1 micro:bit Micropython API

Warning

As we work towards a 1.0 release, this API is subject to frequent changes. This page reflects the current micro:bit API in a developer-friendly (but not necessarily kid-friendly) way. The tutorials associated with this documentation are a good place to start for non-developers looking for information.

The microbit module

Everything directly related to interacting with the hardware lives in the microbit module.
For ease of use it's recommended you start all scripts with:

```
from microbit import *
```

The following documentation assumes you have done this.

There are a few functions available directly: sleep for the given number of milliseconds.

```
sleep(ms)
```

Returns the number of milliseconds since the micro:bit was last switched on.

```
running_time()
```

Makes the micro:bit enter panic mode (this usually happens when the DAL runs out of memory, and causes a sad face to be drawn on the display).
The error code can be any arbitrary integer value.

```
panic(error_code)
```

Resets the micro:bit.

```
reset()
```

The rest of the functionality is provided by objects and classes in the microbit module, as described below.

Note that the API exposes integers only (ie no floats are needed, but they may be accepted). We thus use milliseconds for the standard time unit.

Buttons

There are 2 buttons:

`button_a`

`button_b`

These are both objects and have the following methods:

Returns True or False to indicate if the button is pressed at the time of the method call.

```
button.is_pressed()
```

Returns True or False to indicate if the button was pressed since the device started or the last time this method was called.

```
button.was_pressed()
```

Returns the running total of button presses, and resets this counter to zero

```
button.get_presses()
```

The LED display

The `LED` display is exposed via the display object:

Gets the brightness of the pixel (x,y). Brightness can be from 0 (the pixel is off) to 9 (the pixel is at maximum brightness).

```
display.get_pixel(x, y)
```

Sets the brightness of the pixel (x,y) to val (between 0 [off] and 9 [max brightness], inclusive).

```
display.set_pixel(x, y, val)
```

Clears the display.

```
display.clear()
```

Shows the image.

```
display.show(image, delay=0, wait=True, loop=False, clear=False)
```

Shows each image or letter in the iterable, with delay ms. in between each.

`display.show(iterable, delay=400, wait=True, loop=False, clear=False)`

Scrolls a string across the display (more exciting than display.show for written messages).

`display.scroll(string, delay=400)`

Pins

Provide digital and analog input and output functionality, for the pins in the connector. Some pins are connected internally to the I/O that drives the LED matrix and the buttons. Each pin is provided as an object directly in the microbit module. This keeps the `API` relatively flat, making it very easy to use:

`pin0`...`pin16` and `pin19`...`pin20`

Warning: `P17-P18` (inclusive) are unavailable. Each of these pins are instances of the MicroBitPin class, which offers the following `API`:

Value can be 0, 1, False, True

`pin.write_digital(value)`

Returns either 1 or 0

`pin.read_digital()`

Value is between 0 and 1023

`pin.write_analog(value)`

Returns an integer between 0 and 1023

`pin.read_analog()`

Sets the period of the PWM output of the pin in milliseconds

`pin.set_analog_period(int)`

Sets the period of the PWM output of the pin in microseconds

`pin.set_analog_period_microseconds(int)`

Returns boolean

`pin.is_touched()`

2 Images

Note

You don't always need to create one of these yourself - you can access the image shown on the display directly with display.image. display.image is just an instance of Image, so you can use all of the same methods.

Images API:

Creates an empty 5x5 image

```
image = Image()
```

Create an image from a string - each character in the string represents an LED - 0 (or space) is off and 9 is maximum brightness. The colon : indicates the end of a line.

```
image = Image('90009:09090:00900:09090:90009:')
```

Create an empty image of given size

```
image = Image(width, height)
```

Initialises an Image with the specified width and height. The buffer should be an array of length width * height

```
image = Image(width, height, buffer)
```

Methods

Returns the image's width (most often 5)

```
image.width()
```

Returns the image's height (most often 5)

```
image.height()
```

Sets the pixel at the specified position (between 0 and 9). May fail for constant images.

```
image.set_pixel(x, y, value)
```

Gets the pixel at the specified position (between 0 and 9)

`image.get_pixel(x, y)`

Returns a new image created by shifting the picture left 'n' times.

`image.shift_left(n)`

Returns a new image created by shifting the picture right 'n' times.

`image.shift_right(n)`

Returns a new image created by shifting the picture up 'n' times.

`image.shift_up(n)`

Returns a new image created by shifting the picture down 'n' times.

`image.shift_down(n)`

Get a compact string representation of the image

`repr(image)`

Get a more readable string representation of the image

`str(image)`

Operators

Returns a new image created by superimposing the two images `image + image`

Returns a new image created by multiplying the brightness of each pixel by `n`

`image * n`

Built-in images

1. `Image.HEART`
2. `Image.HEART_SMALL`
3. `Image.HAPPY`
4. `Image.SMILE`
5. `Image.SAD`
6. `Image.CONFUSED`
7. `Image.ANGRY`
8. `Image.ASLEEP`
9. `Image.SURPRISED`
10. `Image.SILLY`
11. `Image.FABULOUS`
12. `Image.MEH`
13. `Image.YES`
14. `Image.NO`
15. `Image.CLOCKn`
16. `Image.ARROW_N`
17. `Image.TRIANGLE`
18. `Image.TRIANGLE_LEFT`

19. `Image.CHESSBOARD`
20. `Image.DIAMOND`
21. `Image.DIAMOND_SMALL`
22. `Image.SQUARE`
23. `Image.SQUARE_SMALL`
24. `Image.RABBIT`
25. `Image.COW`
26. `Image.MUSIC_CROTCHET`
27. `Image.MUSIC_QUAVER`
28. `Image.MUSIC_QUAVERS`
29. `Image.PITCHFORK`
30. `Image.XMAS`
31. `Image.PACMAN`
32. `Image.TARGET`
33. `Image.TSHIRT`
34. `Image.ROLLERSKATE`
35. `Image.DUCK`
36. `Image.HOUSE`
37. `Image.TORTOISE`
38. `Image.BUTTERFLY`
39. `Image.STICKFIGURE`
40. `Image.GHOST`
41. `Image.SWORD`
42. `Image.GIRAFFE`
43. `Image.SKULL`
44. `Image.UMBRELLA`
45. `Image.SNAKE`

Arrows pointing

1. `N`,
2. `NE`,
3. `E`,
4. `SE`,
5. `S`,
6. `SW`,
7. `W`,
8. `NW`

`Image.ARROW_direction` `Image.ARROW_NW`

Built-in lists - useful for animations, e.g. `display.show(Image.ALL_CLOCKS)`
`Image.ALL_CLOCKS`
`Image.ALL_ARROWS`

3 The accelerometer

The accelerometer is accessed via the accelerometer object:

Read the X axis of the device. Measured in milli-g.

```
accelerometer.get_x()
```

Read the Y axis of the device. Measured in milli-g.

```
accelerometer.get_y()
```

Read the Z axis of the device. Measured in milli-g.

```
accelerometer.get_z()
```

Get tuple of all three X, Y and Z readings (listed in that order).

```
accelerometer.get_values()
```

Return the name of the current gesture.

```
accelerometer.current_gesture()
```

Return True or False to indicate if the named gesture is currently active.

```
accelerometer.is_gesture(name)
```

Return True or False to indicate if the named gesture was active since the last call.

```
accelerometer.was_gesture(name)
```

Return a tuple of the gesture history. The most recent is listed last.

```
accelerometer.get_gestures()
```

The recognised gestures are:

1. `up`,
2. `down`,
3. `left`,
4. `right`,
5. `face up`,
6. `face down`,
7. `freefall`,
8. `3g`,
9. `6g`,
10. `8g`,
11. `shake`.

4 The compass

The compass is accessed via the compass object:

Calibrate the compass (this is needed to get accurate readings).

```
compass.calibrate()
```

Return a numeric indication of degrees offset from ”north”.

```
compass.heading()
```

Return an numeric indication of the strength of magnetic field around the `micro:bit`.

```
compass.get_field_strength()
```

Returns True or False to indicate if the compass is calibrated.

```
compass.is_calibrated()
```

Resets the compass to a pre-calibration state.

```
compass.clear_calibration()
```

5 I2C bus

There is an I2C bus on the `micro:bit` that is exposed via the i2c object. It has the following methods:

Read `n` bytes from device with addr; `repeat=True` means a stop bit won't be sent.

```
i2c.read(addr, n, repeat=False)
```

Write buf to device with addr; `repeat=True` means a stop bit won't be sent.

```
i2c.write(addr, buf, repeat=False)
```

6 UART

Use uart to communicate with a serial device connected to the device's I/O pins:

Set up communication (use pins 0 `[TX]` and 1 `[RX]`) with a baud rate of 9600.

```
uart.init()
```

Return True or False to indicate if there are incoming characters waiting to be read.

```
uart.any()
```

Return (read) n incoming characters.

```
uart.read(n)
```

Return (read) as much incoming data as possible.

`uart.readall()`

Return (read) all the characters to a newline character is reached.

`uart.readline()`

Read bytes into the referenced buffer.

`uart.readinto(buffer)`

Write bytes from the buffer to the connected device.

`uart.write(buffer)`