MicroPython Documentation

Release 1.9.2

Damien P. George, Paul Sokolovsky, and contributors

August 23, 2017

CHAPTER

ONE

QUICK REFERENCE FOR THE PYBOARD

The below pinout is for PYBv1.0. You can also view pinouts for other versions of the pyboard: PYBv1.1 or PYBLITEv1.0-AC or PYBLITEv1.0.

1.1 General board control

See *pyb*.

```
import pyb

pyb.repl_uart(pyb.UART(1, 9600)) # duplicate REPL on UART(1)
pyb.wfi() # pause CPU, waiting for interrupt
pyb.freq() # get CPU and bus frequencies
pyb.freq(60000000) # set CPU freq to 60MHz
pyb.stop() # stop CPU, waiting for external interrupt
```

1.2 Delay and timing

Use the *time* module:

```
import time

time.sleep(1)           # sleep for 1 second
time.sleep_ms(500)      # sleep for 500 milliseconds
time.sleep_us(10)       # sleep for 10 microseconds
start = time.ticks_ms() # get value of millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start) # compute time difference
```

1.3 LEDs

See *pyb.LED*.

```
from pyb import LED

led = LED(1) # red led
led.toggle()
led.on()
led.off()
```

1.4 Pins and GPIO

See *pyb.Pin*.

```
from pyb import Pin

p_out = Pin('X1', Pin.OUT_PP)
p_out.high()
p_out.low()

p_in = Pin('X2', Pin.IN, Pin.PULL_UP)
p_in.value() # get value, 0 or 1
```

1.5 Servo control

See *pyb.Servo*.

```
from pyb import Servo

s1 = Servo(1) # servo on position 1 (X1, VIN, GND)
s1.angle(45) # move to 45 degrees
s1.angle(-60, 1500) # move to -60 degrees in 1500ms
s1.speed(50) # for continuous rotation servos
```

1.6 External interrupts

See *pyb.ExtInt*.

```
from pyb import Pin, ExtInt

callback = lambda e: print("intr")
ext = ExtInt(Pin('Y1'), ExtInt.IRQ_RISING, Pin.PULL_NONE, callback)
```

1.7 Timers

See *pyb.Timer*.

```
from pyb import Timer

tim = Timer(1, freq=1000)
tim.counter() # get counter value
tim.freq(0.5) # 0.5 Hz
tim.callback(lambda t: pyb.LED(1).toggle())
```

1.8 PWM (pulse width modulation)

See *pyb.Pin* and *pyb.Timer*.

```
from pyb import Pin, Timer

p = Pin('X1') # X1 has TIM2, CH1
tim = Timer(2, freq=1000)
ch = tim.channel(1, Timer.PWM, pin=p)
ch.pulse_width_percent(50)
```

1.9 ADC (analog to digital conversion)

See *pyb.Pin* and *pyb.ADC*.

```
from pyb import Pin, ADC

adc = ADC(Pin('X19'))
adc.read() # read value, 0-4095
```

1.10 DAC (digital to analog conversion)

See *pyb.Pin* and *pyb.DAC*.

```
from pyb import Pin, DAC

dac = DAC(Pin('X5'))
dac.write(120) # output between 0 and 255
```

1.11 UART (serial bus)

See *pyb.UART*.

```
from pyb import UART

uart = UART(1, 9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

1.12 SPI bus

See *pyb.SPI*.

```
from pyb import SPI

spi = SPI(1, SPI.MASTER, baudrate=200000, polarity=1, phase=0)
spi.send('hello')
spi.recv(5) # receive 5 bytes on the bus
spi.send_recv('hello') # send and receive 5 bytes
```

1.13 I2C bus

See *pyb.I2C*.

```
from pyb import I2C

i2c = I2C(1, I2C.MASTER, baudrate=100000)
i2c.scan() # returns list of slave addresses
i2c.send('hello', 0x42) # send 5 bytes to slave with address 0x42
i2c.recv(5, 0x42) # receive 5 bytes from slave
i2c.mem_read(2, 0x42, 0x10) # read 2 bytes from slave 0x42, slave memory 0x10
i2c.mem_write('xy', 0x42, 0x10) # write 2 bytes to slave 0x42, slave memory 0x10
```

CHAPTER

TWO

GENERAL INFORMATION ABOUT THE PYBOARD

Contents

2.1 Local filesystem and SD card

There is a small internal filesystem (a drive) on the pyboard, called `/flash`, which is stored within the microcontroller's flash memory. If a micro SD card is inserted into the slot, it is available as `/sd`.

When the pyboard boots up, it needs to choose a filesystem to boot from. If there is no SD card, then it uses the internal filesystem `/flash` as the boot filesystem, otherwise, it uses the SD card `/sd`. After the boot, the current directory is set to one of the directories above.

If needed, you can prevent the use of the SD card by creating an empty file called `/flash/SKIPSD`. If this file exists when the pyboard boots up then the SD card will be skipped and the pyboard will always boot from the internal filesystem (in this case the SD card won't be mounted but you can still mount and use it later in your program using `os.mount`).

(Note that on older versions of the board, `/flash` is called `0:/` and `/sd` is called `1:/`).

The boot filesystem is used for 2 things: it is the filesystem from which the `boot.py` and `main.py` files are searched for, and it is the filesystem which is made available on your PC over the USB cable.

The filesystem will be available as a USB flash drive on your PC. You can save files to the drive, and edit `boot.py` and `main.py`.

Remember to eject (on Linux, unmount) the USB drive before you reset your pyboard.

2.2 Boot modes

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

You can override this boot sequence by holding down the user switch as the board is booting up. Hold down user switch and press reset, and then as you continue to hold the user switch, the LEDs will count in binary. When the LEDs have reached the mode you want, let go of the user switch, the LEDs for the selected mode will flash quickly, and the board will boot.

The modes are:

1. Green LED only, *standard boot*: run `boot.py` then `main.py`.
2. Orange LED only, *safe boot*: don't run any scripts on boot-up.
3. Green and orange LED together, *filesystem reset*: resets the flash filesystem to its factory state, then boots in safe mode.

If your filesystem becomes corrupt, boot into mode 3 to fix it. If resetting the filesystem while plugged into your compute doesn't work, you can try doing the same procedure while the board is plugged into a USB charger, or other USB power supply without data connection.

2.3 Errors: flashing LEDs

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

1. **If the red and green LEDs flash alternatively, then a Python script** (eg `main.py`) has an error. Use the REPL to debug it.
2. If all 4 LEDs cycle on and off slowly, then there was a hard fault. This cannot be recovered from and you need to do a hard reset.

2.4 Guide for using the pyboard with Windows

The following PDF guide gives information about using the pyboard with Windows, including setting up the serial prompt and downloading new firmware using DFU programming: PDF guide.

2.5 The pyboard hardware

For the pyboard:

- PYBv1.0 schematics and layout (2.4MiB PDF)
- PYBv1.0 metric dimensions (360KiB PDF)
- PYBv1.0 imperial dimensions (360KiB PDF)

For the official skin modules:

- LCD32MKv1.0 schematics (194KiB PDF)
- AMPv1.0 schematics (209KiB PDF)
- LCD160CRv1.0: see *lcd160cr*

2.6 Datasheets for the components on the pyboard

- The microcontroller: STM32F405RGT6 (link to manufacturer's site)

- The accelerometer: Freescale MMA7660 (800kiB PDF)
- The LDO voltage regulator: Microchip MCP1802 (400kiB PDF)

2.7 Datasheets for other components

- The LCD display on the LCD touch-sensor skin: Newhaven Display NHD-C12832A1Z-FSW-FBW-3V3 (460KiB PDF)
- The touch sensor chip on the LCD touch-sensor skin: Freescale MPR121 (280KiB PDF)
- The digital potentiometer on the audio skin: Microchip MCP4541 (2.7MiB PDF)

CHAPTER

THREE

MICROPYTHON TUTORIAL FOR THE PYBOARD

This tutorial is intended to get you started with your pyboard. All you need is a pyboard and a micro-USB cable to connect it to your PC. If it is your first time, it is recommended to follow the tutorial through in the order below.

3.1 Introduction to the pyboard

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

3.1.1 Caring for your pyboard

Because the pyboard does not have a housing it needs a bit of care:

- Be gentle when plugging/unplugging the USB cable. Whilst the USB connector is soldered through the board and is relatively strong, if it breaks off it can be very difficult to fix.
- Static electricity can shock the components on the pyboard 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 pyboard. If your pyboard came in a black plastic box, then this box is the best way to store and carry the pyboard as it is an anti-static box (it is made of a conductive plastic, with conductive foam inside).

As long as you take care of the hardware, you should be okay. It's almost impossible to break the software on the pyboard, 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 reflash the MicroPython software, but that can be done over USB.

3.1.2 Layout of the pyboard

The micro USB connector is on the top right, the micro SD card slot on the top left of the board. There are 4 LEDs between the SD slot and USB connector. The colours are: red on the bottom, then green, orange, and blue on the top. There are 2 switches: the right one is the reset switch, the left is the user switch.

3.1.3 Plugging in and powering on

The pyboard can be powered via USB. Connect it to your PC via a micro USB cable. There is only one way that the cable will fit. Once connected, the green LED on the board should flash quickly.

3.1.4 Powering by an external power source

The pyboard 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 pyboard so you must be careful when connecting anything to VIN.

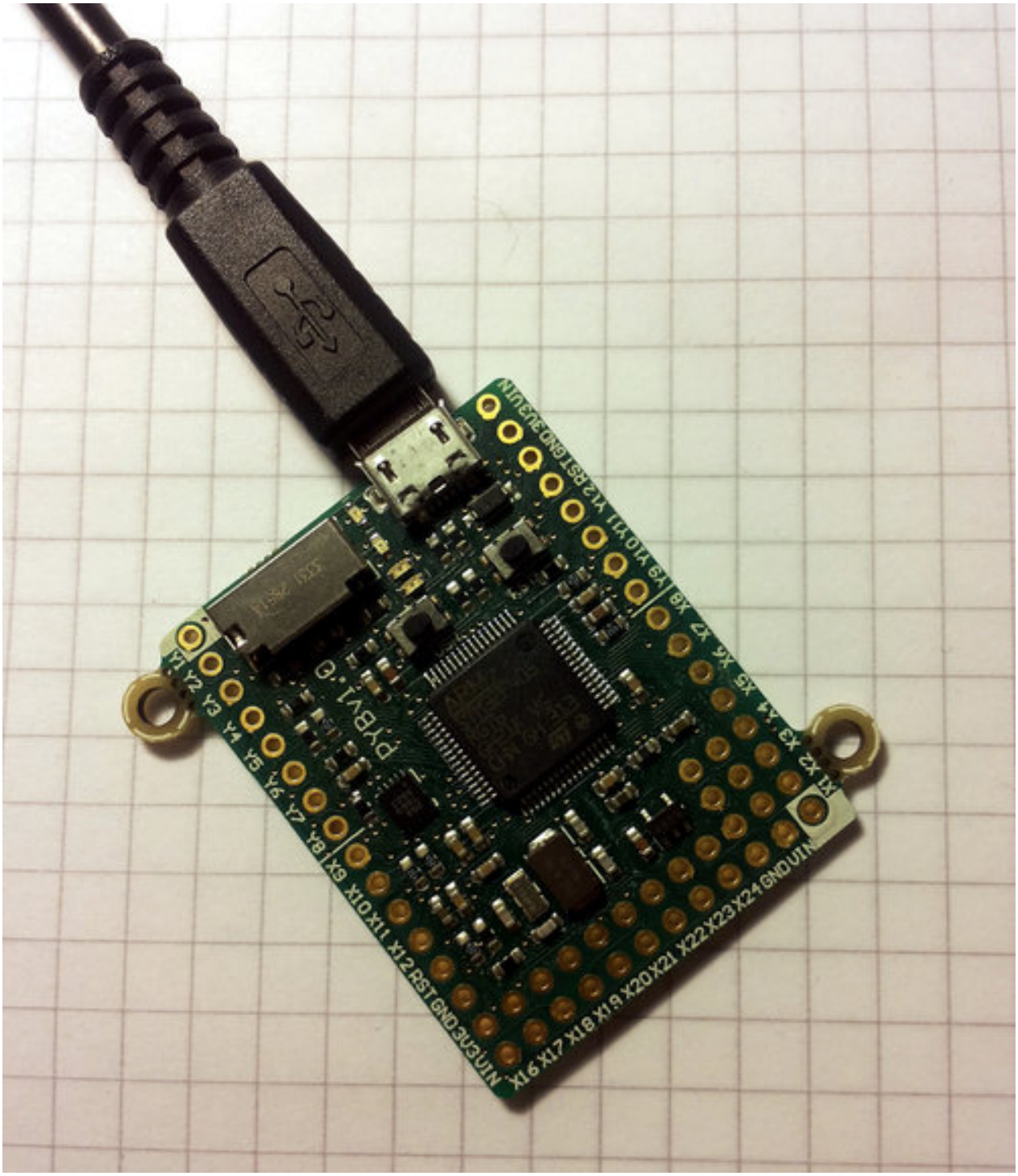
The input voltage must be between 3.6V and 10V.

3.2 Running your first script

Let's jump right in and get a Python script running on the pyboard. After all, that's what it's all about!

3.2.1 Connecting your pyboard

Connect your pyboard to your PC (Windows, Mac or Linux) with a micro USB cable. There is only one way that the cable will connect, so you can't get it wrong.



When the pyboard is connected to your PC it will power on and enter the start up process (the boot process). The green LED should light up for half a second or less, and when it turns off it means the boot process has completed.

3.2.2 Opening the pyboard USB drive

Your PC should now recognise the pyboard. It depends on the type of PC you have as to what happens next:

- **Windows**: Your pyboard will appear as a removable USB flash drive. Windows may automatically pop-up a

window, or you may need to go there using Explorer.

Windows will also see that the pyboard has a serial device, and it will try to automatically configure this device. If it does, cancel the process. We will get the serial device working in the next tutorial.

- **Mac**: Your pyboard will appear on the desktop as a removable disc. It will probably be called "NONAME". Click on it to open the pyboard folder.
- **Linux**: Your pyboard will appear as a removable medium. On Ubuntu it will mount automatically and pop-up a window with the pyboard folder. On other Linux distributions, the pyboard may be mounted automatically, or you may need to do it manually. At a terminal command line, type `lsblk` to see a list of connected drives, and then `mount /dev/sdb1` (replace `sdb1` with the appropriate device). You may need to be root to do this.

Okay, so you should now have the pyboard connected as a USB flash drive, and a window (or command line) should be showing the files on the pyboard drive.

The drive you are looking at is known as `/flash` by the pyboard, and should contain the following 4 files:

- **boot.py – this script is executed when the pyboard boots up. It sets** up various configuration options for the pyboard.
- **main.py – this is the main script that will contain your Python program.** It is executed after `boot.py`.
- **README.txt – this contains some very basic information about getting** started with the pyboard.
- **pybcdc.inf – this is a Windows driver file to configure the serial USB** device. More about this in the next tutorial.

3.2.3 Editing `main.py`

Now we are going to write our Python program, so open the `main.py` file in a text editor. On Windows you can use notepad, or any other editor. On Mac and Linux, use your favourite text editor. With the file open you will see it contains 1 line:

```
# main.py -- put your code here!
```

This line starts with a # character, which means that it is a *comment*. Such lines will not do anything, and are there for you to write notes about your program.

Let's add 2 lines to this `main.py` file, to make it look like this:

```
# main.py -- put your code here!
import pyb
pyb.LED(4).on()
```

The first line we wrote says that we want to use the `pyb` module. This module contains all the functions and classes to control the features of the pyboard.

The second line that we wrote turns the blue LED on: it first gets the `LED` class from the `pyb` module, creates LED number 4 (the blue LED), and then turns it on.

3.2.4 Resetting the pyboard

To run this little script, you need to first save and close the `main.py` file, and then eject (or unmount) the pyboard USB drive. Do this like you would a normal USB flash drive.

When the drive is safely ejected/unmounted you can get to the fun part: press the RST switch on the pyboard to reset and run your script. The RST switch is the small black button just below the USB connector on the board, on the right edge.

When you press RST the green LED will flash quickly, and then the blue LED should turn on and stay on.

Congratulations! You have written and run your very first MicroPython program!

3.3 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 pyboard. 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 serial USB device on the pyboard. How you do this depends on your operating system.

3.3.1 Windows

You need to install the pyboard driver to use the serial USB device. The driver is on the pyboard's USB flash drive, and is called `pybcdc.inf`.

To install this driver you need to go to Device Manager for your computer, find the pyboard in the list of devices (it should have a warning sign next to it because it's not working yet), right click on the pyboard device, select Properties, then Install Driver. You need to then select the option to find the driver manually (don't use Windows auto update), navigate to the pyboard's USB drive, and select that. It should then install. After installing, go back to the Device Manager to find the installed pyboard, and see which COM port it is (eg COM4). More comprehensive instructions can be found in the Guide for pyboard on Windows (PDF). Please consult this guide if you are having problems installing the driver.

You now need to run your terminal program. You can use HyperTerminal if you have it installed, or download the free program PuTTY: putty.exe. 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.

3.3.2 Mac OS X

Open a terminal and run:

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

When you are finished and want to exit screen, type CTRL-A CTRL-\.

3.3.3 Linux

Open a terminal and run:

```
screen /dev/ttyACM0
```

You can also try `picocom` or `minicom` instead of screen. You may have to use `/dev/ttyACM1` or a higher number for `ttyACM`. And, you may need to give yourself the correct permissions to access this devices (eg group `uucp` or `dialout`, or use sudo).

3.3.4 Using the REPL prompt

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

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 pyboard!")
hello pyboard!
```

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 pyboard!")` 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:

```
>>> pyb.LED(1).on()
>>> pyb.LED(2).on()
>>> 1 + 2
3
>>> 1 / 2
0.5
>>> 20 * 'py'
'pypypypypypypypypypypypypypypypypypypypypypypypypypypypypypypypypypypypy'
```

3.3.5 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: sync filesystems
PYB: soft reboot
Micro Python v1.0 on 2014-05-03; PYBv1.0 with STM32F405RG
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 closest to the micro-USB socket on the board). This will end your session, disconnecting whatever program (PuTTY, screen, etc) that you used to connect to the pyboard.

If you are going to do a hard-reset, it's recommended to first close your serial program and eject/unmount the pyboard drive.

3.4 Turning on LEDs and basic Python concepts

The easiest thing to do on the pyboard is to turn on the LEDs attached to the board. Connect the board, and log in as described in tutorial 1. We will start by turning and LED on in the interpreter, type the following

```
>>> myled = pyb.LED(1)
>>> myled.on()
>>> myled.off()
```

These commands turn the LED on and off.

This is all very well but we would like this process to be automated. Open the file MAIN.PY on the pyboard in your favourite text editor. Write or paste the following lines into the file. If you are new to python, then make sure you get the indentation correct since this matters!

```
led = pyb.LED(2)
while True:
    led.toggle()
    pyb.delay(1000)
```

When you save, the red light on the pyboard should turn on for about a second. To run the script, do a soft reset (CTRL-D). The pyboard will then restart and you should see a green light continuously flashing on and off. Success, the first step on your path to building an army of evil robots! When you are bored of the annoying flashing light then press CTRL-C at your terminal to stop it running.

So what does this code do? First we need some terminology. Python is an object-oriented language, almost everything in python is a *class* and when you create an instance of a class you get an *object*. Classes have *methods* associated to them. A method (also called a member function) is used to interact with or control the object.

The first line of code creates an LED object which we have then called led. When we create the object, it takes a single parameter which must be between 1 and 4, corresponding to the 4 LEDs on the board. The pyb.LED class has three important member functions that we will use: on(), off() and toggle(). The other function that we use is pyb.delay() this simply waits for a given time in miliseconds. Once we have created the LED object, the statement while True: creates an infinite loop which toggles the led between on and off and waits for 1 second.

Exercise: Try changing the time between toggling the led and turning on a different LED.

Exercise: Connect to the pyboard directly, create a pyb.LED object and turn it on using the on() method.

3.4.1 A Disco on your pyboard

So far we have only used a single LED but the pyboard has 4 available. Let's start by creating an object for each LED so we can control each of them. We do that by creating a list of LEDS with a list comprehension.

```
leds = [pyb.LED(i) for i in range(1,5)]
```

If you call pyb.LED() with a number that isn't 1,2,3,4 you will get an error message. Next we will set up an infinite loop that cycles through each of the LEDs turning them on and off.

```
n = 0
while True:
  n = (n + 1) % 4
  leds[n].toggle()
  pyb.delay(50)
```

Here, n keeps track of the current LED and every time the loop is executed we cycle to the next n (the % sign is a modulus operator that keeps n between 0 and 3.) Then we access the nth LED and toggle it. If you run this you should see each of the LEDs turning on then all turning off again in sequence.

One problem you might find is that if you stop the script and then start it again that the LEDs are stuck on from the previous run, ruining our carefully choreographed disco. We can fix this by turning all the LEDs off when we initialise the script and then using a try/finally block. When you press CTRL-C, MicroPython generates a VCPInterrupt exception. Exceptions normally mean something has gone wrong and you can use a try: command to "catch" an exception. In this case it is just the user interrupting the script, so we don't need to catch the error but just tell MicroPython what to do when we exit. The finally block does this, and we use it to make sure all the LEDs are off. The full code is:

```
leds = [pyb.LED(i) for i in range(1,5)]
for l in leds:
    l.off()

n = 0
try:
   while True:
      n = (n + 1) % 4
      leds[n].toggle()
      pyb.delay(50)
finally:
    for l in leds:
        l.off()
```

3.4.2 The Fourth Special LED

The blue LED is special. As well as turning it on and off, you can control the intensity using the intensity() method. This takes a number between 0 and 255 that determines how bright it is. The following script makes the blue LED gradually brighter then turns it off again.

```
led = pyb.LED(4)
intensity = 0
while True:
    intensity = (intensity + 1) % 255
    led.intensity(intensity)
    pyb.delay(20)
```

You can call intensity() on the other LEDs but they can only be off or on. 0 sets them off and any other number up to 255 turns them on.

3.5 The Switch, callbacks and interrupts

The pyboard has 2 small switches, labelled USR and RST. The RST switch is a hard-reset switch, and if you press it then it restarts the pyboard from scratch, equivalent to turning the power off then back on.

The USR switch is for general use, and is controlled via a Switch object. To make a switch object do:

```
>>> sw = pyb.Switch()
```

Remember that you may need to type `import pyb` if you get an error that the name `pyb` does not exist.

With the switch object you can get its status:

```
>>> sw.value()
False
```

This will print `False` if the switch is not held, or `True` if it is held. Try holding the USR switch down while running the above command.

There is also a shorthand notation to get the switch status, by "calling" the switch object:

```
>>> sw()
False
```

3.5.1 Switch callbacks

The switch is a very simple object, but it does have one advanced feature: the `sw.callback()` function. The callback function sets up something to run when the switch is pressed, and uses an interrupt. It's probably best to start with an example before understanding how interrupts work. Try running the following at the prompt:

```
>>> sw.callback(lambda:print('press!'))
```

This tells the switch to print `press!` each time the switch is pressed down. Go ahead and try it: press the USR switch and watch the output on your PC. Note that this print will interrupt anything you are typing, and is an example of an interrupt routine running asynchronously.

As another example try:

```
>>> sw.callback(lambda:pyb.LED(1).toggle())
```

This will toggle the red LED each time the switch is pressed. And it will even work while other code is running.

To disable the switch callback, pass `None` to the callback function:

```
>>> sw.callback(None)
```

You can pass any function (that takes zero arguments) to the switch callback. Above we used the `lambda` feature of Python to create an anonymous function on the fly. But we could equally do:

```
>>> def f():
...   pyb.LED(1).toggle()
...
>>> sw.callback(f)
```

This creates a function called `f` and assigns it to the switch callback. You can do things this way when your function is more complicated than a `lambda` will allow.

Note that your callback functions must not allocate any memory (for example they cannot create a tuple or list). Callback functions should be relatively simple. If you need to make a list, make it beforehand and store it in a global variable (or make it local and close over it). If you need to do a long, complicated calculation, then use the callback to set a flag which some other code then responds to.

3.5.2 Technical details of interrupts

Let's step through the details of what is happening with the switch callback. When you register a function with `sw.callback()`, the switch sets up an external interrupt trigger (falling edge) on the pin that the switch is connected to. This means that the microcontroller will listen on the pin for any changes, and the following will occur:

1. When the switch is pressed a change occurs on the pin (the pin goes from low to high), and the microcontroller registers this change.
2. The microcontroller finishes executing the current machine instruction, stops execution, and saves its current state (pushes the registers on the stack). This has the effect of pausing any code, for example your running Python script.
3. The microcontroller starts executing the special interrupt handler associated with the switch's external trigger. This interrupt handler get the function that you registered with `sw.callback()` and executes it.
4. Your callback function is executed until it finishes, returning control to the switch interrupt handler.
5. The switch interrupt handler returns, and the microcontroller is notified that the interrupt has been dealt with.
6. The microcontroller restores the state that it saved in step 2.

7. Execution continues of the code that was running at the beginning. Apart from the pause, this code does not notice that it was interrupted.

The above sequence of events gets a bit more complicated when multiple interrupts occur at the same time. In that case, the interrupt with the highest priority goes first, then the others in order of their priority. The switch interrupt is set at the lowest priority.

3.5.3 Further reading

For further information about using hardware interrupts see *writing interrupt handlers*.

3.6 The accelerometer

Here you will learn how to read the accelerometer and signal using LEDs states like tilt left and tilt right.

3.6.1 Using the accelerometer

The pyboard has an accelerometer (a tiny mass on a tiny spring) that can be used to detect the angle of the board and motion. There is a different sensor for each of the x, y, z directions. To get the value of the accelerometer, create a pyb.Accel() object and then call the x() method.

```
>>> accel = pyb.Accel()
>>> accel.x()
7
```

This returns a signed integer with a value between around -30 and 30. Note that the measurement is very noisy, this means that even if you keep the board perfectly still there will be some variation in the number that you measure. Because of this, you shouldn't use the exact value of the x() method but see if it is in a certain range.

We will start by using the accelerometer to turn on a light if it is not flat.

```
accel = pyb.Accel()
light = pyb.LED(3)
SENSITIVITY = 3

while True:
    x = accel.x()
    if abs(x) > SENSITIVITY:
        light.on()
    else:
        light.off()

    pyb.delay(100)
```

We create Accel and LED objects, then get the value of the x direction of the accelerometer. If the magnitude of x is bigger than a certain value `SENSITIVITY`, then the LED turns on, otherwise it turns off. The loop has a small `pyb.delay()` otherwise the LED flashes annoyingly when the value of x is close to `SENSITIVITY`. Try running this on the pyboard and tilt the board left and right to make the LED turn on and off.

Exercise: Change the above script so that the blue LED gets brighter the more you tilt the pyboard. HINT: You will need to rescale the values, intensity goes from 0-255.

3.6.2 Making a spirit level

The example above is only sensitive to the angle in the x direction but if we use the `y()` value and more LEDs we can turn the pyboard into a spirit level.

```
xlights = (pyb.LED(2), pyb.LED(3))
ylights = (pyb.LED(1), pyb.LED(4))

accel = pyb.Accel()
SENSITIVITY = 3

while True:
    x = accel.x()
    if x > SENSITIVITY:
        xlights[0].on()
        xlights[1].off()
    elif x < -SENSITIVITY:
        xlights[1].on()
        xlights[0].off()
    else:
        xlights[0].off()
        xlights[1].off()

    y = accel.y()
    if y > SENSITIVITY:
        ylights[0].on()
        ylights[1].off()
    elif y < -SENSITIVITY:
        ylights[1].on()
        ylights[0].off()
    else:
        ylights[0].off()
        ylights[1].off()

    pyb.delay(100)
```

We start by creating a tuple of LED objects for the x and y directions. Tuples are immutable objects in python which means they can't be modified once they are created. We then proceed as before but turn on a different LED for positive and negative x values. We then do the same for the y direction. This isn't particularly sophisticated but it does the job. Run this on your pyboard and you should see different LEDs turning on depending on how you tilt the board.

3.7 Safe mode and factory reset

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

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

If you have problems with the filesystem you can do a factory reset, which restores the filesystem to its original state.

3.7.1 Safe mode

To enter safe mode, do the following steps:

1. Connect the pyboard to USB so it powers up.

2. Hold down the USR switch.
3. While still holding down USR, press and release the RST switch.
4. The LEDs will then cycle green to orange to green+orange and back again.
5. Keep holding down USR until *only the orange LED is lit*, and then let go of the USR switch.
6. The orange LED should flash quickly 4 times, and then turn off.
7. You are now in safe mode.

In safe mode, the `boot.py` and `main.py` files are not executed, and so the pyboard boots up with default settings. This means you now have access to the filesystem (the USB drive should appear), 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 pyboard.

3.7.2 Factory reset the filesystem

If you pyboard's filesystem gets corrupted (for example, you forgot to eject/unmount it), or you have some code in `boot.py` or `main.py` which you can't escape from, then you can reset the filesystem.

Resetting the filesystem deletes all files on the internal pyboard storage (not the SD card), and restores the files `boot.py`, `main.py`, `README.txt` and `pybcdc.inf` back to their original state.

To do a factory reset of the filesystem you follow a similar procedure as you did to enter safe mode, but release USR on green+orange:

1. Connect the pyboard to USB so it powers up.
2. Hold down the USR switch.
3. While still holding down USR, press and release the RST switch.
4. The LEDs will then cycle green to orange to green+orange and back again.
5. Keep holding down USR until *both the green and orange LEDs are lit*, and then let go of the USR switch.
6. The green and orange LEDs should flash quickly 4 times.
7. The red LED will turn on (so red, green and orange are now on).
8. The pyboard is now resetting the filesystem (this takes a few seconds).
9. The LEDs all turn off.
10. You now have a reset filesystem, and are in safe mode.
11. Press and release the RST switch to boot normally.

3.8 Making the pyboard act as a USB mouse

The pyboard is a USB device, and can configured to act as a mouse instead of the default USB flash drive.

To do this we must first edit the `boot.py` file to change the USB configuration. If you have not yet touched your `boot.py` file then it will look something like this:

```
# boot.py -- run on boot-up
# can run arbitrary Python, but best to keep it minimal

import pyb
```

```
#pyb.main('main.py') # main script to run after this one
#pyb.usb_mode('VCP+MSC') # act as a serial and a storage device
#pyb.usb_mode('VCP+HID') # act as a serial device and a mouse
```

To enable the mouse mode, uncomment the last line of the file, to make it look like:

```
pyb.usb_mode('VCP+HID') # act as a serial device and a mouse
```

If you already changed your `boot.py` file, then the minimum code it needs to work is:

```
import pyb
pyb.usb_mode('VCP+HID')
```

This tells the pyboard to configure itself as a VCP (Virtual COM Port, ie serial port) and HID (human interface device, in our case a mouse) USB device when it boots up.

Eject/unmount the pyboard drive and reset it using the RST switch. Your PC should now detect the pyboard as a mouse!

3.8.1 Sending mouse events by hand

To get the py-mouse to do anything we need to send mouse events to the PC. We will first do this manually using the REPL prompt. Connect to your pyboard using your serial program and type the following:

```
>>> hid = pyb.USB_HID()
>>> hid.send((0, 10, 0, 0))
```

Your mouse should move 10 pixels to the right! In the command above you are sending 4 pieces of information: button status, x, y and scroll. The number 10 is telling the PC that the mouse moved 10 pixels in the x direction.

Let's make the mouse oscillate left and right:

```
>>> import math
>>> def osc(n, d):
...   for i in range(n):
...     hid.send((0, int(20 * math.sin(i / 10)), 0, 0))
...     pyb.delay(d)
...
>>> osc(100, 50)
```

The first argument to the function `osc` is the number of mouse events to send, and the second argument is the delay (in milliseconds) between events. Try playing around with different numbers.

Exercise: make the mouse go around in a circle.

3.8.2 Making a mouse with the accelerometer

Now lets make the mouse move based on the angle of the pyboard, using the accelerometer. The following code can be typed directly at the REPL prompt, or put in the `main.py` file. Here, we'll put in in `main.py` because to do that we will learn how to go into safe mode.

At the moment the pyboard is acting as a serial USB device and an HID (a mouse). So you cannot access the filesystem to edit your `main.py` file.

You also can't edit your `boot.py` to get out of HID-mode and back to normal mode with a USB drive...

To get around this we need to go into *safe mode*. This was described in the [safe mode tutorial](tut-reset), but we repeat the instructions here:

1. Hold down the USR switch.
2. While still holding down USR, press and release the RST switch.
3. The LEDs will then cycle green to orange to green+orange and back again.
4. Keep holding down USR until *only the orange LED is lit*, and then let go of the USR switch.
5. The orange LED should flash quickly 4 times, and then turn off.
6. You are now in safe mode.

In safe mode, the `boot.py` and `main.py` files are not executed, and so the pyboard boots up with default settings. This means you now have access to the filesystem (the USB drive should appear), and you can edit `main.py`. (Leave `boot.py` as-is, because we still want to go back to HID-mode after we finish editing `main.py`.)

In `main.py` put the following code:

```
import pyb

switch = pyb.Switch()
accel = pyb.Accel()
hid = pyb.USB_HID()

while not switch():
    hid.send((0, accel.x(), accel.y(), 0))
    pyb.delay(20)
```

Save your file, eject/unmount your pyboard drive, and reset it using the RST switch. It should now act as a mouse, and the angle of the board will move the mouse around. Try it out, and see if you can make the mouse stand still!

Press the USR switch to stop the mouse motion.

You'll note that the y-axis is inverted. That's easy to fix: just put a minus sign in front of the y-coordinate in the `hid.send()` line above.

3.8.3 Restoring your pyboard to normal

If you leave your pyboard as-is, it'll behave as a mouse everytime you plug it in. You probably want to change it back to normal. To do this you need to first enter safe mode (see above), and then edit the `boot.py` file. In the `boot.py` file, comment out (put a # in front of) the line with the `VCP+HID` setting, so it looks like:

```
#pyb.usb_mode('VCP+HID') # act as a serial device and a mouse
```

Save your file, eject/unmount the drive, and reset the pyboard. It is now back to normal operating mode.

3.9 The Timers

The pyboard has 14 timers which each consist of an independent counter running at a user-defined frequency. They can be set up to run a function at specific intervals. The 14 timers are numbered 1 through 14, but 3 is reserved for internal use, and 5 and 6 are used for servo and ADC/DAC control. Avoid using these timers if possible.

Let's create a timer object:

```
>>> tim = pyb.Timer(4)
```

Now let's see what we just created:

```
>>> tim
Timer(4)
```

The pyboard is telling us that `tim` is attached to timer number 4, but it's not yet initialised. So let's initialise it to trigger at 10 Hz (that's 10 times per second):

```
>>> tim.init(freq=10)
```

Now that it's initialised, we can see some information about the timer:

```
>>> tim
Timer(4, prescaler=624, period=13439, mode=UP, div=1)
```

The information means that this timer is set to run at the peripheral clock speed divided by 624+1, and it will count from 0 up to 13439, at which point it triggers an interrupt, and then starts counting again from 0. These numbers are set to make the timer trigger at 10 Hz: the source frequency of the timer is 84MHz (found by running `tim.source_freq()`) so we get 84MHz / 625 / 13440 = 10Hz.

3.9.1 Timer counter

So what can we do with our timer? The most basic thing is to get the current value of its counter:

```
>>> tim.counter()
21504
```

This counter will continuously change, and counts up.

3.9.2 Timer callbacks

The next thing we can do is register a callback function for the timer to execute when it triggers (see the [switch tutorial](tut-switch) for an introduction to callback functions):

```
>>> tim.callback(lambda t:pyb.LED(1).toggle())
```

This should start the red LED flashing right away. It will be flashing at 5 Hz (2 toggle's are needed for 1 flash, so toggling at 10 Hz makes it flash at 5 Hz). You can change the frequency by re-initialising the timer:

```
>>> tim.init(freq=20)
```

You can disable the callback by passing it the value `None`:

```
>>> tim.callback(None)
```

The function that you pass to callback must take 1 argument, which is the timer object that triggered. This allows you to control the timer from within the callback function.

We can create 2 timers and run them independently:

```
>>> tim4 = pyb.Timer(4, freq=10)
>>> tim7 = pyb.Timer(7, freq=20)
>>> tim4.callback(lambda t: pyb.LED(1).toggle())
>>> tim7.callback(lambda t: pyb.LED(2).toggle())
```

Because the callbacks are proper hardware interrupts, we can continue to use the pyboard for other things while these timers are running.

3.9.3 Making a microsecond counter

You can use a timer to create a microsecond counter, which might be useful when you are doing something which requires accurate timing. We will use timer 2 for this, since timer 2 has a 32-bit counter (so does timer 5, but if you use timer 5 then you can't use the Servo driver at the same time).

We set up timer 2 as follows:

```
>>> micros = pyb.Timer(2, prescaler=83, period=0x3fffffff)
```

The prescaler is set at 83, which makes this timer count at 1 MHz. This is because the CPU clock, running at 168 MHz, is divided by 2 and then by prescaler+1, giving a frequency of 168 MHz/2/(83+1)=1 MHz for timer 2. The period is set to a large number so that the timer can count up to a large number before wrapping back around to zero. In this case it will take about 17 minutes before it cycles back to zero.

To use this timer, it's best to first reset it to 0:

```
>>> micros.counter(0)
```

and then perform your timing:

```
>>> start_micros = micros.counter()

... do some stuff ...

>>> end_micros = micros.counter()
```

3.10 Inline assembler

Here you will learn how to write inline assembler in MicroPython.

Note: this is an advanced tutorial, intended for those who already know a bit about microcontrollers and assembly language.

MicroPython includes an inline assembler. It allows you to write assembly routines as a Python function, and you can call them as you would a normal Python function.

3.10.1 Returning a value

Inline assembler functions are denoted by a special function decorator. Let's start with the simplest example:

```
@micropython.asm_thumb
def fun():
    movw(r0, 42)
```

You can enter this in a script or at the REPL. This function takes no arguments and returns the number 42. `r0` is a register, and the value in this register when the function returns is the value that is returned. MicroPython always interprets the `r0` as an integer, and converts it to an integer object for the caller.

If you run `print(fun())` you will see it print out 42.

3.10.2 Accessing peripherals

For something a bit more complicated, let's turn on an LED:

```
@micropython.asm_thumb
def led_on():
    movwt(r0, stm.GPIOA)
    movw(r1, 1 << 13)
    strh(r1, [r0, stm.GPIO_BSRRL])
```

This code uses a few new concepts:

- `stm` is a module which provides a set of constants for easy access to the registers of the pyboard's microcontroller. Try running `import stm` and then `help(stm)` at the REPL. It will give you a list of all the available constants.
- `stm.GPIOA` is the address in memory of the GPIOA peripheral. On the pyboard, the red LED is on port A, pin PA13.
- `movwt` moves a 32-bit number into a register. It is a convenience function that turns into 2 thumb instructions: `movw` followed by `movt`. The `movt` also shifts the immediate value right by 16 bits.
- `strh` stores a half-word (16 bits). The instruction above stores the lower 16-bits of `r1` into the memory location `r0 + stm.GPIO_BSRRL`. This has the effect of setting high all those pins on port A for which the corresponding bit in `r0` is set. In our example above, the 13th bit in `r0` is set, so PA13 is pulled high. This turns on the red LED.

3.10.3 Accepting arguments

Inline assembler functions can accept up to 4 arguments. If they are used, they must be named `r0`, `r1`, `r2` and `r3` to reflect the registers and the calling conventions.

Here is a function that adds its arguments:

```
@micropython.asm_thumb
def asm_add(r0, r1):
    add(r0, r0, r1)
```

This performs the computation `r0 = r0 + r1`. Since the result is put in `r0`, that is what is returned. Try `asm_add(1, 2)`, it should return 3.

3.10.4 Loops

We can assign labels with `label(my_label)`, and branch to them using `b(my_label)`, or a conditional branch like `bgt(my_label)`.

The following example flashes the green LED. It flashes it `r0` times.

```
@micropython.asm_thumb
def flash_led(r0):
    # get the GPIOA address in r1
    movwt(r1, stm.GPIOA)

    # get the bit mask for PA14 (the pin LED #2 is on)
    movw(r2, 1 << 14)

    b(loop_entry)

    label(loop1)

    # turn LED on
```

```
    strh(r2, [r1, stm.GPIO_BSRRL])

    # delay for a bit
    movwt(r4, 5599900)
    label(delay_on)
    sub(r4, r4, 1)
    cmp(r4, 0)
    bgt(delay_on)

    # turn LED off
    strh(r2, [r1, stm.GPIO_BSRRH])

    # delay for a bit
    movwt(r4, 5599900)
    label(delay_off)
    sub(r4, r4, 1)
    cmp(r4, 0)
    bgt(delay_off)

    # loop r0 times
    sub(r0, r0, 1)
    label(loop_entry)
    cmp(r0, 0)
    bgt(loop1)
```

3.10.5 Further reading

For further information about supported instructions of the inline assembler, see the *reference documentation*.

3.11 Power control

pyb.wfi() is used to reduce power consumption while waiting for an event such as an interrupt. You would use it in the following situation:

```
while True:
    do_some_processing()
    pyb.wfi()
```

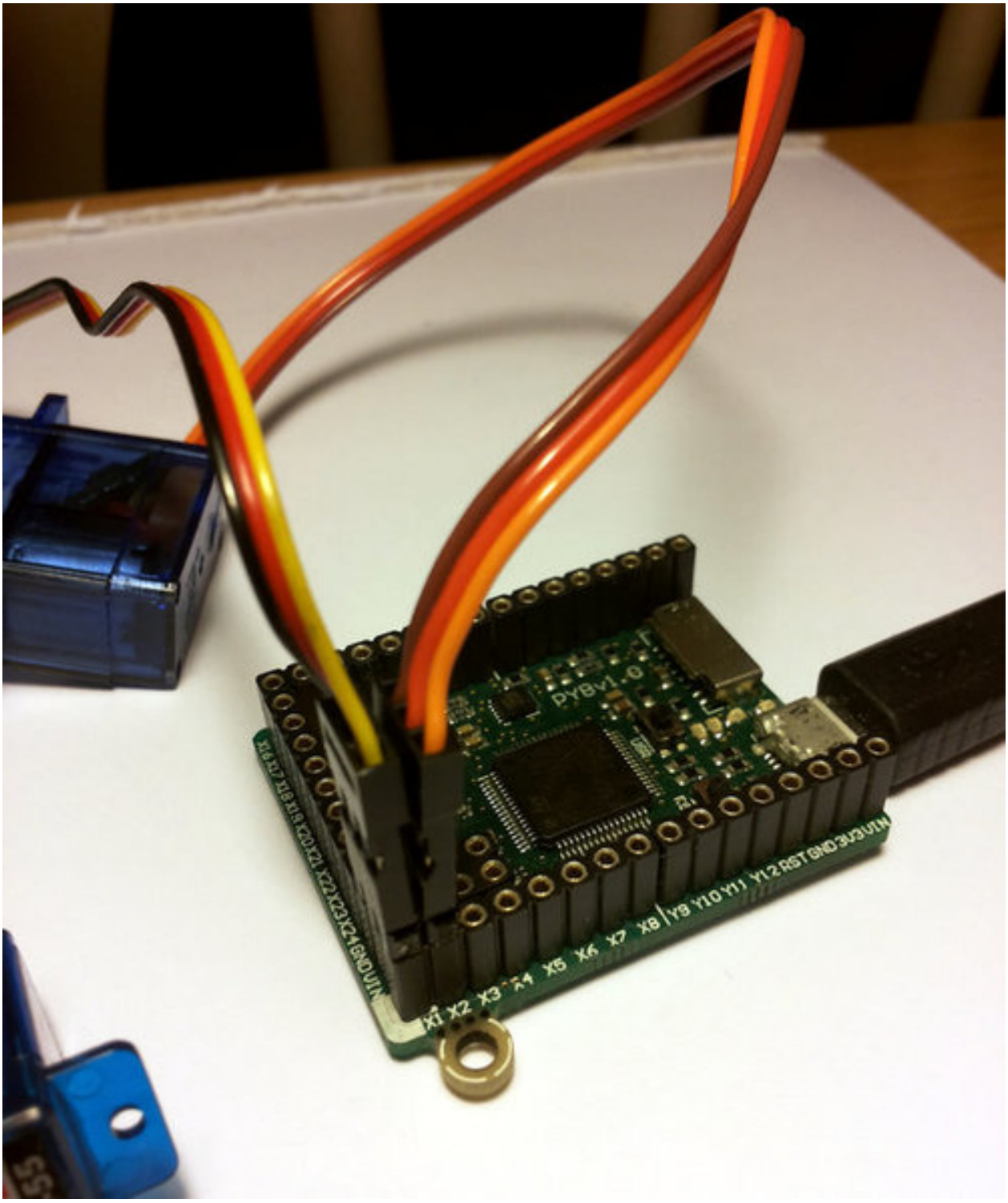
Control the frequency using *pyb.freq()*:

```
pyb.freq(30000000) # set CPU frequency to 30MHz
```

3.12 Tutorials requiring extra components

3.12.1 Controlling hobby servo motors

There are 4 dedicated connection points on the pyboard for connecting up hobby servo motors (see eg [Wikipedia](http://en.wikipedia.org/wiki/Servo_%28radio_control%29)). These motors have 3 wires: ground, power and signal. On the pyboard you can connect them in the bottom right corner, with the signal pin on the far right. Pins X1, X2, X3 and X4 are the 4 dedicated servo signal pins.



In this picture there are male-male double adaptors to connect the servos to the header pins on the pyboard.

The ground wire on a servo is usually the darkest coloured one, either black or dark brown. The power wire will most likely be red.

The power pin for the servos (labelled VIN) is connected directly to the input power source of the pyboard. When powered via USB, VIN is powered through a diode by the 5V USB power line. Connect to USB, the pyboard can power at least 4 small to medium sized servo motors.

If using a battery to power the pyboard and run servo motors, make sure it is not greater than 6V, since this is the maximum voltage most servo motors can take. (Some motors take only up to 4.8V, so check what type you are using.)

Creating a Servo object

Plug in a servo to position 1 (the one with pin X1) and create a servo object using:

```
>>> servo1 = pyb.Servo(1)
```

To change the angle of the servo use the `angle` method:

```
>>> servo1.angle(45)
>>> servo1.angle(-60)
```

The angle here is measured in degrees, and ranges from about -90 to +90, depending on the motor. Calling `angle` without parameters will return the current angle:

```
>>> servo1.angle()
-60
```

Note that for some angles, the returned angle is not exactly the same as the angle you set, due to rounding errors in setting the pulse width.

You can pass a second parameter to the `angle` method, which specifies how long to take (in milliseconds) to reach the desired angle. For example, to take 1 second (1000 milliseconds) to go from the current position to 50 degrees, use

```
>>> servo1.angle(50, 1000)
```

This command will return straight away and the servo will continue to move to the desired angle, and stop when it gets there. You can use this feature as a speed control, or to synchronise 2 or more servo motors. If we have another servo motor (`servo2 = pyb.Servo(2)`) then we can do

```
>>> servo1.angle(-45, 2000); servo2.angle(60, 2000)
```

This will move the servos together, making them both take 2 seconds to reach their final angles.

Note: the semicolon between the 2 expressions above is used so that they are executed one after the other when you press enter at the REPL prompt. In a script you don't need to do this, you can just write them one line after the other.

Continuous rotation servos

So far we have been using standard servos that move to a specific angle and stay at that angle. These servo motors are useful to create joints of a robot, or things like pan-tilt mechanisms. Internally, the motor has a variable resistor (potentiometer) which measures the current angle and applies power to the motor proportional to how far it is from the desired angle. The desired angle is set by the width of a high-pulse on the servo signal wire. A pulse width of 1500 microsecond corresponds to the centre position (0 degrees). The pulses are sent at 50 Hz, ie 50 pulses per second.

You can also get **continuous rotation** servo motors which turn continuously clockwise or counterclockwise. The direction and speed of rotation is set by the pulse width on the signal wire. A pulse width of 1500 microseconds corresponds to a stopped motor. A pulse width smaller or larger than this means rotate one way or the other, at a given speed.

On the pyboard, the servo object for a continuous rotation motor is the same as before. In fact, using `angle` you can set the speed. But to make it easier to understand what is intended, there is another method called `speed` which sets the speed:

```
>>> servo1.speed(30)
```

`speed` has the same functionality as `angle`: you can get the speed, set it, and set it with a time to reach the final speed.

```
>>> servo1.speed()
30
>>> servo1.speed(-20)
>>> servo1.speed(0, 2000)
```

The final command above will set the motor to stop, but take 2 seconds to do it. This is essentially a control over the acceleration of the continuous servo.

A servo speed of 100 (or -100) is considered maximum speed, but actually you can go a bit faster than that, depending on the particular motor.

The only difference between the `angle` and `speed` methods (apart from the name) is the way the input numbers (angle or speed) are converted to a pulse width.

Calibration

The conversion from angle or speed to pulse width is done by the servo object using its calibration values. To get the current calibration, use

```
>>> servo1.calibration()
(640, 2420, 1500, 2470, 2200)
```

There are 5 numbers here, which have meaning:

1. Minimum pulse width; the smallest pulse width that the servo accepts.
2. Maximum pulse width; the largest pulse width that the servo accepts.
3. Centre pulse width; the pulse width that puts the servo at 0 degrees or 0 speed.
4. The pulse width corresponding to 90 degrees. This sets the conversion in the method `angle` of angle to pulse width.
5. The pulse width corresponding to a speed of 100. This sets the conversion in the method `speed` of speed to pulse width.

You can recalibrate the servo (change its default values) by using:

```
>>> servo1.calibration(700, 2400, 1510, 2500, 2000)
```

Of course, you would change the above values to suit your particular servo motor.

3.12.2 Fading LEDs

In addition to turning LEDs on and off, it is also possible to control the brightness of an LED using Pulse-Width Modulation (PWM), a common technique for obtaining variable output from a digital pin. This allows us to fade an LED:

Components

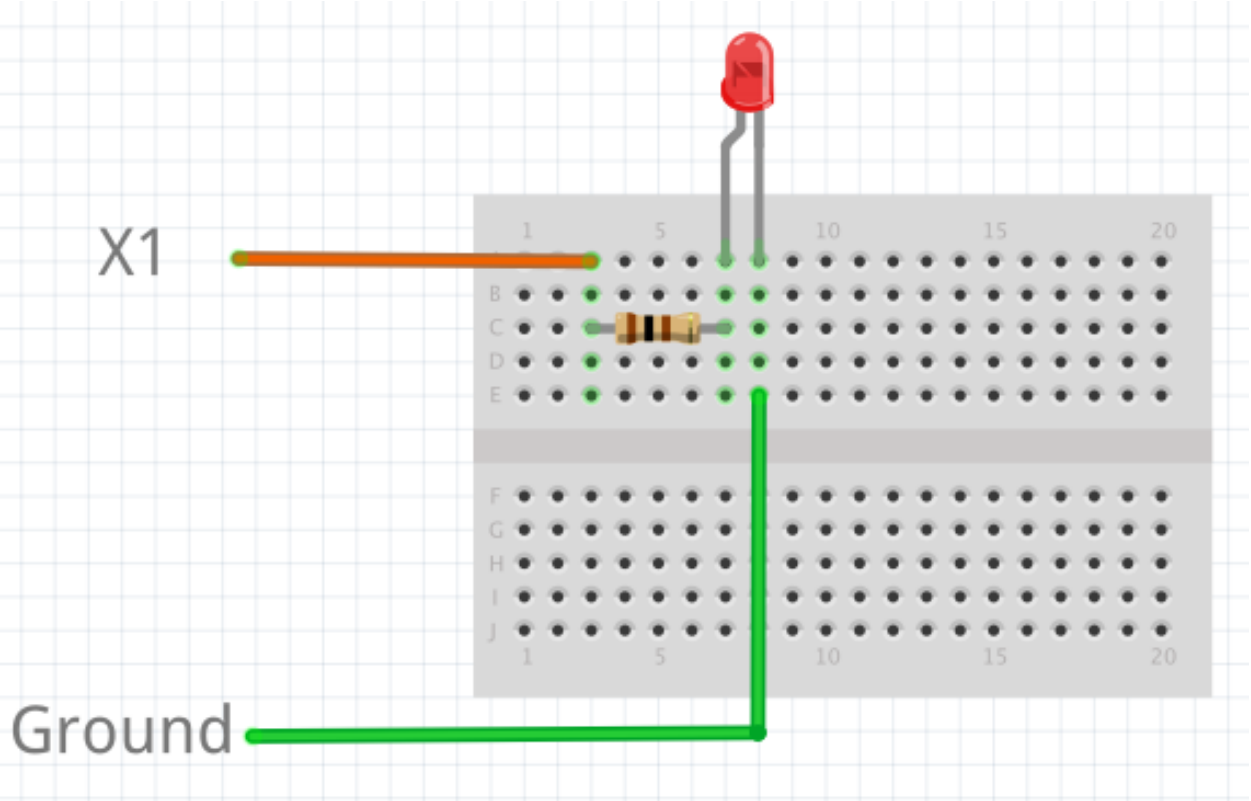
You will need:

- Standard 5 or 3 mm LED
- 100 Ohm resistor

- Wires
- Breadboard (optional, but makes things easier)

Connecting Things Up

For this tutorial, we will use the `X1` pin. Connect one end of the resistor to `X1`, and the other end to the **anode** of the LED, which is the longer leg. Connect the **cathode** of the LED to ground.



Code

By examining the *Quick reference for the pyboard*, we see that `X1` is connected to channel 1 of timer 5 (`TIM5 CH1`). Therefore we will first create a `Timer` object for timer 5, then create a `TimerChannel` object for channel 1:

```
from pyb import Timer
from time import sleep

# timer 5 will be created with a frequency of 100 Hz
tim = pyb.Timer(5, freq=100)
tchannel = tim.channel(1, Timer.PWM, pin=pyb.Pin.board.X1, pulse_width=0)
```

Brightness of the LED in PWM is controlled by controlling the pulse-width, that is the amount of time the LED is on every cycle. With a timer frequency of 100 Hz, each cycle takes 0.01 second, or 10 ms.

To achieve the fading effect shown at the beginning of this tutorial, we want to set the pulse-width to a small value, then slowly increase the pulse-width to brighten the LED, and start over when we reach some maximum brightness:

```
# maximum and minimum pulse-width, which corresponds to maximum
# and minimum brightness
max_width = 200000
```

```
min_width = 20000

# how much to change the pulse-width by each step
wstep = 1500
cur_width = min_width

while True:
  tchannel.pulse_width(cur_width)

  # this determines how often we change the pulse-width. It is
  # analogous to frames-per-second
  sleep(0.01)

  cur_width += wstep

  if cur_width > max_width:
    cur_width = min_width
```

Breathing Effect

If we want to have a breathing effect, where the LED fades from dim to bright then bright to dim, then we simply need to reverse the sign of `wstep` when we reach maximum brightness, and reverse it again at minimum brightness. To do this we modify the `while` loop to be:

```
while True:
  tchannel.pulse_width(cur_width)

  sleep(0.01)

  cur_width += wstep

  if cur_width > max_width:
    cur_width = max_width
    wstep *= -1
  elif cur_width < min_width:
    cur_width = min_width
    wstep *= -1
```

Advanced Exercise

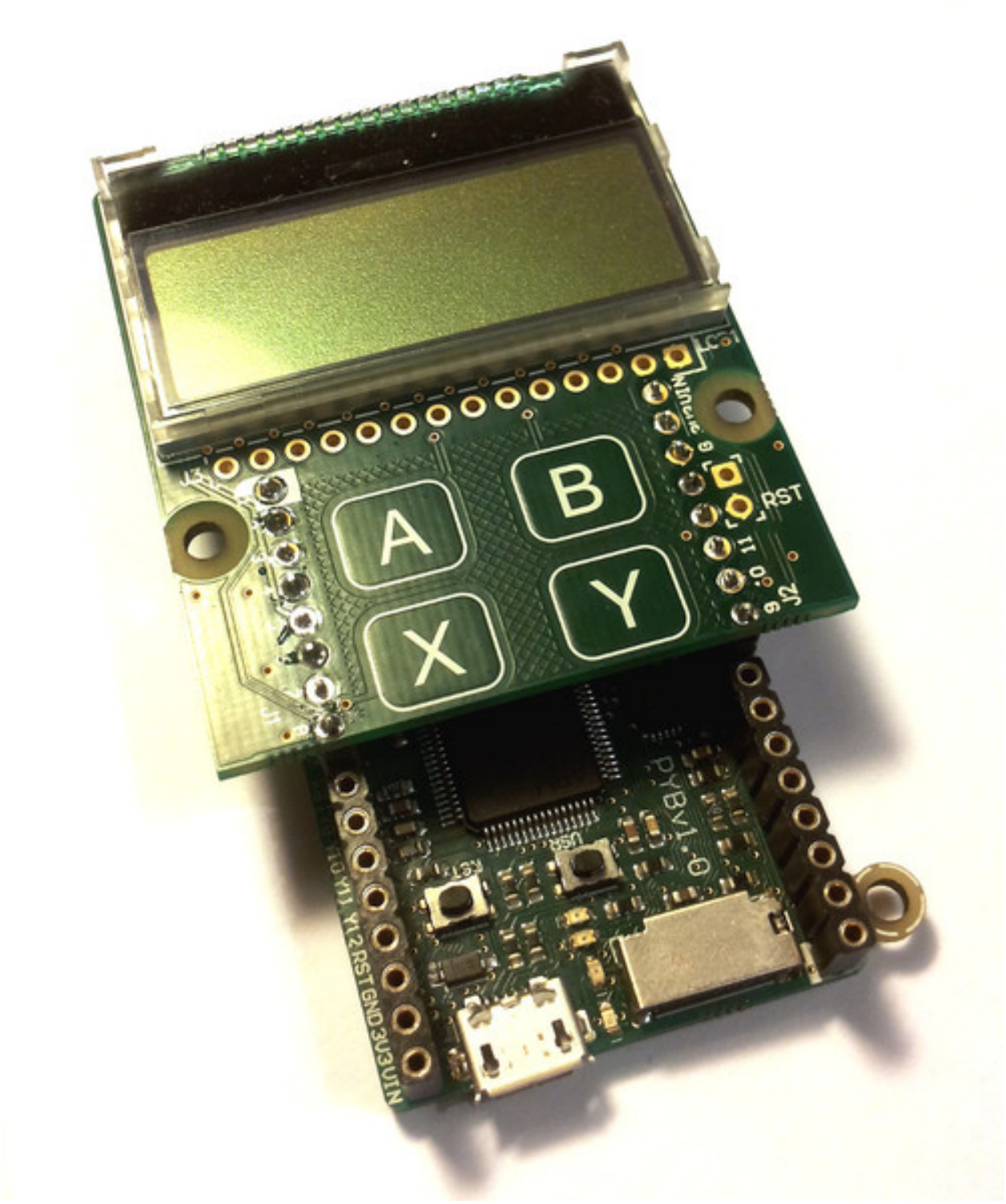
You may have noticed that the LED brightness seems to fade slowly, but increases quickly. This is because our eyes interprets brightness logarithmically (Weber's Law), while the LED's brightness changes linearly, that is by the same amount each time. How do you solve this problem? (Hint: what is the opposite of the logarithmic function?)

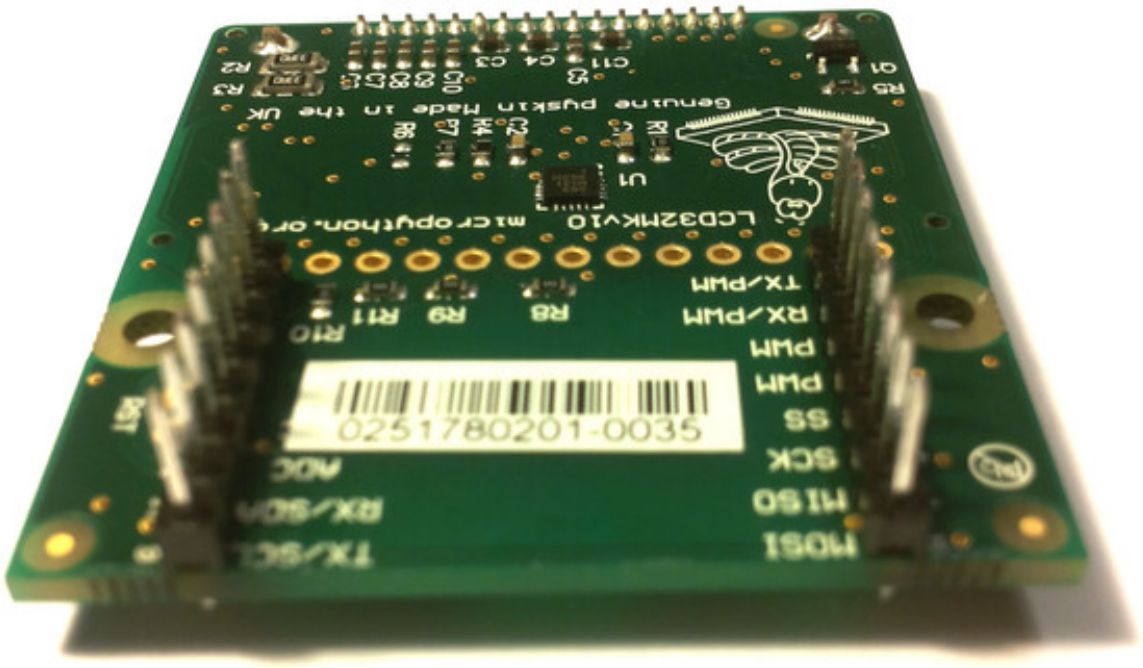
Addendum

We could have also used the digital-to-analog converter (DAC) to achieve the same effect. The PWM method has the advantage that it drives the LED with the same current each time, but for different lengths of time. This allows better control over the brightness, because LEDs do not necessarily exhibit a linear relationship between the driving current and brightness.

3.12.3 The LCD and touch-sensor skin

Soldering and using the LCD and touch-sensor skin.





The following video shows how to solder the headers onto the LCD skin. At the end of the video, it shows you how to correctly connect the LCD skin to the pyboard.

For circuit schematics and datasheets for the components on the skin see hardware_index.

Using the LCD

To get started using the LCD, try the following at the MicroPython prompt. Make sure the LCD skin is attached to the pyboard as pictured at the top of this page.

```
>>> import pyb
>>> lcd = pyb.LCD('X')
>>> lcd.light(True)
>>> lcd.write('Hello uPy!\n')
```

You can make a simple animation using the code:

```
import pyb
lcd = pyb.LCD('X')
lcd.light(True)
for x in range(-80, 128):
    lcd.fill(0)
    lcd.text('Hello uPy!', x, 10, 1)
    lcd.show()
    pyb.delay(25)
```

Using the touch sensor

To read the touch-sensor data you need to use the I2C bus. The MPR121 capacitive touch sensor has address 90.

To get started, try:

```
>>> import pyb
>>> i2c = pyb.I2C(1, pyb.I2C.MASTER)
>>> i2c.mem_write(4, 90, 0x5e)
>>> touch = i2c.mem_read(1, 90, 0)[0]
```

The first line above makes an I2C object, and the second line enables the 4 touch sensors. The third line reads the touch status and the `touch` variable holds the state of the 4 touch buttons (A, B, X, Y).

There is a simple driver here which allows you to set the threshold and debounce parameters, and easily read the touch status and electrode voltage levels. Copy this script to your pyboard (either flash or SD card, in the top directory or `lib/` directory) and then try:

```
>>> import pyb
>>> import mpr121
>>> m = mpr121.MPR121(pyb.I2C(1, pyb.I2C.MASTER))
>>> for i in range(100):
...   print(m.touch_status())
...   pyb.delay(100)
...
```

This will continuously print out the touch status of all electrodes. Try touching each one in turn.

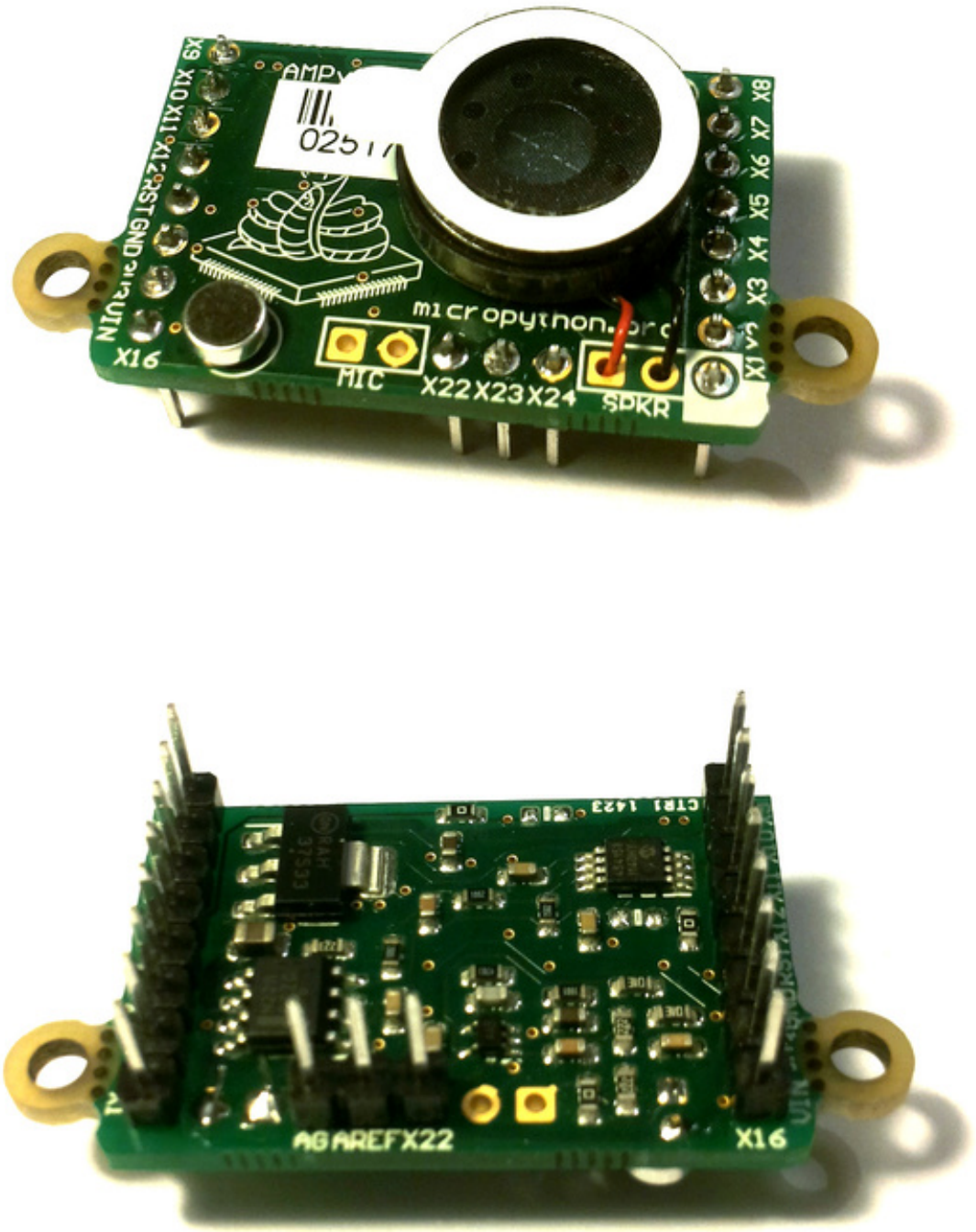
Note that if you put the LCD skin in the Y-position, then you need to initialise the I2C bus using:

```
>>> m = mpr121.MPR121(pyb.I2C(2, pyb.I2C.MASTER))
```

There is also a demo which uses the LCD and the touch sensors together, and can be found here.

3.12.4 The AMP audio skin

Soldering and using the AMP audio skin.



The following video shows how to solder the headers, microphone and speaker onto the AMP skin.

For circuit schematics and datasheets for the components on the skin see hardware_index.

Example code

The AMP skin has a speaker which is connected to `DAC(1)` via a small power amplifier. The volume of the amplifier is controlled by a digital potentiometer, which is an I2C device with address 46 on the `IC2(1)` bus.

To set the volume, define the following function:

```
import pyb
def volume(val):
    pyb.I2C(1, pyb.I2C.MASTER).mem_write(val, 46, 0)
```

Then you can do:

```
>>> volume(0)    # minimum volume
>>> volume(127)  # maximum volume
```

To play a sound, use the `write_timed` method of the `DAC` object. For example:

```
import math
from pyb import DAC

# create a buffer containing a sine-wave
buf = bytearray(100)
for i in range(len(buf)):
    buf[i] = 128 + int(127 * math.sin(2 * math.pi * i / len(buf)))

# output the sine-wave at 400Hz
dac = DAC(1)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

You can also play WAV files using the Python `wave` module. You can get the wave module here and you will also need the chunk module available here. Put these on your pyboard (either on the flash or the SD card in the top-level directory). You will need an 8-bit WAV file to play, such as this one, or to convert any file you have with the command:

```
avconv -i original.wav -ar 22050 -codec pcm_u8 test.wav
```

Then you can do:

```
>>> import wave
>>> from pyb import DAC
>>> dac = DAC(1)
>>> f = wave.open('test.wav')
>>> dac.write_timed(f.readframes(f.getnframes()), f.getframerate())
```

This should play the WAV file. Note that this will read the whole file into RAM so it has to be small enough to fit in it.

To play larger wave files you will have to use the micro-SD card to store it. Also the file must be read and sent to the DAC in small chunks that will fit the RAM limit of the microcontroller. Here is an example function that can play 8-bit wave files with up to 16kHz sampling:

```
import wave
from pyb import DAC
from pyb import delay
dac = DAC(1)

def play(filename):
    f = wave.open(filename, 'r')
    total_frames = f.getnframes()
    framerate = f.getframerate()
```

```
    for position in range(0, total_frames, framerate):
        f.setpos(position)
        dac.write_timed(f.readframes(framerate), framerate)
        delay(1000)
```

This function reads one second worth of data and sends it to DAC. It then waits one second and moves the file cursor to the new position to read the next second of data in the next iteration of the for-loop. It plays one second of audio at a time every one second.

3.12.5 The LCD160CR skin

This tutorial shows how to get started using the LCD160CR skin.

For detailed documentation of the driver for the display see the *lcd160cr* module.

Plugging in the display

The display can be plugged directly into a pyboard (all pyboard versions are supported). You plug the display onto the top of the pyboard either in the X or Y positions. The display should cover half of the pyboard. See the picture above for how to achieve this; the left half of the picture shows the X position, and the right half shows the Y position.

Getting the driver

You can control the display directly using a power/enable pin and an I2C bus, but it is much more convenient to use the driver provided by the *lcd160cr* module. This driver is included in recent version of the pyboard firmware (see here). You can also find the driver in the GitHub repository here, and to use this version you will need to copy the file to your board, into a directory that is searched by import (usually the lib/ directory).

Once you have the driver installed you need to import it to use it:

```
import lcd160cr
```

Testing the display

There is a test program which you can use to test the features of the display, and which also serves as a basis to start creating your own code that uses the LCD. This test program is included in recent versions of the pyboard firmware and is also available on GitHub here.

To run the test from the MicroPython prompt do:

```
>>> import lcd160cr_test
```

It will then print some brief instructions. You will need to know which position your display is connected to (X or Y) and then you can run (assuming you have the display on position X):

```
>>> test_all('X')
```

Drawing some graphics

You must first create an LCD160CR object which will control the display. Do this using:

```
>>> import lcd160cr
>>> lcd = lcd160cr.LCD160CR('X')
```

This assumes your display is connected in the X position. If it's in the Y position then use `lcd = lcd160cr.LCD160CR('Y')` instead.

To erase the screen and draw a line, try:

```
>>> lcd.set_pen(lcd.rgb(255, 0, 0), lcd.rgb(64, 64, 128))
>>> lcd.erase()
>>> lcd.line(10, 10, 50, 80)
```

The next example draws random rectangles on the screen. You can copy-and-paste it into the MicroPython prompt by first pressing "Ctrl-E" at the prompt, then "Ctrl-D" once you have pasted the text.

```
from random import randint
for i in range(1000):
    fg = lcd.rgb(randint(128, 255), randint(128, 255), randint(128, 255))
    bg = lcd.rgb(randint(0, 128), randint(0, 128), randint(0, 128))
    lcd.set_pen(fg, bg)
    lcd.rect(randint(0, lcd.w), randint(0, lcd.h), randint(10, 40), randint(10, 40))
```

Using the touch sensor

The display includes a resistive touch sensor that can report the position (in pixels) of a single force-based touch on the screen. To see if there is a touch on the screen use:

```
>>> lcd.is_touched()
```

This will return either `False` or `True`. Run the above command while touching the screen to see the result.

To get the location of the touch you can use the method:

```
>>> lcd.get_touch()
```

This will return a 3-tuple, with the first entry being 0 or 1 depending on whether there is currently anything touching the screen (1 if there is), and the second and third entries in the tuple being the x and y coordinates of the current (or most recent) touch.

Directing the MicroPython output to the display

The display supports input from a UART and implements basic VT100 commands, which means it can be used as a simple, general purpose terminal. Let's set up the pyboard to redirect its output to the display.

First you need to create a UART object:

```
>>> import pyb
>>> uart = pyb.UART('XA', 115200)
```

This assumes your display is connected to position X. If it's on position Y then use `uart = pyb.UART('YA', 115200)` instead.

Now, connect the REPL output to this UART:

```
>>> pyb.repl_uart(uart)
```

From now on anything you type at the MicroPython prompt, and any output you receive, will appear on the display.

No set-up commands are required for this mode to work and you can use the display to monitor the output of any UART, not just from the pyboard. All that is needed is for the display to have power, ground and the power/enable pin driven high. Then any characters on the display's UART input will be printed to the screen. You can adjust the UART baudrate from the default of 115200 using the *set_uart_baudrate* method.

3.13 Tips, tricks and useful things to know

3.13.1 Debouncing a pin input

A pin used as input from a switch or other mechanical device can have a lot of noise on it, rapidly changing from low to high when the switch is first pressed or released. This noise can be eliminated using a capacitor (a debouncing circuit). It can also be eliminated using a simple function that makes sure the value on the pin is stable.

The following function does just this. It gets the current value of the given pin, and then waits for the value to change. The new pin value must be stable for a continuous 20ms for it to register the change. You can adjust this time (to say 50ms) if you still have noise.

```
import pyb

def wait_pin_change(pin):
    # wait for pin to change value
    # it needs to be stable for a continuous 20ms
    cur_value = pin.value()
    active = 0
    while active < 20:
        if pin.value() != cur_value:
            active += 1
        else:
            active = 0
        pyb.delay(1)
```

Use it something like this:

```
import pyb

pin_x1 = pyb.Pin('X1', pyb.Pin.IN, pyb.Pin.PULL_DOWN)
while True:
    wait_pin_change(pin_x1)
    pyb.LED(4).toggle()
```

3.13.2 Making a UART - USB pass through

It's as simple as:

```
import pyb
import select

def pass_through(usb, uart):
    usb.setinterrupt(-1)
    while True:
        select.select([usb, uart], [], [])
        if usb.any():
            uart.write(usb.read(256))
        if uart.any():
            usb.write(uart.read(256))

pass_through(pyb.USB_VCP(), pyb.UART(1, 9600))
```

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.

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

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

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

Beyond the built-in libraries described in this documentation, many more modules from the Python standard library, as well as further MicroPython extensions to it, can be found in *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 package path. For example, `import json` will first search for a file `json.py` or directory `json` and load that package if it is found. If nothing is found, it will fallback to loading the built-in `ujson` module.

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

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

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

4.1.3 cmath – mathematical functions for complex numbers

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

The cmath module provides some basic mathematical functions for working with complex numbers.

Availability: not available on WiPy and ESP8266. Floating point support required for this module.

Functions

cmath.**cos**(*z*)
Return the cosine of z.

cmath.**exp**(*z*)
Return the exponential of z.

cmath.**log**(*z*)
Return the natural logarithm of z. The branch cut is along the negative real axis.

cmath.**log10**(*z*)
Return the base-10 logarithm of z. The branch cut is along the negative real axis.

cmath.**phase**(*z*)
Returns the phase of the number z, in the range (-pi, +pi].

cmath.**polar**(*z*)
Returns, as a tuple, the polar form of z.

cmath.**rect**(*r*, *phi*)
Returns the complex number with modulus r and phase phi.

cmath.**sin**(*z*)
Return the sine of z.

cmath.**sqrt**(*z*)
Return the square-root of z.

Constants

cmath.**e**
base of the natural logarithm

cmath.**pi**
the ratio of a circle's circumference to its diameter

4.1.4 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.

4.1.5 math – mathematical functions

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

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

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

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

Functions

`math.`**`acos`**(*x*)
Return the inverse cosine of `x`.

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

`math.`**`asin`**(*x*)
Return the inverse sine of `x`.

`math.`**`asinh`**(*x*)
Return the inverse hyperbolic sine of `x`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Constants

`math.`**`e`**
base of the natural logarithm

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

4.1.6 `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.

4.1.7 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.

4.1.8 ucollections – collection and container types

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

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

Classes

ucollections.**namedtuple**(*name*, *fields*)
This is factory function to create a new namedtuple type with a specific name and set of fields. A namedtuple is a subclass of tuple which allows to access its fields not just by numeric index, but also with an attribute access syntax using symbolic field names. Fields is a sequence of strings specifying field names. For compatibility with CPython it can also be a a string with space-separated field named (but this is less efficient). Example of use:

```
from ucollections import namedtuple

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

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

```
from ucollections import OrderedDict

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

```
d["b"] = 3
for k, v in d.items():
    print(k, v)
```

Output:

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

4.1.9 uerrno – system error codes

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

This module provides access to symbolic error codes for *OSError* exception. A particular inventory of codes depends on *MicroPython port*.

Constants

EEXIST, EAGAIN, etc.
Error codes, based on ANSI C/POSIX standard. All error codes start with "E". As mentioned above, inventory of the codes depends on *MicroPython port*. Errors are usually accessible as `exc.args[0]` where `exc` is an instance of *OSError*. Usage example:

```
try:
    uos.mkdir("my_dir")
except OSError as exc:
    if exc.args[0] == uerrno.EEXIST:
        print("Directory already exists")
```

uerrno.**errorcode**
Dictionary mapping numeric error codes to strings with symbolic error code (see above):

```
>>> print(uerrno.errorcode[uerrno.EEXIST])
EEXIST
```

4.1.10 uhashlib – hashing algorithms

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

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

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

Constructors

class uhashlib.**sha256**([*data*])
Create an SHA256 hasher object and optionally feed `data` into it.

class uhashlib.**sha1**([*data*])
Create an SHA1 hasher object and optionally feed `data` into it.

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

Methods

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

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

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

4.1.11 uheapq – heap queue algorithm

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

This module implements the heap queue algorithm.

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

Functions

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

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

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

4.1.12 uio – input/output streams

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

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

Conceptual hierarchy

Difference to CPython

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

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

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

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

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

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

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

Functions

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

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

Classes

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

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

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

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

getvalue()
Get the current contents of the underlying buffer which holds data.

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

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

4.1.14 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 `urandom` function.

Functions

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

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

uos.**ilistdir**([*dir*])
This function returns an iterator which then yields 3-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 3-tuples have the form *(name, type, inode)*:

- *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.

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

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

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

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

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

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

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

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

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

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

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

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

uos.**dupterm**(*stream_object*)
Duplicate or switch MicroPython terminal (the REPL) on the passed stream-like object. The given object

must implement the `readinto()` and `write()` methods. If `None` is passed, previously set redirection is cancelled.

4.1.15 `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.

`'^'`

`'$'`

`'?'`

`'*'`

`'+'`

`'??'`

`'*?'`

`'+?'`

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

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

Functions

`ure.`**`compile`**`(`*regex_str*`)`
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.

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.

4.1.16 `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 *obj* for polling. *eventmask* is logical OR of:

- `select.POLLIN` - data available for reading
- `select.POLLOUT` - more data can be written
- `select.POLLERR` - error occurred
- `select.POLLHUP` - end of stream/connection termination detected

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

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

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

`poll.poll`([*timeout*])
Wait for at least one of the registered objects to become ready. Returns list of (`obj`, `event`, ...) tuples, `event` element specifies which events happened with a stream and is a combination of `select.POLL*` constants described above. There may be other elements in tuple, depending on a platform and version, so don't assume that its size is 2. In case of timeout, an empty list is returned.

Timeout is in milliseconds.

Difference to CPython

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

`poll.ipoll`([*timeout*])
Like *poll.poll()*, but instead returns an iterator which yields callee-owned tuples. This function provides efficient, allocation-free way to poll on streams.

Difference to CPython

This function is a MicroPython extension.

4.1.17 `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.

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

Difference to CPython

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

Constants

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

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

usocket.**IPPROTO_UDP**

usocket.**IPPROTO_TCP**
IP protocol numbers.

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

Difference to CPython

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

---

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

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

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

socket.**makefile**(*mode='rb'*, *buffering=0*)
Return a file object associated with the socket. The exact returned type depends on the arguments given to makefile(). The support is limited to binary modes only ('rb', 'wb', and 'rwb'). CPython's arguments: *encoding*, *errors* and *newline* are not supported.

Difference to CPython

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

Difference to CPython

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

---

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

socket.**readinto**(*buf*[, *nbytes*])
Read bytes into the *buf*. If *nbytes* is specified then read at most that many bytes. Otherwise, read at most *len(buf)* bytes. Just as *read()*, this method follows "no short reads" policy.

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

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

Return value: the line read.

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

Return value: number of bytes written.

exception `socket.`**`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.

4.1.18 `ustruct` – pack and unpack primitive data types

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

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

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

Functions

`ustruct.`**`calcsize`**`(`*fmt*`)`
Return the number of bytes needed to store the given *fmt*.

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

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

`ustruct.`**`unpack`**`(`*fmt*, *data*`)`
Unpack from the *data* according to the format string *fmt*. The return value is a tuple of the unpacked values.

`ustruct.`**`unpack_from`**`(`*fmt*, *data*, *offset=0*`)`
Unpack from the *data* starting at *offset* according to the format string *fmt*. *offset* may be negative to count from the end of *buffer*. The return value is a tuple of the unpacked values.

4.1.19 `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(now, scheduled_time) > 0:
    print("Too early, let's nap")
    sleep_ms(ticks_diff(now, scheduled_time))
    task.run()
elif ticks_diff(now, scheduled_time) == 0:
    print("Right at time!")
    task.run()
elif ticks_diff(now, scheduled_time) < 0:
    print("Oops, running late, tell task to run faster!")
    task.run(run_faster=true)
```

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

utime.**time**()

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

Difference to CPython

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

4.1.20 uzlib – zlib decompression

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

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

Functions

uzlib.**decompress**(*data*, *wbits=0*, *bufsize=0*)
Return decompressed *data* as bytes. *wbits* is DEFLATE dictionary window size used during compression (8-15, the dictionary size is power of 2 of that value). Additionally, if value is positive, *data* is assumed to be zlib stream (with zlib header). Otherwise, if it's negative, it's assumed to be raw DEFLATE stream. *bufsize* parameter is for compatibility with CPython and is ignored.

class uzlib.**DecompIO**(*stream*, *wbits=0*)
Create a stream wrapper which allows transparent decompression of compressed data in another *stream*. This allows to process compressed streams with data larger than available heap size. In addition to values described in *decompress()*, *wbits* may take values 24..31 (16 + 8..15), meaning that input stream has gzip header.

Difference to CPython

This class is MicroPython extension. It's included on provisional basis and may be changed considerably or removed in later versions.

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*, *cachesize=0*, *pagesize=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.
- *cachesize* - Suggested maximum memory cache size in bytes. For a board with enough memory using larger values may improve performance. The value is only a recommendation, the module may use more memory if values set too low.

- *pagesize* - Page size used for the nodes in BTree. Acceptable range is 512-65536. If 0, underlying I/O block size will be used (the best compromise between memory usage and performance).
- *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; valid values are `framebuf.MVLSB`, `framebuf.RGB565` and `framebuf.GS4_HMSB`. MVLSB is monochrome 1-bit color, RGB565 is RGB 16-bit color, and GS4_HMSB is grayscale 4-bit color. Where a color value c is passed to a method, c is a small integer with an encoding that is dependent on the format of the FrameBuffer.
- *stride* is the number of pixels between each horizontal line of pixels in the FrameBuffer. This defaults to *width* but may need adjustments when implementing a FrameBuffer within another larger FrameBuffer or screen. The *buffer* size must accommodate an increased step size.

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

Drawing primitive shapes

The following methods draw shapes onto the FrameBuffer.

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

Fill the entire FrameBuffer with the specified color.

FrameBuffer.**pixel**(*x*, *y*[, *c*])

If *c* is not given, get the color value of the specified pixel. If *c* is given, set the specified pixel to the given color.

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

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

FrameBuffer.**line**(*x1*, *y1*, *x2*, *y2*, *c*)

Draw a line from a set of coordinates using the given color and a thickness of 1 pixel. The *line* method draws the line up to a second set of coordinates whereas the *hline* and *vline* methods draw horizontal and vertical lines respectively up to a given length.

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

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

Drawing text

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

Other methods

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

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

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

Constants

framebuf.**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.**GS4_HMSB**
Grayscale (4-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.

Miscellaneous functions

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

machine.**time_pulse_us**(*pin*, *pulse_level*, *timeout_us=1000000*)
Time a pulse on the given *pin*, and return the duration of the pulse in microseconds. The *pulse_level* argument should be 0 to time a low pulse or 1 to time a high pulse.

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

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

Constants

machine.**IDLE**
machine.**SLEEP**
machine.**DEEPSLEEP**
IRQ wake values.

machine.**PWRON_RESET**
machine.**HARD_RESET**
machine.**WDT_RESET**
machine.**DEEPSLEEP_RESET**
machine.**SOFT_RESET**
Reset causes.

machine.**WLAN_WAKE**
machine.**PIN_WAKE**
machine.**RTC_WAKE**
Wake-up reasons.

Classes

class 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 Signal – control and sense external I/O devices

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

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

- Use Signal: If you want to control a simple on/off (including software PWM!) devices like LEDs, multi-segment indicators, relays, buzzers, or read simple binary sensors, like normally open or normally closed buttons, pulled high or low, Reed switches, moisture/flame detectors, etc. etc. Summing up, if you have a real physical device/sensor requiring GPIO access, you likely should use a Signal.
- Use Pin: If you implement a higher-level protocol or bus to communicate with more complex devices.

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

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

Constructors

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

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

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

The arguments are:

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

Methods

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

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

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

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

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

`Signal.`**`on`**()

Activate signal.

`Signal.`**`off`**()

Deactivate signal.

class 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.**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.

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.`**`deinit`**`()`

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

Constants

`Timer.`**`ONE_SHOT`**

`Timer.`**`PERIODIC`**

Timer operating mode.

class WDT – watchdog timer

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

Example usage:

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

Availability of this class: pyboard, WiPy.

Constructors

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

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

Methods

`wdt.`**`feed`**`()`

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

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.

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.

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 socket 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 different ports of MicroPython 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 by BSSID (MAC address) instead of access point name

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**()
Return detailed status of the interface, values are dependent on the network medium/technology.

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 and status values. 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'))
# Extended status information also available this way
print(sta.config('rssi'))
```

class CC3K

This class provides a driver for CC3000 WiFi modules. Example usage:

```
import network
nic = network.CC3K(pyb.SPI(2), pyb.Pin.board.Y5, pyb.Pin.board.Y4, pyb.Pin.board.Y3)
nic.connect('your-ssid', 'your-password')
```

```
while not nic.isconnected():
    pyb.delay(50)
print(nic.ifconfig())

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

For this example to work the CC3000 module must have the following connections:

- MOSI connected to Y8
- MISO connected to Y7
- CLK connected to Y6
- CS connected to Y5
- VBEN connected to Y4
- IRQ connected to Y3

It is possible to use other SPI busses and other pins for CS, VBEN and IRQ.

Constructors

class network.**CC3K**(*spi*, *pin_cs*, *pin_en*, *pin_irq*)

Create a CC3K driver object, initialise the CC3000 module using the given SPI bus and pins, and return the CC3K object.

Arguments are:

- *spi* is an *SPI object* which is the SPI bus that the CC3000 is connected to (the MOSI, MISO and CLK pins).
- *pin_cs* is a *Pin object* which is connected to the CC3000 CS pin.
- *pin_en* is a *Pin object* which is connected to the CC3000 VBEN pin.
- *pin_irq* is a *Pin object* which is connected to the CC3000 IRQ pin.

All of these objects will be initialised by the driver, so there is no need to initialise them yourself. For example, you can use:

```
nic = network.CC3K(pyb.SPI(2), pyb.Pin.board.Y5, pyb.Pin.board.Y4, pyb.Pin.board.Y3)
```

Methods

cc3k.**connect**(*ssid*, *key=None*, ***, *security=WPA2*, *bssid=None*)

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

cc3k.**disconnect**()

Disconnect from the WiFi access point.

cc3k.**isconnected**()

Returns True if connected to a WiFi access point and has a valid IP address, False otherwise.

cc3k.**ifconfig**()

Returns a 7-tuple with (ip, subnet mask, gateway, DNS server, DHCP server, MAC address, SSID).

cc3k.**patch_version**()

Return the version of the patch program (firmware) on the CC3000.

cc3k.**patch_program**(*'pgm'*)
Upload the current firmware to the CC3000. You must pass 'pgm' as the first argument in order for the upload to proceed.

Constants

CC3K.**WEP**

CC3K.**WPA**

CC3K.**WPA2**
security type to use

class WIZNET5K

This class allows you to control WIZnet5x00 Ethernet adaptors based on the W5200 and W5500 chipsets (only W5200 tested).

Example usage:

```
import network
nic = network.WIZNET5K(pyb.SPI(1), pyb.Pin.board.X5, pyb.Pin.board.X4)
print(nic.ifconfig())

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

For this example to work the WIZnet5x00 module must have the following connections:

- MOSI connected to X8
- MISO connected to X7
- SCLK connected to X6
- nSS connected to X5
- nRESET connected to X4

It is possible to use other SPI busses and other pins for nSS and nRESET.

Constructors

class network.**WIZNET5K**(*spi*, *pin_cs*, *pin_rst*)
Create a WIZNET5K driver object, initialise the WIZnet5x00 module using the given SPI bus and pins, and return the WIZNET5K object.

Arguments are:

- *spi* is an *SPI object* which is the SPI bus that the WIZnet5x00 is connected to (the MOSI, MISO and SCLK pins).
- *pin_cs* is a *Pin object* which is connected to the WIZnet5x00 nSS pin.
- *pin_rst* is a *Pin object* which is connected to the WIZnet5x00 nRESET pin.

All of these objects will be initialised by the driver, so there is no need to initialise them yourself. For example, you can use:

```
    nic = network.WIZNET5K(pyb.SPI(1), pyb.Pin.board.X5, pyb.Pin.board.X4)
```

Methods

wiznet5k.**ifconfig**([*ip*, *subnet*, *gateway*, *dns*])
Get/set IP address, subnet mask, gateway and DNS.

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'))
```

wiznet5k.**regs**()
Dump the WIZnet5x00 registers. Useful for debugging.

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 the access it using familiar dot-syntax to reference sub-fields.

See also:

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

Defining structure layout

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

Following are encoding examples for various field types:

- Scalar types:

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

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

- Recursive structures:

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

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

- Arrays of primitive types:

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

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

- Arrays of aggregate types:

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

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

- Pointer to a primitive type:

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

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

- Pointer to an aggregate type:

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

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

- Bitfields:

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

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

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

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

Module contents

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

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

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

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

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

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

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

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

Structure descriptors and instantiating structure objects

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

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

Structure objects

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

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

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

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

Limitations

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

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

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

4.3 Libraries specific to the pyboard

The following libraries are specific to the pyboard.

4.3.1 pyb — functions related to the board

The pyb module contains specific functions related to the board.

Time related functions

pyb.**delay**(*ms*)
Delay for the given number of milliseconds.

pyb.**udelay**(*us*)
Delay for the given number of microseconds.

pyb.**millis**()
Returns the number of milliseconds since the board was last reset.

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

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

pyb.**micros**()
Returns the number of microseconds since the board was last reset.

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

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

pyb.**elapsed_millis**(*start*)
Returns the number of milliseconds which have elapsed since start.

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

Example:

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

pyb.**elapsed_micros**(*start*)
Returns the number of microseconds which have elapsed since start.

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

Example:

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

Reset related functions

pyb.**hard_reset**()
Resets the pyboard in a manner similar to pushing the external RESET button.

pyb.**bootloader**()
Activate the bootloader without BOOT* pins.

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

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

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

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

Interrupt related functions

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

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

Power related functions

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

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

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

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

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

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

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

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

pyb.**wfi**()
Wait for an internal or external interrupt.

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

pyb.**stop**()
Put the pyboard in a "sleeping" state.

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

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

pyb.**standby**()
Put the pyboard into a "deep sleep" state.

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

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

Miscellaneous functions

pyb.**have_cdc**()
Return True if USB is connected as a serial device, False otherwise.

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

---

pyb.**hid**((*buttons*, *x*, *y*, *z)*)
Takes a 4-tuple (or list) and sends it to the USB host (the PC) to signal a HID mouse-motion event.

Note: This function is deprecated. Use *pyb.USB_HID.send()* instead.

---

pyb.**info**([*dump_alloc_table*])
Print out lots of information about the board.

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

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

pyb.**mount**(*device*, *mountpoint*, *, *readonly=False*, *mkfs=False*)
Mount a block device and make it available as part of the filesystem. `device` must be an object that provides the block protocol:

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

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

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

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

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

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

To unmount a device, pass `None` as the device and the mount location as `mountpoint`.

pyb.**repl_uart**(*uart*)
Get or set the UART object where the REPL is repeated on.

pyb.**rng**()
Return a 30-bit hardware generated random number.

pyb.**sync**()
Sync all file systems.

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

pyb.**usb_mode**([*modestr*], *vid=0xf055*, *pid=0x9801*, *hid=pyb.hid_mouse*)
If called with no arguments, return the current USB mode as a string.

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

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

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

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

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

Classes

class Accel – accelerometer control

Accel is an object that controls the accelerometer. Example usage:

```
accel = pyb.Accel()
for i in range(10):
    print(accel.x(), accel.y(), accel.z())
```

Raw values are between -32 and 31.

Constructors

class pyb.**Accel**
Create and return an accelerometer object.

Methods

Accel.**filtered_xyz**()
Get a 3-tuple of filtered x, y and z values.

Implementation note: this method is currently implemented as taking the sum of 4 samples, sampled from the 3 previous calls to this function along with the sample from the current call. Returned values are therefore 4 times the size of what they would be from the raw x(), y() and z() calls.

Accel.**tilt**()
Get the tilt register.

Accel.**x**()
Get the x-axis value.

Accel.**y**()
Get the y-axis value.

Accel.**z**()
Get the z-axis value.

Hardware Note The accelerometer uses I2C bus 1 to communicate with the processor. Consequently when readings are being taken pins X9 and X10 should be unused (other than for I2C). Other devices using those pins, and which therefore cannot be used concurrently, are UART 1 and Timer 4 channels 1 and 2.

class ADC – analog to digital conversion

Usage:

```
import pyb

adc = pyb.ADC(pin)              # create an analog object from a pin
val = adc.read()                # read an analog value

adc = pyb.ADCAll(resolution)    # create an ADCAll object
val = adc.read_channel(channel) # read the given channel
val = adc.read_core_temp()      # read MCU temperature
val = adc.read_core_vbat()      # read MCU VBAT
val = adc.read_core_vref()      # read MCU VREF
```

Constructors

class pyb.**ADC**(*pin*)

Create an ADC object associated with the given pin. This allows you to then read analog values on that pin.

Methods

ADC.**read**()

Read the value on the analog pin and return it. The returned value will be between 0 and 4095.

ADC.**read_timed**(*buf*, *timer*)

Read analog values into `buf` at a rate set by the `timer` object.

`buf` can be bytearray or array.array for example. The ADC values have 12-bit resolution and are stored directly into `buf` if its element size is 16 bits or greater. If `buf` has only 8-bit elements (eg a bytearray) then the sample resolution will be reduced to 8 bits.

`timer` should be a Timer object, and a sample is read each time the timer triggers. The timer must already be initialised and running at the desired sampling frequency.

To support previous behaviour of this function, `timer` can also be an integer which specifies the frequency (in Hz) to sample at. In this case Timer(6) will be automatically configured to run at the given frequency.

Example using a Timer object (preferred way):

```
adc = pyb.ADC(pyb.Pin.board.X19)     # create an ADC on pin X19
tim = pyb.Timer(6, freq=10)          # create a timer running at 10Hz
buf = bytearray(100)                 # creat a buffer to store the samples
adc.read_timed(buf, tim)             # sample 100 values, taking 10s
```

Example using an integer for the frequency:

```
adc = pyb.ADC(pyb.Pin.board.X19)     # create an ADC on pin X19
buf = bytearray(100)                 # create a buffer of 100 bytes
adc.read_timed(buf, 10)              # read analog values into buf at 10Hz
                                     #   this will take 10 seconds to finish
for val in buf:                      # loop over all values
    print(val)                       # print the value out
```

This function does not allocate any memory.

The ADCAll Object Instantiating this changes all ADC pins to analog inputs. The raw MCU temperature, VREF and VBAT data can be accessed on ADC channels 16, 17 and 18 respectively. Appropriate scaling will need to be applied. The temperature sensor on the chip has poor absolute accuracy and is suitable only for detecting temperature changes.

The `ADCAll` `read_core_vbat()` and `read_core_vref()` methods read the backup battery voltage and the (1.21V nominal) reference voltage using the 3.3V supply as a reference. Assuming the `ADCAll` object has been Instantiated with `adc = pyb.ADCAll(12)` the 3.3V supply voltage may be calculated:

```
v33 = 3.3 * 1.21 / adc.read_core_vref()
```

If the 3.3V supply is correct the value of `adc.read_core_vbat()` will be valid. If the supply voltage can drop below 3.3V, for example in in battery powered systems with a discharging battery, the regulator will fail to preserve the 3.3V supply resulting in an incorrect reading. To produce a value which will remain valid under these circumstances use the following:

```
vback = adc.read_core_vbat() * 1.21 / adc.read_core_vref()
```

It is possible to access these values without incurring the side effects of `ADCAll`:

```
def adcread(chan):                              # 16 temp 17 vbat 18 vref
    assert chan >= 16 and chan <= 18, 'Invalid ADC channel'
    start = pyb.millis()
    timeout = 100
    stm.mem32[stm.RCC + stm.RCC_APB2ENR] |= 0x100 # enable ADC1 clock.0x4100
    stm.mem32[stm.ADC1 + stm.ADC_CR2] = 1       # Turn on ADC
    stm.mem32[stm.ADC1 + stm.ADC_CR1] = 0       # 12 bit
    if chan == 17:
        stm.mem32[stm.ADC1 + stm.ADC_SMPR1] = 0x200000 # 15 cycles
        stm.mem32[stm.ADC + 4] = 1 << 23
    elif chan == 18:
        stm.mem32[stm.ADC1 + stm.ADC_SMPR1] = 0x1000000
        stm.mem32[stm.ADC + 4] = 0xc00000
    else:
        stm.mem32[stm.ADC1 + stm.ADC_SMPR1] = 0x40000
        stm.mem32[stm.ADC + 4] = 1 << 23
    stm.mem32[stm.ADC1 + stm.ADC_SQR3] = chan
    stm.mem32[stm.ADC1 + stm.ADC_CR2] = 1 | (1 << 30) | (1 << 10) # start conversion
    while not stm.mem32[stm.ADC1 + stm.ADC_SR] & 2: # wait for EOC
        if pyb.elapsed_millis(start) > timeout:
            raise OSError('ADC timout')
    data = stm.mem32[stm.ADC1 + stm.ADC_DR]     # clear down EOC
    stm.mem32[stm.ADC1 + stm.ADC_CR2] = 0       # Turn off ADC
    return data

def v33():
    return 4096 * 1.21 / adcread(17)

def vbat():
    return  1.21 * 2 * adcread(18) / adcread(17)  # 2:1 divider on Vbat channel

def vref():
    return 3.3 * adcread(17) / 4096

def temperature():
    return 25 + 400 * (3.3 * adcread(16) / 4096 - 0.76)
```

class CAN – controller area network communication bus

CAN implements the standard CAN communications protocol. At the physical level it consists of 2 lines: RX and TX. Note that to connect the pyboard to a CAN bus you must use a CAN transceiver to convert the CAN logic signals from the pyboard to the correct voltage levels on the bus.

Example usage (works without anything connected):

```
from pyb import CAN
can = CAN(1, CAN.LOOPBACK)
can.setfilter(0, CAN.LIST16, 0, (123, 124, 125, 126))  # set a filter to receive messages with id=12
can.send('message!', 123)   # send a message with id 123
can.recv(0)                 # receive message on FIFO 0
```

Constructors

class pyb.**CAN**(*bus*, ...)

Construct a CAN object on the given bus. `bus` can be 1-2, or 'YA' or 'YB'. With no additional parameters, the CAN 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.

The physical pins of the CAN busses are:

- `CAN(1)` is on `YA`: `(RX, TX) = (Y3, Y4) = (PB8, PB9)`
- `CAN(2)` is on `YB`: `(RX, TX) = (Y5, Y6) = (PB12, PB13)`

Class Methods

classmethod `CAN.`**`initfilterbanks`**(*nr*)

Reset and disable all filter banks and assign how many banks should be available for CAN(1).

STM32F405 has 28 filter banks that are shared between the two available CAN bus controllers. This function configures how many filter banks should be assigned to each. `nr` is the number of banks that will be assigned to CAN(1), the rest of the 28 are assigned to CAN(2). At boot, 14 banks are assigned to each controller.

Methods

`CAN.`**`init`**(*mode*, *extframe=False*, *prescaler=100*, ***, *sjw=1*, *bs1=6*, *bs2=8*)

Initialise the CAN bus with the given parameters:

- `mode` is one of: NORMAL, LOOPBACK, SILENT, SILENT_LOOPBACK
- if `extframe` is True then the bus uses extended identifiers in the frames (29 bits); otherwise it uses standard 11 bit identifiers
- `prescaler` is used to set the duration of 1 time quanta; the time quanta will be the input clock (PCLK1, see *`pyb.freq()`*) divided by the prescaler
- `sjw` is the resynchronisation jump width in units of the time quanta; it can be 1, 2, 3, 4
- `bs1` defines the location of the sample point in units of the time quanta; it can be between 1 and 1024 inclusive
- `bs2` defines the location of the transmit point in units of the time quanta; it can be between 1 and 16 inclusive

The time quanta tq is the basic unit of time for the CAN bus. tq is the CAN prescaler value divided by PCLK1 (the frequency of internal peripheral bus 1); see *`pyb.freq()`* to determine PCLK1.

A single bit is made up of the synchronisation segment, which is always 1 tq. Then follows bit segment 1, then bit segment 2. The sample point is after bit segment 1 finishes. The transmit point is after bit segment 2 finishes. The baud rate will be 1/bittime, where the bittime is 1 + BS1 + BS2 multiplied by the time quanta tq.

For example, with PCLK1=42MHz, prescaler=100, sjw=1, bs1=6, bs2=8, the value of tq is 2.38 microseconds. The bittime is 35.7 microseconds, and the baudrate is 28kHz.

See page 680 of the STM32F405 datasheet for more details.

`CAN.`**`deinit`**()

Turn off the CAN bus.

`CAN.`**`setfilter`**(*bank*, *mode*, *fifo*, *params*, ***, *rtr*)

Configure a filter bank:

- `bank` is the filter bank that is to be configured.
- `mode` is the mode the filter should operate in.
- `fifo` is which fifo (0 or 1) a message should be stored in, if it is accepted by this filter.
- `params` is an array of values the defines the filter. The contents of the array depends on the `mode` argument.

mode	contents of parameter array
CAN.LIST16	Four 16 bit ids that will be accepted
CAN.LIST32	Two 32 bit ids that will be accepted
CAN.MASK16	**Two 16 bit id/mask pairs. E.g. (1, 3, 4, 4)** The first pair, 1 and 3 will accept all ids that have bit 0 = 1 and bit 1 = 0. The second pair, 4 and 4, will accept all ids that have bit 2 = 1.
CAN.MASK32	As with CAN.MASK16 but with only one 32 bit id/mask pair.

- `rtr` is an array of booleans that states if a filter should accept a remote transmission request message. If this argument is not given then it defaults to False for all entries. The length of the array depends on the `mode` argument.

mode	length of rtr array
CAN.LIST16	4
CAN.LIST32	2
CAN.MASK16	2
CAN.MASK32	1

`CAN.`**`clearfilter`**(*bank*)
Clear and disables a filter bank:

- `bank` is the filter bank that is to be cleared.

`CAN.`**`any`**(*fifo*)
Return `True` if any message waiting on the FIFO, else `False`.

`CAN.`**`recv`**(*fifo*, *, *timeout=5000*)
Receive data on the bus:

- `fifo` is an integer, which is the FIFO to receive on
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: A tuple containing four values.

- The id of the message.
- A boolean that indicates if the message is an RTR message.
- The FMI (Filter Match Index) value.
- An array containing the data.

`CAN.`**`send`**(*data*, *id*, *, *timeout=0*, *rtr=False*)
Send a message on the bus:

- `data` is the data to send (an integer to send, or a buffer object).
- `id` is the id of the message to be sent.
- `timeout` is the timeout in milliseconds to wait for the send.
- `rtr` is a boolean that specifies if the message shall be sent as a remote transmission request. If `rtr` is True then only the length of `data` is used to fill in the DLC slot of the frame; the actual bytes in `data` are unused.

If timeout is 0 the message is placed in a buffer in one of three hardware buffers and the method returns immediately. If all three buffers are in use an exception is thrown. If timeout is not 0, the method waits until the message is transmitted. If the message can't be transmitted within the specified time an exception is thrown.

Return value: `None`.

`CAN.`**`rxcallback`**(*fifo*, *fun*)

Register a function to be called when a message is accepted into a empty fifo:

- `fifo` is the receiving fifo.
- `fun` is the function to be called when the fifo becomes non empty.

The callback function takes two arguments the first is the can object it self the second is a integer that indicates the reason for the callback.

Reason	
0	A message has been accepted into a empty FIFO.
1	The FIFO is full
2	A message has been lost due to a full FIFO

Example use of rxcallback:

```
def cb0(bus, reason):
  print('cb0')
  if reason == 0:
      print('pending')
  if reason == 1:
      print('full')
  if reason == 2:
      print('overflow')

can = CAN(1, CAN.LOOPBACK)
can.rxcallback(0, cb0)
```

Constants

`CAN.`**`NORMAL`**

`CAN.`**`LOOPBACK`**

`CAN.`**`SILENT`**

`CAN.`**`SILENT_LOOPBACK`**

the mode of the CAN bus

`CAN.`**`LIST16`**

`CAN.`**`MASK16`**

`CAN.`**`LIST32`**

`CAN.`**`MASK32`**

the operation mode of a filter

class DAC – digital to analog conversion

The DAC is used to output analog values (a specific voltage) on pin X5 or pin X6. The voltage will be between 0 and 3.3V.

This module will undergo changes to the API.

Example usage:

```
from pyb import DAC

dac = DAC(1)            # create DAC 1 on pin X5
dac.write(128)          # write a value to the DAC (makes X5 1.65V)

dac = DAC(1, bits=12)   # use 12 bit resolution
dac.write(4095)         # output maximum value, 3.3V
```

To output a continuous sine-wave:

```
import math
from pyb import DAC

# create a buffer containing a sine-wave
buf = bytearray(100)
for i in range(len(buf)):
    buf[i] = 128 + int(127 * math.sin(2 * math.pi * i / len(buf)))

# output the sine-wave at 400Hz
dac = DAC(1)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

To output a continuous sine-wave at 12-bit resolution:

```
import math
from array import array
from pyb import DAC

# create a buffer containing a sine-wave, using half-word samples
buf = array('H', 2048 + int(2047 * math.sin(2 * math.pi * i / 128)) for i in range(128))

# output the sine-wave at 400Hz
dac = DAC(1, bits=12)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

Constructors

class pyb.**DAC** (*port*, *bits=8*)

Construct a new DAC object.

`port` can be a pin object, or an integer (1 or 2). DAC(1) is on pin X5 and DAC(2) is on pin X6.

`bits` is an integer specifying the resolution, and can be 8 or 12. The maximum value for the write and write_timed methods will be 2**‘‘bits‘‘-1.

Methods

DAC.**init** (*bits=8*)

Reinitialise the DAC. `bits` can be 8 or 12.

DAC.**deinit** ()

De-initialise the DAC making its pin available for other uses.

DAC.**noise** (*freq*)

Generate a pseudo-random noise signal. A new random sample is written to the DAC output at the given frequency.

DAC.**triangle**(*freq*)
Generate a triangle wave. The value on the DAC output changes at the given frequency, and the frequency of the repeating triangle wave itself is 2048 times smaller.

DAC.**write**(*value*)
Direct access to the DAC output. The minimum value is 0. The maximum value is 2**‘‘bits‘‘-1, where `bits` is set when creating the DAC object or by using the `init` method.

DAC.**write_timed**(*data*, *freq*, *, *mode=DAC.NORMAL*)
Initiates a burst of RAM to DAC using a DMA transfer. The input data is treated as an array of bytes in 8-bit mode, and an array of unsigned half-words (array typecode ‘H’) in 12-bit mode.

`freq` can be an integer specifying the frequency to write the DAC samples at, using Timer(6). Or it can be an already-initialised Timer object which is used to trigger the DAC sample. Valid timers are 2, 4, 5, 6, 7 and 8.

`mode` can be `DAC.NORMAL` or `DAC.CIRCULAR`.

Example using both DACs at the same time:

```
dac1 = DAC(1)
dac2 = DAC(2)
dac1.write_timed(buf1, pyb.Timer(6, freq=100), mode=DAC.CIRCULAR)
dac2.write_timed(buf2, pyb.Timer(7, freq=200), mode=DAC.CIRCULAR)
```

class ExtInt – configure I/O pins to interrupt on external events

There are a total of 22 interrupt lines. 16 of these can come from GPIO pins and the remaining 6 are from internal sources.

For lines 0 through 15, a given line can map to the corresponding line from an arbitrary port. So line 0 can map to Px0 where x is A, B, C, ... and line 1 can map to Px1 where x is A, B, C, ...

```
def callback(line):
    print("line =", line)
```

Note: ExtInt will automatically configure the gpio line as an input.

```
extint = pyb.ExtInt(pin, pyb.ExtInt.IRQ_FALLING, pyb.Pin.PULL_UP, callback)
```

Now every time a falling edge is seen on the X1 pin, the callback will be called. Caution: mechanical pushbuttons 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.

Trying to register 2 callbacks onto the same pin will throw an exception.

If pin is passed as an integer, then it is assumed to map to one of the internal interrupt sources, and must be in the range 16 through 22.

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

```
extint = pyb.ExtInt(pin, mode, pull, callback)
```

Valid modes are pyb.ExtInt.IRQ_RISING, pyb.ExtInt.IRQ_FALLING, pyb.ExtInt.IRQ_RISING_FALLING, pyb.ExtInt.EVT_RISING, pyb.ExtInt.EVT_FALLING, and pyb.ExtInt.EVT_RISING_FALLING.

Only the IRQ_xxx modes have been tested. The EVT_xxx modes have something to do with sleep mode and the WFE instruction.

Valid pull values are pyb.Pin.PULL_UP, pyb.Pin.PULL_DOWN, pyb.Pin.PULL_NONE.

There is also a C API, so that drivers which require EXTI interrupt lines can also use this code. See extint.h for the available functions and usrsw.h for an example of using this.

Constructors

class pyb.**ExtInt** (*pin*, *mode*, *pull*, *callback*)

Create an ExtInt object:

- `pin` is the pin on which to enable the interrupt (can be a pin object or any valid pin name).
- `mode` can be one of: - `ExtInt.IRQ_RISING` - trigger on a rising edge; - `ExtInt.IRQ_FALLING` - trigger on a falling edge; - `ExtInt.IRQ_RISING_FALLING` - trigger on a rising or falling edge.
- `pull` can be one of: - `pyb.Pin.PULL_NONE` - no pull up or down resistors; - `pyb.Pin.PULL_UP` - enable the pull-up resistor; - `pyb.Pin.PULL_DOWN` - enable the pull-down resistor.
- `callback` is the function to call when the interrupt triggers. The callback function must accept exactly 1 argument, which is the line that triggered the interrupt.

Class methods

classmethod ExtInt.**regs**()

Dump the values of the EXTI registers.

Methods

ExtInt.**disable**()

Disable the interrupt associated with the ExtInt object. This could be useful for debouncing.

ExtInt.**enable**()

Enable a disabled interrupt.

ExtInt.**line**()

Return the line number that the pin is mapped to.

ExtInt.**swint**()

Trigger the callback from software.

Constants

ExtInt.**IRQ_FALLING**

interrupt on a falling edge

ExtInt.**IRQ_RISING**

interrupt on a rising edge

ExtInt.**IRQ_RISING_FALLING**

interrupt on a rising or falling edge

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.

Example:

```
from pyb import I2C

i2c = I2C(1)                            # create on bus 1
```

```
i2c = I2C(1, I2C.MASTER)              # create and init as a master
i2c.init(I2C.MASTER, baudrate=20000) # init as a master
i2c.init(I2C.SLAVE, addr=0x42)       # init as a slave with given address
i2c.deinit()                          # turn off the peripheral
```

Printing the i2c object gives you information about its configuration.

The basic methods are send and recv:

```
i2c.send('abc')      # send 3 bytes
i2c.send(0x42)       # send a single byte, given by the number
data = i2c.recv(3)   # receive 3 bytes
```

To receive inplace, first create a bytearray:

```
data = bytearray(3)  # create a buffer
i2c.recv(data)       # receive 3 bytes, writing them into data
```

You can specify a timeout (in ms):

```
i2c.send(b'123', timeout=2000)    # timeout after 2 seconds
```

A master must specify the recipient's address:

```
i2c.init(I2C.MASTER)
i2c.send('123', 0x42)        # send 3 bytes to slave with address 0x42
i2c.send(b'456', addr=0x42)  # keyword for address
```

Master also has other methods:

```
i2c.is_ready(0x42)           # check if slave 0x42 is ready
i2c.scan()                   # scan for slaves on the bus, returning
                             #   a list of valid addresses
i2c.mem_read(3, 0x42, 2)     # read 3 bytes from memory of slave 0x42,
                             #   starting at address 2 in the slave
i2c.mem_write('abc', 0x42, 2, timeout=1000) # write 'abc' (3 bytes) to memory of slave 0x42
                                            # starting at address 2 in the slave, timeout after 1 se
```

Constructors

class pyb.**I2C**(*bus*, ...)

Construct an I2C object on the given bus. `bus` can be 1 or 2, 'X' or 'Y'. With no additional parameters, the I2C 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.

The physical pins of the I2C busses on Pyboards V1.0 and V1.1 are:

- `I2C(1)` is on the X position: `(SCL, SDA) = (X9, X10) = (PB6, PB7)`
- `I2C(2)` is on the Y position: `(SCL, SDA) = (Y9, Y10) = (PB10, PB11)`

On the Pyboard Lite:

- `I2C(1)` is on the X position: `(SCL, SDA) = (X9, X10) = (PB6, PB7)`
- `I2C(3)` is on the Y position: `(SCL, SDA) = (Y9, Y10) = (PA8, PB8)`

Calling the constructor with 'X' or 'Y' enables portability between Pyboard types.

Methods

I2C.**deinit**()
Turn off the I2C bus.

I2C.**init**(*mode*, *, *addr=0x12*, *baudrate=400000*, *gencall=False*, *dma=False*)
Initialise the I2C bus with the given parameters:

- `mode` must be either `I2C.MASTER` or `I2C.SLAVE`
- `addr` is the 7-bit address (only sensible for a slave)
- `baudrate` is the SCL clock rate (only sensible for a master)
- `gencall` is whether to support general call mode
- `dma` is whether to allow the use of DMA for the I2C transfers (note that DMA transfers have more precise timing but currently do not handle bus errors properly)

I2C.**is_ready**(*addr*)
Check if an I2C device responds to the given address. Only valid when in master mode.

I2C.**mem_read**(*data*, *addr*, *memaddr*, *, *timeout=5000*, *addr_size=8*)
Read from the memory of an I2C device:

- `data` can be an integer (number of bytes to read) or a buffer to read into
- `addr` is the I2C device address
- `memaddr` is the memory location within the I2C device
- `timeout` is the timeout in milliseconds to wait for the read
- `addr_size` selects width of memaddr: 8 or 16 bits

Returns the read data. This is only valid in master mode.

I2C.**mem_write**(*data*, *addr*, *memaddr*, *, *timeout=5000*, *addr_size=8*)
Write to the memory of an I2C device:

- `data` can be an integer or a buffer to write from
- `addr` is the I2C device address
- `memaddr` is the memory location within the I2C device
- `timeout` is the timeout in milliseconds to wait for the write
- `addr_size` selects width of memaddr: 8 or 16 bits

Returns `None`. This is only valid in master mode.

I2C.**recv**(*recv*, *addr=0x00*, *, *timeout=5000*)
Receive data on the bus:

- `recv` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes
- `addr` is the address to receive from (only required in master mode)
- `timeout` is the timeout in milliseconds to wait for the receive

Return value: if `recv` is an integer then a new buffer of the bytes received, otherwise the same buffer that was passed in to `recv`.

I2C.**send**(*send*, *addr=0x00*, *, *timeout=5000*)
Send data on the bus:

- `send` is the data to send (an integer to send, or a buffer object)

- `addr` is the address to send to (only required in master mode)
- `timeout` is the timeout in milliseconds to wait for the send

Return value: `None`.

`I2C.`**`scan`**`()`
Scan all I2C addresses from 0x01 to 0x7f and return a list of those that respond. Only valid when in master mode.

Constants

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

`I2C.`**`SLAVE`**
for initialising the bus to slave mode

class LCD – LCD control for the LCD touch-sensor pyskin

The LCD class is used to control the LCD on the LCD touch-sensor pyskin, LCD32MKv1.0. The LCD is a 128x32 pixel monochrome screen, part NHD-C12832A1Z.

The pyskin must be connected in either the X or Y positions, and then an LCD object is made using:

```
lcd = pyb.LCD('X')      # if pyskin is in the X position
lcd = pyb.LCD('Y')      # if pyskin is in the Y position
```

Then you can use:

```
lcd.light(True)                 # turn the backlight on
lcd.write('Hello world!\n')     # print text to the screen
```

This driver implements a double buffer for setting/getting pixels. For example, to make a bouncing dot, try:

```
x = y = 0
dx = dy = 1
while True:
    # update the dot's position
    x += dx
    y += dy

    # make the dot bounce of the edges of the screen
    if x <= 0 or x >= 127: dx = -dx
    if y <= 0 or y >= 31: dy = -dy

    lcd.fill(0)                 # clear the buffer
    lcd.pixel(x, y, 1)          # draw the dot
    lcd.show()                  # show the buffer
    pyb.delay(50)               # pause for 50ms
```

Constructors

class `pyb.`**`LCD`**`(`*skin_position*`)`
Construct an LCD object in the given skin position. `skin_position` can be 'X' or 'Y', and should match the position where the LCD pyskin is plugged in.

Methods

LCD.**command**(*instr_data*, *buf*)
Send an arbitrary command to the LCD. Pass 0 for `instr_data` to send an instruction, otherwise pass 1 to send data. `buf` is a buffer with the instructions/data to send.

LCD.**contrast**(*value*)
Set the contrast of the LCD. Valid values are between 0 and 47.

LCD.**fill**(*colour*)
Fill the screen with the given colour (0 or 1 for white or black).

This method writes to the hidden buffer. Use `show()` to show the buffer.

LCD.**get**(*x*, *y*)
Get the pixel at the position `(x, y)`. Returns 0 or 1.

This method reads from the visible buffer.

LCD.**light**(*value*)
Turn the backlight on/off. True or 1 turns it on, False or 0 turns it off.

LCD.**pixel**(*x*, *y*, *colour*)
Set the pixel at `(x, y)` to the given colour (0 or 1).

This method writes to the hidden buffer. Use `show()` to show the buffer.

LCD.**show**()
Show the hidden buffer on the screen.

LCD.**text**(*str*, *x*, *y*, *colour*)
Draw the given text to the position `(x, y)` using the given colour (0 or 1).

This method writes to the hidden buffer. Use `show()` to show the buffer.

LCD.**write**(*str*)
Write the string `str` to the screen. It will appear immediately.

class LED – LED object

The LED object controls an individual LED (Light Emitting Diode).

Constructors

class pyb.**LED**(*id*)
Create an LED object associated with the given LED:

- `id` is the LED number, 1-4.

Methods

LED.**intensity**([*value*])
Get or set the LED intensity. Intensity ranges between 0 (off) and 255 (full on). If no argument is given, return the LED intensity. If an argument is given, set the LED intensity and return `None`.

Note: Only LED(3) and LED(4) can have a smoothly varying intensity, and they use timer PWM to implement it. LED(3) uses Timer(2) and LED(4) uses Timer(3). These timers are only configured for PWM if the intensity of the relevant LED is set to a value between 1 and 254. Otherwise the timers are free for general purpose use.

LED.**off**()
Turn the LED off.

LED.**on**()
Turn the LED on, to maximum intensity.

LED.**toggle**()
Toggle the LED between on (maximum intensity) and off. If the LED is at non-zero intensity then it is considered "on" and toggle will turn it off.

class Pin – control I/O pins

A pin is the basic object to control I/O pins. It has methods to set the mode of the pin (input, output, etc) and methods to get and set the digital logic level. For analog control of a pin, see the ADC class.

Usage Model:

All Board Pins are predefined as pyb.Pin.board.Name:

```
x1_pin = pyb.Pin.board.X1

g = pyb.Pin(pyb.Pin.board.X1, pyb.Pin.IN)
```

CPU pins which correspond to the board pins are available as `pyb.cpu.Name`. For the CPU pins, the names are the port letter followed by the pin number. On the PYBv1.0, `pyb.Pin.board.X1` and `pyb.Pin.cpu.A0` are the same pin.

You can also use strings:

```
g = pyb.Pin('X1', pyb.Pin.OUT_PP)
```

Users can add their own names:

```
MyMapperDict = { 'LeftMotorDir' : pyb.Pin.cpu.C12 }
pyb.Pin.dict(MyMapperDict)
g = pyb.Pin("LeftMotorDir", pyb.Pin.OUT_OD)
```

and can query mappings:

```
pin = pyb.Pin("LeftMotorDir")
```

Users can also add their own mapping function:

```
def MyMapper(pin_name):
   if pin_name == "LeftMotorDir":
       return pyb.Pin.cpu.A0

pyb.Pin.mapper(MyMapper)
```

So, if you were to call: `pyb.Pin("LeftMotorDir", pyb.Pin.OUT_PP)` then `"LeftMotorDir"` is passed directly to the mapper function.

To summarise, the following order determines how things get mapped into an ordinal pin number:

1. Directly specify a pin object
2. User supplied mapping function
3. User supplied mapping (object must be usable as a dictionary key)
4. Supply a string which matches a board pin
5. Supply a string which matches a CPU port/pin

You can set `pyb.Pin.debug(True)` to get some debug information about how a particular object gets mapped to a pin.

When a pin has the `Pin.PULL_UP` or `Pin.PULL_DOWN` pull-mode enabled, that pin has an effective 40k Ohm resistor pulling it to 3V3 or GND respectively (except pin Y5 which has 11k Ohm resistors).

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.

Constructors

class `pyb.`**`Pin`**(*id*, *...*)

Create a new Pin object associated with the id. If additional arguments are given, they are used to initialise the pin. See `pin.init()`.

Class methods

classmethod `Pin.`**`debug`**([*state*])

Get or set the debugging state (`True` or `False` for on or off).

classmethod `Pin.`**`dict`**([*dict*])

Get or set the pin mapper dictionary.

classmethod `Pin.`**`mapper`**([*fun*])

Get or set the pin mapper function.

Methods

`Pin.`**`init`**(*mode*, *pull=Pin.PULL_NONE*, *af=-1*)

Initialise the pin:

- `mode` can be one of:
  - `Pin.IN` - configure the pin for input;
  - `Pin.OUT_PP` - configure the pin for output, with push-pull control;
  - `Pin.OUT_OD` - configure the pin for output, with open-drain control;
  - `Pin.AF_PP` - configure the pin for alternate function, pull-pull;
  - `Pin.AF_OD` - configure the pin for alternate function, open-drain;
  - `Pin.ANALOG` - configure the pin for analog.
- `pull` can be one of:
  - `Pin.PULL_NONE` - no pull up or down resistors;
  - `Pin.PULL_UP` - enable the pull-up resistor;
  - `Pin.PULL_DOWN` - enable the pull-down resistor.
- when mode is `Pin.AF_PP` or `Pin.AF_OD`, then af can be the index or name of one of the alternate functions associated with a pin.

Returns: `None`.

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

Get or set the digital logic level of the pin:

- With no argument, return 0 or 1 depending on the logic level of the pin.
- With `value` given, set the logic level of the pin. `value` can be anything that converts to a boolean. If it converts to `True`, the pin is set high, otherwise it is set low.

Pin.**__str__**()
Return a string describing the pin object.

Pin.**af**()
Returns the currently configured alternate-function of the pin. The integer returned will match one of the allowed constants for the af argument to the init function.

Pin.**af_list**()
Returns an array of alternate functions available for this pin.

Pin.**gpio**()
Returns the base address of the GPIO block associated with this pin.

Pin.**mode**()
Returns the currently configured mode of the pin. The integer returned will match one of the allowed constants for the mode argument to the init function.

Pin.**name**()
Get the pin name.

Pin.**names**()
Returns the cpu and board names for this pin.

Pin.**pin**()
Get the pin number.

Pin.**port**()
Get the pin port.

Pin.**pull**()
Returns the currently configured pull of the pin. The integer returned will match one of the allowed constants for the pull argument to the init function.

Constants

Pin.**AF_OD**
initialise the pin to alternate-function mode with an open-drain drive

Pin.**AF_PP**
initialise the pin to alternate-function mode with a push-pull drive

Pin.**ANALOG**
initialise the pin to analog mode

Pin.**IN**
initialise the pin to input mode

Pin.**OUT_OD**
initialise the pin to output mode with an open-drain drive

Pin.**OUT_PP**
initialise the pin to output mode with a push-pull drive

Pin.**PULL_DOWN**
enable the pull-down resistor on the pin

Pin.**PULL_NONE**
don't enable any pull up or down resistors on the pin

Pin.**PULL_UP**
enable the pull-up resistor on the pin

class PinAF – Pin Alternate Functions

A Pin represents a physical pin on the microprocessor. Each pin can have a variety of functions (GPIO, I2C SDA, etc). Each PinAF object represents a particular function for a pin.

Usage Model:

```
x3 = pyb.Pin.board.X3
x3_af = x3.af_list()
```

x3_af will now contain an array of PinAF objects which are available on pin X3.

For the pyboard, x3_af would contain: [Pin.AF1_TIM2, Pin.AF2_TIM5, Pin.AF3_TIM9, Pin.AF7_USART2]

Normally, each peripheral would configure the af automatically, but sometimes the same function is available on multiple pins, and having more control is desired.

To configure X3 to expose TIM2_CH3, you could use:

```
pin = pyb.Pin(pyb.Pin.board.X3, mode=pyb.Pin.AF_PP, af=pyb.Pin.AF1_TIM2)
```

or:

```
pin = pyb.Pin(pyb.Pin.board.X3, mode=pyb.Pin.AF_PP, af=1)
```

Methods

`pinaf.__str__()`
: Return a string describing the alternate function.

`pinaf.index()`
: Return the alternate function index.

`pinaf.name()`
: Return the name of the alternate function.

`pinaf.reg()`
: Return the base register associated with the peripheral assigned to this alternate function. For example, if the alternate function were TIM2_CH3 this would return stm.TIM2

class RTC – real time clock

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

Example usage:

```
rtc = pyb.RTC()
rtc.datetime((2014, 5, 1, 4, 13, 0, 0, 0))
print(rtc.datetime())
```

Constructors

class `pyb.RTC`
: Create an RTC object.

Methods

RTC.**datetime**([*datetimetuple*])
Get or set the date and time of the RTC.

With no arguments, this method returns an 8-tuple with the current date and time. With 1 argument (being an 8-tuple) it sets the date and time.

The 8-tuple has the following format:

(year, month, day, weekday, hours, minutes, seconds, subseconds)

`weekday` is 1-7 for Monday through Sunday.

`subseconds` counts down from 255 to 0

RTC.**wakeup**(*timeout*, *callback=None*)
Set the RTC wakeup timer to trigger repeatedly at every `timeout` milliseconds. This trigger can wake the pyboard from both the sleep states: *`pyb.stop()`* and *`pyb.standby()`*.

If `timeout` is `None` then the wakeup timer is disabled.

If `callback` is given then it is executed at every trigger of the wakeup timer. `callback` must take exactly one argument.

RTC.**info**()
Get information about the startup time and reset source.

- The lower 0xffff are the number of milliseconds the RTC took to start up.
- Bit 0x10000 is set if a power-on reset occurred.
- Bit 0x20000 is set if an external reset occurred

RTC.**calibration**(*cal*)
Get or set RTC calibration.

With no arguments, `calibration()` returns the current calibration value, which is an integer in the range [-511 : 512]. With one argument it sets the RTC calibration.

The RTC Smooth Calibration mechanism adjusts the RTC clock rate by adding or subtracting the given number of ticks from the 32768 Hz clock over a 32 second period (corresponding to 2^20 clock ticks.) Each tick added will speed up the clock by 1 part in 2^20, or 0.954 ppm; likewise the RTC clock it slowed by negative values. The usable calibration range is: (-511 * 0.954) ~= -487.5 ppm up to (512 * 0.954) ~= 488.5 ppm

class Servo – 3-wire hobby servo driver

Servo objects control standard hobby servo motors with 3-wires (ground, power, signal). There are 4 positions on the pyboard where these motors can be plugged in: pins X1 through X4 are the signal pins, and next to them are 4 sets of power and ground pins.

Example usage:

```
import pyb

s1 = pyb.Servo(1)    # create a servo object on position X1
s2 = pyb.Servo(2)    # create a servo object on position X2

s1.angle(45)         # move servo 1 to 45 degrees
s2.angle(0)          # move servo 2 to 0 degrees

# move servo1 and servo2 synchronously, taking 1500ms
s1.angle(-60, 1500)
s2.angle(30, 1500)
```

Note: The Servo objects use Timer(5) to produce the PWM output. You can use Timer(5) for Servo control, or your own purposes, but not both at the same time.

Constructors

class pyb.**Servo**(*id*)

Create a servo object. id is 1-4, and corresponds to pins X1 through X4.

Methods

Servo.**angle**([*angle*, *time=0*])

If no arguments are given, this function returns the current angle.

If arguments are given, this function sets the angle of the servo:

- angle is the angle to move to in degrees.
- time is the number of milliseconds to take to get to the specified angle. If omitted, then the servo moves as quickly as possible to its new position.

Servo.**speed**([*speed*, *time=0*])

If no arguments are given, this function returns the current speed.

If arguments are given, this function sets the speed of the servo:

- speed is the speed to change to, between -100 and 100.
- time is the number of milliseconds to take to get to the specified speed. If omitted, then the servo accelerates as quickly as possible.

Servo.**pulse_width**([*value*])

If no arguments are given, this function returns the current raw pulse-width value.

If an argument is given, this function sets the raw pulse-width value.

Servo.**calibration**([*pulse_min*, *pulse_max*, *pulse_centre*[, *pulse_angle_90*, *pulse_speed_100*]])

If no arguments are given, this function returns the current calibration data, as a 5-tuple.

If arguments are given, this function sets the timing calibration:

- pulse_min is the minimum allowed pulse width.
- pulse_max is the maximum allowed pulse width.
- pulse_centre is the pulse width corresponding to the centre/zero position.
- pulse_angle_90 is the pulse width corresponding to 90 degrees.
- pulse_speed_100 is the pulse width corresponding to a speed of 100.

class SPI – a master-driven serial protocol

SPI is a serial protocol that is driven by a master. At the physical level there are 3 lines: SCK, MOSI, MISO.

See usage model of I2C; SPI is very similar. Main difference is parameters to init the SPI bus:

```
from pyb import SPI
spi = SPI(1, SPI.MASTER, baudrate=600000, polarity=1, phase=0, crc=0x7)
```

Only required parameter is mode, SPI.MASTER or SPI.SLAVE. 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. Crc can be None for no CRC, or a polynomial specifier.

Additional methods for SPI:

```
data = spi.send_recv(b'1234')          # send 4 bytes and receive 4 bytes
buf = bytearray(4)
spi.send_recv(b'1234', buf)            # send 4 bytes and receive 4 into buf
spi.send_recv(buf, buf)                # send/recv 4 bytes from/to buf
```

Constructors

class pyb.**SPI**(*bus*, ...)

Construct an SPI object on the given bus. bus can be 1 or 2, or 'X' or 'Y'. 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.

The physical pins of the SPI busses are:

- SPI(1) is on the X position: (NSS, SCK, MISO, MOSI) = (X5, X6, X7, X8) = (PA4, PA5, PA6, PA7)
- SPI(2) is on the Y position: (NSS, SCK, MISO, MOSI) = (Y5, Y6, Y7, Y8) = (PB12, PB13, PB14, PB15)

At the moment, the NSS pin is not used by the SPI driver and is free for other use.

Methods

SPI.**deinit**()

Turn off the SPI bus.

SPI.**init**(*mode*, *baudrate=328125*, ***, *prescaler*, *polarity=1*, *phase=0*, *bits=8*, *firstbit=SPI.MSB*, *ti=False*, *crc=None*)

Initialise the SPI bus with the given parameters:

- mode must be either SPI.MASTER or SPI.SLAVE.
- baudrate is the SCK clock rate (only sensible for a master).
- prescaler is the prescaler to use to derive SCK from the APB bus frequency; use of prescaler overrides baudrate.
- 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 can be 8 or 16, and is the number of bits in each transferred word.
- firstbit can be SPI.MSB or SPI.LSB.
- crc can be None for no CRC, or a polynomial specifier.

Note that the SPI clock frequency will not always be the requested baudrate. The hardware only supports baudrates that are the APB bus frequency (see *pyb.freq()*) divided by a prescaler, which can be 2, 4, 8, 16, 32, 64, 128 or 256. SPI(1) is on AHB2, and SPI(2) is on AHB1. For precise control over the SPI clock frequency, specify prescaler instead of baudrate.

Printing the SPI object will show you the computed baudrate and the chosen prescaler.

SPI.**recv**(*recv*, ***, *timeout=5000*)

Receive data on the bus:

- `recv` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: if `recv` is an integer then a new buffer of the bytes received, otherwise the same buffer that was passed in to `recv`.

`SPI.`**`send`**(*send*, *, *timeout=5000*)
Send data on the bus:

- `send` is the data to send (an integer to send, or a buffer object).
- `timeout` is the timeout in milliseconds to wait for the send.

Return value: `None`.

`SPI.`**`send_recv`**(*send*, *recv=None*, *, *timeout=5000*)
Send and receive data on the bus at the same time:

- `send` is the data to send (an integer to send, or a buffer object).
- `recv` is a mutable buffer which will be filled with received bytes. It can be the same as `send`, or omitted. If omitted, a new buffer will be created.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: the buffer with the received bytes.

Constants

`SPI.`**`MASTER`**
`SPI.`**`SLAVE`**
for initialising the SPI bus to master or slave mode

`SPI.`**`LSB`**

`SPI.`**`MSB`**
set the first bit to be the least or most significant bit

class Switch – switch object

A Switch object is used to control a push-button switch.

Usage:

```
sw = pyb.Switch()        # create a switch object
sw.value()               # get state (True if pressed, False otherwise)
sw()                     # shorthand notation to get the switch state
sw.callback(f)           # register a callback to be called when the
                         #   switch is pressed down
sw.callback(None)        # remove the callback
```

Example:

```
pyb.Switch().callback(lambda: pyb.LED(1).toggle())
```

Constructors

class `pyb.`**`Switch`**
Create and return a switch object.

Methods

`Switch.__call__()`
Call switch object directly to get its state: `True` if pressed down, `False` otherwise.

`Switch.value()`
Get the switch state. Returns `True` if pressed down, otherwise `False`.

`Switch.callback(`*fun*`)`
Register the given function to be called when the switch is pressed down. If `fun` is `None`, then it disables the callback.

class Timer – control internal timers

Timers can be used for a great variety of tasks. At the moment, only the simplest case is implemented: that of calling a function periodically.

Each timer consists of a counter that counts up at a certain rate. The rate at which it counts is the peripheral clock frequency (in Hz) divided by the timer prescaler. When the counter reaches the timer period it triggers an event, and the counter resets back to zero. By using the callback method, the timer event can call a Python function.

Example usage to toggle an LED at a fixed frequency:

```
tim = pyb.Timer(4)                  # create a timer object using timer 4
tim.init(freq=2)                    # trigger at 2Hz
tim.callback(lambda t:pyb.LED(1).toggle())
```

Example using named function for the callback:

```
def tick(timer):                    # we will receive the timer object when being called
    print(timer.counter())          # show current timer's counter value
tim = pyb.Timer(4, freq=1)          # create a timer object using timer 4 - trigger at 1Hz
tim.callback(tick)                  # set the callback to our tick function
```

Further examples:

```
tim = pyb.Timer(4, freq=100)      # freq in Hz
tim = pyb.Timer(4, prescaler=0, period=99)
tim.counter()                     # get counter (can also set)
tim.prescaler(2)                  # set prescaler (can also get)
tim.period(199)                   # set period (can also get)
tim.callback(lambda t: ...)       # set callback for update interrupt (t=tim instance)
tim.callback(None)                # clear callback
```

Note: Timer(2) and Timer(3) are used for PWM to set the intensity of LED(3) and LED(4) respectively. But these timers are only configured for PWM if the intensity of the relevant LED is set to a value between 1 and 254. If the intensity feature of the LEDs is not used then these timers are free for general purpose use. Similarly, Timer(5) controls the servo driver, and Timer(6) is used for timed ADC/DAC reading/writing. It is recommended to use the other timers in your programs.

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

Constructors

class `pyb.Timer(`*id*, *...*`)`
Construct a new timer object of the given id. If additional arguments are given, then the timer is initialised by `init(...)`. `id` can be 1 to 14.

Methods

`Timer.`**`init`**`(*, `*`freq`*`, `*`prescaler`*`, `*`period`*`)`

Initialise the timer. Initialisation must be either by frequency (in Hz) or by prescaler and period:

```
tim.init(freq=100)                  # set the timer to trigger at 100Hz
tim.init(prescaler=83, period=999)  # set the prescaler and period directly
```

Keyword arguments:

- `freq` — specifies the periodic frequency of the timer. You might also view this as the frequency with which the timer goes through one complete cycle.
- `prescaler` [0-0xffff] - specifies the value to be loaded into the timer's Prescaler Register (PSC). The timer clock source is divided by (`prescaler + 1`) to arrive at the timer clock. Timers 2-7 and 12-14 have a clock source of 84 MHz (pyb.freq()[2] * 2), and Timers 1, and 8-11 have a clock source of 168 MHz (pyb.freq()[3] * 2).
- `period` [0-0xffff] for timers 1, 3, 4, and 6-15. [0-0x3fffffff] for timers 2 & 5. Specifies the value to be loaded into the timer's AutoReload Register (ARR). This determines the period of the timer (i.e. when the counter cycles). The timer counter will roll-over after `period + 1` timer clock cycles.
- `mode` can be one of:
  - `Timer.UP` - configures the timer to count from 0 to ARR (default)
  - `Timer.DOWN` - configures the timer to count from ARR down to 0.
  - `Timer.CENTER` - configures the timer to count from 0 to ARR and then back down to 0.
- `div` can be one of 1, 2, or 4. Divides the timer clock to determine the sampling clock used by the digital filters.
- `callback` - as per Timer.callback()
- `deadtime` - specifies the amount of "dead" or inactive time between transitions on complimentary channels (both channels will be inactive) for this time). `deadtime` may be an integer between 0 and 1008, with the following restrictions: 0-128 in steps of 1. 128-256 in steps of 2, 256-512 in steps of 8, and 512-1008 in steps of 16. `deadtime` measures ticks of `source_freq` divided by `div` clock ticks. `deadtime` is only available on timers 1 and 8.

You must either specify freq or both of period and prescaler.

`Timer.`**`deinit`**`()`

Deinitialises the timer.

Disables the callback (and the associated irq).

Disables any channel callbacks (and the associated irq). Stops the timer, and disables the timer peripheral.

`Timer.`**`callback`**`(`*`fun`*`)`

Set the function to be called when the timer triggers. `fun` is passed 1 argument, the timer object. If `fun` is `None` then the callback will be disabled.

`Timer.`**`channel`**`(`*`channel`*`, `*`mode`*`, ...)`

If only a channel number is 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.

Each channel can be configured to perform pwm, output compare, or input capture. All channels share the same underlying timer, which means that they share the same timer clock.

Keyword arguments:

- `mode` can be one of:
  - `Timer.PWM` — configure the timer in PWM mode (active high).
  - `Timer.PWM_INVERTED` — configure the timer in PWM mode (active low).
  - `Timer.OC_TIMING` — indicates that no pin is driven.
  - `Timer.OC_ACTIVE` — the pin will be made active when a compare match occurs (active is determined by polarity)
  - `Timer.OC_INACTIVE` — the pin will be made inactive when a compare match occurs.
  - `Timer.OC_TOGGLE` — the pin will be toggled when an compare match occurs.
  - `Timer.OC_FORCED_ACTIVE` — the pin is forced active (compare match is ignored).
  - `Timer.OC_FORCED_INACTIVE` — the pin is forced inactive (compare match is ignored).
  - `Timer.IC` — configure the timer in Input Capture mode.
  - `Timer.ENC_A` — configure the timer in Encoder mode. The counter only changes when CH1 changes.
  - `Timer.ENC_B` — configure the timer in Encoder mode. The counter only changes when CH2 changes.
  - `Timer.ENC_AB` — configure the timer in Encoder mode. The counter changes when CH1 or CH2 changes.
- `callback` - as per TimerChannel.callback()
- `pin` None (the default) or a Pin object. If specified (and not None) this will cause the alternate function of the the indicated pin to be configured for this timer channel. An error will be raised if the pin doesn't support any alternate functions for this timer channel.

Keyword arguments for Timer.PWM modes:

- `pulse_width` - determines the initial pulse width value to use.
- `pulse_width_percent` - determines the initial pulse width percentage to use.

Keyword arguments for Timer.OC modes:

- `compare` - determines the initial value of the compare register.
- `polarity` can be one of:
  - `Timer.HIGH` - output is active high
  - `Timer.LOW` - output is active low

Optional keyword arguments for Timer.IC modes:

- `polarity` can be one of:
  - `Timer.RISING` - captures on rising edge.
  - `Timer.FALLING` - captures on falling edge.
  - `Timer.BOTH` - captures on both edges.

Note that capture only works on the primary channel, and not on the complimentary channels.

Notes for Timer.ENC modes:

- Requires 2 pins, so one or both pins will need to be configured to use the appropriate timer AF using the Pin API.

- Read the encoder value using the timer.counter() method.
- Only works on CH1 and CH2 (and not on CH1N or CH2N)
- The channel number is ignored when setting the encoder mode.

PWM Example:

```
timer = pyb.Timer(2, freq=1000)
ch2 = timer.channel(2, pyb.Timer.PWM, pin=pyb.Pin.board.X2, pulse_width=8000)
ch3 = timer.channel(3, pyb.Timer.PWM, pin=pyb.Pin.board.X3, pulse_width=16000)
```

`Timer.`**`counter`**`(`[*value*]`)`
Get or set the timer counter.

`Timer.`**`freq`**`(`[*value*]`)`
Get or set the frequency for the timer (changes prescaler and period if set).

`Timer.`**`period`**`(`[*value*]`)`
Get or set the period of the timer.

`Timer.`**`prescaler`**`(`[*value*]`)`
Get or set the prescaler for the timer.

`Timer.`**`source_freq`**`()`
Get the frequency of the source of the timer.

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.`**`callback`**`(`*fun*`)`
Set the function to be called when the timer channel triggers. `fun` is passed 1 argument, the timer object. If `fun` is `None` then the callback will be disabled.

`timerchannel.`**`capture`**`(`[*value*]`)`
Get or set the capture value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. capture is the logical name to use when the channel is in input capture mode.

`timerchannel.`**`compare`**`(`[*value*]`)`
Get or set the compare value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. compare is the logical name to use when the channel is in output compare mode.

`timerchannel.`**`pulse_width`**`(`[*value*]`)`
Get or set the pulse width value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. pulse_width is the logical name to use when the channel is in PWM mode.

In edge aligned mode, a pulse_width of `period + 1` corresponds to a duty cycle of 100% In center aligned mode, a pulse width of `period` corresponds to a duty cycle of 100%

`timerchannel.`**`pulse_width_percent`**`(`[*value*]`)`
Get or set the pulse width percentage associated with a channel. The value is a number between 0 and 100 and sets the percentage of the timer period for which the pulse is active. The value can be an integer or floating-point number for more accuracy. For example, a value of 25 gives a duty cycle of 25%.

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 pyb import UART

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

Bits can be 7, 8 or 9. Parity can be None, 0 (even) or 1 (odd). Stop can be 1 or 2.

Note: with parity=None, only 8 and 9 bits are supported. With parity enabled, only 7 and 8 bits are supported.

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
```

Individual characters can be read/written using:

```
uart.readchar()     # read 1 character and returns it as an integer
uart.writechar(42)  # write 1 character
```

To check if there is anything to be read, use:

```
uart.any()          # returns the number of characters waiting
```

Note: The stream functions `read`, `write`, etc. are new in MicroPython v1.3.4. Earlier versions use `uart.send` and `uart.recv`.

Constructors

class `pyb.`**`UART`**(*bus*, ...)

Construct a UART object on the given bus. `bus` can be 1-6, or 'XA', 'XB', 'YA', or 'YB'. With no additional parameters, the UART 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.

The physical pins of the UART busses are:

- `UART(4)` is on `XA`: `(TX, RX) = (X1, X2) = (PA0, PA1)`
- `UART(1)` is on `XB`: `(TX, RX) = (X9, X10) = (PB6, PB7)`
- `UART(6)` is on `YA`: `(TX, RX) = (Y1, Y2) = (PC6, PC7)`
- `UART(3)` is on `YB`: `(TX, RX) = (Y9, Y10) = (PB10, PB11)`
- `UART(2)` is on: `(TX, RX) = (X3, X4) = (PA2, PA3)`

The Pyboard Lite supports UART(1), UART(2) and UART(6) only. Pins are as above except:

- `UART(2)` is on: `(TX, RX) = (X1, X2) = (PA2, PA3)`

Methods

UART.**init**(*baudrate*, *bits=8*, *parity=None*, *stop=1*, ***, *timeout=1000*, *flow=0*, *timeout_char=0*, *read_buf_len=64*)

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.
- `flow` sets the flow control type. Can be 0, `UART.RTS`, `UART.CTS` or `UART.RTS | UART.CTS`.
- `timeout` is the timeout in milliseconds to wait for writing/reading the first character.
- `timeout_char` is the timeout in milliseconds to wait between characters while writing or reading.
- `read_buf_len` is the character length of the read buffer (0 to disable).

This method will raise an exception if the baudrate could not be set within 5% of the desired value. The minimum baudrate is dictated by the frequency of the bus that the UART is on; UART(1) and UART(6) are APB2, the rest are on APB1. The default bus frequencies give a minimum baudrate of 1300 for UART(1) and UART(6) and 650 for the others. Use *pyb.freq* to reduce the bus frequencies to get lower baudrates.

Note: with parity=None, only 8 and 9 bits are supported. With parity enabled, only 7 and 8 bits are supported.

UART.**deinit**()

Turn off the UART bus.

UART.**any**()

Returns the number of bytes waiting (may be 0).

UART.**read**([*nbytes*])

Read characters. If `nbytes` is specified then read at most that many bytes. If `nbytes` are available in the buffer, returns immediately, otherwise returns when sufficient characters arrive or the timeout elapses.

If `nbytes` is not given then the method reads as much data as possible. It returns after the timeout has elapsed.

Note: for 9 bit characters each character takes two bytes, `nbytes` must be even, and the number of characters is `nbytes/2`.

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

UART.**readchar**()

Receive a single character on the bus.

Return value: The character read, as an integer. Returns -1 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. If such a line exists, return is immediate. If the timeout elapses, all available data is returned regardless of whether a newline exists.

Return value: the line read or `None` on timeout if no data is available.

UART.**write**(*buf*)

Write the buffer of bytes to the bus. If characters are 7 or 8 bits wide then each byte is one character. If characters are 9 bits wide then two bytes are used for each character (little endian), and `buf` must contain an even number of bytes.

Return value: number of bytes written. If a timeout occurs and no bytes were written returns `None`.

`UART.`**`writechar`**`(`*`char`*`)`
Write a single character on the bus. `char` is an integer to write. Return value: `None`. See note below if CTS flow control is used.

`UART.`**`sendbreak`**`()`
Send a break condition on the bus. This drives the bus low for a duration of 13 bits. Return value: `None`.

Constants
`UART.`**`RTS`**
`UART.`**`CTS`**
to select the flow control type.

Flow Control On Pyboards V1 and V1.1 `UART(2)` and `UART(3)` support RTS/CTS hardware flow control using the following pins:

- `UART(2)` is on: `(TX, RX, nRTS, nCTS) = (X3, X4, X2, X1) = (PA2, PA3, PA1, PA0)`
- `UART(3)` is on `:(TX, RX, nRTS, nCTS) = (Y9, Y10, Y7, Y6) = (PB10, PB11, PB14, PB13)`

On the Pyboard Lite only `UART(2)` supports flow control on these pins:

```
(TX, RX, nRTS, nCTS) = (X1, X2, X4, X3) = (PA2, PA3, PA1, PA0)
```

In the following paragraphs the term "target" refers to the device connected to the UART.

When the UART's `init()` method is called with `flow` set to one or both of `UART.RTS` and `UART.CTS` the relevant flow control pins are configured. `nRTS` is an active low output, `nCTS` is an active low input with pullup enabled. To achieve flow control the Pyboard's `nCTS` signal should be connected to the target's `nRTS` and the Pyboard's `nRTS` to the target's `nCTS`.

CTS: target controls Pyboard transmitter If CTS flow control is enabled the write behaviour is as follows:

If the Pyboard's `UART.write(buf)` method is called, transmission will stall for any periods when `nCTS` is `False`. This will result in a timeout if the entire buffer was not transmitted in the timeout period. The method returns the number of bytes written, enabling the user to write the remainder of the data if required. In the event of a timeout, a character will remain in the UART pending `nCTS`. The number of bytes composing this character will be included in the return value.

If `UART.writechar()` is called when `nCTS` is `False` the method will time out unless the target asserts `nCTS` in time. If it times out `OSError 116` will be raised. The character will be transmitted as soon as the target asserts `nCTS`.

RTS: Pyboard controls target's transmitter If RTS flow control is enabled, behaviour is as follows:

If buffered input is used (`read_buf_len` > 0), incoming characters are buffered. If the buffer becomes full, the next character to arrive will cause `nRTS` to go `False`: the target should cease transmission. `nRTS` will go `True` when characters are read from the buffer.

Note that the `any()` method returns the number of bytes in the buffer. Assume a buffer length of `N` bytes. If the buffer becomes full, and another character arrives, `nRTS` will be set False, and `any()` will return the count `N`. When characters are read the additional character will be placed in the buffer and will be included in the result of a subsequent `any()` call.

If buffered input is not used (`read_buf_len` == 0) the arrival of a character will cause `nRTS` to go `False` until the character is read.

class USB_HID – USB Human Interface Device (HID)

The USB_HID class allows creation of an object representing the USB Human Interface Device (HID) interface. It can be used to emulate a peripheral such as a mouse or keyboard.

Before you can use this class, you need to use *pyb.usb_mode()* to set the USB mode to include the HID interface.

Constructors

class pyb.**USB_HID**
Create a new USB_HID object.

Methods

USB_HID.**recv**(*data*, *, *timeout=5000*)
Receive data on the bus:

- data can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes.
- timeout is the timeout in milliseconds to wait for the receive.

Return value: if data is an integer then a new buffer of the bytes received, otherwise the number of bytes read into data is returned.

USB_HID.**send**(*data*)
Send data over the USB HID interface:

- data is the data to send (a tuple/list of integers, or a bytearray).

class USB_VCP – USB virtual comm port

The USB_VCP class allows creation of an object representing the USB virtual comm port. It can be used to read and write data over USB to the connected host.

Constructors

class pyb.**USB_VCP**
Create a new USB_VCP object.

Methods

USB_VCP.**setinterrupt**(*chr*)
Set the character which interrupts running Python code. This is set to 3 (CTRL-C) by default, and when a CTRL-C character is received over the USB VCP port, a KeyboardInterrupt exception is raised.

Set to -1 to disable this interrupt feature. This is useful when you want to send raw bytes over the USB VCP port.

USB_VCP.**isconnected**()
Return True if USB is connected as a serial device, else False.

USB_VCP.**any**()
Return True if any characters waiting, else False.

USB_VCP.**close**()
This method does nothing. It exists so the USB_VCP object can act as a file.

USB_VCP.**read**([*nbytes*])
Read at most `nbytes` from the serial device and return them as a bytes object. If `nbytes` is not specified then the method reads all available bytes from the serial device. USB_VCP stream implicitly works in non-blocking mode, so if no pending data available, this method will return immediately with `None` value.

USB_VCP.**readinto**(*buf*[, *maxlen*])
Read bytes from the serial device and store them into `buf`, which should be a buffer-like object. At most `len(buf)` bytes are read. If `maxlen` is given and then at most `min(maxlen, len(buf))` bytes are read.

Returns the number of bytes read and stored into `buf` or `None` if no pending data available.

USB_VCP.**readline**()
Read a whole line from the serial device.

Returns a bytes object containing the data, including the trailing newline character or `None` if no pending data available.

USB_VCP.**readlines**()
Read as much data as possible from the serial device, breaking it into lines.

Returns a list of bytes objects, each object being one of the lines. Each line will include the newline character.

USB_VCP.**write**(*buf*)
Write the bytes from `buf` to the serial device.

Returns the number of bytes written.

USB_VCP.**recv**(*data*, *, *timeout=5000*)
Receive data on the bus:

- `data` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: if `data` is an integer then a new buffer of the bytes received, otherwise the number of bytes read into `data` is returned.

USB_VCP.**send**(*data*, *, *timeout=5000*)
Send data over the USB VCP:

- `data` is the data to send (an integer to send, or a buffer object).
- `timeout` is the timeout in milliseconds to wait for the send.

Return value: number of bytes sent.

4.3.2 `lcd160cr` — control of LCD160CR display

This module provides control of the MicroPython LCD160CR display.

Further resources are available via the following links:

- LCD160CRv1.0 reference manual (100KiB PDF)
- LCD160CRv1.0 schematics (1.6MiB PDF)

class LCD160CR

The LCD160CR class provides an interface to the display. Create an instance of this class and use its methods to draw to the LCD and get the status of the touch panel.

For example:

```
import lcd160cr

lcd = lcd160cr.LCD160CR('X')
lcd.set_orient(lcd160cr.PORTRAIT)
lcd.set_pos(0, 0)
lcd.set_text_color(lcd.rgb(255, 0, 0), lcd.rgb(0, 0, 0))
lcd.set_font(1)
lcd.write('Hello MicroPython!')
print('touch:', lcd.get_touch())
```

Constructors

class lcd160cr.**LCD160CR**(*connect=None*, *, *pwr=None*, *i2c=None*, *spi=None*, *i2c_addr=98*)

Construct an LCD160CR object. The parameters are:

- *connect* is a string specifying the physical connection of the LCD display to the board; valid values are "X", "Y", "XY", "YX". Use "X" when the display is connected to a pyboard in the X-skin position, and "Y" when connected in the Y-skin position. "XY" and "YX" are used when the display is connected to the right or left side of the pyboard, respectively.
- *pwr* is a Pin object connected to the LCD's power/enabled pin.
- *i2c* is an I2C object connected to the LCD's I2C interface.
- *spi* is an SPI object connected to the LCD's SPI interface.
- *i2c_addr* is the I2C address of the display.

One must specify either a valid *connect* or all of *pwr*, *i2c* and *spi*. If a valid *connect* is given then any of *pwr*, *i2c* or *spi* which are not passed as parameters (i.e. they are `None`) will be created based on the value of *connect*. This allows to override the default interface to the display if needed.

The default values are:

- "X" is for the X-skin and uses: `pwr=Pin("X4")`, `i2c=I2C("X")`, `spi=SPI("X")`
- "Y" is for the Y-skin and uses: `pwr=Pin("Y4")`, `i2c=I2C("Y")`, `spi=SPI("Y")`
- "XY" is for the right-side and uses: `pwr=Pin("X4")`, `i2c=I2C("Y")`, `spi=SPI("X")`
- "YX" is for the left-side and uses: `pwr=Pin("Y4")`, `i2c=I2C("X")`, `spi=SPI("Y")`

See this image for how the display can be connected to the pyboard.

Static methods

static LCD160CR.**rgb**(*r*, *g*, *b*)

Return a 16-bit integer representing the given rgb color values. The 16-bit value can be used to set the font color (see *LCD160CR.set_text_color()*) pen color (see *LCD160CR.set_pen()*) and draw individual pixels.

LCD160CR.clip_line(data, w, h):

Clip the given line data. This is for internal use.

Instance members

The following instance members are publicly accessible.

LCD160CR.**w**

LCD160CR.**h**
The width and height of the display, respectively, in pixels. These members are updated when calling *LCD160CR.set_orient()* and should be considered read-only.

Setup commands

LCD160CR.**set_power**(*on*)
Turn the display on or off, depending on the given value of *on*: 0 or `False` will turn the display off, and 1 or `True` will turn it on.

LCD160CR.**set_orient**(*orient*)
Set the orientation of the display. The *orient* parameter can be one of *PORTRAIT*, *LANDSCAPE*, *PORTRAIT_UPSIDEDOWN*, *LANDSCAPE_UPSIDEDOWN*.

LCD160CR.**set_brightness**(*value*)
Set the brightness of the display, between 0 and 31.

LCD160CR.**set_i2c_addr**(*addr*)
Set the I2C address of the display. The *addr* value must have the lower 2 bits cleared.

LCD160CR.**set_uart_baudrate**(*baudrate*)
Set the baudrate of the UART interface.

LCD160CR.**set_startup_deco**(*value*)
Set the start-up decoration of the display. The *value* parameter can be a logical or of *STARTUP_DECO_NONE*, *STARTUP_DECO_MLOGO*, *STARTUP_DECO_INFO*.

LCD160CR.**save_to_flash**()
Save the following parameters to flash so they persist on restart and power up: initial decoration, orientation, brightness, UART baud rate, I2C address.

Pixel access methods

The following methods manipulate individual pixels on the display.

LCD160CR.**set_pixel**(*x*, *y*, *c*)
Set the specified pixel to the given color. The color should be a 16-bit integer and can be created by *LCD160CR.rgb()*.

LCD160CR.**get_pixel**(*x*, *y*)
Get the 16-bit value of the specified pixel.

LCD160CR.**get_line**(*x*, *y*, *buf*)
Low-level method to get a line of pixels into the given buffer. To read *n* pixels *buf* should be *2*n+1* bytes in length. The first byte is a dummy byte and should be ignored, and subsequent bytes represent the pixels in the line starting at coordinate *(x, y)*.

LCD160CR.**screen_dump**(*buf*, *x=0*, *y=0*, *w=None*, *h=None*)
Dump the contents of the screen to the given buffer. The parameters *x* and *y* specify the starting coordinate, and *w* and *h* the size of the region. If *w* or *h* are `None` then they will take on their maximum values, set by the size of the screen minus the given *x* and *y* values. *buf* should be large enough to hold `2*w*h` bytes. If it's smaller then only the initial horizontal lines will be stored.

LCD160CR.**screen_load**(*buf*)
Load the entire screen from the given buffer.

Drawing text

To draw text one sets the position, color and font, and then uses *write* to draw the text.

LCD160CR.**set_pos**(*x*, *y*)
Set the position for text output using *LCD160CR.write()*. The position is the upper-left corner of the text.

LCD160CR.**set_text_color**(*fg*, *bg*)
Set the foreground and background color of the text.

LCD160CR.**set_font**(*font*, *scale=0*, *bold=0*, *trans=0*, *scroll=0*)
Set the font for the text. Subsequent calls to *write* will use the newly configured font. The parameters are:

- *font* is the font family to use, valid values are 0, 1, 2, 3.
- *scale* is a scaling value for each character pixel, where the pixels are drawn as a square with side length equal to *scale* + *1*. The value can be between 0 and 63.
- *bold* controls the number of pixels to overdraw each character pixel, making a bold effect. The lower 2 bits of *bold* are the number of pixels to overdraw in the horizontal direction, and the next 2 bits are for the vertical direction. For example, a *bold* value of 5 will overdraw 1 pixel in both the horizontal and vertical directions.
- *trans* can be either 0 or 1 and if set to 1 the characters will be drawn with a transparent background.
- *scroll* can be either 0 or 1 and if set to 1 the display will do a soft scroll if the text moves to the next line.

LCD160CR.**write**(*s*)
Write text to the display, using the current position, color and font. As text is written the position is automatically incremented. The display supports basic VT100 control codes such as newline and backspace.

Drawing primitive shapes

Primitive drawing commands use a foreground and background color set by the *set_pen* method.

LCD160CR.**set_pen**(*line*, *fill*)
Set the line and fill color for primitive shapes.

LCD160CR.**erase**()
Erase the entire display to the pen fill color.

LCD160CR.**dot**(*x*, *y*)
Draw a single pixel at the given location using the pen line color.

LCD160CR.**rect**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_outline**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_interior**(*x*, *y*, *w*, *h*)
Draw a rectangle at the given location and size using the pen line color for the outline, and the pen fill color for the interior. The *rect* method draws the outline and interior, while the other methods just draw one or the other.

LCD160CR.**line**(*x1*, *y1*, *x2*, *y2*)
Draw a line between the given coordinates using the pen line color.

LCD160CR.**dot_no_clip**(*x*, *y*)

LCD160CR.**rect_no_clip**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_outline_no_clip**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_interior_no_clip**(*x*, *y*, *w*, *h*)

LCD160CR.**line_no_clip**(*x1*, *y1*, *x2*, *y2*)
These methods are as above but don't do any clipping on the input coordinates. They are faster than the clipping versions and can be used when you know that the coordinates are within the display.

LCD160CR.**poly_dot**(*data*)
Draw a sequence of dots using the pen line color. The *data* should be a buffer of bytes, with each successive pair of bytes corresponding to coordinate pairs (x, y).

LCD160CR.**poly_line**(*data*)
Similar to *LCD160CR.poly_dot()* but draws lines between the dots.

Touch screen methods

LCD160CR.**touch_config**(*calib=False*, *save=False*, *irq=None*)
Configure the touch panel:

- If *calib* is `True` then the call will trigger a touch calibration of the resistive touch sensor. This requires the user to touch various parts of the screen.
- If *save* is `True` then the touch parameters will be saved to NVRAM to persist across reset/power up.
- If *irq* is `True` then the display will be configured to pull the IRQ line low when a touch force is detected. If *irq* is `False` then this feature is disabled. If *irq* is `None` (the default value) then no change is made to this setting.

LCD160CR.**is_touched**()
Returns a boolean: `True` if there is currently a touch force on the screen, `False` otherwise.

LCD160CR.**get_touch**()
Returns a 3-tuple of: *(active, x, y)*. If there is currently a touch force on the screen then *active* is 1, otherwise it is 0. The *x* and *y* values indicate the position of the current or most recent touch.

Advanced commands

LCD160CR.**set_spi_win**(*x*, *y*, *w*, *h*)
Set the window that SPI data is written to.

LCD160CR.**fast_spi**(*flush=True*)
Ready the display to accept RGB pixel data on the SPI bus, resetting the location of the first byte to go to the top-left corner of the window set by *LCD160CR.set_spi_win()*. The method returns an SPI object which can be used to write the pixel data.

Pixels should be sent as 16-bit RGB values in the 5-6-5 format. The destination counter will increase as data is sent, and data can be sent in arbitrary sized chunks. Once the destination counter reaches the end of the window specified by *LCD160CR.set_spi_win()* it will wrap around to the top-left corner of that window.

LCD160CR.**show_framebuf**(*buf*)
Show the given buffer on the display. *buf* should be an array of bytes containing the 16-bit RGB values for the pixels, and they will be written to the area specified by *LCD160CR.set_spi_win()*, starting from the top-left corner.

The framebuf module can be used to construct frame buffers and provides drawing primitives. Using a frame buffer will improve performance of animations when compared to drawing directly to the screen.

LCD160CR.**set_scroll**(*on*)
Turn scrolling on or off. This controls globally whether any window regions will scroll.

LCD160CR.**set_scroll_win**(*win*, *x=-1*, *y=0*, *w=0*, *h=0*, *vec=0*, *pat=0*, *fill=0x07e0*, *color=0*)
Configure a window region for scrolling:

- *win* is the window id to configure. There are 0..7 standard windows for general purpose use. Window 8 is the text scroll window (the ticker).
- *x*, *y*, *w*, *h* specify the location of the window in the display.
- *vec* specifies the direction and speed of scroll: it is a 16-bit value of the form `0bF.ddSSSSSSSSSSSS`. *dd* is 0, 1, 2, 3 for +x, +y, -x, -y scrolling. *F* sets the speed format, with 0 meaning that the window is shifted *S % 256* pixel every frame, and 1 meaning that the window is shifted 1 pixel every *S* frames.
- *pat* is a 16-bit pattern mask for the background.
- *fill* is the fill color.
- *color* is the extra color, either of the text or pattern foreground.

LCD160CR.**set_scroll_win_param**(*win*, *param*, *value*)
Set a single parameter of a scrolling window region:

- *win* is the window id, 0..8.
- *param* is the parameter number to configure, 0..7, and corresponds to the parameters in the *set_scroll_win* method.
- *value* is the value to set.

LCD160CR.**set_scroll_buf**(*s*)
Set the string for scrolling in window 8. The parameter *s* must be a string with length 32 or less.

LCD160CR.**jpeg**(*buf*)
Display a JPEG. *buf* should contain the entire JPEG data. JPEG data should not include EXIF information. The following encodings are supported: Baseline DCT, Huffman coding, 8 bits per sample, 3 color components, YCbCr4:2:2. The origin of the JPEG is set by *LCD160CR.set_pos()*.

LCD160CR.**jpeg_start**(*total_len*)

LCD160CR.**jpeg_data**(*buf*)
Display a JPEG with the data split across multiple buffers. There must be a single call to *jpeg_start* to begin with, specifying the total number of bytes in the JPEG. Then this number of bytes must be transferred to the display using one or more calls to the *jpeg_data* command.

LCD160CR.**feed_wdt**()
The first call to this method will start the display's internal watchdog timer. Subsequent calls will feed the watchdog. The timeout is roughly 30 seconds.

LCD160CR.**reset**()
Reset the display.

Constants

lcd160cr.**PORTRAIT**
lcd160cr.**LANDSCAPE**
lcd160cr.**PORTRAIT_UPSIDEDOWN**
lcd160cr.**LANDSCAPE_UPSIDEDOWN**
Orientations of the display, used by *LCD160CR.set_orient()*.

lcd160cr.**STARTUP_DECO_NONE**
lcd160cr.**STARTUP_DECO_MLOGO**
lcd160cr.**STARTUP_DECO_INFO**
Types of start-up decoration, can be OR'ed together, used by *LCD160CR.set_startup_deco()*.

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 (full-fledged) OS, like an *MCU*. 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*).

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).

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, MacOS, 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.

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(3):
...     _
```

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

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

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

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

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

```
>>> for i in range(30):
...     if i > 3:
...         break
...     print(i)
...
0
1
2
3
>>>
```

Auto-indent won't be applied if the previous two lines were all spaces. This means that you can finish entering a compound statement by pressing RETURN twice, and then a third press will finish and execute.

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 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:

```
raise SystemExit
```

For example, if you reset your MicroPython board, and you execute a dir() command, you'd see something like this:

```
>>> dir()
['__name__', 'pyb']
```

Now create some variables and repeat the dir() command:

```
>>> i = 1
>>> j = 23
>>> x = 'abc'
>>> dir()
['j', 'x', '__name__', 'pyb', 'i']
>>>
```

Now if you enter Ctrl-D, and repeat the dir() command, you'll see that your variables no longer exist:

```
PYB: sync filesystems
PYB: soft reboot
MicroPython v1.5-51-g6f70283-dirty on 2015-10-30; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>> dir()
['__name__', 'pyb']
>>>
```

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.
- 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))
greeen = Foo(pyb.Timer(2, freq=0.8), pyb.LED(2))
```

In this example the `red` instance associates timer 4 with LED 1: when a timer 4 interrupt occurs `red.cb()` is called causing LED 1 to change state. The `green` instance operates similarly: a timer 2 interrupt results in the execution of `green.cb()` and toggles LED 2. The use of instance methods confers two benefits. Firstly a single class enables code to be shared between multiple hardware instances. Secondly, as a bound method the callback function's first argument is `self`. This enables the callback to access instance data and to save state between successive calls. For example, if the class above had a variable `self.count` set to zero in the constructor, `cb()` could increment the counter. The `red` and `green` instances would then maintain independent counts of the number of times each LED had changed state.

Creation of Python objects

ISR's cannot create instances of Python objects. This is because MicroPython needs to allocate memory for the object from a store of free memory block called the heap. This is not permitted in an interrupt handler because heap allocation is not re-entrant. In other words the interrupt might occur when the main program is part way through performing an allocation - to maintain the integrity of the heap the interpreter disallows memory allocations in ISR code.

A consequence of this is that ISR's can't use floating point arithmetic; this is because floats are Python objects. Similarly an ISR can't append an item to a list. In practice it can be hard to determine exactly which code constructs will attempt to perform memory allocation and provoke an error message: another reason for keeping ISR code short and simple.

One way to avoid this issue is for the ISR to use pre-allocated buffers. For example a class constructor creates a `bytearray` instance and a boolean flag. The ISR method assigns data to locations in the buffer and sets the flag. The memory allocation occurs in the main program code when the object is instantiated rather than in the ISR.

The MicroPython library I/O methods usually provide an option to use a pre-allocated buffer. For example `pyb.i2c.recv()` can accept a mutable buffer as its first argument: this enables its use in an ISR.

A means of creating an object without employing a class or globals is as follows:

```
def set_volume(t, buf=bytearray(3)):
    buf[0] = 0xa5
    buf[1] = t >> 4
    buf[2] = 0x5a
    return buf
```

The compiler instantiates the default `buf` argument when the function is loaded for the first time (usually when the module it's in is imported).

Use of Python objects

A further restriction on objects arises because of the way Python works. When an `import` statement is executed the Python code is compiled to bytecode, with one line of code typically mapping to multiple bytecodes. When the code runs the interpreter reads each bytecode and executes it as a series of machine code instructions. Given that an interrupt can occur at any time between machine code instructions, the original line of Python code may be only partially executed. Consequently a Python object such as a set, list or dictionary modified in the main loop may lack internal consistency at the moment the interrupt occurs.

A typical outcome is as follows. On rare occasions the ISR will run at the precise moment in time when the object is partially updated. When the ISR tries to read the object, a crash results. Because such problems typically occur on

rare, random occasions they can be hard to diagnose. There are ways to circumvent this issue, described in *Critical Sections* below.

It is important to be clear about what constitutes the modification of an object. An alteration to a built-in type such as a dictionary is problematic. Altering the contents of an array or bytearray is not. This is because bytes or words are written as a single machine code instruction which is not interruptible: in the parlance of real time programming the write is atomic. A user defined object might instantiate an integer, array or bytearray. It is valid for both the main loop and the ISR to alter the contents of these.

MicroPython supports integers of arbitrary precision. Values between 2**30 -1 and -2**30 will be stored in a single machine word. Larger values are stored as Python objects. Consequently changes to long integers cannot be considered atomic. The use of long integers in ISR's is unsafe because memory allocation may be attempted as the variable's value changes.

Overcoming the float limitation

In general it is best to avoid using floats in ISR code: hardware devices normally handle integers and conversion to floats is normally done in the main loop. However there are a few DSP algorithms which require floating point. On platforms with hardware floating point (such as the Pyboard) the inline ARM Thumb assembler can be used to work round this limitation. This is because the processor stores float values in a machine word; values can be shared between the ISR and main program code via an array of floats.

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

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 stmhal (or the equivalent directory for the architecture in use). Copy and paste the corrected lines at the end of the file. Save the file, rebuild and flash the firmware. The outcome can be checked by importing the modules and again issuing:

```
micropython.qstr_info(1)
```

The Q(xxx) lines should be gone.

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 Inline Assembler for Thumb2 architectures

This document assumes some familiarity with assembly language programming and should be read after studying the *tutorial*. For a detailed description of the instruction set consult the Architecture Reference Manual detailed below. The inline assembler supports a subset of the ARM Thumb-2 instruction set described here. The syntax tries to be as close as possible to that defined in the above ARM manual, converted to Python function calls.

Instructions operate on 32 bit signed integer data except where stated otherwise. Most supported instructions operate on registers `R0-R7` only: where `R8-R15` are supported this is stated. Registers `R8-R12` must be restored to their initial value before return from a function. Registers `R13-R15` constitute the Link Register, Stack Pointer and Program Counter respectively.

5.6.1 Document conventions

Where possible the behaviour of each instruction is described in Python, for example

- add(Rd, Rn, Rm) `Rd = Rn + Rm`

This enables the effect of instructions to be demonstrated in Python. In certain case this is impossible because Python doesn't support concepts such as indirection. The pseudocode employed in such cases is described on the relevant page.

5.6.2 Instruction Categories

The following sections details the subset of the ARM Thumb-2 instruction set supported by MicroPython.

Register move instructions

Document conventions

Notation: `Rd, Rn` denote ARM registers R0-R15. `immN` denotes an immediate value having a width of N bits. These instructions affect the condition flags.

Register moves

Where immediate values are used, these are zero-extended to 32 bits. Thus `mov(R0, 0xff)` will set R0 to 255.

- mov(Rd, imm8) `Rd = imm8`
- mov(Rd, Rn) `Rd = Rn`
- movw(Rd, imm16) `Rd = imm16`
- movt(Rd, imm16) `Rd = (Rd & 0xffff) | (imm16 << 16)`

movt writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

- movwt(Rd, imm32) `Rd = imm32`

movwt is a pseudo-instruction: the MicroPython assembler emits a `movw` followed by a `movt` to move a 32-bit value into Rd.

Load register from memory

Document conventions

Notation: `Rt, Rn` denote ARM registers R0-R7 except where stated. `immN` represents an immediate value having a width of N bits hence `imm5` is constrained to the range 0-31. `[Rn + immN]` is the contents of the memory address obtained by adding Rn and the offset `immN`. Offsets are measured in bytes. These instructions affect the condition flags.

Register Load

- ldr(Rt, [Rn, imm7]) `Rt = [Rn + imm7]` Load a 32 bit word
- ldrb(Rt, [Rn, imm5]) `Rt = [Rn + imm5]` Load a byte
- ldrh(Rt, [Rn, imm6]) `Rt = [Rn + imm6]` Load a 16 bit half word

Where a byte or half word is loaded, it is zero-extended to 32 bits.

The specified immediate offsets are measured in bytes. Hence in the case of `ldr` the 7 bit value enables 32 bit word aligned values to be accessed with a maximum offset of 31 words. In the case of `ldrh` the 6 bit value enables 16 bit half-word aligned values to be accessed with a maximum offset of 31 half-words.

Store register to memory

Document conventions

Notation: `Rt, Rn` denote ARM registers R0-R7 except where stated. `immN` represents an immediate value having a width of N bits hence `imm5` is constrained to the range 0-31. `[Rn + imm5]` is the contents of the memory address obtained by adding Rn and the offset `imm5`. Offsets are measured in bytes. These instructions do not affect the condition flags.

Register Store

- str(Rt, [Rn, imm7]) `[Rn + imm7] = Rt` Store a 32 bit word
- strb(Rt, [Rn, imm5]) `[Rn + imm5] = Rt` Store a byte (b0-b7)
- strh(Rt, [Rn, imm6]) `[Rn + imm6] = Rt` Store a 16 bit half word (b0-b15)

The specified immediate offsets are measured in bytes. Hence in the case of `str` the 7 bit value enables 32 bit word aligned values to be accessed with a maximum offset of 31 words. In the case of `strh` the 6 bit value enables 16 bit half-word aligned values to be accessed with a maximum offset of 31 half-words.

Logical & Bitwise instructions

Document conventions

Notation: `Rd, Rn` denote ARM registers R0-R7 except in the case of the special instructions where R0-R15 may be used. `Rn<a-b>` denotes an ARM register whose contents must lie in range `a <= contents <= b`. In the case of instructions with two register arguments, it is permissible for them to be identical. For example the following will zero R0 (Python `R0 ^= R0`) regardless of its initial contents.

- eor(r0, r0)

These instructions affect the condition flags except where stated.

Logical instructions

- and_(Rd, Rn) `Rd &= Rn`
- orr(Rd, Rn) `Rd |= Rn`
- eor(Rd, Rn) `Rd ^= Rn`
- mvn(Rd, Rn) `Rd = Rn ^ 0xffffffff` i.e. Rd = 1's complement of Rn
- bic(Rd, Rn) `Rd &= ~Rn` bit clear Rd using mask in Rn

Note the use of "and_" instead of "and", because "and" is a reserved keyword in Python.

Shift and rotation instructions

- lsl(Rd, Rn<0-31>) `Rd <<= Rn`
- lsr(Rd, Rn<1-32>) `Rd = (Rd & 0xffffffff) >> Rn` Logical shift right
- asr(Rd, Rn<1-32>) `Rd >>= Rn` arithmetic shift right

- ror(Rd, Rn<1-31>) `Rd = rotate_right(Rd, Rn)` Rd is rotated right Rn bits.

A rotation by (for example) three bits works as follows. If Rd initially contains bits `b31 b30..b0` after rotation it will contain `b2 b1 b0 b31 b30..b3`

Special instructions

Condition codes are unaffected by these instructions.

- clz(Rd, Rn) `Rd = count_leading_zeros(Rn)`

count_leading_zeros(Rn) returns the number of binary zero bits before the first binary one bit in Rn.

- rbit(Rd, Rn) `Rd = bit_reverse(Rn)`

bit_reverse(Rn) returns the bit-reversed contents of Rn. If Rn contains bits `b31 b30..b0` Rd will be set to `b0 b1 b2..b31`

Trailing zeros may be counted by performing a bit reverse prior to executing clz.

Arithmetic instructions

Document conventions

Notation: `Rd, Rm, Rn` denote ARM registers R0-R7. `immN` denotes an immediate value having a width of N bits e.g. `imm8`, `imm3`. `carry` denotes the carry condition flag, `not(carry)` denotes its complement. In the case of instructions with more than one register argument, it is permissible for some to be identical. For example the following will add the contents of R0 to itself, placing the result in R0:

- add(r0, r0, r0)

Arithmetic instructions affect the condition flags except where stated.

Addition

- add(Rdn, imm8) `Rdn = Rdn + imm8`
- add(Rd, Rn, imm3) `Rd = Rn + imm3`
- add(Rd, Rn, Rm) `Rd = Rn +Rm`
- adc(Rd, Rn) `Rd = Rd + Rn + carry`

Subtraction

- sub(Rdn, imm8) `Rdn = Rdn - imm8`
- sub(Rd, Rn, imm3) `Rd = Rn - imm3`
- sub(Rd, Rn, Rm) `Rd = Rn - Rm`
- sbc(Rd, Rn) `Rd = Rd - Rn - not(carry)`

Negation

- neg(Rd, Rn) `Rd = -Rn`

Multiplication and division

- mul(Rd, Rn) `Rd = Rd * Rn`

This produces a 32 bit result with overflow lost. The result may be treated as signed or unsigned according to the definition of the operands.

- sdiv(Rd, Rn, Rm) `Rd = Rn / Rm`
- udiv(Rd, Rn, Rm) `Rd = Rn / Rm`

These functions perform signed and unsigned division respectively. Condition flags are not affected.

Comparison instructions

These perform an arithmetic or logical instruction on two arguments, discarding the result but setting the condition flags. Typically these are used to test data values without changing them prior to executing a conditional branch.

Document conventions

Notation: `Rd, Rm, Rn` denote ARM registers R0-R7. `imm8` denotes an immediate value having a width of 8 bits.

The Application Program Status Register (APSR)

This contains four bits which are tested by the conditional branch instructions. Typically a conditional branch will test multiple bits, for example `bge(LABEL)`. The meaning of condition codes can depend on whether the operands of an arithmetic instruction are viewed as signed or unsigned integers. Thus `bhi(LABEL)` assumes unsigned numbers were processed while `bgt(LABEL)` assumes signed operands.

APSR Bits

- Z (zero)

This is set if the result of an operation is zero or the operands of a comparison are equal.

- N (negative)

Set if the result is negative.

- C (carry)

An addition sets the carry flag when the result overflows out of the MSB, for example adding 0x80000000 and 0x80000000. By the nature of two's complement arithmetic this behaviour is reversed on subtraction, with a borrow indicated by the carry bit being clear. Thus 0x10 - 0x01 is executed as 0x10 + 0xffffffff which will set the carry bit.

- V (overflow)

The overflow flag is set if the result, viewed as a two's compliment number, has the "wrong" sign in relation to the operands. For example adding 1 to 0x7fffffff will set the overflow bit because the result (0x8000000), viewed as a two's complement integer, is negative. Note that in this instance the carry bit is not set.

Comparison instructions

These set the APSR (Application Program Status Register) N (negative), Z (zero), C (carry) and V (overflow) flags.

- cmp(Rn, imm8) `Rn - imm8`
- cmp(Rn, Rm) `Rn - Rm`
- cmn(Rn, Rm) `Rn + Rm`
- tst(Rn, Rm) `Rn & Rm`

Conditional execution

The `it` and `ite` instructions provide a means of conditionally executing from one to four subsequent instructions without the need for a label.

- it(<condition>) If then

Execute the next instruction if <condition> is true:

```
cmp(r0, r1)
it(eq)
mov(r0, 100) # runs if r0 == r1
# execution continues here
```

- ite(<condition>) If then else

If <condtion> is true, execute the next instruction, otherwise execute the subsequent one. Thus:

```
cmp(r0, r1)
ite(eq)
mov(r0, 100) # runs if r0 == r1
mov(r0, 200) # runs if r0 != r1
# execution continues here
```

This may be extended to control the execution of upto four subsequent instructions: it[x[y[z]]] where x,y,z=t/e; e.g. itt, itee, itete, ittte, itttt, iteee, etc.

Branch instructions

These cause execution to jump to a target location usually specified by a label (see the `label` assembler directive). Conditional branches and the `it` and `ite` instructions test the Application Program Status Register (APSR) N (negative), Z (zero), C (carry) and V (overflow) flags to determine whether the branch should be executed.

Most of the exposed assembler instructions (including move operations) set the flags but there are explicit comparison instructions to enable values to be tested.

Further detail on the meaning of the condition flags is provided in the section describing comparison functions.

Document conventions

Notation: `Rm` denotes ARM registers R0-R15. `LABEL` denotes a label defined with the `label()` assembler directive. `<condition>` indicates one of the following condition specifiers:

- eq Equal to (result was zero)
- ne Not equal

- cs Carry set
- cc Carry clear
- mi Minus (negative)
- pl Plus (positive)
- vs Overflow set
- vc Overflow clear
- hi > (unsigned comparison)
- ls <= (unsigned comparison)
- ge >= (signed comparison)
- lt < (signed comparison)
- gt > (signed comparison)
- le <= (signed comparison)

Branch to label

- b(LABEL) Unconditional branch
- beq(LABEL) branch if equal
- bne(LABEL) branch if not equal
- bge(LABEL) branch if greater than or equal
- bgt(LABEL) branch if greater than
- blt(LABEL) branch if less than (<) (signed)
- ble(LABEL) branch if less than or equal to (<=) (signed)
- bcs(LABEL) branch if carry flag is set
- bcc(LABEL) branch if carry flag is clear
- bmi(LABEL) branch if negative
- bpl(LABEL) branch if positive
- bvs(LABEL) branch if overflow flag set
- bvc(LABEL) branch if overflow flag is clear
- bhi(LABEL) branch if higher (unsigned)
- bls(LABEL) branch if lower or equal (unsigned)

Long branches

The code produced by the branch instructions listed above uses a fixed bit width to specify the branch destination, which is PC relative. Consequently in long programs where the branch instruction is remote from its destination the assembler will produce a "branch not in range" error. This can be overcome with the "wide" variants such as

- beq_w(LABEL) long branch if equal

Wide branches use 4 bytes to encode the instruction (compared with 2 bytes for standard branch instructions).

Subroutines (functions)

When entering a subroutine the processor stores the return address in register r14, also known as the link register (lr). Return to the instruction after the subroutine call is performed by updating the program counter (r15 or pc) from the link register, This process is handled by the following instructions.

- bl(LABEL)

Transfer execution to the instruction after `LABEL` storing the return address in the link register (r14).

- bx(Rm) Branch to address specified by Rm.

Typically `bx(lr)` is issued to return from a subroutine. For nested subroutines the link register of outer scopes must be saved (usually on the stack) before performing inner subroutine calls.

Stack push and pop

Document conventions

The `push()` and `pop()` instructions accept as their argument a register set containing a subset, or possibly all, of the general-purpose registers R0-R12 and the link register (lr or R14). As with any Python set the order in which the registers are specified is immaterial. Thus the in the following example the pop() instruction would restore R1, R7 and R8 to their contents prior to the push():

- push({r1, r8, r7}) Save three registers on the stack.
- pop({r7, r1, r8}) Restore them

Stack operations

- push({regset}) Push a set of registers onto the stack
- pop({regset}) Restore a set of registers from the stack

Miscellaneous instructions

- nop() `pass` no operation.
- wfi() Suspend execution in a low power state until an interrupt occurs.
- cpsid(flags) set the Priority Mask Register - disable interrupts.
- cpsie(flags) clear the Priority Mask Register - enable interrupts.
- mrs(Rd, special_reg) `Rd = special_reg` copy a special register to a general register. The special register may be IPSR (Interrupt Status Register) or BASEPRI (Base Priority Register). The IPSR provides a means of determining the exception number of an interrupt being processed. It contains zero if no interrupt is being processed.

Currently the `cpsie()` and `cpsid()` functions are partially implemented. They require but ignore the flags argument and serve as a means of enabling and disabling interrupts.

Floating Point instructions

These instructions support the use of the ARM floating point coprocessor (on platforms such as the Pyboard which are equipped with one). The FPU has 32 registers known as `s0-s31` each of which can hold a single precision float. Data can be passed between the FPU registers and the ARM core registers with the `vmov` instruction.

Note that MicroPython doesn't support passing floats to assembler functions, nor can you put a float into `r0` and expect a reasonable result. There are two ways to overcome this. The first is to use arrays, and the second is to pass and/or return integers and convert to and from floats in code.

Document conventions

Notation: `Sd, Sm, Sn` denote FPU registers, `Rd, Rm, Rn` denote ARM core registers. The latter can be any ARM core register although registers `R13-R15` are unlikely to be appropriate in this context.

Arithmetic

- vadd(Sd, Sn, Sm) `Sd = Sn + Sm`
- vsub(Sd, Sn, Sm) `Sd = Sn - Sm`
- vneg(Sd, Sm) `Sd = -Sm`
- vmul(Sd, Sn, Sm) `Sd = Sn * Sm`
- vdiv(Sd, Sn, Sm) `Sd = Sn / Sm`
- vsqrt(Sd, Sm) `Sd = sqrt(Sm)`

Registers may be identical: `vmul(S0, S0, S0)` will execute `S0 = S0*S0`

Move between ARM core and FPU registers

- vmov(Sd, Rm) `Sd = Rm`
- vmov(Rd, Sm) `Rd = Sm`

The FPU has a register known as FPSCR, similar to the ARM core's APSR, which stores condition codes plus other data. The following instructions provide access to this.

- vmrs(APSR_nzcv, FPSCR)

Move the floating-point N, Z, C, and V flags to the APSR N, Z, C, and V flags.

This is done after an instruction such as an FPU comparison to enable the condition codes to be tested by the assembler code. The following is a more general form of the instruction.

- vmrs(Rd, FPSCR) `Rd = FPSCR`

Move between FPU register and memory

- vldr(Sd, [Rn, offset]) `Sd = [Rn + offset]`
- vstr(Sd, [Rn, offset]) `[Rn + offset] = Sd`

Where `[Rn + offset]` denotes the memory address obtained by adding Rn to the offset. This is specified in bytes. Since each float value occupies a 32 bit word, when accessing arrays of floats the offset must always be a multiple of four bytes.

Data Comparison

- vcmp(Sd, Sm)

Compare the values in Sd and Sm and set the FPU N, Z, C, and V flags. This would normally be followed by `vmrs(APSR_nzcv, FPSCR)` to enable the results to be tested.

Convert between integer and float

- vcvt_f32_s32(Sd, Sm) `Sd = float(Sm)`
- vcvt_s32_f32(Sd, Sm) `Sd = int(Sm)`

Assembler Directives

Labels

- label(INNER1)

This defines a label for use in a branch instruction. Thus elsewhere in the code a `b(INNER1)` will cause execution to continue with the instruction after the label directive.

Defining inline data

The following assembler directives facilitate embedding data in an assembler code block.

- data(size, d0, d1 .. dn)

The data directive creates n array of data values in memory. The first argument specifies the size in bytes of the subsequent arguments. Hence the first statement below will cause the assembler to put three bytes (with values 2, 3 and 4) into consecutive memory locations while the second will cause it to emit two four byte words.

```
data(1, 2, 3, 4)
data(4, 2, 100000)
```

Data values longer than a single byte are stored in memory in little-endian format.

- align(nBytes)

Align the following instruction to an nBytes value. ARM Thumb-2 instructions must be two byte aligned, hence it's advisable to issue `align(2)` after `data` directives and prior to any subsequent code. This ensures that the code will run irrespective of the size of the data array.

5.6.3 Usage examples

These sections provide further code examples and hints on the use of the assembler.

Hints and tips

The following are some examples of the use of the inline assembler and some information on how to work around its limitations. In this document the term "assembler function" refers to a function declared in Python with the `@micropython.asm_thumb` decorator, whereas "subroutine" refers to assembler code called from within an assembler function.

Code branches and subroutines

It is important to appreciate that labels are local to an assembler function. There is currently no way for a subroutine defined in one function to be called from another.

To call a subroutine the instruction `bl(LABEL)` is issued. This transfers control to the instruction following the `label(LABEL)` directive and stores the return address in the link register (`lr` or `r14`). To return the instruction `bx(lr)` is issued which causes execution to continue with the instruction following the subroutine call. This mechanism implies that, if a subroutine is to call another, it must save the link register prior to the call and restore it before terminating.

The following rather contrived example illustrates a function call. Note that it's necessary at the start to branch around all subroutine calls: subroutines end execution with `bx(lr)` while the outer function simply "drops off the end" in the style of Python functions.

```
@micropython.asm_thumb
def quad(r0):
    b(START)
    label(DOUBLE)
    add(r0, r0, r0)
    bx(lr)
    label(START)
    bl(DOUBLE)
    bl(DOUBLE)

print(quad(10))
```

The following code example demonstrates a nested (recursive) call: the classic Fibonacci sequence. Here, prior to a recursive call, the link register is saved along with other registers which the program logic requires to be preserved.

```
@micropython.asm_thumb
def fib(r0):
    b(START)
    label(DOFIB)
    push({r1, r2, lr})
    cmp(r0, 1)
    ble(FIBDONE)
    sub(r0, 1)
    mov(r2, r0) # r2 = n -1
    bl(DOFIB)
    mov(r1, r0) # r1 = fib(n -1)
    sub(r0, r2, 1)
    bl(DOFIB)   # r0 = fib(n -2)
    add(r0, r0, r1)
    label(FIBDONE)
    pop({r1, r2, lr})
    bx(lr)
    label(START)
    bl(DOFIB)

for n in range(10):
    print(fib(n))
```

Argument passing and return

The tutorial details the fact that assembler functions can support from zero to three arguments, which must (if used) be named `r0`, `r1` and `r2`. When the code executes the registers will be initialised to those values.

The data types which can be passed in this way are integers and memory addresses. With current firmware all possible 32 bit values may be passed and returned. If the return value may have the most significant bit set a Python type hint should be employed to enable MicroPython to determine whether the value should be interpreted as a signed or unsigned integer: types are `int` or `uint`.

```
@micropython.asm_thumb
def uadd(r0, r1) -> uint:
    add(r0, r0, r1)
```

`hex(uadd(0x40000000,0x40000000))` will return 0x80000000, demonstrating the passing and return of integers where bits 30 and 31 differ.

The limitations on the number of arguments and return values can be overcome by means of the `array` module which enables any number of values of any type to be accessed.

Multiple arguments If a Python array of integers is passed as an argument to an assembler function, the function will receive the address of a contiguous set of integers. Thus multiple arguments can be passed as elements of a single array. Similarly a function can return multiple values by assigning them to array elements. Assembler functions have no means of determining the length of an array: this will need to be passed to the function.

This use of arrays can be extended to enable more than three arrays to be used. This is done using indirection: the `uctypes` module supports `addressof()` which will return the address of an array passed as its argument. Thus you can populate an integer array with the addresses of other arrays:

```
from uctypes import addressof
@micropython.asm_thumb
def getindirect(r0):
    ldr(r0, [r0, 0]) # Address of array loaded from passed array
    ldr(r0, [r0, 4]) # Return element 1 of indirect array (24)

def testindirect():
    a = array.array('i',[23, 24])
    b = array.array('i',[0,0])
    b[0] = addressof(a)
    print(getindirect(b))
```

Non-integer data types These may be handled by means of arrays of the appropriate data type. For example, single precision floating point data may be processed as follows. This code example takes an array of floats and replaces its contents with their squares.

```
from array import array

@micropython.asm_thumb
def square(r0, r1):
    label(LOOP)
    vldr(s0, [r0, 0])
    vmul(s0, s0, s0)
    vstr(s0, [r0, 0])
    add(r0, 4)
    sub(r1, 1)
    bgt(LOOP)

a = array('f', (x for x in range(10)))
square(a, len(a))
print(a)
```

The uctypes module supports the use of data structures beyond simple arrays. It enables a Python data structure to be mapped onto a bytearray instance which may then be passed to the assembler function.

Named constants

Assembler code may be made more readable and maintainable by using named constants rather than littering code with numbers. This may be achieved thus:

```
MYDATA = const(33)

@micropython.asm_thumb
def foo():
    mov(r0, MYDATA)
```

The const() construct causes MicroPython to replace the variable name with its value at compile time. If constants are declared in an outer Python scope they can be shared between multiple assembler functions and with Python code.

Assembler code as class methods

MicroPython passes the address of the object instance as the first argument to class methods. This is normally of little use to an assembler function. It can be avoided by declaring the function as a static method thus:

```
class foo:
  @staticmethod
  @micropython.asm_thumb
  def bar(r0):
    add(r0, r0, r0)
```

Use of unsupported instructions

These can be coded using the data statement as shown below. While `push()` and `pop()` are supported the example below illustrates the principle. The necessary machine code may be found in the ARM v7-M Architecture Reference Manual. Note that the first argument of data calls such as

```
data(2, 0xe92d, 0x0f00) # push r8,r9,r10,r11
```

indicates that each subsequent argument is a two byte quantity.

Overcoming MicroPython's integer restriction

The Pyboard chip includes a CRC generator. Its use presents a problem in MicroPython because the returned values cover the full gamut of 32 bit quantities whereas small integers in MicroPython cannot have differing values in bits 30 and 31. This limitation is overcome with the following code, which uses assembler to put the result into an array and Python code to coerce the result into an arbitrary precision unsigned integer.

```
from array import array
import stm

def enable_crc():
    stm.mem32[stm.RCC + stm.RCC_AHB1ENR] |= 0x1000

def reset_crc():
    stm.mem32[stm.CRC+stm.CRC_CR] = 1
```

```
@micropython.asm_thumb
def getval(r0, r1):
    movwt(r3, stm.CRC + stm.CRC_DR)
    str(r1, [r3, 0])
    ldr(r2, [r3, 0])
    str(r2, [r0, 0])

def getcrc(value):
    a = array('i', [0])
    getval(a, value)
    return a[0] & 0xffffffff # coerce to arbitrary precision

enable_crc()
reset_crc()
for x in range(20):
    print(hex(getcrc(0)))
```

5.6.4 References

- *Assembler Tutorial*
- Wiki hints and tips
- uPy Inline Assembler source-code, emitinlinethumb.c
- ARM Thumb2 Instruction Set Quick Reference Card
- RM0090 Reference Manual
- ARM v7-M Architecture Reference Manual (Available on the ARM site after a simple registration procedure. Also available on academic sites but beware of out of date versions.)

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 Wed 23 Aug 2017 02:09:21 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 Wed 23 Aug 2017 02:09:21 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 g`

Unpacking function arguments in non-last position isn't detected as an error

Workaround: The syntax below is invalid, never use it in applications.

Sample code:

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

CPy output:	uPy output:

`1 2 3`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`SyntaxError: non-keyword arg after */**`

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`

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 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 Wed 23 Aug 2017 02:09:21 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 attribut`

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__ raises TypeError if overridden and called by subclass

Sample code:

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

a = A()
```

CPy output:	uPy output:

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

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

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 i`

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`

6.3.4 float

uPy and CPython outputs formats may differ

Sample code:

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

CPy output:	uPy output:

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

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`

Incorrect error message when passing float into to_bytes

Sample code:

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

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

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

UnicodeDecodeError not raised when expected

Sample code:

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

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

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

Sample code:

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

CPy output:	uPy output:

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

Attributes/subscr not implemented

Sample code:

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

CPy output:	uPy output:

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

str(...) with keywords not implemented

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

Sample code:

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

CPy output:	uPy output:

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

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 'lju`

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`

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`

6.4 Modules

Generated Wed 23 Aug 2017 02:09:21 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 de`

Subscript with step != 1 is not yet implemented

Sample code:

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

CPy output:	uPy output:

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

6.4.2 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>`
	`AttributeError: 'module' object has no attribute '`

6.4.3 json

JSON module does not throw exception when object is not serialisable

Sample code:

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

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

6.4.4 struct

Struct pack with too few args, not checked by uPy

Sample code:

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

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

Struct pack with too many args, not checked by uPy

Sample code:

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

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

6.4.5 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 '`

CHAPTER

SEVEN

MICROPYTHON LICENSE INFORMATION

Symbols

A

B

C

D

E

F

G

H

I

J

P

Q

R

S

T

U

V

W

X

Y

Z