MicroPython Documentation

Release 1.9.4

Damien P. George, Paul Sokolovsky, and contributors

Oct 01, 2018

CONTENTS

1 Quick reference for the WiPy **1**
- 1.1 General board control (including sleep modes) . . . 1
- 1.2 Pins and GPIO . . . 2
- 1.3 Timers . . . 2
- 1.4 PWM (pulse width modulation) . . . 2
- 1.5 ADC (analog to digital conversion) . . . 2
- 1.6 UART (serial bus) . . . 3
- 1.7 SPI bus . . . 3
- 1.8 I2C bus . . . 3
- 1.9 Watchdog timer (WDT) . . . 3
- 1.10 Real time clock (RTC) . . . 4
- 1.11 SD card . . . 4
- 1.12 WLAN (WiFi) . . . 4
- 1.13 Telnet and FTP server . . . 5
- 1.14 Heart beat LED . . . 5

2 General information about the WiPy **7**
- 2.1 No floating point support . . . 7
- 2.2 Before applying power . . . 7
- 2.3 WLAN default behaviour . . . 7
- 2.4 Telnet REPL . . . 7
- 2.5 Local file system and FTP access . . . 8
- 2.6 FileZilla settings . . . 8
- 2.7 Upgrading the firmware Over The Air . . . 8
- 2.8 Boot modes and safe boot . . . 9
- 2.9 The heartbeat LED . . . 9
- 2.10 Details on sleep modes . . . 10
- 2.11 Additional details for machine.Pin . . . 10
- 2.12 Additional details for machine.I2C . . . 11
- 2.13 Known issues . . . 11
  - 2.13.1 Incompatible way to create SSL sockets . . . 11
  - 2.13.2 Incompatibilities in uhashlib module . . . 12
  - 2.13.3 Unrelated function in machine module . . . 12
  - 2.13.4 Adhoc way to control telnet/FTP server via network module . . . 12
  - 2.13.5 Adhoc VFS-like support . . . 13

3 WiPy tutorials and examples **15**
- 3.1 Introduction to the WiPy . . . 15
  - 3.1.1 Caring for your WiPy and expansion board . . . 15
  - 3.1.2 Plugging into the expansion board and powering on . . . 15

3.1.3 Expansion board hardware guide . . . 15
3.1.4 Powering by an external power source . . . 15
3.1.5 Performing firmware upgrades . . . 16
3.2 Getting a MicroPython REPL prompt . . . 16
3.2.1 Windows . . . 16
3.2.2 Mac OS X . . . 16
3.2.3 Linux . . . 17
3.2.4 Using the REPL prompt . . . 17
3.2.5 Resetting the board . . . 18
3.3 Getting started with Blynk and the WiPy . . . 18
3.4 WLAN step by step . . . 18
3.4.1 Connecting to your home router . . . 19
3.4.2 Assigning a static IP address when booting . . . 19
3.5 Hardware timers . . . 19
3.5.1 Additional constants for Timer class . . . 20
3.6 Reset and boot modes . . . 21
3.6.1 Safe boot . . . 21
3.6.2 Factory reset the filesystem . . . 21

4 MicroPython libraries **23**
4.1 Python standard libraries and micro-libraries . . . 23
4.1.1 Builtin functions and exceptions . . . 24
4.1.2 `array` – arrays of numeric data . . . 27
4.1.3 `gc` – control the garbage collector . . . 27
4.1.4 `sys` – system specific functions . . . 28
4.1.5 `ubinascii` – binary/ASCII conversions . . . 30
4.1.6 `ujson` – JSON encoding and decoding . . . 30
4.1.7 `uos` – basic "operating system" services . . . 31
4.1.8 `ure` – simple regular expressions . . . 34
4.1.9 `uselect` – wait for events on a set of streams . . . 35
4.1.10 `usocket` – socket module . . . 37
4.1.11 `ussl` – SSL/TLS module . . . 41
4.1.12 `utime` – time related functions . . . 42
4.2 MicroPython-specific libraries . . . 45
4.2.1 `btree` – simple BTree database . . . 45
4.2.2 `framebuf` — Frame buffer manipulation . . . 48
4.2.3 `machine` — functions related to the hardware . . . 50
4.2.4 `micropython` – access and control MicroPython internals . . . 66
4.2.5 `network` — network configuration . . . 68
4.2.6 `uctypes` – access binary data in a structured way . . . 72
4.3 Libraries specific to the WiPy . . . 75
4.3.1 `wipy` – WiPy specific features . . . 75

5 The MicroPython language **77**
5.1 Glossary . . . 77
5.2 The MicroPython Interactive Interpreter Mode (aka REPL) . . . 79
5.2.1 Auto-indent . . . 79
5.2.2 Auto-completion . . . 79
5.2.3 Interrupting a running program . . . 80
5.2.4 Paste Mode . . . 80
5.2.5 Soft Reset . . . 81
5.2.6 The special variable _ (underscore) . . . 82
5.2.7 Raw Mode . . . 82
5.3 Writing interrupt handlers . . . 82

5.3.1 Tips and recommended practices . . . 82
5.3.2 MicroPython Issues . . . 83
5.3.3 Exceptions . . . 85
5.3.4 General Issues . . . 86
5.4 Maximising MicroPython Speed . . . 88
5.4.1 Designing for speed . . . 89
5.4.2 Identifying the slowest section of code . . . 90
5.4.3 MicroPython code improvements . . . 91
5.4.4 The Native code emitter . . . 91
5.4.5 The Viper code emitter . . . 92
5.4.6 Accessing hardware directly . . . 93
5.5 MicroPython on Microcontrollers . . . 94
5.5.1 Flash Memory . . . 94
5.5.2 RAM . . . 94
5.5.3 The Heap . . . 98
5.5.4 String Operations . . . 99
5.5.5 Postscript . . . 100
5.6 Distribution packages, package management, and deploying applications . . . 100
5.6.1 Overview . . . 100
5.6.2 Distribution packages . . . 100
5.6.3 `upip` package manager . . . 101
5.6.4 Cross-installing packages . . . 102
5.6.5 Cross-installing packages with freezing . . . 102
5.6.6 Creating distribution packages . . . 103
5.6.7 Application resources . . . 103
5.6.8 References . . . 104

6 MicroPython differences from CPython 105
6.1 Syntax . . . 105
6.1.1 Spaces . . . 105
6.1.2 Unicode . . . 105
6.2 Core Language . . . 106
6.2.1 Classes . . . 106
6.2.2 Functions . . . 108
6.2.3 Generator . . . 109
6.2.4 Runtime . . . 109
6.2.5 import . . . 110
6.3 Builtin Types . . . 112
6.3.1 Exception . . . 112
6.3.2 bytearray . . . 114
6.3.3 bytes . . . 114
6.3.4 float . . . 115
6.3.5 int . . . 115
6.3.6 list . . . 115
6.3.7 str . . . 117
6.3.8 tuple . . . 119
6.4 Modules . . . 119
6.4.1 array . . . 119
6.4.2 builtins . . . 120
6.4.3 deque . . . 121
6.4.4 json . . . 121
6.4.5 struct . . . 121
6.4.6 sys . . . 122

7 MicroPython license information 123

Python Module Index 125

Index 127

CHAPTER

ONE

QUICK REFERENCE FOR THE WIPY

WiPy pinout and alternate functions table

F	E	D	C	B	A	PIN	WiPy	PIN	A	B	C	D	E
						RESET		VIN (3.6-6.5V)					
		ADC0_CH0		UART1_RX[6]	UART0_RX[3]	GP2		GND					
				UART1_TX[6]	UART0_TX[3]	GP1		3V3 OUT					
				I2C0_SCL[9]	UART1_TX[2]	GP23		GP10		UART1_TX[7]		PWM_7[3]	SD0_CLK[
		PWM_1[5]		I2C0_SDA[9]	UART1_RX[2]	GP24		GP9				PWM_6[3]	SD0_DATA0
I2S0_FS[13]	SD0_CMD[6]	PWM_8[3]		I2C0_SDA[1]	UART1_RX[7]	GP11		GP8					
I2S0_CLK[3]				I2C0_SCL[5]	UART0_TX[7]	GP12		GP7	UART0_RTS[10]	UART1_RTS[3]	UART1_TX[11]		
				I2C0_SDA[5]	UART0_RX[7]	GP13		GP6	UART0_CTS[6]	UART1_CTS[3]			
SPI0_CLK[7]				I2C0_SCL[5]		GP14		GP30	UART0_TX[9]			SPI0_MISO[7]	I2S0_FS[3
SPI0_MISO[7]	SD0_DATA0[8]			I2C0_SDA[5]		GP15		GP31	UART0_RX[9]	UART1_RX[2]		SPI0_CLK[7]	I2S0_FS[1
SPI0_MOSI[7]	SD0_CLK[8]				UART1_TX[5]	GP16		GP3		UART1_TX[6]			
SPI0_CS[7]	SD0_CMD[8]				UART1_RX[5]	GP17		GP0	UART0_RTS[3]	UART1_RTS[10]		SPI0_CS[9]	I2S0_DATA0
I2S0_FS[7]						GP22		GP4		UART1_RX[6]			
					SAFE_BOOT	GP28		GP5					I2S0_DATA1
							Antenna						

Timer	Channel	PWM pin
0	A	PWM_1
	B	
1	A	PWM_3
	B	
2	A	
	B	PWM_6
3	A	PWM_7
	B	PWM_8

Remarks:

- *The number in brackets next to each function is the one to be used when remapping the pin. In order to use the pin in GPIO mode, alternate function 0 must be selected*
- *ADC pin input range is 0-1.4V (being 1.8V the absolute maximum that it can withstand). When GP2, GP3, GP4 or GP5 are remapped to the ADC block, 1.8 V is the maximum. If these pins are used in digital mode, then the maximum allowed input is 3.6V.*
- *The heart beat LED is connected to GP25 and also has PWM_3 functionality with the alternate function 9.*

1.1 General board control (including sleep modes)

See the *machine* module:

```
import machine

help(machine) # display all members from the machine module
machine.freq() # get the CPU frequency
machine.unique_id() # return the 6-byte unique id of the board (the WiPy's MAC␣
↪address)

machine.idle()          # average current decreases to (~12mA), any interrupts wake it␣
↪up
machine.sleep()         # everything except for WLAN is powered down (~950uA avg.␣
↪current)
                        # wakes from Pin, RTC or WLAN
machine.deepsleep()     # deepest sleep mode, MCU starts from reset. Wakes from Pin and␣
↪RTC.
```

1.2 Pins and GPIO

See *machine.Pin*.

```
from machine import Pin

# initialize GP2 in gpio mode (alt=0) and make it an output
p_out = Pin('GP2', mode=Pin.OUT)
p_out.value(1)
p_out.value(0)
p_out.toggle()
p_out(True)

# make GP1 an input with the pull-up enabled
p_in = Pin('GP1', mode=Pin.IN, pull=Pin.PULL_UP)
p_in() # get value, 0 or 1
```

1.3 Timers

See *machine.Timer* and *machine.Pin*. Timer `id`'s take values from 0 to 3.:

```
from machine import Timer
from machine import Pin

tim = Timer(0, mode=Timer.PERIODIC)
tim_a = tim.channel(Timer.A, freq=1000)
tim_a.freq(5) # 5 Hz

p_out = Pin('GP2', mode=Pin.OUT)
tim_a.irq(trigger=Timer.TIMEOUT, handler=lambda t: p_out.toggle())
```

1.4 PWM (pulse width modulation)

See *machine.Pin* and *machine.Timer*.

```
from machine import Timer

# timer 1 in PWM mode and width must be 16 buts
tim = Timer(1, mode=Timer.PWM, width=16)

# enable channel A @1KHz with a 50.55% duty cycle
tim_a = tim.channel(Timer.A, freq=1000, duty_cycle=5055)
```

1.5 ADC (analog to digital conversion)

See *machine.ADC*.

```
from machine import ADC

adc = ADC()
apin = adc.channel(pin='GP3')
apin() # read value, 0-4095
```

1.6 UART (serial bus)

See *machine.UART*.

```
from machine import UART
uart = UART(0, baudrate=9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

1.7 SPI bus

See *machine.SPI*.

```
from machine import SPI

# configure the SPI master @ 2MHz
spi = SPI(0, SPI.MASTER, baudrate=200000, polarity=0, phase=0)
spi.write('hello')
spi.read(5) # receive 5 bytes on the bus
rbuf = bytearray(5)
spi.write_readinto('hello', rbuf) # send and receive 5 bytes
```

1.8 I2C bus

See *machine.I2C*.

```
from machine import I2C
# configure the I2C bus
i2c = I2C(baudrate=100000)
i2c.scan() # returns list of slave addresses
i2c.writeto(0x42, 'hello') # send 5 bytes to slave with address 0x42
i2c.readfrom(0x42, 5) # receive 5 bytes from slave
i2c.readfrom_mem(0x42, 0x10, 2) # read 2 bytes from slave 0x42, slave memory 0x10
i2c.writeto_mem(0x42, 0x10, 'xy') # write 2 bytes to slave 0x42, slave memory 0x10
```

1.9 Watchdog timer (WDT)

See *machine.WDT*.

```
from machine import WDT

# enable the WDT with a timeout of 5s (1s is the minimum)
```

```
wdt = WDT(timeout=5000)
wdt.feed()
```

1.10 Real time clock (RTC)

See *machine.RTC*

```
from machine import RTC

rtc = RTC() # init with default time and date
rtc = RTC(datetime=(2015, 8, 29, 9, 0, 0, 0, None)) # init with a specific time and
↪date
print(rtc.now())

def alarm_handler (rtc_o):
    pass
    # do some non blocking operations
    # warning printing on an irq via telnet is not
    # possible, only via UART

# create a RTC alarm that expires after 5 seconds
rtc.alarm(time=5000, repeat=False)

# enable RTC interrupts
rtc_i = rtc.irq(trigger=RTC.ALARM0, handler=alarm_handler, wake=machine.SLEEP)

# go into suspended mode waiting for the RTC alarm to expire and wake us up
machine.sleep()
```

1.11 SD card

See *machine.SD*.

```
from machine import SD
import os

# clock pin, cmd pin, data0 pin
sd = SD(pins=('GP10', 'GP11', 'GP15'))
# or use default ones for the expansion board
sd = SD()
os.mount(sd, '/sd')
```

1.12 WLAN (WiFi)

See *network.WLAN* and *machine*.

```
import machine
from network import WLAN

# configure the WLAN subsystem in station mode (the default is AP)
```

```
wlan = WLAN(mode=WLAN.STA)
# go for fixed IP settings
wlan.ifconfig(config=('192.168.0.107', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
wlan.scan()     # scan for available networks
wlan.connect(ssid='mynetwork', auth=(WLAN.WPA2, 'mynetworkkey'))
while not wlan.isconnected():
    pass
print(wlan.ifconfig())
# enable wake on WLAN
wlan.irq(trigger=WLAN.ANY_EVENT, wake=machine.SLEEP)
# go to sleep
machine.sleep()
# now, connect to the FTP or the Telnet server and the WiPy will wake-up
```

1.13 Telnet and FTP server

See network.Server

```
from network import Server

# init with new user, password and seconds timeout
server = Server(login=('user', 'password'), timeout=60)
server.timeout(300) # change the timeout
server.timeout() # get the timeout
server.isrunning() # check whether the server is running or not
```

1.14 Heart beat LED

See *wipy*.

```
import wipy

wipy.heartbeat(False)   # disable the heartbeat LED
wipy.heartbeat(True)    # enable the heartbeat LED
wipy.heartbeat()        # get the heartbeat state
```

CHAPTER

TWO

GENERAL INFORMATION ABOUT THE WIPY

2.1 No floating point support

Due to space reasons, there's no floating point support, and no math module. This means that floating point numbers cannot be used anywhere in the code, and that all divisions must be performed using '//' instead of '/'. Example:

```
>>> r = 4 // 2  # this will work
>>> r = 4 / 2   # this WON'T
```

2.2 Before applying power

Warning: The GPIO pins of the WiPy are NOT 5V tolerant, connecting them to voltages higher than 3.6V will cause irreparable damage to the board. ADC pins, when configured in analog mode cannot withstand voltages above 1.8V. Keep these considerations in mind when wiring your electronics.

2.3 WLAN default behaviour

When the WiPy boots with the default factory configuration starts in Access Point mode with `ssid` that starts with: `wipy-wlan` and `key: www.wipy.io`. Connect to this network and the WiPy will be reachable at `192.168.1.1`. In order to gain access to the interactive prompt, open a telnet session to that IP address on the default port (23). You will be asked for credentials: `login: micro` and `password: python`

2.4 Telnet REPL

Linux stock telnet works like a charm (also on OSX), but other tools like putty work quite well too. The default credentials are: **user:** `micro`, **password:** `python`. See network.server for info on how to change the defaults. For instance, on a linux shell (when connected to the WiPy in AP mode):

```
$ telnet 192.168.1.1
```

2.5 Local file system and FTP access

There is a small internal file system (a drive) on the WiPy, called `/flash`, which is stored within the external serial flash memory. If a micro SD card is hooked-up and mounted, it will be available as well.

When the WiPy starts up, it always boots from the `boot.py` located in the `/flash` file system. On boot up, the current directory is `/flash`.

The file system is accessible via the native FTP server running in the WiPy. Open your FTP client of choice and connect to:

url: `ftp://192.168.1.1`, **user:** `micro`, **password:** `python`

See network.server for info on how to change the defaults. The recommended clients are: Linux stock FTP (also in OSX), Filezilla and FireFTP. For example, on a linux shell:

```
$ ftp 192.168.1.1
```

The FTP server on the WiPy doesn't support active mode, only passive, therefore, if using the native unix ftp client, just after logging in do:

```
ftp> passive
```

Besides that, the FTP server only supports one data connection at a time. Check out the Filezilla settings section below for more info.

2.6 FileZilla settings

Do not use the quick connect button, instead, open the site manager and create a new configuration. In the `General` tab make sure that encryption is set to: `Only use plain FTP (insecure)`. In the Transfer Settings tab limit the max number of connections to one, otherwise FileZilla will try to open a second command connection when retrieving and saving files, and for simplicity and to reduce code size, only one command and one data connections are possible. Other FTP clients might behave in a similar way.

2.7 Upgrading the firmware Over The Air

OTA software updates can be performed through the FTP server. Upload the `mcuimg.bin` file to: `/flash/sys/mcuimg.bin` it will take around 6s. You won't see the file being stored inside `/flash/sys/` because it's actually saved bypassing the user file system, so it ends up inside the internal **hidden** file system, but rest assured that it was successfully transferred, and it has been signed with a MD5 checksum to verify its integrity. Now, reset the WiPy by pressing the switch on the board, or by typing:

```
>>> import machine
>>> machine.reset()
```

Software updates can be found in: https://github.com/wipy/wipy/releases (**Binaries.zip**). It's always recommended to update to the latest software, but make sure to read the **release notes** before.

Note: The `bootloader.bin` found inside `Binaries.zip` is there only for reference, it's not needed for the Over The Air update.

In order to check your software version, do:

```
>>> import os
>>> os.uname().release
```

If the version number is lower than the latest release found in the releases, go ahead and update your WiPy!

2.8 Boot modes and safe boot

If you power up normally, or press the reset button, the WiPy will boot into standard mode; the `boot.py` file will be executed first, then `main.py` will run.

You can override this boot sequence by pulling `GP28` **up** (connect it to the 3v3 output pin) during reset. This procedure also allows going back in time to old firmware versions. The WiPy can hold up to 3 different firmware versions, which are: the factory firmware plus 2 user updates.

After reset, if `GP28` is held high, the heartbeat LED will start flashing slowly, if after 3 seconds the pin is still being held high, the LED will start blinking a bit faster and the WiPy will select the previous user update to boot. If the previous user update is the desired firmware image, `GP28` must be released before 3 more seconds elapse. If 3 seconds later the pin is still high, the factory firmware will be selected, the LED will flash quickly for 1.5 seconds and the WiPy will proceed to boot. The firmware selection mechanism is as follows:

Safe Boot Pin `GP28` released during:

1st 3 secs window	2nd 3 secs window	Final 1.5 secs window
Safe boot, *latest* firmware is selected	Safe boot, *previous* user update selected	Safe boot, the *factory* firmware is selected

On all of the above 3 scenarios, safe boot mode is entered, meaning that the execution of both `boot.py` and `main.py` is skipped. This is useful to recover from crash situations caused by the user scripts. The selection made during safe boot is not persistent, therefore after the next normal reset the latest firmware will run again.

2.9 The heartbeat LED

By default the heartbeat LED flashes once every 4s to signal that the system is alive. This can be overridden through the *wipy* module:

```
>>> import wipy
>>> wipy.heartbeat(False)
```

There are currently 2 kinds of errors that you might see:

1. If the heartbeat LED flashes quickly, then a Python script (eg `main.py`) has an error. Use the REPL to debug it.
2. If the heartbeat LED stays on, then there was a hard fault, you cannot recover from this, the only way out is to press the reset switch.

2.10 Details on sleep modes

- `machine.idle()`: Power consumption: ~12mA (in WLAN STA mode). Wake sources: any hardware interrupt (including systick with period of 1ms), no special configuration required.
- `machine.sleep()`: 950uA (in WLAN STA mode). Wake sources are `Pin`, `RTC` and `WLAN`
- `machine.deepsleep()`: ~350uA. Wake sources are `Pin` and `RTC`.

2.11 Additional details for machine.Pin

On the WiPy board the pins are identified by their string id:

```
from machine import Pin
g = machine.Pin('GP9', mode=Pin.OUT, pull=None, drive=Pin.MED_POWER, alt=-1)
```

You can also configure the Pin to generate interrupts. For instance:

```
from machine import Pin

def pincb(pin):
    print(pin.id())

pin_int = Pin('GP10', mode=Pin.IN, pull=Pin.PULL_DOWN)
pin_int.irq(trigger=Pin.IRQ_RISING, handler=pincb)
# the callback can be triggered manually
pin_int.irq()()
# to disable the callback
pin_int.irq().disable()
```

Now every time a falling edge is seen on the gpio pin, the callback will be executed. Caution: mechanical push buttons have "bounce" and pushing or releasing a switch will often generate multiple edges. See: http://www.eng.utah.edu/~cs5780/debouncing.pdf for a detailed explanation, along with various techniques for debouncing.

All pin objects go through the pin mapper to come up with one of the gpio pins.

For the `drive` parameter the strengths are:

- `Pin.LOW_POWER` - 2mA drive capability.
- `Pin.MED_POWER` - 4mA drive capability.
- `Pin.HIGH_POWER` - 6mA drive capability.

For the `alt` parameter please refer to the pinout and alternate functions table at <https://raw.githubusercontent.com/wipy/wipy/master/docs/PinOUT.png>‘_ for the specific alternate functions that each pin supports.

For interrupts, the `priority` can take values in the range 1-7. And the `wake` parameter has the following properties:

- If `wake_from=machine.Sleep.ACTIVE` any pin can wake the board.
- If `wake_from=machine.Sleep.SUSPENDED` pins `GP2`, `GP4`, `GP10`, `GP11`, GP17‘‘ or `GP24` can wake the board. Note that only 1 of this pins can be enabled as a wake source at the same time, so, only the last enabled pin as a `machine.Sleep.SUSPENDED` wake source will have effect.
- If `wake_from=machine.Sleep.SUSPENDED` pins `GP2`, `GP4`, `GP10`, `GP11`, `GP17` and `GP24` can wake the board. In this case all of the 6 pins can be enabled as a `machine.Sleep.HIBERNATE` wake source at the same time.

Additional Pin methods:

machine.Pin.**alt_list**()
Returns a list of the alternate functions supported by the pin. List items are a tuple of the form: ('ALT_FUN_NAME', ALT_FUN_INDEX)

2.12 Additional details for machine.I2C

On the WiPy there is a single hardware I2C peripheral, identified by "0". By default this is the peripheral that is used when constructing an I2C instance. The default pins are GP23 for SCL and GP13 for SDA, and one can create the default I2C peripheral simply by doing:

```
i2c = machine.I2C()
```

The pins and frequency can be specified as:

```
i2c = machine.I2C(freq=400000, scl='GP23', sda='GP13')
```

Only certain pins can be used as SCL/SDA. Please refer to the pinout for further information.

2.13 Known issues

2.13.1 Incompatible way to create SSL sockets

SSL sockets need to be created the following way before wrapping them with. `ssl.wrap_socket`:

```
import socket
import ssl
s = socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_SEC)
ss = ssl.wrap_socket(s)
```

Certificates must be used in order to validate the other side of the connection, and also to authenticate ourselves with the other end. Such certificates must be stored as files using the FTP server, and they must be placed in specific paths with specific names.

- The certificate to validate the other side goes in: **'/flash/cert/ca.pem'**
- The certificate to authenticate ourselves goes in: **'/flash/cert/cert.pem'**
- The key for our own certificate goes in: **'/flash/cert/private.key'**

Note: When these files are stored, they are placed inside the internal **hidden** file system (just like firmware updates), and therefore they are never visible.

For instance to connect to the Blynk servers using certificates, take the file `ca.pem` located in the blynk examples folder. and put it in '/flash/cert/'. Then do:

```
import socket
import ssl
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_SEC)
ss = ssl.wrap_socket(s, cert_reqs=ssl.CERT_REQUIRED, ca_certs='/flash/cert/ca.pem')
ss.connect(socket.getaddrinfo('cloud.blynk.cc', 8441)[0][-1])
```

2.13.2 Incompatibilities in uhashlib module

Due to hardware implementation details of the WiPy, data must be buffered before being digested, which would make it impossible to calculate the hash of big blocks of data that do not fit in RAM. In this case, since most likely the total size of the data is known in advance, the size can be passed to the constructor and hence the HASH hardware engine of the WiPy can be properly initialized without needing buffering. If `block_size` is to be given, an initial chunk of `data` must be passed as well. **When using this extension, care must be taken to make sure that the length of all intermediate chunks (including the initial one) is a multiple of 4 bytes.** The last chunk may be of any length.

Example:

```
hash = uhashlib.sha1('abcd1234', 1001)     # length of the initial piece is multiple␣
↪of 4 bytes
hash.update('1234')                        # also multiple of 4 bytes
...
hash.update('12345')                       # last chunk may be of any length
hash.digest()
```

2.13.3 Unrelated function in machine module

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

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

2.13.4 Adhoc way to control telnet/FTP server via network module

The `Server` class controls the behaviour and the configuration of the FTP and telnet services running on the WiPy. Any changes performed using this class' methods will affect both.

Example:

```
import network
server = network.Server()
server.deinit() # disable the server
# enable the server again with new settings
server.init(login=('user', 'password'), timeout=600)
```

class network.**Server** (*id, ...*)
Create a server instance, see `init` for parameters of initialization.

server.**init** (**, login=('micro', 'python'), timeout=300*)
Init (and effectively start the server). Optionally a new `user`, `password` and `timeout` (in seconds) can be passed.

server.**deinit** ()
Stop the server

server.**timeout** ([*timeout_in_seconds*])
Get or set the server timeout.

server.**isrunning** ()
Returns `True` if the server is running, `False` otherwise.

2.13.5 Adhoc VFS-like support

WiPy doesn't implement full MicroPython VFS support, instead following functions are defined in `uos` module:

mount (*block_device*, *mount_point*, *, *readonly=False*)
Mounts a block device (like an `SD` object) in the specified mount point. Example:

```
os.mount(sd, '/sd')
```

unmount (*path*)
Unmounts a previously mounted block device from the given path.

mkfs (*block_device or path*)
Formats the specified path, must be either `/flash` or `/sd`. A block device can also be passed like an `SD` object before being mounted.

CHAPTER

THREE

WIPY TUTORIALS AND EXAMPLES

Before starting, make sure that you are running the latest firmware, for instructions see *OTA How-To*.

3.1 Introduction to the WiPy

To get the most out of your WiPy, there are a few basic things to understand about how it works.

3.1.1 Caring for your WiPy and expansion board

Because the WiPy/expansion board does not have a housing it needs a bit of care:

- Be gentle when plugging/unplugging the USB cable. Whilst the USB connector is well soldered and is relatively strong, if it breaks off it can be very difficult to fix.
- Static electricity can shock the components on the WiPy and destroy them. If you experience a lot of static electricity in your area (eg dry and cold climates), take extra care not to shock the WiPy. If your WiPy came in a ESD bag, then this bag is the best way to store and carry the WiPy as it will protect it against static discharges.

As long as you take care of the hardware, you should be okay. It's almost impossible to break the software on the WiPy, so feel free to play around with writing code as much as you like. If the filesystem gets corrupt, see below on how to reset it. In the worst case you might need to do a safe boot, which is explained in detail *here*.

3.1.2 Plugging into the expansion board and powering on

The expansion board can power the WiPy via USB. The WiPy comes with a sticker on top of the RF shield that labels all pins, and this should match the label numbers on the expansion board headers. When plugging it in, the WiPy antenna will end up on top of the SD card connector of the expansion board. A video showing how to do this can be found here.

3.1.3 Expansion board hardware guide

The document explaining the hardware details of the expansion board can be found here.

3.1.4 Powering by an external power source

The WiPy can be powered by a battery or other external power source.

Be sure to connect the positive lead of the power supply to VIN, and ground to GND. There is no polarity protection on the WiPy so you must be careful when connecting anything to VIN.

- When powering via `VIN`:

  The input voltage must be between 3.6V and 5.5V.

- When powering via `3V3`:

  The input voltage must be exactly 3V3, ripple free and from a supply capable of sourcing at least 300mA of current

3.1.5 Performing firmware upgrades

For detailed instructions see *OTA How-To*.

3.2 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 WiPy. Using the REPL is by far the easiest way to test out your code and run commands. You can use the REPL in addition to writing scripts in `main.py`. To use the REPL, you must connect to the WiPy either via *telnet*, or with a USB to serial converter wired to one of the two UARTs on the WiPy. To enable REPL duplication on UART0 (the one accessible via the expansion board) do:

```
>>> from machine import UART
>>> import os
>>> uart = UART(0, 115200)
>>> os.dupterm(uart)
```

Place this piece of code inside your `boot.py` so that it's done automatically after reset.

3.2.1 Windows

First you need to install the FTDI drivers for the expansion board's USB to serial converter. Then you need a terminal software. The best option is to download the free program PuTTY: putty.exe.

In order to get to the telnet REPL:

Using putty, select `Telnet` as connection type, leave the default port (23) and enter the IP address of your WiPy (192.168.1.1 when in `WLAN.AP` mode), then click open.

In order to get to the REPL UART:

Using your serial program you must connect to the COM port that you found in the previous step. With PuTTY, click on "Session" in the left-hand panel, then click the "Serial" radio button on the right, then enter you COM port (eg COM4) in the "Serial Line" box. Finally, click the "Open" button.

3.2.2 Mac OS X

Open a terminal and run:

```
$ telnet 192.168.1.1
```

or:

```
$ screen /dev/tty.usbmodem* 115200
```

When you are finished and want to exit `screen`, type CTRL-A CTRL-. If your keyboard does not have a -key (i.e. you need an obscure combination for \ like ALT-SHIFT-7) you can remap the `quit` command:

- create `~/.screenrc`
- add `bind q quit`

This will allow you to quit `screen` by hitting CTRL-A Q.

3.2.3 Linux

Open a terminal and run:

```
$ telnet 192.168.1.1
```

or:

```
$ screen /dev/ttyUSB0 115200
```

You can also try `picocom` or `minicom` instead of screen. You may have to use `/dev/ttyUSB01` or a higher number for `ttyUSB`. And, you may need to give yourself the correct permissions to access this devices (eg group `uucp` or `dialout`, or use sudo).

3.2.4 Using the REPL prompt

Now let's try running some MicroPython code directly on the WiPy.

With your serial program open (PuTTY, screen, picocom, etc) you may see a blank screen with a flashing cursor. Press Enter and you should be presented with a MicroPython prompt, i.e. `>>>`. Let's make sure it is working with the obligatory test:

```
>>> print("hello WiPy!")
hello WiPy!
```

In the above, you should not type in the `>>>` characters. They are there to indicate that you should type the text after it at the prompt. In the end, once you have entered the text `print("hello WiPy!")` and pressed Enter, the output on your screen should look like it does above.

If you already know some Python you can now try some basic commands here.

If any of this is not working you can try either a hard reset or a soft reset; see below.

Go ahead and try typing in some other commands. For example:

```
>>> from machine import Pin
>>> import wipy
>>> wipy.heartbeat(False)   # disable the heartbeat
>>> led = Pin('GP25', mode=Pin.OUT)
>>> led(1)
>>> led(0)
>>> led.toggle()
>>> 1 + 2
3
>>> 4 // 2
2
>>> 20 * 'py'
'pypypyp
```

3.2.5 Resetting the board

If something goes wrong, you can reset the board in two ways. The first is to press CTRL-D at the MicroPython prompt, which performs a soft reset. You will see a message something like:

```
>>>
PYB: soft reboot
MicroPython v1.4.6-146-g1d8b5e5 on 2015-10-21; WiPy with CC3200
Type "help()" for more information.
>>>
```

If that isn't working you can perform a hard reset (turn-it-off-and-on-again) by pressing the RST switch (the small black button next to the heartbeat LED). During telnet, this will end your session, disconnecting whatever program that you used to connect to the WiPy.

3.3 Getting started with Blynk and the WiPy

Blynk is a platform with iOS and Android apps to control Arduino, Raspberry Pi and the likes over the Internet. You can easily build graphic interfaces for all your projects by simply dragging and dropping widgets.

There are several examples available that work out-of-the-box with the WiPy. Before anything else, make sure that your WiPy is running the latest software, check *OTA How-To* for instructions.

1. Get the Blynk library and put it in `/flash/lib/` via FTP.
2. Get the Blynk examples, edit the network settings, and afterwards upload them to `/flash/lib/` via FTP as well.
3. Follow the instructions on each example to setup the Blynk dashboard on your smartphone or tablet.
4. Give it a try, for instance:

```
>>> execfile('01_simple.py')
```

3.4 WLAN step by step

The WLAN is a system feature of the WiPy, therefore it is always enabled (even while in `machine.SLEEP`), except when deepsleep mode is entered.

In order to retrieve the current WLAN instance, do:

```
>>> from network import WLAN
>>> wlan = WLAN() # we call the constructor without params
```

You can check the current mode (which is always `WLAN.AP` after power up):

```
>>> wlan.mode()
```

Warning: When you change the WLAN mode following the instructions below, your WLAN connection to the WiPy will be broken. This means you will not be able to run these commands interactively over the WLAN.

There are two ways around this::

1. put this setup code into your *boot.py file* so that it gets executed automatically after reset.

2. *duplicate the REPL on UART*, so that you can run commands via USB.

3.4.1 Connecting to your home router

The WLAN network card always boots in `WLAN.AP` mode, so we must first configure it as a station:

```
from network import WLAN
wlan = WLAN(mode=WLAN.STA)
```

Now you can proceed to scan for networks:

```
nets = wlan.scan()
for net in nets:
    if net.ssid == 'mywifi':
        print('Network found!')
        wlan.connect(net.ssid, auth=(net.sec, 'mywifikey'), timeout=5000)
        while not wlan.isconnected():
            machine.idle() # save power while waiting
        print('WLAN connection succeeded!')
        break
```

3.4.2 Assigning a static IP address when booting

If you want your WiPy to connect to your home router after boot-up, and with a fixed IP address so that you can access it via telnet or FTP, use the following script as /flash/boot.py:

```
import machine
from network import WLAN
wlan = WLAN() # get current object, without changing the mode

if machine.reset_cause() != machine.SOFT_RESET:
    wlan.init(WLAN.STA)
    # configuration below MUST match your home router settings!!
    wlan.ifconfig(config=('192.168.178.107', '255.255.255.0', '192.168.178.1', '8.8.8.
↪8'))

if not wlan.isconnected():
    # change the line below to match your network ssid, security and password
    wlan.connect('mywifi', auth=(WLAN.WPA2, 'mywifikey'), timeout=5000)
    while not wlan.isconnected():
        machine.idle() # save power while waiting
```

Note: Notice how we check for the reset cause and the connection status, this is crucial in order to be able to soft reset the WiPy during a telnet session without breaking the connection.

3.5 Hardware timers

Timers can be used for a great variety of tasks, calling a function periodically, counting events, and generating a PWM signal are among the most common use cases. Each timer consists of two 16-bit channels and this channels can be

tied together to form one 32-bit timer. The operating mode needs to be configured per timer, but then the period (or the frequency) can be independently configured on each channel. By using the callback method, the timer event can call a Python function.

Example usage to toggle an LED at a fixed frequency:

```
from machine import Timer
from machine import Pin
led = Pin('GP16', mode=Pin.OUT)                     # enable GP16 as output to drive the␣
↪LED
tim = Timer(3)                                      # create a timer object using timer 3
tim.init(mode=Timer.PERIODIC)                       # initialize it in periodic mode
tim_ch = tim.channel(Timer.A, freq=5)               # configure channel A at a frequency␣
↪of 5Hz
tim_ch.irq(handler=lambda t:led.toggle(), trigger=Timer.TIMEOUT)        # toggle a␣
↪LED on every cycle of the timer
```

Example using named function for the callback:

```
from machine import Timer
from machine import Pin
tim = Timer(1, mode=Timer.PERIODIC, width=32)
tim_a = tim.channel(Timer.A | Timer.B, freq=1)    # 1 Hz frequency requires a 32 bit␣
↪timer

led = Pin('GP16', mode=Pin.OUT) # enable GP16 as output to drive the LED

def tick(timer):                # we will receive the timer object when being called
    global led
    led.toggle()                # toggle the LED

tim_a.irq(handler=tick, trigger=Timer.TIMEOUT)          # create the interrupt
```

Further examples:

```
from machine import Timer
tim1 = Timer(1, mode=Timer.ONE_SHOT)                                    # initialize it in␣
↪one shot mode
tim2 = Timer(2, mode=Timer.PWM)                                         # initialize it in␣
↪PWM mode
tim1_ch = tim1.channel(Timer.A, freq=10, polarity=Timer.POSITIVE)   # start the event␣
↪counter with a frequency of 10Hz and triggered by positive edges
tim2_ch = tim2.channel(Timer.B, freq=10000, duty_cycle=5000)        # start the PWM on␣
↪channel B with a 50% duty cycle
tim2_ch.freq(20)                                                        # set the␣
↪frequency (can also get)
tim2_ch.duty_cycle(3010)                                                # set the duty␣
↪cycle to 30.1% (can also get)
tim2_ch.duty_cycle(3020, Timer.NEGATIVE)                                # set the duty␣
↪cycle to 30.2% and change the polarity to negative
tim2_ch.period(2000000)                                                 # change the␣
↪period to 2 seconds
```

3.5.1 Additional constants for Timer class

`Timer.PWM`
PWM timer operating mode.

`Timer.A`

`Timer.B`
Selects the timer channel. Must be ORed (`Timer.A | Timer.B`) when using a 32-bit timer.

`Timer.POSITIVE`

`Timer.NEGATIVE`
Timer channel polarity selection (only relevant in PWM mode).

`Timer.TIMEOUT`

`Timer.MATCH`
Timer channel IRQ triggers.

3.6 Reset and boot modes

There are soft resets and hard resets.

- A soft reset simply clears the state of the MicroPython virtual machine, but leaves hardware peripherals unaffected. To do a soft reset, simply press **Ctrl+D** on the REPL, or within a script do:

```
import sys
sys.exit()
```

- A hard reset is the same as performing a power cycle to the board. In order to hard reset the WiPy, press the switch on the board or:

```
import machine
machine.reset()
```

3.6.1 Safe boot

If something goes wrong with your WiPy, don't panic! It is almost impossible for you to break the WiPy by programming the wrong thing.

The first thing to try is to boot in safe mode: this temporarily skips execution of `boot.py` and `main.py` and gives default WLAN settings.

If you have problems with the filesystem you can *format the internal flash drive*.

To boot in safe mode, follow the detailed instructions described *here*.

In safe mode, the `boot.py` and `main.py` files are not executed, and so the WiPy boots up with default settings. This means you now have access to the filesystem, and you can edit `boot.py` and `main.py` to fix any problems.

Entering safe mode is temporary, and does not make any changes to the files on the WiPy.

3.6.2 Factory reset the filesystem

If you WiPy's filesystem gets corrupted (very unlikely, but possible), you can format it very easily by doing:

```
>>> import os
>>> os.mkfs('/flash')
```

Resetting the filesystem deletes all files on the internal WiPy storage (not the SD card), and restores the files `boot.py` and `main.py` back to their original state after the next reset.

CHAPTER

FOUR

MICROPYTHON LIBRARIES

Warning: Important summary of this section

- MicroPython implements a subset of Python functionality for each module.
- To ease extensibility, MicroPython versions of standard Python modules usually have `u` ("micro") prefix.
- Any particular MicroPython variant or port may miss any feature/function described in this general documentation (due to resource constraints or other limitations).

This chapter describes modules (function and class libraries) which are built into MicroPython. There are a few categories of such 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 *MicroPython port* and thus not portable.

Note about the availability of the modules and their contents: This documentation in general aspires to describe all modules and functions/classes which are implemented in MicroPython project. 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 system. 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 *MicroPython 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 *micropython-lib*.

4.1 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 the *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 library path (`sys.path`). For example, `import json` will first search for a file `json.py` (or package directory `json`) and load that module if it is found. If nothing is found, it will fallback to loading the built-in `ujson` module.

4.1.1 Builtin functions and exceptions

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

Functions and types

abs()

all()

any()

bin()

class bool

class bytearray

class bytes
See CPython documentation: `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

classmethod from_bytes(*bytes*, *byteorder*)
In MicroPython, `byteorder` parameter must be positional (this is compatible with CPython).

to_bytes(*size*, *byteorder*)
In MicroPython, `byteorder` parameter must be positional (this is compatible with CPython).

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()

Exceptions

exception AssertionError

exception AttributeError

exception Exception

exception ImportError

exception IndexError

exception KeyboardInterrupt

exception KeyError

exception MemoryError

exception NameError

exception NotImplementedError

exception OSError
See CPython documentation: `OSError`. MicroPython doesn't implement `errno` attribute, instead use the standard way to access exception arguments: `exc.args[0]`.

exception RuntimeError

exception StopIteration

exception SyntaxError

exception SystemExit
See CPython documentation: `SystemExit`.

exception TypeError
See CPython documentation: `TypeError`.

exception ValueError

exception ZeroDivisionError

4.1.2 array – arrays of numeric data

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: array.

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 *iterable*. If it is not provided, an empty array is created.

append(*val*)

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

extend(*iterable*)

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

4.1.3 gc – control the garbage collector

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: gc.

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.

Difference to CPython

This function is MicroPython extension.

---

gc.**mem_free**()

Return the number of bytes of available heap RAM, or -1 if this amount is not known.

Difference to CPython

This function is MicroPython extension.

---

gc.**threshold**([*amount*])

Set or query the additional GC allocation threshold. Normally, a collection is triggered only when a new allocation cannot be satisfied, i.e. on an out-of-memory (OOM) condition. If this function is called, in addition to

OOM, a collection will be triggered each time after *amount* bytes have been allocated (in total, since the previous time such an amount of bytes have been allocated). *amount* is usually specified as less than the full heap size, with the intention to trigger a collection earlier than when the heap becomes exhausted, and in the hope that an early collection will prevent excessive memory fragmentation. This is a heuristic measure, the effect of which will vary from application to application, as well as the optimal value of the *amount* parameter.

Calling the function without argument will return the current value of the threshold. A value of -1 means a disabled allocation threshold.

Difference to CPython

This function is a MicroPython extension. CPython has a similar function - `set_threshold()`, but due to different GC implementations, its signature and semantics are different.

4.1.4 `sys` – system specific functions

This module implements a subset of the corresponding `CPython` module, as described below. For more information, refer to the original CPython documentation: `sys`.

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.

4.1.5 ubinascii – binary/ASCII conversions

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: binascii.

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*)
Decode base64-encoded data, ignoring invalid characters in the input. Conforms to RFC 2045 s.6.8. Returns a bytes object.

ubinascii.**b2a_base64**(*data*)
Encode binary data in base64 format, as in RFC 3548. Returns the encoded data followed by a newline character, as a bytes object.

4.1.6 ujson – JSON encoding and decoding

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: json.

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

Functions

ujson.**dump**(*obj*, *stream*)
Serialise *obj* to a JSON string, writing it to the given *stream*.

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

ujson.**load**(*stream*)
Parse the given *stream*, interpreting it as a JSON string and deserialising the data to a Python object. The resulting object is returned.

Parsing continues until end-of-file is encountered. A *ValueError* is raised if the data in *stream* is not correctly formed.

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

4.1.7 uos – basic "operating system" services

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: os.

The uos module contains functions for filesystem access and mounting, terminal redirection and duplication, and the uname and urandom functions.

General functions

uos.**uname**()
Return a tuple (possibly a named tuple) containing information about the underlying machine and/or its operating system. The tuple has five fields in the following order, each of them being a string:

- sysname – the name of the underlying system
- nodename – the network name (can be the same as sysname)
- release – the version of the underlying system
- version – the MicroPython version and build date
- machine – an identifier for the underlying hardware (eg board, CPU)

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

Filesystem access

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

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

uos.**ilistdir**([*dir*])
This function returns an iterator which then yields tuples corresponding to the entries in the directory that it is listing. With no argument it lists the current directory, otherwise it lists the directory given by *dir*.

The tuples have the form *(name, type, inode[, size])*:

- *name* is a string (or bytes if *dir* is a bytes object) and is the name of the entry;
- *type* is an integer that specifies the type of the entry, with 0x4000 for directories and 0x8000 for regular files;
- *inode* is an integer corresponding to the inode of the file, and may be 0 for filesystems that don't have such a notion.
- Some platforms may return a 4-tuple that includes the entry's *size*. For file entries, *size* is an integer representing the size of the file or -1 if unknown. Its meaning is currently undefined for directory entries.

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.

Terminal redirection and duplication

uos.**dupterm**(*stream_object*, *index=0*)
Duplicate or switch the MicroPython terminal (the REPL) on the given *stream*-like object. The *stream_object* argument must implement the `readinto()` and `write()` methods. The stream should be in non-blocking mode and `readinto()` should return `None` if there is no data available for reading.

After calling this function all terminal output is repeated on this stream, and any input that is available on the stream is passed on to the terminal input.

The *index* parameter should be a non-negative integer and specifies which duplication slot is set. A given port may implement more than one slot (slot 0 will always be available) and in that case terminal input and output is duplicated on all the slots that are set.

If `None` is passed as the *stream_object* then duplication is cancelled on the slot given by *index*.

The function returns the previous stream-like object in the given slot.

Filesystem mounting

Some ports provide a Virtual Filesystem (VFS) and the ability to mount multiple "real" filesystems within this VFS. Filesystem objects can be mounted at either the root of the VFS, or at a subdirectory that lives in the root. This allows

dynamic and flexible configuration of the filesystem that is seen by Python programs. Ports that have this functionality provide the *mount()* and *umount()* functions, and possibly various filesystem implementations represented by VFS classes.

uos.**mount**(*fsobj*, *mount_point*, *, *readonly*)
Mount the filesystem object *fsobj* at the location in the VFS given by the *mount_point* string. *fsobj* can be a a VFS object that has a `mount()` method, or a block device. If it's a block device then the filesystem type is automatically detected (an exception is raised if no filesystem was recognised). *mount_point* may be `'/'` to mount *fsobj* at the root, or `'/<name>'` to mount it at a subdirectory under the root.

If *readonly* is `True` then the filesystem is mounted read-only.

During the mount process the method `mount()` is called on the filesystem object.

Will raise `OSError(EPERM)` if *mount_point* is already mounted.

uos.**umount**(*mount_point*)
Unmount a filesystem. *mount_point* can be a string naming the mount location, or a previously-mounted filesystem object. During the unmount process the method `umount()` is called on the filesystem object.

Will raise `OSError(EINVAL)` if *mount_point* is not found.

class uos.**VfsFat**(*block_dev*)
Create a filesystem object that uses the FAT filesystem format. Storage of the FAT filesystem is provided by *block_dev*. Objects created by this constructor can be mounted using *mount()*.

static mkfs(*block_dev*)
Build a FAT filesystem on *block_dev*.

Block devices

A block device is an object which implements the block protocol, which is a set of methods described below by the *AbstractBlockDev* class. A concrete implementation of this class will usually allow access to the memory-like functionality a piece of hardware (like flash memory). A block device can be used by a particular filesystem driver to store the data for its filesystem.

class uos.**AbstractBlockDev**(...)
Construct a block device object. The parameters to the constructor are dependent on the specific block device.

readblocks(*block_num*, *buf*)
Starting at *block_num*, read blocks from the device into *buf* (an array of bytes). The number of blocks to read is given by the length of *buf*, which will be a multiple of the block size.

writeblocks(*block_num*, *buf*)
Starting at *block_num*, write blocks from *buf* (an array of bytes) to the device. The number of blocks to write is given by the length of *buf*, which will be a multiple of the block size.

ioctl(*op*, *arg*)
Control the block device and query its parameters. The operation to perform is given by *op* which is one of the following integers:

- 1 – initialise the device (*arg* is unused)
- 2 – shutdown the device (*arg* is unused)
- 3 – sync the device (*arg* is unused)
- 4 – get a count of the number of blocks, should return an integer (*arg* is unused)
- 5 – get the number of bytes in a block, should return an integer, or `None` in which case the default value of 512 is used (*arg* is unused)

By way of example, the following class will implement a block device that stores its data in RAM using a `bytearray`:

```
class RAMBlockDev:
    def __init__(self, block_size, num_blocks):
        self.block_size = block_size
        self.data = bytearray(block_size * num_blocks)

    def readblocks(self, block_num, buf):
        for i in range(len(buf)):
            buf[i] = self.data[block_num * self.block_size + i]

    def writeblocks(self, block_num, buf):
        for i in range(len(buf)):
            self.data[block_num * self.block_size + i] = buf[i]

    def ioctl(self, op, arg):
        if op == 4: # get number of blocks
            return len(self.data) // self.block_size
        if op == 5: # get block size
            return self.block_size
```

It can be used as follows:

```
import uos

bdev = RAMBlockDev(512, 50)
uos.VfsFat.mkfs(bdev)
vfs = uos.VfsFat(bdev)
uos.mount(vfs, '/ramdisk')
```

4.1.8 `ure` – simple regular expressions

This module implements a subset of the corresponding `CPython` module, as described below. For more information, refer to the original CPython documentation: `re`.

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, including negated sets (e.g. `[^a-c]`).

`'^'`

`'$'`

`'?'`

`'*'`

`'+'`

`'??'`

`'*?'`

`'+?'`

`'|'`

`'(...)'` Grouping. Each group is capturing (a substring it captures can be accessed with *match.group()* method).

NOT SUPPORTED: Counted repetitions (`{m,n}`), more advanced assertions (`\b`, `\B`), named groups (`(?P<name>...)`), non-capturing groups (`(?:...)`), etc.

Functions

`ure.`**`compile`**`(`*regex_str*`[,` *flags*`])`
Compile regular expression, return *regex* object.

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

`ure.`**`search`**`(`*regex_str*`,` *string*`)`
Compile *regex_str* and search it 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. (Availability depends on *MicroPython port*.)

Regex objects

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

`regex.`**`match`**`(`*string*`)`
`regex.`**`search`**`(`*string*`)`
Similar to the module-level functions *match()* and *search()*. Using methods is (much) more efficient if the same regex is applied to multiple strings.

`regex.`**`split`**`(`*string*`,` *max_split=-1*`)`
Split a *string* using regex. If *max_split* is given, it specifies maximum number of splits to perform. Returns list of strings (there may be up to *max_split+1* elements if it's specified).

Match objects

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

`match.`**`group`**`(`*[index]*`)`
Return matching (sub)string. *index* is 0 for entire match, 1 and above for each capturing group. Only numeric groups are supported.

4.1.9 `uselect` – wait for events on a set of streams

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `select`.

This module provides functions to efficiently wait for events on multiple *streams* (select streams which are ready for operations).

Functions

`uselect.poll()`
Create an instance of the Poll class.

`uselect.select(rlist, wlist, xlist[, timeout])`
Wait for activity on a set of objects.

This function is provided by some MicroPython ports for compatibility and is not efficient. Usage of `Poll` is recommended instead.

class Poll

Methods

`poll.register(obj[, eventmask])`
Register *stream* *obj* for polling. *eventmask* is logical OR of:

- `uselect.POLLIN` - data available for reading
- `uselect.POLLOUT` - more data can be written

Note that flags like `uselect.POLLHUP` and `uselect.POLLERR` are *not* valid as input eventmask (these are unsolicited events which will be returned from *poll()* regardless of whether they are asked for). This semantics is per POSIX.

eventmask defaults to `uselect.POLLIN | uselect.POLLOUT`.

`poll.unregister(obj)`
Unregister *obj* from polling.

`poll.modify(obj, eventmask)`
Modify the *eventmask* for *obj*.

`poll.poll(timeout=-1)`
Wait for at least one of the registered objects to become ready or have an exceptional condition, with optional timeout in milliseconds (if *timeout* arg is not specified or -1, there is no timeout).

Returns list of (`obj`, `event`, ...) tuples. There may be other elements in tuple, depending on a platform and version, so don't assume that its size is 2. The `event` element specifies which events happened with a stream and is a combination of `uselect.POLL*` constants described above. Note that flags `uselect.POLLHUP` and `uselect.POLLERR` can be returned at any time (even if were not asked for), and must be acted on accordingly (the corresponding stream unregistered from poll and likely closed), because otherwise all further invocations of *poll()* may return immediately with these flags set for this stream again.

In case of timeout, an empty list is returned.

Difference to CPython

Tuples returned may contain more than 2 elements as described above.

`poll.ipoll(timeout=-1, flags=0)`
Like *poll.poll()*, but instead returns an iterator which yields a *callee-owned tuple*. This function provides an efficient, allocation-free way to poll on streams.

If *flags* is 1, one-shot behavior for events is employed: streams for which events happened will have their event masks automatically reset (equivalent to `poll.modify(obj, 0)`), so new events for such a stream won't

be processed until new mask is set with *poll.modify()*. This behavior is useful for asynchronous I/O schedulers.

Difference to CPython

This function is a MicroPython extension.

4.1.10 usocket – socket module

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: socket.

This module provides access to the BSD socket interface.

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 native socket address format of the usocket module is an opaque data type returned by *getaddrinfo* function, which must be used to resolve textual address (including numeric addresses):

```
sockaddr = usocket.getaddrinfo('www.micropython.org', 80)[0][-1]
# You must use getaddrinfo() even for numeric addresses
sockaddr = usocket.getaddrinfo('127.0.0.1', 80)[0][-1]
# Now you can use that address
sock.connect(addr)
```

Using *getaddrinfo* is the most efficient (both in terms of memory and processing power) and portable way to work with addresses.

However, socket module (note the difference with native MicroPython usocket module described here) provides CPython-compatible way to specify addresses using tuples, as described below. Note that depending on a *MicroPython port*, socket module can be builtin or need to be installed from *micropython-lib* (as in the case of *MicroPython Unix port*), and some ports still accept only numeric addresses in the tuple format, and require to use *getaddrinfo* function to resolve domain names.

Summing up:

- Always use *getaddrinfo* when writing portable applications.
- Tuple addresses described below can be used as a shortcut for quick hacks and interactive use, if your port supports them.

Tuple address format for socket module:

- IPv4: *(ipv4_address, port)*, where *ipv4_address* is a string with dot-notation numeric IPv4 address, e.g. "8.8.8.8", and *port* is and integer port number in the range 1-65535. Note the domain names are not accepted as *ipv4_address*, they should be resolved first using *usocket.getaddrinfo()*.
- IPv6: *(ipv6_address, port, flowinfo, scopeid)*, where *ipv6_address* is a string with colon-notation numeric IPv6 address, e.g. "2001:db8::1", and *port* is an integer port number in the range 1-65535. *flowinfo* must be

0. *scopeid* is the interface scope identifier for link-local addresses. Note the domain names are not accepted as *ipv6_address*, they should be resolved first using *usocket.getaddrinfo()*. Availability of IPv6 support depends on a *MicroPython port*.

Functions

usocket.**socket**(*af=AF_INET*, *type=SOCK_STREAM*, *proto=IPPROTO_TCP*)

Create a new socket using the given address family, socket type and protocol number. Note that specifying *proto* in most cases is not required (and not recommended, as some MicroPython ports may omit `IPPROTO_*` constants). Instead, *type* argument will select needed protocol automatically:

```
# Create STREAM TCP socket
socket(AF_INET, SOCK_STREAM)
# Create DGRAM UDP socket
socket(AF_INET, SOCK_DGRAM)
```

usocket.**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 = usocket.socket()
s.connect(usocket.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 the `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.

---

usocket.**inet_ntop**(*af*, *bin_addr*)

Convert a binary network address *bin_addr* of the given address family *af* to a textual representation:

```
>>> usocket.inet_ntop(usocket.AF_INET, b"\x7f\0\0\1")
'127.0.0.1'
```

usocket.**inet_pton**(*af*, *txt_addr*)

Convert a textual network address *txt_addr* of the given address family *af* to a binary representation:

```
>>> usocket.inet_pton(usocket.AF_INET, "1.2.3.4")
b'\x01\x02\x03\x04'
```

Constants

usocket.**AF_INET**
usocket.**AF_INET6**

Address family types. Availability depends on a particular *MicroPython port*.

usocket.**SOCK_STREAM**
usocket.**SOCK_DGRAM**
Socket types.

usocket.**IPPROTO_UDP**
usocket.**IPPROTO_TCP**
IP protocol numbers. Availability depends on a particular *MicroPython port*. Note that you don't need to specify these in a call to *usocket.socket()*, because *SOCK_STREAM* socket type automatically selects *IPPROTO_TCP*, and *SOCK_DGRAM* - *IPPROTO_UDP*. Thus, the only real use of these constants is as an argument to *setsockopt()*.

usocket.SOL_*
Socket option levels (an argument to *setsockopt()*). The exact inventory depends on a *MicroPython port*.

usocket.SO_*
Socket options (an argument to *setsockopt()*). The exact inventory depends on a *MicroPython port*.

Constants specific to WiPy:

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

class socket

Methods

socket.**close**()
Mark the socket closed and release all resources. Once that happens, all future operations on the socket object will fail. The remote end will receive EOF indication if supported by protocol.

Sockets are automatically closed when they are garbage-collected, but it is recommended to *close()* them explicitly as soon you finished working with 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*)
Note: Not every port supports this method, see below.

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.

Not every *MicroPython port* supports this method. A more portable and generic solution is to use *uselect.poll* object. This allows to wait on multiple objects at the same time (and not just on sockets, but on generic *stream* objects which support polling). Example:

```
# Instead of:
s.settimeout(1.0)  # time in seconds
s.read(10)  # may timeout

# Use:
poller = uselect.poll()
poller.register(s, uselect.POLLIN)
res = poller.poll(1000)  # time in milliseconds
if not res:
    # s is still not ready for input, i.e. operation timed out
```

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.

exception usocket.**error**
MicroPython does NOT have this exception.

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.

4.1.11 ussl – SSL/TLS module

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: `ssl`.

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

Functions

ussl.**wrap_socket**(*sock*, *server_side=False*, *keyfile=None*, *certfile=None*, *cert_reqs=CERT_NONE*, *ca_certs=None*)

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.

Depending on the underlying module implementation in a particular *MicroPython port*, some or all keyword arguments above may be not supported.

Warning: Some implementations of ussl module do NOT validate server certificates, which makes an SSL connection established prone to man-in-the-middle attacks.

Exceptions

ssl.**SSLError**

This exception does NOT exist. Instead its base class, OSError, is used.

Constants

ussl.**CERT_NONE**
ussl.**CERT_OPTIONAL**
ussl.**CERT_REQUIRED**

Supported values for *cert_reqs* parameter.

4.1.12 utime – time related functions

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: time.

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. Some boards may accept *seconds* as a floating-point number to sleep for a fractional number of seconds. Note that other boards may not accept a 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.

The wrap-around 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, as a signed value which may wrap around.

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* occurred earlier in time than *ticks2*. Otherwise, it means that *ticks1* occurred 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(scheduled_time, now) > 0:
    print("Too early, let's nap")
    sleep_ms(ticks_diff(scheduled_time, now))
    task.run()
elif ticks_diff(scheduled_time, now) == 0:
    print("Right at time!")
    task.run()
elif ticks_diff(scheduled_time, now) < 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).

4.2 MicroPython-specific libraries

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

4.2.1 btree – simple BTree database

The btree module implements a simple key-value database using external storage (disk files, or in general case, a random-access *stream*). Keys are stored sorted in the database, and besides efficient retrieval by a key value, a database also supports efficient ordered range scans (retrieval of values with the keys in a given range). On the application interface side, BTree database work as close a possible to a way standard *dict* type works, one notable difference is that both keys and values must be *bytes* objects (so, if you want to store objects of other types, you need to serialize them to *bytes* first).

The module is based on the well-known BerkelyDB library, version 1.xx.

Example:

```
import btree

# First, we need to open a stream which holds a database
# This is usually a file, but can be in-memory database
# using uio.BytesIO, a raw flash partition, etc.
# Oftentimes, you want to create a database file if it doesn't
# exist and open if it exists. Idiom below takes care of this.
# DO NOT open database with "a+b" access mode.
try:
    f = open("mydb", "r+b")
except OSError:
    f = open("mydb", "w+b")

# Now open a database itself
db = btree.open(f)

# The keys you add will be sorted internally in the database
db[b"3"] = b"three"
db[b"1"] = b"one"
db[b"2"] = b"two"

# Assume that any changes are cached in memory unless
# explicitly flushed (or database closed). Flush database
# at the end of each "transaction".
db.flush()

# Prints b'two'
print(db[b"2"])

# Iterate over sorted keys in the database, starting from b"2"
# until the end of the database, returning only values.
# Mind that arguments passed to values() method are *key* values.
# Prints:
#   b'two'
#   b'three'
for word in db.values(b"2"):
    print(word)

del db[b"2"]

# No longer true, prints False
print(b"2" in db)

# Prints:
#  b"1"
#  b"3"
for key in db:
    print(key)

db.close()

# Don't forget to close the underlying stream!
f.close()
```

Functions

btree.**open**(*stream*, *, *flags=0*, *pagesize=0*, *cachesize=0*, *minkeypage=0*)

Open a database from a random-access *stream* (like an open file). All other parameters are optional and keyword-only, and allow to tweak advanced parameters of the database operation (most users will not need them):

- *flags* - Currently unused.
- *pagesize* - Page size used for the nodes in BTree. Acceptable range is 512-65536. If 0, a port-specific default will be used, optimized for port's memory usage and/or performance.
- *cachesize* - Suggested memory cache size in bytes. For a board with enough memory using larger values may improve performance. Cache policy is as follows: entire cache is not allocated at once; instead, accessing a new page in database will allocate a memory buffer for it, until value specified by *cachesize* is reached. Then, these buffers will be managed using LRU (least recently used) policy. More buffers may still be allocated if needed (e.g., if a database contains big keys and/or values). Allocated cache buffers aren't reclaimed.
- *minkeypage* - Minimum number of keys to store per page. Default value of 0 equivalent to 2.

Returns a BTree object, which implements a dictionary protocol (set of methods), and some additional methods described below.

Methods

btree.**close**()

Close the database. It's mandatory to close the database at the end of processing, as some unwritten data may be still in the cache. Note that this does not close underlying stream with which the database was opened, it should be closed separately (which is also mandatory to make sure that data flushed from buffer to the underlying storage).

btree.**flush**()

Flush any data in cache to the underlying stream.

btree.**__getitem__**(*key*)
btree.**get**(*key*, *default=None*)
btree.**__setitem__**(*key*, *val*)
btree.**__delitem__**(*key*)
btree.**__contains__**(*key*)

Standard dictionary methods.

btree.**__iter__**()

A BTree object can be iterated over directly (similar to a dictionary) to get access to all keys in order.

btree.**keys**([*start_key*[, *end_key*[, *flags*]]])
btree.**values**([*start_key*[, *end_key*[, *flags*]]])
btree.**items**([*start_key*[, *end_key*[, *flags*]]])

These methods are similar to standard dictionary methods, but also can take optional parameters to iterate over a key sub-range, instead of the entire database. Note that for all 3 methods, *start_key* and *end_key* arguments represent key values. For example, *values()* method will iterate over values corresponding to they key range given. None values for *start_key* means "from the first key", no *end_key* or its value of None means "until the end of database". By default, range is inclusive of *start_key* and exclusive of *end_key*, you can include *end_key* in iteration by passing *flags* of *btree.INCL*. You can iterate in descending key direction by passing *flags* of *btree.DESC*. The flags values can be ORed together.

Constants

btree.**INCL**
A flag for *keys()*, *values()*, *items()* methods to specify that scanning should be inclusive of the end key.

btree.**DESC**
A flag for *keys()*, *values()*, *items()* methods to specify that scanning should be in descending direction of keys.

4.2.2 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; permissible values are listed under Constants below. These set the number of bits used to encode a color value and the layout of these bits in *buffer*. 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 instances utilising different formats, but the resulting colors may be unexpected due to the mismatch in color formats.

Constants

`framebuf.`**`MONO_VLSB`**
Monochrome (1-bit) color format This defines a mapping where the bits in a byte are vertically mapped with bit 0 being nearest the top of the screen. Consequently each byte occupies 8 vertical pixels. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered at locations starting at the leftmost edge, 8 pixels lower.

`framebuf.`**`MONO_HLSB`**
Monochrome (1-bit) color format This defines a mapping where the bits in a byte are horizontally mapped. Each

byte occupies 8 horizontal pixels with bit 0 being the leftmost. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered on the next row, one pixel lower.

framebuf.**MONO_HMSB**
Monochrome (1-bit) color format This defines a mapping where the bits in a byte are horizontally mapped. Each byte occupies 8 horizontal pixels with bit 7 being the leftmost. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered on the next row, one pixel lower.

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

framebuf.**GS2_HMSB**
Grayscale (2-bit) color format

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

framebuf.**GS8**
Grayscale (8-bit) color format

4.2.3 machine — functions related to the hardware

The machine module contains specific functions related to the hardware on a particular board. Most functions in this module allow to achieve direct and unrestricted access to and control of hardware blocks on a system (like CPU, timers, buses, etc.). Used incorrectly, this can lead to malfunction, lockups, crashes of your board, and in extreme cases, hardware damage. A note of callbacks used by functions and class methods of *machine* module: all these callbacks should be considered as executing in an interrupt context. This is true for both physical devices with IDs >= 0 and "virtual" devices with negative IDs like -1 (these "virtual" devices are still thin shims on top of real hardware and real hardware interrupts). See *Writing interrupt handlers*.

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.

machine.**wake_reason**()
: Get the wake reason. See *constants* for the possible return values.

Miscellaneous functions

machine.**rng**()
: Return a 24-bit software generated random number.

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 Pin – control I/O pins

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

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

Usage Model:

```
from machine import Pin

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

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

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

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

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

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

Constructors

class machine.**Pin**(*id*, *mode=-1*, *pull=-1*, ***, *value*, *drive*, *alt*)
Access the pin peripheral (GPIO pin) associated with the given `id`. If additional arguments are given in the constructor then they are used to initialise the pin. Any settings that are not specified will remain in their previous state.

The arguments are:

- `id` is mandatory and can be an arbitrary object. Among possible value types are: int (an internal Pin identifier), str (a Pin name), and tuple (pair of [port, pin]).
- `mode` specifies the pin mode, which can be one of:
  - `Pin.IN` - Pin is configured for input. If viewed as an output the pin is in high-impedance state.
  - `Pin.OUT` - Pin is configured for (normal) output.

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

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

Methods

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

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

Returns `None`.

`Pin.`**`value`**([*x*])

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

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

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

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

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

When setting the value this method returns `None`.

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

`Pin.on`()
Set pin to "1" output level.

`Pin.off`()
Set pin to "0" output level.

`Pin.mode`([*mode*])
Get or set the pin mode. See the constructor documentation for details of the `mode` argument.

`Pin.pull`([*pull*])
Get or set the pin pull state. See the constructor documentation for details of the `pull` argument.

`Pin.drive`([*drive*])
Get or set the pin drive strength. See the constructor documentation for details of the `drive` argument.

Not all ports implement this method.

Availability: WiPy.

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

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 UART – duplex serial communication bus

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

UART objects can be created and initialised using:

```
from machine import UART

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

Supported parameters differ on a board:

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

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

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

```
uart.read(10)       # read 10 characters, returns a bytes object
uart.read()         # read all available characters
uart.readline()     # read a line
```

```
uart.readinto(buf)  # read and store into the given buffer
uart.write('abc')   # write the 3 characters
```

Constructors

class machine.**UART**(*id*, ...)
Construct a UART object of the given id.

Methods

UART.**init**(*baudrate=9600*, *bits=8*, *parity=None*, *stop=1*, ***, *pins=(TX*, *RX*, *RTS*, *CTS)*)
Initialise the UART bus with the given parameters:

- `baudrate` is the clock rate.
- `bits` is the number of bits per character, 7, 8 or 9.
- `parity` is the parity, `None`, 0 (even) or 1 (odd).
- `stop` is the number of stop bits, 1 or 2.
- `pins` is a 4 or 2 item list indicating the TX, RX, RTS and CTS pins (in that order). Any of the pins can be None if one wants the UART to operate with limited functionality. If the RTS pin is given the the RX pin must be given as well. The same applies to CTS. When no pins are given, then the default set of TX and RX pins is taken, and hardware flow control will be disabled. If pins=None, no pin assignment will be made.

UART.**deinit**()
Turn off the UART bus.

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

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

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

UART.**read**([*nbytes*])
Read characters. If `nbytes` is specified then read at most that many bytes, otherwise read as much data as possible.

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

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

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

UART.**readline**()
Read a line, ending in a newline character.

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

UART.**write**(*buf*)
Write the buffer of bytes to the bus.

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

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

UART.**irq**(*trigger*, *priority=1*, *handler=None*, *wake=machine.IDLE*)
Create a callback to be triggered when data is received on the UART.

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

Note: The handler will be called whenever any of the following two conditions are met:

- 8 new characters have been received.
- At least 1 new character is waiting in the Rx buffer and the Rx line has been silent for the duration of 1 complete frame.

This means that when the handler function is called there will be between 1 to 8 characters waiting.

Returns an irq object.

Constants

UART.**RX_ANY**
IRQ trigger sources

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 I2C – a two-wire serial protocol

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

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

Printing the I2C object gives you information about its configuration.

Example usage:

```
from machine import I2C

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

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

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

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

Constructors

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

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

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

General Methods

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

Initialise the I2C bus with the given arguments:

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

I2C.**deinit**()

Turn off the I2C bus.

Availability: WiPy.

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

Primitive I2C operations

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

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).

The method returns `None`.

I2C.**writeto_mem**(*addr*, *memaddr*, *buf*, ***, *addrsize=8*)
Write *buf* to the slave specified by *addr* starting from the memory address specified by *memaddr*. The argument *addrsize* specifies the address size in bits (on ESP8266 this argument is not recognised and the address size is always 8 bits).

The method returns `None`.

class RTC – real time clock

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

Example usage:

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

Constructors

class machine.**RTC**(*id=0*, *...*)
Create an RTC object. See init for parameters of initialization.

Methods

RTC.**init**(*datetime*)
Initialise the RTC. Datetime is a tuple of the form:

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

RTC.**now**()
Get get the current datetime tuple.

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

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

RTC.**alarm_left**(*alarm_id=0*)
Get the number of milliseconds left before the alarm expires.

RTC.**cancel**(*alarm_id=0*)
Cancel a running alarm.

RTC.**irq**(*, *trigger*, *handler=None*, *wake=machine.IDLE*)
Create an irq object triggered by a real time clock alarm.

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

Constants

RTC.**ALARM0**
irq trigger source

class Timer – control hardware timers

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

See discussion of *important constraints* on Timer callbacks.

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

Constructors

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

Methods

Timer.**init**(*mode*, *, *width=16*)
Initialise the timer. Example:

```
tim.init(Timer.PERIODIC)              # periodic 16-bit timer
tim.init(Timer.ONE_SHOT, width=32)    # one shot 32-bit timer
```

Keyword arguments:

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

- `width` must be either 16 or 32 (bits). For really low frequencies < 5Hz (or large periods), 32-bit timers should be used. 32-bit mode is only available for `ONE_SHOT` AND `PERIODIC` modes.

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

`Timer.`**`channel`**`(`*channel*, ****, *freq*, *period*, *polarity=Timer.POSITIVE*, *duty_cycle=0*`)`
If only a channel identifier passed, then a previously initialized channel object is returned (or `None` if there is no previous channel).

Otherwise, a TimerChannel object is initialized and returned.

The operating mode is is the one configured to the Timer object that was used to create the channel.

- `channel` if the width of the timer is 16-bit, then must be either `TIMER.A`, `TIMER.B`. If the width is 32-bit then it **must be** `TIMER.A | TIMER.B`.

Keyword only arguments:

- `freq` sets the frequency in Hz.
- `period` sets the period in microseconds.

Note: Either `freq` or `period` must be given, never both.

- `polarity` this is applicable for `PWM`, and defines the polarity of the duty cycle
- `duty_cycle` only applicable to `PWM`. It's a percentage (0.00-100.00). Since the WiPy doesn't support floating point numbers the duty cycle must be specified in the range 0-10000, where 10000 would represent 100.00, 5050 represents 50.50, and so on.

Note: When the channel is in PWM mode, the corresponding pin is assigned automatically, therefore there's no need to assign the alternate function of the pin via the `Pin` class. The pins which support PWM functionality are the following:

- `GP24` on Timer 0 channel A.
- `GP25` on Timer 1 channel A.
- `GP9` on Timer 2 channel B.
- `GP10` on Timer 3 channel A.
- `GP11` on Timer 3 channel B.

`timerchannel.`**`freq`**`(`[*value*]`)`
Get or set the timer channel frequency (in Hz).

`timerchannel.`**`period`**`(`[*value*]`)`
Get or set the timer channel period (in microseconds).

`timerchannel.`**`duty_cycle`**`(`[*value*]`)`
Get or set the duty cycle of the PWM signal. It's a percentage (0.00-100.00). Since the WiPy doesn't support floating point numbers the duty cycle must be specified in the range 0-10000, where 10000 would represent 100.00, 5050 represents 50.50, and so on.

Constants

`Timer.`**`ONE_SHOT`**

`Timer.`**`PERIODIC`**
Timer operating mode.

class TimerChannel — setup a channel for a timer

Timer channels are used to generate/capture a signal using a timer.

TimerChannel objects are created using the Timer.channel() method.

Methods

`timerchannel.`**`irq`**(*, *trigger*, *priority=1*, *handler=None*)
The behavior of this callback is heavily dependent on the operating mode of the timer channel:

- If mode is `Timer.PERIODIC` the callback is executed periodically with the configured frequency or period.
- If mode is `Timer.ONE_SHOT` the callback is executed once when the configured timer expires.
- If mode is `Timer.PWM` the callback is executed when reaching the duty cycle value.

The accepted params are:

- `priority` level of the interrupt. Can take values in the range 1-7. Higher values represent higher priorities.
- `handler` is an optional function to be called when the interrupt is triggered.
- `trigger` must be `Timer.TIMEOUT` when the operating mode is either `Timer.PERIODIC` or `Timer.ONE_SHOT`. In the case that mode is `Timer.PWM` then trigger must be equal to `Timer.MATCH`.

Returns a callback object.

class WDT – watchdog timer

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

Example usage:

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

Availability of this class: pyboard, WiPy.

Constructors

`class` `machine.`**`WDT`**(*id=0*, *timeout=5000*)
Create a WDT object and start it. The timeout must be given in seconds and the minimum value that is accepted is 1 second. Once it is running the timeout cannot be changed and the WDT cannot be stopped either.

Methods

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

class ADC – analog to digital conversion

Usage:

```
import machine

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

Constructors

class machine.**ADC**(*id=0*, ***, *bits=12*)
Create an ADC object associated with the given pin. This allows you to then read analog values on that pin. For more info check the pinout and alternate functions table.

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

Methods

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

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

ADC.**init**()
Enable the ADC block.

ADC.**deinit**()
Disable the ADC block.

class ADCChannel — read analog values from internal or external sources

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

machine.**adcchannel**()
Fast method to read the channel value.

adcchannel.**value**()
Read the channel value.

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

adcchannel.**deinit**()
Disable the ADC channel.

class SD – secure digital memory card

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

Example usage:

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

Constructors

class machine.**SD**(*id*, ...)
Create a SD card object. See `init()` for parameters if initialization.

Methods

SD.**init**(*id=0*, *pins=('GP10', 'GP11', 'GP15')*)
Enable the SD card. In order to initialize the card, give it a 3-tuple: `(clk_pin, cmd_pin, dat0_pin)`.

SD.**deinit**()
Disable the SD card.

4.2.4 micropython – access and control MicroPython internals

Functions

micropython.**const**(*expr*)
Used to declare that the expression is a constant so that the compile can optimise it. The use of this function should be as follows:

```
from micropython import const

CONST_X = const(123)
CONST_Y = const(2 * CONST_X + 1)
```

Constants declared this way are still accessible as global variables from outside the module they are declared in. On the other hand, if a constant begins with an underscore then it is hidden, it is not available as a global variable, and does not take up any memory during execution.

This *const* function is recognised directly by the MicroPython parser and is provided as part of the *micropython* module mainly so that scripts can be written which run under both CPython and MicroPython, by following the above pattern.

micropython.**opt_level**([*level*])

If *level* is given then this function sets the optimisation level for subsequent compilation of scripts, and returns `None`. Otherwise it returns the current optimisation level.

The optimisation level controls the following compilation features:

- Assertions: at level 0 assertion statements are enabled and compiled into the bytecode; at levels 1 and higher assertions are not compiled.
- Built-in `__debug__` variable: at level 0 this variable expands to `True`; at levels 1 and higher it expands to `False`.
- Source-code line numbers: at levels 0, 1 and 2 source-code line number are stored along with the bytecode so that exceptions can report the line number they occurred at; at levels 3 and higher line numbers are not stored.

The default optimisation level is usually level 0.

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.

micropython.**mem_info**([*verbose*])

Print information about currently used memory. If the *verbose* argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the amount of stack and heap used. In verbose mode it prints out the entire heap indicating which blocks are used and which are free.

micropython.**qstr_info**([*verbose*])

Print information about currently interned strings. If the *verbose* argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the number of interned strings and the amount of RAM they use. In verbose mode it prints out the names of all RAM-interned strings.

micropython.**stack_use**()

Return an integer representing the current amount of stack that is being used. The absolute value of this is not particularly useful, rather it should be used to compute differences in stack usage at different points.

micropython.**heap_lock**()

micropython.**heap_unlock**()

Lock or unlock the heap. When locked no memory allocation can occur and a *MemoryError* will be raised if any heap allocation is attempted.

These functions can be nested, ie *heap_lock()* can be called multiple times in a row and the lock-depth will increase, and then *heap_unlock()* must be called the same number of times to make the heap available again.

micropython.**kbd_intr**(*chr*)
Set the character that will raise a *KeyboardInterrupt* exception. By default this is set to 3 during script execution, corresponding to Ctrl-C. Passing -1 to this function will disable capture of Ctrl-C, and passing 3 will restore it.

This function can be used to prevent the capturing of Ctrl-C on the incoming stream of characters that is usually used for the REPL, in case that stream is used for other purposes.

micropython.**schedule**(*func*, *arg*)
Schedule the function *func* to be executed "very soon". The function is passed the value *arg* as its single argument. "Very soon" means that the MicroPython runtime will do its best to execute the function at the earliest possible time, given that it is also trying to be efficient, and that the following conditions hold:

- A scheduled function will never preempt another scheduled function.
- Scheduled functions are always executed "between opcodes" which means that all fundamental Python operations (such as appending to a list) are guaranteed to be atomic.
- A given port may define "critical regions" within which scheduled functions will never be executed. Functions may be scheduled within a critical region but they will not be executed until that region is exited. An example of a critical region is a preempting interrupt handler (an IRQ).

A use for this function is to schedule a callback from a preempting IRQ. Such an IRQ puts restrictions on the code that runs in the IRQ (for example the heap may be locked) and scheduling a function to call later will lift those restrictions.

Note: If *schedule()* is called from a preempting IRQ, when memory allocation is not allowed and the callback to be passed to *schedule()* is a bound method, passing this directly will fail. This is because creating a reference to a bound method causes memory allocation. A solution is to create a reference to the method in the class constructor and to pass that reference to *schedule()*. This is discussed in detail here *reference documentation* under "Creation of Python objects".

There is a finite stack to hold the scheduled functions and *schedule()* will raise a *RuntimeError* if the stack is full.

4.2.5 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 *usocket* module.

For example:

```
# connect/ show IP config a specific network interface
# see below for examples of specific drivers
import network
import utime
nic = network.Driver(...)
if not nic.isconnected():
    nic.connect()
    print("Waiting for connection...")
    while not nic.isconnected():
        utime.sleep(1)
print(nic.ifconfig())

# now use usocket as usual
import usocket as 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()
```

Common network adapter interface

This section describes an (implied) abstract base class for all network interface classes implemented by *MicroPython ports* for different hardware. This means that MicroPython does not actually provide `AbstractNIC` class, but any actual NIC class, as described in the following sections, implements methods as described here.

class network.**AbstractNIC**(*id=None*, ...)

Instantiate a network interface object. Parameters are network interface dependent. If there are more than one interface of the same type, the first parameter should be *id*.

network.**active**([*is_active*])
: Activate ("up") or deactivate ("down") the network interface, if a boolean argument is passed. Otherwise, query current state if no argument is provided. Most other methods require an active interface (behavior of calling them on inactive interface is undefined).

network.**connect**([*service_id*, *key=None*, *, ...])
: Connect the interface to a network. This method is optional, and available only for interfaces which are not "always connected". If no parameters are given, connect to the default (or the only) service. If a single parameter is given, it is the primary identifier of a service to connect to. It may be accompanied by a key (password) required to access said service. There can be further arbitrary keyword-only parameters, depending on the networking medium type and/or particular device. Parameters can be used to: a) specify alternative service identifer types; b) provide additional connection parameters. For various medium types, there are different sets of predefined/recommended parameters, among them:

  - WiFi: *bssid* keyword to connect to a specific BSSID (MAC address)

network.**disconnect**()
: Disconnect from network.

network.**isconnected**()
: Returns `True` if connected to network, otherwise returns `False`.

network.**scan**(*, ...)
: Scan for the available network services/connections. Returns a list of tuples with discovered service parameters. For various network media, there are different variants of predefined/ recommended tuple formats, among them:

  - WiFi: (ssid, bssid, channel, RSSI, authmode, hidden). There may be further fields, specific to a particular device.

  The function may accept additional keyword arguments to filter scan results (e.g. scan for a particular service, on a particular channel, for services of a particular set, etc.), and to affect scan duration and other parameters. Where possible, parameter names should match those in connect().

network.**status**([*param*])
: Query dynamic status information of the interface. When called with no argument the return value describes the network link status. Otherwise *param* should be a string naming the particular status parameter to retrieve.

The return types and values are dependent on the network medium/technology. Some of the parameters that may be supported are:

- WiFi STA: use `'rssi'` to retrieve the RSSI of the AP signal
- WiFi AP: use `'stations'` to retrieve a list of all the STAs connected to the AP. The list contains tuples of the form (MAC, RSSI).

`network.`**`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'))
```

`network.`**`config`**`(`*'param'*`)`

`network.`**`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 *`ifconfig()`*). These include network-specific and hardware-specific parameters. For setting parameters, the keyword argument syntax should be used, and multiple parameters can be set at once. For querying, a parameter name should be quoted as a string, and only one parameter can be queried at a 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'))
```

class WLAN

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

```
import network
import time
# setup as a station
wlan = network.WLAN(mode=WLAN.STA)
wlan.connect('your-ssid', auth=(WLAN.WPA2, 'your-key'))
while not wlan.isconnected():
    time.sleep_ms(50)
print(wlan.ifconfig())

# now use socket as usual
...
```

Constructors

`class` `network.`**`WLAN`**`(`*id=0, ...*`)`

Create a WLAN object, and optionally configure it. See *`init()`* for params of configuration.

Note: The `WLAN` constructor is special in the sense that if no arguments besides the id are given, it will return the already existing `WLAN` instance without re-configuring it. This is because `WLAN` is a system feature of the WiPy. If the already existing instance is not initialized it will do the same as the other constructors an will initialize it with default values.

Methods

`wlan.`**`init`**(*mode*, *, *ssid*, *auth*, *channel*, *antenna*)

Set or get the WiFi network processor configuration.

Arguments are:

- *mode* can be either `WLAN.STA` or `WLAN.AP`.
- *ssid* is a string with the ssid name. Only needed when mode is `WLAN.AP`.
- *auth* is a tuple with (sec, key). Security can be `None`, `WLAN.WEP`, `WLAN.WPA` or `WLAN.WPA2`. The key is a string with the network password. If `sec` is `WLAN.WEP` the key must be a string representing hexadecimal values (e.g. 'ABC1DE45BF'). Only needed when mode is `WLAN.AP`.
- *channel* a number in the range 1-11. Only needed when mode is `WLAN.AP`.
- *antenna* selects between the internal and the external antenna. Can be either `WLAN.INT_ANT` or `WLAN.EXT_ANT`.

For example, you can do:

```
# create and configure as an access point
wlan.init(mode=WLAN.AP, ssid='wipy-wlan', auth=(WLAN.WPA2,'www.wipy.io'),␣
↪channel=7, antenna=WLAN.INT_ANT)
```

or:

```
# configure as an station
wlan.init(mode=WLAN.STA)
```

`wlan.`**`connect`**(*ssid*, *, *auth=None*, *bssid=None*, *timeout=None*)

Connect to a WiFi access point using the given SSID, and other security parameters.

- *auth* is a tuple with (sec, key). Security can be `None`, `WLAN.WEP`, `WLAN.WPA` or `WLAN.WPA2`. The key is a string with the network password. If `sec` is `WLAN.WEP` the key must be a string representing hexadecimal values (e.g. 'ABC1DE45BF').
- *bssid* is the MAC address of the AP to connect to. Useful when there are several APs with the same ssid.
- *timeout* is the maximum time in milliseconds to wait for the connection to succeed.

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

Performs a network scan and returns a list of named tuples with (ssid, bssid, sec, channel, rssi). Note that channel is always `None` since this info is not provided by the WiPy.

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

Disconnect from the WiFi access point.

`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, `False` otherwise.

`wlan.`**`ifconfig`**(*if_id=0, config=['dhcp' or configtuple]*)

With no parameters given returns a 4-tuple of *(ip, subnet_mask, gateway, DNS_server)*.

if `'dhcp'` is passed as a parameter then the DHCP client is enabled and the IP params are negotiated with the AP.

If the 4-tuple config is given then a static IP is configured. For instance:

```
wlan.ifconfig(config=('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
```

`wlan.`**`mode`**`(`[*mode*]`)`
Get or set the WLAN mode.

`wlan.`**`ssid`**`(`[*ssid*]`)`
Get or set the SSID when in AP mode.

`wlan.`**`auth`**`(`[*auth*]`)`
Get or set the authentication type when in AP mode.

`wlan.`**`channel`**`(`[*channel*]`)`
Get or set the channel (only applicable in AP mode).

`wlan.`**`antenna`**`(`[*antenna*]`)`
Get or set the antenna type (external or internal).

`wlan.`**`mac`**`(`[*mac_addr*]`)`
Get or set a 6-byte long bytes object with the MAC address.

`wlan.`**`irq`**`(*, handler, wake)`
Create a callback to be triggered when a WLAN event occurs during `machine.SLEEP` mode. Events are triggered by socket activity or by WLAN connection/disconnection.

- *handler* is the function that gets called when the IRQ is triggered.
- *wake* must be `machine.SLEEP`.

Returns an IRQ object.

Constants

`WLAN.`**`STA`**

`WLAN.`**`AP`**
selects the WLAN mode

`WLAN.`**`WEP`**

`WLAN.`**`WPA`**

`WLAN.`**`WPA2`**
selects the network security

`WLAN.`**`INT_ANT`**

`WLAN.`**`EXT_ANT`**
selects the antenna type

4.2.6 `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 then 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": offset | uctypes.UINT32
```

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

- Recursive structures:

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

  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 the structure it defines).

- Arrays of primitive types:

```
"arr": (offset | uctypes.ARRAY, size | uctypes.UINT8),
```

  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": (offset | uctypes.ARRAY, size, {"b": 0 | uctypes.UINT8}),
```

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

  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": offset | uctypes.BFUINT16 | lsbit << uctypes.BF_POS | bitsize << 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 extract the bitfield).

In the example above, first a 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 *lsbit* bit of this UINT16, and length is *bitsize* bits, will be extracted. For example, if *lsbit* is 0 and *bitsize* is 8, then effectively it 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 the 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).

4.3 Libraries specific to the WiPy

The following libraries are specific to the WiPy.

4.3.1 `wipy` – WiPy specific features

The `wipy` module contains functions to control specific features of the WiPy, such as the heartbeat LED.

Functions

`wipy.`**`heartbeat`**`(`[*enable*]`)`
Get or set the state (enabled or disabled) of the heartbeat LED. Accepts and returns boolean values (`True` or `False`).

CHAPTER

FIVE

THE MICROPYTHON LANGUAGE

MicroPython aims to implement the Python 3.4 standard (with selected features from later versions) with respect to language syntax, and most of the features of MicroPython are identical to those described by the "Language Reference" documentation at docs.python.org.

The MicroPython standard library is described in the *corresponding chapter*. The *MicroPython differences from CPython* chapter describes differences between MicroPython and CPython (which mostly concern standard library and types, but also some language-level features).

This chapter describes features and peculiarities of MicroPython implementation and the best practices to use them.

5.1 Glossary

baremetal A system without a (full-fledged) OS, for example an *MCU*-based system. When running on a baremetal system, MicroPython effectively becomes its user-facing OS with a command interpreter (REPL).

board A PCB board. Oftentimes, the term is used to denote a particular model of an *MCU* system. Sometimes, it is used to actually refer to *MicroPython port* to a particular board (and then may also refer to "boardless" ports like *Unix port*).

callee-owned tuple A tuple returned by some builtin function/method, containing data which is valid for a limited time, usually until next call to the same function (or a group of related functions). After next call, data in the tuple may be changed. This leads to the following restriction on the usage of callee-owned tuples - references to them cannot be stored. The only valid operation is extracting values from them (including making a copy). Callee-owned tuples is a MicroPython-specific construct (not available in the general Python language), introduced for memory allocation optimization. The idea is that callee-owned tuple is allocated once and stored on the callee side. Subsequent calls don't require allocation, allowing to return multiple values when allocation is not possible (e.g. in interrupt context) or not desirable (because allocation inherently leads to memory fragmentation). Note that callee-owned tuples are effectively mutable tuples, making an exception to Python's rule that tuples are immutable. (It may be interesting why tuples were used for such a purpose then, instead of mutable lists - the reason for that is that lists are mutable from user application side too, so a user could do things to a callee-owned list which the callee doesn't expect and could lead to problems; a tuple is protected from this.)

CPython CPython is the reference implementation of Python programming language, and the most well-known one, which most of the people run. It is however one of many implementations (among which Jython, IronPython, PyPy, and many more, including MicroPython). As there is no formal specification of the Python language, only CPython documentation, it is not always easy to draw a line between Python the language and CPython its particular implementation. This however leaves more freedom for other implementations. For example, MicroPython does a lot of things differently than CPython, while still aspiring to be a Python language implementation.

GPIO General-purpose input/output. The simplest means to control electrical signals. With GPIO, user can configure hardware signal pin to be either input or output, and set or get its digital signal value (logical "0" or "1"). MicroPython abstracts GPIO access using `machine.Pin` and `machine.Signal` classes.

GPIO port A group of *GPIO* pins, usually based on hardware properties of these pins (e.g. controllable by the same register).

interned string A string referenced by its (unique) identity rather than its address. Interned strings are thus can be quickly compared just by their identifiers, instead of comparing by content. The drawbacks of interned strings are that interning operation takes time (proportional to the number of existing interned strings, i.e. becoming slower and slower over time) and that the space used for interned strings is not reclaimable. String interning is done automatically by MicroPython compiler and runtimer when it's either required by the implementation (e.g. function keyword arguments are represented by interned string id's) or deemed beneficial (e.g. for short enough strings, which have a chance to be repeated, and thus interning them would save memory on copies). Most of string and I/O operations don't produce interned strings due to drawbacks described above.

MCU Microcontroller. Microcontrollers usually have much less resources than a full-fledged computing system, but smaller, cheaper and require much less power. MicroPython is designed to be small and optimized enough to run on an average modern microcontroller.

micropython-lib MicroPython is (usually) distributed as a single executable/binary file with just few builtin modules. There is no extensive standard library comparable with *CPython*. Instead, there is a related, but separate project micropython-lib which provides implementations for many modules from CPython's standard library. However, large subset of these modules require POSIX-like environment (Linux, FreeBSD, MacOS, etc.; Windows may be partially supported), and thus would work or make sense only with *`MicroPython Unix port`*. Some subset of modules is however usable for *`baremetal`* ports too.

Unlike monolithic *CPython* stdlib, micropython-lib modules are intended to be installed individually - either using manual copying or using *upip*.

MicroPython port MicroPython supports different *boards*, RTOSes, and OSes, and can be relatively easily adapted to new systems. MicroPython with support for a particular system is called a "port" to that system. Different ports may have widely different functionality. This documentation is intended to be a reference of the generic APIs available across different ports ("MicroPython core"). Note that some ports may still omit some APIs described here (e.g. due to resource constraints). Any such differences, and port-specific extensions beyond MicroPython core functionality, would be described in the separate port-specific documentation.

MicroPython Unix port Unix port is one of the major *MicroPython ports*. It is intended to run on POSIX-compatible operating systems, like Linux, MacOS, FreeBSD, Solaris, etc. It also serves as the basis of Windows port. The importance of Unix port lies in the fact that while there are many different *boards*, so two random users unlikely have the same board, almost all modern OSes have some level of POSIX compatibility, so Unix port serves as a kind of "common ground" to which any user can have access. So, Unix port is used for initial prototyping, different kinds of testing, development of machine-independent features, etc. All users of MicroPython, even those which are interested only in running MicroPython on *MCU* systems, are recommended to be familiar with Unix (or Windows) port, as it is important productivity helper and a part of normal MicroPython workflow.

port Either *MicroPython port* or *GPIO port*. If not clear from context, it's recommended to use full specification like one of the above.

stream Also known as a "file-like object". An object which provides sequential read-write access to the underlying data. A stream object implements a corresponding interface, which consists of methods like `read()`, `write()`, `readinto()`, `seek()`, `flush()`, `close()`, etc. A stream is an important concept in MicroPython, many I/O objects implement the stream interface, and thus can be used consistently and interchangeably in different contexts. For more information on streams in MicroPython, see `uio` module.

upip (Literally, "micro pip"). A package manage for MicroPython, inspired by *CPython*'s pip, but much smaller and with reduced functionality. upip runs both on *Unix port* and on *baremetal* ports (those which offer filesystem and networking support).

5.2 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.

5.2.1 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(30):
...     _
```

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.

5.2.2 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, first import the machine module by entering `import machine` and pressing RETURN. Then 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
```

5.2.3 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:
>>>
```

5.2.4 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.

5.2.5 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:

```
machine.soft_reset()
```

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']
>>>
```

5.2.6 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
>>>
```

5.2.7 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 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.

5.3.1 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.
- Consider using `micropython.schedule` to work around the above constraint.
- 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).

5.3.2 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))
green = 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).

An instance of object creation occurs when a reference to a bound method is created. This means that an ISR cannot pass a bound method to a function. One solution is to create a reference to the bound method in the class constructor and to pass that reference in the ISR. For example:

```
class Foo():
    def __init__(self):
        self.bar_ref = self.bar  # Allocation occurs here
        self.x = 0.1
        tim = pyb.Timer(4)
        tim.init(freq=2)
        tim.callback(self.cb)

    def bar(self, _):
        self.x *= 1.2
        print(self.x)

    def cb(self, t):
        # Passing self.bar would cause allocation.
        micropython.schedule(self.bar_ref, 0)
```

Other techniques are to define and instantiate the method in the constructor or to pass `Foo.bar()` with the argument *self*.

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.

Using micropython.schedule

This function enables an ISR to schedule a callback for execution "very soon". The callback is queued for execution which will take place at a time when the heap is not locked. Hence it can create Python objects and use floats. The callback is also guaranteed to run at a time when the main program has completed any update of Python objects, so the callback will not encounter partially updated objects.

Typical usage is to handle sensor hardware. The ISR acquires data from the hardware and enables it to issue a further interrupt. It then schedules a callback to process the data.

Scheduled callbacks should comply with the principles of interrupt handler design outlined below. This is to avoid problems resulting from I/O activity and the modification of shared data which can arise in any code which pre-empts the main program loop.

Execution time needs to be considered in relation to the frequency with which interrupts can occur. If an interrupt occurs while the previous callback is executing, a further instance of the callback will be queued for execution; this will run after the current instance has completed. A sustained high interrupt repetition rate therefore carries a risk of unconstrained queue growth and eventual failure with a `RuntimeError`.

If the callback to be passed to *`schedule()`* is a bound method, consider the note in "Creation of Python objects".

5.3.3 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.

5.3.4 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`.

5.4 Maximising MicroPython Speed

Contents

- *Maximising MicroPython Speed*
  - *Designing for speed*
    - *Algorithms*
    - *RAM Allocation*
    - *Buffers*
    - *Floating Point*
    - *Arrays*
  - *Identifying the slowest section of code*
  - *MicroPython code improvements*

        * *The const() declaration*
        * *Caching object references*
        * *Controlling garbage collection*
    - *The Native code emitter*
    - *The Viper code emitter*
    - *Accessing hardware directly*

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 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.
- Use hardware-specific optimisations.

5.4.1 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 allocates 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()`.

5.4.2 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 in *utime*. 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 = utime.ticks_us()
        result = f(*args, **kwargs)
        delta = utime.ticks_diff(utime.ticks_us(), t)
        print('Function {} Time = {:6.3f}ms'.format(myname, delta/1000))
        return result
    return new_func
```

5.4.3 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 may be 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.

5.4.4 The Native code emitter

This causes the MicroPython compiler to emit native CPU opcodes rather than bytecode. It covers the bulk of the MicroPython functionality, 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.

5.4.5 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

5.4.6 Accessing hardware directly

Note: Code examples in this section are given for the Pyboard. The techniques described however may be applied to other MicroPython ports too.

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:

```
import machine
import stm

BIT14 = const(1 << 14)
machine.mem16[stm.GPIOA + stm.GPIO_ODR] ^= BIT14
```

5.5 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.

5.5.1 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.

5.5.2 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. `str.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 `str.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 funcitons *eval* and *exec* invoke the compiler at runtime, which requires significant amounts of RAM. Note that the `pickle` library from *micropython-lib* employs *exec*. It may be more RAM efficient to use the *ujson* 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 ports/stm32 (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.

5.5.3 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.

5.5.4 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.

5.5.5 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.

5.6 Distribution packages, package management, and deploying applications

Just as the "big" Python, MicroPython supports creation of "third party" packages, distributing them, and easily installing them in each user's environment. This chapter discusses how these actions are achieved. Some familiarity with Python packaging is recommended.

5.6.1 Overview

Steps below represent a high-level workflow when creating and consuming packages:

1. Python modules and packages are turned into distribution package archives, and published at the Python Package Index (PyPI).
2. *upip* package manager can be used to install a distribution package on a *MicroPython port* with networking capabilities (for example, on the Unix port).
3. For ports without networking capabilities, an "installation image" can be prepared on the Unix port, and transferred to a device by suitable means.
4. For low-memory ports, the installation image can be frozen as the bytecode into MicroPython executable, thus minimizing the memory storage overheads.

The sections below describe this process in details.

5.6.2 Distribution packages

Python modules and packages can be packaged into archives suitable for transfer between systems, storing at the well-known location (PyPI), and downloading on demand for deployment. These archives are known as *distribution packages* (to differentiate them from Python packages (means to organize Python source code)).

The MicroPython distribution package format is a well-known tar.gz format, with some adaptations however. The Gzip compressor, used as an external wrapper for TAR archives, by default uses 32KB dictionary size, which means that to uncompress a compressed stream, 32KB of contguous memory needs to be allocated. This requirement may be

not satisfiable on low-memory devices, which may have total memory available less than that amount, and even if not, a contiguous block like that may be hard to allocate due to memory fragmentation. To accommodate these constraints, MicroPython distribution packages use Gzip compression with the dictionary size of 4K, which should be a suitable compromise with still achieving some compression while being able to uncompressed even by the smallest devices.

Besides the small compression dictionary size, MicroPython distribution packages also have other optimizations, like removing any files from the archive which aren't used by the installation process. In particular, *upip* package manager doesn't execute `setup.py` during installation (see below), and thus that file is not included in the archive.

At the same time, these optimizations make MicroPython distribution packages not compatible with *CPython*'s package manager, `pip`. This isn't considered a big problem, because:

1. Packages can be installed with *upip*, and then can be used with CPython (if they are compatible with it).
2. In the other direction, majority of CPython packages would be incompatible with MicroPython by various reasons, first of all, the reliance on features not implemented by MicroPython.

Summing up, the MicroPython distribution package archives are highly optimized for MicroPython's target environments, which are highly resource constrained devices.

5.6.3 `upip` package manager

MicroPython distribution packages are intended to be installed using the *upip* package manager. *upip* is a Python application which is usually distributed (as frozen bytecode) with network-enabled *MicroPython ports*. At the very least, *upip* is available in the *MicroPython Unix port*.

On any *MicroPython port* providing *upip*, it can be accessed as following:

```
import upip
upip.help()
upip.install(package_or_package_list, [path])
```

Where *package_or_package_list* is the name of a distribution package to install, or a list of such names to install multiple packages. Optional *path* parameter specifies filesystem location to install under and defaults to the standard library location (see below).

An example of installing a specific package and then using it:

```
>>> import upip
>>> upip.install("micropython-pystone_lowmem")
[...]
>>> import pystone_lowmem
>>> pystone_lowmem.main()
```

Note that the name of Python package and the name of distribution package for it in general don't have to match, and oftentimes they don't. This is because PyPI provides a central package repository for all different Python implementations and versions, and thus distribution package names may need to be namespaced for a particular implementation. For example, all packages from *micropython-lib* follow this naming convention: for a Python module or package named `foo`, the distribution package name is `micropython-foo`.

For the ports which run MicroPython executable from the OS command prompts (like the Unix port), *upip* can be (and indeed, usually is) run from the command line instead of MicroPython's own REPL. The commands which corresponds to the example above are:

```
micropython -m upip -h
micropython -m upip install [-p <path>] <packages>...
micropython -m upip install micropython-pystone_lowmem
```

[TODO: Describe installation path.]

5.6.4 Cross-installing packages

For *MicroPython ports* without native networking capabilities, the recommend process is "cross-installing" them into a "directory image" using the *MicroPython Unix port*, and then transferring this image to a device by suitable means.

Installing to a directory image involves using `-p` switch to *upip*:

```
micropython -m upip install -p install_dir micropython-pystone_lowmem
```

After this command, the package content (and contents of every depenency packages) will be available in the `install_dir/` subdirectory. You would need to transfer contents of this directory (without the `install_dir/` prefix) to the device, at the suitable location, where it can be found by the Python `import` statement (see discussion of the *upip* installation path above).

5.6.5 Cross-installing packages with freezing

For the low-memory *MicroPython ports*, the process described in the previous section does not provide the most efficient resource usage,because the packages are installed in the source form, so need to be compiled to the bytecome on each import. This compilation requires RAM, and the resulting bytecode is also stored in RAM, reducing its amount available for storing application data. Moreover, the process above requires presence of the filesystem on a device, and the most resource-constrained devices may not even have it.

The bytecode freezing is a process which resolves all the issues mentioned above:

- The source code is pre-compiled into bytecode and store as such.
- The bytecode is stored in ROM, not RAM.
- Filesystem is not required for frozen packages.

Using frozen bytecode requires building the executable (firmware) for a given *MicroPython port* from the C source code. Consequently, the process is:

1. Follow the instructions for a particular port on setting up a toolchain and building the port. For example, for ESP8266 port, study instructions in `ports/esp8266/README.md` and follow them. Make sure you can build the port and deploy the resulting executable/firmware successfully before proceeding to the next steps.
2. Build *MicroPython Unix port* and make sure it is in your PATH and you can execute `micropython`.
3. Change to port's directory (e.g. `ports/esp8266/` for ESP8266).
4. Run `make clean-frozen`. This step cleans up any previous modules which were installed for freezing (consequently, you need to skip this step to add additional modules, instead of starting from scratch).
5. Run `micropython -m upip install -p modules <packages>...` to install packages you want to freeze.
6. Run `make clean`.
7. Run `make`.

After this, you should have the executable/firmware with modules as the bytecode inside, which you can deploy the usual way.

Few notes:

1. Step 5 in the sequence above assumes that the distribution package is available from PyPI. If that is not the case, you would need to copy Python source files manually to `modules/` subdirectory of the port port directory. (Note that upip does not support installing from e.g. version control repositories).

2. The firmware for baremetal devices usually has size restrictions, so adding too many frozen modules may overflow it. Usually, you would get a linking error if this happens. However, in some cases, an image may be produced, which is not runnable on a device. Such cases are in general bugs, and should be reported and further investigated. If you face such a situation, as an initial step, you may want to decrease the amount of frozen modules included.

5.6.6 Creating distribution packages

Distribution packages for MicroPython are created in the same manner as for CPython or any other Python implementation, see references at the end of chapter. Setuptools (instead of distutils) should be used, because distutils do not support dependencies and other features. "Source distribution" (`sdist`) format is used for packaging. The post-processing discussed above, (and pre-processing discussed in the following section) is achieved by using custom `sdist` command for setuptools. Thus, packaging steps remain the same as for the standard setuptools, the user just needs to override `sdist` command implementation by passing the appropriate argument to `setup()` call:

```
from setuptools import setup
import sdist_upip

setup(
    ...,
    cmdclass={'sdist': sdist_upip.sdist}
)
```

The sdist_upip.py module as referenced above can be found in *micropython-lib*: https://github.com/micropython/micropython-lib/blob/master/sdist_upip.py

5.6.7 Application resources

A complete application, besides the source code, oftentimes also consists of data files, e.g. web page templates, game images, etc. It's clear how to deal with those when application is installed manually - you just put those data files in the filesystem at some location and use the normal file access functions.

The situation is different when deploying applications from packages - this is more advanced, streamlined and flexible way, but also requires more advanced approach to accessing data files. This approach is treating the data files as "resources", and abstracting away access to them.

Python supports resource access using its "setuptools" library, using `pkg_resources` module. MicroPython, following its usual approach, implements subset of the functionality of that module, specifically `pkg_resources.resource_stream(package, resource)` function. The idea is that an application calls this function, passing a resource identifier, which is a relative path to data file within the specified package (usually top-level application package). It returns a stream object which can be used to access resource contents. Thus, the `resource_stream()` emulates interface of the standard *open()* function.

Implementation-wise, `resource_stream()` uses file operations underlyingly, if distribution package is install in the filesystem. However, it also supports functioning without the underlying filesystem, e.g. if the package is frozen as the bytecode. This however requires an extra intermediate step when packaging application - creation of "Python resource module".

The idea of this module is to convert binary data to a Python bytes object, and put it into the dictionary, indexed by the resource name. This conversion is done automatically using overridden `sdist` command described in the previous section.

Let's trace the complete process using the following example. Suppose your application has the following structure:

```
my_app/
    __main__.py
    utils.py
    data/
        page.html
        image.png
```

`__main__.py` and `utils.py` should access resources using the following calls:

```
import pkg_resources

pkg_resources.resource_stream(__name__, "data/page.html")
pkg_resources.resource_stream(__name__, "data/image.png")
```

You can develop and debug using the *MicroPython Unix port* as usual. When time comes to make a distribution package out of it, just use overridden "sdist" command from sdist_upip.py module as described in the previous section.

This will create a Python resource module named `R.py`, based on the files declared in `MANIFEST` or `MANIFEST.in` files (any non-`.py` file will be considered a resource and added to `R.py`) - before proceeding with the normal packaging steps.

Prepared like this, your application will work both when deployed to filesystem and as frozen bytecode.

If you would like to debug `R.py` creation, you can run:

```
python3 setup.py sdist --manifest-only
```

Alternatively, you can use tools/mpy_bin2res.py script from the MicroPython distribution, in which can you will need to pass paths to all resource files:

```
mpy_bin2res.py data/page.html data/image.png
```

5.6.8 References

- Python Packaging User Guide: https://packaging.python.org/
- Setuptools documentation: https://setuptools.readthedocs.io/
- Distutils documentation: https://docs.python.org/3/library/distutils.html

CHAPTER

SIX

MICROPYTHON DIFFERENCES FROM CPYTHON

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

6.1 Syntax

Generated Mon 01 Oct 2018 04:54:58 UTC

6.1.1 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

6.1.2 Unicode

Unicode name escapes are not implemented

Sample code:

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

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

6.2 Core Language

Generated Mon 01 Oct 2018 04:54:58 UTC

6.2.1 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__	

Method Resolution Order (MRO) is not compliant with CPython

Cause: Depth first non-exhaustive method resolution order

Workaround: Avoid complex class hierarchies with multiple inheritance and complex method overrides. Keep in mind that many languages don't support multiple inheritance at all.

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)`

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

Cause: See *Method Resolution Order (MRO) is not compliant with CPython*

Workaround: See *Method Resolution Order (MRO) is not compliant with CPython*

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>`

6.2.2 Functions

Error messages for methods may display unexpected argument counts

Cause: MicroPython counts "self" as an argument.

Workaround: Interpret error messages with the information above in mind.

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 given`

User-defined attributes for functions are not supported

Cause: MicroPython is highly optimized for memory usage.

Workaround: Use external dictionary, e.g. `FUNC_X[f] = 0`.

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 'x'

6.2.3 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

6.2.4 Runtime

Local variables aren't included in locals() result

Cause: MicroPython doesn't maintain symbolic local environment, it is optimized to an array of slots. Thus, local variables can't be accessed by a name.

Sample code:

```
def test():
    val = 2
```

```
    print(locals())

test()
```

CPy output:	uPy output:
`{'val': 2}`	`{'test': <function test at 0xf7acb180>, '__name__': '__main__', '__file__': '<stdin>'}`

Code running in eval() function doesn't have access to local variables

Cause: MicroPython doesn't maintain symbolic local environment, it is optimized to an array of slots. Thus, local variables can't be accessed by a name. Effectively, `eval(expr)` in MicroPython is equivalent to `eval(expr, globals(), globals())`.

Sample code:

```
val = 1

def test():
    val = 2
    print(val)
    eval("print(val)")

test()
```

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

6.2.5 import

__path__ attribute of a package has a different type (single string instead of list of strings) in MicroPython

Cause: MicroPython does't support namespace packages split across filesystem. Beyond that, MicroPython's import system is highly optimized for minimal memory usage.

Workaround: Details of import handling is inherently implementation dependent. Don't rely on such details in portable applications.

Sample code:

```
import modules

print(modules.__path__)
```

CPy output:	uPy output:
['/home/micropython/micropython-docs/tests/cpydiff/modules']	../tests/cpydiff//modules

Failed to load modules are still registered as loaded

Cause: To make module handling more efficient, it's not wrapped with exception handling.

Workaround: Test modules before production use; during development, use `del sys.modules["name"]`, or just soft or hard reset the board.

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

MicroPython doesn't support namespace packages split across filesystem.

Cause: MicroPython's import system is highly optimized for simplicity, minimal memory usage, and minimal filesystem search overhead.

Workaround: Don't install modules belonging to the same namespace package in different directories. For MicroPython, it's recommended to have at most 3-component module search paths: for your current application, per-user (writable), system-wide (non-writable).

Sample code:

```
import sys
sys.path.append(sys.path[1] + "/modules")
sys.path.append(sys.path[1] + "/modules2")

import subpkg.foo
import subpkg.bar

print("Two modules of a split namespace package imported")
```

CPy output:	uPy output:

`Two modules of a split namespace package imported`	`Traceback (most recent call last):`
	`  File "<stdin>", line 12, in <module>`
	`ImportError: no module named 'subpkg.bar'`

6.3 Builtin Types

Generated Mon 01 Oct 2018 04:54:58 UTC

6.3.1 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:`

User-defined attributes for builtin exceptions are not supported

Cause: MicroPython is highly optimized for memory usage.

Workaround: Use user-defined exception subclasses.

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 attribute 'x'

Exception in while loop condition may have unexpected line number

Cause: Condition checks are optimized to happen at the end of loop body, and that line number is reported.

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__ method does not exist.

Cause: Subclassing native classes is not fully supported in MicroPython.

Workaround: Call using `super()` instead:

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

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 15, in <module>
	  File "<stdin>", line 13, in __init__
	AttributeError: type object 'Exception' has no attribute '__init__'

6.3.2 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')	Traceback (most recent call last): File "<stdin>", line 8, in <module> NotImplementedError: array/bytes required on right side

6.3.3 bytes

bytes() with keywords not implemented

Workaround: Pass the encoding as a positional paramter, e.g. `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 implemented - use normal args instead

Bytes subscription with step != 1 not implemented

Cause: MicroPython is highly optimized for memory usage.

Workaround: Use explicit loop for this very rare operation.

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 None) are supported`

6.3.4 float

uPy and CPython outputs formats may differ

Sample code:

```
print('%.1g' % -9.9)
```

CPy output:	uPy output:
`-1e+01`	`-10`

6.3.5 int

No int conversion for int-derived types available

Workaround: Avoid subclassing builtin types unless really needed. Prefer https://en.wikipedia.org/wiki/Composition_over_inheritance .

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`

6.3.6 list

List delete with step != 1 not implemented

Workaround: Use explicit loop for this rare operation.

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

Workaround: Use explicit loop for this rare operation.

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:`

6.3.7 str

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 implemented - use normal args␣`
	`↪instead`

str.ljust() and str.rjust() not implemented

Cause: MicroPython is highly optimized for memory usage. Easy workarounds available.

Workaround: Instead of `s.ljust(10)` use `"%-10s" % s`, instead of `s.rjust(10)` use `"% 10s" % s`. Alternatively, `"{:<10}".format(s)` or `"{:>10}".format(s)`.

Sample code:

```
print('abc'.ljust(10))
```

CPy output:	uPy output:
abc	Traceback (most recent call last): File "<stdin>", line 7, in <module> AttributeError: 'str' object has no attribute 'ljust'

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 None) are supported`

6.3.8 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 None) are supported`

6.4 Modules

Generated Mon 01 Oct 2018 04:54:58 UTC

6.4.1 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 deletion`

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 None) are supported`

6.4.2 builtins

Second argument to next() is not implemented

Cause: MicroPython is optimised for code space.

Workaround: Instead of `val = next(it, deflt)` use:

```
try:
    val = next(it)
except StopIteration:
    val = deflt
```

Sample code:

```
print(next(iter(range(0)), 42))
```

CPy output:	uPy output:

`42`	`Traceback (most recent call last):`
	`  File "<stdin>", line 12, in <module>`
	`TypeError: function takes 1 positional arguments but 2 were given`

6.4.3 deque

Deque not implemented

Workaround: Use regular lists. micropython-lib has implementation of collections.deque.

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>`
	`TypeError: function missing 2 required positional arguments`

6.4.4 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`

6.4.5 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`

6.4.6 sys

Overriding sys.stdin, sys.stdout and sys.stderr not possible

Cause: 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 'stdin'`

CHAPTER

SEVEN

MICROPYTHON LICENSE INFORMATION

The MIT License (MIT)

Copyright (c) 2013-2017 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.

PYTHON MODULE INDEX

a

array, 27

b

btree, 45

f

framebuf, 48

g

gc, 27

m

machine, 50
micropython, 66

n

network, 68

s

sys, 28

u

ubinascii, 30
uctypes, 72
ujson, 30
uos, 31
ure, 34
uselect, 35
usocket, 37
ussl, 41
utime, 42

w

wipy, 75

INDEX

Symbols

__call__() (machine.Pin method), 54
__contains__() (btree.btree method), 47
__detitem__() (btree.btree method), 47
__getitem__() (btree.btree method), 47
__iter__() (btree.btree method), 47
__setitem__() (btree.btree method), 47

A

a2b_base64() (in module ubinascii), 30
abs() (built-in function), 24
AbstractBlockDev (class in uos), 33
AbstractNIC (class in network), 69
accept() (usocket.socket method), 39
active() (in module network), 69
ADC (class in machine), 65
adcchannel() (in module machine), 65
addressof() (in module uctypes), 74
AF_INET (in module usocket), 38
AF_INET6 (in module usocket), 38
alarm() (machine.RTC method), 61
alarm_left() (machine.RTC method), 61
all() (built-in function), 24
alloc_emergency_exception_buf() (in module micropython), 67
alt_list() (machine.Pin method), 10
antenna() (network.wlan method), 72
any() (built-in function), 24
any() (machine.UART method), 56
append() (array.array.array method), 27
argv (in module sys), 28
array (module), 27
array.array (class in array), 27
AssertionError, 26
AttributeError, 26
auth() (network.wlan method), 72

B

b2a_base64() (in module ubinascii), 30
baremetal, **77**
BIG_ENDIAN (in module uctypes), 74
bin() (built-in function), 24
bind() (usocket.socket method), 39
blit() (framebuf.FrameBuffer method), 49
board, **77**
bool (built-in class), 24
btree (module), 45
bytearray (built-in class), 24
bytearray_at() (in module uctypes), 74
byteorder (in module sys), 28
bytes (built-in class), 24
bytes_at() (in module uctypes), 74

C

callable() (built-in function), 24
callee-owned tuple, **77**
cancel() (machine.RTC method), 61
channel() (machine.ADC method), 65
channel() (machine.Timer method), 63
channel() (network.wlan method), 72
chdir() (in module uos), 31
chr() (built-in function), 24
classmethod() (built-in function), 24
close() (btree.btree method), 47
close() (usocket.socket method), 39
collect() (in module gc), 27
compile() (built-in function), 24
compile() (in module ure), 35
complex (built-in class), 24
config() (in module network), 70
connect() (in module network), 69
connect() (network.wlan method), 71
connect() (usocket.socket method), 39
const() (in module micropython), 66
CPython, **77**

D

DEBUG (in module ure), 35
deepsleep() (in module machine), 51
deinit() (machine.ADC method), 65
deinit() (machine.adcchannel method), 66
deinit() (machine.I2C method), 59
deinit() (machine.RTC method), 61
deinit() (machine.SD method), 66

deinit() (machine.SPI method), 58
deinit() (machine.Timer method), 63
deinit() (machine.UART method), 56
deinit() (server method), 12
delattr() (built-in function), 24
DESC (in module btree), 48
dict (built-in class), 24
dir() (built-in function), 24
disable() (in module gc), 27
disable_irq() (in module machine), 50
disconnect() (in module network), 69
disconnect() (network.wlan method), 71
divmod() (built-in function), 24
drive() (machine.Pin method), 54
dump() (in module ujson), 30
dumps() (in module ujson), 30
dupterm() (in module uos), 32
duty_cycle() (machine.timerchannel method), 63

E

enable() (in module gc), 27
enable_irq() (in module machine), 50
enumerate() (built-in function), 24
eval() (built-in function), 24
Exception, 26
exec() (built-in function), 24
exit() (in module sys), 28
extend() (array.array.array method), 27

F

feed() (machine.wdt method), 65
fill() (framebuf.FrameBuffer method), 49
fill_rect() (framebuf.FrameBuffer method), 49
filter() (built-in function), 24
float (built-in class), 24
flush() (btree.btree method), 47
framebuf (module), 48
framebuf.GS2_HMSB (in module framebuf), 50
framebuf.GS4_HMSB (in module framebuf), 50
framebuf.GS8 (in module framebuf), 50
framebuf.MONO_HLSB (in module framebuf), 49
framebuf.MONO_HMSB (in module framebuf), 50
framebuf.MONO_VLSB (in module framebuf), 49
framebuf.RGB565 (in module framebuf), 50
FrameBuffer (class in framebuf), 48
freq() (in module machine), 50
freq() (machine.timerchannel method), 63
from_bytes() (int class method), 25
frozenset (built-in class), 24

G

gc (module), 27
get() (btree.btree method), 47
getaddrinfo() (in module usocket), 38
getattr() (built-in function), 24
getcwd() (in module uos), 31
globals() (built-in function), 24
GPIO, **77**
GPIO port, **78**
group() (ure.match method), 35

H

hasattr() (built-in function), 25
hash() (built-in function), 25
heap_lock() (in module micropython), 67
heap_unlock() (in module micropython), 67
heartbeat() (in module wipy), 75
hex() (built-in function), 25
hexlify() (in module ubinascii), 30
hline() (framebuf.FrameBuffer method), 49

I

I2C (class in machine), 59
id() (built-in function), 25
idle() (in module machine), 50
ifconfig() (in module network), 70
ifconfig() (network.wlan method), 71
ilistdir() (in module uos), 31
implementation (in module sys), 28
ImportError, 26
INCL (in module btree), 48
IndexError, 26
inet_ntop() (in module usocket), 38
inet_pton() (in module usocket), 38
init() (machine.ADC method), 65
init() (machine.adcchannel method), 66
init() (machine.I2C method), 59
init() (machine.Pin method), 53
init() (machine.RTC method), 61
init() (machine.SD method), 66
init() (machine.SPI method), 58
init() (machine.Timer method), 62
init() (machine.UART method), 56
init() (network.wlan method), 71
init() (server method), 12
input() (built-in function), 25
int (built-in class), 25
interned string, **78**
ioctl() (uos.AbstractBlockDev method), 33
ipoll() (uselect.poll method), 36
IPPROTO_SEC (in module usocket), 39
IPPROTO_TCP (in module usocket), 39
IPPROTO_UDP (in module usocket), 39
irq() (machine.Pin method), 54
irq() (machine.RTC method), 62
irq() (machine.timerchannel method), 64
irq() (machine.UART method), 57
irq() (network.wlan method), 72

isconnected() (in module network), 69
isconnected() (network.wlan method), 71
isinstance() (built-in function), 25
isrunning() (server method), 12
issubclass() (built-in function), 25
items() (btree.btree method), 47
iter() (built-in function), 25

K

kbd_intr() (in module micropython), 67
KeyboardInterrupt, 26
KeyError, 26
keys() (btree.btree method), 47

L

len() (built-in function), 25
line() (framebuf.FrameBuffer method), 49
list (built-in class), 25
listdir() (in module uos), 31
listen() (usocket.socket method), 39
LITTLE_ENDIAN (in module uctypes), 74
load() (in module ujson), 30
loads() (in module ujson), 30
locals() (built-in function), 25
localtime() (in module utime), 43

M

mac() (network.wlan method), 72
machine (module), 50
machine.DEEPSLEEP (in module machine), 51
machine.DEEPSLEEP_RESET (in module machine), 51
machine.HARD_RESET (in module machine), 51
machine.IDLE (in module machine), 51
machine.PIN_WAKE (in module machine), 51
machine.PWRON_RESET (in module machine), 51
machine.RTC_WAKE (in module machine), 51
machine.SLEEP (in module machine), 51
machine.SOFT_RESET (in module machine), 51
machine.WDT_RESET (in module machine), 51
machine.WLAN_WAKE (in module machine), 51
main() (built-in function), 12
makefile() (usocket.socket method), 40
map() (built-in function), 25
match() (in module ure), 35
match() (ure.regex method), 35
max() (built-in function), 25
maxsize (in module sys), 29
MCU, **78**
mem_alloc() (in module gc), 27
mem_free() (in module gc), 27
mem_info() (in module micropython), 67
MemoryError, 26
memoryview (built-in class), 25
micropython (module), 66
MicroPython port, **78**
MicroPython Unix port, **78**
micropython-lib, **78**
min() (built-in function), 25
mkdir() (in module uos), 31
mkfs() (built-in function), 13
mkfs() (uos.VfsFat static method), 33
mktime() (in module utime), 43
mode() (machine.Pin method), 54
mode() (network.wlan method), 71
modify() (uselect.poll method), 36
modules (in module sys), 29
mount() (built-in function), 13
mount() (in module uos), 33

N

NameError, 26
NATIVE (in module uctypes), 74
network (module), 68
network.Server (built-in class), 12
next() (built-in function), 25
NotImplementedError, 26
now() (machine.RTC method), 61

O

object (built-in class), 25
oct() (built-in function), 25
off() (machine.Pin method), 54
on() (machine.Pin method), 54
open() (built-in function), 25
open() (in module btree), 47
opt_level() (in module micropython), 67
ord() (built-in function), 25
OSError, 26

P

path (in module sys), 29
period() (machine.timerchannel method), 63
Pin (class in machine), 52
Pin.ALT (in module machine), 55
Pin.ALT_OPEN_DRAIN (in module machine), 55
Pin.HIGH_POWER (in module machine), 55
Pin.IN (in module machine), 55
Pin.IRQ_FALLING (in module machine), 55
Pin.IRQ_HIGH_LEVEL (in module machine), 55
Pin.IRQ_LOW_LEVEL (in module machine), 55
Pin.IRQ_RISING (in module machine), 55
Pin.LOW_POWER (in module machine), 55
Pin.MED_POWER (in module machine), 55
Pin.OPEN_DRAIN (in module machine), 55
Pin.OUT (in module machine), 55
Pin.PULL_DOWN (in module machine), 55
Pin.PULL_UP (in module machine), 55
pixel() (framebuf.FrameBuffer method), 49

platform (in module sys), 29
poll() (in module uselect), 36
poll() (uselect.poll method), 36
port, **78**
pow() (built-in function), 25
print() (built-in function), 25
print_exception() (in module sys), 28
property() (built-in function), 25
pull() (machine.Pin method), 54

Q

qstr_info() (in module micropython), 67

R

range() (built-in function), 25
read() (machine.SPI method), 58
read() (machine.UART method), 56
read() (usocket.socket method), 41
readblocks() (uos.AbstractBlockDev method), 33
readfrom() (machine.I2C method), 60
readfrom_into() (machine.I2C method), 60
readfrom_mem() (machine.I2C method), 61
readfrom_mem_into() (machine.I2C method), 61
readinto() (machine.I2C method), 60
readinto() (machine.SPI method), 58
readinto() (machine.UART method), 56
readinto() (usocket.socket method), 41
readline() (machine.UART method), 56
readline() (usocket.socket method), 41
rect() (framebuf.FrameBuffer method), 49
recv() (usocket.socket method), 40
recvfrom() (usocket.socket method), 40
register() (uselect.poll method), 36
remove() (in module uos), 31
rename() (in module uos), 32
repr() (built-in function), 25
reset() (in module machine), 50
reset_cause() (in module machine), 50
reversed() (built-in function), 25
rmdir() (in module uos), 31
rng() (in module machine), 51
round() (built-in function), 25
RTC (class in machine), 61
RTC.ALARM0 (in module machine), 62
RuntimeError, 26

S

scan() (in module network), 69
scan() (machine.I2C method), 60
scan() (network.wlan method), 71
schedule() (in module micropython), 68
scroll() (framebuf.FrameBuffer method), 49
SD (class in machine), 66
search() (in module ure), 35
search() (ure.regex method), 35
select() (in module uselect), 36
send() (usocket.socket method), 39
sendall() (usocket.socket method), 39
sendbreak() (machine.UART method), 57
sendto() (usocket.socket method), 40
set (built-in class), 25
setattr() (built-in function), 25
setblocking() (usocket.socket method), 40
setsockopt() (usocket.socket method), 40
settimeout() (usocket.socket method), 40
sizeof() (in module uctypes), 74
sleep() (in module machine), 51
sleep() (in module utime), 43
sleep_ms() (in module utime), 43
sleep_us() (in module utime), 43
slice (built-in class), 25
SOCK_DGRAM (in module usocket), 38
SOCK_STREAM (in module usocket), 38
socket() (in module usocket), 38
sorted() (built-in function), 26
SPI (class in machine), 57
SPI.LSB (in module machine), 58
SPI.MASTER (in module machine), 58
SPI.MSB (in module machine), 58
split() (ure.regex method), 35
ssid() (network.wlan method), 72
ssl.SSLError (in module ussl), 42
stack_use() (in module micropython), 67
start() (machine.I2C method), 60
stat() (in module uos), 32
staticmethod() (built-in function), 26
status() (in module network), 69
statvfs() (in module uos), 32
stderr (in module sys), 29
stdin (in module sys), 29
stdout (in module sys), 29
stop() (machine.I2C method), 60
StopIteration, 26
str (built-in class), 26
stream, **78**
struct (class in uctypes), 74
sum() (built-in function), 26
super() (built-in function), 26
sync() (in module uos), 32
SyntaxError, 26
sys (module), 28
SystemExit, 26

T

text() (framebuf.FrameBuffer method), 49
threshold() (in module gc), 27
ticks_add() (in module utime), 44
ticks_cpu() (in module utime), 43

ticks_diff() (in module utime), 44
ticks_ms() (in module utime), 43
ticks_us() (in module utime), 43
time() (in module utime), 45
time_pulse_us() (in module machine), 51
timeout() (server method), 12
Timer (class in machine), 62
Timer.A (built-in variable), 20
Timer.B (built-in variable), 21
Timer.MATCH (built-in variable), 21
Timer.NEGATIVE (built-in variable), 21
Timer.ONE_SHOT (in module machine), 64
Timer.PERIODIC (in module machine), 64
Timer.POSITIVE (built-in variable), 21
Timer.PWM (built-in variable), 20
Timer.TIMEOUT (built-in variable), 21
to_bytes() (int method), 25
tuple (built-in class), 26
type() (built-in function), 26
TypeError, 26

U

UART (class in machine), 56
UART.RX_ANY (in module machine), 57
ubinascii (module), 30
uctypes (module), 72
ujson (module), 30
umount() (in module uos), 33
uname() (in module uos), 31
unhexlify() (in module ubinascii), 30
unique_id() (in module machine), 51
unmount() (built-in function), 13
unregister() (uselect.poll method), 36
uos (module), 31
upip, **78**
urandom() (in module uos), 31
ure (module), 34
uselect (module), 35
usocket (module), 37
usocket.error, 41
ussl (module), 41
ussl.CERT_NONE (in module ussl), 42
ussl.CERT_OPTIONAL (in module ussl), 42
ussl.CERT_REQUIRED (in module ussl), 42
ussl.wrap_socket() (in module ussl), 42
utime (module), 42

V

value() (machine.adcchannel method), 65
value() (machine.Pin method), 53
ValueError, 26
values() (btree.btree method), 47
version (in module sys), 29
version_info (in module sys), 29
VfsFat (class in uos), 33
vline() (framebuf.FrameBuffer method), 49

W

wake_reason() (in module machine), 51
WDT (class in machine), 64
wipy (module), 75
WLAN (class in network), 70
WLAN.AP (in module network), 72
WLAN.EXT_ANT (in module network), 72
WLAN.INT_ANT (in module network), 72
WLAN.STA (in module network), 72
WLAN.WEP (in module network), 72
WLAN.WPA (in module network), 72
WLAN.WPA2 (in module network), 72
write() (machine.I2C method), 60
write() (machine.SPI method), 58
write() (machine.UART method), 56
write() (usocket.socket method), 41
write_readinto() (machine.SPI method), 58
writeblocks() (uos.AbstractBlockDev method), 33
writeto() (machine.I2C method), 60
writeto_mem() (machine.I2C method), 61

Z

ZeroDivisionError, 26
zip() (built-in function), 26