1 Microbit Module

The microbit module gives you access to all the hardware that is built-in into your board.

Functions

`microbit.panic(n)`

Enter a panic mode. Requires restart. Pass in an arbitrary integer <= 255 to indicate a status:

`microbit.panic(255)`

`microbit.reset()`

Restart the board.

`microbit.sleep(n)`

Wait for n milliseconds. One second is 1000 milliseconds, so:

`microbit.sleep(1000)`

will pause the execution for one second. n can be an integer or a floating point number.

`microbit.running_time()`

Return the number of milliseconds since the board was switched on or restarted.

`microbit.temperature()`

Return the temperature of the `micro:bit` in degrees Celcius.

2 Buttons

There are two buttons on the board, called `button_a` and `button_b`.

Attributes

`button_a`

A `Button` instance (see below) representing the left button.

`button_b`

Represents the right button.

Classes

`class Button`

Represents a button.

Note

This class is not actually available to the user, it is only used by the two button instances, which are provided already initialized.

`is_pressed()`

Returns True if the specified button button is pressed, and False otherwise.

`was_pressed()`

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

`get_presses()`

Returns the running total of button presses, and resets this total to zero before returning.

Example

Buttons

```
import microbit
while True:
    if microbit.button_a.is_pressed() and
microbit.button_b.is_pressed():
        microbit.display.scroll("AB")
        break
    elif microbit.button_a.is_pressed():
        microbit.display.scroll("A")
    elif microbit.button_b.is_pressed():
        microbit.display.scroll("B")
    microbit.sleep(100)
```

3 Input/Output Pins

The pins are your board's way to communicate with external devices connected to it.
There are 19 pins for your disposal, numbered 0-16 and 19-20. Pins 17 and 18 are not available.

For example, the script below will change the display on the `micro:bit` depending upon the digital reading on pin 0:

Pins

```
from microbit import *

while True:
    if pin0.read_digital():
        display.show(Image.HAPPY)
    else:
        display.show(Image.SAD)
```

Pin Functions

Those pins are available as attributes on the microbit module:microbit.pin0 - microbit.pin20.

Pin	Type	Function
0	Touch	Pad 0
1	Touch	Pad 1
2	Touch	Pad 2
3	Analog	Column 1
4	Analog	Column 2
5	Digital	Button A
6	Digital	Row 2
7	Digital	Row 1
8	Digital	
9	Digital	Row 3
10	Analog	Column 3
11	Digital	Button B
12	Digital	
13	Digital	SPI MOSI
14	Digital	SPI MISO
15	Digital	SPI SCK
16	Digital	
19	Digital	I2C SCL
20	Digital	I2C SDA

The above table summarizes the pins available, their types (see below) and what they are internally connected to.

Pulse-Width Modulation

The pins of your board cannot output analog signal the way an audio amplifier can do it – by modulating the voltage on the pin. Those pins can only either enable the full 3.3V output, or pull it down to 0V.

However, it is still possible to control the brightness of LEDs or speed of an electric motor, by switching that voltage on and off very fast, and controlling how long it is on and how long it is off. This technique is called Pulse-Width Modulation (PWM), and that's what the `write_analog` method below does.

Above you can see the diagrams of three different PWM signals. All of them have the same period (and thus frequency), but they have different duty cycles.

The first one would be generated by `write_analog(511)`, as it has exactly 50 % duty – the power is on half of the time, and off half of the time. The result of that is that the total energy of this signal is the same, as if it was 1.65V instead of 3.3V.

The second signal has 25 % duty cycle, and could be generated with `write_analog(255)`. It has similar effect as if 0.825V was being output on that pin.

The third signal has 75 % duty cycle, and can be generated with `write_analog(767)`. It has three times as much energy, as the second signal, and is equivalent to outputting 2.475V on th pin.

Note that this works well with devices such as motors, which have huge inertia by themselves, or LEDs, which blink too fast for the human eye to see the difference, but will not work so good with generating sound waves. This board can only generate square wave sounds on itself, which sound pretty much like the very old computer games – mostly because those games also only could do that.

Classes

There are three kinds of pins, differing in what is available for them. They are represented by the classes listed below. Note that they form a hierarchy, so that each class has all the functionality of the previous class, and adds its own to that.

Note

Those classes are not actually available for the user, you can't create new instances of them. You can only use the instances already provided, representing the physical pins on your board.

`class microbit.MicroBitDigitalPin`

`read_digital()` Return 1 if the pin is high, and 0 if it's low.

`write_digital(value)` Set the pin to high if value is 1, or to low, if it is 0.

`class microbit.MicroBitAnalogDigitalPin`

`read_analog()`

Read the voltage applied to the pin, and return it as an integer between 0 (meaning 0V) and 1023 (meaning 3.3V).

`write_analog(value)`

Output a PWM signal on the pin, with the duty cycle proportional to the provided value. The value may be either an integer or a floating point number between 0 (0 % duty cycle) and 1023 (100 % duty).

`set_analog_period(period)`

Set the period of the PWM signal being output to period in milliseconds. The minimum valid value is 1ms.

`set_analog_period_microseconds(period)`

Set the period of the PWM signal being output to period in microseconds. The minimum valid value is 256μs.

`class microbit.MicroBitTouchPin`

`is_touched()`

Return `True` if the pin is being touched with a finger, otherwise return `False`.

This test is done by measuring how much resistance there is between the pin and ground. A low resistance gives a reading of True. To get a reliable reading using a finger you may need to touch the ground pin with another part of your body, for example your other hand.

The pull mode for a pin is automatically configured when the pin changes to an input mode. Input modes are when you call `read_analog` / `read_digital` / `is_touched`. The default pull mode for these is, respectively, `NO_PULL`, `PULL_DOWN`, `PULL_UP`. Calling `set_pull` will configure the pin to be in `read_digital` mode with the given pull mode.

Note

Also note, the `micro:bit` has external weak (10M) pull-ups fitted on pins 0, 1 and 2 only, in order for the touch sensing to work. See the edge connector data sheet here:

http://tech.microbit.org/hardware/edgeconnector_ds/

4 Image

The Image class is used to create images that can be displayed easily on the device's LED matrix. Given an image object it's possible to display it via the display API:

```
display.show(Image.HAPPY)
```

Classes

```
class microbit.Image(string)
```

```
class microbit.Image(width=None, height=None, buffer=None)
```

If string is used, it has to consist of digits 0-9 arranged into lines, describing the image, for example:

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

will create a 5×5 image of an X. The end of a line is indicated by a colon. It's also possible to use a newline (n) to indicate the end of a line like this:

```
image = Image("90009\n"
              "09090\n"
              "00900\n"
              "09090\n"
              "90009")
```

The other form creates an empty image with width columns and height rows. Optionally buffer can be an array of width × height integers in range 0-9 to initialize the image.

```
width()
```

Return the number of columns in the image.

```
height()
```

Return the numbers of rows in the image.

`set_pixel(x, y, value)`

Set the brightness of the pixel at column x and row y to the value, which has to be between 0 (dark) and 9 (bright).

This method will raise an exception when called on any of the built-in read-only images, like `Image.HEART`.

`get_pixel(x, y)`

Return the brightness of pixel at column x and row y as an integer between 0 and 9.

`shift_left(n)`

Return a new image created by shifting the picture left by n columns.

`shift_right(n)`

Same as `image.shift_left(-n)`.

`shift_up(n)`

Return a new image created by shifting the picture up by n rows.

`shift_down(n)`

Same as `image.shift_up(-n)`.

`crop(x, y, w, h)`

Return a new image by cropping the picture to a width of w and a height of h, starting with the pixel at column x and row y.

`copy()`

Return an exact copy of the image.

`invert()`

Return a new image by inverting the brightness of the pixels in the source image.

`fill(value)`

Set the brightness of all the pixels in the image to the value, which has to be between 0 (dark) and 9 (bright).

This method will raise an exception when called on any of the built-in read-only images, like `Image.HEART`.

`blit(src, x, y, w, h, xdest=0, ydest=0)`

Copy the rectangle defined by x, y, w, h from the image src into this image at xdest, ydest. Areas in the source rectangle, but outside the source image are treated as having a value of 0.

`shift_left()`, `shift_right()`, `shift_up()`, `shift_down()` and `crop()` can are all implemented by using `blit()`. For example, `img.crop(x, y, w, h)` can be implemented as:

crop

```
def crop(self, x, y, w, h):
    res = Image(w, h)
    res.blit(self, x, y, w, h)
    return res
```

Attributes

The Image class also has the following built-in instances of itself included as its attributes (the attribute names indicate what the image represents):

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.CLOCK12`,
16. `Image.CLOCK11`,
17. `Image.CLOCK10`,
18. `Image.CLOCK9`,
19. `Image.CLOCK8`,
20. `Image.CLOCK7`,
21. `Image.CLOCK6`,
22. `Image.CLOCK5`,
23. `Image.CLOCK4`,
24. `Image.CLOCK3`,
25. `Image.CLOCK2`,
26. `Image.CLOCK1`,
27. `Image.ARROW_N`,
28. `Image.ARROW_NE`,
29. `Image.ARROW_E`,
30. `Image.ARROW_SE`,
31. `Image.ARROW_S`,
32. `Image.ARROW_SW`,
33. `Image.ARROW_W`,
34. `Image.ARROW_NW`,
35. `Image.TRIANGLE`,
36. `Image.TRIANGLE_LEFT`,

37. `Image.CHESSBOARD`,
38. `Image.DIAMOND`,
39. `Image.DIAMOND_SMALL`,
40. `Image.SQUARE`,
41. `Image.SQUARE_SMALL`,
42. `Image.RABBIT`,
43. `Image.COW`,
44. `Image.MUSIC_CROTCHET`,
45. `Image.MUSIC_QUAVER`,
46. `Image.MUSIC_QUAVERS`,
47. `Image.PITCHFORK`,
48. `Image.XMAS`,
49. `Image.PACMAN`,
50. `Image.TARGET`,
51. `Image.TSHIRT`,
52. `Image.ROLLERSKATE`,
53. `Image.DUCK`,
54. `Image.HOUSE`,
55. `Image.TORTOISE`,
56. `Image.BUTTERFLY`,
57. `Image.STICKFIGURE`,
58. `Image.GHOST`,
59. `Image.SWORD`,
60. `Image.GIRAFFE`,
61. `Image.SKULL`,
62. `Image.UMBRELLA`,
63. `Image.SNAKE`.

Finally, related collections of images have been grouped together:

`Image.ALL_CLOCKS`

`Image.ALL_ARROWS`

Operations

`repr(image)`

Get a compact string representation of the image.

`str(image)`

Get a readable string representation of the image.

`image1 + image2`

Create a new image by adding the brightness values from the two images for each pixel.

`image * n`

Create a new image by multiplying the brightness of each pixel by `n`.

Next Previous

5 Display

This module controls the 5×5 LED display on the front of your board. It can be used to display images, animations and even text.

Functions

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

Return the brightness of the `LED` at column x and row y as an integer between 0 (off) and 9 (bright).

`microbit.display.set_pixel(x, y, value)`

Set the brightness of the `LED` at column x and row y to $value$, which has to be an integer between 0 and 9.

`microbit.display.clear()`

Set the brightness of all LEDs to 0 (off).

`microbit.display.show(image)`

Display the image

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

Display images or letters from the iterable in sequence, with delay milliseconds between them.

If `wait` is `True`, this function will block until the animation is finished, otherwise the animation will happen in the background.

If `loop` is `True`, the animation will repeat forever.

If `clear` is `True`, the display will be cleared after the iterable has finished.

Note that the wait, loop and clear arguments must be specified using their keyword.

Note

If using a generator as the `iterable`, then take care not to allocate any memory in the generator as allocating memory in an interrupt is prohibited and will raise a MemoryError.

`microbit.display.scroll(string,delay=150,*, wait=True,loop=False,monospace=False)`

Similar to show, but scrolls the string horizontally instead. The delay parameter controls how fast the text is scrolling.

If wait is True, this function will block until the animation is finished, otherwise the animation will happen in the background.

If loop is True, the animation will repeat forever.

If monospace is True, the characters will all take up 5 pixel-columns in width, otherwise there will be exactly 1 blank pixel-column between each character as they scroll.

Note that the wait, loop and monospace arguments must be specified using their keyword.

`microbit.display.on()`

Use `on()` to turn on the display.

`microbit.display.off()`

Use `off()` to turn off the display (thus allowing you to re-use the `GPIO` pins associated with the display for other purposes).

`microbit.display.is_on()`

Returns `True` if the display is on, otherwise returns `False`.

Example

To continuously scroll a string across the display, and do it in the background, you can use:

```
import microbit

microbit.display.scroll('Hello!', wait=False, loop=True)
```

6 UART

The `uart` module lets you talk to a device connected to your board using a serial interface.

Functions

`microbit.uart.init(baudrate=9600, bits=8,parity=None,stop=1,*,tx=None,rx=None)`

Initialize serial communication with the specified parameters on the specified `tx` and `rx` pins. Note that for correct communication, the parameters have to be the same on both communicating devices.

Warning

Initializing the `UART` on external pins will cause the Python console on `USB` to become unaccessible, as it uses the same hardware. To bring the console back you must reinitialize the `UART` without passing anything for `tx` or `rx` (or passing None to these arguments). This means that calling `uart.init(115200)` is enough to restore the Python console.

The baudrate defines the speed of communication. Common baud rates include:

- 9600
- 14400
- 19200
- 28800
- 38400
- 57600
- 115200

The bits defines the size of bytes being transmitted, and the board only supports 8. The parity parameter defines how parity is checked, and it can be `None`, `microbit.uart.ODD` or `microbit.uart.EVEN`. The `stop` parameter tells the number of stop bits, and has to be 1 for this board.

If `tx` and `rx` are not specified then the internal `USB-UART TX/RX` pins are used which connect to the `USB` serial convertor on the `micro:bit`, thus connecting the `UART` to your `PC`. You can specify any other pins you want by passing the desired pin objects to the `tx` and `rx` parameters.

Note

When connecting the device, make sure you *cross* the wires – the `TX` pin on your board needs to be connected with the `RX` pin on the device, and the `RX` pin – with the `TX` pin on the device. Also make sure the ground pins of both devices are connected.

`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.

`uart.readall()`

Read as much data as possible.

Return value: a bytes object or `None` on timeout.

`uart.readinto(buf[, nbytes])`

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

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

`uart.readline()`

Read a line, ending in a newline character.

Return value: the line read or None on timeout. The newline character is included in the returned bytes.

`uart.write(buf)`

Write the buffer of bytes to the bus.

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

7 SPI

The spi module lets you talk to a device connected to your board using a serial peripheral interface (`SPI`) bus. `SPI` uses a so-called master-slave architecture with a single master. You will need to specify the connections for three signals:

- `SCLK` : Serial Clock (output from master).
- `MOSI` : Master Output, Slave Input (output from master).
- `MISO` : Master Input, Slave Output (output from slave).

Functions

`microbit.spi.init(baudrate=1000000, bits=8, mode=0, sclk=pin13,mosi=pin15, miso=pin14)`

Initialize `SPI` communication with the specified parameters on the specified pins. Note that for correct communication, the parameters have to be the same on both communicating devices.

The baudrate defines the speed of communication.

The bits defines the size of bytes being transmitted. Currently only `bits=8` is supported. However, this may change in the future.

The mode determines the combination of clock polarity and phase according to the following convention, with polarity as the high order bit and phase as the low order bit:

SPI	Mode	Polarity (CPOL) Phase (CPHA)
0	0	0
1	0	1
2	1	0
3	1	1

Polarity (aka CPOL) 0 means that the clock is at logic value 0 when idle and goes high (logic value 1) when active; polarity 1 means the clock is at logic value 1 when idle and goes low (logic value 0) when active. Phase (aka CPHA) 0 means that data is sampled on the leading edge of the clock, and 1 means on the trailing edge.

https://en.wikipedia.org/wiki/Signal_edge.

The `sclk`, `mosi` and `miso` arguments specify the pins to use for each type of signal.

`spi.read(nbytes)`

Read at most nbytes. Returns what was read.

`spi.write(buffer)`

Write the buffer of bytes to the bus.

`spi.write_readinto(out, in)`

Write the out buffer to the bus and read any response into the in buffer. The length of the buffers should be the same. The buffers can be the same object.

8 Bus I^{2}C

The `i2c` module lets you communicate with devices connected to your board using the I^{2}C bus protocol. There can be multiple slave devices connected at the same time, and each one has its own unique

address, that is either fixed for the device or configured on it. Your board acts as the I²C master.

We use 7-bit addressing for devices because of the reasons stated here.

This may be different to other `micro:bit` related solutions.

How exactly you should communicate with the devices, that is, what bytes to send and how to interpret the responses, depends on the device in question and should be described separately in that device's documentation.

Functions

```
microbit.i2c.init(freq=100000, sda=pin20, scl=pin19)
```

Re-initialize peripheral with the specified clock frequency freq on the specified sda and scl pins.

Warning: Changing the I²C pins from defaults will make the accelerometer and compass stop working, as they are connected internally to those pins.

```
microbit.i2c.scan()
```

Scan the bus for devices. Returns a list of 7-bit addresses corresponding to those devices that responded to the scan.

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

Read n bytes from the device with 7-bit address addr. If repeat is True, no stop bit will be sent.

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

Write bytes from buf to the device with 7-bit address addr. If repeat is True, no stop bit will be sent.

Connecting

You should connect the device's `SCL` pin to `micro:bit` pin 19, and the device's SDA pin to `micro:bit` pin 20. You also must connect the device's ground to the `micro:bit` ground (pin `GND`). You may need to power the device using an external power supply or the `micro:bit`.

There are internal pull-up resistors on the I²C lines of the board, but with particularly long wires or large number of devices you may need to add additional pull-up resistors, to ensure noise-free communication.

9 Accelerometer

This object gives you access to the on-board accelerometer. The accelerometer also provides convenience functions for detecting gestures. The recognised gestures are: up, down, left, right, face up, face down, freefall, 3g, 6g, 8g, shake.

Functions

`microbit.accelerometer.get_x()`

Get the acceleration measurement in the x axis, as a positive or negative integer, depending on the direction.

`microbit.accelerometer.get_y()`

Get the acceleration measurement in the y axis, as a positive or negative integer, depending on the direction.

`microbit.accelerometer.get_z()`

Get the acceleration measurement in the z axis, as a positive or negative integer, depending on the direction.

`microbit.accelerometer.get_values()`

Get the acceleration measurements in all axes at once, as a three-element tuple of integers ordered as X, Y, Z.

`microbit.accelerometer.current_gesture()`

Return the name of the current gesture.

Note

`MicroPython` understands the following gesture names:

- 'up',
- 'down',
- 'left',
- 'right',
- 'face up',
- 'face down',
- 'freefall',
- '3g',
- '6g',
- '8g',
- 'shake'.

Gestures are always represented as strings.

`microbit.accelerometer.is_gesture(name)`

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

`microbit.accelerometer.was_gesture(name)`

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

`microbit.accelerometer.get_gestures()`

Return a tuple of the gesture history. The most recent is listed last. Also clears the gesture history before returning.

Examples

A fortune telling magic 8-ball. Ask a question then shake the device for an answer.

Magic 8

```
# Magic 8 ball by Nicholas Tollervey. February 2016.
#
# Ask a question then shake.
#
# This program has been placed into the public domain.
from microbit import *
import random

answers = [
    "It is certain",
    "It is decidedly so",
    "Without a doubt",
    "Yes, definitely",
    "You may rely on it",
    "As I see it, yes",
    "Most likely",
    "Outlook good",
    "Yes",
    "Signs point to yes",
    "Reply hazy try again",
    "Ask again later",
    "Better not tell you now",
    "Cannot predict now",
```

```
    "Concentrate and ask again",
    "Don't count on it",
    "My reply is no",
    "My sources say no",
    "Outlook not so good",
    "Very doubtful",
]

while True:
    display.show('8')
    if accelerometer.was_gesture('shake'):
        display.clear()
        sleep(1000)
        display.scroll(random.choice(answers))
    sleep(10)
```

Simple Slalom. Move the device to avoid the obstacles.

Simple Slalom

```
# Simple Slalom by Larry Hastings, September 2015
#
# This program has been placed into the public domain.

import microbit as m
import random

p = m.display.show

min_x = -1024
max_x = 1024
range_x = max_x - min_x

wall_min_speed = 400
player_min_speed = 200

wall_max_speed = 100
player_max_speed = 50
```

```
speed_max = 12

while True:
    i = m.Image('00000:'*5)
    s = i.set_pixel
    player_x = 2
    wall_y = -1

    hole = 0
    score = 0
    handled_this_wall = False

    wall_speed = wall_min_speed
    player_speed = player_min_speed

    wall_next = 0
    player_next = 0

    while True:
        t = m.running_time()
        player_update = t >= player_next
        wall_update = t >= wall_next
        if not (player_update or wall_update):
            next_event = min(wall_next, player_next)
            delta = next_event - t
            m.sleep(delta)
            continue

        if wall_update:
            # calculate new speeds
            speed = min(score, speed_max)
            wall_speed = wall_min_speed + int((wall_max_speed -\
            wall_min_speed) * speed / speed_max)
            player_speed = player_min_speed + int((player_max_speed -\
            player_min_speed) * speed / speed_max)

            wall_next = t + wall_speed
            if wall_y < 5:
                # erase old wall
```

```
            use_wall_y = max(wall_y, 0)
            for wall_x in range(5):
                if wall_x != hole:
                    s(wall_x, use_wall_y, 0)

    wall_reached_player = (wall_y == 4)
    if player_update:
        player_next = t + player_speed
        # find new x coord
        x = m.accelerometer.get_x()
        x = min(max(min_x, x), max_x)
        # print("x accel", x)
        s(player_x, 4, 0) # turn off old pixel
        x = ((x - min_x) / range_x) * 5
        x = min(max(0, x), 4)
        x = int(x + 0.5)
        # print("have", position, "want", x)

        if not handled_this_wall:
            if player_x < x:
                player_x += 1
            elif player_x > x:
                player_x -= 1
        # print("new", position)
        # print()

    if wall_update:
        # update wall position
        wall_y += 1
        if wall_y == 7:
            wall_y = -1
            hole = random.randrange(5)
            handled_this_wall = False

        if wall_y < 5:
            # draw new wall
            use_wall_y = max(wall_y, 0)
            for wall_x in range(5):
                if wall_x != hole:
```

```
                    s(wall_x, use_wall_y, 6)

        if wall_reached_player and not handled_this_wall:
            handled_this_wall = True
            if (player_x != hole):
                # collision! game over!
                break
            score += 1

        if player_update:
            s(player_x, 4, 9) # turn on new pixel

        p(i)

    p(i.SAD)
    m.sleep(1000)
    m.display.scroll("Score:" + str(score))

    while True:
        if (m.button_a.is_pressed() and m.button_a.is_pressed()):
            break
        m.sleep(100)
```

10 Compass

This module lets you access the built-in electronic compass. Before using, the compass should be calibrated, otherwise the readings may be wrong.

Warning

Calibrating the compass will cause your program to pause until calibration is complete. Calibration consists of a little game to draw a circle on the LED display by rotating the device.

Functions

`microbit.compass.calibrate()`

Starts the calibration process. An instructive message will be scrolled to the user after which they will

need to rotate the device in order to draw a circle on the LED display.

`microbit.compass.is_calibrated()`

Returns True if the compass has been successfully calibrated, and returns False otherwise.

`microbit.compass.clear_calibration()`

Undoes the calibration, making the compass uncalibrated again.

`microbit.compass.get_x()`

Gives the reading of the magnetic force on the x axis, as a positive or negative integer, depending on the direction of the force.

`microbit.compass.get_y()`

Gives the reading of the magnetic force on the x axis, as a positive or negative integer, depending on the direction of the force.

`microbit.compass.get_z()`

Gives the reading of the magnetic force on the x axis, as a positive or negative integer, depending on the direction of the force.

`microbit.compass.heading()`

Gives the compass heading, calculated from the above readings, as an integer in the range from 0 to 360, representing the angle in degrees, clockwise, with north as 0.

`microbit.compass.get_field_strength()`

Returns an integer indication of the magnitude of the magnetic field around the device.

Example

compass.py

```
"""
    Creates a compass.
    The user will need to calibrate the compass first.
    The compass uses the built-in clock images to display
    the position of the needle.
"""
```

```
from microbit import *

# Start calibrating
compass.calibrate()

# Try to keep the needle pointed in (roughly) the correct direction
while True:
    sleep(100)
    needle = ((15 - compass.heading()) // 30) % 12
    display.show(Image.ALL_CLOCKS[needle])
```

11 Bluetooth

While the BBC micro:bit has hardware capable of allowing the device to work as a Bluetooth Low Energy (BLE) device, it only has 16k of RAM. The BLE stack alone takes up 12k RAM which means there's not enough room to run MicroPython.

Future versions of the device may come with 32k RAM which would be sufficient. However, until such time it's highly unlikely MicroPython will support BLE.

Note

MicroPython uses the radio hardware with the radio module. This allows users to create simple yet effective wireless networks of micro:bit devices.

Furthermore, the protocol used in the radio module is a lot simpler than BLE, making it far easier to use in an educational context.

12 Local Persistent File System

It is useful to store data in a persistent manner so that it remains intact between restarts of the device. On traditional computers this is often achieved by a file system consisting of named files that hold raw data, and named directories that contain files. Python supports the various operations needed to work with such file systems.

However, since the micro:bit is a limited device in terms of both hardware and storage capacity MicroPython provides a small subset of the functions needed to persist data on the device. Because of memory constraints there is approximately 30k of storage available on the file system.

Warning

Re-flashing the device will DESTROY YOUR DATA.

Since the file system is stored in the micro:bit's flash memory and flashing the device rewrites all the available flash memory then all your data will be lost if you flash your device.

However, if you switch your device off the data will remain intact until you either delete it (see below) or re-flash the device.

MicroPython on the micro:bit provides a flat file system; i.e. there is no notion of a directory hierarchy, the file system is just a list of named files. Reading and writing a file is achieved via the standard Python open function and the resulting file-like object (representing the file) of types TextIO or BytesIO. Operations for working with files on the file system (for example, listing or deleting files) are contained within the os module.

If a file ends in the .py file extension then it can be imported. For example, a file named hello.py can be imported like this: import hello.

An example session in the MicroPython REPL may look something like this:

```
>>> with open('hello.py', 'w') as hello:
...     hello.write("print('Hello')")
...
>>> import hello
Hello
>>> with open('hello.py') as hello:
...   print(hello.read())
...
print('Hello')
>>> import os
>>> os.listdir()
['hello.py']
>>> os.remove('hello.py')
>>> os.listdir()
```

[]
`open(filename, mode='r')`

Returns a file object representing the file named in the argument filename. The mode defaults to `'r'` which means open for reading in text mode. The other common mode is `'w'` for writing (overwriting the content of the file if it already exists). Two other modes are available to be used in conjunction with the ones describes above: `'t'` means text mode (for reading and writing strings) and `'b'` means binary mode (for reading and writing bytes). If these are not specified then `'t'` (text mode) is assumed. When in text mode the file object will be an instance of `TextIO`. When in binary mode the file object will be an instance of `BytesIO`. For example, use `'rb'` to read binary data from a file.

`class TextIO` `class BytesIO`

Instances of these classes represent files in the `micro:bit`'s flat file system. The `TextIO` class is used to represent text files. The `BytesIO` class is used to represent binary files. They work in exactly the same except that `TextIO` works with strings and `BytesIO` works with bytes.

You do not directly instantiate these classes. Rather, an appropriately configured instance of the class is returned by the open function described above.

`close()`

Flush and close the file. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise an exception.

`name()`

Returns the name of the file the object represents. This will be the same as the filename argument passed into the call to the open function that instantiated the object.

`read(size)`

Read and return at most size characters as a single string or size bytes from the file. As a convenience, if size is unspecified or -1, all the data contained in the file is returned. Fewer than size characters or bytes may be returned if there are less than size characters or bytes remaining to be read from the file.

If 0 characters or bytes are returned, and size was not 0, this indicates end of file.

A `MemoryError` exception will occur if size is larger than the available `RAM`.

`readinto(buf, n=-1)`

Read characters or bytes into the buffer buf. If n is supplied, read n number of bytes or characters into the buffer buf.

`readline(size)`

Read and return one line from the file. If size is specified, at most size characters will be read.

The line terminator is always `'\n'` for strings or `b'\n'` for bytes.

`writable()` Return True if the file supports writing. If False, write() will raise OSError.

`write(buf)`

Write the string or bytes buf to the file and return the number of characters or bytes written.

13 The os Module

`MicroPython` contains an `os` module based upon the `os` module in the Python standard library. It's used for accessing what would traditionally be termed as operating system dependent functionality. Since there is no operating system in `MicroPython` the module provides functions relating to the management of the simple on-device persistent file system and information about the current system.

To access this module you need to:

```
import os
```

We assume you have done this for the examples below.

Functions

`os.listdir()`

Returns a list of the names of all the files contained within the local persistent on-device file system.

`os.remove(filename)`

Removes (deletes) the file named in the argument filename. If the file does not exist an OSError exception will occur.

`os.size(filename)`

Returns the size, in bytes, of the file named in the argument filename. If the file does not exist an OSError

exception will occur.

`os.uname()`

Returns information identifying the current operating system. The return value is an object with five attributes:

- `sysname` - operating system name
- `nodename` - name of machine on network (implementation-defined)
- `release` - operating system release
- `version` - operating system version
- `machine` - hardware identifier

Note

There is no underlying operating system in `MicroPython`. As a result the information returned by the uname function is mostly useful for versioning details.

14 Radio

The radio module allows devices to work together via simple wireless networks.

The radio module is conceptually very simple:

Broadcast messages are of a certain configurable length (up to 251 bytes). Messages received are read from a queue of configurable size (the larger the queue the more `RAM` is used). If the queue is full, new messages are ignored. Reading a message removes it from the queue. Messages are broadcast and received on a preselected channel (numbered 0-83). Broadcasts are at a certain level of power - more power means more range. Messages are filtered by address (like a house number) and group (like a named recipient at the specified address). The rate of throughput can be one of three pre-determined settings. Send and receive bytes to work with arbitrary data. Use `receive_full` to obtain full details about an incoming message: the data, receiving signal strength, and a microsecond timestamp when the message arrived. As a convenience for children, it's easy to send and receive messages as strings. The default configuration is both sensible and compatible with other platforms that target the BBC `micro:bit`. To access this module you need to:

```
import radio
```

We assume you have done this for the examples below.

Constants

`radio.RATE_250KBIT`

Constant used to indicate a throughput of 256 Kbit a second.

`radio.RATE_1MBIT`

Constant used to indicate a throughput of 1 MBit a second.

`radio.RATE_2MBIT`

Constant used to indicate a throughput of 2 MBit a second.

Functions

`radio.on()`

Turns the radio on. This needs to be explicitly called since the radio draws power and takes up memory that you may otherwise need.

`radio.off()`

Turns off the radio, thus saving power and memory.

`radio.config(**kwargs)`

Configures various keyword based settings relating to the radio. The available settings and their sensible default values are listed below.

The `length (default=32)` defines the maximum length, in bytes, of a message sent via the radio. It can be up to 251 bytes long (254 - 3 bytes for S0, LENGTH and S1 preamble).

The `queue (default=3)` specifies the number of messages that can be stored on the incoming message queue. If there are no spaces left on the queue for incoming messages, then the incoming message is dropped.

The `channel (default=7)` can be an integer value from 0 to 83 (inclusive) that defines an arbitrary *channel* to which the radio is tuned. Messages will be sent via this channel and only messages received via this channel will be put onto the incoming message queue. Each step is 1 MHz wide, based at 2400 MHz.

The `power (default=6)` is an integer value from 0 to 7 (inclusive) to indicate the strength of signal used when broadcasting a message. The higher the value the stronger the signal, but the more power is consumed by the device. The numbering translates to positions in the following list of dBm (decibel milliwatt) values: -30, -20, -16, -12, -8, -4, 0, 4.

The `address (default=0x75626974)` is an arbitrary name, expressed as a 32-bit address, that's used to filter incoming packets at the hardware level, keeping only those that match the address you set. The default used by other `micro:bit` related platforms is the default setting used here.

The `group (default=0)` is an 8-bit value (0-255) used with the address when filtering messages. Conceptually, *address* is like a house/office address and *group* is like the person at that address to which you want to send your message.

The `data_rate (default=radio.RATE_1MBIT)` indicates the speed at which data throughput takes place. Can be one of the following contants defined in the radio module : `RATE_250KBIT`, `RATE_1MBIT` or `RATE_2MBIT`.

If `config` is not called then the defaults described above are assumed.

`radio.reset()`

Reset the settings to their default values (as listed in the documentation for the `config` function above).

Note

None of the following send or receive methods will work until the radio is turned on.

`radio.send_bytes(message)`

Sends a message containing bytes.

`radio.receive_bytes()`

Receive the next incoming message on the message queue. Returns None if there are no pending messages. Messages are returned as bytes.

`radio.receive_bytes_into(buffer)`

Receive the next incoming message on the message queue. Copies the message into buffer, trimming the end of the message if necessary. Returns None if there are no pending messages, otherwise it returns the length of the message (which might be more than the length of the buffer).

`radio.send(message)`

Sends a message string. This is the equivalent of `send_bytes(bytes(message, 'utf8'))` but with `b'\x01\x00\x01'` prepended to the front (to make it compatible with other platforms that target the `micro:bit`).

`radio.receive()`

Works in exactly the same way as `receive_bytes` but returns whatever was sent.

Currently, it's equivalent to `str(receive_bytes(), 'utf8')` but with a check that the the first three bytes are `b'\x01\x00\x01'` (to make it compatible with other platforms that may target the `micro:bit`). It strips the prepended bytes before converting to a string.

A ValueError exception is raised if conversion to string fails.

`radio.receive_full()`

Returns a tuple containing three values representing the next incoming message on the message queue. If there are no pending messages then None is returned.

The three values in the tuple represent:

the next incoming message on the message queue as bytes. the `RSSI` (signal strength): a value between 0 (strongest) and -255 (weakest) as measured in dBm. a microsecond timestamp: the value returned by `time.ticks_us()` when the message was received. For example:

```
details = radio.receive_full()
if details:
    msg, rssi, timestamp = details
```

This function is useful for providing information needed for triangulation and/or triliteration with other `micro:bit` devices.

Examples

```
# A \verb`micro:bit` Firefly.
# By Nicholas H.Tollervey. Released to the public domain.
import radio
import random
from microbit import display, Image, button_a, sleep

# Create the "flash" animation frames. Can you work out how it's done?
```

```
flash = [Image().invert()*(i/9) for i in range(9, -1, -1)]

# The radio won't work unless it's switched on.
radio.on()

# Event loop.
while True:
    # Button A sends a "flash" message.
    if button_a.was_pressed():
        radio.send('flash')  # a-ha
    # Read any incoming messages.
    incoming = radio.receive()
    if incoming == 'flash':
        # If there's an incoming "flash" message display
        # the firefly flash animation after a random short
        # pause.
        sleep(random.randint(50, 350))
        display.show(flash, delay=100, wait=False)
        # Randomly re-broadcast the flash message after a
        # slight delay.
        if random.randint(0, 9) == 0:
            sleep(500)
            radio.send('flash')  # a-ha
```

15 Random Number Generation

This module is based upon the random module in the `Python` standard library. It contains functions for generating random behaviour.

To access this module you need to:

```
import random
```

We assume you have done this for the examples below.

Functions

`random.getrandbits(n)`

Returns an integer with n random bits.

Warning

Because the underlying generator function returns at most 30 bits, n may only be a value between 1-30 (inclusive).

`random.seed(n)`

Initialize the random number generator with a known integer `n`. This will give you reproducibly deterministic randomness from a given starting state `(n)`.

`random.randint(a, b)`

Return a random integer N such that $a \leq Nleb$. Alias for

`randrange(a, b+1)`.

`random.randrange(stop)`

Return a randomly selected integer between zero and up to (but not including) stop.

`random.randrange(start, stop)`

Return a randomly selected integer from `range(start, stop)`.

`random.randrange(start, stop, step)`

Return a randomly selected element from `range(start, stop, step)`.

`random.choice(seq)`

Return a random element from the non-empty sequence seq. If seq is empty, raises `IndexError`.

`random.random()`

Return the next random floating point number in the range $[0.0, 1.0)$.

`random.uniform(a, b)`

Return a random floating point number N such that $a \leq Nleb$ for $a \leq b$ and $b \leq N \leq a$ for $b < a$.