micro:bit Micropython API

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.

Note

You can see a list of all available modules by writing `help('modules')` in the REPL.

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
- pin1
- ...
- pin15
- pin16
- *Warning: P17-P18 (inclusive) are unavailable.*
- pin19
- pin20

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
# (see https://en.wikipedia.org/wiki/Pulse-width_modulation)
pin.set_analog_period(int)
# sets the period of the PWM output of the pin in microseconds
# (see https://en.wikipedia.org/wiki/Pulse-width_modulation)
pin.set_analog_period_microseconds(int)
# returns boolean
pin.is_touched()
```

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.
Image.HEART
Image.HEART_SMALL
Image.HAPPY
Image.SMILE
Image.SAD
Image.CONFUSED
Image.ANGRY
Image.ASLEEP
Image.SURPRISED
Image.SILLY
Image.FABULOUS
Image.MEH
Image.YES
Image.NO
```

```
Image.CLOCK12 # clock at 12 o' clock
Image.CLOCK11
... # many clocks (Image.CLOCKn)
Image.CLOCK1 # clock at 1 o'clock
Image.ARROW_N
... # arrows pointing N, NE, E, SE, S, SW, W, NW (microbit.Image.ARROW_direction)
Image.ARROW_NW
Image.TRIANGLE
Image.TRIANGLE_LEFT
Image.CHESSBOARD
Image.DIAMOND
Image.DIAMOND_SMALL
Image.SQUARE
Image.SQUARE_SMALL
Image.RABBIT
Image.COW
Image.MUSIC_CROTCHET
Image.MUSIC_QUAVER
Image.MUSIC_QUAVERS
Image.PITCHFORK
Image.XMAS
Image.PACMAN
Image.TARGET
Image.TSHIRT
Image.ROLLERSKATE
Image.DUCK
Image.HOUSE
Image.TORTOISE
Image.BUTTERFLY
Image.STICKFIGURE
Image.GHOST
Image.SWORD
Image.GIRAFFE
Image.SKULL
Image.UMBRELLA
Image.SNAKE
# built-in lists - useful for animations, e.g. display.show(Image.ALL_CLOCKS)
Image.ALL_CLOCKS
Image.ALL_ARROWS
```

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: `up`, `down`, `left`, `right`, `face up`, `face down`, `freefall`, `3g`, `6g`, `8g`, `shake`.

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()
```

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)
```

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.read()
# 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)
```