9 bits are supported. With parity enabled, only 7 and 8 bits are supported.

WiPy/CC3200: Bits can be 5, 6, 7, 8. Stop can be 1 or 2.

A UART object acts like a `stream` object and reading and writing is done using the standard stream methods:

```
uart.read(10)       # read 10 characters, returns a bytes object
uart.read()         # read all available characters
uart.readline()     # read a line
uart.readinto(buf)  # read and store into the given buffer
uart.write('abc')   # write the 3 characters
```

Constructors

class **machine.UART**(*id, ...*)

Construct a UART object of the given id.

Methods

UART.init(*baudrate=9600, bits=8, parity=None, stop=1, *, ...*)

Initialise the UART bus with the given parameters:

- *baudrate* is the clock rate.
- *bits* is the number of bits per character, 7, 8 or 9.
- *parity* is the parity, `None`, 0 (even) or 1 (odd).
- *stop* is the number of stop bits, 1 or 2.

Additional keyword-only parameters that may be supported by a port are:

- *tx* specifies the TX pin to use.
- *rx* specifies the RX pin to use.
- *txbuf* specifies the length in characters of the TX buffer.
- *rxbuf* specifies the length in characters of the RX buffer.

On the WiPy only the following keyword-only parameter is supported:

- *pins* is a 4 or 2 item list indicating the TX, RX, RTS and CTS pins (in that order). Any of the pins can be None if one wants the UART to operate with limited functionality. If the RTS pin is given the the RX pin must be given as well. The same applies to CTS. When no pins are given, then the default set of TX and RX pins is taken, and hardware flow control will be disabled. If *pins* is `None`, no pin assignment will be made.

UART.deinit()

Turn off the UART bus.

UART.any()

Returns an integer counting the number of characters that can be read without blocking. It will return 0 if there are no characters available and a positive number if there are characters. The method may return 1 even if there is more than one character available for reading.

For more sophisticated querying of available characters use select.poll:

```
poll = select.poll()
poll.register(uart, select.POLLIN)
poll.poll(timeout)
```

UART.read([*nbytes*])

Read characters. If `nbytes` is specified then read at most that many bytes, otherwise read as much data as possible.

Return value: a bytes object containing the bytes read in. Returns `None` on timeout.

UART.readinto(*buf* [, *nbytes*])

Read bytes into the `buf`. If `nbytes` is specified then read at most that many bytes. Otherwise, read at most `len(buf)` bytes.

Return value: number of bytes read and stored into `buf` or `None` on timeout.

UART.readline()

Read a line, ending in a newline character.

Return value: the line read or `None` on timeout.

UART.write(*buf*)

Write the buffer of bytes to the bus.

Return value: number of bytes written or `None` on timeout.

UART.sendbreak()

Send a break condition on the bus. This drives the bus low for a duration longer than required for a normal transmission of a character.

UART.irq(*trigger, priority=1, handler=None, wake=machine.IDLE*)

Create a callback to be triggered when data is received on the UART.

- *trigger* can only be `UART.RX_ANY`
- *priority* level of the interrupt. Can take values in the range 1-7. Higher values represent higher priorities.
- *handler* an optional function to be called when new characters arrive.
- *wake* can only be `machine.IDLE`.

> **Note**
>
> The handler will be called whenever any of the following two conditions are met:
>
> - 8 new characters have been received.
> - At least 1 new character is waiting in the Rx buffer and the Rx line has been silent for the duration of 1 complete frame.
>
> This means that when the handler function is called there will be between 1 to 8 characters waiting.

Returns an irq object.

Availability: WiPy.

Constants

UART.RX_ANY

IRQ trigger sources

Availability: WiPy.

class SPI – a Serial Peripheral Interface bus protocol (master side)

SPI is a synchronous serial protocol that is driven by a master. At the physical level, a bus consists of 3 lines: SCK, MOSI, MISO. Multiple devices can share the same bus. Each device should have a separate, 4th signal, SS (Slave Select), to select a particular device on a bus with which communication takes place. Management of an SS signal should happen in user code (via machine.Pin class).

Constructors

class **machine.SPI**(*id, ...*)

Construct an SPI object on the given bus, `id`. Values of `id` depend on a particular port and its hardware. Values 0, 1, etc. are commonly used to select hardware SPI block #0, #1, etc. Value -1 can be used for bitbanging (software) implementation of SPI (if supported by a port).

With no additional parameters, the SPI object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See `init` for parameters of initialisation.

Methods

SPI.init(*baudrate=1000000, *, polarity=0, phase=0, bits=8, firstbit=SPI.MSB, sck=None, mosi=None, miso=None, pins=(SCK, MOSI, MISO)*)

Initialise the SPI bus with the given parameters:

- `baudrate` is the SCK clock rate.
- `polarity` can be 0 or 1, and is the level the idle clock line sits at.
- `phase` can be 0 or 1 to sample data on the first or second clock edge respectively.
- `bits` is the width in bits of each transfer. Only 8 is guaranteed to be supported by all hardware.
- `firstbit` can be `SPI.MSB` or `SPI.LSB`.
- `sck`, `mosi`, `miso` are pins (machine.Pin) objects to use for bus signals. For most hardware SPI blocks (as selected by `id` parameter to the constructor), pins are fixed and cannot be changed. In some cases, hardware blocks allow 2-3 alternative pin sets for a hardware SPI block. Arbitrary pin assignments are possible only for a bitbanging SPI driver (`id` = -1).
- `pins` - WiPy port doesn't `sck`, `mosi`, `miso` arguments, and instead allows to specify them as a tuple of `pins` parameter.

In the case of hardware SPI the actual clock frequency may be lower than the requested baudrate. This is dependant on the platform hardware. The actual rate may be determined by printing the SPI object.

SPI.deinit()

Turn off the SPI bus.

SPI.read(*nbytes, write=0x00*)

Read a number of bytes specified by `nbytes` while continuously writing the single byte given by `write`. Returns a `bytes` object with the data that was read.

SPI.readinto(*buf, write=0x00*)

Read into the buffer specified by `buf` while continuously writing the single byte given by `write`. Returns `None`.

Note: on WiPy this function returns the number of bytes read.

SPI.write(*buf*)

Write the bytes contained in `buf`. Returns `None`.

Note: on WiPy this function returns the number of bytes written.

SPI.write_readinto(*write_buf, read_buf*)

Write the bytes from `write_buf` while reading into `read_buf`. The buffers can be the same or different, but both buffers must have the same length. Returns `None`.

Note: on WiPy this function returns the number of bytes written.

Constants

`SPI.MASTER`

for initialising the SPI bus to master; this is only used for the WiPy

`SPI.MSB`

set the first bit to be the most significant bit

`SPI.LSB`

set the first bit to be the least significant bit

class I2C – a two-wire serial protocol

I2C is a two-wire protocol for communicating between devices. At the physical level it consists of 2 wires: SCL and SDA, the clock and data lines respectively.

I2C objects are created attached to a specific bus. They can be initialised when created, or initialised later on.

Printing the I2C object gives you information about its configuration.

Example usage:

```
from machine import I2C

i2c = I2C(freq=400000)          # create I2C peripheral at frequency of 400kHz
                                # depending on the port, extra parameters may be required
                                # to select the peripheral and/or pins to use

i2c.scan()                      # scan for slaves, returning a list of 7-bit addresses

i2c.writeto(42, b'123')         # write 3 bytes to slave with 7-bit address 42
i2c.readfrom(42, 4)             # read 4 bytes from slave with 7-bit address 42

i2c.readfrom_mem(42, 8, 3)      # read 3 bytes from memory of slave 42,
                                #   starting at memory-address 8 in the slave
i2c.writeto_mem(42, 2, b'\x10') # write 1 byte to memory of slave 42
                                #   starting at address 2 in the slave
```

Constructors

class **machine.I2C**(*id=-1, *, scl, sda, freq=400000*)

Construct and return a new I2C object using the following parameters:

- *id* identifies a particular I2C peripheral. The default value of -1 selects a software implementation of I2C which can work (in most cases) with arbitrary pins for SCL and SDA. If *id* is -1 then *scl* and *sda* must be specified. Other allowed values for *id* depend on the particular port/board, and specifying *scl* and *sda* may or may not be required or allowed in this case.
- *scl* should be a pin object specifying the pin to use for SCL.
- *sda* should be a pin object specifying the pin to use for SDA.
- *freq* should be an integer which sets the maximum frequency for SCL.

General Methods

I2C.init(*scl*, *sda*, *, *freq=400000*)

Initialise the I2C bus with the given arguments:

- *scl* is a pin object for the SCL line
- *sda* is a pin object for the SDA line
- *freq* is the SCL clock rate

I2C.deinit()

Turn off the I2C bus.

Availability: WiPy.

I2C.scan()

Scan all I2C addresses between 0x08 and 0x77 inclusive and return a list of those that respond. A device responds if it pulls the SDA line low after its address (including a write bit) is sent on the bus.

Primitive I2C operations

The following methods implement the primitive I2C master bus operations and can be combined to make any I2C transaction. They are provided if you need more control over the bus, otherwise the standard methods (see below) can be used.

These methods are available on software I2C only.

I2C.start()

Generate a START condition on the bus (SDA transitions to low while SCL is high).

I2C.stop()

Generate a STOP condition on the bus (SDA transitions to high while SCL is high).

I2C.readinto(*buf*, *nack=True*)

Reads bytes from the bus and stores them into *buf*. The number of bytes read is the length of *buf*. An ACK will be sent on the bus after receiving all but the last byte. After the last byte is received, if *nack* is true then a NACK will be sent, otherwise an ACK will be sent (and in this case the slave assumes more bytes are going to be read in a later call).

I2C.write(*buf*)

Write the bytes from *buf* to the bus. Checks that an ACK is received after each byte and stops transmitting the remaining bytes if a NACK is received. The function returns the number of ACKs that were received.

Standard bus operations

The following methods implement the standard I2C master read and write operations that target a given slave device.

I2C.readfrom(*addr, nbytes, stop=True*)

Read *nbytes* from the slave specified by *addr*. If *stop* is true then a STOP condition is generated at the end of the transfer. Returns a `bytes` object with the data read.

I2C.readfrom_into(*addr, buf, stop=True*)

Read into *buf* from the slave specified by *addr*. The number of bytes read will be the length of *buf*. If *stop* is true then a STOP condition is generated at the end of the transfer.

The method returns `None`.

I2C.writeto(*addr, buf, stop=True*)

Write the bytes from *buf* to the slave specified by *addr*. If a NACK is received following the write of a byte from *buf* then the remaining bytes are not sent. If *stop* is true then a STOP condition is generated at the end of the transfer, even if a NACK is received. The function returns the number of ACKs that were received.

I2C.writevto(*addr, vector, stop=True*)

Write the bytes contained in *vector* to the slave specified by *addr*. *vector* should be a tuple or list of objects with the buffer protocol. The *addr* is sent once and then the bytes from each object in *vector* are written out sequentially. The objects in *vector* may be zero bytes in length in which case they don't contribute to the output.

If a NACK is received following the write of a byte from one of the objects in *vector* then the remaining bytes, and any remaining objects, are not sent. If *stop* is true then a STOP condition is generated at the end of the transfer, even if a NACK is received. The function returns the number of ACKs that were received.

Memory operations

Some I2C devices act as a memory device (or set of registers) that can be read from and written to. In this case there are two addresses associated with an I2C transaction: the slave address and the memory address. The following methods are convenience functions to communicate with such devices.

I2C.readfrom_mem(*addr, memaddr, nbytes, *, addrsize=8*)

Read *nbytes* from the slave specified by *addr* starting from the memory address specified by *memaddr*. The argument *addrsize* specifies the address size in bits. Returns a `bytes` object with the data read.

I2C.readfrom_mem_into(*addr, memaddr, buf, *, addrsize=8*)

Read into *buf* from the slave specified by *addr* starting from the memory address specified by *memaddr*. The number of bytes read is the length of *buf*. The argument *addrsize* specifies the address size in bits (on ESP8266 this argument is not recognised and the address size is always 8 bits).

The method returns `None`.

I2C.writeto_mem(*addr, memaddr, buf, *, addrsize=8*)

Write *buf* to the slave specified by *addr* starting from the memory address specified by *memaddr*. The argument *addrsize* specifies the address size in bits (on ESP8266 this argument is not recognised and the address size is always 8 bits).

The method returns `None`.

class RTC – real time clock

The RTC is and independent clock that keeps track of the date and time.

Example usage:

```
rtc = machine.RTC()
rtc.init((2014, 5, 1, 4, 13, 0, 0, 0))
print(rtc.now())
```

Constructors

class **machine.RTC**(*id=0, ...*)

Create an RTC object. See init for parameters of initialization.

Methods

RTC.init(*datetime*)

Initialise the RTC. Datetime is a tuple of the form:

```
(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]])
```

RTC.now()

Get get the current datetime tuple.

RTC.deinit()

Resets the RTC to the time of January 1, 2015 and starts running it again.

RTC.alarm(*id, time, *, repeat=False*)

Set the RTC alarm. Time might be either a millisecond value to program the alarm to current time + time_in_ms in the future, or a datetimetuple. If the time passed is in milliseconds, repeat can be set to `True` to make the alarm periodic.

RTC.alarm_left(*alarm_id=0*)

Get the number of milliseconds left before the alarm expires.

RTC.cancel(*alarm_id=0*)

Cancel a running alarm.

RTC.irq(**, trigger, handler=None, wake=machine.IDLE*)

Create an irq object triggered by a real time clock alarm.

- `trigger` must be `RTC.ALARM0`
- `handler` is the function to be called when the callback is triggered.
- `wake` specifies the sleep mode from where this interrupt can wake up the system.

Constants

RTC.ALARM0

irq trigger source

class Timer – control hardware timers

Hardware timers deal with timing of periods and events. Timers are perhaps the most flexible and heterogeneous kind of hardware in MCUs and SoCs, differently greatly from a model to a model. MicroPython's Timer class defines a baseline operation of executing a callback with a given period (or once after some delay), and allow specific boards to define more non-standard behavior (which thus won't be portable to other boards).

See discussion of important constraints on Timer callbacks.

> **Note**
>
> Memory can't be allocated inside irq handlers (an interrupt) and so exceptions raised within a handler don't give much information. See `micropython.alloc_emergency_exception_buf()` for how to get around this limitation.

If you are using a WiPy board please refer to machine.TimerWiPy instead of this class.

Constructors

class **machine.Timer**(*id, ...*)

Construct a new timer object of the given id. Id of -1 constructs a virtual timer (if supported by a board).

Methods

Timer.init(**, mode=Timer.PERIODIC, period=-1, callback=None*)

Initialise the timer. Example:

```
tim.init(period=100)                      # periodic with 100ms period
tim.init(mode=Timer.ONE_SHOT, period=1000)   # one shot firing after 1000ms
```

Keyword arguments:

- `mode` can be one of:
  - `Timer.ONE_SHOT` - The timer runs once until the configured period of the channel expires.
  - `Timer.PERIODIC` - The timer runs periodically at the configured frequency of the channel.

`Timer.deinit()`

Deinitialises the timer. Stops the timer, and disables the timer peripheral.

Constants

`Timer.ONE_SHOT`

`Timer.PERIODIC`

Timer operating mode.

class WDT – watchdog timer

The WDT is used to restart the system when the application crashes and ends up into a non recoverable state. Once started it cannot be stopped or reconfigured in any way. After enabling, the application must “feed” the watchdog periodically to prevent it from expiring and resetting the system.

Example usage:

```
from machine import WDT
wdt = WDT(timeout=2000)  # enable it with a timeout of 2s
wdt.feed()
```

Availability of this class: pyboard, WiPy.

Constructors

class **machine.WDT**(*id=0, timeout=5000*)

Create a WDT object and start it. The timeout must be given in seconds and the minimum value that is accepted is 1 second. Once it is running the timeout cannot be changed and the WDT cannot be stopped either.

Methods

wdt.feed()

Feed the WDT to prevent it from resetting the system. The application should place this call in a sensible place ensuring that the WDT is only fed after verifying that everything is functioning correctly.

class SD – secure digital memory card (cc3200 port only)

> **Warning**
>
> This is a non-standard class and is only available on the cc3200 port.

The SD card class allows to configure and enable the memory card module of the WiPy and automatically mount it as `/sd` as part of the file system. There are several pin combinations that can be used to wire the SD card socket to the WiPy and the pins used can be specified in the constructor. Please check the pinout and alternate functions table. for more info regarding the pins which can be remapped to be used with a SD card.

Example usage:

```
from machine import SD
import os
# clk cmd and dat0 pins must be passed along with
# their respective alternate functions
sd = machine.SD(pins=('GP10', 'GP11', 'GP15'))
os.mount(sd, '/sd')
# do normal file operations
```

Constructors

class **machine.SD**(*id, ...*)

Create a SD card object. See `init()` for parameters if initialization.

Methods

SD.init(*id=0, pins=('GP10', 'GP11', 'GP15')*)

Enable the SD card. In order to initialize the card, give it a 3-tuple: `(clk_pin, cmd_pin, dat0_pin)`.

SD.deinit()

Disable the SD card.

class SDCard – secure digital memory card

SD cards are one of the most common small form factor removable storage media. SD cards come in a variety of sizes and phsyical form factors. MMC cards are similar removable storage devices while eMMC devices are electically similar storage devices designed to be embedded into other systems. All three form share a common protocol for communication with their host system and high-level support looks the same for them all. As such in MicroPython they are implemented in a single class called `machine.SDCard` .

Both SD and MMC interfaces support being accessed with a variety of bus widths. When being accessed with a 1-bit wide interface they can be accessed using the SPI protocol. Different MicroPython hardware platforms support different widths and pin configurations but for most platforms there is a standard configuation for any given hardware. In general constructing an `SDCard`` object with without passing any parameters will initialise the interface to the default card slot for the current hardware. The arguments listed below represent the common arguments that might need to be set in order to use either a non-stanard slot or a non-standard pin assignment. The exact subset of arguments suported will vary from platform to platform.

---

class **machine.SDCard**(*slot=1, width=1, cd=None, wp=None, sck=None, miso=None, mosi=None, cs=None*)

This class provides access to SD or MMC storage cards using either a dedicated SD/MMC interface hardware or through an SPI channel. The class implements the block protocol defined by `uos.AbstractBlockDev`. This allows the mounting of an SD card to be as simple as:

```
uos.mount(machine.SDCard(), "/sd")
```

The constrcutor takes the following paramters:

- *slot* selects which of the available interfaces to use. Leaving this unset will select the default interface.
- *width* selects the bus width for the SD/MMC interface.
- *cd* can be used to specify a card-detect pin.
- *wp* can be used to specify a write-protect pin.
- *sck* can be used to specify an SPI clock pin.
- *miso* can be used to specify an SPI miso pin.
- *mosi* can be used to specify an SPI mosi pin.
- *cs* can be used to specify an SPI chip select pin.

Implementation-specific details

Different implementations of the `SDCard` class on different hardware support varying subsets of the options above.

PyBoard

The standard PyBoard has just one slot. No arguments are necessary or supported.

ESP32

The ESP32 provides two channels of SD/MMC hardware and also supports access to SD Cards through either of the two SPI ports that are generally available to the user. As a result the *slot* argument can take a value between 0 and 3, inclusive. Slots 0 and 1 use the built-in SD/MMC hardware while slots 2 and 3 use the SPI ports. Slot 0 supports 1, 4 or 8-bit wide access while slot 1 supports 1 or 4-bit access; the SPI slots only support 1-bit access.

> **Note**
>
> Slot 0 is used to communicate with on-board flash memory on most ESP32 modules and so will be unavailable to the user.

> **Note**
>
> Most ESP32 modules that provide an SD card slot using the dedicated hardware only wire up 1 data pin, so the default value for *width* is 1.

The pins used by the dedicated SD/MMC hardware are fixed. The pins used by the SPI hardware can be reassigned.

Note

If any of the SPI signals are remapped then all of the SPI signals will pass through a GPIO multiplexer unit which can limit the performance of high frequency signals. Since the normal operating speed for SD cards is 40MHz this can cause problems on some cards.

The default (and preferred) pin assignment are as follows:

Slot	**0**	**1**	**2**	**3**
Signal	**Pin**	**Pin**	**Pin**	**Pin**
sck	6	14	18	14
cmd	11	15		
cs			5	15
miso			19	12
mosi			23	13
D0	7	2		
D1	8	4		
D2	9	12		
D3	10	13		
D4	16			
D5	17			
D6	5			
D7	18			

cc3200

You can set the pins used for SPI access by passing a tuple as the *pins* argument.

Note: The current cc3200 SD card implementation names the this class `machine.SD` rather than `machine.SDCard` .