MicroPython Documentation

Release 1.8.7

Damien P. George, Paul Sokolovsky, and contributors

March 24, 2017

CONTENTS

1 Quick reference for the ESP8266 **1**
- 1.1 Installing MicroPython . . . 1
- 1.2 General board control . . . 1
- 1.3 Networking . . . 2
- 1.4 Delay and timing . . . 2
- 1.5 Timers . . . 3
- 1.6 Pins and GPIO . . . 3
- 1.7 PWM (pulse width modulation) . . . 3
- 1.8 ADC (analog to digital conversion) . . . 4
- 1.9 Software SPI bus . . . 4
- 1.10 Hardware SPI bus . . . 4
- 1.11 I2C bus . . . 4
- 1.12 Deep-sleep mode . . . 5
- 1.13 OneWire driver . . . 5
- 1.14 NeoPixel driver . . . 6
- 1.15 APA102 driver . . . 6
- 1.16 DHT driver . . . 6
- 1.17 WebREPL (web browser interactive prompt) . . . 7

2 MicroPython libraries **9**
- 2.1 Python standard libraries and micro-libraries . . . 9
- 2.2 MicroPython-specific libraries . . . 30
- 2.3 Libraries specific to the ESP8266 . . . 48

3 MicroPython Differences from CPython **51**
- 3.1 Syntax . . . 51
- 3.2 Core Language . . . 52
- 3.3 Builtin Types . . . 55
- 3.4 Modules . . . 61

4 MicroPython license information **65**

5 MicroPython documentation contents **67**
- 5.1 General information about the ESP8266 port . . . 67
- 5.2 MicroPython tutorial for ESP8266 . . . 69
- 5.3 The MicroPython language . . . 86

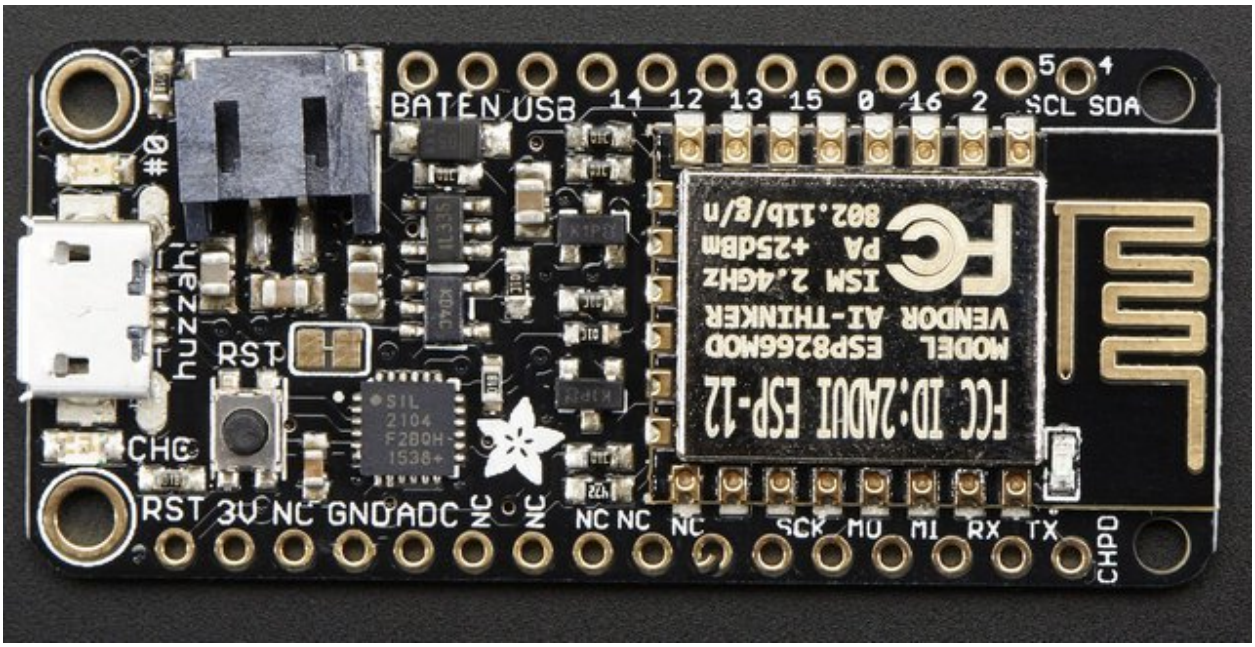
6 Indices and tables **107**

Python Module Index **109**

CHAPTER

ONE

QUICK REFERENCE FOR THE ESP8266



The Adafruit Feather HUZZAH board (image attribution: Adafruit).

1.1 Installing MicroPython

See the corresponding section of tutorial: *Getting started with MicroPython on the ESP8266*. It also includes a troubleshooting subsection.

1.2 General board control

The MicroPython REPL is on UART0 (GPIO1=TX, GPIO3=RX) at baudrate 115200. Tab-completion is useful to find out what methods an object has. Paste mode (ctrl-E) is useful to paste a large slab of Python code into the REPL.

The `machine` module:

```
import machine

machine.freq()          # get the current frequency of the CPU
machine.freq(160000000) # set the CPU frequency to 160 MHz
```

The *esp* module:

```
import esp

esp.osdebug(None)        # turn off vendor O/S debugging messages
esp.osdebug(0)           # redirect vendor O/S debugging messages to UART(0)
```

1.3 Networking

The *network* module:

```
import network

wlan = network.WLAN(network.STA_IF) # create station interface
wlan.active(True)        # activate the interface
wlan.scan()              # scan for access points
wlan.isconnected()       # check if the station is connected to an AP
wlan.connect('essid', 'password') # connect to an AP
wlan.config('mac')       # get the interface's MAC adddress
wlan.ifconfig()          # get the interface's IP/netmask/gw/DNS addresses

ap = network.WLAN(network.AP_IF) # create access-point interface
ap.active(True)          # activate the interface
ap.config(essid='ESP-AP') # set the ESSID of the access point
```

A useful function for connecting to your local WiFi network is:

```
def do_connect():
    import network
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    if not wlan.isconnected():
        print('connecting to network...')
        wlan.connect('essid', 'password')
        while not wlan.isconnected():
            pass
    print('network config:', wlan.ifconfig())
```

Once the network is established the *socket* module can be used to create and use TCP/UDP sockets as usual.

1.4 Delay and timing

Use the *time* module:

```
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(time.ticks_ms(), start) # compute time difference
```

1.5 Timers

Virtual (RTOS-based) timers are supported. Use the `machine.Timer` class with timer ID of -1:

```
from machine import Timer

tim = Timer(-1)
tim.init(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(1))
tim.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(2))
```

The period is in milliseconds.

1.6 Pins and GPIO

Use the `machine.Pin` class:

```
from machine import Pin

p0 = Pin(0, Pin.OUT)    # create output pin on GPIO0
p0.high()               # set pin to high
p0.low()                # set pin to low
p0.value(1)             # set pin to high

p2 = Pin(2, Pin.IN)     # create input pin on GPIO2
print(p2.value())       # get value, 0 or 1

p4 = Pin(4, Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor
p5 = Pin(5, Pin.OUT, value=1) # set pin high on creation
```

Available pins are: 0, 1, 2, 3, 4, 5, 12, 13, 14, 15, 16, which correspond to the actual GPIO pin numbers of ESP8266 chip. Note that many end-user boards use their own adhoc pin numbering (marked e.g. D0, D1, ...). As MicroPython supports different boards and modules, physical pin numbering was chosen as the lowest common denominator. For mapping between board logical pins and physical chip pins, consult your board documentation.

Note that Pin(1) and Pin(3) are REPL UART TX and RX respectively. Also note that Pin(16) is a special pin (used for wakeup from deepsleep mode) and may be not available for use with higher-level classes like `Neopixel`.

1.7 PWM (pulse width modulation)

PWM can be enabled on all pins except Pin(16). There is a single frequency for all channels, with range between 1 and 1000 (measured in Hz). The duty cycle is between 0 and 1023 inclusive.

Use the `machine.PWM` class:

```
from machine import Pin, PWM

pwm0 = PWM(Pin(0))      # create PWM object from a pin
pwm0.freq()             # get current frequency
pwm0.freq(1000)         # set frequency
pwm0.duty()             # get current duty cycle
pwm0.duty(200)          # set duty cycle
pwm0.deinit()           # turn off PWM on the pin

pwm2 = PWM(Pin(2), freq=500, duty=512) # create and configure in one go
```

1.8 ADC (analog to digital conversion)

ADC is available on a dedicated pin. Note that input voltages on the ADC pin must be between 0v and 1.0v.

Use the `machine.ADC` class:

```
from machine import ADC

adc = ADC(0)             # create ADC object on ADC pin
adc.read()               # read value, 0-1024
```

1.9 Software SPI bus

There are two SPI drivers. One is implemented in software (bit-banging) and works on all pins:

```
from machine import Pin, SPI

# construct an SPI bus on the given pins
# polarity is the idle state of SCK
# phase=0 means sample on the first edge of SCK, phase=1 means the second
spi = SPI(-1, baudrate=100000, polarity=1, phase=0, sck=Pin(0), mosi=Pin(2), miso=Pin(4))

spi.init(baudrate=200000) # set the baudrate

spi.read(10)             # read 10 bytes on MISO
spi.read(10, 0xff)       # read 10 bytes while outputing 0xff on MOSI

buf = bytearray(50)      # create a buffer
spi.readinto(buf)        # read into the given buffer (reads 50 bytes in this case)
spi.readinto(buf, 0xff) # read into the given buffer and output 0xff on MOSI

spi.write(b'12345')      # write 5 bytes on MOSI

buf = bytearray(4)       # create a buffer
spi.write_readinto(b'1234', buf) # write to MOSI and read from MISO into the buffer
spi.write_readinto(buf, buf) # write buf to MOSI and read MISO back into buf
```

1.10 Hardware SPI bus

The hardware SPI is faster (up to 80Mhz), but only works on following pins: `MISO` is GPIO12, `MOSI` is GPIO13, and `SCK` is GPIO14. It has the same methods as the bitbanging SPI class above, except for the pin parameters for the constructor and init (as those are fixed):

```
from machine import Pin, SPI

hspi = SPI(1, baudrate=80000000, polarity=0, phase=0)
```

(`SPI(0)` is used for FlashROM and not available to users.)

1.11 I2C bus

The I2C driver is implemented in software and works on all pins:

```
from machine import Pin, I2C

# construct an I2C bus
i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)

i2c.readfrom(0x3a, 4)    # read 4 bytes from slave device with address 0x3a
i2c.writeto(0x3a, '12')  # write '12' to slave device with address 0x3a

buf = bytearray(10)      # create a buffer with 10 bytes
i2c.writeto(0x3a, buf)   # write the given buffer to the slave
```

1.12 Deep-sleep mode

Connect GPIO16 to the reset pin (RST on HUZZAH). Then the following code can be used to sleep, wake and check the reset cause:

```
import machine

# configure RTC.ALARM0 to be able to wake the device
rtc = machine.RTC()
rtc.irq(trigger=rtc.ALARM0, wake=machine.DEEPSLEEP)

# check if the device woke from a deep sleep
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
    print('woke from a deep sleep')

# set RTC.ALARM0 to fire after 10 seconds (waking the device)
rtc.alarm(rtc.ALARM0, 10000)

# put the device to sleep
machine.deepsleep()
```

1.13 OneWire driver

The OneWire driver is implemented in software and works on all pins:

```
from machine import Pin
import onewire

ow = onewire.OneWire(Pin(12)) # create a OneWire bus on GPIO12
ow.scan()                     # return a list of devices on the bus
ow.reset()                    # reset the bus
ow.readbyte()                 # read a byte
ow.writebyte(0x12)            # write a byte on the bus
ow.write('123')               # write bytes on the bus
ow.select_rom(b'12345678') # select a specific device by its ROM code
```

There is a specific driver for DS18S20 and DS18B20 devices:

```
import time, ds18x20
ds = ds18x20.DS18X20(ow)
roms = ds.scan()
ds.convert_temp()
time.sleep_ms(750)
```

```
for rom in roms:
    print(ds.read_temp(rom))
```

Be sure to put a 4.7k pull-up resistor on the data line. Note that the `convert_temp()` method must be called each time you want to sample the temperature.

1.14 NeoPixel driver

Use the `neopixel` module:

```
from machine import Pin
from neopixel import NeoPixel

pin = Pin(0, Pin.OUT)    # set GPIO0 to output to drive NeoPixels
np = NeoPixel(pin, 8)    # create NeoPixel driver on GPIO0 for 8 pixels
np[0] = (255, 255, 255) # set the first pixel to white
np.write()               # write data to all pixels
r, g, b = np[0]          # get first pixel colour
```

For low-level driving of a NeoPixel:

```
import esp
esp.neopixel_write(pin, grb_buf, is800khz)
```

1.15 APA102 driver

Use the `apa102` module:

```
from machine import Pin
from apa102 import APA102

clock = Pin(14, Pin.OUT)      # set GPIO14 to output to drive the clock
data = Pin(13, Pin.OUT)       # set GPIO13 to output to drive the data
apa = APA102(clock, data, 8) # create APA102 driver on the clock and the data pin for 8 pixels
apa[0] = (255, 255, 255, 31) # set the first pixel to white with a maximum brightness of 31
apa.write()                   # write data to all pixels
r, g, b, brightness = apa[0] # get first pixel colour
```

For low-level driving of an APA102:

```
import esp
esp.apa102_write(clock_pin, data_pin, rgbi_buf)
```

1.16 DHT driver

The DHT driver is implemented in software and works on all pins:

```
import dht
import machine

d = dht.DHT11(machine.Pin(4))
d.measure()
```

```
d.temperature() # eg. 23 (°C)
d.humidity()    # eg. 41 (% RH)

d = dht.DHT22(machine.Pin(4))
d.measure()
d.temperature() # eg. 23.6 (°C)
d.humidity()    # eg. 41.3 (% RH)
```

1.17 WebREPL (web browser interactive prompt)

WebREPL (REPL over WebSockets, accessible via a web browser) is an experimental feature available in ESP8266 port. Download web client from https://github.com/micropython/webrepl (hosted version available at http://micropython.org/webrepl), and configure it by executing:

```
import webrepl_setup
```

and following on-screen instructions. After reboot, it will be available for connection. If you disabled automatic start-up on boot, you may run configured daemon on demand using:

```
import webrepl
webrepl.start()
```

The supported way to use WebREPL is by connecting to ESP8266 access point, but the daemon is also started on STA interface if it is active, so if your router is set up and works correctly, you may also use WebREPL while connected to your normal Internet access point (use the ESP8266 AP connection method if you face any issues).

Besides terminal/command prompt access, WebREPL also has provision for file transfer (both upload and download). Web client has buttons for the corresponding functions, or you can use command-line client `webrepl_cli.py` from the repository above.

See the MicroPython forum for other community-supported alternatives to transfer files to ESP8266.

CHAPTER

TWO

MICROPYTHON LIBRARIES

This chapter describes modules (function and class libraries) which are built into MicroPython. There are a few categories of modules:

- Modules which implement a subset of standard Python functionality and are not intended to be extended by the user.
- Modules which implement a subset of Python functionality, with a provision for extension by the user (via Python code).
- Modules which implement MicroPython extensions to the Python standard libraries.
- Modules specific to a particular port and thus not portable.

Note about the availability of modules and their contents: This documentation in general aspires to describe all modules and functions/classes which are implemented in MicroPython. However, MicroPython is highly configurable, and each port to a particular board/embedded system makes available only a subset of MicroPython libraries. For officially supported ports, there is an effort to either filter out non-applicable items, or mark individual descriptions with "Availability:" clauses describing which ports provide a given feature. With that in mind, please still be warned that some functions/classes in a module (or even the entire module) described in this documentation may be unavailable in a particular build of MicroPython on a particular board. The best place to find general information of the availability/non-availability of a particular feature is the "General Information" section which contains information pertaining to a specific port.

Beyond the built-in libraries described in this documentation, many more modules from the Python standard library, as well as further MicroPython extensions to it, can be found in the micropython-lib repository.

2.1 Python standard libraries and micro-libraries

The following standard Python libraries have been "micro-ified" to fit in with the philosophy of MicroPython. They provide the core functionality of that module and are intended to be a drop-in replacement for the standard Python library. Some modules below use a standard Python name, but prefixed with "u", e.g. `ujson` instead of `json`. This is to signify that such a module is micro-library, i.e. implements only a subset of CPython module functionality. By naming them differently, a user has a choice to write a Python-level module to extend functionality for better compatibility with CPython (indeed, this is what done by micropython-lib project mentioned above).

On some embedded platforms, where it may be cumbersome to add Python-level wrapper modules to achieve naming compatibility with CPython, micro-modules are available both by their u-name, and also by their non-u-name. The non-u-name can be overridden by a file of that name in your package path. For example, `import json` will first search for a file `json.py` or directory `json` and load that package if it is found. If nothing is found, it will fallback to loading the built-in `ujson` module.

2.1.1 Builtin Functions

All builtin functions are described here. They are also available via `builtins` module.

abs()

all()

any()

bin()

class bool

class bytearray

class bytes

callable()

chr()

classmethod()

compile()

class complex

delattr(*obj*, *name*)

The argument *name* should be a string, and this function deletes the named attribute from the object given by *obj*.

class dict

dir()

divmod()

enumerate()

eval()

exec()

filter()

class float

class frozenset

getattr()

globals()

hasattr()

hash()

hex()

id()

input()

class int

isinstance()

issubclass()

iter()

len()

class **list**

locals()

map()

max()

class **memoryview**

min()

next()

class **object**

oct()

open()

ord()

pow()

print()

property()

range()

repr()

reversed()

round()

class **set**

setattr()

class **slice**

The *slice* builtin is the type that slice objects have.

sorted()

staticmethod()

class **str**

sum()

super()

class **tuple**

type()

zip()

2.1.2 `array` – arrays of numeric data

See Python array for more information.

Supported format codes: `b`, `B`, `h`, `H`, `i`, `I`, `l`, `L`, `q`, `Q`, `f`, `d` (the latter 2 depending on the floating-point support).

Classes

class `array.`**`array`**(*typecode*[, *iterable*])

Create array with elements of given type. Initial contents of the array are given by an *iterable*. If it is not provided, an empty array is created.

`append`(*val*)

Append new element to the end of array, growing it.

`extend`(*iterable*)

Append new elements as contained in an iterable to the end of array, growing it.

2.1.3 `gc` – control the garbage collector

Functions

`gc.`**`enable`**()

Enable automatic garbage collection.

`gc.`**`disable`**()

Disable automatic garbage collection. Heap memory can still be allocated, and garbage collection can still be initiated manually using `gc.collect()`.

`gc.`**`collect`**()

Run a garbage collection.

`gc.`**`mem_alloc`**()

Return the number of bytes of heap RAM that are allocated.

`gc.`**`mem_free`**()

Return the number of bytes of available heap RAM.

2.1.4 `math` – mathematical functions

The `math` module provides some basic mathematical functions for working with floating-point numbers.

Note: On the pyboard, floating-point numbers have 32-bit precision.

Availability: not available on WiPy. Floating point support required for this module.

Functions

`math.`**`acos`**(*x*)

Return the inverse cosine of `x`.

`math.`**`acosh`**(*x*)

Return the inverse hyperbolic cosine of `x`.

`math.`**`asin`**(*x*)

Return the inverse sine of `x`.

`math.`**`asinh`**(*x*)

Return the inverse hyperbolic sine of `x`.

`math.`**`atan`**(*x*)

Return the inverse tangent of `x`.

`math.`**`atan2`**(*y*, *x*)
Return the principal value of the inverse tangent of `y/x`.

`math.`**`atanh`**(*x*)
Return the inverse hyperbolic tangent of `x`.

`math.`**`ceil`**(*x*)
Return an integer, being `x` rounded towards positive infinity.

`math.`**`copysign`**(*x*, *y*)
Return `x` with the sign of `y`.

`math.`**`cos`**(*x*)
Return the cosine of `x`.

`math.`**`cosh`**(*x*)
Return the hyperbolic cosine of `x`.

`math.`**`degrees`**(*x*)
Return radians `x` converted to degrees.

`math.`**`erf`**(*x*)
Return the error function of `x`.

`math.`**`erfc`**(*x*)
Return the complementary error function of `x`.

`math.`**`exp`**(*x*)
Return the exponential of `x`.

`math.`**`expm1`**(*x*)
Return `exp(x) - 1`.

`math.`**`fabs`**(*x*)
Return the absolute value of `x`.

`math.`**`floor`**(*x*)
Return an integer, being `x` rounded towards negative infinity.

`math.`**`fmod`**(*x*, *y*)
Return the remainder of `x/y`.

`math.`**`frexp`**(*x*)
Decomposes a floating-point number into its mantissa and exponent. The returned value is the tuple `(m, e)` such that `x == m * 2**e` exactly. If `x == 0` then the function returns `(0.0, 0)`, otherwise the relation `0.5 <= abs(m) < 1` holds.

`math.`**`gamma`**(*x*)
Return the gamma function of `x`.

`math.`**`isfinite`**(*x*)
Return `True` if `x` is finite.

`math.`**`isinf`**(*x*)
Return `True` if `x` is infinite.

`math.`**`isnan`**(*x*)
Return `True` if `x` is not-a-number

`math.`**`ldexp`**(*x*, *exp*)
Return `x * (2**exp)`.

`math.`**`lgamma`**(*x*)
Return the natural logarithm of the gamma function of `x`.

`math.log(x)`
Return the natural logarithm of `x`.

`math.log10(x)`
Return the base-10 logarithm of `x`.

`math.log2(x)`
Return the base-2 logarithm of `x`.

`math.modf(x)`
Return a tuple of two floats, being the fractional and integral parts of `x`. Both return values have the same sign as `x`.

`math.pow(x, y)`
Returns `x` to the power of `y`.

`math.radians(x)`
Return degrees `x` converted to radians.

`math.sin(x)`
Return the sine of `x`.

`math.sinh(x)`
Return the hyperbolic sine of `x`.

`math.sqrt(x)`
Return the square root of `x`.

`math.tan(x)`
Return the tangent of `x`.

`math.tanh(x)`
Return the hyperbolic tangent of `x`.

`math.trunc(x)`
Return an integer, being `x` rounded towards 0.

Constants

`math.e`
base of the natural logarithm

`math.pi`
the ratio of a circle's circumference to its diameter

2.1.5 `sys` – system specific functions

Functions

`sys.exit(retval=0)`
Terminate current program with a given exit code. Underlyingly, this function raise as `SystemExit` exception. If an argument is given, its value given as an argument to `SystemExit`.

`sys.print_exception(exc, file=sys.stdout)`
Print exception with a traceback to a file-like object *file* (or `sys.stdout` by default).

Difference to CPython

This is simplified version of a function which appears in the `traceback` module in CPython. Unlike `traceback.print_exception()`, this function takes just exception value instead of exception type, exception value, and traceback object; *file* argument should be positional; further arguments are not supported. CPython-compatible `traceback` module can be found in micropython-lib.

Constants

`sys.`**`argv`**
A mutable list of arguments the current program was started with.

`sys.`**`byteorder`**
The byte order of the system ("little" or "big").

`sys.`**`implementation`**
Object with information about the current Python implementation. For MicroPython, it has following attributes:

- *name* - string "micropython"
- *version* - tuple (major, minor, micro), e.g. (1, 7, 0)

This object is the recommended way to distinguish MicroPython from other Python implementations (note that it still may not exist in the very minimal ports).

Difference to CPython

CPython mandates more attributes for this object, but the actual useful bare minimum is implemented in MicroPython.

`sys.`**`maxsize`**
Maximum value which a native integer type can hold on the current platform, or maximum value representable by MicroPython integer type, if it's smaller than platform max value (that is the case for MicroPython ports without long int support).

This attribute is useful for detecting "bitness" of a platform (32-bit vs 64-bit, etc.). It's recommended to not compare this attribute to some value directly, but instead count number of bits in it:

```
bits = 0
v = sys.maxsize
while v:
    bits += 1
    v >>= 1
if bits > 32:
    # 64-bit (or more) platform
    ...
else:
    # 32-bit (or less) platform
    # Note that on 32-bit platform, value of bits may be less than 32
    # (e.g. 31) due to peculiarities described above, so use "> 16",
    # "> 32", "> 64" style of comparisons.
```

`sys.`**`modules`**
Dictionary of loaded modules. On some ports, it may not include builtin modules.

`sys.`**`path`**
A mutable list of directories to search for imported modules.

`sys.platform`
The platform that MicroPython is running on. For OS/RTOS ports, this is usually an identifier of the OS, e.g. `"linux"`. For baremetal ports it is an identifier of a board, e.g. "pyboard" for the original MicroPython reference board. It thus can be used to distinguish one board from another. If you need to check whether your program runs on MicroPython (vs other Python implementation), use `sys.implementation` instead.

`sys.stderr`
Standard error stream.

`sys.stdin`
Standard input stream.

`sys.stdout`
Standard output stream.

`sys.version`
Python language version that this implementation conforms to, as a string.

`sys.version_info`
Python language version that this implementation conforms to, as a tuple of ints.

2.1.6 `ubinascii` – binary/ASCII conversions

This module implements conversions between binary data and various encodings of it in ASCII form (in both directions).

Functions

`ubinascii.hexlify(data[, sep])`
Convert binary data to hexadecimal representation. Returns bytes string.

Difference to CPython

If additional argument, *sep* is supplied, it is used as a separator between hexadecimal values.

`ubinascii.unhexlify(data)`
Convert hexadecimal data to binary representation. Returns bytes string. (i.e. inverse of hexlify)

`ubinascii.a2b_base64(data)`
Convert Base64-encoded data to binary representation. Returns bytes string.

`ubinascii.b2a_base64(data)`
Encode binary data in Base64 format. Returns string.

2.1.7 `ucollections` – collection and container types

This module implements advanced collection and container types to hold/accumulate various objects.

Classes

`ucollections.namedtuple(name, fields)`
This is factory function to create a new namedtuple type with a specific name and set of fields. A namedtuple is a subclass of tuple which allows to access its fields not just by numeric index, but also with an attribute access syntax using symbolic field names. Fields is a sequence of strings specifying field names. For compatibility

with CPython it can also be a a string with space-separated field named (but this is less efficient). Example of use:

```
from ucollections import namedtuple

MyTuple = namedtuple("MyTuple", ("id", "name"))
t1 = MyTuple(1, "foo")
t2 = MyTuple(2, "bar")
print(t1.name)
assert t2.name == t2[1]
```

ucollections.**OrderedDict**(...)

`dict` type subclass which remembers and preserves the order of keys added. When ordered dict is iterated over, keys/items are returned in the order they were added:

```
from ucollections import OrderedDict

# To make benefit of ordered keys, OrderedDict should be initialized
# from sequence of (key, value) pairs.
d = OrderedDict([("z", 1), ("a", 2)])
# More items can be added as usual
d["w"] = 5
d["b"] = 3
for k, v in d.items():
    print(k, v)
```

Output:

```
z 1
a 2
w 5
b 3
```

2.1.8 `uhashlib` – hashing algorithms

This module implements binary data hashing algorithms. The exact inventory of available algorithms depends on a board. Among the algorithms which may be implemented:

- SHA256 - The current generation, modern hashing algorithm (of SHA2 series). It is suitable for cryptographically-secure purposes. Included in the MicroPython core and any board is recommended to provide this, unless it has particular code size constraints.
- SHA1 - A previous generation algorithm. Not recommended for new usages, but SHA1 is a part of number of Internet standards and existing applications, so boards targetting network connectivity and interoperatiability will try to provide this.
- MD5 - A legacy algorithm, not considered cryptographically secure. Only selected boards, targetting interoperatibility with legacy applications, will offer this.

Constructors

class uhashlib.**sha256**([*data*])

Create an SHA256 hasher object and optionally feed `data` into it.

class uhashlib.**sha1**([*data*])

Create an SHA1 hasher object and optionally feed `data` into it.

class `uhashlib.md5([data])`
Create an MD5 hasher object and optionally feed `data` into it.

Methods

`hash.update(data)`
Feed more binary data into hash.

`hash.digest()`
Return hash for all data passed through hash, as a bytes object. After this method is called, more data cannot be fed into the hash any longer.

`hash.hexdigest()`
This method is NOT implemented. Use `ubinascii.hexlify(hash.digest())` to achieve a similar effect.

2.1.9 `uheapq` – heap queue algorithm

This module implements the heap queue algorithm.

A heap queue is simply a list that has its elements stored in a certain way.

Functions

`uheapq.heappush(heap, item)`
Push the `item` onto the `heap`.

`uheapq.heappop(heap)`
Pop the first item from the `heap`, and return it. Raises IndexError if heap is empty.

`uheapq.heapify(x)`
Convert the list `x` into a heap. This is an in-place operation.

2.1.10 `uio` – input/output streams

This module contains additional types of stream (file-like) objects and helper functions.

Conceptual hierarchy

Difference to CPython

Conceptual hierarchy of stream base classes is simplified in MicroPython, as described in this section.

(Abstract) base stream classes, which serve as a foundation for behavior of all the concrete classes, adhere to few dichotomies (pair-wise classifications) in CPython. In MicroPython, they are somewhat simplified and made implicit to achieve higher efficiencies and save resources.

An important dichotomy in CPython is unbuffered vs buffered streams. In MicroPython, all streams are currently unbuffered. This is because all modern OSes, and even many RTOSes and filesystem drivers already perform buffering on their side. Adding another layer of buffering is counter- productive (an issue known as "bufferbloat") and takes precious memory. Note that there still cases where buffering may be useful, so we may introduce optional buffering support at a later time.

But in CPython, another important dichotomy is tied with "bufferedness" - it's whether a stream may incur short read/writes or not. A short read is when a user asks e.g. 10 bytes from a stream, but gets less, similarly for writes. In CPython, unbuffered streams are automatically short operation susceptible, while buffered are guarantee against them. The no short read/writes is an important trait, as it allows to develop more concise and efficient programs - something which is highly desirable for MicroPython. So, while MicroPython doesn't support buffered streams, it still provides for no-short-operations streams. Whether there will be short operations or not depends on each particular class' needs, but developers are strongly advised to favor no-short-operations behavior for the reasons stated above. For example, MicroPython sockets are guaranteed to avoid short read/writes. Actually, at this time, there is no example of a short-operations stream class in the core, and one would be a port-specific class, where such a need is governed by hardware peculiarities.

The no-short-operations behavior gets tricky in case of non-blocking streams, blocking vs non-blocking behavior being another CPython dichotomy, fully supported by MicroPython. Non-blocking streams never wait for data either to arrive or be written - they read/write whatever possible, or signal lack of data (or ability to write data). Clearly, this conflicts with "no-short-operations" policy, and indeed, a case of non-blocking buffered (and this no-short-ops) streams is convoluted in CPython - in some places, such combination is prohibited, in some it's undefined or just not documented, in some cases it raises verbose exceptions. The matter is much simpler in MicroPython: non-blocking stream are important for efficient asynchronous operations, so this property prevails on the "no-short-ops" one. So, while blocking streams will avoid short reads/writes whenever possible (the only case to get a short read is if end of file is reached, or in case of error (but errors don't return short data, but raise exceptions)), non-blocking streams may produce short data to avoid blocking the operation.

The final dichotomy is binary vs text streams. MicroPython of course supports these, but while in CPython text streams are inherently buffered, they aren't in MicroPython. (Indeed, that's one of the cases for which we may introduce buffering support.)

Note that for efficiency, MicroPython doesn't provide abstract base classes corresponding to the hierarchy above, and it's not possible to implement, or subclass, a stream class in pure Python.

Functions

`uio.`**`open`**(*name*, *mode='r'*, ***kwargs*)

Open a file. Builtin `open()` function is aliased to this function. All ports (which provide access to file system) are required to support *mode* parameter, but support for other arguments vary by port.

Classes

class `uio.`**`FileIO`**`(...)`

This is type of a file open in binary mode, e.g. using `open(name, "rb")`. You should not instantiate this class directly.

class `uio.`**`TextIOWrapper`**`(...)`

This is type of a file open in text mode, e.g. using `open(name, "rt")`. You should not instantiate this class directly.

class `uio.`**`StringIO`**`(`[*string*]`)`

class `uio.`**`BytesIO`**`(`[*string*]`)`

In-memory file-like objects for input/output. *StringIO* is used for text-mode I/O (similar to a normal file opened with "t" modifier). *BytesIO* is used for binary-mode I/O (similar to a normal file opened with "b" modifier). Initial contents of file-like objects can be specified with *string* parameter (should be normal string for *StringIO* or bytes object for *BytesIO*). All the usual file methods like `read()`, `write()`, `seek()`, `flush()`, `close()` are available on these objects, and additionally, a following method:

`getvalue``()`

Get the current contents of the underlying buffer which holds data.

2.1.11 `ujson` – JSON encoding and decoding

This modules allows to convert between Python objects and the JSON data format.

Functions

`ujson.`**`dumps`**(*obj*)
Return `obj` represented as a JSON string.

`ujson.`**`loads`**(*str*)
Parse the JSON `str` and return an object. Raises ValueError if the string is not correctly formed.

2.1.12 `uos` – basic "operating system" services

The `os` module contains functions for filesystem access and `urandom` function.

Port specifics

The filesystem has `/` as the root directory and the available physical drives are accessible from here. They are currently:

`/flash` – the internal flash filesystem

`/sd` – the SD card (if it exists)

Functions

`uos.`**`chdir`**(*path*)
Change current directory.

`uos.`**`getcwd`**()
Get the current directory.

`uos.`**`listdir`**([*dir*])
With no argument, list the current directory. Otherwise list the given directory.

`uos.`**`mkdir`**(*path*)
Create a new directory.

`uos.`**`remove`**(*path*)
Remove a file.

`uos.`**`rmdir`**(*path*)
Remove a directory.

`uos.`**`rename`**(*old_path*, *new_path*)
Rename a file.

`uos.`**`stat`**(*path*)
Get the status of a file or directory.

`uos.`**`statvfs`**(*path*)
Get the status of a fileystem.

Returns a tuple with the filesystem information in the following order:

- `f_bsize` – file system block size
- `f_frsize` – fragment size

- `f_blocks` – size of fs in f_frsize units
- `f_bfree` – number of free blocks
- `f_bavail` – number of free blocks for unpriviliged users
- `f_files` – number of inodes
- `f_ffree` – number of free inodes
- `f_favail` – number of free inodes for unpriviliged users
- `f_flag` – mount flags
- `f_namemax` – maximum filename length

Parameters related to inodes: `f_files`, `f_ffree`, `f_avail` and the `f_flags` parameter may return `0` as they can be unavailable in a port-specific implementation.

`uos.`**`sync`**`()`
Sync all filesystems.

`uos.`**`urandom`**`(`*n*`)`
Return a bytes object with n random bytes, generated by the hardware random number generator.

2.1.13 `ure` – regular expressions

This module implements regular expression operations. Regular expression syntax supported is a subset of CPython `re` module (and actually is a subset of POSIX extended regular expressions).

Supported operators are:

`'.'` Match any character.

`'[]'` Match set of characters. Individual characters and ranges are supported.

`'^'`

`'$'`

`'?'`

`'*'`

`'+'`

`'??'`

`'*?'`

`'+?'`

Counted repetitions (`{m,n}`), more advanced assertions, named groups, etc. are not supported.

Functions

`ure.`**`compile`**`(`*regex*`)`
Compile regular expression, return `regex` object.

`ure.`**`match`**`(`*regex*, *string*`)`
Match `regex` against `string`. Match always happens from starting position in a string.

`ure.search(regex, string)`
Search `regex` in a `string`. Unlike `match`, this will search string for first position which matches regex (which still may be 0 if regex is anchored).

`ure.DEBUG`
Flag value, display debug information about compiled expression.

Regex objects

Compiled regular expression. Instances of this class are created using `ure.compile()`.

`regex.match(string)`

`regex.search(string)`

`regex.split(string, max_split=-1)`

Match objects

Match objects as returned by `match()` and `search()` methods.

`match.group([index])`
Only numeric groups are supported.

2.1.14 `usocket` – socket module

This module provides access to the BSD socket interface.

See the corresponding CPython module for comparison.

Difference to CPython

CPython used to have a `socket.error` exception which is now deprecated, and is an alias of OSError. In MicroPython, use OSError directly.

Difference to CPython

For efficiency and consistency, socket objects in MicroPython implement a stream (file-like) interface directly. In CPython, you need to convert a socket to a file-like object using `makefile()` method. This method is still supported by MicroPython (but is a no-op), so where compatibility with CPython matters, be sure to use it.

Socket address format(s)

The functions below which expect a network address, accept it in the format of *(ipv4_address, port)*, where *ipv4_address* is a string with dot-notation numeric IPv4 address, e.g. `"8.8.8.8"`, and port is integer port number in the range 1-65535. Note the domain names are not accepted as *ipv4_address*, they should be resolved first using `socket.getaddrinfo()`.

Functions

socket.**socket**(*socket.AF_INET*, *socket.SOCK_STREAM*, *socket.IPPROTO_TCP*)
Create a new socket using the given address family, socket type and protocol number.

socket.**getaddrinfo**(*host*, *port*)
Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. The list of 5-tuples has following structure:

```
(family, type, proto, canonname, sockaddr)
```

The following example shows how to connect to a given url:

```
s = socket.socket()
s.connect(socket.getaddrinfo('www.micropython.org', 80)[0][-1])
```

Difference to CPython

CPython raises a `socket.gaierror` exception (OSError subclass) in case of error in this function. MicroPython doesn't have `socket.gaierror` and raises OSError directly. Note that error numbers of `getaddrinfo()` form a separate namespace and may not match error numbers from `uerrno` module. To distinguish `getaddrinfo()` errors, they are represented by negative numbers, whereas standard system errors are positive numbers (error numbers are accessible using `e.args[0]` property from an exception object). The use of negative values is a provisional detail which may change in the future.

Constants

socket.**AF_INET**
socket.**AF_INET6**
Address family types. Availability depends on a particular board.

socket.**SOCK_STREAM**
socket.**SOCK_DGRAM**
Socket types.

socket.**IPPROTO_UDP**
socket.**IPPROTO_TCP**
IP protocol numbers.

socket.SOL_*
Socket option levels (an argument to `setsockopt()`). The exact inventory depends on a board.

socket.SO_*
Socket options (an argument to `setsockopt()`). The exact inventory depends on a board.

Constants specific to WiPy:

socket.**IPPROTO_SEC**
Special protocol value to create SSL-compatible socket.

class socket

Methods

socket.**close**()
Mark the socket closed. Once that happens, all future operations on the socket object will fail. The remote end

will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to close() them explicitly, or to use a with statement around them.

`socket.bind(address)`
Bind the socket to address. The socket must not already be bound.

`socket.listen([backlog])`
Enable a server to accept connections. If backlog is specified, it must be at least 0 (if it's lower, it will be set to 0); and specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

`socket.accept()`
Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

`socket.connect(address)`
Connect to a remote socket at address.

`socket.send(bytes)`
Send data to the socket. The socket must be connected to a remote socket. Returns number of bytes sent, which may be smaller than the length of data ("short write").

`socket.sendall(bytes)`
Send all data to the socket. The socket must be connected to a remote socket. Unlike `send()`, this method will try to send all of data, by sending data chunk by chunk consecutively.

The behavior of this method on non-blocking sockets is undefined. Due to this, on MicroPython, it's recommended to use `write()` method instead, which has the same "no short writes" policy for blocking sockets, and will return number of bytes sent on non-blocking sockets.

`socket.recv(bufsize)`
Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by bufsize.

`socket.sendto(bytes, address)`
Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*.

`socket.recvfrom(bufsize)`
Receive data from the socket. The return value is a pair (bytes, address) where bytes is a bytes object representing the data received and address is the address of the socket sending the data.

`socket.setsockopt(level, optname, value)`
Set the value of the given socket option. The needed symbolic constants are defined in the socket module (SO_* etc.). The value can be an integer or a bytes-like object representing a buffer.

`socket.settimeout(value)`
Set a timeout on blocking socket operations. The value argument can be a nonnegative floating point number expressing seconds, or None. If a non-zero value is given, subsequent socket operations will raise an `OSError` exception if the timeout period value has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If None is given, the socket is put in blocking mode.

Difference to CPython

CPython raises a `socket.timeout` exception in case of timeout, which is an `OSError` subclass. MicroPython raises an OSError directly instead. If you use `except OSError:` to catch the exception, your code will

work both in MicroPython and CPython.

`socket.setblocking(flag)`

Set blocking or non-blocking mode of the socket: if flag is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain `settimeout()` calls:

- `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
- `sock.setblocking(False)` is equivalent to `sock.settimeout(0)`

`socket.makefile(mode='rb', buffering=0)`

Return a file object associated with the socket. The exact returned type depends on the arguments given to makefile(). The support is limited to binary modes only ('rb', 'wb', and 'rwb'). CPython's arguments: `encoding`, `errors` and `newline` are not supported.

Difference to CPython

As MicroPython doesn't support buffered streams, values of `buffering` parameter is ignored and treated as if it was 0 (unbuffered).

Difference to CPython

Closing the file object returned by makefile() WILL close the original socket as well.

`socket.read([size])`

Read up to size bytes from the socket. Return a bytes object. If `size` is not given, it reads all data available from the socket until `EOF`; as such the method will not return until the socket is closed. This function tries to read as much data as requested (no "short reads"). This may be not possible with non-blocking socket though, and then less data will be returned.

`socket.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. Just as `read()`, this method follows "no short reads" policy.

Return value: number of bytes read and stored into `buf`.

`socket.readline()`

Read a line, ending in a newline character.

Return value: the line read.

`socket.write(buf)`

Write the buffer of bytes to the socket. This function will try to write all data to a socket (no "short writes"). This may be not possible with a non-blocking socket though, and returned value will be less than the length of `buf`.

Return value: number of bytes written.

2.1.15 `ussl` – ssl module

This module provides access to Transport Layer Security (often known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side.

Functions

`ssl.wrap_socket`(*sock*, *server_side=False*)

Takes a stream *sock* (usually usocket.socket instance of `SOCK_STREAM` type), and returns an instance of ssl.SSLSocket, which wraps the underlying stream in an SSL context. Returned object has the usual stream interface methods like *read()*, *write()*, etc. In MicroPython, the returned object does not expose socket interface and methods like *recv()*, *send()*. In particular, a server-side SSL socket should be created from a normal socket returned from *accept()* on a non-SSL listening server socket.

Warning: Currently, this function does NOT validate server certificates, which makes an SSL connection established prone to man-in-the-middle attacks.

2.1.16 `ustruct` – pack and unpack primitive data types

See Python struct for more information.

Supported size/byte order prefixes: `@`, `<`, `>`, `!`.

Supported format codes: `b`, `B`, `h`, `H`, `i`, `I`, `l`, `L`, `q`, `Q`, `s`, `P`, `f`, `d` (the latter 2 depending on the floating-point support).

Functions

`ustruct.calcsize`(*fmt*)

Return the number of bytes needed to store the given *fmt*.

`ustruct.pack`(*fmt*, *v1*, *v2*, ...)

Pack the values *v1*, *v2*, ... according to the format string *fmt*. The return value is a bytes object encoding the values.

`ustruct.pack_into`(*fmt*, *buffer*, *offset*, *v1*, *v2*, ...)

Pack the values *v1*, *v2*, ... according to the format string *fmt* into a *buffer* starting at *offset*. *offset* may be negative to count from the end of *buffer*.

`ustruct.unpack`(*fmt*, *data*)

Unpack from the *data* according to the format string *fmt*. The return value is a tuple of the unpacked values.

`ustruct.unpack_from`(*fmt*, *data*, *offset=0*)

Unpack from the *data* starting at *offset* according to the format string *fmt*. *offset* may be negative to count from the end of *buffer*. The return value is a tuple of the unpacked values.

2.1.17 `utime` – time related functions

The `utime` module provides functions for getting the current time and date, measuring time intervals, and for delays.

Time Epoch: Unix port uses standard for POSIX systems epoch of 1970-01-01 00:00:00 UTC. However, embedded ports use epoch of 2000-01-01 00:00:00 UTC.

Maintaining actual calendar date/time: This requires a Real Time Clock (RTC). On systems with underlying OS (including some RTOS), an RTC may be implicit. Setting and maintaining actual calendar time is responsibility of OS/RTOS and is done outside of MicroPython, it just uses OS API to query date/time. On baremetal ports however system time depends on `machine.RTC()` object. The current calendar time may be set using `machine.RTC().datetime(tuple)` function, and maintained by following means:

- By a backup battery (which may be an additional, optional component for a particular board).
- Using networked time protocol (requires setup by a port/user).

- Set manually by a user on each power-up (many boards then maintain RTC time across hard resets, though some may require setting it again in such case).

If actual calendar time is not maintained with a system/MicroPython RTC, functions below which require reference to current absolute time may behave not as expected.

Functions

`utime.localtime([secs])`

Convert a time expressed in seconds since the Epoch (see above) into an 8-tuple which contains: (year, month, mday, hour, minute, second, weekday, yearday) If secs is not provided or None, then the current time from the RTC is used.

- year includes the century (for example 2014).
- month is 1-12
- mday is 1-31
- hour is 0-23
- minute is 0-59
- second is 0-59
- weekday is 0-6 for Mon-Sun
- yearday is 1-366

`utime.mktime()`

This is inverse function of localtime. It's argument is a full 8-tuple which expresses a time as per localtime. It returns an integer which is the number of seconds since Jan 1, 2000.

`utime.sleep(seconds)`

Sleep for the given number of seconds. Seconds can be a floating-point number to sleep for a fractional number of seconds. Note that other MicroPython ports may not accept floating-point argument, for compatibility with them use `sleep_ms()` and `sleep_us()` functions.

`utime.sleep_ms(ms)`

Delay for given number of milliseconds, should be positive or 0.

`utime.sleep_us(us)`

Delay for given number of microseconds, should be positive or 0.

`utime.ticks_ms()`

Returns an increasing millisecond counter with an arbitrary reference point, that wraps around after some value. This value is not explicitly exposed, but we will refer to it as `TICKS_MAX` to simplify discussion. Period of the values is `TICKS_PERIOD = TICKS_MAX + 1`. `TICKS_PERIOD` is guaranteed to be a power of two, but otherwise may differ from port to port. The same period value is used for all of `ticks_ms()`, `ticks_us()`, `ticks_cpu()` functions (for simplicity). Thus, these functions will return a value in range [`0` .. `TICKS_MAX`], inclusive, total `TICKS_PERIOD` values. Note that only non-negative values are used. For the most part, you should treat values returned by these functions as opaque. The only operations available for them are `ticks_diff()` and `ticks_add()` functions described below.

Note: Performing standard mathematical operations (+, -) or relational operators (<, <=, >, >=) directly on these value will lead to invalid result. Performing mathematical operations and then passing their results as arguments to `ticks_diff()` or `ticks_add()` will also lead to invalid results from the latter functions.

`utime.ticks_us()`

Just like `ticks_ms()` above, but in microseconds.

utime.**ticks_cpu**()

Similar to `ticks_ms()` and `ticks_us()`, but with the highest possible resolution in the system. This is usually CPU clocks, and that's why the function is named that way. But it doesn't have to be a CPU clock, some other timing source available in a system (e.g. high-resolution timer) can be used instead. The exact timing unit (resolution) of this function is not specified on `utime` module level, but documentation for a specific port may provide more specific information. This function is intended for very fine benchmarking or very tight real-time loops. Avoid using it in portable code.

Availability: Not every port implements this function.

utime.**ticks_add**(*ticks*, *delta*)

Offset ticks value by a given number, which can be either positive or negative. Given a `ticks` value, this function allows to calculate ticks value `delta` ticks before or after it, following modular-arithmetic definition of tick values (see `ticks_ms()` above). `ticks` parameter must be a direct result of call to `ticks_ms()`, `ticks_us()`, or `ticks_cpu()` functions (or from previous call to `ticks_add()`). However, `delta` can be an arbitrary integer number or numeric expression. `ticks_add()` is useful for calculating deadlines for events/tasks. (Note: you must use `ticks_diff()` function to work with deadlines.)

Examples:

```
# Find out what ticks value there was 100ms ago
print(ticks_add(time.ticks_ms(), -100))

# Calculate deadline for operation and test for it
deadline = ticks_add(time.ticks_ms(), 200)
while ticks_diff(deadline, time.ticks_ms()) > 0:
    do_a_little_of_something()

# Find out TICKS_MAX used by this port
print(ticks_add(0, -1))
```

utime.**ticks_diff**(*ticks1*, *ticks2*)

Measure ticks difference between values returned from `ticks_ms()`, `ticks_us()`, or `ticks_cpu()` functions. The argument order is the same as for subtraction operator, `ticks_diff(ticks1, ticks2)` has the same meaning as `ticks1 - ticks2`. However, values returned by `ticks_ms()`, etc. functions may wrap around, so directly using subtraction on them will produce incorrect result. That is why `ticks_diff()` is needed, it implements modular (or more specifically, ring) arithmetics to produce correct result even for wrap-around values (as long as they not too distant inbetween, see below). The function returns **signed** value in the range [`-TICKS_PERIOD/2` .. `TICKS_PERIOD/2-1`] (that's a typical range definition for two's-complement signed binary integers). If the result is negative, it means that `ticks1` occured earlier in time than `ticks2`. Otherwise, it means that `ticks1` occured after `ticks2`. This holds `only` if `ticks1` and `ticks2` are apart from each other for no more than `TICKS_PERIOD/2-1` ticks. If that does not hold, incorrect result will be returned. Specifically, if two tick values are apart for `TICKS_PERIOD/2-1` ticks, that value will be returned by the function. However, if `TICKS_PERIOD/2` of real-time ticks has passed between them, the function will return `-TICKS_PERIOD/2` instead, i.e. result value will wrap around to the negative range of possible values.

Informal rationale of the constraints above: Suppose you are locked in a room with no means to monitor passing of time except a standard 12-notch clock. Then if you look at dial-plate now, and don't look again for another 13 hours (e.g., if you fall for a long sleep), then once you finally look again, it may seem to you that only 1 hour has passed. To avoid this mistake, just look at the clock regularly. Your application should do the same. "Too long sleep" metaphor also maps directly to application behavior: don't let your application run any single task for too long. Run tasks in steps, and do time-keeping inbetween.

`ticks_diff()` is designed to accommodate various usage patterns, among them:

Polling with timeout. In this case, the order of events is known, and you will deal only with positive results of `ticks_diff()`:

```
# Wait for GPIO pin to be asserted, but at most 500us
start = time.ticks_us()
while pin.value() == 0:
    if time.ticks_diff(time.ticks_us(), start) > 500:
        raise TimeoutError
```

Scheduling events. In this case, `ticks_diff()` result may be negative if an event is overdue:

```
# This code snippet is not optimized
now = time.ticks_ms()
scheduled_time = task.scheduled_time()
if ticks_diff(now, scheduled_time) > 0:
    print("Too early, let's nap")
    sleep_ms(ticks_diff(now, scheduled_time))
    task.run()
elif ticks_diff(now, scheduled_time) == 0:
    print("Right at time!")
    task.run()
elif ticks_diff(now, scheduled_time) < 0:
    print("Oops, running late, tell task to run faster!")
    task.run(run_faster=true)
```

Note: Do not pass `time()` values to `ticks_diff()`, you should use normal mathematical operations on them. But note that `time()` may (and will) also overflow. This is known as https://en.wikipedia.org/wiki/Year_2038_problem .

`utime.time()`

Returns the number of seconds, as an integer, since the Epoch, assuming that underlying RTC is set and maintained as described above. If an RTC is not set, this function returns number of seconds since a port-specific reference point in time (for embedded boards without a battery-backed RTC, usually since power up or reset). If you want to develop portable MicroPython application, you should not rely on this function to provide higher than second precision. If you need higher precision, use `ticks_ms()` and `ticks_us()` functions, if you need calendar time, `localtime()` without an argument is a better choice.

Difference to CPython

In CPython, this function returns number of seconds since Unix epoch, 1970-01-01 00:00 UTC, as a floating-point, usually having microsecond precision. With MicroPython, only Unix port uses the same Epoch, and if floating-point precision allows, returns sub-second precision. Embedded hardware usually doesn't have floating-point precision to represent both long time ranges and subsecond precision, so they use integer value with second precision. Some embedded hardware also lacks battery-powered RTC, so returns number of seconds since last power-up or from other relative, hardware-specific point (e.g. reset).

2.1.18 `uzlib` – zlib decompression

This modules allows to decompress binary data compressed with DEFLATE algorithm (commonly used in zlib library and gzip archiver). Compression is not yet implemented.

Functions

`uzlib.decompress(data)`

Return decompressed data as bytes.

2.2 MicroPython-specific libraries

Functionality specific to the MicroPython implementation is available in the following libraries.

2.2.1 `framebuf` — Frame buffer manipulation

This module provides a general frame buffer which can be used to create bitmap images, which can then be sent to a display.

class FrameBuffer

The FrameBuffer class provides a pixel buffer which can be drawn upon with pixels, lines, rectangles, text and even other FrameBuffer's. It is useful when generating output for displays.

For example:

```
import framebuf

# FrameBuffer needs 2 bytes for every RGB565 pixel
fbuf = FrameBuffer(bytearray(10 * 100 * 2), 10, 100, framebuf.RGB565)

fbuf.fill(0)
fbuf.text('MicroPython!', 0, 0, 0xffff)
fbuf.hline(0, 10, 96, 0xffff)
```

Constructors

class `framebuf.`**`FrameBuffer`**(*buffer*, *width*, *height*, *format*, *stride*=*width*)

Construct a FrameBuffer object. The parameters are:

- *buffer* is an object with a buffer protocol which must be large enough to contain every pixel defined by the width, height and format of the FrameBuffer.
- *width* is the width of the FrameBuffer in pixels
- *height* is the height of the FrameBuffer in pixels
- *format* specifies the type of pixel used in the FrameBuffer; valid values are `framebuf.MVLSB`, `framebuf.RGB565` and `framebuf.GS4_HMSB`. MVLSB is monochrome 1-bit color, RGB565 is RGB 16-bit color, and GS4_HMSB is grayscale 4-bit color. Where a color value c is passed to a method, c is a small integer with an encoding that is dependent on the format of the FrameBuffer.
- *stride* is the number of pixels between each horizontal line of pixels in the FrameBuffer. This defaults to *width* but may need adjustments when implementing a FrameBuffer within another larger FrameBuffer or screen. The *buffer* size must accommodate an increased step size.

One must specify valid *buffer*, *width*, *height*, *format* and optionally *stride*. Invalid *buffer* size or dimensions may lead to unexpected errors.

Drawing primitive shapes

The following methods draw shapes onto the FrameBuffer.

`FrameBuffer.`**`fill`**(*c*)

Fill the entire FrameBuffer with the specified color.

`FrameBuffer.pixel(x, y[, c])`
If *c* is not given, get the color value of the specified pixel. If *c* is given, set the specified pixel to the given color.

`FrameBuffer.hline(x, y, w, c)`

`FrameBuffer.vline(x, y, h, c)`

`FrameBuffer.line(x1, y1, x2, y2, c)`
Draw a line from a set of coordinates using the given color and a thickness of 1 pixel. The *line* method draws the line up to a second set of coordinates whereas the *hline* and *vline* methods draw horizontal and vertical lines respectively up to a given length.

`FrameBuffer.rect(x, y, w, h, c)`

`FrameBuffer.fill_rect(x, y, w, h, c)`
Draw a rectangle at the given location, size and color. The *rect* method draws only a 1 pixel outline whereas the *fill_rect* method draws both the outline and interior.

Drawing text

`FrameBuffer.text(s, x, y[, c])`
Write text to the FrameBuffer using the the coordinates as the upper-left corner of the text. The color of the text can be defined by the optional argument but is otherwise a default value of 1. All characters have dimensions of 8x8 pixels and there is currently no way to change the font.

Other methods

`FrameBuffer.scroll(xstep, ystep)`
Shift the contents of the FrameBuffer by the given vector. This may leave a footprint of the previous colors in the FrameBuffer.

`FrameBuffer.blit(fbuf, x, y[, key])`
Draw another FrameBuffer on top of the current one at the given coordinates. If *key* is specified then it should be a color integer and the corresponding color will be considered transparent: all pixels with that color value will not be drawn.

This method works between FrameBuffer's utilising different formats, but the resulting colors may be unexpected due to the mismatch in color formats.

Constants

`framebuf.MVLSB`
Monochrome (1-bit) color format

`framebuf.RGB565`
Red Green Blue (16-bit, 5+6+5) color format

`framebuf.GS4_HMSB`
Grayscale (4-bit) color format

2.2.2 `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 intrerrupts). See *Writing interrupt handlers*.

Reset related functions

`machine.reset()`
Resets the device in a manner similar to pushing the external RESET button.

`machine.reset_cause()`
Get the reset cause. See *constants* for the possible return values.

Interrupt related functions

`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()`
Returns CPU frequency in hertz.

`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 (on many ports this includes system timer interrupt occurring at regular intervals on the order of millisecond).

`machine.sleep()`
Stops the CPU and disables all peripherals except for WLAN. Execution is resumed from the point where the sleep was requested. For wake up to actually happen, wake sources should be configured first.

`machine.deepsleep()`
Stops the CPU and all peripherals (including networking interfaces, if any). Execution is resumed from the main script, just as with a reset. The reset cause can be checked to know that we are coming from `machine.DEEPSLEEP`. For wake up to actually happen, wake sources should be configured first, like `Pin` change or `RTC` timeout.

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).

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

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.

Constructors

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

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

- *id* identifies the 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.

Note: on WiPy the I2C object must be in master mode for this method to be valid.

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.

`I2C.start()`

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

Availability: ESP8266.

`I2C.stop()`

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

Availability: ESP8266.

`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).

Availability: ESP8266.

`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.

Availability: ESP8266.

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.

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).

On WiPy the return value is the number of bytes read. Otherwise the return value is *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).

On WiPy the return value is the number of bytes written. Otherwise the return value is *None*.

Constants

`I2C.MASTER`
for initialising the bus to master mode

Availability: WiPy.

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.`**`out_value`**()

Return the value stored in the output buffer of a pin, regardless of its mode.

Not all ports implement this method.

`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.`**`toggle`**()

Toggle the output value of the pin. Equivalent to `pin.value(not pin.out_value())`. Returns `None`.

Not all ports implement this method.

Availability: WiPy.

`Pin.`**`id`**()

Get the pin identifier. This may return the `id` as specified in the constructor. Or it may return a canonical software-specific pin id.

`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)`
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.
- `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.

This method returns a callback object.

Attributes

class `Pin.board`
Contains all `Pin` objects supported by the board. Examples:

```
Pin.board.GP25
led = Pin(Pin.board.GP25, mode=Pin.OUT)
Pin.board.GP2.alt_list()
```

Availability: WiPy.

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`
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 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 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.

`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 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.

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.deinit()`
Deinitialises the timer. Stops the timer, and disables the timer peripheral.

Constants

`Timer.ONE_SHOT`
`Timer.PERIODIC`
Timer operating mode.

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
```

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

Methods

`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.

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.

2.2.3 `micropython` – access and control MicroPython internals

Functions

`micropython.alloc_emergency_exception_buf(size)`

Allocate `size` bytes of RAM for the emergency exception buffer (a good size is around 100 bytes). The buffer is used to create exceptions in cases when normal RAM allocation would fail (eg within an interrupt handler) and therefore give useful traceback information in these situations.

A good way to use this function is to put it at the start of your main script (eg boot.py or main.py) and then the emergency exception buffer will be active for all the code following it.

2.2.4 `network` — network configuration

This module provides network drivers and routing configuration. To use this module, a MicroPython variant/build with network capabilities must be installed. Network drivers for specific hardware are available within this module and are used to configure hardware network interface(s). Network services provided by configured interfaces are then available for use via the `socket` module.

For example:

```
# configure a specific network interface
# see below for examples of specific drivers
import network
nic = network.Driver(...)
print(nic.ifconfig())

# now use socket as usual
import socket
addr = socket.getaddrinfo('micropython.org', 80)[0][-1]
s = socket.socket()
s.connect(addr)
s.send(b'GET / HTTP/1.1\r\nHost: micropython.org\r\n\r\n')
data = s.recv(1000)
s.close()
```

Functions

`network.phy_mode([mode])`

Get or set the PHY mode.

If the `mode` parameter is provided, sets the mode to its value. If the function is called without parameters, returns the current mode.

The possible modes are defined as constants:

- `MODE_11B` – IEEE 802.11b,
- `MODE_11G` – IEEE 802.11g,
- `MODE_11N` – IEEE 802.11n.

class WLAN

This class provides a driver for WiFi network processor in the ESP8266. Example usage:

```
import network
# enable station interface and connect to WiFi access point
nic = network.WLAN(network.STA_IF)
nic.active(True)
nic.connect('your-ssid', 'your-password')
# now use sockets as usual
```

Constructors

class `network.`**`WLAN`**(*interface_id*)

Create a WLAN network interface object. Supported interfaces are `network.STA_IF` (station aka client, connects to upstream WiFi access points) and `network.AP_IF` (access point, allows other WiFi clients to connect). Availability of the methods below depends on interface type. For example, only STA interface may `connect()` to an access point.

Methods

`wlan.`**`active`**([*is_active*])

Activate ("up") or deactivate ("down") network interface, if boolean argument is passed. Otherwise, query current state if no argument is provided. Most other methods require active interface.

`wlan.`**`connect`**(*ssid*, *password*)

Connect to the specified wireless network, using the specified password.

`wlan.`**`disconnect`**()

Disconnect from the currently connected wireless network.

`wlan.`**`scan`**()

Scan for the available wireless networks.

Scanning is only possible on STA interface. Returns list of tuples with the information about WiFi access points:

(ssid, bssid, channel, RSSI, authmode, hidden)

bssid is hardware address of an access point, in binary form, returned as bytes object. You can use `ubinascii.hexlify()` to convert it to ASCII form.

There are five values for authmode:

- 0 – open
- 1 – WEP
- 2 – WPA-PSK
- 3 – WPA2-PSK

- 4 – WPA/WPA2-PSK

and two for hidden:

- 0 – visible
- 1 – hidden

`wlan.status()`

Return the current status of the wireless connection.

The possible statuses are defined as constants:

- `STAT_IDLE` – no connection and no activity,
- `STAT_CONNECTING` – connecting in progress,
- `STAT_WRONG_PASSWORD` – failed due to incorrect password,
- `STAT_NO_AP_FOUND` – failed because no access point replied,
- `STAT_CONNECT_FAIL` – failed due to other problems,
- `STAT_GOT_IP` – connection successful.

`wlan.isconnected()`

In case of STA mode, returns `True` if connected to a WiFi access point and has a valid IP address. In AP mode returns `True` when a station is connected. Returns `False` otherwise.

`wlan.ifconfig([(ip, subnet, gateway, dns)])`

Get/set IP-level network interface parameters: IP address, subnet mask, gateway and DNS server. When called with no arguments, this method returns a 4-tuple with the above information. To set the above values, pass a 4-tuple with the required information. For example:

```
nic.ifconfig(('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
```

`wlan.config('param')`

`wlan.config(param=value, ...)`

Get or set general network interface parameters. These methods allow to work with additional parameters beyond standard IP configuration (as dealt with by `wlan.ifconfig()`). These include network-specific and hardware-specific parameters. For setting parameters, keyword argument syntax should be used, multiple parameters can be set at once. For querying, parameters name should be quoted as a string, and only one parameter can be queries at time:

```
# Set WiFi access point name (formally known as ESSID) and WiFi channel
ap.config(essid='My AP', channel=11)
# Query params one by one
print(ap.config('essid'))
print(ap.config('channel'))
```

Following are commonly supported parameters (availability of a specific parameter depends on network technology type, driver, and MicroPython port).

Parameter	Description
mac	MAC address (bytes)
essid	WiFi access point name (string)
channel	WiFi channel (integer)
hidden	Whether ESSID is hidden (boolean)
authmode	Authentication mode supported (enumeration, see module constants)
password	Access password (string)

2.2.5 `uctypes` – access binary data in a structured way

This module implements "foreign data interface" for MicroPython. The idea behind it is similar to CPython's `ctypes` modules, but the actual API is different, streamlined and optimized for small size. The basic idea of the module is to define data structure layout with about the same power as the C language allows, and the access it using familiar dot-syntax to reference sub-fields.

See also:

Module *`ustruct`* Standard Python way to access binary data structures (doesn't scale well to large and complex structures).

Defining structure layout

Structure layout is defined by a "descriptor" - a Python dictionary which encodes field names as keys and other properties required to access them as associated values. Currently, uctypes requires explicit specification of offsets for each field. Offset are given in bytes from a structure start.

Following are encoding examples for various field types:

- Scalar types:

```
"field_name": uctypes.UINT32 | 0
```

  in other words, value is scalar type identifier ORed with field offset (in bytes) from the start of the structure.

- Recursive structures:

```
"sub": (2, {
    "b0": uctypes.UINT8 | 0,
    "b1": uctypes.UINT8 | 1,
})
```

  i.e. value is a 2-tuple, first element of which is offset, and second is a structure descriptor dictionary (note: offsets in recursive descriptors are relative to a structure it defines).

- Arrays of primitive types:

```
"arr": (uctypes.ARRAY | 0, uctypes.UINT8 | 2),
```

  i.e. value is a 2-tuple, first element of which is ARRAY flag ORed with offset, and second is scalar element type ORed number of elements in array.

- Arrays of aggregate types:

```
"arr2": (uctypes.ARRAY | 0, 2, {"b": uctypes.UINT8 | 0}),
```

  i.e. value is a 3-tuple, first element of which is ARRAY flag ORed with offset, second is a number of elements in array, and third is descriptor of element type.

- Pointer to a primitive type:

```
"ptr": (uctypes.PTR | 0, uctypes.UINT8),
```

  i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, and second is scalar element type.

- Pointer to an aggregate type:

```
"ptr2": (uctypes.PTR | 0, {"b": uctypes.UINT8 | 0}),
```

i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, second is descriptor of type pointed to.

- Bitfields:

```
"bitf0": uctypes.BFUINT16 | 0 | 0 << uctypes.BF_POS | 8 << uctypes.BF_LEN,
```

i.e. value is type of scalar value containing given bitfield (typenames are similar to scalar types, but prefixes with "BF"), ORed with offset for scalar value containing the bitfield, and further ORed with values for bit offset and bit length of the bitfield within scalar value, shifted by BF_POS and BF_LEN positions, respectively. Bitfield position is counted from the least significant bit, and is the number of right-most bit of a field (in other words, it's a number of bits a scalar needs to be shifted right to extra the bitfield).

In the example above, first UINT16 value will be extracted at offset 0 (this detail may be important when accessing hardware registers, where particular access size and alignment are required), and then bitfield whose rightmost bit is least-significant bit of this UINT16, and length is 8 bits, will be extracted - effectively, this will access least-significant byte of UINT16.

Note that bitfield operations are independent of target byte endianness, in particular, example above will access least-significant byte of UINT16 in both little- and big-endian structures. But it depends on the least significant bit being numbered 0. Some targets may use different numbering in their native ABI, but `uctypes` always uses normalized numbering described above.

Module contents

class `uctypes.`**`struct`**(*addr*, *descriptor*, *layout_type=NATIVE*)
Instantiate a "foreign data structure" object based on structure address in memory, descriptor (encoded as a dictionary), and layout type (see below).

`uctypes.`**`LITTLE_ENDIAN`**
Layout type for a little-endian packed structure. (Packed means that every field occupies exactly as many bytes as defined in the descriptor, i.e. the alignment is 1).

`uctypes.`**`BIG_ENDIAN`**
Layout type for a big-endian packed structure.

`uctypes.`**`NATIVE`**
Layout type for a native structure - with data endianness and alignment conforming to the ABI of the system on which MicroPython runs.

`uctypes.`**`sizeof`**(*struct*)
Return size of data structure in bytes. Argument can be either structure class or specific instantiated structure object (or its aggregate field).

`uctypes.`**`addressof`**(*obj*)
Return address of an object. Argument should be bytes, bytearray or other object supporting buffer protocol (and address of this buffer is what actually returned).

`uctypes.`**`bytes_at`**(*addr*, *size*)
Capture memory at the given address and size as bytes object. As bytes object is immutable, memory is actually duplicated and copied into bytes object, so if memory contents change later, created object retains original value.

`uctypes.`**`bytearray_at`**(*addr*, *size*)
Capture memory at the given address and size as bytearray object. Unlike bytes_at() function above, memory is captured by reference, so it can be both written too, and you will access current value at the given memory address.

Structure descriptors and instantiating structure objects

Given a structure descriptor dictionary and its layout type, you can instantiate a specific structure instance at a given memory address using `uctypes.struct()` constructor. Memory address usually comes from following sources:

- Predefined address, when accessing hardware registers on a baremetal system. Lookup these addresses in datasheet for a particular MCU/SoC.
- As a return value from a call to some FFI (Foreign Function Interface) function.
- From uctypes.addressof(), when you want to pass arguments to an FFI function, or alternatively, to access some data for I/O (for example, data read from a file or network socket).

Structure objects

Structure objects allow accessing individual fields using standard dot notation: `my_struct.substruct1.field1`. If a field is of scalar type, getting it will produce a primitive value (Python integer or float) corresponding to the value contained in a field. A scalar field can also be assigned to.

If a field is an array, its individual elements can be accessed with the standard subscript operator `[]` - both read and assigned to.

If a field is a pointer, it can be dereferenced using `[0]` syntax (corresponding to C `*` operator, though `[0]` works in C too). Subscripting a pointer with other integer values but 0 are supported too, with the same semantics as in C.

Summing up, accessing structure fields generally follows C syntax, except for pointer dereference, when you need to use `[0]` operator instead of `*`.

Limitations

Accessing non-scalar fields leads to allocation of intermediate objects to represent them. This means that special care should be taken to layout a structure which needs to be accessed when memory allocation is disabled (e.g. from an interrupt). The recommendations are:

- Avoid nested structures. For example, instead of `mcu_registers.peripheral_a.register1`, define separate layout descriptors for each peripheral, to be accessed as `peripheral_a.register1`.
- Avoid other non-scalar data, like array. For example, instead of `peripheral_a.register[0]` use `peripheral_a.register0`.

Note that these recommendations will lead to decreased readability and conciseness of layouts, so they should be used only if the need to access structure fields without allocation is anticipated (it's even possible to define 2 parallel layouts - one for normal usage, and a restricted one to use when memory allocation is prohibited).

2.3 Libraries specific to the ESP8266

The following libraries are specific to the ESP8266.

2.3.1 `esp` — functions related to the ESP8266

The `esp` module contains specific functions related to the ESP8266 module.

Functions

esp.**sleep_type**([*sleep_type*])

Get or set the sleep type.

If the `sleep_type` parameter is provided, sets the sleep type to its value. If the function is called without parameters, returns the current sleep type.

The possible sleep types are defined as constants:

- `SLEEP_NONE` – all functions enabled,
- `SLEEP_MODEM` – modem sleep, shuts down the WiFi Modem circuit.
- `SLEEP_LIGHT` – light sleep, shuts down the WiFi Modem circuit and suspends the processor periodically.

The system enters the set sleep mode automatically when possible.

esp.**deepsleep**(*time=0*)

Enter deep sleep.

The whole module powers down, except for the RTC clock circuit, which can be used to restart the module after the specified time if the pin 16 is connected to the reset pin. Otherwise the module will sleep until manually reset.

esp.**flash_id**()

Read the device ID of the flash memory.

esp.**flash_read**(*byte_offset*, *length_or_buffer*)

esp.**flash_write**(*byte_offset*, *bytes*)

esp.**flash_erase**(*sector_no*)

esp.**set_native_code_location**(*start*, *length*)

Set the location that native code will be placed for execution after it is compiled. Native code is emitted when the `@micropython.native`, `@micropython.viper` and `@micropython.asm_xtensa` decorators are applied to a function. The ESP8266 must execute code from either iRAM or the lower 1MByte of flash (which is memory mapped), and this function controls the location.

If *start* and *length* are both *None* then the native code location is set to the unused portion of memory at the end of the iRAM1 region. The size of this unused portion depends on the firmware and is typically quite small (around 500 bytes), and is enough to store a few very small functions. The advantage of using this iRAM1 region is that it does not get worn out by writing to it.

If neither *start* nor *length* are *None* then they should be integers. *start* should specify the byte offset from the beginning of the flash at which native code should be stored. *length* specifies how many bytes of flash from *start* can be used to store native code. *start* and *length* should be multiples of the sector size (being 4096 bytes). The flash will be automatically erased before writing to it so be sure to use a region of flash that is not otherwise used, for example by the firmware or the filesystem.

When using the flash to store native code *start+length* must be less than or equal to 1MByte. Note that the flash can be worn out if repeated erasures (and writes) are made so use this feature sparingly. In particular, native code needs to be recompiled and rewritten to flash on each boot (including wake from deepsleep).

In both cases above, using iRAM1 or flash, if there is no more room left in the specified region then the use of a native decorator on a function will lead to *MemoryError* exception being raised during compilation of that function.

CHAPTER

THREE

MICROPYTHON DIFFERENCES FROM CPYTHON

The operations listed in this section produce conflicting results in MicroPython when compared to standard Python.

3.1 Syntax

Generated Fri 24 Mar 2017 00:12:07 UTC

3.1.1 Spaces

uPy requires spaces between literal numbers and keywords, CPy doesn't

Sample code:

```
try:
    print(eval('1and 0'))
except SyntaxError:
    print('Should have worked')
try:
    print(eval('1or 0'))
except SyntaxError:
    print('Should have worked')
try:
    print(eval('1if 1else 0'))
except SyntaxError:
    print('Should have worked')
```

CPy output:	uPy output:

`0`
`1`
`1`
	`Should have worked`
	`Should have worked`
	`Should have worked`

3.1.2 Unicode

Unicode name escapes are not implemented

Sample code:

```
print("\N{LATIN SMALL LETTER A}")
```

CPy output:	uPy output:
a	NotImplementedError: unicode name escapes

3.2 Core Language

Generated Fri 24 Mar 2017 00:12:07 UTC

Error messages may display incorrect argument counts

Sample code:

```
try:
    [].append()
except Exception as e:
    print(e)
```

CPy output:	uPy output:
append() takes exactly one argument (0 given)	function takes 2 positional arguments but 1 were g

Method Resolution Order (MRO) is not compliant with CPython

Sample code:

```
class Foo:
    def __str__(self):
        return "Foo"

class C(tuple, Foo):
    pass

t = C((1, 2, 3))
print(t)
```

CPy output:	uPy output:
Foo	(1, 2, 3)

3.2.1 Classes

Special method __del__ not implemented for user-defined classes

Sample code:

```
import gc

class Foo():
    def __del__(self):
        print('__del__')

f = Foo()
del f

gc.collect()
```

CPy output:	uPy output:
`__del__`	

Bug when using "super" as a local

Sample code:

```
class A:
    def foo(self):
        super = [1]
        super.pop()
        print(super)

A().foo()
```

CPy output:	uPy output:
`[]`	`micropython: ../py/emitbc.c:406: mp_emit_bc_end_pa`

When inheriting from multiple classes super() only calls one class

Cause: Depth first non-exhaustive method resolution order

Sample code:

```
class A:
    def __init__(self):
        print("A.__init__")

class B(A):
    def __init__(self):
        print("B.__init__")
        super().__init__()

class C(A):
    def __init__(self):
        print("C.__init__")
        super().__init__()


class D(B,C):
    def __init__(self):
        print("D.__init__")
        super().__init__()

D()
```

CPy output:	uPy output:

`D.__init__`
`B.__init__`
`C.__init__`
`A.__init__`
	`D.__init__`
	`B.__init__`
	`A.__init__`

Calling super() getter property in subclass will return a property object, not the value

Sample code:

```
class A:
    @property
    def p(self):
        return {"a":10}

class AA(A):
    @property
    def p(self):
        return super().p

a = AA()
print(a.p)
```

CPy output:	uPy output:
`{'a': 10}`	`<property>`

3.2.2 Functions

Assign instance variable to function

Sample code:

```
def f():
    pass

f.x = 0
print(f.x)
```

CPy output:	uPy output:

`0`	`Traceback (most recent call last):`
	`  File "<stdin>", line 10, in <module>`
	`AttributeError: 'function' object has no attribute`

Unpacking function arguments in non-last position gives incorrect result

Sample code:

```
print(*(1, 2), 3)
```

CPy output:	uPy output:
`1 2 3`	`3 1 2`

3.2.3 Generator

Context manager __exit__() not called in a generator which does not run to completion

Sample code:

```
class foo(object):
    def __enter__(self):
        print('Enter')
    def __exit__(self, *args):
        print('Exit')
```

```
def bar(x):
    with foo():
        while True:
            x += 1
            yield x

def func():
    g = bar(0)
    for _ in range(3):
        print(next(g))

func()
```

CPy output:	uPy output:

`Enter`
`1`
`2`
`3`
`Exit`
	`Enter`
	`1`
	`2`
	`3`

3.2.4 import

Failed to load modules are still registered as loaded

Sample code:

```
import sys

try:
    from modules import foo
except NameError as e:
    print(e)
try:
    from modules import foo
    print('Should not get here')
except NameError as e:
    print(e)
```

CPy output:	uPy output:

`foo`
`name 'xxx' is not defined`
`foo`
`name 'xxx' is not defined`
	`foo`
	`name 'xxx' is not defined`
	`Should not get here`

3.3 Builtin Types

Generated Fri 24 Mar 2017 00:12:07 UTC

3.3.1 Exception

Exception chaining not implemented

Sample code:

```
try:
    raise TypeError
except TypeError:
    raise ValueError
```

CPy output:	uPy output:

`Traceback (most recent call last):`
`  File "<stdin>", line 8, in <module>`
`TypeError`
 	
`During handling of the above exception, another exception occurred:`
 	
`Traceback (most recent call last):`
`  File "<stdin>", line 10, in <module>`
`ValueError`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 10, in <module>`
	`ValueError:`

Assign instance variable to exception

Sample code:

```
e = Exception()
e.x = 0
print(e.x)
```

CPy output:	uPy output:

`0`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`AttributeError: 'Exception' object has no attribut`

While loop guards will obscure exception line number reporting due to being optimised onto the end of the code block

Sample code:

```
l = ["-foo", "-bar"]

i = 0
while l[i][0] == "-":
    print("iter")
    i += 1
```

CPy output:	uPy output:

`iter`
`iter`
`Traceback (most recent call last):`
`  File "<stdin>", line 10, in <module>`
`IndexError: list index out of range`
	`iter`
	`iter`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 12, in <module>`
	`IndexError: list index out of range`

Exception.__init__ raises TypeError if overridden and called by subclass

Sample code:

```
class A(Exception):
    def __init__(self):
        Exception.__init__(self)

a = A()
```

CPy output:	uPy output:

	`Traceback (most recent call last):`
	`  File "<stdin>", line 11, in <module>`
	`  File "<stdin>", line 9, in __init__`
	`TypeError: argument should be a 'Exception' not a`

3.3.2 bytearray

Array slice assignment with unsupported RHS

Sample code:

```
b = bytearray(4)
b[0:1] = [1, 2]
print(b)
```

CPy output:	uPy output:

`bytearray(b'\x01\x02\x00\x00\x00')`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError: array/bytes required on right`

3.3.3 bytes

bytes(...) with keywords not implemented

Workaround: Input the encoding format directly. eg. `print(bytes('abc', 'utf-8'))`

Sample code:

```
print(bytes('abc', encoding='utf8'))
```

CPy output:	uPy output:

`b'abc'`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: keyword argument(s) not yet i`

Bytes subscr with step != 1 not implemented

Sample code:

```
print(b'123'[0:3:2])
```

CPy output:	uPy output:

`b'13'`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: only slices with step=1 (aka`

3.3.4 float

uPy and CPython outputs formats differ

Sample code:

```
print('%.1g' % -9.9)
print('%.1e' % 9.99)
print('%.1e' % 0.999)
```

CPy output:	uPy output:

`-1e+01`
`1.0e+01`
`1.0e+00`
	`-10`
	`1.00e+01`
	`1.00e-00`

3.3.5 int

No int conversion for int-derived types available

Sample code:

```
class A(int):
    __add__ = lambda self, other: A(int(self) + other)

a = A(42)
print(a+a)
```

CPy output:	uPy output:

`84`	`Traceback (most recent call last):`
	`  File "<stdin>", line 11, in <module>`
	`  File "<stdin>", line 8, in <lambda>`
	`TypeError: can't convert A to int`

Incorrect error message when passing float into to_bytes

Sample code:

```
try:
    int('1').to_bytes(1.0)
except TypeError as e:
    print(e)
```

CPy output:	uPy output:
`integer argument expected, got float`	`function missing 1 required positional arguments`

3.3.6 list

List delete with step != 1 not implemented

Sample code:

```
l = [1, 2, 3, 4]
del l[0:4:2]
print(l)
```

CPy output:	uPy output:

`[2, 4]`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError:`

List slice-store with non-iterable on RHS is not implemented

Cause: RHS is restricted to be a tuple or list

Workaround: Use `list(<iter>)` on RHS to convert the iterable to a list

Sample code:

```
l = [10, 20]
l[0:1] = range(4)
print(l)
```

CPy output:	uPy output:

`[0, 1, 2, 3, 20]`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`TypeError: object 'range' is not a tuple or list`

List store with step != 1 not implemented

Sample code:

```
l = [1, 2, 3, 4]
l[0:4:2] = [5, 6]
print(l)
```

CPy output:	uPy output:

`[5, 2, 6, 4]`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError:`

3.3.7 str

UnicodeDecodeError not raised when expected

Sample code:

```
try:
    print(repr(str(b"\xa1\x80", 'utf8')))
    print('Should not get here')
except UnicodeDecodeError:
    print('UnicodeDecodeError')
```

CPy output:	uPy output:
`UnicodeDecodeError`	`'\u0840'` `Should not get here`

Start/end indices such as str.endswith(s, start) not implemented

Sample code:

```
print('abc'.endswith('c', 1))
```

CPy output:	uPy output:

`True`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: start/end indices`

Attributes/subscr not implemented

Sample code:

```
print('{a[0]}'.format(a=[1, 2]))
```

CPy output:	uPy output:

`1`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: attributes not supported yet`

str(...) with keywords not implemented

Workaround: Input the encoding format directly. eg `print(bytes('abc', 'utf-8'))`

Sample code:

```
print(str(b'abc', encoding='utf8'))
```

CPy output:	uPy output:

`abc`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: keyword argument(s) not yet i`

None as first argument for rsplit such as str.rsplit(None, n) not implemented

Sample code:

```
print('a a a'.rsplit(None, 1))
```

CPy output:	uPy output:

`['a a', 'a']`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: rsplit(None,n)`

Instance of a subclass of str cannot be compared for equality with an instance of a str

Sample code:

```
class S(str):
    pass

s = S('hello')
print(s == 'hello')
```

CPy output:	uPy output:
`True`	`False`

Subscript with step != 1 is not yet implemented

Sample code:

```
print('abcdefghi'[0:9:2])
```

CPy output:	uPy output:

`acegi`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: only slices with step=1 (aka`

3.3.8 tuple

Tuple load with step != 1 not implemented

Sample code:

```
print((1, 2, 3, 4)[0:4:2])
```

CPy output:	uPy output:

`(1, 3)`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: only slices with step=1 (aka`

3.4 Modules

Generated Fri 24 Mar 2017 00:12:07 UTC

3.4.1 array

Looking for integer not implemented

Sample code:

```
import array
print(1 in array.array('B', b'12'))
```

CPy output:	uPy output:

`False`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError:`

Array deletion not implemented

Sample code:

```
import array
a = array.array('b', (1, 2, 3))
del a[1]
print(a)
```

CPy output:	uPy output:

`array('b', [1, 3])`	`Traceback (most recent call last):`
	`  File "<stdin>", line 9, in <module>`
	`TypeError: 'array' object does not support item de`

Subscript with step != 1 is not yet implemented

Sample code:

```
import array
a = array.array('b', (1, 2, 3))
print(a[3:2:2])
```

CPy output:	uPy output:

`array('b')`	`Traceback (most recent call last):`
	`  File "<stdin>", line 9, in <module>`
	`NotImplementedError: only slices with step=1 (aka`

3.4.2 deque

Deque not implemented

Workaround: Use regular queues or lists creatively

Sample code:

```
import collections
D = collections.deque()
print(D)
```

CPy output:	uPy output:

`deque([])`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`AttributeError: 'module' object has no attribute '`

3.4.3 json

JSON module does not throw exception when object is not serialisable

Sample code:

```
import json
a = bytes(x for x in range(256))
try:
    z = json.dumps(a)
    x = json.loads(z)
    print('Should not get here')
except TypeError:
    print('TypeError')
```

CPy output:	uPy output:
`TypeError`	`Should not get here`

3.4.4 struct

Struct pack with too few args, not checked by uPy

Sample code:

```
import struct
try:
    print(struct.pack('bb', 1))
    print('Should not get here')
except:
    print('struct.error')
```

CPy output:	uPy output:
`struct.error`	`b'\x01\x00'` `Should not get here`

Struct pack with too many args, not checked by uPy

Sample code:

```
import struct
try:
    print(struct.pack('bb', 1, 2, 3))
    print('Should not get here')
except:
    print('struct.error')
```

CPy output:	uPy output:
`struct.error`	`b'\x01\x02'` `Should not get here`

3.4.5 sys

Override sys.stdin, sys.stdout and sys.stderr. Impossible as they are stored in read-only memory.

Sample code:

```
import sys
sys.stdin = None
print(sys.stdin)
```

CPy output:	uPy output:

`None`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`AttributeError: 'module' object has no attribute '`

CHAPTER

FOUR

MICROPYTHON LICENSE INFORMATION

The MIT License (MIT)

Copyright (c) 2013-2015 Damien P. George, and others

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER

FIVE

MICROPYTHON DOCUMENTATION CONTENTS

5.1 General information about the ESP8266 port

ESP8266 is a popular WiFi-enabled System-on-Chip (SoC) by Espressif Systems.

5.1.1 Multitude of boards

There are a multitude of modules and boards from different sources which carry the ESP8266 chip. MicroPython tries to provide a generic port which would run on as many boards/modules as possible, but there may be limitations. Adafruit Feather HUZZAH board is taken as a reference board for the port (for example, testing is performed on it). If you have another board, please make sure you have datasheet, schematics and other reference materials for your board handy to look up various aspects of your board functioning.

To make a generic ESP8266 port and support as many boards as possible, following design and implementation decision were made:

- GPIO pin numbering is based on ESP8266 chip numbering, not some "logical" numbering of a particular board. Please have the manual/pin diagram of your board at hand to find correspondence between your board pins and actual ESP8266 pins. We also encourage users of various boards to share this mapping via MicroPython forum, with the idea to collect community-maintained reference materials eventually.
- All pins which make sense to support, are supported by MicroPython (for example, pins which are used to connect SPI flash are not exposed, as they're unlikely useful for anything else, and operating on them will lead to board lock-up). However, any particular board may expose only subset of pins. Consult your board reference manual.
- Some boards may lack external pins/internal connectivity to support ESP8266 deepsleep mode.

5.1.2 Technical specifications and SoC datasheets

The datasheets and other reference material for ESP8266 chip are available from the vendor site: http://bbs.espressif.com/viewtopic.php?f=67&t=225 . They are the primary reference for the chip technical specifications, capabilities, operating modes, internal functioning, etc.

For your convenience, some of technical specifications are provided below:

- Architecture: Xtensa lx106
- CPU frequency: 80MHz overclockable to 160MHz
- Total RAM available: 96KB (part of it reserved for system)
- BootROM: 64KB

- Internal FlashROM: None
- External FlashROM: code and data, via SPI Flash. Normal sizes 512KB-4MB.
- GPIO: 16 + 1 (GPIOs are multiplexed with other functions, including external FlashROM, UART, deep sleep wake-up, etc.)
- UART: One RX/TX UART (no hardware handshaking), one TX-only UART.
- SPI: 2 SPI interfaces (one used for FlashROM).
- I2C: No native external I2C (bitbang implementation available on any pins).
- I2S: 1.
- Programming: using BootROM bootloader from UART. Due to external FlashROM and always-available BootROM bootloader, ESP8266 is not brickable.

5.1.3 Scarcity of runtime resources

ESP8266 has very modest resources (first of all, RAM memory). So, please avoid allocating too big container objects (lists, dictionaries) and buffers. There is also no full-fledged OS to keep track of resources and automatically clean them up, so that's the task of a user/user application: please be sure to close open files, sockets, etc. as soon as possible after use.

5.1.4 Boot process

On boot, MicroPython EPS8266 port executes `_boot.py` script from internal frozen modules. It mounts filesystem in FlashROM, or if it's not available, performs first-time setup of the module and creates the filesystem. This part of the boot process is considered fixed, and not available for customization for end users (even if you build from source, please refrain from changes to it; customization of early boot process is available only to advanced users and developers, who can diagnose themselves any issues arising from modifying the standard process).

Once the filesystem is mounted, `boot.py` is executed from it. The standard version of this file is created during first-time module set up and has commands to start a WebREPL daemon (disabled by default, configurable with `webrepl_setup` module), etc. This file is customizable by end users (for example, you may want to set some parameters or add other services which should be run on a module start-up). But keep in mind that incorrect modifications to boot.py may still lead to boot loops or lock ups, requiring to reflash a module from scratch. (In particular, it's recommended that you use either `webrepl_setup` module or manual editing to configure WebREPL, but not both).

As a final step of boot procedure, `main.py` is executed from filesystem, if exists. This file is a hook to start up a user application each time on boot (instead of going to REPL). For small test applications, you may name them directly as `main.py`, and upload to module, but instead it's recommended to keep your application(s) in separate files, and have just the following in `main.py`:

```
import my_app
my_app.main()
```

This will allow to keep the structure of your application clear, as well as allow to install multiple applications on a board, and switch among them.

5.1.5 Real-time clock

Due to limitations of the ESP8266 chip the internal real-time clock (RTC) will overflow every 7:45h. If a long-term working RTC time is required then `time()` or `localtime()` must be called at least once within 7 hours. MicroPython will then handle the overflow.

5.2 MicroPython tutorial for ESP8266

This tutorial is intended to get you started using MicroPython on the ESP8266 system-on-a-chip. If it is your first time it is recommended to follow the tutorial through in the order below. Otherwise the sections are mostly self contained, so feel free to skip to those that interest you.

The tutorial does not assume that you know Python, but it also does not attempt to explain any of the details of the Python language. Instead it provides you with commands that are ready to run, and hopes that you will gain a bit of Python knowledge along the way. To learn more about Python itself please refer to https://www.python.org.

5.2.1 Getting started with MicroPython on the ESP8266

Using MicroPython is a great way to get the most of your ESP8266 board. And vice versa, the ESP8266 chip is a great platform for using MicroPython. This tutorial will guide you through setting up MicroPython, getting a prompt, using WebREPL, connecting to the network and communicating with the Internet, using the hardware peripherals, and controlling some external components.

Let's get started!

Requirements

The first thing you need is a board with an ESP8266 chip. The MicroPython software supports the ESP8266 chip itself and any board should work. The main characteristic of a board is how much flash it has, how the GPIO pins are connected to the outside world, and whether it includes a built-in USB-serial convertor to make the UART available to your PC.

The minimum requirement for flash size is 512k. A board with this amount of flash will not have room for a filesystem, but otherwise is fully functional. If your board has 1Mbyte or more of flash then it will support a filesystem.

Names of pins will be given in this tutorial using the chip names (eg GPIO0) and it should be straightforward to find which pin this corresponds to on your particular board.

Powering the board

If your board has a USB connector on it then most likely it is powered through this when connected to your PC. Otherwise you will need to power it directly. Please refer to the documentation for your board for further details.

Getting the firmware

The first thing you need to do is download the most recent MicroPython firmware .bin file to load onto your ESP8266 device. You can download it from the MicroPython downloads page. From here, you have 3 main choices

- Stable firmware builds for 1024kb modules and above.
- Daily firmware builds for 1024kb modules and above.
- Daily firmware builds for 512kb modules.

The best bet is nearly always to go for the Stable firmware builds. An exception to this though is if you have an ESP8266 module with only 512kb of onboard storage. You can easily tell by trying to load a Stable firmware build and if you get the error below, then you may have to use the Daily firmware builds for 512kb modules.

> WARNING: Unlikely to work as data goes beyond end of flash.

Deploying the firmware

Once you have the MicroPython firmware (compiled code), you need to load it onto your ESP8266 device. There are two main steps to do this: first you need to put your device in boot-loader mode, and second you need to copy across the firmware. The exact procedure for these steps is highly dependent on the particular board and you will need to refer to its documentation for details.

If you have a board that has a USB connector, a USB-serial convertor, and has the DTR and RTS pins wired in a special way then deploying the firmware should be easy as all steps can be done automatically. Boards that have such features include the Adafruit Feather HUZZAH and NodeMCU boards.

For best results it is recommended to first erase the entire flash of your device before putting on new MicroPython firmware.

Currently we only support esptool.py to copy across the firmware. You can find this tool here: https://github.com/espressif/esptool/, or install it using pip:

```
pip install esptool
```

Versions starting with 1.3 support both Python 2.7 and Python 3.4 (or newer). An older version (at least 1.2.1 is needed) works fine but will require Python 2.7.

Any other flashing program should work, so feel free to try them out or refer to the documentation for your board to see its recommendations.

Using esptool.py you can erase the flash with the command:

```
esptool.py --port /dev/ttyUSB0 erase_flash
```

And then deploy the new firmware using:

```
esptool.py --port /dev/ttyUSB0 --baud 460800 write_flash --flash_size=detect 0 esp8266-20170108-v1.8
```

You might need to change the "port" setting to something else relevant for your PC. You may also need to reduce the baudrate if you get errors when flashing (eg down to 115200). The filename of the firmware should also match the file that you have.

For some boards with a particular FlashROM configuration (e.g. some variants of a NodeMCU board) you may need to use the following command to deploy the firmware (note the `-fm dio` option):

```
esptool.py --port /dev/ttyUSB0 --baud 460800 write_flash --flash_size=detect -fm dio 0 esp8266-20170
```

If the above commands run without error then MicroPython should be installed on your board!

Serial prompt

Once you have the firmware on the device you can access the REPL (Python prompt) over UART0 (GPIO1=TX, GPIO3=RX), which might be connected to a USB-serial convertor, depending on your board. The baudrate is 115200. The next part of the tutorial will discuss the prompt in more detail.

WiFi

After a fresh install and boot the device configures itself as a WiFi access point (AP) that you can connect to. The ESSID is of the form MicroPython-xxxxxx where the x's are replaced with part of the MAC address of your device (so will be the same everytime, and most likely different for all ESP8266 chips). The password for the WiFi is micropythoN (note the upper-case N). Its IP address will be 192.168.4.1 once you connect to its network. WiFi configuration will be discussed in more detail later in the tutorial.

Troubleshooting installation problems

If you experience problems during flashing or with running firmware immediately after it, here are troubleshooting recommendations:

- Be aware of and try to exclude hardware problems. There are 2 common problems: bad power source quality and worn-out/defective FlashROM. Speaking of power source, not just raw amperage is important, but also low ripple and noise/EMI in general. If you experience issues with self-made or wall-wart style power supply, try USB power from a computer. Unearthed power supplies are also known to cause problems as they source of increased EMI (electromagnetic interference) - at the very least, and may lead to electrical devices breakdown. So, you are advised to avoid using unearthed power connections when working with ESP8266 and other boards. In regard to FlashROM hardware problems, there are independent (not related to MicroPython in any way) reports (e.g.) that on some ESP8266 modules, FlashROM can be programmed as little as 20 times before programming errors occur. This is *much* less than 100,000 programming cycles cited for FlashROM chips of a type used with ESP8266 by reputable vendors, which points to either production rejects, or second-hand worn-out flash chips to be used on some (apparently cheap) modules/boards. You may want to use your best judgement about source, price, documentation, warranty, post-sales support for the modules/boards you purchase.
- The flashing instructions above use flashing speed of 460800 baud, which is good compromise between speed and stability. However, depending on your module/board, USB-UART convertor, cables, host OS, etc., the above baud rate may be too high and lead to errors. Try a more common 115200 baud rate instead in such cases.
- If lower baud rate didn't help, you may want to try older version of esptool.py, which had a different programming algorithm:

```
pip install esptool==1.0.1
```

  This version doesn't support `--flash_size=detect` option, so you will need to specify FlashROM size explicitly (in megabits). It also requires Python 2.7, so you may need to use `pip2` instead of `pip` in the command above.
- The `--flash_size` option in the commands above is mandatory. Omitting it will lead to a corrupted firmware.
- To catch incorrect flash content (e.g. from a defective sector on a chip), add `--verify` switch to the commands above.
- Additionally, you can check the firmware integrity from a MicroPython REPL prompt (assuming you were able to flash it and `--verify` option doesn't report errors):

```
import esp
esp.check_fw()
```

  If the last output value is True, the firmware is OK. Otherwise, it's corrupted and need to be reflashed correctly.
- If you experience any issues with another flashing application (not esptool.py), try esptool.py, it is a generally accepted flashing application in the ESP8266 community.
- If you still experience problems with even flashing the firmware, please refer to esptool.py project page, https://github.com/espressif/esptool for additional documentation and bug tracker where you can report problems.
- If you are able to flash firmware, but `--verify` option or `esp.check_fw()` return errors even after multiple retries, you may have a defective FlashROM chip, as explained above.

5.2.2 Getting a MicroPython REPL prompt

REPL stands for Read Evaluate Print Loop, and is the name given to the interactive MicroPython prompt that you can access on the ESP8266. Using the REPL is by far the easiest way to test out your code and run commands.

There are two ways to access the REPL: either via a wired connection through the UART serial port, or via WiFi.

REPL over the serial port

The REPL is always available on the UART0 serial peripheral, which is connected to the pins GPIO1 for TX and GPIO3 for RX. The baudrate of the REPL is 115200. If your board has a USB-serial convertor on it then you should be able to access the REPL directly from your PC. Otherwise you will need to have a way of communicating with the UART.

To access the prompt over USB-serial you need to use a terminal emulator program. On Windows TeraTerm is a good choice, on Mac you can use the built-in screen program, and Linux has picocom and minicom. Of course, there are many other terminal programs that will work, so pick your favourite!

For example, on Linux you can try running:

```
picocom /dev/ttyUSB0 -b115200
```

Once you have made the connection over the serial port you can test if it is working by hitting enter a few times. You should see the Python REPL prompt, indicated by >>>.

WebREPL - a prompt over WiFi

WebREPL allows you to use the Python prompt over WiFi, connecting through a browser. The latest versions of Firefox and Chrome are supported.

For your convenience, WebREPL client is hosted at http://micropython.org/webrepl . Alternatively, you can install it locally from the the GitHub repository https://github.com/micropython/webrepl .

Before connecting to WebREPL, you should set a password and enable it via a normal serial connection. Initial versions of MicroPython for ESP8266 came with WebREPL automatically enabled on the boot and with the ability to set a password via WiFi on the first connection, but as WebREPL was becoming more widely known and popular, the initial setup has switched to a wired connection for improved security:

```
import webrepl_setup
```

Follow the on-screen instructions and prompts. To make any changes active, you will need to reboot your device.

To use WebREPL connect your computer to the ESP8266's access point (MicroPython-xxxxxx, see the previous section about this). If you have already reconfigured your ESP8266 to connect to a router then you can skip this part.

Once you are on the same network as the ESP8266 you click the "Connect" button (if you are connecting via a router then you may need to change the IP address, by default the IP address is correct when connected to the ESP8266's access point). If the connection succeeds then you should see a password prompt.

Once you type the password configured at the setup step above, press Enter once more and you should get a prompt looking like >>>. You can now start typing Python commands!

Using the REPL

Once you have a prompt you can start experimenting! Anything you type at the prompt will be executed after you press the Enter key. MicroPython will run the code that you enter and print the result (if there is one). If there is an error with the text that you enter then an error message is printed.

Try typing the following at the prompt:

```
>>> print('hello esp8266!')
hello esp8266!
```

Note that you shouldn't type the >>> arrows, they are there to indicate that you should type the text after it at the prompt. And then the line following is what the device should respond with. In the end, once you have entered the text `print("hello esp8266!")` and pressed the Enter key, the output on your screen should look exactly like it does above.

If you already know some python you can now try some basic commands here. For example:

```
>>> 1 + 2
3
>>> 1 / 2
0.5
>>> 12**34
4922235242952026704037113243122008064
```

If your board has an LED attached to GPIO2 (the ESP-12 modules do) then you can turn it on and off using the following code:

```
>>> import machine
>>> pin = machine.Pin(2, machine.Pin.OUT)
>>> pin.high()
>>> pin.low()
```

Note that `high` might turn the LED off and `low` might turn it on (or vice versa), depending on how the LED is wired on your board.

Line editing

You can edit the current line that you are entering using the left and right arrow keys to move the cursor, as well as the delete and backspace keys. Also, pressing Home or ctrl-A moves the cursor to the start of the line, and pressing End or ctrl-E moves to the end of the line.

Input history

The REPL remembers a certain number of previous lines of text that you entered (up to 8 on the ESP8266). To recall previous lines use the up and down arrow keys.

Tab completion

Pressing the Tab key will do an auto-completion of the current word that you are entering. This can be very useful to find out functions and methods that a module or object has. Try it out by typing "ma" and then pressing Tab. It should complete to "machine" (assuming you imported machine in the above example). Then type "." and press Tab again to see a list of all the functions that the machine module has.

Line continuation and auto-indent

Certain things that you type will need "continuing", that is, will need more lines of text to make a proper Python statement. In this case the prompt will change to `...` and the cursor will auto-indent the correct amount so you can start typing the next line straight away. Try this by defining the following function:

```
>>> def toggle(p):
...     p.value(not p.value())
...
...
```

```
...
>>>
```

In the above, you needed to press the Enter key three times in a row to finish the compound statement (that's the three lines with just dots on them). The other way to finish a compound statement is to press backspace to get to the start of the line, then press the Enter key. (If you did something wrong and want to escape the continuation mode then press ctrl-C; all lines will be ignored.)

The function you just defined allows you to toggle a pin. The pin object you created earlier should still exist (recreate it if it doesn't) and you can toggle the LED using:

```
>>> toggle(pin)
```

Let's now toggle the LED in a loop (if you don't have an LED then you can just print some text instead of calling toggle, to see the effect):

```
>>> import time
>>> while True:
...     toggle(pin)
...     time.sleep_ms(500)
...
...
...
>>>
```

This will toggle the LED at 1Hz (half a second on, half a second off). To stop the toggling press ctrl-C, which will raise a KeyboardInterrupt exception and break out of the loop.

The time module provides some useful functions for making delays and doing timing. Use tab completion to find out what they are and play around with them!

Paste mode

Pressing ctrl-E will enter a special paste mode. This allows you to copy and paste a chunk of text into the REPL. If you press ctrl-E you will see the paste-mode prompt:

```
paste mode; Ctrl-C to cancel, Ctrl-D to finish
===
```

You can then paste (or type) your text in. Note that none of the special keys or commands work in paste mode (eg Tab or backspace), they are just accepted as-is. Press ctrl-D to finish entering the text and execute it.

Other control commands

There are four other control commands:

- Ctrl-A on a blank line will enter raw REPL mode. This is like a permanent paste mode, except that characters are not echoed back.
- Ctrl-B on a blank like goes to normal REPL mode.
- Ctrl-C cancels any input, or interrupts the currently running code.
- Ctrl-D on a blank line will do a soft reset.

Note that ctrl-A and ctrl-D do not work with WebREPL.

5.2.3 The internal filesystem

If your devices has 1Mbyte or more of storage then it will be set up (upon first boot) to contain a filesystem. This filesystem uses the FAT format and is stored in the flash after the MicroPython firmware.

Creating and reading files

MicroPython on the ESP8266 supports the standard way of accessing files in Python, using the built-in `open()` function.

To create a file try:

```
>>> f = open('data.txt', 'w')
>>> f.write('some data')
9
>>> f.close()
```

The "9" is the number of bytes that were written with the `write()` method. Then you can read back the contents of this new file using:

```
>>> f = open('data.txt')
>>> f.read()
'some data'
>>> f.close()
```

Note that the default mode when opening a file is to open it in read-only mode, and as a text file. Specify `'wb'` as the second argument to `open()` to open for writing in binary mode, and `'rb'` to open for reading in binary mode.

Listing file and more

The os module can be used for further control over the filesystem. First import the module:

```
>>> import os
```

Then try listing the contents of the filesystem:

```
>>> os.listdir()
['boot.py', 'port_config.py', 'data.txt']
```

You can make directories:

```
>>> os.mkdir('dir')
```

And remove entries:

```
>>> os.remove('data.txt')
```

Start up scripts

There are two files that are treated specially by the ESP8266 when it starts up: boot.py and main.py. The boot.py script is executed first (if it exists) and then once it completes the main.py script is executed. You can create these files yourself and populate them with the code that you want to run when the device starts up.

Accessing the filesystem via WebREPL

You can access the filesystem over WebREPL using the web client in a browser or via the command-line tool. Please refer to Quick Reference and Tutorial sections for more information about WebREPL.

5.2.4 Network basics

The network module is used to configure the WiFi connection. There are two WiFi interfaces, one for the station (when the ESP8266 connects to a router) and one for the access point (for other devices to connect to the ESP8266). Create instances of these objects using:

```
>>> import network
>>> sta_if = network.WLAN(network.STA_IF)
>>> ap_if = network.WLAN(network.AP_IF)
```

You can check if the interfaces are active by:

```
>>> sta_if.active()
False
>>> ap_if.active()
True
```

You can also check the network settings of the interface by:

```
>>> ap_if.ifconfig()
('192.168.4.1', '255.255.255.0', '192.168.4.1', '8.8.8.8')
```

The returned values are: IP address, netmask, gateway, DNS.

Configuration of the WiFi

Upon a fresh install the ESP8266 is configured in access point mode, so the AP_IF interface is active and the STA_IF interface is inactive. You can configure the module to connect to your own network using the STA_IF interface.

First activate the station interface:

```
>>> sta_if.active(True)
```

Then connect to your WiFi network:

```
>>> sta_if.connect('<your ESSID>', '<your password>')
```

To check if the connection is established use:

```
>>> sta_if.isconnected()
```

Once established you can check the IP address:

```
>>> sta_if.ifconfig()
('192.168.0.2', '255.255.255.0', '192.168.0.1', '8.8.8.8')
```

You can then disable the access-point interface if you no longer need it:

```
>>> ap_if.active(False)
```

Here is a function you can run (or put in your boot.py file) to automatically connect to your WiFi network:

```
def do_connect():
    import network
    sta_if = network.WLAN(network.STA_IF)
    if not sta_if.isconnected():
        print('connecting to network...')
        sta_if.active(True)
        sta_if.connect('<essid>', '<password>')
        while not sta_if.isconnected():
            pass
    print('network config:', sta_if.ifconfig())
```

Sockets

Once the WiFi is set up the way to access the network is by using sockets. A socket represents an endpoint on a network device, and when two sockets are connected together communication can proceed. Internet protocols are built on top of sockets, such as email (SMTP), the web (HTTP), telnet, ssh, among many others. Each of these protocols is assigned a specific port, which is just an integer. Given an IP address and a port number you can connect to a remote device and start talking with it.

The next part of the tutorial discusses how to use sockets to do some common and useful network tasks.

5.2.5 Network - TCP sockets

The building block of most of the internet is the TCP socket. These sockets provide a reliable stream of bytes between the connected network devices. This part of the tutorial will show how to use TCP sockets in a few different cases.

Star Wars Asciimation

The simplest thing to do is to download data from the internet. In this case we will use the Star Wars Asciimation service provided by the blinkenlights.nl website. It uses the telnet protocol on port 23 to stream data to anyone that connects. It's very simple to use because it doesn't require you to authenticate (give a username or password), you can just start downloading data straight away.

The first thing to do is make sure we have the socket module available:

```
>>> import socket
```

Then get the IP address of the server:

```
>>> addr_info = socket.getaddrinfo("towel.blinkenlights.nl", 23)
```

The `getaddrinfo` function actually returns a list of addresses, and each address has more information than we need. We want to get just the first valid address, and then just the IP address and port of the server. To do this use:

```
>>> addr = addr_info[0][-1]
```

If you type `addr_info` and `addr` at the prompt you will see exactly what information they hold.

Using the IP address we can make a socket and connect to the server:

```
>>> s = socket.socket()
>>> s.connect(addr)
```

Now that we are connected we can download and display the data:

```
>>> while True:
...     data = s.recv(500)
...     print(str(data, 'utf8'), end='')
...
```

When this loop executes it should start showing the animation (use ctrl-C to interrupt it).

You should also be able to run this same code on your PC using normal Python if you want to try it out there.

HTTP GET request

The next example shows how to download a webpage. HTTP uses port 80 and you first need to send a "GET" request before you can download anything. As part of the request you need to specify the page to retrieve.

Let's define a function that can download and print a URL:

```
def http_get(url):
    _, _, host, path = url.split('/', 3)
    addr = socket.getaddrinfo(host, 80)[0][-1]
    s = socket.socket()
    s.connect(addr)
    s.send(bytes('GET /%s HTTP/1.0\r\nHost: %s\r\n\r\n' % (path, host), 'utf8'))
    while True:
        data = s.recv(100)
        if data:
            print(str(data, 'utf8'), end='')
        else:
            break
    s.close()
```

Make sure that you import the socket module before running this function. Then you can try:

```
>>> http_get('http://micropython.org/ks/test.html')
```

This should retrieve the webpage and print the HTML to the console.

Simple HTTP server

The following code creates an simple HTTP server which serves a single webpage that contains a table with the state of all the GPIO pins:

```
import machine
pins = [machine.Pin(i, machine.Pin.IN) for i in (0, 2, 4, 5, 12, 13, 14, 15)]

html = """<!DOCTYPE html>
<html>
    <head> <title>ESP8266 Pins</title> </head>
    <body> <h1>ESP8266 Pins</h1>
        <table border="1"> <tr><th>Pin</th><th>Value</th></tr> %s </table>
    </body>
</html>
"""

import socket
addr = socket.getaddrinfo('0.0.0.0', 80)[0][-1]

s = socket.socket()
```

```
s.bind(addr)
s.listen(1)

print('listening on', addr)

while True:
    cl, addr = s.accept()
    print('client connected from', addr)
    cl_file = cl.makefile('rwb', 0)
    while True:
        line = cl_file.readline()
        if not line or line == b'\r\n':
            break
    rows = ['<tr><td>%s</td><td>%d</td></tr>' % (str(p), p.value()) for p in pins]
    response = html % '\n'.join(rows)
    cl.send(response)
    cl.close()
```

5.2.6 GPIO Pins

The way to connect your board to the external world, and control other components, is through the GPIO pins. Not all pins are available to use, in most cases only pins 0, 2, 4, 5, 12, 13, 14, 15, and 16 can be used.

The pins are available in the machine module, so make sure you import that first. Then you can create a pin using:

```
>>> pin = machine.Pin(0)
```

Here, the "0" is the pin that you want to access. Usually you want to configure the pin to be input or output, and you do this when constructing it. To make an input pin use:

```
>>> pin = machine.Pin(0, machine.Pin.IN, machine.Pin.PULL_UP)
```

You can either use PULL_UP or None for the input pull-mode. If it's not specified then it defaults to None, which is no pull resistor. You can read the value on the pin using:

```
>>> pin.value()
0
```

The pin on your board may return 0 or 1 here, depending on what it's connected to. To make an output pin use:

```
>>> pin = machine.Pin(0, machine.Pin.OUT)
```

Then set its value using:

```
>>> pin.value(0)
>>> pin.value(1)
```

Or:

```
>>> pin.low()
>>> pin.high()
```

External interrupts

All pins except number 16 can be configured to trigger a hard interrupt if their input changes. You can set code (a callback function) to be executed on the trigger.

Let's first define a callback function, which must take a single argument, being the pin that triggered the function. We will make the function just print the pin:

```
>>> def callback(p):
...     print('pin change', p)
```

Next we will create two pins and configure them as inputs:

```
>>> from machine import Pin
>>> p0 = Pin(0, Pin.IN)
>>> p2 = Pin(2, Pin.IN)
```

An finally we need to tell the pins when to trigger, and the function to call when they detect an event:

```
>>> p0.irq(trigger=Pin.IRQ_FALLING, handler=callback)
>>> p2.irq(trigger=Pin.IRQ_RISING | Pin.IRQ_FALLING, handler=callback)
```

We set pin 0 to trigger only on a falling edge of the input (when it goes from high to low), and set pin 2 to trigger on both a rising and falling edge. After entering this code you can apply high and low voltages to pins 0 and 2 to see the interrupt being executed.

A hard interrupt will trigger as soon as the event occurs and will interrupt any running code, including Python code. As such your callback functions are limited in what they can do (they cannot allocate memory, for example) and should be as short and simple as possible.

5.2.7 Pulse Width Modulation

Pulse width modulation (PWM) is a way to get an artificial analog output on a digital pin. It achieves this by rapidly toggling the pin from low to high. There are two parameters associated with this: the frequency of the toggling, and the duty cycle. The duty cycle is defined to be how long the pin is high compared with the length of a single period (low plus high time). Maximum duty cycle is when the pin is high all of the time, and minimum is when it is low all of the time.

On the ESP8266 the pins 0, 2, 4, 5, 12, 13, 14 and 15 all support PWM. The limitation is that they must all be at the same frequency, and the frequency must be between 1Hz and 1kHz.

To use PWM on a pin you must first create the pin object, for example:

```
>>> import machine
>>> p12 = machine.Pin(12)
```

Then create the PWM object using:

```
>>> pwm12 = machine.PWM(p12)
```

You can set the frequency and duty cycle using:

```
>>> pwm12.freq(500)
>>> pwm12.duty(512)
```

Note that the duty cycle is between 0 (all off) and 1023 (all on), with 512 being a 50% duty. If you print the PWM object then it will tell you its current configuration:

```
>>> pwm12
PWM(12, freq=500, duty=512)
```

You can also call the `freq()` and `duty()` methods with no arguments to get their current values.

The pin will continue to be in PWM mode until you deinitialise it using:

```
>>> pwm12.deinit()
```

Fading an LED

Let's use the PWM feature to fade an LED. Assuming your board has an LED connected to pin 2 (ESP-12 modules do) we can create an LED-PWM object using:

```
>>> led = machine.PWM(machine.Pin(2), freq=1000)
```

Notice that we can set the frequency in the PWM constructor.

For the next part we will use timing and some math, so import these modules:

```
>>> import time, math
```

Then create a function to pulse the LED:

```
>>> def pulse(l, t):
...     for i in range(20):
...         l.duty(int(math.sin(i / 10 * math.pi) * 500 + 500))
...         time.sleep_ms(t)
```

You can try this function out using:

```
>>> pulse(led, 50)
```

For a nice effect you can pulse many times in a row:

```
>>> for i in range(10):
...     pulse(led, 20)
```

Remember you can use ctrl-C to interrupt the code.

Control a hobby servo

Hobby servo motors can be controlled using PWM. They require a frequency of 50Hz and then a duty between about 40 and 115, with 77 being the centre value. If you connect a servo to the power and ground pins, and then the signal line to pin 12 (other pins will work just as well), you can control the motor using:

```
>>> servo = machine.PWM(machine.Pin(12), freq=50)
>>> servo.duty(40)
>>> servo.duty(115)
>>> servo.duty(77)
```

5.2.8 Analog to Digital Conversion

The ESP8266 has a single pin (separate to the GPIO pins) which can be used to read analog voltages and convert them to a digital value. You can construct such an ADC pin object using:

```
>>> import machine
>>> adc = machine.ADC(0)
```

Then read its value with:

```
>>> adc.read()
58
```

The values returned from the `read()` function are between 0 (for 0.0 volts) and 1024 (for 1.0 volts). Please note that this input can only tolerate a maximum of 1.0 volts and you must use a voltage divider circuit to measure larger voltages.

5.2.9 Power control

The ESP8266 provides the ability to change the CPU frequency on the fly, and enter a deep-sleep state. Both can be used to manage power consumption.

Changing the CPU frequency

The machine module has a function to get and set the CPU frequency. To get the current frequency use:

```
>>> import machine
>>> machine.freq()
80000000
```

By default the CPU runs at 80MHz. It can be change to 160MHz if you need more processing power, at the expense of current consumption:

```
>>> machine.freq(160000000)
>>> machine.freq()
160000000
```

You can change to the higher frequency just while your code does the heavy processing and then change back when its finished.

Deep-sleep mode

The deep-sleep mode will shut down the ESP8266 and all its peripherals, including the WiFi (but not including the real-time-clock, which is used to wake the chip). This drastically reduces current consumption and is a good way to make devices that can run for a while on a battery.

To be able to use the deep-sleep feature you must connect GPIO16 to the reset pin (RST on the Adafruit Feather HUZZAH board). Then the following code can be used to sleep and wake the device:

```
import machine

# configure RTC.ALARM0 to be able to wake the device
rtc = machine.RTC()
rtc.irq(trigger=rtc.ALARM0, wake=machine.DEEPSLEEP)

# set RTC.ALARM0 to fire after 10 seconds (waking the device)
rtc.alarm(rtc.ALARM0, 10000)

# put the device to sleep
machine.deepsleep()
```

Note that when the chip wakes from a deep-sleep it is completely reset, including all of the memory. The boot scripts will run as usual and you can put code in them to check the reset cause to perhaps do something different if the device just woke from a deep-sleep. For example, to print the reset cause you can use:

```
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
    print('woke from a deep sleep')
else:
    print('power on or hard reset')
```

5.2.10 Controlling 1-wire devices

The 1-wire bus is a serial bus that uses just a single wire for communication (in addition to wires for ground and power). The DS18B20 temperature sensor is a very popular 1-wire device, and here we show how to use the onewire module to read from such a device.

For the following code to work you need to have at least one DS18S20 or DS18B20 temperature sensor with its data line connected to GPIO12. You must also power the sensors and connect a 4.7k Ohm resistor between the data pin and the power pin.

```
import time
import machine
import onewire, ds18x20

# the device is on GPIO12
dat = machine.Pin(12)

# create the onewire object
ds = ds18x20.DS18X20(onewire.OneWire(dat))

# scan for devices on the bus
roms = ds.scan()
print('found devices:', roms)

# loop 10 times and print all temperatures
for i in range(10):
    print('temperatures:', end=' ')
    ds.convert_temp()
    time.sleep_ms(750)
    for rom in roms:
        print(ds.read_temp(rom), end=' ')
    print()
```

Note that you must execute the `convert_temp()` function to initiate a temperature reading, then wait at least 750ms before reading the value.

5.2.11 Controlling NeoPixels

NeoPixels, also known as WS2812 LEDs, are full-colour LEDs that are connected in serial, are individually addressable, and can have their red, green and blue components set between 0 and 255. They require precise timing to control them and there is a special neopixel module to do just this.

To create a NeoPixel object do the following:

```
>>> import machine, neopixel
>>> np = neopixel.NeoPixel(machine.Pin(4), 8)
```

This configures a NeoPixel strip on GPIO4 with 8 pixels. You can adjust the "4" (pin number) and the "8" (number of pixel) to suit your set up.

To set the colour of pixels use:

```
>>> np[0] = (255, 0, 0) # set to red, full brightness
>>> np[1] = (0, 128, 0) # set to green, half brightness
>>> np[2] = (0, 0, 64)  # set to blue, quarter brightness
```

Then use the `write()` method to output the colours to the LEDs:

```
>>> np.write()
```

The following demo function makes a fancy show on the LEDs:

```
import time

def demo(np):
    n = np.n

    # cycle
    for i in range(4 * n):
        for j in range(n):
            np[j] = (0, 0, 0)
        np[i % n] = (255, 255, 255)
        np.write()
        time.sleep_ms(25)

    # bounce
    for i in range(4 * n):
        for j in range(n):
            np[j] = (0, 0, 128)
        if (i // n) % 2 == 0:
            np[i % n] = (0, 0, 0)
        else:
            np[n - 1 - (i % n)] = (0, 0, 0)
        np.write()
        time.sleep_ms(60)

    # fade in/out
    for i in range(0, 4 * 256, 8):
        for j in range(n):
            if (i // 256) % 2 == 0:
                val = i & 0xff
            else:
                val = 255 - (i & 0xff)
            np[j] = (val, 0, 0)
        np.write()

    # clear
    for i in range(n):
        np[i] = (0, 0, 0)
    np.write()
```

Execute it using:

```
>>> demo(np)
```

5.2.12 Temperature and Humidity

DHT (Digital Humidity & Temperature) sensors are low cost digital sensors with capacitive humidity sensors and thermistors to measure the surrounding air. They feature a chip that handles analog to digital conversion and provide a 1-wire interface. Newer sensors additionally provide an I2C interface.

The DHT11 (blue) and DHT22 (white) sensors provide the same 1-wire interface, however, the DHT22 requires a separate object as it has more complex calculation. DHT22 have 1 decimal place resolution for both humidity and temperature readings. DHT11 have whole number for both.

A custom 1-wire protocol, which is different to Dallas 1-wire, is used to get the measurements from the sensor. The payload consists of a humidity value, a temperature value and a checksum.

To use the 1-wire interface, construct the objects referring to their data pin:

```
>>> import dht
>>> import machine
>>> d = dht.DHT11(machine.Pin(4))

>>> import dht
>>> import machine
>>> d = dht.DHT22(machine.Pin(4))
```

Then measure and read their values with:

```
>>> d.measure()
>>> d.temperature()
>>> d.humidity()
```

Values returned from `temperature()` are in degrees Celsius and values returned from `humidity()` are a percentage of relative humidity.

The DHT11 can be called no more than once per second and the DHT22 once every two seconds for most accurate results. Sensor accuracy will degrade over time. Each sensor supports a different operating range. Refer to the product datasheets for specifics.

In 1-wire mode, only three of the four pins are used and in I2C mode, all four pins are used. Older sensors may still have 4 pins even though they do not support I2C. The 3rd pin is simply not connected.

Pin configurations:

Sensor without I2C in 1-wire mode (eg. DHT11, DHT22, AM2301, AM2302):

> 1=VDD, 2=Data, 3=NC, 4=GND

Sensor with I2C in 1-wire mode (eg. DHT12, AM2320, AM2321, AM2322):

> 1=VDD, 2=Data, 3=GND, 4=GND

Sensor with I2C in I2C mode (eg. DHT12, AM2320, AM2321, AM2322):

> 1=VDD, 2=SDA, 3=GND, 4=SCL

You should use pull-up resistors for the Data, SDA and SCL pins.

To make newer I2C sensors work in backwards compatible 1-wire mode, you must connect both pins 3 and 4 to GND. This disables the I2C interface.

DHT22 sensors are now sold under the name AM2302 and are otherwise identical.

5.2.13 Next steps

That brings us to the end of the tutorial! Hopefully by now you have a good feel for the capabilities of MicroPython on the ESP8266 and understand how to control both the WiFi and IO aspects of the chip.

There are many features that were not covered in this tutorial. The best way to learn about them is to read the full documentation of the modules, and to experiment!

Good luck creating your Internet of Things devices!

5.3 The MicroPython language

MicroPython aims to implement the Python 3.4 standard, and most of the features of MicroPython are identical to those described by the documentation at docs.python.org.

Differences to standard Python as well as additional features of MicroPython are described in the sections here.

5.3.1 The MicroPython Interactive Interpreter Mode (aka REPL)

This section covers some characteristics of the MicroPython Interactive Interpreter Mode. A commonly used term for this is REPL (read-eval-print-loop) which will be used to refer to this interactive prompt.

Auto-indent

When typing python statements which end in a colon (for example if, for, while) then the prompt will change to three dots (...) and the cursor will be indented by 4 spaces. When you press return, the next line will continue at the same level of indentation for regular statements or an additional level of indentation where appropriate. If you press the backspace key then it will undo one level of indentation.

If your cursor is all the way back at the beginning, pressing RETURN will then execute the code that you've entered. The following shows what you'd see after entering a for statement (the underscore shows where the cursor winds up):

```
>>> for i in range(3):
...     _
```

If you then enter an if statement, an additional level of indentation will be provided:

```
>>> for i in range(30):
...     if i > 3:
...         _
```

Now enter `break` followed by RETURN and press BACKSPACE:

```
>>> for i in range(30):
...     if i > 3:
...         break
...     _
```

Finally type `print(i)`, press RETURN, press BACKSPACE and press RETURN again:

```
>>> for i in range(30):
...     if i > 3:
...         break
...     print(i)
...
0
1
2
3
>>>
```

Auto-indent won't be applied if the previous two lines were all spaces. This means that you can finish entering a compound statement by pressing RETURN twice, and then a third press will finish and execute.

Auto-completion

While typing a command at the REPL, if the line typed so far corresponds to the beginning of the name of something, then pressing TAB will show possible things that could be entered. For example type `m` and press TAB and it should expand to `machine`. Enter a dot `.` and press TAB again. You should see something like:

```
>>> machine.
__name__        info            unique_id       reset
bootloader      freq            rng             idle
sleep           deepsleep       disable_irq     enable_irq
Pin
```

The word will be expanded as much as possible until multiple possibilities exist. For example, type `machine.Pin.AF3` and press TAB and it will expand to `machine.Pin.AF3_TIM`. Pressing TAB a second time will show the possible expansions:

```
>>> machine.Pin.AF3_TIM
AF3_TIM10       AF3_TIM11       AF3_TIM8        AF3_TIM9
>>> machine.Pin.AF3_TIM
```

Interrupting a running program

You can interrupt a running program by pressing Ctrl-C. This will raise a KeyboardInterrupt which will bring you back to the REPL, providing your program doesn't intercept the KeyboardInterrupt exception.

For example:

```
>>> for i in range(1000000):
...     print(i)
...
0
1
2
3
...
6466
6467
6468
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt:
>>>
```

Paste Mode

If you want to paste some code into your terminal window, the auto-indent feature will mess things up. For example, if you had the following python code:

```
def foo():
    print('This is a test to show paste mode')
    print('Here is a second line')
foo()
```

and you try to paste this into the normal REPL, then you will see something like this:

```
>>> def foo():
...         print('This is a test to show paste mode')
...             print('Here is a second line')
...             foo()
...
  File "<stdin>", line 3
IndentationError: unexpected indent
```

If you press Ctrl-E, then you will enter paste mode, which essentially turns off the auto-indent feature, and changes the prompt from >>> to ===. For example:

```
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
=== def foo():
===     print('This is a test to show paste mode')
===     print('Here is a second line')
=== foo()
===
This is a test to show paste mode
Here is a second line
>>>
```

Paste Mode allows blank lines to be pasted. The pasted text is compiled as if it were a file. Pressing Ctrl-D exits paste mode and initiates the compilation.

Soft Reset

A soft reset will reset the python interpreter, but tries not to reset the method by which you're connected to the MicroPython board (USB-serial, or Wifi).

You can perform a soft reset from the REPL by pressing Ctrl-D, or from your python code by executing:

```
raise SystemExit
```

For example, if you reset your MicroPython board, and you execute a dir() command, you'd see something like this:

```
>>> dir()
['__name__', 'pyb']
```

Now create some variables and repeat the dir() command:

```
>>> i = 1
>>> j = 23
>>> x = 'abc'
>>> dir()
['j', 'x', '__name__', 'pyb', 'i']
>>>
```

Now if you enter Ctrl-D, and repeat the dir() command, you'll see that your variables no longer exist:

```
PYB: sync filesystems
PYB: soft reboot
MicroPython v1.5-51-g6f70283-dirty on 2015-10-30; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>> dir()
['__name__', 'pyb']
>>>
```

The special variable _ (underscore)

When you use the REPL, you may perform computations and see the results. MicroPython stores the results of the previous statement in the variable _ (underscore). So you can use the underscore to save the result in a variable. For example:

```
>>> 1 + 2 + 3 + 4 + 5
15
>>> x = _
>>> x
15
>>>
```

Raw Mode

Raw mode is not something that a person would normally use. It is intended for programmatic use. It essentially behaves like paste mode with echo turned off.

Raw mode is entered using Ctrl-A. You then send your python code, followed by a Ctrl-D. The Ctrl-D will be acknowledged by 'OK' and then the python code will be compiled and executed. Any output (or errors) will be sent back. Entering Ctrl-B will leave raw mode and return the the regular (aka friendly) REPL.

The `tools/pyboard.py` program uses the raw REPL to execute python files on the MicroPython board.

5.3.2 Writing interrupt handlers

On suitable hardware MicroPython offers the ability to write interrupt handlers in Python. Interrupt handlers - also known as interrupt service routines (ISR's) - are defined as callback functions. These are executed in response to an event such as a timer trigger or a voltage change on a pin. Such events can occur at any point in the execution of the program code. This carries significant consequences, some specific to the MicroPython language. Others are common to all systems capable of responding to real time events. This document covers the language specific issues first, followed by a brief introduction to real time programming for those new to it.

This introduction uses vague terms like "slow" or "as fast as possible". This is deliberate, as speeds are application dependent. Acceptable durations for an ISR are dependent on the rate at which interrupts occur, the nature of the main program, and the presence of other concurrent events.

Tips and recommended practices

This summarises the points detailed below and lists the principal recommendations for interrupt handler code.

- Keep the code as short and simple as possible.
- Avoid memory allocation: no appending to lists or insertion into dictionaries, no floating point.
- Where an ISR returns multiple bytes use a pre-allocated `bytearray`. If multiple integers are to be shared between an ISR and the main program consider an array (`array.array`).
- Where data is shared between the main program and an ISR, consider disabling interrupts prior to accessing the data in the main program and re-enabling them immediately afterwards (see Critical Sections).
- Allocate an emergency exception buffer (see below).

MicroPython Issues

The emergency exception buffer

If an error occurs in an ISR, MicroPython is unable to produce an error report unless a special buffer is created for the purpose. Debugging is simplified if the following code is included in any program using interrupts.

```
import micropython
micropython.alloc_emergency_exception_buf(100)
```

Simplicity

For a variety of reasons it is important to keep ISR code as short and simple as possible. It should do only what has to be done immediately after the event which caused it: operations which can be deferred should be delegated to the main program loop. Typically an ISR will deal with the hardware device which caused the interrupt, making it ready for the next interrupt to occur. It will communicate with the main loop by updating shared data to indicate that the interrupt has occurred, and it will return. An ISR should return control to the main loop as quickly as possible. This is not a specific MicroPython issue so is covered in more detail *below*.

Communication between an ISR and the main program

Normally an ISR needs to communicate with the main program. The simplest means of doing this is via one or more shared data objects, either declared as global or shared via a class (see below). There are various restrictions and hazards around doing this, which are covered in more detail below. Integers, `bytes` and `bytearray` objects are commonly used for this purpose along with arrays (from the array module) which can store various data types.

The use of object methods as callbacks

MicroPython supports this powerful technique which enables an ISR to share instance variables with the underlying code. It also enables a class implementing a device driver to support multiple device instances. The following example causes two LED's to flash at different rates.

```
import pyb, micropython
micropython.alloc_emergency_exception_buf(100)
class Foo(object):
    def __init__(self, timer, led):
        self.led = led
        timer.callback(self.cb)
    def cb(self, tim):
        self.led.toggle()

red = Foo(pyb.Timer(4, freq=1), pyb.LED(1))
greeen = Foo(pyb.Timer(2, freq=0.8), pyb.LED(2))
```

In this example the `red` instance associates timer 4 with LED 1: when a timer 4 interrupt occurs `red.cb()` is called causing LED 1 to change state. The `green` instance operates similarly: a timer 2 interrupt results in the execution of `green.cb()` and toggles LED 2. The use of instance methods confers two benefits. Firstly a single class enables code to be shared between multiple hardware instances. Secondly, as a bound method the callback function's first argument is `self`. This enables the callback to access instance data and to save state between successive calls. For example, if the class above had a variable `self.count` set to zero in the constructor, `cb()` could increment the counter. The `red` and `green` instances would then maintain independent counts of the number of times each LED had changed state.

Creation of Python objects

ISR's cannot create instances of Python objects. This is because MicroPython needs to allocate memory for the object from a store of free memory block called the heap. This is not permitted in an interrupt handler because heap allocation is not re-entrant. In other words the interrupt might occur when the main program is part way through performing an allocation - to maintain the integrity of the heap the interpreter disallows memory allocations in ISR code.

A consequence of this is that ISR's can't use floating point arithmetic; this is because floats are Python objects. Similarly an ISR can't append an item to a list. In practice it can be hard to determine exactly which code constructs will attempt to perform memory allocation and provoke an error message: another reason for keeping ISR code short and simple.

One way to avoid this issue is for the ISR to use pre-allocated buffers. For example a class constructor creates a `bytearray` instance and a boolean flag. The ISR method assigns data to locations in the buffer and sets the flag. The memory allocation occurs in the main program code when the object is instantiated rather than in the ISR.

The MicroPython library I/O methods usually provide an option to use a pre-allocated buffer. For example `pyb.i2c.recv()` can accept a mutable buffer as its first argument: this enables its use in an ISR.

A means of creating an object without employing a class or globals is as follows:

```
def set_volume(t, buf=bytearray(3)):
    buf[0] = 0xa5
    buf[1] = t >> 4
    buf[2] = 0x5a
    return buf
```

The compiler instantiates the default `buf` argument when the function is loaded for the first time (usually when the module it's in is imported).

Use of Python objects

A further restriction on objects arises because of the way Python works. When an `import` statement is executed the Python code is compiled to bytecode, with one line of code typically mapping to multiple bytecodes. When the code runs the interpreter reads each bytecode and executes it as a series of machine code instructions. Given that an interrupt can occur at any time between machine code instructions, the original line of Python code may be only partially executed. Consequently a Python object such as a set, list or dictionary modified in the main loop may lack internal consistency at the moment the interrupt occurs.

A typical outcome is as follows. On rare occasions the ISR will run at the precise moment in time when the object is partially updated. When the ISR tries to read the object, a crash results. Because such problems typically occur on rare, random occasions they can be hard to diagnose. There are ways to circumvent this issue, described in *Critical Sections* below.

It is important to be clear about what constitutes the modification of an object. An alteration to a built-in type such as a dictionary is problematic. Altering the contents of an array or bytearray is not. This is because bytes or words are written as a single machine code instruction which is not interruptible: in the parlance of real time programming the write is atomic. A user defined object might instantiate an integer, array or bytearray. It is valid for both the main loop and the ISR to alter the contents of these.

MicroPython supports integers of arbitrary precision. Values between 2**30 -1 and -2**30 will be stored in a single machine word. Larger values are stored as Python objects. Consequently changes to long integers cannot be considered atomic. The use of long integers in ISR's is unsafe because memory allocation may be attempted as the variable's value changes.

Overcoming the float limitation

In general it is best to avoid using floats in ISR code: hardware devices normally handle integers and conversion to floats is normally done in the main loop. However there are a few DSP algorithms which require floating point. On platforms with hardware floating point (such as the Pyboard) the inline ARM Thumb assembler can be used to work round this limitation. This is because the processor stores float values in a machine word; values can be shared between the ISR and main program code via an array of floats.

Exceptions

If an ISR raises an exception it will not propagate to the main loop. The interrupt will be disabled unless the exception is handled by the ISR code.

General Issues

This is merely a brief introduction to the subject of real time programming. Beginners should note that design errors in real time programs can lead to faults which are particularly hard to diagnose. This is because they can occur rarely and at intervals which are essentially random. It is crucial to get the initial design right and to anticipate issues before they arise. Both interrupt handlers and the main program need to be designed with an appreciation of the following issues.

Interrupt Handler Design

As mentioned above, ISR's should be designed to be as simple as possible. They should always return in a short, predictable period of time. This is important because when the ISR is running, the main loop is not: inevitably the main loop experiences pauses in its execution at random points in the code. Such pauses can be a source of hard to diagnose bugs particularly if their duration is long or variable. In order to understand the implications of ISR run time, a basic grasp of interrupt priorities is required.

Interrupts are organised according to a priority scheme. ISR code may itself be interrupted by a higher priority interrupt. This has implications if the two interrupts share data (see Critical Sections below). If such an interrupt occurs it interposes a delay into the ISR code. If a lower priority interrupt occurs while the ISR is running, it will be delayed until the ISR is complete: if the delay is too long, the lower priority interrupt may fail. A further issue with slow ISR's is the case where a second interrupt of the same type occurs during its execution. The second interrupt will be handled on termination of the first. However if the rate of incoming interrupts consistently exceeds the capacity of the ISR to service them the outcome will not be a happy one.

Consequently looping constructs should be avoided or minimised. I/O to devices other than to the interrupting device should normally be avoided: I/O such as disk access, `print` statements and UART access is relatively slow, and its duration may vary. A further issue here is that filesystem functions are not reentrant: using filesystem I/O in an ISR and the main program would be hazardous. Crucially ISR code should not wait on an event. I/O is acceptable if the code can be guaranteed to return in a predictable period, for example toggling a pin or LED. Accessing the interrupting device via I2C or SPI may be necessary but the time taken for such accesses should be calculated or measured and its impact on the application assessed.

There is usually a need to share data between the ISR and the main loop. This may be done either through global variables or via class or instance variables. Variables are typically integer or boolean types, or integer or byte arrays (a pre-allocated integer array offers faster access than a list). Where multiple values are modified by the ISR it is necessary to consider the case where the interrupt occurs at a time when the main program has accessed some, but not all, of the values. This can lead to inconsistencies.

Consider the following design. An ISR stores incoming data in a bytearray, then adds the number of bytes received to an integer representing total bytes ready for processing. The main program reads the number of bytes, processes the

bytes, then clears down the number of bytes ready. This will work until an interrupt occurs just after the main program has read the number of bytes. The ISR puts the added data into the buffer and updates the number received, but the main program has already read the number, so processes the data originally received. The newly arrived bytes are lost.

There are various ways of avoiding this hazard, the simplest being to use a circular buffer. If it is not possible to use a structure with inherent thread safety other ways are described below.

Reentrancy

A potential hazard may occur if a function or method is shared between the main program and one or more ISR's or between multiple ISR's. The issue here is that the function may itself be interrupted and a further instance of that function run. If this is to occur, the function must be designed to be reentrant. How this is done is an advanced topic beyond the scope of this tutorial.

Critical Sections

An example of a critical section of code is one which accesses more than one variable which can be affected by an ISR. If the interrupt happens to occur between accesses to the individual variables, their values will be inconsistent. This is an instance of a hazard known as a race condition: the ISR and the main program loop race to alter the variables. To avoid inconsistency a means must be employed to ensure that the ISR does not alter the values for the duration of the critical section. One way to achieve this is to issue `pyb.disable_irq()` before the start of the section, and `pyb.enable_irq()` at the end. Here is an example of this approach:

```
import pyb, micropython, array
micropython.alloc_emergency_exception_buf(100)

class BoundsException(Exception):
    pass

ARRAYSIZE = const(20)
index = 0
data = array.array('i', 0 for x in range(ARRAYSIZE))

def callback1(t):
    global data, index
    for x in range(5):
        data[index] = pyb.rng() # simulate input
        index += 1
        if index >= ARRAYSIZE:
            raise BoundsException('Array bounds exceeded')

tim4 = pyb.Timer(4, freq=100, callback=callback1)

for loop in range(1000):
    if index > 0:
        irq_state = pyb.disable_irq() # Start of critical section
        for x in range(index):
            print(data[x])
        index = 0
        pyb.enable_irq(irq_state) # End of critical section
        print('loop {}'.format(loop))
    pyb.delay(1)

tim4.callback(None)
```

A critical section can comprise a single line of code and a single variable. Consider the following code fragment.

```
count = 0
def cb(): # An interrupt callback
    count +=1
def main():
    # Code to set up the interrupt callback omitted
    while True:
        count += 1
```

This example illustrates a subtle source of bugs. The line `count += 1` in the main loop carries a specific race condition hazard known as a read-modify-write. This is a classic cause of bugs in real time systems. In the main loop MicroPython reads the value of `t.counter`, adds 1 to it, and writes it back. On rare occasions the interrupt occurs after the read and before the write. The interrupt modifies `t.counter` but its change is overwritten by the main loop when the ISR returns. In a real system this could lead to rare, unpredictable failures.

As mentioned above, care should be taken if an instance of a Python built in type is modified in the main code and that instance is accessed in an ISR. The code performing the modification should be regarded as a critical section to ensure that the instance is in a valid state when the ISR runs.

Particular care needs to be taken if a dataset is shared between different ISR's. The hazard here is that the higher priority interrupt may occur when the lower priority one has partially updated the shared data. Dealing with this situation is an advanced topic beyond the scope of this introduction other than to note that mutex objects described below can sometimes be used.

Disabling interrupts for the duration of a critical section is the usual and simplest way to proceed, but it disables all interrupts rather than merely the one with the potential to cause problems. It is generally undesirable to disable an interrupt for long. In the case of timer interrupts it introduces variability to the time when a callback occurs. In the case of device interrupts, it can lead to the device being serviced too late with possible loss of data or overrun errors in the device hardware. Like ISR's, a critical section in the main code should have a short, predictable duration.

An approach to dealing with critical sections which radically reduces the time for which interrupts are disabled is to use an object termed a mutex (name derived from the notion of mutual exclusion). The main program locks the mutex before running the critical section and unlocks it at the end. The ISR tests whether the mutex is locked. If it is, it avoids the critical section and returns. The design challenge is defining what the ISR should do in the event that access to the critical variables is denied. A simple example of a mutex may be found here. Note that the mutex code does disable interrupts, but only for the duration of eight machine instructions: the benefit of this approach is that other interrupts are virtually unaffected.

Interrupts and the REPL

Interrupt handlers, such as those associated with timers, can continue to run after a program terminates. This may produce unexpected results where you might have expected the object raising the callback to have gone out of scope. For example on the Pyboard:

```
def bar():
    foo = pyb.Timer(2, freq=4, callback=lambda t: print('.', end=''))

bar()
```

This continues to run until the timer is explicitly disabled or the board is reset with `ctrl D`.

5.3.3 Maximising Python Speed

This tutorial describes ways of improving the performance of MicroPython code. Optimisations involving other languages are covered elsewhere, namely the use of modules written in C and the MicroPython inline ARM Thumb-2 assembler.

The process of developing high performance code comprises the following stages which should be performed in the order listed.

- Design for speed.
- Code and debug.

Optimisation steps:

- Identify the slowest section of code.
- Improve the efficiency of the Python code.
- Use the native code emitter.
- Use the viper code emitter.

Designing for speed

Performance issues should be considered at the outset. This involves taking a view on the sections of code which are most performance critical and devoting particular attention to their design. The process of optimisation begins when the code has been tested: if the design is correct at the outset optimisation will be straightforward and may actually be unnecessary.

Algorithms

The most important aspect of designing any routine for performance is ensuring that the best algorithm is employed. This is a topic for textbooks rather than for a MicroPython guide but spectacular performance gains can sometimes be achieved by adopting algorithms known for their efficiency.

RAM Allocation

To design efficient MicroPython code it is necessary to have an understanding of the way the interpreter allocates RAM. When an object is created or grows in size (for example where an item is appended to a list) the necessary RAM is allocated from a block known as the heap. This takes a significant amount of time; further it will on occasion trigger a process known as garbage collection which can take several milliseconds.

Consequently the performance of a function or method can be improved if an object is created once only and not permitted to grow in size. This implies that the object persists for the duration of its use: typically it will be instantiated in a class constructor and used in various methods.

This is covered in further detail *Controlling garbage collection* below.

Buffers

An example of the above is the common case where a buffer is required, such as one used for communication with a device. A typical driver will create the buffer in the constructor and use it in its I/O methods which will be called repeatedly.

The MicroPython libraries typically provide support for pre-allocated buffers. For example, objects which support stream interface (e.g., file or UART) provide `read()` method which allocate new buffer for read data, but also a `readinto()` method to read data into an existing buffer.

Floating Point

Some MicroPython ports allocate floating point numbers on heap. Some other ports may lack dedicated floating-point coprocessor, and perform arithmetic operations on them in "software" at considerably lower speed than on integers. Where performance is important, use integer operations and restrict the use of floating point to sections of the code where performance is not paramount. For example, capture ADC readings as integers values to an array in one quick go, and only then convert them to floating-point numbers for signal processing.

Arrays

Consider the use of the various types of array classes as an alternative to lists. The `array` module supports various element types with 8-bit elements supported by Python's built in `bytes` and `bytearray` classes. These data structures all store elements in contiguous memory locations. Once again to avoid memory allocation in critical code these should be pre-allocated and passed as arguments or as bound objects.

When passing slices of objects such as `bytearray` instances, Python creates a copy which involves allocation of the size proportional to the size of slice. This can be alleviated using a `memoryview` object. `memoryview` itself is allocated on heap, but is a small, fixed-size object, regardless of the size of slice it points too.

```
ba = bytearray(10000)  # big array
func(ba[30:2000])      # a copy is passed, ~2K new allocation
mv = memoryview(ba)    # small object is allocated
func(mv[30:2000])      # a pointer to memory is passed
```

A `memoryview` can only be applied to objects supporting the buffer protocol - this includes arrays but not lists. Small caveat is that while memoryview object is live, it also keeps alive the original buffer object. So, a memoryview isn't a universal panacea. For instance, in the example above, if you are done with 10K buffer and just need those bytes 30:2000 from it, it may be better to make a slice, and let the 10K buffer go (be ready for garbage collection), instead of making a long-living memoryview and keeping 10K blocked for GC.

Nonetheless, `memoryview` is indispensable for advanced preallocated buffer management. `.readinto()` method discussed above puts data at the beginning of buffer and fills in entire buffer. What if you need to put data in the middle of existing buffer? Just create a memoryview into the needed section of buffer and pass it to `.readinto()`.

Identifying the slowest section of code

This is a process known as profiling and is covered in textbooks and (for standard Python) supported by various software tools. For the type of smaller embedded application likely to be running on MicroPython platforms the slowest function or method can usually be established by judicious use of the timing `ticks` group of functions documented here. Code execution time can be measured in ms, us, or CPU cycles.

The following enables any function or method to be timed by adding an `@timed_function` decorator:

```
def timed_function(f, *args, **kwargs):
    myname = str(f).split(' ')[1]
    def new_func(*args, **kwargs):
        t = time.ticks_us()
        result = f(*args, **kwargs)
        delta = time.ticks_diff(time.ticks_us(), t)
        print('Function {} Time = {:6.3f}ms'.format(myname, delta/1000))
        return result
    return new_func
```

MicroPython code improvements

The const() declaration

MicroPython provides a `const()` declaration. This works in a similar way to `#define` in C in that when the code is compiled to bytecode the compiler substitutes the numeric value for the identifier. This avoids a dictionary lookup at runtime. The argument to `const()` may be anything which, at compile time, evaluates to an integer e.g. `0x100` or `1 << 8`.

Caching object references

Where a function or method repeatedly accesses objects performance is improved by caching the object in a local variable:

```
class foo(object):
    def __init__(self):
        ba = bytearray(100)
    def bar(self, obj_display):
        ba_ref = self.ba
        fb = obj_display.framebuffer
        # iterative code using these two objects
```

This avoids the need repeatedly to look up `self.ba` and `obj_display.framebuffer` in the body of the method `bar()`.

Controlling garbage collection

When memory allocation is required, MicroPython attempts to locate an adequately sized block on the heap. This may fail, usually because the heap is cluttered with objects which are no longer referenced by code. If a failure occurs, the process known as garbage collection reclaims the memory used by these redundant objects and the allocation is then tried again - a process which can take several milliseconds.

There are benefits in pre-empting this by periodically issuing `gc.collect()`. Firstly doing a collection before it is actually required is quicker - typically on the order of 1ms if done frequently. Secondly you can determine the point in code where this time is used rather than have a longer delay occur at random points, possibly in a speed critical section. Finally performing collections regularly can reduce fragmentation in the heap. Severe fragmentation can lead to non-recoverable allocation failures.

Accessing hardware directly

This comes into the category of more advanced programming and involves some knowledge of the target MCU. Consider the example of toggling an output pin on the Pyboard. The standard approach would be to write

```
mypin.value(mypin.value() ^ 1) # mypin was instantiated as an output pin
```

This involves the overhead of two calls to the `Pin` instance's `value()` method. This overhead can be eliminated by performing a read/write to the relevant bit of the chip's GPIO port output data register (odr). To facilitate this the `stm` module provides a set of constants providing the addresses of the relevant registers. A fast toggle of pin `P4` (CPU pin `A14`) - corresponding to the green LED - can be performed as follows:

```
BIT14 = const(1 << 14)
stm.mem16[stm.GPIOA + stm.GPIO_ODR] ^= BIT14
```

The Native code emitter

This causes the MicroPython compiler to emit ARM native opcodes rather than bytecode. It covers the bulk of the Python language so most functions will require no adaptation (but see below). It is invoked by means of a function decorator:

```
@micropython.native
def foo(self, arg):
    buf = self.linebuf # Cached object
    # code
```

There are certain limitations in the current implementation of the native code emitter.

- Context managers are not supported (the `with` statement).
- Generators are not supported.
- If `raise` is used an argument must be supplied.

The trade-off for the improved performance (roughly twices as fast as bytecode) is an increase in compiled code size.

The Viper code emitter

The optimisations discussed above involve standards-compliant Python code. The Viper code emitter is not fully compliant. It supports special Viper native data types in pursuit of performance. Integer processing is non-compliant because it uses machine words: arithmetic on 32 bit hardware is performed modulo 2**32.

Like the Native emitter Viper produces machine instructions but further optimisations are performed, substantially increasing performance especially for integer arithmetic and bit manipulations. It is invoked using a decorator:

```
@micropython.viper
def foo(self, arg: int) -> int:
    # code
```

As the above fragment illustrates it is beneficial to use Python type hints to assist the Viper optimiser. Type hints provide information on the data types of arguments and of the return value; these are a standard Python language feature formally defined here PEP0484. Viper supports its own set of types namely `int`, `uint` (unsigned integer), `ptr`, `ptr8`, `ptr16` and `ptr32`. The `ptrX` types are discussed below. Currently the `uint` type serves a single purpose: as a type hint for a function return value. If such a function returns `0xffffffff` Python will interpret the result as 2**32 -1 rather than as -1.

In addition to the restrictions imposed by the native emitter the following constraints apply:

- Functions may have up to four arguments.
- Default argument values are not permitted.
- Floating point may be used but is not optimised.

Viper provides pointer types to assist the optimiser. These comprise

- `ptr` Pointer to an object.
- `ptr8` Points to a byte.
- `ptr16` Points to a 16 bit half-word.
- `ptr32` Points to a 32 bit machine word.

The concept of a pointer may be unfamiliar to Python programmers. It has similarities to a Python `memoryview` object in that it provides direct access to data stored in memory. Items are accessed using subscript notation, but slices are not supported: a pointer can return a single item only. Its purpose is to provide fast random access to data stored in

contiguous memory locations - such as data stored in objects which support the buffer protocol, and memory-mapped peripheral registers in a microcontroller. It should be noted that programming using pointers is hazardous: bounds checking is not performed and the compiler does nothing to prevent buffer overrun errors.

Typical usage is to cache variables:

```
@micropython.viper
def foo(self, arg: int) -> int:
    buf = ptr8(self.linebuf) # self.linebuf is a bytearray or bytes object
    for x in range(20, 30):
        bar = buf[x] # Access a data item through the pointer
        # code omitted
```

In this instance the compiler "knows" that `buf` is the address of an array of bytes; it can emit code to rapidly compute the address of `buf[x]` at runtime. Where casts are used to convert objects to Viper native types these should be performed at the start of the function rather than in critical timing loops as the cast operation can take several microseconds. The rules for casting are as follows:

- Casting operators are currently: `int`, `bool`, `uint`, `ptr`, `ptr8`, `ptr16` and `ptr32`.
- The result of a cast will be a native Viper variable.
- Arguments to a cast can be a Python object or a native Viper variable.
- If argument is a native Viper variable, then cast is a no-op (i.e. costs nothing at runtime) that just changes the type (e.g. from `uint` to `ptr8`) so that you can then store/load using this pointer.
- If the argument is a Python object and the cast is `int` or `uint`, then the Python object must be of integral type and the value of that integral object is returned.
- The argument to a bool cast must be integral type (boolean or integer); when used as a return type the viper function will return True or False objects.
- If the argument is a Python object and the cast is `ptr`, `ptr`, `ptr16` or `ptr32`, then the Python object must either have the buffer protocol with read-write capabilities (in which case a pointer to the start of the buffer is returned) or it must be of integral type (in which case the value of that integral object is returned).

The following example illustrates the use of a `ptr16` cast to toggle pin X1 `n` times:

```
BIT0 = const(1)
@micropython.viper
def toggle_n(n: int):
    odr = ptr16(stm.GPIOA + stm.GPIO_ODR)
    for _ in range(n):
        odr[0] ^= BIT0
```

A detailed technical description of the three code emitters may be found on Kickstarter here Note 1 and here Note 2

5.3.4 MicroPython on Microcontrollers

MicroPython is designed to be capable of running on microcontrollers. These have hardware limitations which may be unfamiliar to programmers more familiar with conventional computers. In particular the amount of RAM and non-volatile "disk" (flash memory) storage is limited. This tutorial offers ways to make the most of the limited resources. Because MicroPython runs on controllers based on a variety of architectures, the methods presented are generic: in some cases it will be necessary to obtain detailed information from platform specific documentation.

Flash Memory

On the Pyboard the simple way to address the limited capacity is to fit a micro SD card. In some cases this is impractical, either because the device does not have an SD card slot or for reasons of cost or power consumption; hence the on-chip flash must be used. The firmware including the MicroPython subsystem is stored in the onboard flash. The remaining capacity is available for use. For reasons connected with the physical architecture of the flash memory part of this capacity may be inaccessible as a filesystem. In such cases this space may be employed by incorporating user modules into a firmware build which is then flashed to the device.

There are two ways to achieve this: frozen modules and frozen bytecode. Frozen modules store the Python source with the firmware. Frozen bytecode uses the cross compiler to convert the source to bytecode which is then stored with the firmware. In either case the module may be accessed with an import statement:

```
import mymodule
```

The procedure for producing frozen modules and bytecode is platform dependent; instructions for building the firmware can be found in the README files in the relevant part of the source tree.

In general terms the steps are as follows:

- Clone the MicroPython repository.
- Acquire the (platform specific) toolchain to build the firmware.
- Build the cross compiler.
- Place the modules to be frozen in a specified directory (dependent on whether the module is to be frozen as source or as bytecode).
- Build the firmware. A specific command may be required to build frozen code of either type - see the platform documentation.
- Flash the firmware to the device.

RAM

When reducing RAM usage there are two phases to consider: compilation and execution. In addition to memory consumption, there is also an issue known as heap fragmentation. In general terms it is best to minimise the repeated creation and destruction of objects. The reason for this is covered in the section covering the *heap*.

Compilation Phase

When a module is imported, MicroPython compiles the code to bytecode which is then executed by the MicroPython virtual machine (VM). The bytecode is stored in RAM. The compiler itself requires RAM, but this becomes available for use when the compilation has completed.

If a number of modules have already been imported the situation can arise where there is insufficient RAM to run the compiler. In this case the import statement will produce a memory exception.

If a module instantiates global objects on import it will consume RAM at the time of import, which is then unavailable for the compiler to use on subsequent imports. In general it is best to avoid code which runs on import; a better approach is to have initialisation code which is run by the application after all modules have been imported. This maximises the RAM available to the compiler.

If RAM is still insufficient to compile all modules one solution is to precompile modules. MicroPython has a cross compiler capable of compiling Python modules to bytecode (see the README in the mpy-cross directory). The resulting bytecode file has a .mpy extension; it may be copied to the filesystem and imported in the usual way. Alternatively some or all modules may be implemented as frozen bytecode: on most platforms this saves even more RAM as the bytecode is run directly from flash rather than being stored in RAM.

Execution Phase

There are a number of coding techniques for reducing RAM usage.

Constants

MicroPython provides a `const` keyword which may be used as follows:

```
from micropython import const
ROWS = const(33)
_COLS = const(0x10)
a = ROWS
b = _COLS
```

In both instances where the constant is assigned to a variable the compiler will avoid coding a lookup to the name of the constant by substituting its literal value. This saves bytecode and hence RAM. However the `ROWS` value will occupy at least two machine words, one each for the key and value in the globals dictionary. The presence in the dictionary is necessary because another module might import or use it. This RAM can be saved by prepending the name with an underscore as in `_COLS`: this symbol is not visible outside the module so will not occupy RAM.

The argument to `const()` may be anything which, at compile time, evaluates to an integer e.g. `0x100` or `1 << 8`. It can even include other const symbols that have already been defined, e.g. `1 << BIT`.

Constant data structures

Where there is a substantial volume of constant data and the platform supports execution from Flash, RAM may be saved as follows. The data should be located in Python modules and frozen as bytecode. The data must be defined as `bytes` objects. The compiler 'knows' that `bytes` objects are immutable and ensures that the objects remain in flash memory rather than being copied to RAM. The `ustruct` module can assist in converting between `bytes` types and other Python built-in types.

When considering the implications of frozen bytecode, note that in Python strings, floats, bytes, integers and complex numbers are immutable. Accordingly these will be frozen into flash. Thus, in the line

```
mystring = "The quick brown fox"
```

the actual string "The quick brown fox" will reside in flash. At runtime a reference to the string is assigned to the *variable* `mystring`. The reference occupies a single machine word. In principle a long integer could be used to store constant data:

```
bar = 0xDEADBEEF0000DEADBEEF
```

As in the string example, at runtime a reference to the arbitrarily large integer is assigned to the variable `bar`. That reference occupies a single machine word.

It might be expected that tuples of integers could be employed for the purpose of storing constant data with minimal RAM use. With the current compiler this is ineffective (the code works, but RAM is not saved).

```
foo = (1, 2, 3, 4, 5, 6, 100000)
```

At runtime the tuple will be located in RAM. This may be subject to future improvement.

Needless object creation

There are a number of situations where objects may unwittingly be created and destroyed. This can reduce the usability of RAM through fragmentation. The following sections discuss instances of this.

String concatenation

Consider the following code fragments which aim to produce constant strings:

```
var = "foo" + "bar"
var1 = "foo" "bar"
var2 = """\
foo\
bar"""
```

Each produces the same outcome, however the first needlessly creates two string objects at runtime, allocates more RAM for concatenation before producing the third. The others perform the concatenation at compile time which is more efficient, reducing fragmentation.

Where strings must be dynamically created before being fed to a stream such as a file it will save RAM if this is done in a piecemeal fashion. Rather than creating a large string object, create a substring and feed it to the stream before dealing with the next.

The best way to create dynamic strings is by means of the string `format` method:

```
var = "Temperature {:5.2f} Pressure {:06d}\n".format(temp, press)
```

Buffers

When accessing devices such as instances of UART, I2C and SPI interfaces, using pre-allocated buffers avoids the creation of needless objects. Consider these two loops:

```
while True:
    var = spi.read(100)
    # process data

buf = bytearray(100)
while True:
    spi.readinto(buf)
    # process data in buf
```

The first creates a buffer on each pass whereas the second re-uses a pre-allocated buffer; this is both faster and more efficient in terms of memory fragmentation.

Bytes are smaller than ints

On most platforms an integer consumes four bytes. Consider the two calls to the function `foo()`:

```
def foo(bar):
    for x in bar:
        print(x)
foo((1, 2, 0xff))
foo(b'\1\2\xff')
```

In the first call a tuple of integers is created in RAM. The second efficiently creates a `bytes` object consuming the minimum amount of RAM. If the module were frozen as bytecode, the `bytes` object would reside in flash.

Strings Versus Bytes

Python3 introduced Unicode support. This introduced a distinction between a string and an array of bytes. MicroPython ensures that Unicode strings take no additional space so long as all characters in the string are ASCII (i.e. have a value < 126). If values in the full 8-bit range are required `bytes` and `bytearray` objects can be used to ensure that no additional space will be required. Note that most string methods (e.g. `strip()`) apply also to `bytes` instances so the process of eliminating Unicode can be painless.

```
s = 'the quick brown fox'  # A string instance
b = b'the quick brown fox'  # a bytes instance
```

Where it is necessary to convert between strings and bytes the string `encode` and the bytes `decode` methods can be used. Note that both strings and bytes are immutable. Any operation which takes as input such an object and produces

another implies at least one RAM allocation to produce the result. In the second line below a new bytes object is allocated. This would also occur if `foo` were a string.

```
foo = b'   empty whitespace'
foo = foo.lstrip()
```

Runtime compiler execution

The Python keywords `eval` and `exec` invoke the compiler at runtime, which requires significant amounts of RAM. Note that the `pickle` library employs `exec`. It may be more RAM efficient to use the `json` library for object serialisation.

Storing strings in flash

Python strings are immutable hence have the potential to be stored in read only memory. The compiler can place in flash strings defined in Python code. As with frozen modules it is necessary to have a copy of the source tree on the PC and the toolchain to build the firmware. The procedure will work even if the modules have not been fully debugged, so long as they can be imported and run.

After importing the modules, execute:

```
micropython.qstr_info(1)
```

Then copy and paste all the Q(xxx) lines into a text editor. Check for and remove lines which are obviously invalid. Open the file qstrdefsport.h which will be found in stmhal (or the equivalent directory for the architecture in use). Copy and paste the corrected lines at the end of the file. Save the file, rebuild and flash the firmware. The outcome can be checked by importing the modules and again issuing:

```
micropython.qstr_info(1)
```

The Q(xxx) lines should be gone.

The Heap

When a running program instantiates an object the necessary RAM is allocated from a fixed size pool known as the heap. When the object goes out of scope (in other words becomes inaccessible to code) the redundant object is known as "garbage". A process known as "garbage collection" (GC) reclaims that memory, returning it to the free heap. This process runs automatically, however it can be invoked directly by issuing `gc.collect()`.

The discourse on this is somewhat involved. For a 'quick fix' issue the following periodically:

```
gc.collect()
gc.threshold(gc.mem_free() // 4 + gc.mem_alloc())
```

Fragmentation

Say a program creates an object `foo`, then an object `bar`. Subsequently `foo` goes out of scope but `bar` remains. The RAM used by `foo` will be reclaimed by GC. However if `bar` was allocated to a higher address, the RAM reclaimed from `foo` will only be of use for objects no bigger than `foo`. In a complex or long running program the heap can become fragmented: despite there being a substantial amount of RAM available, there is insufficient contiguous space to allocate a particular object, and the program fails with a memory error.

The techniques outlined above aim to minimise this. Where large permanent buffers or other objects are required it is best to instantiate these early in the process of program execution before fragmentation can occur. Further improvements may be made by monitoring the state of the heap and by controlling GC; these are outlined below.

Reporting

A number of library functions are available to report on memory allocation and to control GC. These are to be found in the `gc` and `micropython` modules. The following example may be pasted at the REPL (`ctrl e` to enter paste mode, `ctrl d` to run it).

```
import gc
import micropython
gc.collect()
micropython.mem_info()
print('-----------------------------')
print('Initial free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
def func():
    a = bytearray(10000)
gc.collect()
print('Func definition: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
func()
print('Func run free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
gc.collect()
print('Garbage collect free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
print('-----------------------------')
micropython.mem_info(1)
```

Methods employed above:

- `gc.collect()` Force a garbage collection. See footnote.
- `micropython.mem_info()` Print a summary of RAM utilisation.
- `gc.mem_free()` Return the free heap size in bytes.
- `gc.mem_alloc()` Return the number of bytes currently allocated.
- `micropython.mem_info(1)` Print a table of heap utilisation (detailed below).

The numbers produced are dependent on the platform, but it can be seen that declaring the function uses a small amount of RAM in the form of bytecode emitted by the compiler (the RAM used by the compiler has been reclaimed). Running the function uses over 10KiB, but on return `a` is garbage because it is out of scope and cannot be referenced. The final `gc.collect()` recovers that memory.

The final output produced by `micropython.mem_info(1)` will vary in detail but may be interpreted as follows:

Symbol	Meaning
.	free block
h	head block
=	tail block
m	marked head block
T	tuple
L	list
D	dict
F	float
B	byte code
M	module

Each letter represents a single block of memory, a block being 16 bytes. So each line of the heap dump represents 0x400 bytes or 1KiB of RAM.

Control of Garbage Collection

A GC can be demanded at any time by issuing `gc.collect()`. It is advantageous to do this at intervals, firstly to pre-empt fragmentation and secondly for performance. A GC can take several milliseconds but is quicker when there is little work to do (about 1ms on the Pyboard). An explicit call can minimise that delay while ensuring it occurs at points in the program when it is acceptable.

Automatic GC is provoked under the following circumstances. When an attempt at allocation fails, a GC is performed and the allocation re-tried. Only if this fails is an exception raised. Secondly an automatic GC will be triggered if the amount of free RAM falls below a threshold. This threshold can be adapted as execution progresses:

```
gc.collect()
gc.threshold(gc.mem_free() // 4 + gc.mem_alloc())
```

This will provoke a GC when more than 25% of the currently free heap becomes occupied.

In general modules should instantiate data objects at runtime using constructors or other initialisation functions. The reason is that if this occurs on initialisation the compiler may be starved of RAM when subsequent modules are imported. If modules do instantiate data on import then `gc.collect()` issued after the import will ameliorate the problem.

String Operations

MicroPython handles strings in an efficient manner and understanding this can help in designing applications to run on microcontrollers. When a module is compiled, strings which occur multiple times are stored once only, a process known as string interning. In MicroPython an interned string is known as a `qstr`. In a module imported normally that single instance will be located in RAM, but as described above, in modules frozen as bytecode it will be located in flash.

String comparisons are also performed efficiently using hashing rather than character by character. The penalty for using strings rather than integers may hence be small both in terms of performance and RAM usage - a fact which may come as a surprise to C programmers.

Postscript

MicroPython passes, returns and (by default) copies objects by reference. A reference occupies a single machine word so these processes are efficient in RAM usage and speed.

Where variables are required whose size is neither a byte nor a machine word there are standard libraries which can assist in storing these efficiently and in performing conversions. See the `array`, `ustruct` and `uctypes` modules.

Footnote: gc.collect() return value

On Unix and Windows platforms the `gc.collect()` method returns an integer which signifies the number of distinct memory regions that were reclaimed in the collection (more precisely, the number of heads that were turned into frees). For efficiency reasons bare metal ports do not return this value.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

array, 11

e

esp, 48

f

framebuf, 30

g

gc, 12

m

machine, 31
math, 12
micropython, 43

n

network, 43

s

sys, 14

u

ubinascii, 16
ucollections, 16
uctypes, 46
uhashlib, 17
uheapq, 18
uio, 18
ujson, 20
uos, 20
ure, 21
usocket, 22
ussl, 25
ustruct, 26
utime, 26
uzlib, 29

Symbols

__call__() (machine.Pin method), 37

A

a2b_base64() (in module ubinascii), 16
abs() (built-in function), 10
accept() (usocket.socket method), 24
acos() (in module math), 12
acosh() (in module math), 12
active() (network.wlan method), 44
addressof() (in module uctypes), 47
alarm() (machine.RTC method), 39
alarm_left() (machine.RTC method), 39
all() (built-in function), 10
alloc_emergency_exception_buf() (in module micropy-thon), 43
any() (built-in function), 10
append() (array.array.array method), 12
argv (in module sys), 15
array (module), 11
array.array (class in array), 12
asin() (in module math), 12
asinh() (in module math), 12
atan() (in module math), 12
atan2() (in module math), 12
atanh() (in module math), 13

B

b2a_base64() (in module ubinascii), 16
BIG_ENDIAN (in module uctypes), 47
bin() (built-in function), 10
bind() (usocket.socket method), 24
blit() (framebuf.FrameBuffer method), 31
bool (built-in class), 10
bytearray (built-in class), 10
bytearray_at() (in module uctypes), 47
byteorder (in module sys), 15
bytes (built-in class), 10
bytes_at() (in module uctypes), 47
BytesIO (class in uio), 19

C

calcsize() (in module ustruct), 26
callable() (built-in function), 10
cancel() (machine.RTC method), 39
ceil() (in module math), 13
chdir() (in module uos), 20
chr() (built-in function), 10
classmethod() (built-in function), 10
close() (usocket.socket method), 23
collect() (in module gc), 12
compile() (built-in function), 10
compile() (in module ure), 21
complex (built-in class), 10
config() (network.wlan method), 45
connect() (network.wlan method), 44
connect() (usocket.socket method), 24
copysign() (in module math), 13
cos() (in module math), 13
cosh() (in module math), 13

D

DEBUG (in module ure), 22
decompress() (in module uzlib), 29
deepsleep() (in module esp), 49
deepsleep() (in module machine), 32
degrees() (in module math), 13
deinit() (machine.I2C method), 34
deinit() (machine.RTC method), 39
deinit() (machine.SPI method), 40
deinit() (machine.Timer method), 41
delattr() (built-in function), 10
dict (built-in class), 10
digest() (uhashlib.hash method), 18
dir() (built-in function), 10
disable() (in module gc), 12
disable_irq() (in module machine), 32
disconnect() (network.wlan method), 44
divmod() (built-in function), 10
drive() (machine.Pin method), 38
dumps() (in module ujson), 20

E

e (in module math), 14
enable() (in module gc), 12
enable_irq() (in module machine), 32
enumerate() (built-in function), 10
erf() (in module math), 13
erfc() (in module math), 13
esp (module), 48
eval() (built-in function), 10
exec() (built-in function), 10
exit() (in module sys), 14
exp() (in module math), 13
expm1() (in module math), 13
extend() (array.array.array method), 12

F

fabs() (in module math), 13
feed() (machine.wdt method), 43
FileIO (class in uio), 19
fill() (framebuf.FrameBuffer method), 30
fill_rect() (framebuf.FrameBuffer method), 31
filter() (built-in function), 10
flash_erase() (in module esp), 49
flash_id() (in module esp), 49
flash_read() (in module esp), 49
flash_write() (in module esp), 49
float (built-in class), 10
floor() (in module math), 13
fmod() (in module math), 13
framebuf (module), 30
framebuf.GS4_HMSB (in module framebuf), 31
framebuf.MVLSB (in module framebuf), 31
framebuf.RGB565 (in module framebuf), 31
FrameBuffer (class in framebuf), 30
freq() (in module machine), 32
frexp() (in module math), 13
frozenset (built-in class), 10

G

gamma() (in module math), 13
gc (module), 12
getattr() (built-in function), 10
getcwd() (in module uos), 20
getvalue() (uio.BytesIO method), 19
globals() (built-in function), 10
group() (ure.match method), 22

H

hasattr() (built-in function), 10
hash() (built-in function), 10
heapify() (in module uheapq), 18
heappop() (in module uheapq), 18
heappush() (in module uheapq), 18
hex() (built-in function), 10
hexdigest() (uhashlib.hash method), 18
hexlify() (in module ubinascii), 16
hline() (framebuf.FrameBuffer method), 31

I

I2C (class in machine), 33
I2C.MASTER (in module machine), 35
id() (built-in function), 10
id() (machine.Pin method), 37
idle() (in module machine), 32
ifconfig() (network.wlan method), 45
implementation (in module sys), 15
init() (machine.I2C method), 34
init() (machine.Pin method), 37
init() (machine.RTC method), 39
init() (machine.SPI method), 40
input() (built-in function), 10
int (built-in class), 10
irq() (machine.Pin method), 38
irq() (machine.RTC method), 39
isconnected() (network.wlan method), 45
isfinite() (in module math), 13
isinf() (in module math), 13
isinstance() (built-in function), 10
isnan() (in module math), 13
issubclass() (built-in function), 10
iter() (built-in function), 10

L

ldexp() (in module math), 13
len() (built-in function), 11
lgamma() (in module math), 13
line() (framebuf.FrameBuffer method), 31
list (built-in class), 11
listdir() (in module uos), 20
listen() (usocket.socket method), 24
LITTLE_ENDIAN (in module uctypes), 47
loads() (in module ujson), 20
locals() (built-in function), 11
localtime() (in module utime), 27
log() (in module math), 13
log10() (in module math), 14
log2() (in module math), 14

M

machine (module), 31
machine.DEEPSLEEP (in module machine), 33
machine.DEEPSLEEP_RESET (in module machine), 33
machine.HARD_RESET (in module machine), 33
machine.IDLE (in module machine), 33
machine.PIN_WAKE (in module machine), 33
machine.PWRON_RESET (in module machine), 33
machine.RTC_WAKE (in module machine), 33

machine.SLEEP (in module machine), 33
machine.SOFT_RESET (in module machine), 33
machine.WDT_RESET (in module machine), 33
machine.WLAN_WAKE (in module machine), 33
makefile() (usocket.socket method), 25
map() (built-in function), 11
match() (in module ure), 21
match() (ure.regex method), 22
math (module), 12
max() (built-in function), 11
maxsize (in module sys), 15
mem_alloc() (in module gc), 12
mem_free() (in module gc), 12
memoryview (built-in class), 11
micropython (module), 43
min() (built-in function), 11
mkdir() (in module uos), 20
mktime() (in module utime), 27
mode() (machine.Pin method), 37
modf() (in module math), 14
modules (in module sys), 15

N

namedtuple() (in module ucollections), 16
NATIVE (in module uctypes), 47
network (module), 43
next() (built-in function), 11
now() (machine.RTC method), 39

O

object (built-in class), 11
oct() (built-in function), 11
open() (built-in function), 11
open() (in module uio), 19
ord() (built-in function), 11
OrderedDict() (in module ucollections), 17
out_value() (machine.Pin method), 37

P

pack() (in module ustruct), 26
pack_into() (in module ustruct), 26
path (in module sys), 15
phy_mode() (in module network), 43
pi (in module math), 14
Pin (class in machine), 36
Pin.ALT (in module machine), 38
Pin.ALT_OPEN_DRAIN (in module machine), 38
Pin.board (class in machine), 38
Pin.HIGH_POWER (in module machine), 39
Pin.IN (in module machine), 38
Pin.IRQ_FALLING (in module machine), 39
Pin.IRQ_HIGH_LEVEL (in module machine), 39
Pin.IRQ_LOW_LEVEL (in module machine), 39
Pin.IRQ_RISING (in module machine), 39
Pin.LOW_POWER (in module machine), 39
Pin.MED_POWER (in module machine), 39
Pin.OPEN_DRAIN (in module machine), 38
Pin.OUT (in module machine), 38
Pin.PULL_DOWN (in module machine), 38
Pin.PULL_UP (in module machine), 38
pixel() (framebuf.FrameBuffer method), 30
platform (in module sys), 15
pow() (built-in function), 11
pow() (in module math), 14
print() (built-in function), 11
print_exception() (in module sys), 14
property() (built-in function), 11
pull() (machine.Pin method), 37

R

radians() (in module math), 14
range() (built-in function), 11
read() (machine.SPI method), 40
read() (machine.UART method), 42
read() (usocket.socket method), 25
readfrom() (machine.I2C method), 34
readfrom_into() (machine.I2C method), 34
readfrom_mem() (machine.I2C method), 35
readfrom_mem_into() (machine.I2C method), 35
readinto() (machine.I2C method), 34
readinto() (machine.SPI method), 40
readinto() (machine.UART method), 42
readinto() (usocket.socket method), 25
readline() (machine.UART method), 42
readline() (usocket.socket method), 25
rect() (framebuf.FrameBuffer method), 31
recv() (usocket.socket method), 24
recvfrom() (usocket.socket method), 24
remove() (in module uos), 20
rename() (in module uos), 20
repr() (built-in function), 11
reset() (in module machine), 32
reset_cause() (in module machine), 32
reversed() (built-in function), 11
rmdir() (in module uos), 20
round() (built-in function), 11
RTC (class in machine), 39
RTC.ALARM0 (in module machine), 40

S

scan() (machine.I2C method), 34
scan() (network.wlan method), 44
scroll() (framebuf.FrameBuffer method), 31
search() (in module ure), 21
search() (ure.regex method), 22
send() (usocket.socket method), 24
sendall() (usocket.socket method), 24
sendto() (usocket.socket method), 24

set (built-in class), 11
set_native_code_location() (in module esp), 49
setattr() (built-in function), 11
setblocking() (usocket.socket method), 25
setsockopt() (usocket.socket method), 24
settimeout() (usocket.socket method), 24
sin() (in module math), 14
sinh() (in module math), 14
sizeof() (in module uctypes), 47
sleep() (in module machine), 32
sleep() (in module utime), 27
sleep_ms() (in module utime), 27
sleep_type() (in module esp), 49
sleep_us() (in module utime), 27
slice (built-in class), 11
socket.AF_INET (in module usocket), 23
socket.AF_INET6 (in module usocket), 23
socket.getaddrinfo() (in module usocket), 23
socket.IPPROTO_SEC (in module usocket), 23
socket.IPPROTO_TCP (in module usocket), 23
socket.IPPROTO_UDP (in module usocket), 23
socket.SOCK_DGRAM (in module usocket), 23
socket.SOCK_STREAM (in module usocket), 23
socket.socket() (in module usocket), 23
sorted() (built-in function), 11
SPI (class in machine), 40
SPI.LSB (in module machine), 41
SPI.MASTER (in module machine), 41
SPI.MSB (in module machine), 41
split() (ure.regex method), 22
sqrt() (in module math), 14
ssl.wrap_socket() (in module ussl), 26
start() (machine.I2C method), 34
stat() (in module uos), 20
staticmethod() (built-in function), 11
status() (network.wlan method), 45
statvfs() (in module uos), 20
stderr (in module sys), 16
stdin (in module sys), 16
stdout (in module sys), 16
stop() (machine.I2C method), 34
str (built-in class), 11
StringIO (class in uio), 19
struct (class in uctypes), 47
sum() (built-in function), 11
super() (built-in function), 11
sync() (in module uos), 21
sys (module), 14

T

tan() (in module math), 14
tanh() (in module math), 14
text() (framebuf.FrameBuffer method), 31
TextIOWrapper (class in uio), 19
ticks_add() (in module utime), 28
ticks_cpu() (in module utime), 27
ticks_diff() (in module utime), 28
ticks_ms() (in module utime), 27
ticks_us() (in module utime), 27
time() (in module utime), 29
time_pulse_us() (in module machine), 32
Timer (class in machine), 41
Timer.ONE_SHOT (in module machine), 41
Timer.PERIODIC (in module machine), 41
toggle() (machine.Pin method), 37
trunc() (in module math), 14
tuple (built-in class), 11
type() (built-in function), 11

U

ubinascii (module), 16
ucollections (module), 16
uctypes (module), 46
uhashlib (module), 17
uhashlib.md5 (class in uhashlib), 17
uhashlib.sha1 (class in uhashlib), 17
uhashlib.sha256 (class in uhashlib), 17
uheapq (module), 18
uio (module), 18
ujson (module), 20
unhexlify() (in module ubinascii), 16
unique_id() (in module machine), 32
unpack() (in module ustruct), 26
unpack_from() (in module ustruct), 26
uos (module), 20
update() (uhashlib.hash method), 18
urandom() (in module uos), 21
ure (module), 21
usocket (module), 22
ussl (module), 25
ustruct (module), 26
utime (module), 26
uzlib (module), 29

V

value() (machine.Pin method), 37
version (in module sys), 16
version_info (in module sys), 16
vline() (framebuf.FrameBuffer method), 31

W

WDT (class in machine), 43
WLAN (class in network), 44
write() (machine.I2C method), 34
write() (machine.SPI method), 41
write() (machine.UART method), 42
write() (usocket.socket method), 25
write_readinto() (machine.SPI method), 41

writeto() (machine.I2C method), 34
writeto_mem() (machine.I2C method), 35

Z

zip() (built-in function), 11