Micro Python Documentation

Release 1.3.6

Damien P. George

November 14, 2014

Contents

CHAPTER 1

Quick reference for the pyboard

1.1 General board control

See pyb.

```
import pyb

pyb.delay(50) # wait 50 milliseconds
pyb.millis() # number of milliseconds since bootup
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 LEDs

See *pyb.LED*.

```
from pyb import LED

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

1.3 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
```

1.4 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)
```

1.5 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())
```

1.6 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)
```

1.7 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
```

1.8 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
```

1.9 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
```

1.10 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 a receive 5 bytes
```

1.11 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 2

General information about the pyboard

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`.

(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.

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.

CHAPTER 3

Micro Python tutorial

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 Micro Python 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.

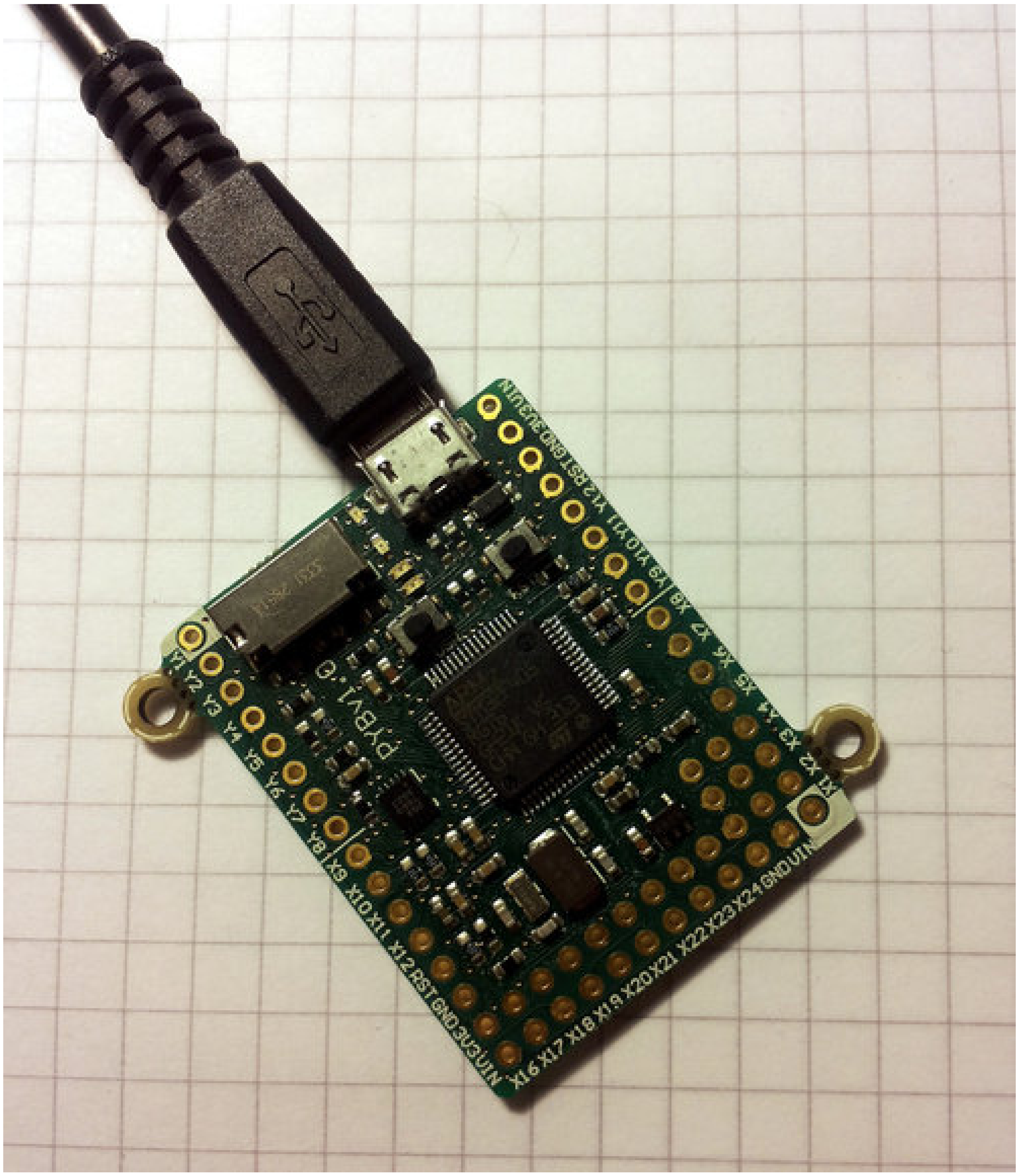
The input voltage must be between 3.6V and 10V.

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.


PYBv1.0
Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
X9 X10 X11 X12 RST GND 3V3 VIN
X16 X17 X18 X19 X20 X21 X22 X23 X24 GND VIN
X1 X2 X3 X4 X5 X6 X7 X8

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 Micro Python program!

3.3 Getting a Micro Python REPL prompt

REPL stands for Read Evaluate Print Loop, and is the name given to the interactive Micro Python 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).

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 Micro Python 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 Micro Python 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'
'pypypypypypypypypypypypypypypypypypypypy'
```

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 Micro Python 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 4.) 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, Micro Python 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 Micro Python 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()
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.

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.

3.6 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('CDC+MSC') # act as a serial and a storage device
#pyb.usb_mode('CDC+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('CDC+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('CDC+HID')
```

This tells the pyboard to configure itself as a CDC (serial) 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:

```
>>> pyb.hid((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):
...     pyb.hid((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.

Excercise: 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 editting `main.py`.)

In `main.py` put the following code:

```
import pyb

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

while not switch():
    pyb.hid((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 `pyb.hid()` 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 `CDC+HID` setting, so it looks like:

```
#pyb.usb_mode('CDC+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=255, period=32811, mode=0, div=0)
```

The information means that this timer is set to run at the peripheral clock speed divided by 255, and it will count up to 32811, 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.

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 freqency 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 Micro Python.

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

Micro Python 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. Micro Python 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 3 arguments. If they are used, they must be named `r0`, `r1` and `r2` 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)
```

3.11 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()
```

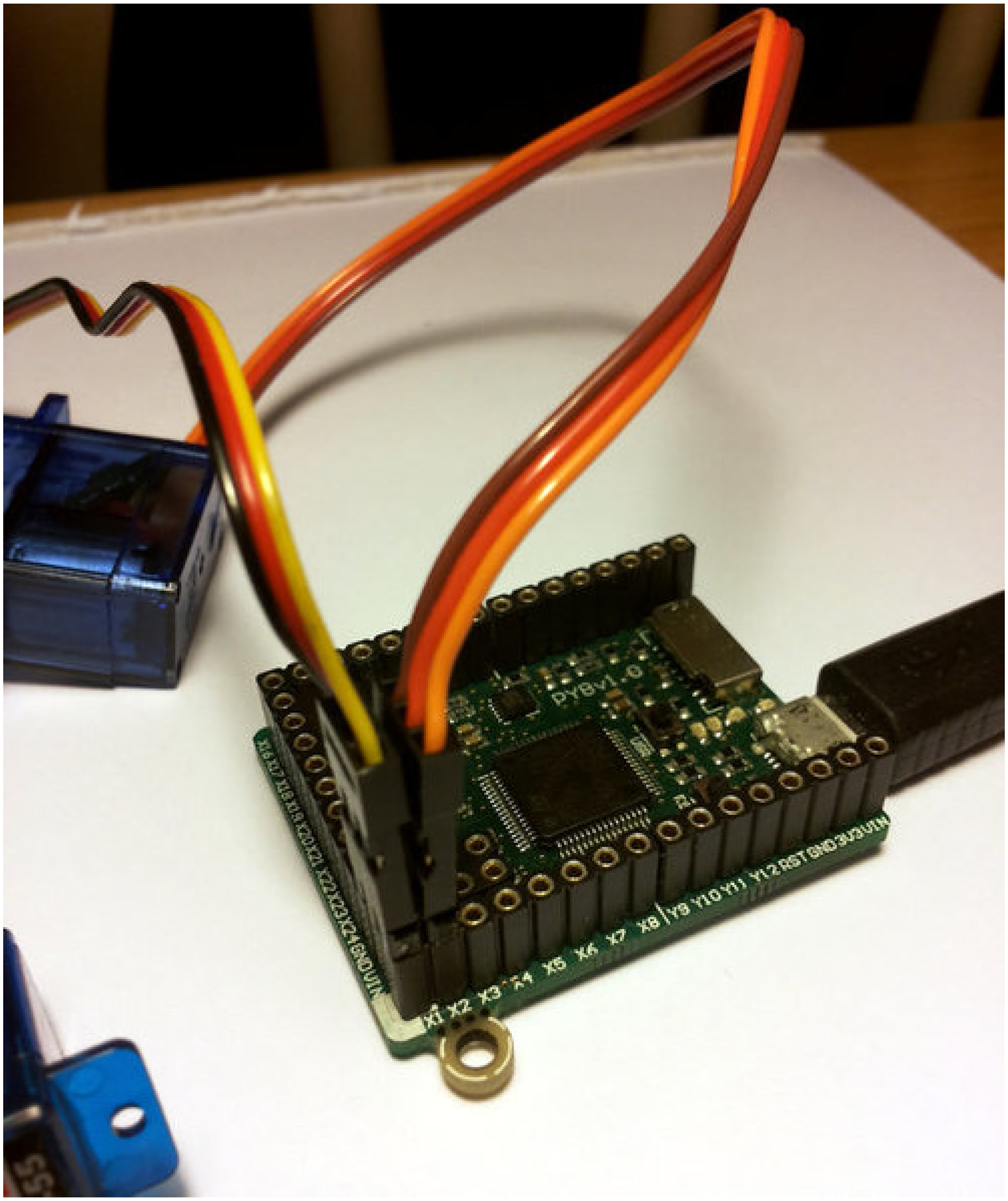
Control the frequency using `pyb.freq()`:

```
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

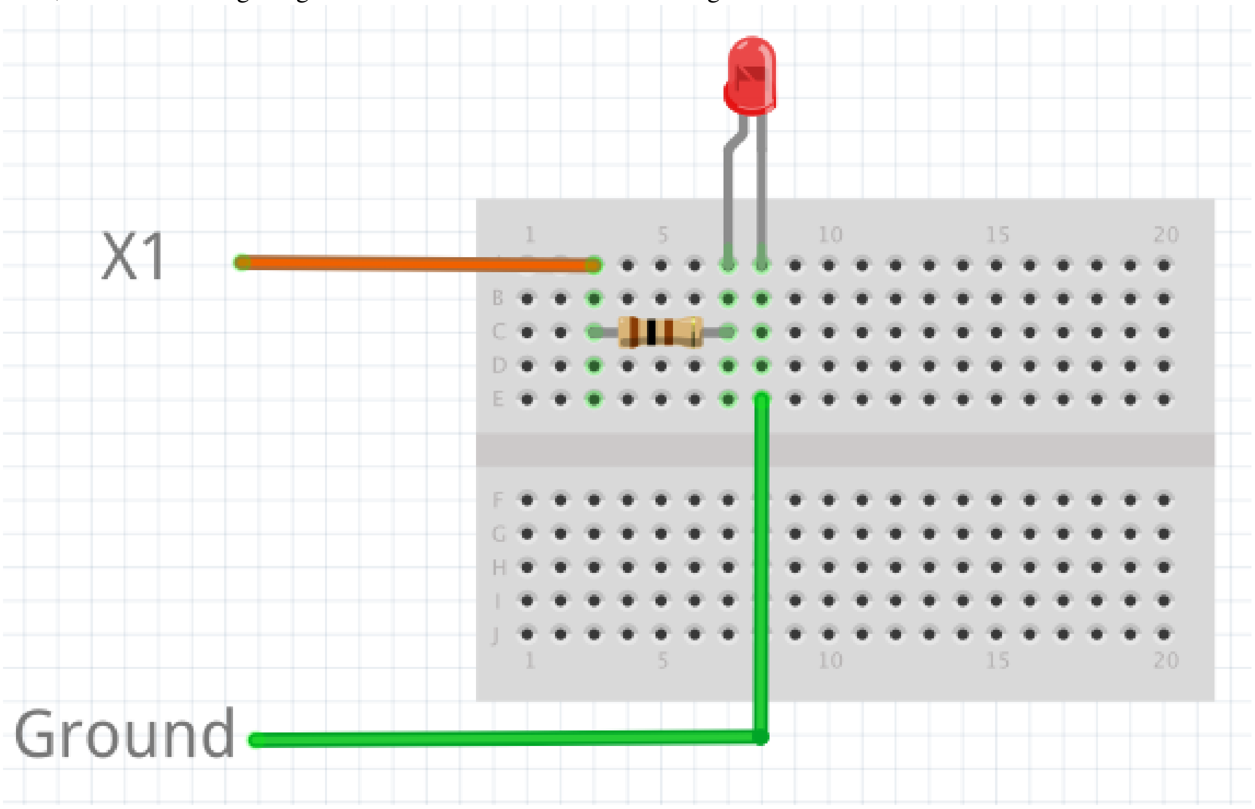
You will need:

- 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

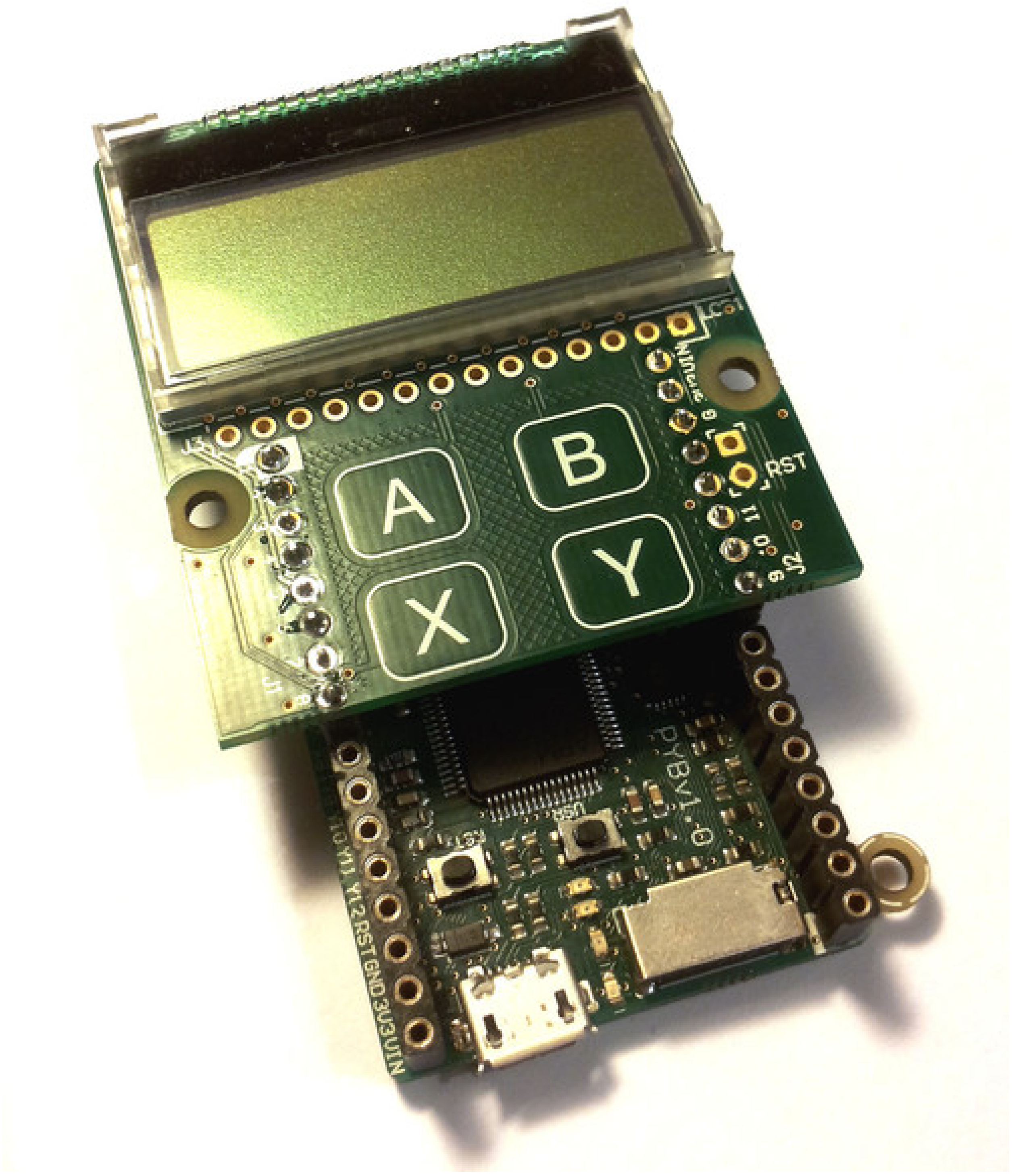
You may have noticed that the LED brightness seems to fade slowly, but increases quickly. This is because our eyes interprets brightness logarithmically (Weber's Law), while the LED's brightness changes linearly, that is by the same amount each time. How do you solve this problem? (Hint: what is the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

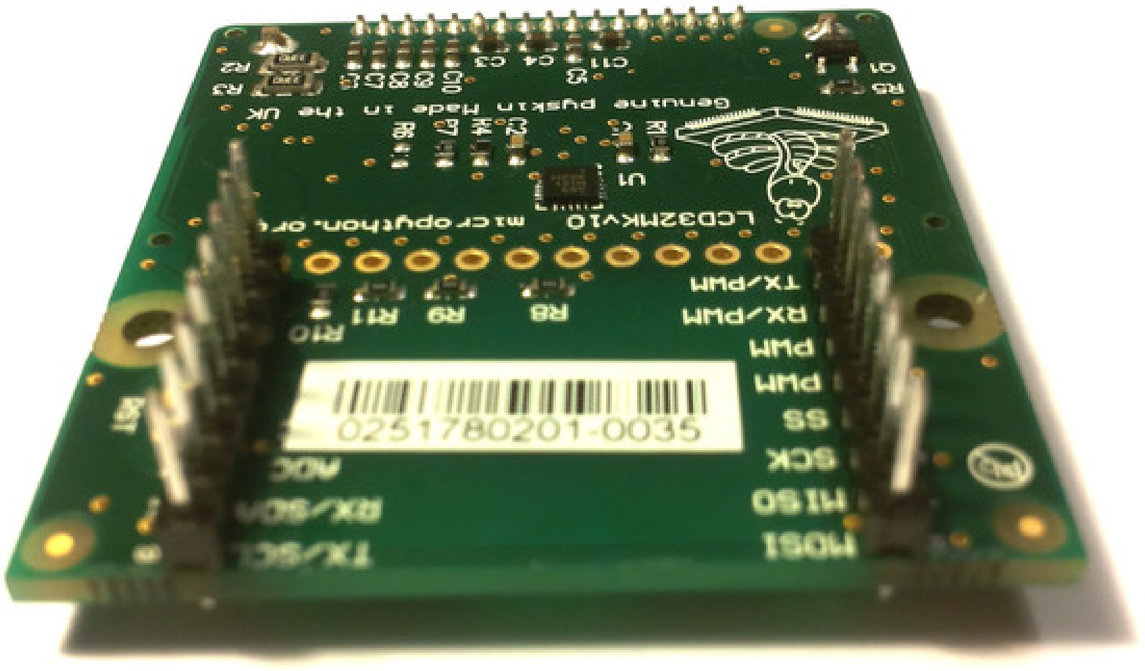
Addendum

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.

Using the LCD

To get started using the LCD, try the following at the Micro Python 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.

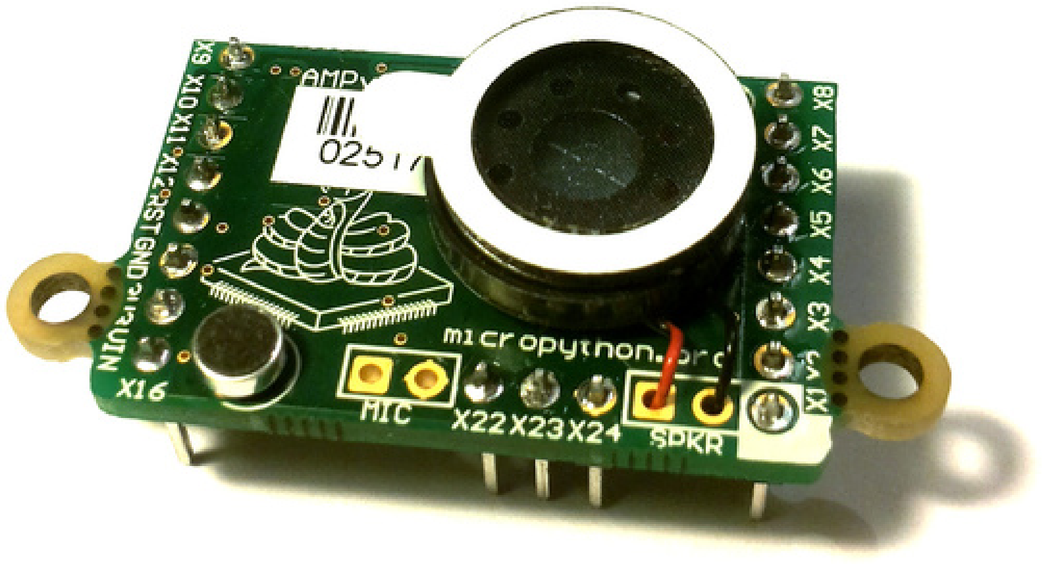
Note that if you put the LCD skin in the Y-position, then you need to initialise the I2C bus using:

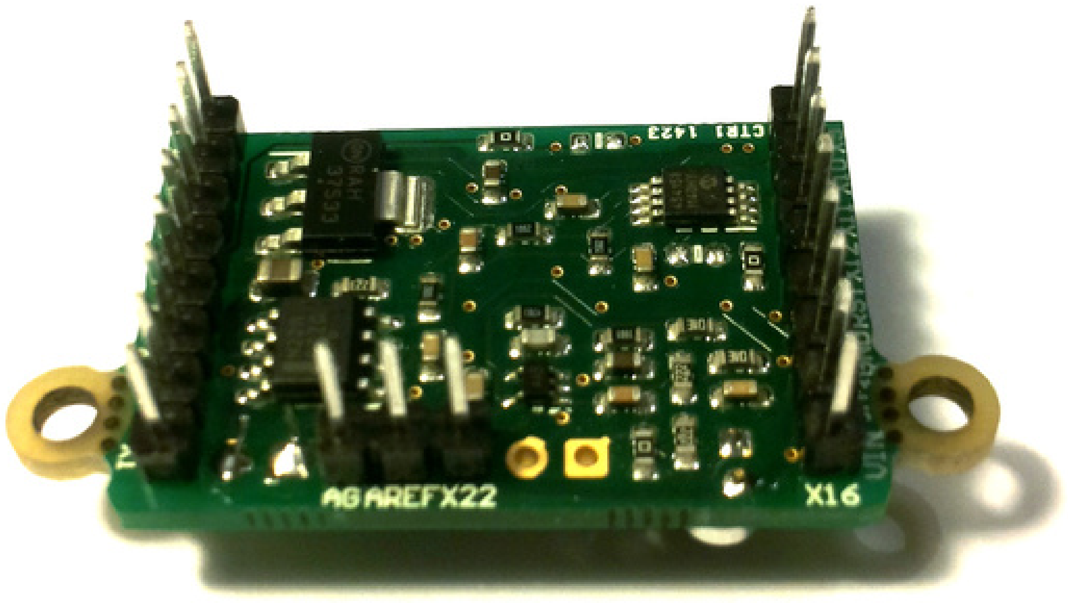
```
>>> m = mpr121.MPR121(pyb.I2C(2, pyb.I2C.MASTER))
```

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.

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. 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.

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):
    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 4

Micro Python libraries

4.1 Python standard libraries

The following standard Python libraries are built in to Micro Python.

For additional libraries, please download them from the micropython-lib repository.

4.1.1 `cmath` – mathematical functions for complex numbers

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

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

4.1.2 `gc` – control the garbage collector

Functions

`gc.`**`enable`**`()`
Enable automatic garbage collection.

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

`gc.`**`collect`**`()`
Run a garbage collection.

`gc.`**`mem_alloc`**`()`
Return the number of bytes of heap RAM that are allocated.

`gc.`**`mem_free`**`()`
Return the number of bytes of available heap RAM.

4.1.3 `math` – mathematical functions

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

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

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*`)`
: Converts a floating-point number to fractional and integral components.

`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

4.1.4 `os` – basic "operating system" services

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

Pyboard specifics

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

`/flash` – the internal flash filesystem

`/sd` – the SD card (if it exists)

On boot up, the current directory is `/flash` if no SD card is inserted, otherwise it is `/sd`.

Functions

`os.chdir(path)`
Change current directory.

`os.getcwd()`
Get the current directory.

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

`os.mkdir(path)`
Create a new directory.

`os.remove(path)`
Remove a file.

`os.rmdir(path)`
Remove a directory.

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

`os.sync()`
Sync all filesystems.

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

Constants

`os.sep`
separation character used in paths

4.1.5 `select` – Provides select function to wait for events on a stream

This module provides the select function.

Pyboard specifics

Polling is an efficient way of waiting for read/write activity on multiple objects. Current objects that support polling are: `pyb.UART`, `pyb.USB_VCP`.

Functions

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

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

class `Poll`

Methods

`poll.`**`register`**(*obj*[, *eventmask*])
Register `obj` for polling. `eventmask` is 1 for read, 2 for write, 3 for read-write.

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

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

`poll.`**`poll`**([*timeout*])
Wait for one of the registered objects to become ready.

Timeout is in milliseconds.

4.1.6 `struct` – pack and unpack primitive data types

See Python struct for more information.

Functions

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

`struct.`**`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.

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

4.1.7 `sys` – system specific functions

Functions

`sys.`**`exit`**([*retval*])
Raise a `SystemExit` exception. If an argument is given, it is the value given to `SystemExit`.

Constants

`sys.`**`argv`**
a mutable list of arguments this program started with

`sys.`**`byteorder`**
the byte order of the system (“little” or “big”)

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

`sys.`**`platform`**
the platform that Micro Python is running on

sys.**stderr**
standard error (connected to USB VCP, and optional UART object)

sys.**stdin**
standard input (connected to USB VCP, and optional UART object)

sys.**stdout**
standard output (connected to USB VCP, and optional UART object)

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

4.1.8 time – time related functions

The time module provides functions for getting the current time and date, and for sleeping.

Functions

time.**localtime**([*secs*])
Convert a time expressed in seconds since Jan 1, 2000 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

time.**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.

time.**sleep**(*seconds*)
Sleep for the given number of seconds. Seconds can be a floating-point number to sleep for a fractional number of seconds.

time.**time**()
Returns the number of seconds, as an integer, since 1/1/2000.

4.2 Python micro-libraries

The following standard Python libraries have been "micro-ified" to fit in with the philosophy of Micro Python. They provide the core functionality of that module and are intended to be a drop-in replacement for the standard Python library.

The modules are available 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.

4.2.1 `usocket` – socket module

Socket functionality.

Functions

`usocket.`**`getaddrinfo`**(*host*, *port*)

`usocket.`**`socket`**(*family=AF_INET*, *type=SOCK_STREAM*, *fileno=-1*)
Create a socket.

4.2.2 `uheapq` – heap queue algorithm

This module implements the heap queue algorithm.

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

Functions

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

`uheapq.`**`heappop`**(*heap*)
Pop the first item froh 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.

4.2.3 `ujson` – JSON encoding and decoding

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

Functions

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

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

4.3 Libraries specific to the pyboard

The following libraries are specific to the pyboard.

4.3.1 `pyb` — functions related to the pyboard

The `pyb` module contains specific functions related to the pyboard.

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.

`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.

`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 upto 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 upto 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.

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`**`(`[*sys_freq*]`)`

If given no arguments, returns a tuple of clock frequencies: (SYSCLK, HCLK, PCLK1, PCLK2).

If given an argument, sets the system frequency to that value in Hz. Eg freq(120000000) gives 120MHz. Note that not all values are supported and the largest supported frequency not greater than the given sys_freq will be selected.

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

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 frequencies below 36MHz do not allow the USB to function correctly.

`pyb.`**`wfi`**`()`

Wait for an interrupt. This executies a `wfi` instruction which reduces power consumption of the MCU until an interrupt occurs, at which point execution continues.

`pyb.`**`standby`**`()`

`pyb.`**`stop`**`()`

Miscellaneous functions

`pyb.`**`have_cdc`**`()`

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

`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.

`pyb.`**`info`**`(`[*dump_alloc_table*]`)`

Print out lots of information about the board.

`pyb.`**`repl_uart`**`(`*uart*`)`

Get or set the UART object that 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 for the MCU.

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.

Note: if you read accelerometer values immediately after creating this object you will get 0. It takes around 20ms for the first sample to be ready, so, unless you have some other code between creating this object and reading its values, you should put a `pyb.delay(20)` after creating it. For example:

```
accel = pyb.Accel()
pyb.delay(20)
print(accel.x())
```

Methods

`accel.`**`filtered_xyz`**`()`

Get a 3-tuple of filtered x, y and z values.

`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.

class ADC – analog to digital conversion: read analog values on a pin

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)        # creale 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, freq*)

Read analog values into the given buffer at the given frequency. Buffer can be bytearray or array.array for example. If a buffer with 8-bit elements is used, sample resolution will be reduced to 8 bits.

Example:

```
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.

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.

Note that this driver does not yet support filter configuration (it defaults to a single filter that lets through all messages), or bus timing configuration (except for setting the prescaler).

Example usage (works without anything connected):

```
from pyb import CAN
can = pyb.CAN(1, pyb.CAN.LOOPBACK)
can.send('message!', 123)   # send message to 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)`

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.

`can.deinit`()
Turn off the CAN bus.

`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: buffer of data bytes.

`can.send`(*send*, *addr*, ***, *timeout=5000*)
Send a message on the bus:

- `send` is the data to send (an integer to send, or a buffer object).
- `addr` is the address to send to
- `timeout` is the timeout in milliseconds to wait for the send.

Return value: `None`.

Constants

`CAN.NORMAL`
`CAN.LOOPBACK`

`CAN.SILENT`

`CAN.SILENT_LOOPBACK`
the mode of the CAN bus

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)
```

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)
```

Constructors

class `pyb.`**`DAC`**`(`*port*`)`

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.

Methods

`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 frequence of the repeating triangle wave itself is 256 (or 1024, need to check) times smaller.

`dac.`**`write`**`(`*value*`)`

Direct access to the DAC output (8 bit only at the moment).

`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 (8 bit data).

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

TIM6 is used to control the frequency of the transfer.

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 thru 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 thru 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

`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:

```
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.

Basic methods for slave 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)   # timout 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)
```

Constructors

class `pyb.`**`I2C`**(*bus*, ...)

Construct an I2C object on the given bus. `bus` can be 1 or 2. 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 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)`

Methods

`i2c.`**`deinit`**`()`
Turn off the I2C bus.

`i2c.`**`init`**`(`*mode*, ***, *addr=0x12*, *baudrate=400000*, *gencall=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

`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 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.`**`scan`**`()`
Scan all I2C addresses from 0x01 to 0x7f and return a list of those that respond. Only valid when in master mode.

`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`.

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`.

`led.`**`off`**`()`
Turn the LED off.

`led.`**`on`**`()`
Turn the LED on.

`led.`**`toggle`**`()`
Toggle the LED between on and 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.B6` 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).

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

`Pin.`**`af_list`**()

Returns an array of alternate functions available for this pin.

`Pin.`**`debug`**([*state*])

Get or set the debugging state (`True` or `False` for on or off).

`Pin.`**`dict`**([*dict*])

Get or set the pin mapper dictionary.

`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.`**`high`**()

Set the pin to a high logic level.

`pin.`**`low`**()

Set the pin to a low logic level.

`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.`**`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 microcprocessor. 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 availble 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.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

class Servo – 3-wire hobby servo driver

Servo controls standard hobby servos with 3-wires (ground, power, signal).

Constructors

class `pyb.`**`Servo`**`(`*id*`)`

Create a servo object. `id` is 1-4.

Methods

`servo.`**`angle`**`(`[*angle*, *time=0*]`)`

Get or set 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.

`servo.`**`calibration`**`(`[*pulse_min*, *pulse_max*, *pulse_centre*[, *pulse_angle_90*, *pulse_speed_100*]]`)`

Get or set the calibration of the servo timing.

`servo.`**`pulse_width`**`(`[*value*]`)`

Get or set the pulse width in milliseconds.

`servo.`**`speed`**`(`[*speed*, *time=0*]`)`

Get or set the speed of a continuous rotation servo.

- `speed` is the speed to move to change to, between -100 and 100.
- `time` is the number of milliseconds to take to get to the specified speed.

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 method 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. 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*, ***, *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).

`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()                    # get state (True if pressed, False otherwise)
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``()`

Return the switch state: `True` if pressed down, `False` otherwise.

`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())
```

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 3 is reserved for internal use. 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.

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, excluding 3.

Methods

`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).

Othwerwise, 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.
- `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 acive 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.

PWM Example:

```
timer = pyb.Timer(2, freq=1000)
ch2 = timer.channel(2, pyb.Timer.PWM, pin=pyb.Pin.board.X2, pulse_width=210000)
ch3 = timer.channel(3, pyb.Timer.PWM, pin=pyb.Pin.board.X3, pulse_width=420000)
```

`timer.`**`counter`**`(`[*value*]`)`
Get or set the timer counter.

`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.`**`freq`**`(`[*value*]`)`
Get or set the frequency for the timer (changes prescaler and period if set).

`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 migh 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` - confgures 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. `deadime` 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.`**`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.readall()      # 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 True if any characters waiting
```

Note: The stream functions `read`, `write` etc Are new in Micro Python since 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)`

Methods

`uart.`**`init`**`(`*baudrate*, *bits=8*, *parity=None*, *stop=1*, ***, *timeout=1000*, *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.
- `timeout` is the timeout in milliseconds to wait for the first character.
- `timeout_char` is the timeout in milliseconds to wait between characters.
- `read_buf_len` is the character length of the read buffer (0 to disable).

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`**`()`

Return `True` if any characters waiting, else `False`.

`uart.`**`read`**`(`[*nbytes*]`)`
Read characters. If `nbytes` is specified then read at most that many bytes.

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 `b''` on timeout.

`uart.`**`readall`**`()`
Read as much data as possible.

Return value: a bytes object.

`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`.

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

Return value: the line read.

`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.

`uart.`**`writechar`**`(`*char*`)`
Write a single character on the bus. `char` is an integer to write. Return value: `None`.

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.`**`any`**`()`
Return `True` if any characters waiting, else `False`.

`usb_vcp.`**`close`**`()`

`usb_vcp.`**`read`**`(`[*nbytes*]`)`

`usb_vcp.`**`readall`**`()`

`usb_vcp.`**`readline`**`()`

`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.

`usb_vcp.`**`write`**`(`*buf*`)`

4.3.2 `network` — network configuration

This module provides network drivers and routing configuration.

class CC3k

Constructors

class `network.`**`CC3k`**`(`*spi*, *pin_cs*, *pin_en*, *pin_irq*`)`

Initialise the CC3000 using the given SPI bus and pins and return a CC3k object.

Methods

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

class WIZnet5k

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

Example usage:

```
import wiznet5k
w = wiznet5k.WIZnet5k()
print(w.ipaddr())
w.gethostbyname('micropython.org')
s = w.socket()
s.connect(('192.168.0.2', 8080))
s.send('hello')
print(s.recv(10))
```

Constructors

class `network.`**`WIZnet5k`**(*spi*, *pin_cs*, *pin_rst*)
Create and return a WIZnet5k object.

Methods

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

`wiznet5k.`**`regs`**()
Dump WIZnet5k registers.

CHAPTER 5

The pyboard hardware

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

CHAPTER 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)

CHAPTER 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 8

Micro Python license information

CHAPTER 9

Micro Python documentation contents

9.1 Quick reference for the pyboard

9.1.1 General board control

See pyb.

```
import pyb

pyb.delay(50) # wait 50 milliseconds
pyb.millis() # number of milliseconds since bootup
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
```

9.1.2 LEDs

See *pyb.LED*.

```
from pyb import LED

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

9.1.3 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
```

9.1.4 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)
```

9.1.5 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())
```

9.1.6 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)
```

9.1.7 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
```

9.1.8 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
```

9.1.9 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
```

9.1.10 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 a receive 5 bytes
```

9.1.11 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
```

9.2 General information about the pyboard

9.2.1 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`.

(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.

9.2.2 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.

9.2.3 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.

9.3 Micro Python tutorial

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.

9.3.1 Introduction to the pyboard

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

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 Micro Python software, but that can be done over USB.

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.

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.

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.

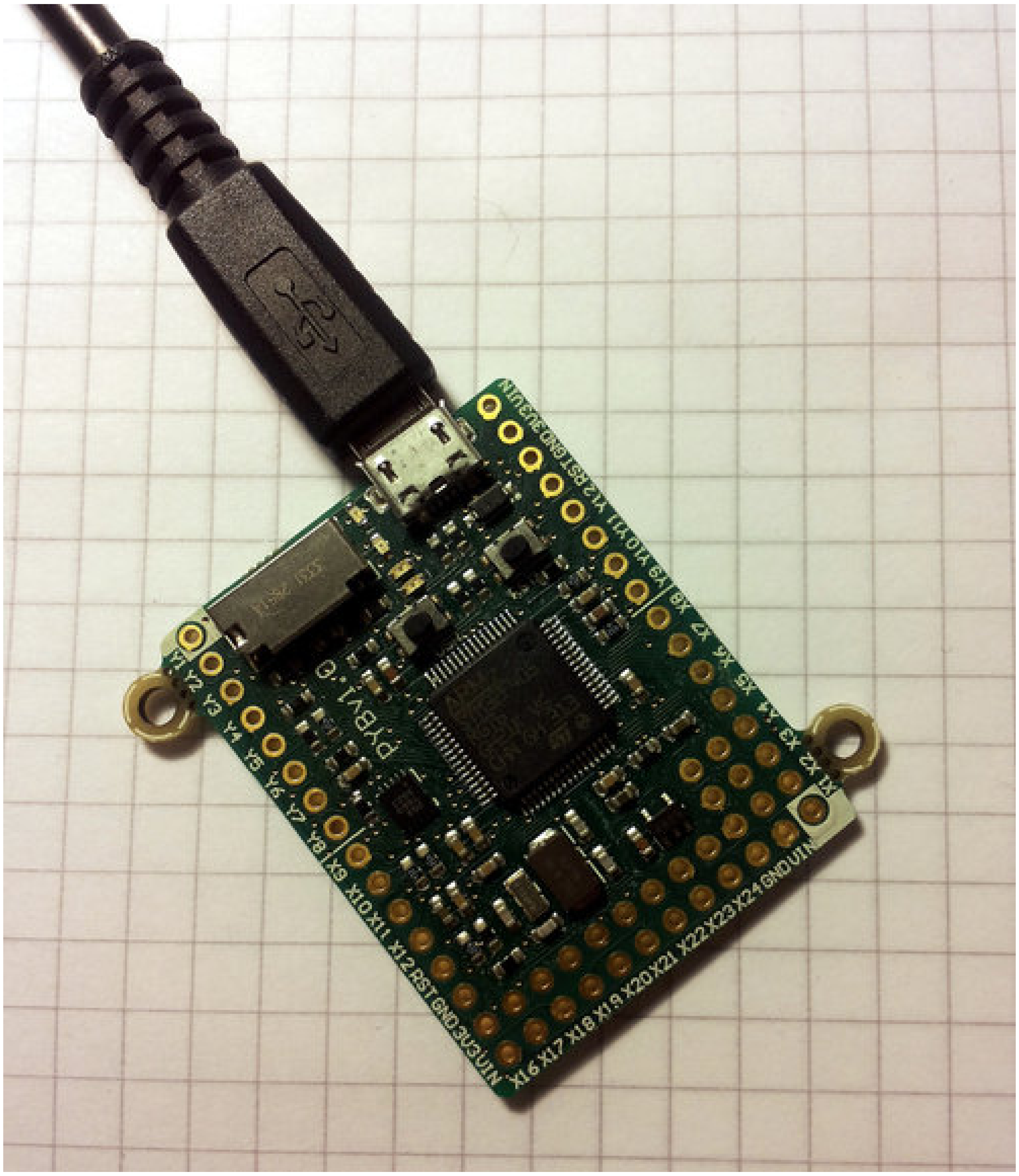
The input voltage must be between 3.6V and 10V.

9.3.2 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!

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.


PYBv1.0
Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
X9 X10 X11 X12 RST GND 3V3 VIN
X16 X17 X18 X19 X20 X21 X22 X23 X24 GND VIN
X1 X2 X3 X4 X5 X6 X7 X8
Y9 Y10 Y11 Y12 RST GND 3V3 VIN

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.

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.

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.

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 Micro Python program!

9.3.3 Getting a Micro Python REPL prompt

REPL stands for Read Evaluate Print Loop, and is the name given to the interactive Micro Python 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.

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).

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.

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-\.

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).

Using the REPL prompt

Now let's try running some Micro Python 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 Micro Python 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'
'pypypypypypypypypypypypypypypypypypypypy'
```

Resetting the board

If something goes wrong, you can reset the board in two ways. The first is to press CTRL-D at the Micro Python 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.

9.3.4 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.

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 4.) 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, Micro Python 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 Micro Python 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()
```

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.

9.3.5 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()
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.

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.

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.

9.3.6 The accelerometer

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

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.

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.

9.3.7 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.

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.

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.

9.3.8 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('CDC+MSC') # act as a serial and a storage device
#pyb.usb_mode('CDC+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('CDC+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('CDC+HID')
```

This tells the pyboard to configure itself as a CDC (serial) 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!

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:

```
>>> pyb.hid((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):
...     pyb.hid((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.

Excercise: make the mouse go around in a circle.

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 editting `main.py`.)

In `main.py` put the following code:

```
import pyb

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

while not switch():
    pyb.hid((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 `pyb.hid()` line above.

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 `CDC+HID` setting, so it looks like:

```
#pyb.usb_mode('CDC+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.

9.3.9 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=255, period=32811, mode=0, div=0)
```

The information means that this timer is set to run at the peripheral clock speed divided by 255, and it will count up to 32811, 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.

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.

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.

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 freqency 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()
```

9.3.10 Inline assembler

Here you will learn how to write inline assembler in Micro Python.

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

Micro Python 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.

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. Micro Python 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.

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.

Accepting arguments

Inline assembler functions can accept up to 3 arguments. If they are used, they must be named `r0`, `r1` and `r2` 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.

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)
```

9.3.11 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()
```

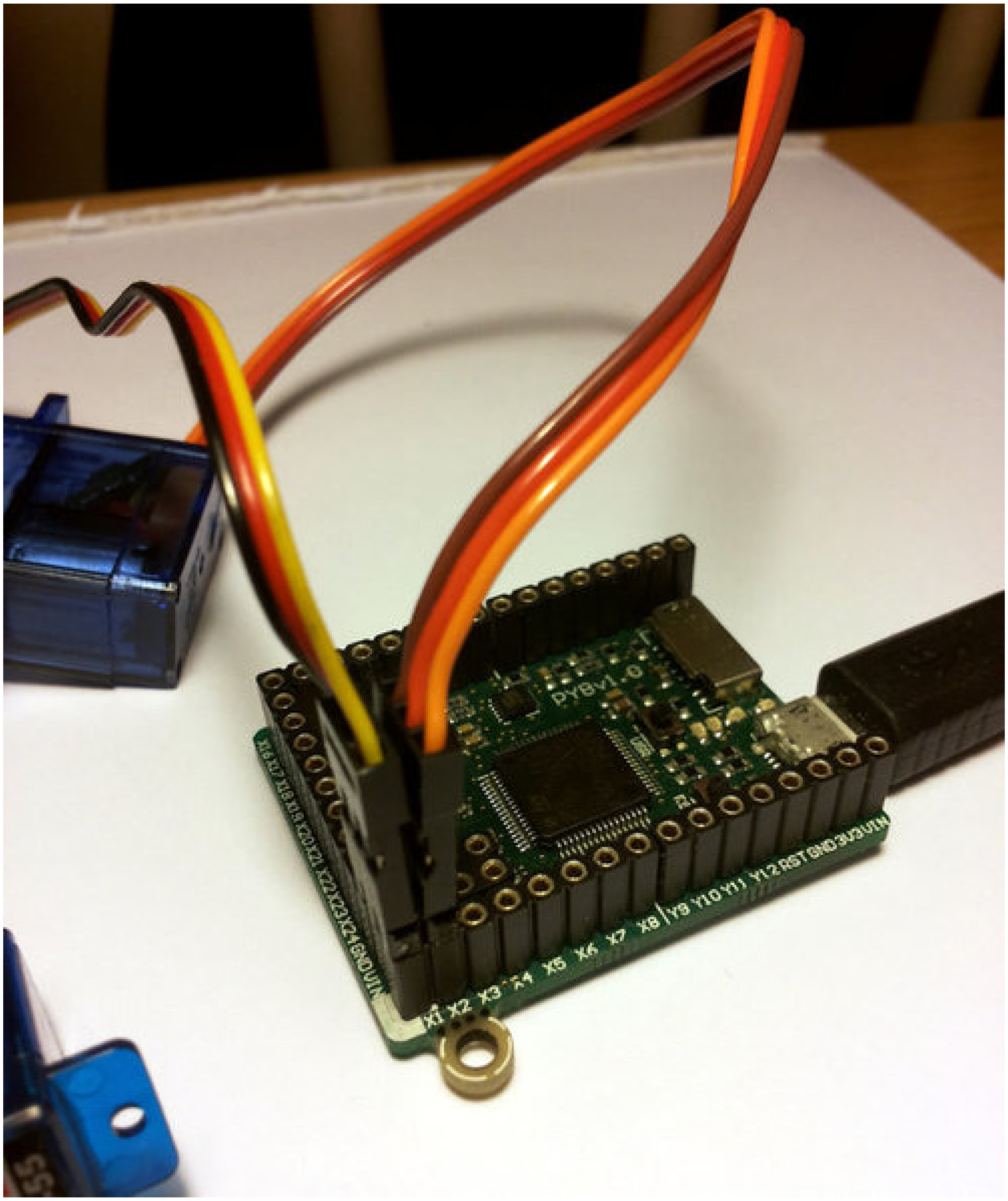
Control the frequency using `pyb.freq()`:

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

9.3.12 Tutorials requiring extra components

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.

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

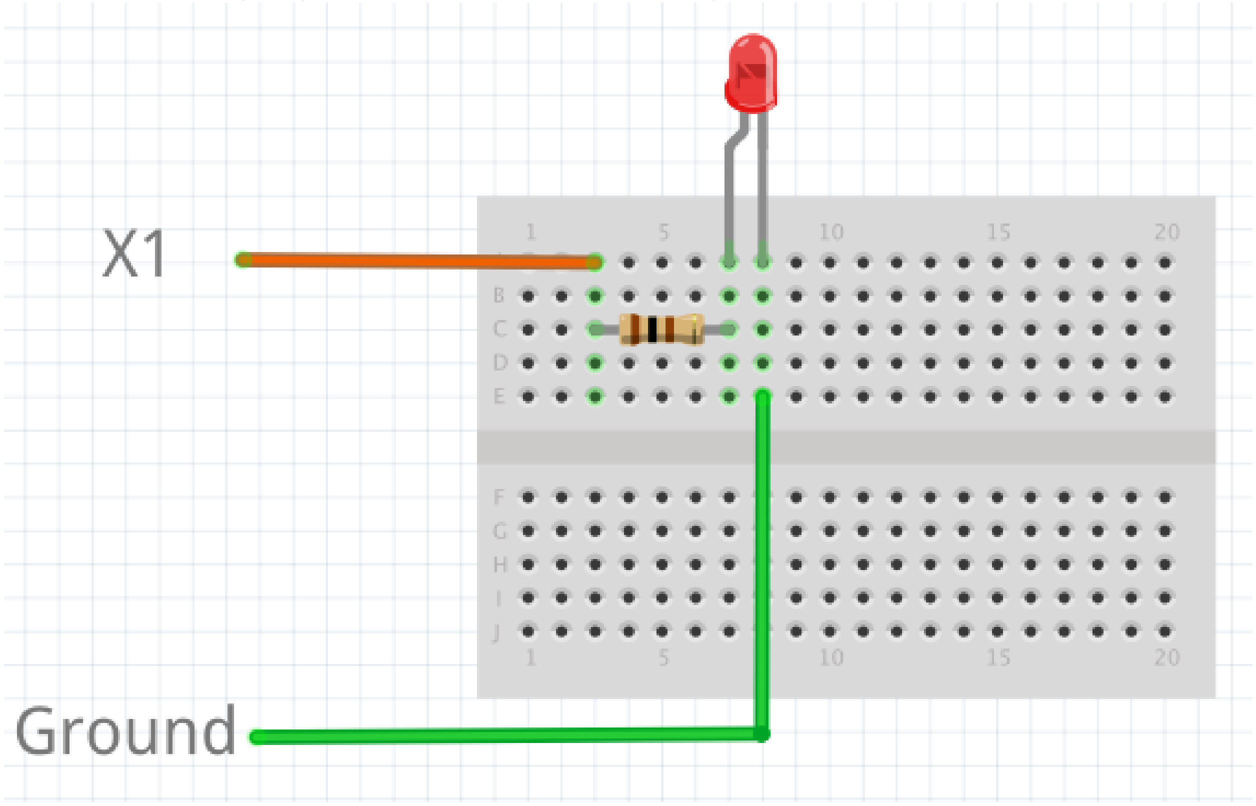
You will need:

- 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

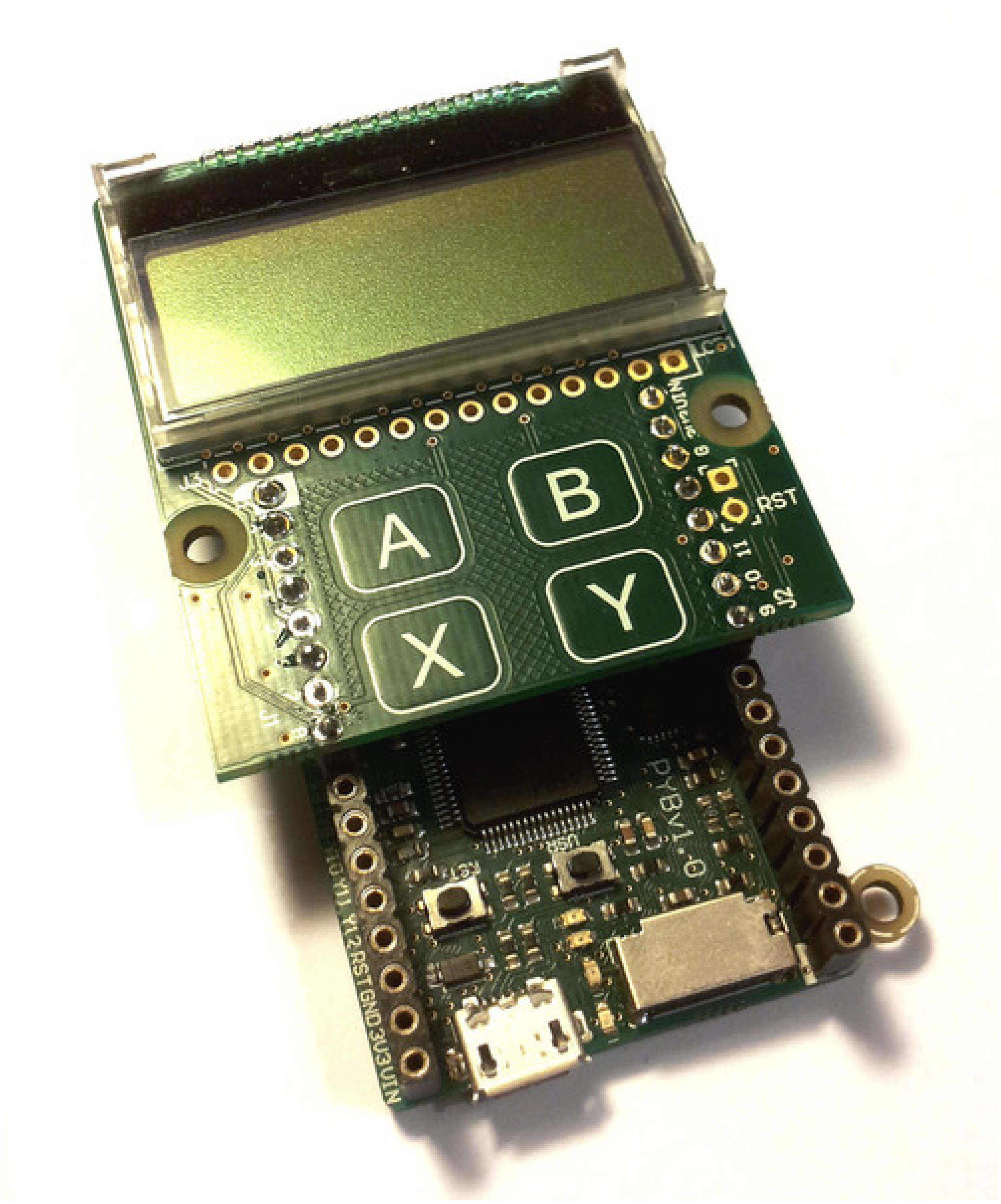
You may have noticed that the LED brightness seems to fade slowly, but increases quickly. This is because our eyes interprets brightness logarithmically (Weber's Law), while the LED's brightness changes linearly, that is by the same amount each time. How do you solve this problem? (Hint: what is the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

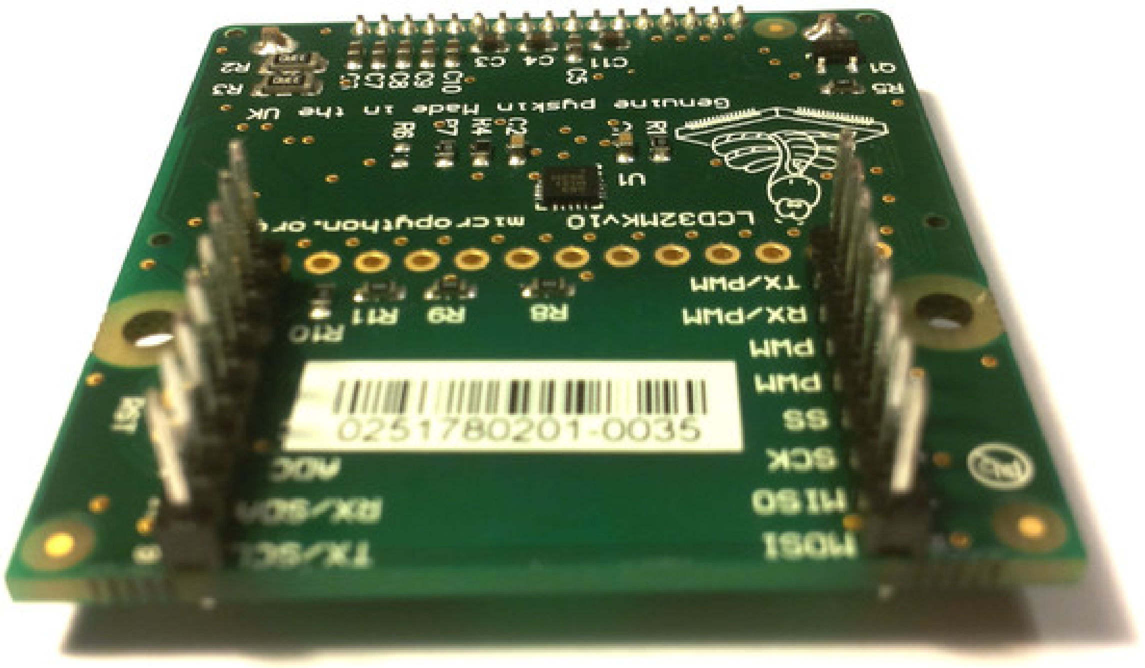
Addendum

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.

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.

Using the LCD

To get started using the LCD, try the following at the Micro Python 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.

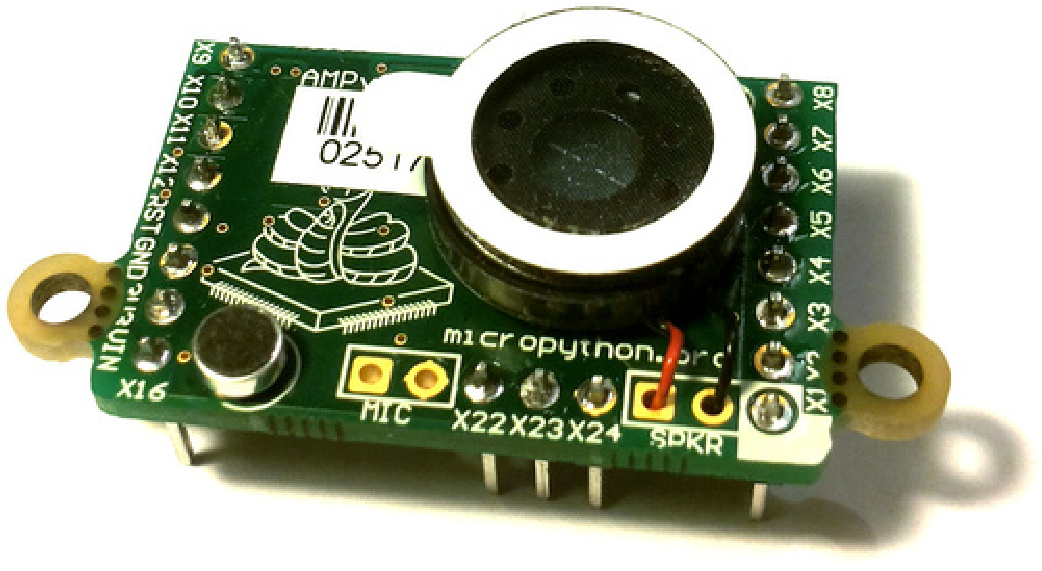
Note that if you put the LCD skin in the Y-position, then you need to initialise the I2C bus using:

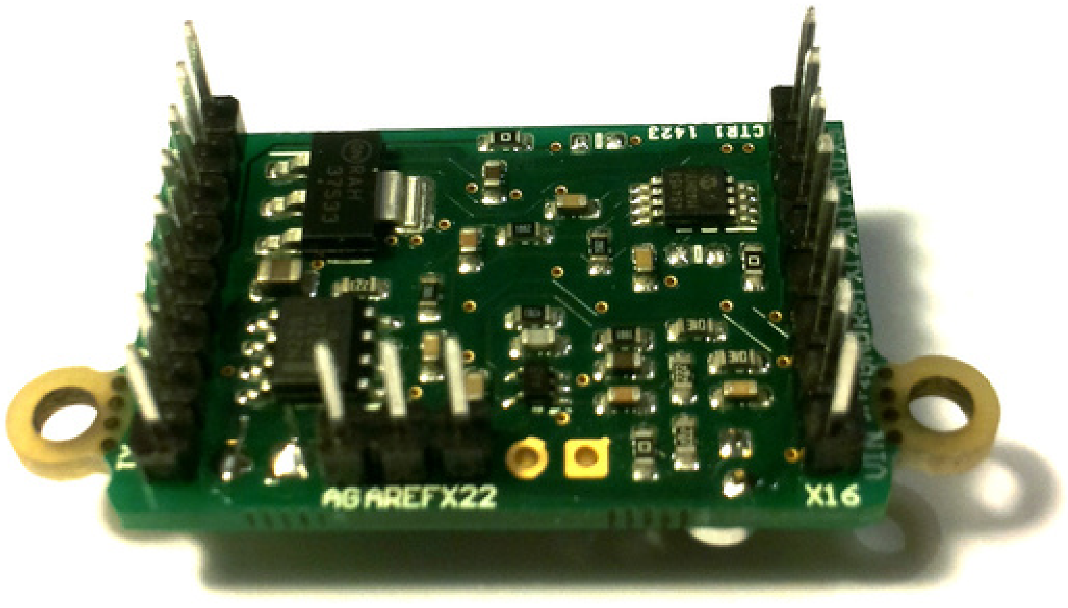
```
>>> m = mpr121.MPR121(pyb.I2C(2, pyb.I2C.MASTER))
```

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

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.

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. 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.

9.3.13 Tips, tricks and useful things to know

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()
```

Making a UART - USB pass through

It's as simple as:

```
import pyb
import select

def pass_through(usb, uart):
    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))
```

9.4 Micro Python libraries

9.4.1 Python standard libraries

The following standard Python libraries are built in to Micro Python.

For additional libraries, please download them from the micropython-lib repository.

`cmath` – mathematical functions for complex numbers

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

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

`gc` – control the garbage collector

Functions

`gc.`**`enable`**`()`
Enable automatic garbage collection.

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

`gc.`**`collect`**`()`
Run a garbage collection.

`gc.`**`mem_alloc`**`()`
Return the number of bytes of heap RAM that are allocated.

`gc.`**`mem_free`**`()`
Return the number of bytes of available heap RAM.

`math` – mathematical functions

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

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

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*)
Converts a floating-point number to fractional and integral components.

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

`os` – basic "operating system" services

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

Pyboard specifics

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

> `/flash` – the internal flash filesystem
>
> `/sd` – the SD card (if it exists)

On boot up, the current directory is `/flash` if no SD card is inserted, otherwise it is `/sd`.

Functions

`os.`**`chdir`**`(`*path*`)`
: Change current directory.

`os.`**`getcwd`**`()`
: Get the current directory.

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

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

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

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

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

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

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

Constants

`os.`**`sep`**
separation character used in paths

`select` – Provides select function to wait for events on a stream

This module provides the select function.

Pyboard specifics

Polling is an efficient way of waiting for read/write activity on multiple objects. Current objects that support polling are: `pyb.UART`, `pyb.USB_VCP`.

Functions

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

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

class `Poll`

Methods

`poll.`**`register`**(*obj*[, *eventmask*])
Register `obj` for polling. `eventmask` is 1 for read, 2 for write, 3 for read-write.

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

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

`poll.poll([timeout])`
: Wait for one of the registered objects to become ready.

  Timeout is in milliseconds.

`struct` – pack and unpack primitive data types

See Python struct for more information.

Functions

`struct.calcsize(fmt)`
: Return the number of bytes needed to store the given `fmt`.

`struct.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.

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

`sys` – system specific functions

Functions

`sys.exit([retval])`
: Raise a `SystemExit` exception. If an argument is given, it is the value given to `SystemExit`.

Constants

`sys.argv`
: a mutable list of arguments this program started with

`sys.byteorder`
: the byte order of the system (“little” or “big”)

`sys.path`
: a mutable list of directories to search for imported modules

`sys.platform`
: the platform that Micro Python is running on

`sys.stderr`
: standard error (connected to USB VCP, and optional UART object)

`sys.stdin`
: standard input (connected to USB VCP, and optional UART object)

`sys.stdout`
: standard output (connected to USB VCP, and optional UART object)

`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

`time` – time related functions

The `time` module provides functions for getting the current time and date, and for sleeping.

Functions

`time.`**`localtime`**`([`*secs*`])`

Convert a time expressed in seconds since Jan 1, 2000 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

`time.`**`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.

`time.`**`sleep`**`(`*seconds*`)`

Sleep for the given number of seconds. Seconds can be a floating-point number to sleep for a fractional number of seconds.

`time.`**`time`**`()`

Returns the number of seconds, as an integer, since 1/1/2000.

9.4.2 Python micro-libraries

The following standard Python libraries have been "micro-ified" to fit in with the philosophy of Micro Python. They provide the core functionality of that module and are intended to be a drop-in replacement for the standard Python library.

The modules are available 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.

`usocket` – socket module

Socket functionality.

Functions

`usocket.`**`getaddrinfo`**`(`*host*, *port*`)`

`usocket.`**`socket`**`(`*family=AF_INET*, *type=SOCK_STREAM*, *fileno=-1*`)`

Create a socket.

`uheapq` – heap queue algorithm

This module implements the heap queue algorithm.

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

Functions

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

`uheapq.`**`heappop`**`(`*heap*`)`
: Pop the first item froh 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.

`ujson` – JSON encoding and decoding

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

Functions

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

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

9.4.3 Libraries specific to the pyboard

The following libraries are specific to the pyboard.

`pyb` — functions related to the pyboard

The `pyb` module contains specific functions related to the pyboard.

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.

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.

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 upto 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 upto 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.

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**([*sys_freq*])

If given no arguments, returns a tuple of clock frequencies: (SYSCLK, HCLK, PCLK1, PCLK2).

If given an argument, sets the system frequency to that value in Hz. Eg freq(120000000) gives 120MHz. Note that not all values are supported and the largest supported frequency not greater than the given sys_freq will be selected.

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

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 frequencies below 36MHz do not allow the USB to function correctly.

`pyb.`**`wfi`**`()`
Wait for an interrupt. This execuites a `wfi` instruction which reduces power consumption of the MCU until an interrupt occurs, at which point execution continues.

`pyb.`**`standby`**`()`

`pyb.`**`stop`**`()`

Miscellaneous functions

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

`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.

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

`pyb.`**`repl_uart`**`(`*uart*`)`
Get or set the UART object that 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 for the MCU.

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.

Note: if you read accelerometer values immediately after creating this object you will get 0. It takes around 20ms for the first sample to be ready, so, unless you have some other code between creating this object and reading its values, you should put a `pyb.delay(20)` after creating it. For example:

```
accel = pyb.Accel()
pyb.delay(20)
print(accel.x())
```

Methods

`accel.filtered_xyz()`

Get a 3-tuple of filtered x, y and z values.

`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.

class ADC – analog to digital conversion: read analog values on a pin

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)        # creale 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, freq*`)`

Read analog values into the given buffer at the given frequency. Buffer can be bytearray or array.array for example. If a buffer with 8-bit elements is used, sample resolution will be reduced to 8 bits.

Example:

```
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.

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.

Note that this driver does not yet support filter configuration (it defaults to a single filter that lets through all messages), or bus timing configuration (except for setting the prescaler).

Example usage (works without anything connected):

```
from pyb import CAN
can = pyb.CAN(1, pyb.CAN.LOOPBACK)
can.send('message!', 123)   # send message to 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)`

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.

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

Turn off the CAN bus.

`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: buffer of data bytes.

`can.`**`send`**`(`*send, addr, *, timeout=5000*`)`

Send a message on the bus:

- `send` is the data to send (an integer to send, or a buffer object).

- `addr` is the address to send to
- `timeout` is the timeout in milliseconds to wait for the send.

Return value: `None`.

Constants

`CAN.`**`NORMAL`**

`CAN.`**`LOOPBACK`**

`CAN.`**`SILENT`**

`CAN.`**`SILENT_LOOPBACK`**

the mode of the CAN bus

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)
```

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)
```

Constructors

class `pyb.`**`DAC`**(*port*)

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.

Methods

`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 frequence of the repeating triangle wave itself is 256 (or 1024, need to check) times smaller.

`dac.`**`write`**(*value*)

Direct access to the DAC output (8 bit only at the moment).

`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 (8 bit data).

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

TIM6 is used to control the frequency of the transfer.

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 thru 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 thru 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

`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:

```
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.

Basic methods for slave 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)   # timout 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)
```

Constructors

class `pyb.`**`I2C`**(*bus*, ...)

Construct an I2C object on the given bus. `bus` can be 1 or 2. 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 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)`

Methods

`i2c.`**`deinit`**()

Turn off the I2C bus.

`i2c.`**`init`**(*mode*, ***, *addr=0x12*, *baudrate=400000*, *gencall=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

`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 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.`**`scan`**`()`
Scan all I2C addresses from 0x01 to 0x7f and return a list of those that respond. Only valid when in master mode.

`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`.

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`.

`led.`**`off`**`()`

Turn the LED off.

`led.`**`on`**`()`

Turn the LED on.

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

Toggle the LED between on and 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.B6` 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).

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

`Pin.`**`af_list`**`()`

Returns an array of alternate functions available for this pin.

`Pin.`**`debug`**`(`[*state*]`)`

Get or set the debugging state (`True` or `False` for on or off).

`Pin.`**`dict`**`(`[*dict*]`)`

Get or set the pin mapper dictionary.

`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.`**`high`**`()`

Set the pin to a high logic level.

`pin.`**`low`**`()`

Set the pin to a low logic level.

`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.`**`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 microcprocessor. 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 availble 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.`**`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

class Servo – 3-wire hobby servo driver Servo controls standard hobby servos with 3-wires (ground, power, signal).

Constructors

class `pyb.`**`Servo`**`(`*id*`)`

Create a servo object. `id` is 1-4.

Methods

`servo.`**`angle`**`(`[*angle*, *time=0*]`)`

Get or set 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.

`servo.`**`calibration`**`(`[*pulse_min*, *pulse_max*, *pulse_centre*[, *pulse_angle_90*, *pulse_speed_100*]]`)`

Get or set the calibration of the servo timing.

`servo.`**`pulse_width`**`(`[*value*]`)`

Get or set the pulse width in milliseconds.

`servo.`**`speed`**`(`[*speed*, *time=0*]`)`

Get or set the speed of a continuous rotation servo.

- `speed` is the speed to move to change to, between -100 and 100.
- `time` is the number of milliseconds to take to get to the specified speed.

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 method 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. 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*, ***, *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).

`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()                    # get state (True if pressed, False otherwise)
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``()`

Return the switch state: `True` if pressed down, `False` otherwise.

`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())
```

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 3 is reserved for internal use. 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.

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, excluding 3.

Methods

`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).

Othwerwise, 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.
- `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 acive 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.

PWM Example:

```
timer = pyb.Timer(2, freq=1000)
ch2 = timer.channel(2, pyb.Timer.PWM, pin=pyb.Pin.board.X2, pulse_width=210000)
ch3 = timer.channel(3, pyb.Timer.PWM, pin=pyb.Pin.board.X3, pulse_width=420000)
```

`timer.`**`counter`**`(`[*value*]`)`
Get or set the timer counter.

`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.`**`freq`**`(`[*value*]`)`
Get or set the frequency for the timer (changes prescaler and period if set).

`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 migh 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` - confgures 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. `deadime` 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.`**`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.readall()      # 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 True if any characters waiting
```

Note: The stream functions `read`, `write` etc Are new in Micro Python since 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)`

Methods

`uart.`**`init`**(*baudrate*, *bits=8*, *parity=None*, *stop=1*, ***, *timeout=1000*, *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.
- `timeout` is the timeout in milliseconds to wait for the first character.
- `timeout_char` is the timeout in milliseconds to wait between characters.
- `read_buf_len` is the character length of the read buffer (0 to disable).

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`**()

Return `True` if any characters waiting, else `False`.

`uart.`**`read`**([*nbytes*])

Read characters. If `nbytes` is specified then read at most that many bytes.

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 `b''` on timeout.

`uart.`**`readall`**`()`
Read as much data as possible.

Return value: a bytes object.

`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`.

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

Return value: the line read.

`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.

`uart.`**`writechar`**`(`*char*`)`
Write a single character on the bus. `char` is an integer to write. Return value: `None`.

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.`**`any`**`()`
Return `True` if any characters waiting, else `False`.

`usb_vcp.`**`close`**`()`

`usb_vcp.`**`read`**`(`[*nbytes*]`)`

`usb_vcp.`**`readall`**`()`

`usb_vcp.`**`readline`**`()`

`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.

`usb_vcp.`**`write`**`(`*buf*`)`

`network` — network configuration

This module provides network drivers and routing configuration.

class CC3k

Constructors

class `network.`**`CC3k`**`(`*spi*, *pin_cs*, *pin_en*, *pin_irq*`)`

Initialise the CC3000 using the given SPI bus and pins and return a CC3k object.

Methods

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

class WIZnet5k

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

Example usage:

```
import wiznet5k
w = wiznet5k.WIZnet5k()
print(w.ipaddr())
w.gethostbyname('micropython.org')
s = w.socket()
s.connect(('192.168.0.2', 8080))
s.send('hello')
print(s.recv(10))
```

Constructors

class `network.`**`WIZnet5k`**`(`*spi*, *pin_cs*, *pin_rst*`)`

Create and return a WIZnet5k object.

Methods

`wiznet5k.`**`ipaddr`**`(`[*(ip*, *subnet*, *gateway*, *dns)*]`)`

Get/set IP address, subnet mask, gateway and DNS.

`wiznet5k.`**`regs`**`()`

Dump WIZnet5k registers.

9.5 The pyboard hardware

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

9.6 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)

9.7 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)

9.8 Micro Python license information

CHAPTER 10

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index