

---

# Micro Python Documentation

*Release 1.3.1*

**Damien P. George**

September 25, 2014



<b>1</b>	<b>Software</b>	<b>3</b>
1.1	General information about the pyboard . . . . .	3
1.2	Micro Python tutorial . . . . .	4
<b>2</b>	<b>The pyboard hardware</b>	<b>25</b>
<b>3</b>	<b>Datasheets for the components on the pyboard</b>	<b>27</b>
<b>4</b>	<b>Indices and tables</b>	<b>29</b>



Here you can find documentation for Micro Python and the pyboard.



## 1.1 General information about the pyboard

### 1.1.1 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.*

### 1.1.2 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.

### 1.1.3 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.

## 1.2 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.

### 1.2.1 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.**



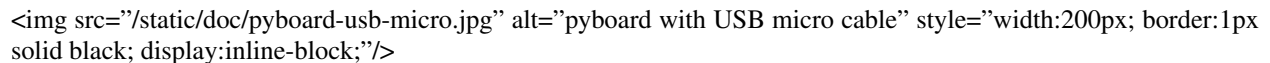
The input voltage must be between 3.6V and 10V.

## 1.2.2 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.

A small image showing a black micro USB cable plugged into the back of a blue and white pyboard.

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](/static/doc/fresh-pyboard/boot.py)` – this script is executed when the pyboard boots up. It sets up various configuration options for the pyboard.
- `[main.py](/static/doc/fresh-pyboard/main.py)` – this is the main script that will contain your Python program. It is executed after `boot.py`.
- `[README.txt](/static/doc/fresh-pyboard/README.txt)` – this contains some very basic information about getting started with the pyboard.
- `[pybcdc.inf](/static/doc/fresh-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!

### 1.2.3 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 `pyboard.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`](<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>). 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 your COM port (eg COM4) in the "Serial Line" box. Finally, click the "Open" button.



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.

### 1.2.4 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 milliseconds. 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 `n`th 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 `VCPIInterrupt` 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.

## 1.2.5 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 USB 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 USB 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 gets 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.

## 1.2.6 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.

## 1.2.7 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 your 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.

## 1.2.8 Making the pyboard act as a USB mouse

The pyboard is a USB device, and can be 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.

**Exercise: make the mouse go around in a circle.**

### Making a mouse with the accelerometer

Now let's 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 it 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 USB switch.
2. While still holding down USB, 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 USB until *only the orange LED is lit*, and then let go of the USB switch.
5. The orange LED should flash quickly 4 times, and then turn off.
6. You are now in safe mode.

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

In `main.py` put the following code:

```
import pyb

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

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 USB 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.

## 1.2.9 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 frequency of  $168\text{ MHz}/2/(83+1)=1\text{ MHz}$  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()
```

## 1.2.10 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_BSRR])
```

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_BSRR`. 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_BSRR])
```

```

# 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)

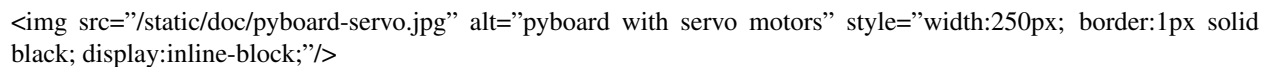
# loop r0 times
sub(r0, r0, 1)
label(loop_entry)
cmp(r0, 0)
bgt(loop1)

```

## 1.2.11 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](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.

A photograph of a Pyboard connected to four hobby servo motors. The servos are plugged into the X1, X2, X3, and X4 pins on the bottom right of the board. Wires connect the servos to the board's headers.

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:

```
>>> serv1.angle(45)
>>> serv1.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:

```
>>> serv1.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

```
>>> serv1.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

```
>>> serv1.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:

```
>>> serv1.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.

```
>>> serv1.speed()
30
>>> serv1.speed(-20)
>>> serv1.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.

### 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.

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

You can make a simple animation using the code:

```
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:

```
>>> 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](/static/doc/examples/mpr121.py) 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](/static/doc/examples/lcddemo.py).

## 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:

```
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 [\[here\]\(/static/doc/examples/wave.py\)](#) and you will also need the `chunk` module available [\[here\]\(/static/doc/examples/chunk.py\)](#). 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 [\[this one\]\(/static/doc/examples/test.wav\)](#). Then you can do:

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

This should play the WAV file.



---

## The pyboard hardware

---



---

## Datasheets for the components on the pyboard

---





---

## Indices and tables

---

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