CircuitPython Documentation

Release 0.0.0

Damien P. George, Paul Sokolovsky, and contributors

Jun 26, 2020

API and Usage

Welcome to the API reference documentation for Adafruit CircuitPython. This contains low-level API reference docs which may link out to separate *"getting started"* guides. Adafruit has many excellent tutorials available through the Adafruit Learning System.

CHAPTER 1

Adafruit CircuitPython

Status | *Supported Boards* | *Download* | *Documentation* | *Contributing* | *Differences from Micropython* | *Project Structure*

CircuitPython is an *education friendly* open source derivative of MicroPython. CircuitPython supports use on educational development boards designed and sold by Adafruit. Adafruit CircuitPython features unified Python core APIs and a growing list of Adafruit libraries and drivers of that work with it.

1.1 Status

This project is stable. Most APIs should be stable going forward. Those that change will change on major version numbers such as 2.0.0 and 3.0.0.

1.2 Supported Boards

1.2.1 Designed for CircuitPython

- Adafruit CircuitPlayground Express (CircuitPython Guide)
- Adafruit Feather M0 Express (CircuitPython Guide)
- Adafruit Metro M0 Express (CircuitPython Guide)
- Adafruit Gemma M0 (CircuitPython Guide)
- Adafruit ItsyBitsy M0 Express (CircuitPython Guide)
- Adafruit Trinket M0 (CircuitPython Guide)
- Adafruit Metro M4 (CircuitPython Guide)

1.2.2 Other

- Adafruit Feather HUZZAH
- Adafruit Feather M0 Basic
- Adafruit Feather M0 Bluefruit LE (uses M0 Basic binaries)
- Adafruit Feather M0 Adalogger (MicroSD card supported using the Adafruit CircuitPython SD library)
- Arduino Zero

1.3 Download

Official binaries are available through the latest GitHub releases. Continuous (one per commit) builds are available here and includes experimental hardware support.

1.4 Documentation

Guides and videos are available through the Adafruit Learning System under the CircuitPython category and MicroPython category. An API reference is also available on Read the Docs. A collection of awesome resources can be found at Awesome CircuitPython.

Specifically useful documentation when starting out:

- Welcome to CircuitPython
- CircuitPython Essentials
- Example Code

1.5 Contributing

See CONTRIBUTING.md for full guidelines but please be aware that by contributing to this project you are agreeing to the Code of Conduct. Contributors who follow the Code of Conduct are welcome to submit pull requests and they will be promptly reviewed by project admins. Please join the Discord too.

1.6 Differences from MicroPython

CircuitPython:

- includes a ports for MicroChip SAMD21 (Commonly known as M0 in Adafruit product names) and SAMD51 (M4).
- supports only SAMD21, SAMD51, and ESP8266 ports. An nRF port is under development.
- tracks MicroPython's releases (not master).
- Longints (arbitrary-length integers) are enabled for most M0 Express boards (those boards with SPI flash chips external to the microcontroller), and for all M4 builds. Longints are disabled on other boards due to lack of flash space.

1.6.1 Behavior

- The order that files are run and the state that is shared between them. CircuitPython's goal is to clarify the role of each file and make each file independent from each other.
- `boot.py` (or `settings.py`) runs only once on start up before USB is initialized. This lays the ground work for configuring USB at startup rather than it being fixed. Since serial is not available, output is written to `boot_out.txt`.
- `code.py` (or `main.py`) is run after every reload until it finishes or is interrupted. After it is done running, the vm and hardware is reinitialized. **This means you cannot read state from ''code.py'' in the REPL anymore.** CircuitPython's goal for this change includes reduce confusion about pins and memory being used.
- After `code.py` the REPL can be entered by pressing any key. It no longer shares state with `code.py` so it is a fresh vm.
- Autoreload state will be maintained across reload.
- Adds a safe mode that does not run user code after a hard crash or brown out. The hope is that this will make it easier to fix code that causes nasty crashes by making it available through mass storage after the crash. A reset (the button) is needed after its fixed to get back into normal mode.

1.6.2 API

- Unified hardware APIs: audioio, analogio, busio, digitalio, pulseio, touchio, microcontroller, board, bitbangio
- No `machine` API on Atmel SAMD21 port.

1.6.3 Modules

- No module aliasing. (`uos` and `utime` are not available as `os` and `time` respectively.) Instead `os`, `time`, and `random` are CPython compatible.
- New `storage` module which manages file system mounts. (Functionality from `uos` in MicroPython.)
- Modules with a CPython counterpart, such as `time`, `os` and `random`, are strict subsets of their CPython version. Therefore, code from CircuitPython is runnable on CPython but not necessarily the reverse.
- tick count is available as time.monotonic()

1.6.4 atmel-samd21 features

- RGB status LED
- Auto-reload after file write over mass storage. (Disable with `samd.disable_autoreload()`)
- Wait state after boot and main run, before REPL.
- Main is one of these: `code.txt`, `code.py`, `main.py`, `main.txt`
- Boot is one of these: `settings.txt`, `settings.py`, `boot.py`, `boot.txt`

1.7 Project Structure

Here is an overview of the top-level source code directories.

1.7.1 Core

The core code of MicroPython is shared amongst ports including CircuitPython:

- `docs` High level user documentation in Sphinx reStructuredText format.
- `drivers` External device drivers written in Python.
- `examples` A few example Python scripts.
- `extmod` Shared C code used in multiple ports' modules.
- `lib` Shared core C code including externally developed libraries such as FATFS.
- `logo` The MicroPython logo.
- `mpy-cross` A cross compiler that converts Python files to byte code prior to being run in MicroPython. Useful for reducing library size.
- `py` Core Python implementation, including compiler, runtime, and core library.
- `shared-bindings` Shared definition of Python modules, their docs and backing C APIs. Ports must implement the C API to support the corresponding module.
- `shared-module` Shared implementation of Python modules that may be based on `common-hal`.
- `tests` Test framework and test scripts.
- `tools` Various tools, including the pyboard.py module.

1.7.2 Ports

Ports include the code unique to a microcontroller line and also variations based on the board.

- `atmel-samd` Support for SAMD21 based boards such as Arduino Zero, Adafruit Feather M0 Basic, and Adafruit Feather M0 Bluefruit LE.
- `bare-arm` A bare minimum version of MicroPython for ARM MCUs.
- `cc3200` Support for boards based CC3200 from TI such as the WiPy 1.0.
- `esp8266` Support for boards based on ESP8266 WiFi modules such as the Adafruit Feather HUZZAH.
- `minimal` A minimal MicroPython port. Start with this if you want to port MicroPython to another microcontroller.
- `pic16bit` Support for 16-bit PIC microcontrollers.
- `qemu-arm` Support for ARM emulation through QEMU.
- `stmhal` Support for boards based on STM32 microcontrollers including the MicroPython flagship PyBoard.
- `teensy` Support for the Teensy line of boards such as the Teensy 3.1.
- `unix` Support for UNIX.
- `windows` Support for Windows.
- `zephyr` Support for Zephyr, a real-time operating system by the Linux Foundation.

CircuitPython only maintains the `atmel-samd` and `esp8266` ports. The rest are here to maintain compatibility with the MicroPython parent project.

back to top

1.8 Full Table of Contents

1.8.1 Core Modules

These core modules are intended on being consistent across ports. Currently they are only implemented in the SAMD21 and ESP8266 ports. A module may not exist in a port if no underlying hardware support is present or if flash space is limited. For example, a microcontroller without analog features will not have *analogio*.

Support Matrix

NOTE 1: **All Supported** means the following ports are supported: SAMD21, SAMD21 Express, SAMD51, SAMD51 Express, and ESP8266.

NOTE 2: **SAMD** and/or **SAMD Express** without additional numbers, means both SAMD21 & SAMD51 versions are supported.

NOTE 3: The pIRkey SAMD21 board is specialized and may not have modules as listed below.

Module	Supported Ports
analogio	**All Supported**
audiobusio	**SAMD/SAMD Express**
audioio	**SAMD Express**
binascii	**ESP8266**
bitbangio	**SAMD Express, ESP8266**
board	**All Supported**
busio	**All Supported**
digitalio	**All Supported**
gamepad	**SAMD Express, nRF**
hashlib	**ESP8266**
math	**All Supported**
microcontroller	**All Supported**
multiterminal	**ESP8266**
neopixel_write	**All Supported**
nvm	**SAMD Express**
os	**All Supported**
pulseio	**SAMD/SAMD Express**
random	**All Supported**
rotaryio	**SAMD51, SAMD Express**
storage	**All Supported**
struct	**All Supported**
supervisor	**SAMD/SAMD Express**
time	**All Supported**
touchio	**SAMD/SAMD Express**
uheap	**Debug (All)**
usb_hid	**SAMD/SAMD Express**

Modules

_stage — C-level helpers for animation of sprites on a stage

The *_stage* module contains native code to speed-up the `stage Library <https://github.com/python-ugame/circuitpython-stage>‘_. Libraries

Layer – Keep information about a single layer of graphics

class _stage.**Layer**(*width*, *height*, *graphic*, *palette*[, *grid*])

Keep internal information about a layer of graphics (either a Grid or a Sprite) in a format suitable for fast rendering with the render() function.

Parameters

- **width** (int) – The width of the grid in tiles, or 1 for sprites.
- **height** (int) – The height of the grid in tiles, or 1 for sprites.
- **graphic** (bytearray) – The graphic data of the tiles.
- **palette** (bytearray) – The color palette to be used.
- **grid** (bytearray) – The contents of the grid map.

This class is intended for internal use in the stage library and it shouldn’t be used on its own.

move(*x*, *y*)

Set the offset of the layer to the specified values.

frame(*frame*, *rotation*)

Set the animation frame of the sprite, and optionally rotation its graphic.

Text – Keep information about a single text of text

class _stage.**Text**(*width*, *height*, *font*, *palette*, *chars*)

Keep internal information about a text of text in a format suitable for fast rendering with the render() function.

Parameters

- **width** (int) – The width of the grid in tiles, or 1 for sprites.
- **height** (int) – The height of the grid in tiles, or 1 for sprites.
- **font** (bytearray) – The font data of the characters.
- **palette** (bytearray) – The color palette to be used.
- **chars** (bytearray) – The contents of the character grid.

This class is intended for internal use in the stage library and it shouldn’t be used on its own.

move(*x*, *y*)

Set the offset of the text to the specified values.

_stage.**render**(*x0*, *y0*, *x1*, *y1*, *layers*, *buffer*, *spi*)

Render and send to the display a fragment of the screen.

Parameters

- **x0** (int) – Left edge of the fragment.

- **y0** (int) – Top edge of the fragment.
- **x1** (int) – Right edge of the fragment.
- **y1** (int) – Bottom edge of the fragment.
- **layers** (list) – A list of the *Layer* objects.
- **buffer** (bytearray) – A buffer to use for rendering.
- **spi** (SPI) – The SPI bus to use.

Note that this function only sends the raw pixel data. Setting up the display for receiving it and handling the chip-select and data-command pins has to be done outside of it. There are also no sanity checks, outside of the basic overflow checking. The caller is responsible for making the passed parameters valid.

This function is intended for internal use in the `stage` library and all the necessary checks are performed there.

`analogio` — Analog hardware support

The *analogio* module contains classes to provide access to analog IO typically implemented with digital-to-analog (DAC) and analog-to-digital (ADC) converters.

Libraries

`AnalogIn` – read analog voltage

Usage:

```
import analogio
from board import *

adc = analogio.AnalogIn(A1)
val = adc.value
```

class analogio.**AnalogIn**(*pin*)

Use the AnalogIn on the given pin. The reference voltage varies by platform so use `reference_voltage` to read the configured setting.

Parameters **pin** (Pin) – the pin to read from

deinit()

Turn off the AnalogIn and release the pin for other use.

__enter__()

No-op used by Context Managers.

__exit__()

Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

value

The value on the analog pin between 0 and 65535 inclusive (16-bit). (read-only)

Even if the underlying analog to digital converter (ADC) is lower resolution, the value is 16-bit.

reference_voltage

The maximum voltage measurable (also known as the reference voltage) as a *float* in Volts.

`AnalogOut` – output analog voltage

The AnalogOut is used to output analog values (a specific voltage).

Example usage:

```
import analogio
from microcontroller import pin

dac = analogio.AnalogOut(pin.PA02)          # output on pin PA02
dac.value = 32768                           # makes PA02 1.65V
```

class analogio.**AnalogOut**(*pin*)
Use the AnalogOut on the given pin.

Parameters **pin** (Pin) – the pin to output to

deinit()
Turn off the AnalogOut and release the pin for other use.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

value
The value on the analog pin between 0 and 65535 inclusive (16-bit). (write-only)

Even if the underlying digital to analog converter (DAC) is lower resolution, the value is 16-bit.

All classes change hardware state and should be deinitialized when they are no longer needed if the program continues after use. To do so, either call `deinit()` or use a context manager. See *Lifetime and ContextManagers* for more info.

For example:

```
import analogio
from board import *

pin = analogio.AnalogIn(A0)
print(pin.value)
pin.deinit()
```

This example will initialize the the device, read *value* and then *deinit()* the hardware. The last step is optional because CircuitPython will do it automatically after the program finishes.

`audiobusio` — Support for audio input and output over digital bus

The *audiobusio* module contains classes to provide access to audio IO over digital buses. These protocols are used to communicate audio to other chips in the same circuit. It doesn't include audio interconnect protocols such as S/PDIF.

Libraries

`I2SOut` – Output an I2S audio signal

I2S is used to output an audio signal on an I2S bus.

class audiobusio.**I2SOut**(*bit_clock*, *word_select*, *data*, ***, *left_justified*)

Create a I2SOut object associated with the given pins.

Parameters

- **bit_clock** (Pin) – The bit clock (or serial clock) pin
- **word_select** (Pin) – The word select (or left/right clock) pin
- **data** (Pin) – The data pin
- **left_justified** (bool) – True when data bits are aligned with the word select clock. False when they are shifted by one to match classic I2S protocol.

Simple 8ksps 440 Hz sine wave on Metro M0 Express using UDA1334 Breakout:

```
import audiobusio
import audioio
import board
import array
import time
import math

# Generate one period of sine wav.
length = 8000 // 440
sine_wave = array.array("H", [0] * length)
for i in range(length):
    sine_wave[i] = int(math.sin(math.pi * 2 * i / 18) * (2 ** 15) + 2 ** 15)

sine_wave = audiobusio.RawSample(sine_wave, sample_rate=8000)
i2s = audiobusio.I2SOut(board.D1, board.D0, board.D9)
i2s.play(sine_wave, loop=True)
time.sleep(1)
i2s.stop()
```

Playing a wave file from flash:

```
import board
import audioio
import audiobusio
import digitalio


f = open("cplay-5.1-16bit-16khz.wav", "rb")
wav = audioio.WaveFile(f)

a = audiobusio.I2SOut(board.D1, board.D0, board.D9)

print("playing")
a.play(wav)
while a.playing:
  pass
print("stopped")
```

deinit()

Deinitialises the I2SOut and releases any hardware resources for reuse.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

play(*sample*, *, *loop=False*)
Plays the sample once when loop=False and continuously when loop=True. Does not block. Use *playing* to block.

Sample must be an *audioio.WaveFile* or *audioio.RawSample*.

The sample itself should consist of 8 bit or 16 bit samples.

stop()
Stops playback.

playing
True when the audio sample is being output. (read-only)

pause()
Stops playback temporarily while remembering the position. Use *resume* to resume playback.

resume()
Resumes sample playback after *pause()*.

paused
True when playback is paused. (read-only)

PDMIn – Record an input PDM audio stream

PDMIn can be used to record an input audio signal on a given set of pins.

class audiobusio.**PDMIn**(*clock_pin*, *data_pin*, *, *sample_rate=16000*, *bit_depth=8*, *mono=True*, *oversample=64*, *startup_delay=0.11*)
Create a PDMIn object associated with the given pins. This allows you to record audio signals from the given pins. Individual ports may put further restrictions on the recording parameters. The overall sample rate is determined by *sample_rate* x oversample, and the total must be 1MHz or higher, so *sample_rate* must be a minimum of 16000.

Parameters

- **clock_pin** (Pin) – The pin to output the clock to
- **data_pin** (Pin) – The pin to read the data from
- **sample_rate** (int) – Target sample_rate of the resulting samples. Check *sample_rate* for actual value. Minimum sample_rate is about 16000 Hz.
- **bit_depth** (int) – Final number of bits per sample. Must be divisible by 8
- **mono** (bool) – True when capturing a single channel of audio, captures two channels otherwise
- **oversample** (int) – Number of single bit samples to decimate into a final sample. Must be divisible by 8
- **startup_delay** (float) – seconds to wait after starting microphone clock to allow microphone to turn on. Most require only 0.01s; some require 0.1s. Longer is safer. Must be in range 0.0-1.0 seconds.

Record 8-bit unsigned samples to buffer:

```
import audiobusio
import board

# Prep a buffer to record into
b = bytearray(200)
with audiobusio.PDMIn(board.MICROPHONE_CLOCK, board.MICROPHONE_DATA, sample_
↪rate=16000) as mic:
    mic.record(b, len(b))
```

Record 16-bit unsigned samples to buffer:

```
import audiobusio
import board

# Prep a buffer to record into. The array interface doesn't allow for
# constructing with a set size so we append to it until we have the size
# we want.
b = array.array("H")
for i in range(200):
    b.append(0)
with audiobusio.PDMIn(board.MICROPHONE_CLOCK, board.MICROPHONE_DATA, sample_
↪rate=16000, bit_depth=16) as mic:
    mic.record(b, len(b))
```

deinit()
Deinitialises the PDMIn and releases any hardware resources for reuse.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context.

record(*destination*, *destination_length*)
Records destination_length bytes of samples to destination. This is blocking.

An IOError may be raised when the destination is too slow to record the audio at the given rate. For internal flash, writing all 1s to the file before recording is recommended to speed up writes.

Returns The number of samples recorded. If this is less than `destination_length`, some samples were missed due to processing time.

sample_rate
The actual sample_rate of the recording. This may not match the constructed sample rate due to internal clock limitations.

All libraries change hardware state and should be deinitialized when they are no longer needed. To do so, either call `deinit()` or use a context manager.

audioio — Support for audio input and output

The *audioio* module contains classes to provide access to audio IO.

Libraries

`AudioOut` – Output an analog audio signal

AudioOut can be used to output an analog audio signal on a given pin.

class audioio.**AudioOut**(*left_channel*, *right_channel=None*)

Create a AudioOut object associated with the given pin(s). This allows you to play audio signals out on the given pin(s).

Parameters

- **left_channel** (Pin) – The pin to output the left channel to
- **right_channel** (Pin) – The pin to output the right channel to

Simple 8ksps 440 Hz sin wave:

```
import audioio
import board
import array
import time
import math

# Generate one period of sine wav.
length = 8000 // 440
sine_wave = array.array("H", [0] * length)
for i in range(length):
    sine_wave[i] = int(math.sin(math.pi * 2 * i / 18) * (2 ** 15) + 2 ** 15)

dac = audioio.AudioOut(board.SPEAKER)
sine_wave = audioio.RawSample(sine_wave, mono=True, sample_rate=8000)
dac.play(sine_wave, loop=True)
time.sleep(1)
sample.stop()
```

Playing a wave file from flash:

```
import board
import audioio
import digitalio

# Required for CircuitPlayground Express
speaker_enable = digitalio.DigitalInOut(board.SPEAKER_ENABLE)
speaker_enable.switch_to_output(value=True)

data = open("cplay-5.1-16bit-16khz.wav", "rb")
wav = audioio.WaveFile(data)
a = audioio.AudioOut(board.A0)

print("playing")
a.play(wav)
while a.playing:
  pass
print("stopped")
```

deinit()

Deinitialises the AudioOut and releases any hardware resources for reuse.

__enter__()

No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

play(*sample*, *, *loop=False*)
Plays the sample once when loop=False and continuously when loop=True. Does not block. Use *playing* to block.

Sample must be an *audioio.WaveFile* or *audioio.RawSample*.

The sample itself should consist of 16 bit samples. Microcontrollers with a lower output resolution will use the highest order bits to output. For example, the SAMD21 has a 10 bit DAC that ignores the lowest 6 bits when playing 16 bit samples.

stop()
Stops playback and resets to the start of the sample.

playing
True when an audio sample is being output even if *paused*. (read-only)

pause()
Stops playback temporarily while remembering the position. Use *resume* to resume playback.

resume()
Resumes sample playback after *pause()*.

paused
True when playback is paused. (read-only)

RawSample – A raw audio sample buffer

An in-memory sound sample

class audioio.**RawSample**(*buffer*, *, *channel_count=1*, *sample_rate=8000*)
Create a RawSample based on the given buffer of signed values. If channel_count is more than 1 then each channel's samples should alternate. In other words, for a two channel buffer, the first sample will be for channel 1, the second sample will be for channel two, the third for channel 1 and so on.

Parameters

- **buffer** (*array*) – An *array.array* with samples
- **channel_count** (*int*) – The number of channels in the buffer
- **sample_rate** (*int*) – The desired playback sample rate

Simple 8ksps 440 Hz sin wave:

```
import audioio
import board
import array
import time
import math

# Generate one period of sine wav.
length = 8000 // 440
sine_wave = array.array("h", [0] * length)
for i in range(length):
    sine_wave[i] = int(math.sin(math.pi * 2 * i / 18) * (2 ** 15))
```

(continues on next page)

(continued from previous page)

```
dac = audioio.AudioOut(board.SPEAKER)
sine_wave = audioio.RawSample(sine_wave)
dac.play(sine_wave, loop=True)
time.sleep(1)
sample.stop()
```

deinit()
Deinitialises the AudioOut and releases any hardware resources for reuse.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

sample_rate
32 bit value that dictates how quickly samples are played in Hertz (cycles per second). When the sample is looped, this can change the pitch output without changing the underlying sample. This will not change the sample rate of any active playback. Call `play` again to change it.

`WaveFile` – Load a wave file for audio playback

A .wav file prepped for audio playback. Only mono and stereo files are supported. Samples must be 8 bit unsigned or 16 bit signed.

class audioio.**WaveFile**(*filename*)
Load a .wav file for playback with *audioio.AudioOut* or *audiobusio.I2SOut*.

Parameters **file** (*bytes-like*) – Already opened wave file

Playing a wave file from flash:

```
import board
import audioio
import digitalio

# Required for CircuitPlayground Express
speaker_enable = digitalio.DigitalInOut(board.SPEAKER_ENABLE)
speaker_enable.switch_to_output(value=True)

data = open("cplay-5.1-16bit-16khz.wav", "rb")
wav = audioio.WaveFile(data)
a = audioio.AudioOut(board.A0)

print("playing")
a.play(wav)
while a.playing:
  pass
print("stopped")
```

deinit()
Deinitialises the WaveFile and releases all memory resources for reuse.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

sample_rate
32 bit value that dictates how quickly samples are loaded into the DAC in Hertz (cycles per second). When the sample is looped, this can change the pitch output without changing the underlying sample.

All classes change hardware state and should be deinitialized when they are no longer needed if the program continues after use. To do so, either call `deinit()` or use a context manager. See *Lifetime and ContextManagers* for more info.

`bitbangio` — Digital protocols implemented by the CPU

The *`bitbangio`* module contains classes to provide digital bus protocol support regardless of whether the underlying hardware exists to use the protocol.

First try to use *`busio`* module instead which may utilize peripheral hardware to implement the protocols. Native implementations will be faster than bitbanged versions and have more capabilities.

Libraries

`I2C` — Two wire serial protocol

class bitbangio.**I2C**(*scl*, *sda*, *, *frequency=400000*)
I2C is a two-wire protocol for communicating between devices. At the physical level it consists of 2 wires: SCL and SDA, the clock and data lines respectively.

Parameters

- **scl** (`Pin`) – The clock pin
- **sda** (`Pin`) – The data pin
- **frequency** (`int`) – The clock frequency of the bus
- **timeout** (`int`) – The maximum clock stretching timeout in microseconds

deinit()
Releases control of the underlying hardware so other classes can use it.

__enter__()
No-op used in Context Managers.

__exit__()
Automatically deinitializes the hardware on context exit. See *Lifetime and ContextManagers* for more info.

scan()
Scan all I2C addresses between 0x08 and 0x77 inclusive and return a list of those that respond. A device responds if it pulls the SDA line low after its address (including a read bit) is sent on the bus.

try_lock()
Attempts to grab the I2C lock. Returns True on success.

unlock()
Releases the I2C lock.

readfrom_into(*address*, *buffer*, ***, *start=0*, *end=len(buffer)*)
Read into `buffer` from the slave specified by `address`. The number of bytes read will be the length of `buffer`. At least one byte must be read.

If `start` or `end` is provided, then the buffer will be sliced as if `buffer[start:end]`. This will not cause an allocation like `buf[start:end]` will so it saves memory.

Parameters

- **address** (`int`) – 7-bit device address
- **buffer** (`bytearray`) – buffer to write into
- **start** (`int`) – Index to start writing at
- **end** (`int`) – Index to write up to but not include

writeto(*address*, *buffer*, ***, *start=0*, *end=len(buffer)*, *stop=True*)
Write the bytes from `buffer` to the slave specified by `address`. Transmits a stop bit if `stop` is set.

If `start` or `end` is provided, then the buffer will be sliced as if `buffer[start:end]`. This will not cause an allocation like `buffer[start:end]` will so it saves memory.

Writing a buffer or slice of length zero is permitted, as it can be used to poll for the existence of a device.

Parameters

- **address** (`int`) – 7-bit device address
- **buffer** (`bytearray`) – buffer containing the bytes to write
- **start** (`int`) – Index to start writing from
- **end** (`int`) – Index to read up to but not include
- **stop** (`bool`) – If true, output an I2C stop condition after the buffer is written

`OneWire` – Lowest-level of the Maxim OneWire protocol

`OneWire` implements the timing-sensitive foundation of the Maxim (formerly Dallas Semi) OneWire protocol.

Protocol definition is here: https://www.maximintegrated.com/en/app-notes/index.mvp/id/126

class bitbangio.**OneWire**(*pin*)
Create a OneWire object associated with the given pin. The object implements the lowest level timing-sensitive bits of the protocol.

Parameters **pin** (`Pin`) – Pin to read pulses from.

Read a short series of pulses:

```
import bitbangio
import board

onewire = bitbangio.OneWire(board.D7)
onewire.reset()
onewire.write_bit(True)
onewire.write_bit(False)
print(onewire.read_bit())
```

deinit()
Deinitialize the OneWire bus and release any hardware resources for reuse.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

reset()
Reset the OneWire bus

read_bit()
Read in a bit

Returns bit state read

Return type *bool*

write_bit(*value*)
Write out a bit based on value.

SPI – a 3-4 wire serial protocol

SPI is a serial protocol that has exclusive pins for data in and out of the master. It is typically faster than *I2C* because a separate pin is used to control the active slave rather than a transmitted address. This class only manages three of the four SPI lines: `clock`, `MOSI`, `MISO`. Its up to the client to manage the appropriate slave select line. (This is common because multiple slaves can share the `clock`, `MOSI` and `MISO` lines and therefore the hardware.)

class bitbangio.**SPI**(*clock*, *MOSI=None*, *MISO=None*)

Construct an SPI object on the given pins.

Parameters

- **clock** (`Pin`) – the pin to use for the clock.
- **MOSI** (`Pin`) – the Master Out Slave In pin.
- **MISO** (`Pin`) – the Master In Slave Out pin.

deinit()
Turn off the SPI bus.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

configure(*, *baudrate=100000*, *polarity=0*, *phase=0*, *bits=8*)
Configures the SPI bus. Only valid when locked.

Parameters

- **baudrate** (`int`) – the clock rate in Hertz
- **polarity** (`int`) – the base state of the clock line (0 or 1)
- **phase** (`int`) – the edge of the clock that data is captured. First (0) or second (1). Rising or falling depends on clock polarity.
- **bits** (`int`) – the number of bits per word

try_lock()
Attempts to grab the SPI lock. Returns True on success.

Returns True when lock has been grabbed

Return type *bool*

unlock()
Releases the SPI lock.

write(*buf*)
Write the data contained in `buf`. Requires the SPI being locked. If the buffer is empty, nothing happens.

readinto(*buf*)
Read into the buffer specified by `buf` while writing zeroes. Requires the SPI being locked. If the number of bytes to read is 0, nothing happens.

write_readinto(*buffer_out*, *buffer_in*, *, *out_start=0*, *out_end=len(buffer_out)*, *in_start=0*, *in_end=len(buffer_in)*)
Write out the data in `buffer_out` while simultaneously reading data into `buffer_in`. The lengths of the slices defined by `buffer_out[out_start:out_end]` and `buffer_in[in_start:in_end]` must be equal. If buffer slice lengths are both 0, nothing happens.

Parameters

- **buffer_out** (`bytearray`) – Write out the data in this buffer
- **buffer_in** (`bytearray`) – Read data into this buffer
- **out_start** (`int`) – Start of the slice of buffer_out to write out: `buffer_out[out_start:out_end]`
- **out_end** (`int`) – End of the slice; this index is not included
- **in_start** (`int`) – Start of the slice of `buffer_in` to read into: `buffer_in[in_start:in_end]`
- **in_end** (`int`) – End of the slice; this index is not included

All classes change hardware state and should be deinitialized when they are no longer needed if the program continues after use. To do so, either call `deinit()` or use a context manager. See *Lifetime and ContextManagers* for more info.

For example:

```
import bitbangio
from board import *

i2c = bitbangio.I2C(SCL, SDA)
print(i2c.scan())
i2c.deinit()
```

This example will initialize the the device, run *scan()* and then *deinit()* the hardware. The last step is optional because CircuitPython automatically resets hardware after a program finishes.

board — Board specific pin names

Common container for board base pin names. These will vary from board to board so don't expect portability when using this module.

busio — Hardware accelerated behavior

The *busio* module contains classes to support a variety of serial protocols.

When the microcontroller does not support the behavior in a hardware accelerated fashion it may internally use a bitbang routine. However, if hardware support is available on a subset of pins but not those provided, then a RuntimeError will be raised. Use the *bitbangio* module to explicitly bitbang a serial protocol on any general purpose pins.

Libraries

I2C — Two wire serial protocol

class busio.**I2C**(*scl*, *sda*, ***, *frequency=400000*)

I2C is a two-wire protocol for communicating between devices. At the physical level it consists of 2 wires: SCL and SDA, the clock and data lines respectively.

See also:

Using this class directly requires careful lock management. Instead, use I2CDevice to manage locks.

See also:

Using this class to directly read registers requires manual bit unpacking. Instead, use an existing driver or make one with Register data descriptors.

Parameters

- **scl** (Pin) – The clock pin
- **sda** (Pin) – The data pin
- **frequency** (int) – The clock frequency in Hertz
- **timeout** (int) – The maximum clock stretching timeut - only for bitbang

deinit()

Releases control of the underlying hardware so other classes can use it.

__enter__()

No-op used in Context Managers.

__exit__()

Automatically deinitializes the hardware on context exit. See *Lifetime and ContextManagers* for more info.

scan()

Scan all I2C addresses between 0x08 and 0x77 inclusive and return a list of those that respond.

Returns List of device ids on the I2C bus

Return type *list*

try_lock()

Attempts to grab the I2C lock. Returns True on success.

Returns True when lock has been grabbed

Return type *bool*

unlock()

Releases the I2C lock.

readfrom_into(*address*, *buffer*, ***, *start=0*, *end=len(buffer)*)
Read into `buffer` from the slave specified by `address`. The number of bytes read will be the length of `buffer`. At least one byte must be read.

If `start` or `end` is provided, then the buffer will be sliced as if `buffer[start:end]`. This will not cause an allocation like `buf[start:end]` will so it saves memory.

Parameters

- **address** (`int`) – 7-bit device address
- **buffer** (`bytearray`) – buffer to write into
- **start** (`int`) – Index to start writing at
- **end** (`int`) – Index to write up to but not include

writeto(*address*, *buffer*, ***, *start=0*, *end=len(buffer)*, *stop=True*)
Write the bytes from `buffer` to the slave specified by `address`. Transmits a stop bit if `stop` is set.

If `start` or `end` is provided, then the buffer will be sliced as if `buffer[start:end]`. This will not cause an allocation like `buffer[start:end]` will so it saves memory.

Writing a buffer or slice of length zero is permitted, as it can be used to poll for the existence of a device.

Parameters

- **address** (`int`) – 7-bit device address
- **buffer** (`bytearray`) – buffer containing the bytes to write
- **start** (`int`) – Index to start writing from
- **end** (`int`) – Index to read up to but not include
- **stop** (`bool`) – If true, output an I2C stop condition after the buffer is written

`OneWire` – Lowest-level of the Maxim OneWire protocol

`OneWire` implements the timing-sensitive foundation of the Maxim (formerly Dallas Semi) OneWire protocol.

Protocol definition is here: https://www.maximintegrated.com/en/app-notes/index.mvp/id/126

class busio.**OneWire**(*pin*)
Create a OneWire object associated with the given pin. The object implements the lowest level timing-sensitive bits of the protocol.

Parameters **pin** (`Pin`) – Pin connected to the OneWire bus

Read a short series of pulses:

```
import busio
import board

onewire = busio.OneWire(board.D7)
onewire.reset()
onewire.write_bit(True)
onewire.write_bit(False)
print(onewire.read_bit())
```

deinit()
Deinitialize the OneWire bus and release any hardware resources for reuse.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

reset()
Reset the OneWire bus and read presence

Returns False when at least one device is present

Return type *bool*

read_bit()
Read in a bit

Returns bit state read

Return type *bool*

write_bit(*value*)
Write out a bit based on value.

SPI – a 3-4 wire serial protocol

SPI is a serial protocol that has exclusive pins for data in and out of the master. It is typically faster than *I2C* because a separate pin is used to control the active slave rather than a transitted address. This class only manages three of the four SPI lines: `clock`, `MOSI`, `MISO`. Its up to the client to manage the appropriate slave select line. (This is common because multiple slaves can share the `clock`, `MOSI` and `MISO` lines and therefore the hardware.)

class busio.**SPI**(*clock*, *MOSI=None*, *MISO=None*)

Construct an SPI object on the given pins.

See also:

Using this class directly requires careful lock management. Instead, use `SPIDevice` to manage locks.

See also:

Using this class to directly read registers requires manual bit unpacking. Instead, use an existing driver or make one with Register data descriptors.

Parameters

- **clock** (`Pin`) – the pin to use for the clock.
- **MOSI** (`Pin`) – the Master Out Slave In pin.
- **MISO** (`Pin`) – the Master In Slave Out pin.

deinit()
Turn off the SPI bus.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

configure(*, *baudrate=100000*, *polarity=0*, *phase=0*, *bits=8*)
Configures the SPI bus. Only valid when locked.

Parameters

- **baudrate** (int) – the desired clock rate in Hertz. The actual clock rate may be higher or lower due to the granularity of available clock settings. Check the *frequency* attribute for the actual clock rate. **Note:** on the SAMD21, it is possible to set the baud rate to 24 MHz, but that speed is not guaranteed to work. 12 MHz is the next available lower speed, and is within spec for the SAMD21.
- **polarity** (int) – the base state of the clock line (0 or 1)
- **phase** (int) – the edge of the clock that data is captured. First (0) or second (1). Rising or falling depends on clock polarity.
- **bits** (int) – the number of bits per word

try_lock()
Attempts to grab the SPI lock. Returns True on success.

Returns True when lock has been grabbed

Return type *bool*

unlock()
Releases the SPI lock.

write(*buffer*, *, *start=0*, *end=len(buffer)*)
Write the data contained in `buffer`. The SPI object must be locked. If the buffer is empty, nothing happens.

Parameters

- **buffer** (bytearray) – Write out the data in this buffer
- **start** (int) – Start of the slice of `buffer` to write out: `buffer[start:end]`
- **end** (int) – End of the slice; this index is not included

readinto(*buffer*, *, *start=0*, *end=len(buffer)*, *write_value=0*)
Read into `buffer` while writing `write_value` for each byte read. The SPI object must be locked. If the number of bytes to read is 0, nothing happens.

Parameters

- **buffer** (bytearray) – Read data into this buffer
- **start** (int) – Start of the slice of `buffer` to read into: `buffer[start:end]`
- **end** (int) – End of the slice; this index is not included
- **write_value** (int) – Value to write while reading. (Usually ignored.)

write_readinto(*buffer_out*, *buffer_in*, *, *out_start=0*, *out_end=len(buffer_out)*, *in_start=0*, *in_end=len(buffer_in)*)
Write out the data in `buffer_out` while simultaneously reading data into `buffer_in`. The SPI object must be locked. The lengths of the slices defined by `buffer_out[out_start:out_end]` and `buffer_in[in_start:in_end]` must be equal. If buffer slice lengths are both 0, nothing happens.

Parameters

- **buffer_out** (bytearray) – Write out the data in this buffer
- **buffer_in** (bytearray) – Read data into this buffer

- **out_start** (int) – Start of the slice of buffer_out to write out: `buffer_out[out_start:out_end]`
- **out_end** (int) – End of the slice; this index is not included
- **in_start** (int) – Start of the slice of `buffer_in` to read into: `buffer_in[in_start:in_end]`
- **in_end** (int) – End of the slice; this index is not included

frequency
The actual SPI bus frequency. This may not match the frequency requested due to internal limitations.

UART – a bidirectional serial protocol

class busio.**UART**(*tx*, *rx*, *, *baudrate=9600*, *bits=8*, *parity=None*, *stop=1*, *timeout=1000*, *receiver_buffer_size=64*)
A common bidirectional serial protocol that uses an an agreed upon speed rather than a shared clock line.

Parameters

- **tx** (Pin) – the pin to transmit with, or None if this UART is receive-only.
- **rx** (Pin) – the pin to receive on, or None if this UART is transmit-only.
- **baudrate** (int) – the transmit and receive speed.

deinit()
Deinitialises the UART and releases any hardware resources for reuse.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

read(*nbytes=None*)
Read characters. If `nbytes` is specified then read at most that many bytes. Otherwise, read everything that arrives until the connection times out. Providing the number of bytes expected is highly recommended because it will be faster.

Returns Data read

Return type *bytes* or None

readinto(*buf*, *nbytes=None*)
Read bytes into the `buf`. If `nbytes` is specified then read at most that many bytes. Otherwise, read at most `len(buf)` bytes.

Returns number of bytes read and stored into `buf`

Return type *bytes* or None

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

Returns the line read

Return type *int* or None

write(*buf*)
Write the buffer of bytes to the bus.

Returns the number of bytes written

Return type *int* or None

baudrate
The current baudrate.

in_waiting
The number of bytes in the input buffer, available to be read

reset_input_buffer()
Discard any unread characters in the input buffer.

class busio.UART.**Parity**
Enum-like class to define the parity used to verify correct data transfer.

ODD
Total number of ones should be odd.

EVEN
Total number of ones should be even.

All classes change hardware state and should be deinitialized when they are no longer needed if the program continues after use. To do so, either call `deinit()` or use a context manager. See *Lifetime and ContextManagers* for more info.

For example:

```
import busio
from board import *

i2c = busio.I2C(SCL, SDA)
print(i2c.scan())
i2c.deinit()
```

This example will initialize the the device, run *scan()* and then *deinit()* the hardware. The last step is optional because CircuitPython automatically resets hardware after a program finishes.

digitalio — Basic digital pin support

The *digitalio* module contains classes to provide access to basic digital IO.

Libraries

DigitalInOut – digital input and output

A DigitalInOut is used to digitally control I/O pins. For analog control of a pin, see the `AnalogIn` and `AnalogOut` classes.

class digitalio.**DigitalInOut**(*pin*)
Create a new DigitalInOut object associated with the pin. Defaults to input with no pull. Use *switch_to_input()* and *switch_to_output()* to change the direction.

Parameters **pin** (`Pin`) – The pin to control

deinit()
Turn off the DigitalInOut and release the pin for other use.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

switch_to_output(*value=False*, *drive_mode=digitalio.DriveMode.PUSH_PULL*)
Set the drive mode and value and then switch to writing out digital values.

Parameters

- **value** (bool) – default value to set upon switching
- **drive_mode** (DriveMode) – drive mode for the output

switch_to_input(*pull=None*)
Set the pull and then switch to read in digital values.

Parameters **pull** (Pull) – pull configuration for the input

Example usage:

```
import digitalio
import board

switch = digitalio.DigitalInOut(board.SLIDE_SWITCH)
switch.switch_to_input(pull=digitalio.Pull.UP)
# Or, after switch_to_input
switch.pull = digitalio.Pull.UP
print(switch.value)
```

direction
The direction of the pin.

Setting this will use the defaults from the corresponding *switch_to_input()* or *switch_to_output()* method. If you want to set pull, value or drive mode prior to switching, then use those methods instead.

value
The digital logic level of the pin.

drive_mode
The pin drive mode. One of:

- *digitalio.DriveMode.PUSH_PULL*
- *digitalio.DriveMode.OPEN_DRAIN*

pull
The pin pull direction. One of:

- *digitalio.Pull.UP*
- *digitalio.Pull.DOWN*
- None

Raises ***AttributeError*** – if *direction* is OUTPUT.

Direction – defines the direction of a digital pin

class digitalio.DigitalInOut.**Direction**
Enum-like class to define which direction the digital values are going.

INPUT
Read digital data in

OUTPUT
Write digital data out

`DriveMode` – defines the drive mode of a digital pin

class digitalio.**DriveMode**
Enum-like class to define the drive mode used when outputting digital values.

PUSH_PULL
Output both high and low digital values

OPEN_DRAIN
Output low digital values but go into high z for digital high. This is useful for i2c and other protocols that share a digital line.

`Pull` – defines the pull of a digital input pin

class digitalio.**Pull**
Enum-like class to define the pull value, if any, used while reading digital values in.

UP
When the input line isn't being driven the pull up can pull the state of the line high so it reads as true.

DOWN
When the input line isn't being driven the pull down can pull the state of the line low so it reads as false.

All classes change hardware state and should be deinitialized when they are no longer needed if the program continues after use. To do so, either call `deinit()` or use a context manager. See *Lifetime and ContextManagers* for more info.

For example:

```
import digitalio
from board import *

pin = digitalio.DigitalInOut(D13)
print(pin.value)
```

This example will initialize the the device, read *value* and then *deinit()* the hardware.

Here is blinky:

```
import digitalio
from board import *
import time

led = digitalio.DigitalInOut(D13)
led.direction = digitalio.Direction.OUTPUT
while True:
    led.value = True
    time.sleep(0.1)
    led.value = False
    time.sleep(0.1)
```

gamepad — Button handling

GamePad – Scan buttons for presses

Usage:

```
import board
import digitalio
import gamepad
import time

B_UP = 1 << 0
B_DOWN = 1 << 1


pad = gamepad.GamePad(
    digitalio.DigitalInOut(board.D10),
    digitalio.DigitalInOut(board.D11),
)

y = 0
while True:
    buttons = pad.get_pressed()
    if buttons & B_UP:
        y -= 1
        print(y)
    elif buttons & B_DOWN:
        y += 1
        print(y)
    time.sleep(0.1)
    while buttons:
        # Wait for all buttons to be released.
        buttons = pad.get_pressed()
        time.sleep(0.1)
```

class gamepad.**GamePad**(*b1*[, *b2*[, *b3*[, *b4*[, *b5*[, *b6*[, *b7*[, *b8*]]]]]]])

Initializes button scanning routines.

The b1-b8 parameters are DigitalInOut objects, which immediately get switched to input with a pull-up, and then scanned regularly for button presses. The order is the same as the order of bits returned by the get_pressed function. You can re-initialize it with different keys, then the new object will replace the previous one.

The basic feature required here is the ability to poll the keys at regular intervals (so that de-bouncing is consistent) and fast enough (so that we don't miss short button presses) while at the same time letting the user code run normally, call blocking functions and wait on delays.

They button presses are accumulated, until the get_pressed method is called, at which point the button state is cleared, and the new button presses start to be recorded.

get_pressed()

Get the status of buttons pressed since the last call and clear it.

Returns an 8-bit number, with bits that correspond to buttons, which have been pressed (or held down) since the last call to this function set to 1, and the remaining bits set to 0. Then it clears the button state, so that new button presses (or buttons that are held down) can be recorded for the next call.

deinit()

Disable button scanning.

`math` — mathematical functions

The *math* module provides some basic mathematical functions for working with floating-point numbers.

Constants

`math.`**`e`**
base of the natural logarithm

`math.`**`pi`**
the ratio of a circle's circumference to its diameter

Functions

`math.`**`acos`**(*x*)
Return the inverse cosine of `x`.

`math.`**`asin`**(*x*)
Return the inverse sine of `x`.

`math.`**`atan`**(*x*)
Return the inverse tangent of `x`.

`math.`**`atan2`**(*y*, *x*)
Return the principal value of the inverse tangent of `y/x`.

`math.`**`ceil`**(*x*)
Return an integer, being `x` rounded towards positive infinity.

`math.`**`copysign`**(*x*, *y*)
Return `x` with the sign of `y`.

`math.`**`cos`**(*x*)
Return the cosine of `x`.

`math.`**`degrees`**(*x*)
Return radians `x` converted to degrees.

`math.`**`exp`**(*x*)
Return the exponential of `x`.

`math.`**`fabs`**(*x*)
Return the absolute value of `x`.

`math.`**`floor`**(*x*)
Return an integer, being `x` rounded towards negative infinity.

`math.`**`fmod`**(*x*, *y*)
Return the remainder of `x/y`.

`math.`**`frexp`**(*x*)
Decomposes a floating-point number into its mantissa and exponent. The returned value is the tuple `(m, e)` such that `x == m * 2**e` exactly. If `x == 0` then the function returns `(0.0, 0)`, otherwise the relation `0.5 <= abs(m) < 1` holds.

`math.`**`isfinite`**(*x*)
Return `True` if `x` is finite.

math.**isinf**(*x*)
Return `True` if `x` is infinite.

math.**isnan**(*x*)
Return `True` if `x` is not-a-number

math.**ldexp**(*x*, *exp*)
Return `x * (2**exp)`.

math.**modf**(*x*)
Return a tuple of two floats, being the fractional and integral parts of `x`. Both return values have the same sign as `x`.

math.**pow**(*x*, *y*)
Returns `x` to the power of `y`.

math.**radians**(*x*)
Return degrees `x` converted to radians.

math.**sin**(*x*)
Return the sine of `x`.

math.**sqrt**(*x*)
Returns the square root of `x`.

math.**tan**(*x*)
Return the tangent of `x`.

math.**trunc**(*x*)
Return an integer, being `x` rounded towards 0.

microcontroller — Pin references and cpu functionality

The *microcontroller* module defines the pins from the perspective of the microcontroller. See *board* for board-specific pin mappings.

Libraries

Pin — Pin reference

Identifies an IO pin on the microcontroller.

class microcontroller.**Pin**
Identifies an IO pin on the microcontroller. They are fixed by the hardware so they cannot be constructed on demand. Instead, use *board* or *microcontroller.pin* to reference the desired pin.

Processor — Microcontroller CPU information and control

Get information about the microcontroller CPU and control it.

Usage:

```
import microcontroller
print(microcontroller.cpu.frequency)
print(microcontroller.cpu.temperature)
```

class microcontroller.**Processor**
You cannot create an instance of *microcontroller.Processor*. Use *microcontroller.cpu* to access the sole instance available.

frequency
The CPU operating frequency as an *int*, in Hertz. (read-only)

temperature
The on-chip temperature, in Celsius, as a float. (read-only)

Is None if the temperature is not available.

uid
The unique id (aka serial number) of the chip as a *bytearray*. (read-only)

RunMode – run state of the microcontroller

class microcontroller.**RunMode**
Enum-like class to define the run mode of the microcontroller and CircuitPython.

NORMAL
Run CircuitPython as normal.

SAFE_MODE
Run CircuitPython in safe mode. User code will not be run and the file system will be writeable over USB.

BOOTLOADER
Run the bootloader.

microcontroller.**cpu**
CPU information and control, such as `cpu.temperature` and `cpu.frequency` (clock frequency). This object is the sole instance of *microcontroller.Processor*.

microcontroller.**delay_us**(*delay*)
Dedicated delay method used for very short delays. **Do not** do long delays because this stops all other functions from completing. Think of this as an empty `while` loop that runs for the specified `(delay)` time. If you have other code or peripherals (e.g audio recording) that require specific timing or processing while you are waiting, explore a different avenue such as using *time.sleep()*.

microcontroller.**disable_interrupts**()
Disable all interrupts. Be very careful, this can stall everything.

microcontroller.**enable_interrupts**()
Enable the interrupts that were enabled at the last disable.

microcontroller.**on_next_reset**(*run_mode*)
Configure the run mode used the next time the microcontroller is reset but not powered down.

Parameters **run_mode** (RunMode) – The next run mode

microcontroller.**reset**()
Reset the microcontroller. After reset, the microcontroller will enter the run mode last set by *on_next_reset*.

> **Warning:** This may result in file system corruption when connected to a host computer. Be very careful when calling this! Make sure the device "Safely removed" on Windows or "ejected" on Mac OSX and Linux.

microcontroller.**nvm**
Available non-volatile memory. This object is the sole instance of *nvm.ByteArray* when available or None otherwise.

microcontroller.pin — Microcontroller pin names

References to pins as named by the microcontroller

multiterminal — Manage additional terminal sources

The *multiterminal* module allows you to configure an additional serial terminal source. Incoming characters are accepted from both the internal serial connection and the optional secondary connection.

multiterminal.**get_secondary_terminal**()
Returns the current secondary terminal.

multiterminal.**set_secondary_terminal**(*stream*)
Read additional input from the given stream and write out back to it. This doesn't replace the core stream (usually UART or native USB) but is mixed in instead.

Parameters **stream** (*stream*) – secondary stream

multiterminal.**clear_secondary_terminal**()
Clears the secondary terminal.

multiterminal.**schedule_secondary_terminal_read**(*socket*)
In cases where the underlying OS is doing task scheduling, this notifies the OS when more data is available on the socket to read. This is useful as a callback for lwip sockets.

neopixel_write — Low-level neopixel implementation

The *neopixel_write* module contains a helper method to write out bytes in the 800khz neopixel protocol.

For example, to turn off a single neopixel (like the status pixel on Express boards.)

```
import board
import neopixel_write
import digitalio

pin = digitalio.DigitalInOut(board.NEOPIXEL)
pin.direction = digitalio.Direction.OUTPUT
pixel_off = bytearray([0, 0, 0])
neopixel_write.neopixel_write(pin, pixel_off)
```

neopixel_write.**neopixel_write**(*digitalinout*, *buf*)
Write buf out on the given DigitalInOut.

Parameters

- **gpio** (DigitalInOut) – the DigitalInOut to output with
- **buf** (bytearray) – The bytes to clock out. No assumption is made about color order

nvm — Non-volatile memory

The *nvm* module allows you to store whatever raw bytes you wish in a reserved section non-volatile memory.

Libraries

ByteArray – Presents a stretch of non-volatile memory as a bytearray.

Non-volatile memory is available as a byte array that persists over reloads and power cycles. Each assignment causes an erase and write cycle so its recommended to assign all values to change at once.

Usage:

```
import microcontroller
microcontroller.nvm[0:3] = b"\xcc\x10\x00"
```

class nvm.**ByteArray**
Not currently dynamically supported. Access the sole instance through *microcontroller.nvm*.

__len__()
Return the length. This is used by (*len*)

os — functions that an OS normally provides

The *os* module is a strict subset of the CPython os module. So, code written in CircuitPython will work in CPython but not necessarily the other way around.

os.**uname**()
Returns a named tuple of operating specific and CircuitPython port specific information.

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

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

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

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

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

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

os.**rename**(*old_path*, *new_path*)
Rename a file.

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

os.**statvfs**(*path*)
Get the status of a fileystem.

Returns a tuple with the filesystem information in the following order:

- `f_bsize` – file system block size
- `f_frsize` – fragment size
- `f_blocks` – size of fs in f_frsize units
- `f_bfree` – number of free blocks
- `f_bavail` – number of free blocks for unpriviliged users
- `f_files` – number of inodes
- `f_ffree` – number of free inodes
- `f_favail` – number of free inodes for unpriviliged users
- `f_flag` – mount flags
- `f_namemax` – maximum filename length

Parameters related to inodes: `f_files`, `f_ffree`, `f_avail` and the `f_flags` parameter may return `0` as they can be unavailable in a port-specific implementation.

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

`os.`**`urandom`**`(`*size*`)`
Returns a string of *size* random bytes based on a hardware True Random Number Generator. When not available, it will raise a NotImplementedError.

`os.`**`sep`**
Separator used to delineate path components such as folder and file names.

`pulseio` — Support for pulse based protocols

The *`pulseio`* module contains classes to provide access to basic pulse IO.

Libraries

`PulseIn` – Read a series of pulse durations

PulseIn is used to measure a series of active and idle pulses. This is commonly used in infrared receivers and low cost temperature sensors (DHT). The pulsed signal consists of timed active and idle periods. Unlike PWM, there is no set duration for active and idle pairs.

`class` `pulseio.`**`PulseIn`**`(`*pin*, *maxlen=2*, ***, *idle_state=False*`)`
Create a PulseIn object associated with the given pin. The object acts as a read-only sequence of pulse lengths with a given max length. When it is active, new pulse lengths are added to the end of the list. When there is no more room (len() == *`maxlen`*) the oldest pulse length is removed to make room.

Parameters

- **`pin`** (`Pin`) – Pin to read pulses from.
- **`maxlen`** (`int`) – Maximum number of pulse durations to store at once
- **`idle_state`** (`bool`) – Idle state of the pin. At start and after *`resume`* the first recorded pulse will the opposite state from idle.

Read a short series of pulses:

```
import pulseio
import board

pulses = pulseio.PulseIn(board.D7)

# Wait for an active pulse
while len(pulses) == 0:
    pass
# Pause while we do something with the pulses
pulses.pause()

# Print the pulses. pulses[0] is an active pulse unless the length
# reached max length and idle pulses are recorded.
print(pulses)

# Clear the rest
pulses.clear()

# Resume with an 80 microsecond active pulse
pulses.resume(80)
```

deinit()
Deinitialises the PulseIn and releases any hardware resources for reuse.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

pause()
Pause pulse capture

resume(*trigger_duration=0*)
Resumes pulse capture after an optional trigger pulse.

> **Warning:** Using trigger pulse with a device that drives both high and low signals risks a short. Make sure your device is open drain (only drives low) when using a trigger pulse. You most likely added a "pull-up" resistor to your circuit to do this.

Parameters **trigger_duration** (int) – trigger pulse duration in microseconds

clear()
Clears all captured pulses

popleft()
Removes and returns the oldest read pulse.

maxlen
The maximum length of the PulseIn. When len() is equal to maxlen, it is unclear which pulses are active and which are idle.

paused
True when pulse capture is paused as a result of *pause()* or an error during capture such as a signal that is too fast.

__len__()
Returns the current pulse length

This allows you to:

```
pulses = pulseio.PulseIn(pin)
print(len(pulses))
```

__get__(*index*)
Returns the value at the given index or values in slice.

This allows you to:

```
pulses = pulseio.PulseIn(pin)
print(pulses[0])
```

PulseOut – Output a pulse train

PulseOut is used to pulse PWM "carrier" output on and off. This is commonly used in infrared remotes. The pulsed signal consists of timed on and off periods. Unlike PWM, there is no set duration for on and off pairs.

class pulseio.**PulseOut**(*carrier*)
Create a PulseOut object associated with the given PWM out experience.

Parameters **carrier** (PWMOut) – PWMOut that is set to output on the desired pin.

Send a short series of pulses:

```
import array
import pulseio
import board

pwm = pulseio.PWMOut(board.D13, duty_cycle=2 ** 15)
pulse = pulseio.PulseOut(pwm)
#                            on    off     on     off    on
pulses = array.array('H', [65000, 1000, 65000, 65000, 1000])
pulse.send(pulses)

# Modify the array of pulses.
pulses[0] = 200
pulse.send(pulses)
```

deinit()
Deinitialises the PulseOut and releases any hardware resources for reuse.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

send(*pulses*)
Pulse alternating on and off durations in microseconds starting with on. pulses must be an *array.array* with data type 'H' for unsigned halfword (two bytes).

This method waits until the whole array of pulses has been sent and ensures the signal is off afterwards.

Parameters **pulses** (array.array) – pulse durations in microseconds

`PWMOut` – Output a Pulse Width Modulated signal

PWMOut can be used to output a PWM signal on a given pin.

class pulseio.**PWMOut**(*pin*, ***, *duty_cycle=0*, *frequency=500*, *variable_frequency=False*)

Create a PWM object associated with the given pin. This allows you to write PWM signals out on the given pin. Frequency is fixed after init unless `variable_frequency` is True.

Note: When `variable_frequency` is True, further PWM outputs may be limited because it may take more internal resources to be flexible. So, when outputting both fixed and flexible frequency signals construct the fixed outputs first.

Parameters

- **pin** (`Pin`) – The pin to output to
- **duty_cycle** (`int`) – The fraction of each pulse which is high. 16-bit
- **frequency** (`int`) – The target frequency in Hertz (32-bit)
- **variable_frequency** (`bool`) – True if the frequency will change over time

Simple LED fade:

```
import pulseio
import board

pwm = pulseio.PWMOut(board.D13)      # output on D13
pwm.duty_cycle = 2 ** 15             # Cycles the pin with 50% duty cycle (half of␣
↪2 ** 16) at the default 500hz
```

PWM at specific frequency (servos and motors):

```
import pulseio
import board

pwm = pulseio.PWMOut(board.D13, frequency=50)
pwm.duty_cycle = 2 ** 15                  # Cycles the pin with 50% duty cycle␣
↪(half of 2 ** 16) at 50hz
```

Variable frequency (usually tones):

```
import pulseio
import board
import time

pwm = pulseio.PWMOut(board.D13, duty_cycle=2 ** 15, frequency=440, variable_
↪frequency=True)
time.sleep(0.2)
pwm.frequency = 880
time.sleep(0.1)
```

deinit()

Deinitialises the PWMOut and releases any hardware resources for reuse.

__enter__()

No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

duty_cycle
16 bit value that dictates how much of one cycle is high (1) versus low (0). 0xffff will always be high, 0 will always be low and 0x7fff will be half high and then half low.

frequency
32 bit value that dictates the PWM frequency in Hertz (cycles per second). Only writeable when constructed with `variable_frequency=True`.

Warning: This module is not available in some SAMD21 builds. See the *Support Matrix* for more info.

All classes change hardware state and should be deinitialized when they are no longer needed if the program continues after use. To do so, either call `deinit()` or use a context manager. See *Lifetime and ContextManagers* for more info.

For example:

```
import pulseio
import time
from board import *

pwm = pulseio.PWMOut(D13)
pwm.duty_cycle = 2 ** 15
time.sleep(0.1)
```

This example will initialize the the device, set *duty_cycle*, and then sleep 0.1 seconds. CircuitPython will automatically turn off the PWM when it resets all hardware after program completion. Use `deinit()` or a `with` statement to do it yourself.

random — psuedo-random numbers and choices

The *random* module is a strict subset of the CPython `random` module. So, code written in CircuitPython will work in CPython but not necessarily the other way around.

Like its CPython cousin, CircuitPython's random seeds itself on first use with a true random from os.urandom() when available or the uptime otherwise. Once seeded, it will be deterministic, which is why its bad for cryptography.

Warning: Numbers from this module are not cryptographically strong! Use bytes from *os.urandom* directly for true randomness.

random.**seed**(*seed*)
Sets the starting seed of the random number generation. Further calls to *random* will return deterministic results afterwards.

random.**getrandbits**(*k*)
Returns an integer with *k* random bits.

random.**randrange**(*stop*)
random.**randrange**(*start*, *stop*, *step=1*)
Returns a randomly selected integer from `range(start, stop, step)`.

random.**randint**(*a*, *b*)
Returns a randomly selected integer between a and b inclusive. Equivalent to `randrange(a, b + 1, 1)`

random.**choice**(*seq*)
Returns a randomly selected element from the given sequence. Raises IndexError when the sequence is empty.

random.**random**()
Returns a random float between 0 and 1.0.

random.**uniform**(*a*, *b*)
Returns a random float between a and b. It may or may not be inclusive depending on float rounding.

rotaryio — Support for reading rotation sensors

The *rotaryio* module contains classes to read different rotation encoding schemes. See Wikipedia's Rotary Encoder page for more background.

Libraries

IncrementalEncoder – Track the relative position of an incremental encoder

IncrementalEncoder determines the relative rotational position based on two series of pulses.

class rotaryio.**IncrementalEncoder**(*pin_a*, *pin_b*)
Create an IncrementalEncoder object associated with the given pins. It tracks the positional state of an incremental rotary encoder (also known as a quadrature encoder.) Position is relative to the position when the object is contructed.

Parameters

- **pin_a** (Pin) – First pin to read pulses from.
- **pin_b** (Pin) – Second pin to read pulses from.

For example:

```
import rotaryio
import time
from board import *

enc = rotaryio.IncrementalEncoder(D1, D2)
last_position = None
while True:
    position = enc.position
    if last_position == None or position != last_position:
        print(position)
    last_position = position
```

deinit()
Deinitializes the IncrementalEncoder and releases any hardware resources for reuse.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

position
The current position in terms of pulses. The number of pulses per rotation is defined by the specific hardware.

Warning: This module is not available in some SAMD21 (aka M0) builds. See the *Support Matrix* for more info.

All classes change hardware state and should be deinitialized when they are no longer needed if the program continues after use. To do so, either call `deinit()` or use a context manager. See *Lifetime and ContextManagers* for more info.

`rtc` — Real Time Clock

The *rtc* module provides support for a Real Time Clock. It also backs the *time.time()* and *time.localtime()* functions using the onboard RTC if present.

Libraries

`RTC` — Real Time Clock

class rtc.**RTC**
This class represents the onboard Real Time Clock. It is a singleton and will always return the same instance.

datetime
The date and time of the RTC.

calibration
The RTC calibration value. A positive value speeds up the clock and a negative value slows it down. Range and value is hardware specific, but one step is often approx. 1 ppm.

rtc.**set_time_source**(*rtc*)
Sets the rtc time source used by time.localtime(). The default is *rtc.RTC()*.

Example usage:

```
import rtc
import time

class RTC(object):
    @property
    def datetime(self):
        return time.struct_time((2018, 3, 17, 21, 1, 47, 0, 0, 0))

r = RTC()
rtc.set_time_source(r)
```

`storage` — storage management

The *storage* provides storage management functionality such as mounting and unmounting which is typically handled by the operating system hosting Python. CircuitPython does not have an OS, so this module provides this functionality directly.

storage.**mount**(*filesystem*, *mount_path*, *, *readonly=False*)
Mounts the given filesystem object at the given path.

This is the CircuitPython analog to the UNIX `mount` command.

`storage.`**`umount`**(*mount*)
Unmounts the given filesystem object or if *mount* is a path, then unmount the filesystem mounted at that location.

This is the CircuitPython analog to the UNIX `umount` command.

`storage.`**`remount`**(*mount_path*, *readonly=False*)
Remounts the given path with new parameters.

`storage.`**`getmount`**(*mount_path*)
Retrieves the mount object associated with the mount path

`storage.`**`erase_filesystem`**()
Erase and re-create the `CIRCUITPY` filesystem.

On boards that present USB-visible `CIRCUITPY` drive (e.g., SAMD21 and SAMD51), then call *microcontroller.reset()* to restart CircuitPython and have the host computer remount CIRCUITPY.

This function can be called from the REPL when `CIRCUITPY` has become corrupted.

> **Warning:** All the data on `CIRCUITPY` will be lost, and CircuitPython will restart on certain boards.

class `storage.`**`VfsFat`**(*block_device*)
Create a new VfsFat filesystem around the given block device.

Parameters **`block_device`** – Block device the the filesystem lives on

`label`
The filesystem label, up to 11 case-insensitive bytes. Note that this property can only be set when the device is writable by the microcontroller.

`mkfs`()
Format the block device, deleting any data that may have been there

`open`(*path*, *mode*)
Like builtin `open()`

`ilistdir`([*path*])
Return an iterator whose values describe files and folders within `path`

`mkdir`(*path*)
Like *os.mkdir*

`rmdir`(*path*)
Like *os.rmdir*

`stat`(*path*)
Like *os.stat*

`statvfs`(*path*)
Like *os.statvfs*

`mount`(*readonly*, *mkfs*)
Don't call this directly, call *storage.mount*.

`umount`()
Don't call this directly, call *storage.umount*.

`struct` — manipulation of c-style data

This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation: struct.

Supported size/byte order prefixes: @, <, >, !.

Supported format codes: *b*, *B*, *h*, *H*, *i*, *I*, *l*, *L*, *q*, *Q*, *s*, *P*, *f*, *d* (the latter 2 depending on the floating-point support).

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

`struct.`**`pack`**(*fmt*, *v1*, *v2*, ...)
Pack the values v1, v2, ... according to the format string fmt. The return value is a bytes object encoding the values.

`struct.`**`pack_into`**(*fmt*, *buffer*, *offset*, *v1*, *v2*, ...)
Pack the values v1, v2, ... according to the format string fmt into a buffer starting at offset. offset may be negative to count from the end of buffer.

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

`struct.`**`unpack_from`**(*fmt*, *data*, *offset*)
Unpack from the data starting at offset according to the format string fmt. offset may be negative to count from the end of buffer. The return value is a tuple of the unpacked values.

`supervisor` — Supervisor settings

The *supervisor* module. (TODO: expand description)

Libraries

`Runtime` — Supervisor Runtime information

Get current status of runtime objects.

Usage:

```
import supervisor
if supervisor.runtime.serial_connected:
    print("Hello World!")
```

class supervisor.**Runtime**
You cannot create an instance of *supervisor.Runtime*. Use *supervisor.runtime* to access the sole instance available.

`runtime.`**`serial_connected`**
Returns the USB serial communication status (read-only).

`runtime.`**`serial_bytes_available`**
Returns the whether any bytes are available to read on the USB serial input. Allows for polling to see whether to call the built-in input() or wait. (read-only)

Note: SAMD: Will return `True` if the USB serial connection has been established at any point. Will not reset if USB is disconnected but power remains (e.g. battery connected)

Feather52 (nRF52832): Currently returns `True` regardless of USB connection status.

`supervisor.`**`runtime`**
Runtime information, such as *`runtime.serial_connected`* (USB serial connection status). This object is the sole instance of *`supervisor.Runtime`*.

`supervisor.`**`enable_autoreload`**`()`
Enable autoreload based on USB file write activity.

`supervisor.`**`disable_autoreload`**`()`
Disable autoreload based on USB file write activity until *`enable_autoreload`* is called.

`supervisor.`**`set_rgb_status_brightness`**`()`
Set brightness of status neopixel from 0-255 *`set_rgb_status_brightness`* is called.

`supervisor.`**`reload`**`()`
Reload the main Python code and run it (equivalent to hitting Ctrl-D at the REPL).

`time` — time and timing related functions

The *`time`* module is a strict subset of the CPython `time` module. So, code written in MicroPython will work in CPython but not necessarily the other way around.

`time.`**`monotonic`**`()`
Returns an always increasing value of time with an unknown reference point. Only use it to compare against other values from *`monotonic`*.

Returns the current monotonic time

Return type *float*

`time.`**`sleep`**`(`*seconds*`)`
Sleep for a given number of seconds.

Parameters **`seconds`** (`float`) – the time to sleep in fractional seconds

`class` `time.`**`struct_time`**`(`*(tm_year, tm_mon, tm_mday, tm_hour, tm_min, tm_sec, tm_wday, tm_yday, tm_isdst)*`)`
Structure used to capture a date and time. Note that it takes a tuple!

Parameters

- **`tm_year`** (`int`) – the year, 2017 for example
- **`tm_mon`** (`int`) – the month, range [1, 12]
- **`tm_mday`** (`int`) – the day of the month, range [1, 31]
- **`tm_hour`** (`int`) – the hour, range [0, 23]
- **`tm_min`** (`int`) – the minute, range [0, 59]
- **`tm_sec`** (`int`) – the second, range [0, 61]
- **`tm_wday`** (`int`) – the day of the week, range [0, 6], Monday is 0
- **`tm_yday`** (`int`) – the day of the year, range [1, 366], -1 indicates not known
- **`tm_isdst`** (`int`) – 1 when in daylight savings, 0 when not, -1 if unknown.

`time.`**`time`**`()`
Return the current time in seconds since since Jan 1, 1970.

Returns the current time

Return type *int*

time.**localtime**([*secs*])
Convert a time expressed in seconds since Jan 1, 1970 to a struct_time in local time. If secs is not provided or None, the current time as returned by time() is used. The earliest date for which it can generate a time is Jan 1, 2000.

Returns the current time

Return type *time.struct_time*

time.**mktime**(*t*)
This is the inverse function of localtime(). Its argument is the struct_time or full 9-tuple (since the dst flag is needed; use -1 as the dst flag if it is unknown) which expresses the time in local time, not UTC. The earliest date for which it can generate a time is Jan 1, 2000.

Returns seconds

Return type *int*

touchio — Touch related IO

The *touchio* module contains classes to provide access to touch IO typically accelerated by hardware on the onboard microcontroller.

Libraries

TouchIn – Read the state of a capacitive touch sensor

Usage:

```
import touchio
from board import *

touch = touchio.TouchIn(A1)
while True:
    if touch.value:
        print("touched!")
```

class touchio.**TouchIn**(*pin*)
Use the TouchIn on the given pin.

Parameters **pin** (Pin) – the pin to read from

deinit()
Deinitialises the TouchIn and releases any hardware resources for reuse.

__enter__()
No-op used by Context Managers.

__exit__()
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

value
Whether the touch pad is being touched or not. (read-only)

True when *raw_value* > *threshold*.

raw_value
The raw touch measurement as an *int*. (read-only)

threshold
Minimum *raw_value* needed to detect a touch (and for *value* to be True).

When the **TouchIn** object is created, an initial *raw_value* is read from the pin, and then *threshold* is set to be 100 + that value.

You can adjust *threshold* to make the pin more or less sensitive.

All classes change hardware state and should be deinitialized when they are no longer needed if the program continues after use. To do so, either call `deinit()` or use a context manager. See *Lifetime and ContextManagers* for more info.

For example:

```
import touchio
from board import *

touch_pin = touchio.TouchIn(D6)
print(touch_pin.value)
```

This example will initialize the the device, and print the *value*.

uheap — Heap size analysis

uheap.**info**(*object*)
Prints memory debugging info for the given object and returns the estimated size.

usb_hid — USB Human Interface Device

The *usb_hid* module allows you to output data as a HID device.

usb_hid.**devices**
Tuple of all active HID device interfaces.

Libraries

Device – HID Device

Usage:

```
import usb_hid

mouse = usb_hid.devices[0]

mouse.send_report()
```

class usb_hid.**Device**
Not currently dynamically supported.

send_report(*buf*)
Send a HID report.

usage_page
The usage page of the device as an *int*. Can be thought of a category. (read-only)

usage
The functionality of the device as an int. (read-only)

For example, Keyboard is 0x06 within the generic desktop usage page 0x01. Mouse is 0x02 within the same usage page.

ustack — Stack information and analysis

ustack.**max_stack_usage**()
Return the maximum excursion of the stack so far.

ustack.**stack_size**()
Return the size of the entire stack. Same as in micropython.mem_info(), but returns a value instead of just printing it.

ustack.**stack_usage**()
Return how much stack is currently in use. Same as micropython.stack_use(); duplicated here for convenience.

help() - Built-in method to provide helpful information

help(*object=None*)
Prints a help method about the given object. When `object` is none, prints general port information.

1.8.2 Supported Ports

Adafruit's CircuitPython currently has limited support with a focus on supporting the Atmel SAMD and ESP8266.

SAMD21x18

This port brings MicroPython to SAMD21x18 based development boards under the name CircuitPython. Supported boards include the Adafruit CircuitPlayground Express, Adafruit Feather M0 Express, Adafruit Metro M0 Express, Arduino Zero, Adafruit Feather M0 Basic and Adafruit M0 Bluefruit LE.

Pinout

All of the boards share the same core pin functionality but call pins by different names. The table below matches the pin order in the datasheet and omits the pins only available on the largest package because all supported boards use smaller version.

microcontroller.pin	*board*			
Datasheet	arduino_zero	circuitplayground_express	feather_m0_adalogger	feather_m0_basi
PA00		ACCELEROMETER_SDA		
PA01		ACCELEROMETER_SCL		
PA02	A0	A0 / SPEAKER	A0	A0
PA03				
PB08	A1	A7 / TX	A1	A1
PB09	A2	A6 / RX	A2	A2
PA04	A3	IR_PROXIMITY	A3	A3
PA05	A4	A1	A4	A4

Table 1 – continued from previous page

microcontroller.pin	*board*			
Datasheet	arduino_zero	circuitplayground_express	feather_m0_adalogger	feather_m0_basi
PA06	D8	A2	D8 / GREEN_LED	
PA07	D9	A3	D9	D9
PA08	D4	MICROPHONE_DO	D4 / SD_CS	
PA09	D3	TEMPERATURE / A9		
PA10	D1 / TX	MICROPHONE_SCK	D1 / TX	D1 / TX
PA11	D0 / RX	LIGHT / A8	D0 / RX	D0 / RX
PB10	MOSI		MOSI	MOSI
PB11	SCK		SCK	SCK
PA12	MISO	REMOTEIN / IR_RX	MISO	MISO
PA13		ACCELEROMETER_INTERRUPT		
PA14	D2	BUTTON_B / D5		
PA15	D5	SLIDE_SWITCH / D7	D5	D5
PA16	D11	MISO	D11	D11
PA17	D13	D13	D13 / RED_LED	D13
PA18	D10		D10	D10
PA19	D12		D12	D12
PA20	D6	MOSI	D6	D6
PA21	D7	SCK	D7 / SD_CD	
PA22	SDA		SDA	SDA
PA23	SCL	REMOTEOUT / IR_TX	SCL	SCL
PA24				
PA25				
PB22		FLASH_CS		
PB23		NEOPIXEL / D8		
PA27				
PA28		BUTTON_A / D4		
PA29				
PA30		SPEAKER_ENABLE		
PA31				
PB02	A5	A5 / SDA	A5	A5
PB03		A4 / SCL		

Here is a table about which pins can do what in CircuitPython terms. However, just because something is listed, doesn't mean it will always work. Existing use of other pins and functionality will impact your ability to use a pin for your desired purpose. For example, only certain combinations of SPI pins will work because they use shared hardware internally.

microcontroller.pin	*analogio*		*audioio*	*bitbangio*
Datasheet	AnalogIn	AnalogOut	AudioOut	I2C
PA00				**Yes**
PA01				**Yes**
PA02	**Yes**	**Yes**	**Yes**	**Yes**
PA03	**Yes**			**Yes**
PB08	**Yes**			**Yes**
PB09	**Yes**			**Yes**
PA04	**Yes**			**Yes**
PA05	**Yes**			**Yes**
PA06	**Yes**			**Yes**

microcontroller.pin	*analogio*		*audioio*	*bitbangio*
Datasheet	AnalogIn	AnalogOut	AudioOut	I2C
PA07	**Yes**			**Yes**
PA08	**Yes**			**Yes**
PA09	**Yes**			**Yes**
PA10	**Yes**			**Yes**
PA11	**Yes**			**Yes**
PB10				**Yes**
PB11				**Yes**
PA12				**Yes**
PA13				**Yes**
PA14				**Yes**
PA15				**Yes**
PA16				**Yes**
PA17				**Yes**
PA18				**Yes**
PA19				**Yes**
PA20				**Yes**
PA21				**Yes**
PA22				**Yes**
PA23				**Yes**
PA24				
PA25				
PB22				**Yes**
PB23				**Yes**
PA27				**Yes**
PA28				**Yes**
PA29				**Yes**
PA30				**Yes**
PA31				**Yes**
PB02	**Yes**			**Yes**
PB03	**Yes**			**Yes**

Setup

An ARM compiler is required for the build, along with the associated binary utilities. On Ubuntu, these can be installed as follows:

```
sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa
sudo apt-get install gcc-arm-embedded
```

On Arch Linux the compiler is available for via the package `arm-none-eabi-gcc`.

For other systems, the GNU Arm Embedded Toolchain may be available in binary form.

The latest available package from team-gcc-arm-embedded is used to produce the binaries shipped by AdaFruit. Other compiler versions, particularly older ones, may not work properly. In particular, the `gcc-arm-none-eabi` package in Debian Stretch is too old.

The compiler can be changed using the `CROSS_COMPILE` variable when invoking `make`.

Building

Before building the firmware for a given board, there are two additional steps. These commands should be executed from the root directory of the repository (`circuitpython/`).

1. There are various submodules that reside in different repositories. In order to have these submodules locally, you must pull them into your clone, using:

```
git submodule update --init --recursive
```

2. The MicroPython cross-compiler must be built; it will be used to pre-compile some of the built-in scripts to bytecode. The cross-compiler is built and run on the host machine, using:

```
make -C mpy-cross
```

Build commands are run from the `circuitpython/ports/atmel-samd` directory.

To build for the Arduino Zero:

```
make
```

To build for other boards you must change it by setting `BOARD`. For example:

```
make BOARD=feather_m0_basic
```

Board names are the directory names in the boards folder.

Deploying

Arduino Bootloader

If your board has an existing Arduino bootloader on it then you can use bossac to flash MicroPython. First, activate the bootloader. On Adafruit Feathers you can double click the reset button and the #13 will fade in and out. Finally, run bossac:

tools/bossac_osx -e -w -v -b -R build-feather_m0_basic/firmware.bin

No Bootloader via GDB

This method works for loading MicroPython onto the Arduino Zero via the programming port rather than the native USB port.

Note: These instructions are tested on Mac OSX and will vary for different platforms.

openocd -f ~/Library/Arduino15/packages/arduino/hardware/samd/1.6.6/variants/arduino_zero/openocd_scripts/arduino_zero.cfg

In another terminal from `micropython/atmel-samd`:

arm-none-eabi-gdb build-arduino_zero/firmware.elf (gdb) tar ext :3333 . . . (gdb) load . . . (gdb) monitor reset init . . . (gdb) continue

Connecting

Serial

All boards are currently configured to work over USB rather than UART. To connect to it from OSX do something like this:

```
screen /dev/tty.usbmodem142422 115200
```

You may not see a prompt immediately because it doesn't know you connected. To get one either hit enter to get >>> or do CTRL-B to get the full header.

Mass storage

All boards will also show up as a mass storage device. Make sure to eject it before resetting or disconnecting the board.

Port Specific modules

`samd` — SAMD implementation settings

Libraries

`Clock` — Clock reference

Identifies a clock on the microcontroller.

class samd.**Clock**

Identifies a clock on the microcontroller. They are fixed by the hardware so they cannot be constructed on demand. Instead, use *samd.clock* to reference the desired clock.

enabled
Is the clock enabled? (read-only)

parent
Clock parent. (read-only)

frequency
Clock frequency. (read-only)

calibration
Clock calibration. Not all clocks can be calibrated.

`samd.clock` — samd clock names

References to clocks as named by the microcontroller

MicroPython port to ESP8266

This is an experimental port of MicroPython for the WiFi modules based on Espressif ESP8266 chip.

WARNING: The port is experimental and many APIs are subject to change.

Supported features include:

- REPL (Python prompt) over UART0.

- Garbage collector, exceptions.
- Unicode support.
- Builtin modules: gc, array, collections, io, struct, sys, esp, network, many more.
- Arbitrary-precision long integers and 30-bit precision floats.
- WiFi support.
- Sockets using modlwip.
- GPIO and bit-banging I2C, SPI support.
- 1-Wire and WS2812 (aka Neopixel) protocols support.
- Internal filesystem using the flash.
- WebREPL over WiFi from a browser (clients at https://github.com/micropython/webrepl).
- Modules for HTTP, MQTT, many other formats and protocols via https://github.com/micropython/micropython-lib .

Work-in-progress documentation is available at http://docs.micropython.org/en/latest/esp8266/ .

Build instructions

The tool chain required for the build is the OpenSource ESP SDK, which can be found at https://github.com/pfalcon/esp-open-sdk. Clone this repository and run `make` in its directory to build and install the SDK locally. Make sure to add toolchain bin directory to your PATH. Read esp-open-sdk's README for additional important information on toolchain setup.

Add the external dependencies to the MicroPython repository checkout:

```
$ git submodule update --init
```

See the README in the repository root for more information about external dependencies.

The MicroPython cross-compiler must be built to pre-compile some of the built-in scripts to bytecode. This can be done using:

```
$ make -C mpy-cross
```

Then, to build MicroPython for the ESP8266, just run:

```
$ cd esp8266
$ make axtls
$ make
```

This will produce binary images in the `build/` subdirectory. If you install MicroPython to your module for the first time, or after installing any other firmware, you should erase flash completely:

```
esptool.py --port /dev/ttyXXX erase_flash
```

Erase flash also as a troubleshooting measure, if a module doesn't behave as expected.

To flash MicroPython image to your ESP8266, use:

```
$ make deploy
```

This will use the `esptool.py` script to download the images. You must have your ESP module in the bootloader mode, and connected to a serial port on your PC. The default serial port is `/dev/ttyACM0`, flash mode is `qio` and flash size is `detect` (auto-detect based on Flash ID). To specify other values, use, eg (note that flash size is in megabits):

```
$ make PORT=/dev/ttyUSB0 FLASH_MODE=qio FLASH_SIZE=32m deploy
```

The image produced is `build/firmware-combined.bin`, to be flashed at 0x00000.

512KB FlashROM version

The normal build described above requires modules with at least 1MB of FlashROM onboard. There's a special configuration for 512KB modules, which can be built with `make 512k`. This configuration is highly limited, lacks filesystem support, WebREPL, and has many other features disabled. It's mostly suitable for advanced users who are interested to fine-tune options to achieve a required setup. If you are an end user, please consider using a module with at least 1MB of FlashROM.

First start

Serial prompt

You can access the REPL (Python prompt) over UART (the same as used for programming).

- Baudrate: 115200

WiFi

Initially, the device configures itself as a WiFi access point (AP).

- ESSID: MicroPython-xxxxxx (x's are replaced with part of the MAC address).
- Password: micropythoN (note the upper-case N).
- IP address of the board: 192.168.4.1.
- DHCP-server is activated.

WebREPL

Python prompt over WiFi, connecting through a browser.

- Hosted at http://micropython.org/webrepl.
- GitHub repository https://github.com/micropython/webrepl.

Please follow the instructions there.

Documentation

More detailed documentation and instructions can be found at http://docs.micropython.org/en/latest/esp8266/ , which includes Quick Reference, Tutorial, General Information related to ESP8266 port, and to MicroPython in general.

Troubleshooting

While the port is in beta, it's known to be generally stable. If you experience strange bootloops, crashes, lockups, here's a list to check against:

- You didn't erase flash before programming MicroPython firmware.
- Firmware can be occasionally flashed incorrectly. Just retry. Recent esptool.py versions have –verify option.

- Power supply you use doesn't provide enough power for ESP8266 or isn't stable enough.
- A module/flash may be defective (not unheard of for cheap modules).

Please consult dedicated ESP8266 forums/resources for hardware-related problems.

Additional information may be available by the documentation links above.

CircuitPython Port To The Nordic Semiconductor nRF52 Series

This is a port of CircuitPython to the Nordic Semiconductor nRF52 series of chips.

Supported Features

- UART
- SPI
- LEDs
- Pins
- ADC
- I2C
- PWM
- Temperature
- RTC (Real Time Counter. Low-Power counter)
- BLE support including:
  - Peripheral role
  - Scanner role
  - *REPL over Bluetooth LE* (optionally using WebBluetooth)
  - ubluepy: Bluetooth LE module for CircuitPython
  - 1 non-connectable advertiser while in connection

Tested Hardware

- nRF52832
  - PCA10040
  - Adafruit Feather nRF52
- nRF52840
  - PCA10056

Compile and Flash

Prerequisite steps for building the nrf port:

```
git clone <URL>.git circuitpython
cd circuitpython
git submodule update --init
make -C mpy-cross
```

By default, the feather52 (nRF52832) is used as compile target. To build and flash issue the following command inside the ports/nrf/ folder:

```
make
make flash
```

Alternatively the target board could be defined:

```
 make BOARD=pca10056
 make flash
```

Compile and Flash with Bluetooth Stack

First prepare the bluetooth folder by downloading Bluetooth LE stacks and headers:

```
 ./drivers/bluetooth/download_ble_stack.sh
```

If the Bluetooth stacks has been downloaded, compile the target with the following command:

```
make BOARD=pca10040 SD=s132
```

The **make sd** will trigger a flash of the bluetooth stack before that application is flashed. Note that **make sd** will perform a full erase of the chip, which could cause 3rd party bootloaders to also be wiped.

```
make BOARD=pca10040 SD=s132 sd
```

Note: further tuning of features to include in bluetooth or even setting up the device to use REPL over Bluetooth can be configured in the `bluetooth_conf.h`.

Target Boards and Make Flags

Target Board (BOARD) | Bluetooth Stack (SD) | Bluetooth Support | Flash Util ——————|——————-|——————|——————- pca10040 | s132 | Peripheral and Scanner | *Segger* feather52 | s132 | Peripheral and Scanner | *UART DFU* pca10056 | s140 | Peripheral and Scanner | *Segger*

Segger Targets

Install the necessary tools to flash and debug using Segger:

JLink Download

nrfjprog linux-32bit Download

nrfjprog linux-64bit Download

nrfjprog osx Download

nrfjprog win32 Download

note: On Linux it might be required to link SEGGER's `libjlinkarm.so` inside nrfjprog's folder.

DFU Targets

```
sudo apt-get install build-essential libffi-dev pkg-config gcc-arm-none-eabi git␣
↪python python-pip
git clone https://github.com/adafruit/Adafruit_nRF52_Arduino.git
cd Adafruit_nRF52_Arduino/tools/nrfutil-0.5.2/
sudo pip install -r requirements.txt
sudo python setup.py install
```

make flash and **make sd** will not work with DFU targets. Hence, **dfu-gen** and **dfu-flash** must be used instead.

- dfu-gen: Generates a Firmware zip to be used by the DFU flash application.
- dfu-flash: Triggers the DFU flash application to upload the firmware from the generated Firmware zip file.

Example on how to generate and flash feather52 target:

```
make BOARD=feather52 SD=s132
make BOARD=feather52 SD=s132 dfu-gen
make BOARD=feather52 SD=s132 dfu-flash
```

Bluetooth LE REPL

The port also implements a BLE REPL driver. This feature is disabled by default, as it will deactivate the UART REPL when activated. As some of the nRF devices only have one UART, using the BLE REPL free's the UART instance such that it can be used as a general UART peripheral not bound to REPL.

The configuration can be enabled by editing the `bluetooth_conf.h` and set `MICROPY_PY_BLE_NUS` to 1.

When enabled you have different options to test it:

- NUS Console for Linux (recommended)
- WebBluetooth REPL (experimental)

Other:

- nRF UART application for IPhone/Android

WebBluetooth mode can also be configured by editing `bluetooth_conf.h` and set `BLUETOOTH_WEBBLUETOOTH_REPL` to 1. This will alternate advertisement between Eddystone URL and regular connectable advertisement. The Eddystone URL will point the phone or PC to download WebBluetooth REPL (experimental), which subsequently can be used to connect to the Bluetooth REPL from the PC or Phone browser.

1.8.3 Troubleshooting

From time to time, an error occurs when working with CircuitPython. Here are a variety of errors that can happen, what they mean and how to fix them.

File system issues

If your host computer starts complaining that your CIRCUITPY drive is corrupted or files cannot be overwritten or deleted, then you will have to erase it completely. When CircuitPython restarts it will create a fresh empty CIRCUITPY filesystem.

This often happens on Windows when the CIRCUITPY disk is not safely ejected before being reset by the button or being disconnected from USB. This can also happen on Linux and Mac OSX but its less likely.

Caution: To erase and re-create CIRCUITPY (for example, to correct a corrupted filesystem), follow one of the procedures below. It's important to note that **any files stored on the CIRCUITPY drive will be erased**.

For boards with CIRCUITPY stored on a separate SPI flash chip, such as Feather M0 Express, Metro M0 Express and Circuit Playground Express:

1. Download the appropriate flash .erase uf2 from the Adafruit_SPIFlash repo.
2. Double-click the reset button.
3. Copy the appropriate .uf2 to the xxxBOOT drive.
4. The on-board NeoPixel will turn blue, indicating the erase has started.
5. After about 15 seconds, the NexoPixel will start flashing green. If it flashes red, the erase failed.
6. Double-click again and load the appropriate CircuitPython .uf2.

For boards without SPI flash, such as Feather M0 Proto, Gemma M0 and, Trinket M0:

1. Download the appropriate erase .uf2 from the Learn repo.
2. Double-click the reset button.
3. Copy the appropriate .uf2 to the xxxBOOT drive.
4. The boot LED will start pulsing again, and the xxxBOOT drive will appear again.
5. Load the appropriate CircuitPython .uf2.

ValueError: Incompatible .mpy file.

This error occurs when importing a module that is stored as a mpy binary file (rather than a py text file) that was generated by a different version of CircuitPython than the one its being loaded into. Most versions are compatible but, rarely they aren't. In particular, the mpy binary format changed between CircuitPython versions 1.x and 2.x, and will change again between 2.x and 3.x.

So, for instance, if you just upgraded to CircuitPython 2.x from 1.x you'll need to download a newer version of the library that triggered the error on import. They are all available in the Adafruit bundle and the Community bundle. Make sure to download a version with 2.0.0 or higher in the filename.

1.8.4 Additional Adafruit Libraries and Drivers on GitHub

These are libraries and drivers available in separate GitHub repos. They are designed for use with CircuitPython and may or may not work with MicroPython.

Bundle

We provide a bundle of all our libraries to ease installation of drivers and their dependencies. The bundle is primarily geared to the Adafruit Express line of boards which feature a relatively large external flash. With Express boards, its easy to copy them all onto the filesystem. However, if you don't have enough space simply copy things over as they are needed.

The bundles are available on GitHub.

To install them:

1. Download and unzip the latest zip that's not a source zip.
2. Copy the `lib` folder to the `CIRCUITPY` or `MICROPYTHON`.

Foundational

These libraries provide critical functionality to many of the drivers below. It is recommended to always have them installed onto the CircuitPython file system in the `lib/` directory. Some drivers may not work without them.

Board-specific Helpers

These libraries tie lower-level libraries together to provide an easy, out-of-box experience for specific boards.

Helper Libraries

These libraries build on top of the low level APIs to simplify common tasks.

Blinky

Multi-color led drivers.

Displays

Drivers used to display information. Either pixel or segment based.

Real-time clocks

Chips that keep current calendar time with a backup battery. The current date and time is available through `datetime`.

Motion Sensors

Motion relating sensing including `acceleration`, `magnetic`, `gyro`, and `orientation`.

Environmental Sensors

Sense attributes of the environment including `temperature`, `relative_humidity`, `pressure`, equivalent carbon dioxide (`eco2` / `eCO2`), and total volatile organic compounds (`tvoc` / `TVOC`).

Light Sensors

These sensors detect light related attributes such as `color`, `light` (unit-less), and `lux` (light in SI lux).

Distance Sensors

These sensors measure the `distance` to another object and may also measure light level (`light` and `lux`).

Radio

These chips communicate to other's over radio.

IO Expansion

These provide functionality similar to *`analogio`*, *`digitalio`*, *`pulseio`*, and *`touchio`*.

Miscellaneous

1.8.5 Design Guide

This guide covers a variety of development practices for CircuitPython core and library APIs. These APIs are both built-into CircuitPython and those that are distributed on GitHub and in the Adafruit and Community bundles. Consistency with these practices ensures that beginners can learn a pattern once and apply it throughout the CircuitPython ecosystem.

Start libraries with the cookiecutter

Cookiecutter is a tool that lets you bootstrap a new repo based on another repo. We've made one here for CircuitPython libraries that include configs for Travis CI and ReadTheDocs along with a setup.py, license, code of conduct and readme.

Cookiecutter will provide a series of prompts relating to the library and then create a new directory with all of the files. See the CircuitPython cookiecutter README for more details.

Module Naming

Adafruit funded libraries should be under the adafruit organization and have the format `Adafruit_CircuitPython_<name>` and have a corresponding `adafruit_<name>` directory (aka package) or `adafruit_<name>.py` file (aka module).

If the name would normally have a space, such as "Thermal Printer", use an underscore instead ("Thermal_Printer"). This underscore will be used everywhere even when the separation between "adafruit" and "circuitpython" is done with a `-`. Use the underscore in the cookiecutter prompts.

Community created libraries should have the repo format `CircuitPython_<name>` and not have the `adafruit_` module or package prefix.

Both should have the CircuitPython repository topic on GitHub.

Lifetime and ContextManagers

A driver should be initialized and ready to use after construction. If the device requires deinitialization, then provide it through `deinit()` and also provide `__enter__` and `__exit__` to create a context manager usable with `with`.

For example, a user can then use `deinit()`:

```
import digitalio
import board

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

for i in range(10):
    led.value = True
    time.sleep(0.5)

    led.value = False
    time.sleep(0.5)
led.deinit()
```

This will deinit the underlying hardware at the end of the program as long as no exceptions occur.

Alternatively, using a `with` statement ensures that the hardware is deinitialized:

```
import digitalio
import board

with digitalio.DigitalInOut(board.D13) as led:
    led.direction = digitalio.Direction.OUTPUT

    for i in range(10):
        led.value = True
        time.sleep(0.5)

        led.value = False
        time.sleep(0.5)
```

Python's `with` statement ensures that the deinit code is run regardless of whether the code within the with statement executes without exceptions.

For small programs like the examples this isn't a major concern because all user usable hardware is reset after programs are run or the REPL is run. However, for more complex programs that may use hardware intermittently and may also handle exceptions on their own, deinitializing the hardware using a with statement will ensure hardware isn't enabled longer than needed.

Verify your device

Whenever possible, make sure device you are talking to is the device you expect. If not, raise a RuntimeError. Beware that I2C addresses can be identical on different devices so read registers you know to make sure they match your expectation. Validating this upfront will help catch mistakes.

Getters/Setters

When designing a driver for a device, use properties for device state and use methods for sequences of abstract actions that the device performs. State is a property of the device as a whole that exists regardless of what the code is doing.

This includes things like temperature, time, sound, light and the state of a switch. For a more complete list see the sensor properties bullet below.

Another way to separate state from actions is that state is usually something the user can sense themselves by sight or feel for example. Actions are something the user can watch. The device does this and then this.

Making this separation clear to the user will help beginners understand when to use what.

Here is more info on properties from Python.

Design for compatibility with CPython

CircuitPython is aimed to be one's first experience with code. It will be the first step into the world of hardware and software. To ease one's exploration out from this first step, make sure that functionality shared with CPython shares the same API. It doesn't need to be the full API it can be a subset. However, do not add non-CPython APIs to the same modules. Instead, use separate non-CPython modules to add extra functionality. By distinguishing API boundaries at modules you increase the likelihood that incorrect expectations are found on import and not randomly during runtime.

When adding a new module for additional functionality related to a CPython module do NOT simply prefix it with u. This is not a large enough differentiation from CPython. This is the MicroPython convention and they use u* modules interchangeably with the CPython name. This is confusing. Instead, think up a new name that is related to the extra functionality you are adding.

For example, storage mounting and unmounting related functions were moved from `uos` into a new *`storage`* module. Terminal related functions were moved into *`multiterminal`*. These names better match their functionality and do not conflict with CPython names. Make sure to check that you don't conflict with CPython libraries too. That way we can port the API to CPython in the future.

Example

When adding extra functionality to CircuitPython to mimic what a normal operating system would do, either copy an existing CPython API (for example file writing) or create a separate module to achieve what you want. For example, mounting and unmount drives is not a part of CPython so it should be done in a module, such as a new `storage` module, that is only available in CircuitPython. That way when someone moves the code to CPython they know what parts need to be adapted.

Document inline

Whenever possible, document your code right next to the code that implements it. This makes it more likely to stay up to date with the implementation itself. Use Sphinx's automodule to format these all nicely in ReadTheDocs. The cookiecutter helps set these up.

Use Sphinx flavor rST for markup.

Lots of documentation is a good thing but it can take a lot of space. To minimize the space used on disk and on load, distribute the library as both .py and .mpy, MicroPython and CircuitPython's bytecode format that omits comments.

Module description

After the license comment:

```
"""
`<module name>` - <Short description>
================================================
<Longer description.>
"""
```

Class description

At the class level document what class does and how to initialize it:

```
class DS3231:
    """DS3231 real-time clock.

       :param ~busio.I2C i2c_bus: The I2C bus the DS3231 is connected to.
       :param int address: The I2C address of the device.
    """

    def __init__(self, i2c_bus, address=0x40):
        self._i2c = i2c_bus
```

Renders as:

class DS3231(*i2c_bus*, *address=64*)
DS3231 real-time clock.

Parameters

- **i2c_bus** (I2C) – The I2C bus the DS3231 is connected to.
- **address** (int) – The I2C address of the device.

Attributes

Attributes are state on objects. (See *Getters/Setters* above for more discussion about when to use them.) They can be defined internally in a number of different ways. Each approach is enumerated below with an explanation of where the comment goes.

Regardless of how the attribute is implemented, it should have a short description of what state it represents including the type, possible values and/or units. It should be marked as `(read-only)` or `(write-only)` at the end of the first line for attributes that are not both readable and writable.

Instance attributes

Comment comes from after the assignment:

```
def __init__(self, drive_mode):
    self.drive_mode = drive_mode
    """
    The pin drive mode. One of:

    - `digitalio.DriveMode.PUSH_PULL`
    - `digitalio.DriveMode.OPEN_DRAIN`
    """
```

Renders as:

drive_mode
The pin drive mode. One of:

- *digitalio.DriveMode.PUSH_PULL*
- *digitalio.DriveMode.OPEN_DRAIN*

Property description

Comment comes from the getter:

```
@property
def datetime(self):
    """The current date and time as a `time.struct_time`."""
    return self.datetime_register

@datetime.setter
def datetime(self, value):
    pass
```

Renders as:

datetime
The current date and time as a *time.struct_time*.

Read-only example:

```
@property
def temperature(self):
    """
    The current temperature in degrees Celsius. (read-only)

    The device may require calibration to get accurate readings.
    """
    return self._read(TEMPERATURE)
```

Renders as:

temperature
The current temperature in degrees Celsius. (read-only)

The device may require calibration to get accurate readings.

Data descriptor description

Comment is after the definition:

```
lost_power = i2c_bit.RWBit(0x0f, 7)
"""True if the device has lost power since the time was set."""
```

Renders as:

lost_power
True if the device has lost power since the time was set.

Method description

First line after the method definition:

```
def turn_right(self, degrees):
    """Turns the bot ``degrees`` right.

       :param float degrees: Degrees to turn right
    """
```

Renders as:

turn_right(*degrees*)
Turns the bot `degrees` right.

Parameters **degrees** (`float`) – Degrees to turn right

Use BusDevice

BusDevice is an awesome foundational library that manages talking on a shared I2C or SPI device for you. The devices manage locking which ensures that a transfer is done as a single unit despite CircuitPython internals and, in the future, other Python threads. For I2C, the device also manages the device address. The SPI device, manages baudrate settings, chip select line and extra post-transaction clock cycles.

I2C Example

```
from adafruit_bus_device import i2c_device

DEVICE_DEFAULT_I2C_ADDR = 0x42

class Widget:
    """A generic widget."""

    def __init__(self, i2c, address=DEVICE_DEFAULT_I2C_ADDR):
        self.i2c_device = i2c_device.I2CDevice(i2c, address)
        self.buf = bytearray(1)

    @property
    def register(self):
        """Widget's one register."""
        with self.i2c_device as i2c:
            i2c.writeto(b'0x00')
            i2c.readfrom_into(self.buf)
        return self.buf[0]
```

SPI Example

```
from adafruit_bus_device import spi_device

class SPIWidget:
    """A generic widget with a weird baudrate."""

    def __init__(self, spi, chip_select):
```

(continues on next page)

(continued from previous page)

```
        # chip_select is a pin reference such as board.D10.
        self.spi_device = spi_device.SPIDevice(spi, chip_select, baudrate=12345)
        self.buf = bytearray(1)

    @property
    def register(self):
        """Widget's one register."""
        with self.spi_device as spi:
            spi.write(b'0x00')
            i2c.readinto(self.buf)
        return self.buf[0]
```

Use composition

When writing a driver, take in objects that provide the functionality you need rather than taking their arguments and constructing them yourself or subclassing a parent class with functionality. This technique is known as composition and leads to code that is more flexible and testable than traditional inheritance.

See also:

Wikipedia has more information on "dependency inversion".

For example, if you are writing a driver for an I2C device, then take in an I2C object instead of the pins themselves. This allows the calling code to provide any object with the appropriate methods such as an I2C expansion board.

Another example is to expect a *DigitalInOut* for a pin to toggle instead of a *Pin* from *board*. Taking in the *Pin* object alone would limit the driver to pins on the actual microcontroller instead of pins provided by another driver such as an IO expander.

Lots of small modules

CircuitPython boards tend to have a small amount of internal flash and a small amount of ram but large amounts of external flash for the file system. So, create many small libraries that can be loaded as needed instead of one large file that does everything.

Speed second

Speed isn't as important as API clarity and code size. So, prefer simple APIs like properties for state even if it sacrifices a bit of speed.

Avoid allocations in drivers

Although Python doesn't require managing memory, its still a good practice for library writers to think about memory allocations. Avoid them in drivers if you can because you never know how much something will be called. Fewer allocations means less time spent cleaning up. So, where you can, prefer bytearray buffers that are created in `__init__` and used throughout the object with methods that read or write into the buffer instead of creating new objects. Unified hardware API classes such as *busio.SPI* are design to read and write to subsections of buffers.

Its ok to allocate an object to return to the user. Just beware of causing more than one allocation per call due to internal logic.

However, this is a memory tradeoff so do not do it for large or rarely used buffers.

Examples

struct.pack

Use *struct.pack_into* instead of *struct.pack*.

Sensor properties and units

The Adafruit Unified Sensor Driver Arduino library has a great list of measurements and their units. Use the same ones including the property name itself so that drivers can be used interchangeably when they have the same properties.

Property name	Python type	Units
acceleration	(float, float, float)	x, y, z meter per second per second
magnetic	(float, float, float)	x, y, z micro-Tesla (uT)
orientation	(float, float, float)	x, y, z degrees
gyro	(float, float, float)	x, y, z radians per second
temperature	float	degrees centigrade
eCO2	float	equivalent CO2 in ppm
TVOC	float	Total Volatile Organic Compounds in ppb
distance	float	centimeters
light	float	non-unit-specific light levels (should be monotonic but is not lux)
lux	float	SI lux
pressure	float	hectopascal (hPa)
relative_humidity	float	percent
current	float	milliamps (mA)
voltage	float	volts (V)
color	int	RGB, eight bits per channel (0xff0000 is red)
alarm	(time.struct, str)	Sample alarm time and string to characterize frequency such as "hourly"
datetime	time.struct	date and time
duty_cycle	int	16-bit PWM duty cycle (regardless of output resolution)
frequency	int	Hertz
value	bool	Digital logic
value	int	16-bit Analog value, unit-less

Adding native modules

The Python API for a new module should be defined and documented in `shared-bindings` and define an underlying C API. If the implementation is port-agnostic or relies on underlying APIs of another module, the code should live in `shared-module`. If it is port specific then it should live in `common-hal` within the port's folder. In either case, the file and folder structure should mimic the structure in `shared-bindings`.

To test your native modules or core enhancements, follow these Adafruit Learning Guides for building local firmware to flash onto your device(s):

SAMD21 - Build Firmware Learning Guide

ESP8266 - Build Firmware Learning Guide

MicroPython compatibility

Keeping compatibility with MicroPython isn't a high priority. It should be done when its not in conflict with any of the above goals.

We love CircuitPython and would love to see it come to more microcontroller platforms. With 3.0 we've reworked CircuitPython to make it easier than ever to add support. While there are some major differences between ports, this page covers the similarities that make CircuitPython what it is and how that core fits into a variety of microcontrollers.

1.8.6 Architecture

There are three core pieces to CircuitPython:

The first is the Python VM that the awesome MicroPython devs have created. These VMs are written to be portable so there is not much needed when moving to a different microcontroller, especially if it is ARM based.

The second is the infrastructure around those VMs which provides super basic operating system functionality such as initializing hardware, running USB, prepping file systems and automatically running user code on boot. In CircuitPython we've dubbed this component the supervisor because it monitors and facilitates the VMs which run user Python code. Porting involves the supervisor because many of the tasks it does while interfacing with the hardware. Once its going though, the REPL works and debugging can migrate to a Python based approach rather than C.

The third core piece is the plethora of low level APIs that CircuitPython provides as the foundation for higher level libraries including device drivers. These APIs are called from within the running VMs through the Python interfaces defined in `shared-bindings`. These bindings rely on the underlying `common_hal` C API to implement the functionality needed for the Python API. By splitting the two, we work to ensure standard functionality across which means that libraries and examples apply across ports with minimal changes.

1.8.7 Porting

Step 1: Getting building

The first step to porting to a new microcontroller is getting a build running. The primary goal of it should be to get `main.c` compiling with the assistance of the `supervisor/supervisor.mk` file. Port specific code should be isolated to the port's directory (in the top level until the `ports` directory is present). This includes the Makefile and any C library resources. Make sure these resources are compatible with the MIT License of the rest of the code!

Step 2: Init

Once your build is setup, the next step should be to get your clocks going as you expect from the supervisor. The supervisor calls `port_init` to allow for initialization at the beginning of main. This function also has the ability to request a safe mode state which prevents the supervisor from running user code while still allowing access to the REPL and other resources.

The core port initialization and reset methods are defined in `supervisor/port.c` and should be the first to be implemented. Its required that they be implemented in the `supervisor` directory within the port directory. That way, they are always in the expected place.

The supervisor also uses three linker variables, `_ezero`, `_estack` and `_ebss` to determine memory layout for stack overflow checking.

Step 3: REPL

Getting the REPL going is a huge step. It involves a bunch of initialization to be done correctly and is a good sign you are well on your porting way. To get the REPL going you must implement the functions and definitions from `supervisor/serial.h` with a corresponding `supervisor/serial.c` in the port directory. This involves sending and receiving characters over some sort of serial connection. It could be UART or USB for example.

1.8.8 Adding `*io` support to other ports

`digitalio` provides a well-defined, cross-port hardware abstraction layer built to support different devices and their drivers. It's backed by the Common HAL, a C api suitable for supporting different hardware in a similar manner. By sharing this C api, developers can support new hardware easily and cross-port functionality to the new hardware.

These instructions also apply to `analogio`, `busio`, `pulseio` and `touchio`. Most drivers depend on `analogio`, `digitalio` and `busio` so start with those.

File layout

Common HAL related files are found in these locations:

- `shared-bindings` Shared home for the Python <-> C bindings which includes inline RST documentation for the created interfaces. The common hal functions are defined in the .h files of the corresponding C files.
- `shared-modules` Shared home for C code built on the Common HAL and used by all ports. This code only uses `common_hal` methods defined in `shared-bindings`.
- `<port>/common-hal` Port-specific implementation of the Common HAL.

Each folder has the substructure of / and they should match 1:1. `__init__.c` is used for module globals that are not classes (similar to `__init__.py`).

Adding support

Modifying the build

The first step is to hook the `shared-bindings` into your build for the modules you wish to support. Here's an example of this step for the `atmel-samd/Makefile`:

```
SRC_BINDINGS = \
        board/__init__.c \
        microcontroller/__init__.c \
        microcontroller/Pin.c \
        analogio/__init__.c \
        analogio/AnalogIn.c \
        analogio/AnalogOut.c \
        digitalio/__init__.c \
        digitalio/DigitalInOut.c \
        pulseio/__init__.c \
        pulseio/PulseIn.c \
        pulseio/PulseOut.c \
        pulseio/PWMOut.c \
        busio/__init__.c \
        busio/I2C.c \
        busio/SPI.c \
        busio/UART.c \
```

(continues on next page)

(continued from previous page)

```
        neopixel_write/__init__.c \
        time/__init__.c \
        usb_hid/__init__.c \
        usb_hid/Device.c

SRC_BINDINGS_EXPANDED = $(addprefix shared-bindings/, $(SRC_BINDINGS)) \
                        $(addprefix common-hal/, $(SRC_BINDINGS))

# Add the resulting objects to the full list
OBJ += $(addprefix $(BUILD)/, $(SRC_BINDINGS_EXPANDED:.c=.o))
# Add the sources for QSTR generation
SRC_QSTR += $(SRC_C) $(SRC_BINDINGS_EXPANDED) $(STM_SRC_C)
```

The `Makefile` defines the modules to build and adds the sources to include the `shared-bindings` version and the `common-hal` version within the port specific directory. You may comment out certain subfolders to reduce the number of modules to add but don't comment out individual classes. It won't compile then.

Hooking the modules in

Built in modules are typically defined in `mpconfigport.h`. To add support you should have something like:

```
extern const struct _mp_obj_module_t microcontroller_module;
extern const struct _mp_obj_module_t analogio_module;
extern const struct _mp_obj_module_t digitalio_module;
extern const struct _mp_obj_module_t pulseio_module;
extern const struct _mp_obj_module_t busio_module;
extern const struct _mp_obj_module_t board_module;
extern const struct _mp_obj_module_t time_module;
extern const struct _mp_obj_module_t neopixel_write_module;

#define MICROPY_PORT_BUILTIN_MODULES \
    { MP_OBJ_NEW_QSTR(MP_QSTR_microcontroller), (mp_obj_t)&microcontroller_module }, \
    { MP_OBJ_NEW_QSTR(MP_QSTR_analogio), (mp_obj_t)&analogio_module }, \
    { MP_OBJ_NEW_QSTR(MP_QSTR_digitalio), (mp_obj_t)&digitalio_module }, \
    { MP_OBJ_NEW_QSTR(MP_QSTR_pulseio), (mp_obj_t)&pulseio_module }, \
    { MP_OBJ_NEW_QSTR(MP_QSTR_busio), (mp_obj_t)&busio_module }, \
    { MP_OBJ_NEW_QSTR(MP_QSTR_board), (mp_obj_t)&board_module }, \
    { MP_OBJ_NEW_QSTR(MP_QSTR_time), (mp_obj_t)&time_module }, \
    { MP_OBJ_NEW_QSTR(MP_QSTR_neopixel_write),(mp_obj_t)&neopixel_write_module } \
```

Implementing the Common HAL

At this point in the port, nothing will compile yet, because there's still work to be done to fix missing sources, compile issues, and link issues. I suggest start with a common-hal directory from another port that implements it such as `atmel-samd` or `esp8266`, deleting the function contents and stubbing out any return statements. Once that is done, you should be able to compile cleanly and import the modules, but nothing will work (though you are getting closer).

The last step is actually implementing each function in a port specific way. I can't help you with this. :-) If you have any questions how a Common HAL function should work then see the corresponding .h file in `shared-bindings`.

Testing

Woohoo! You are almost done. After you implement everything, lots of drivers and sample code should just work. There are a number of drivers and examples written for Adafruit's Feather ecosystem. Here are places to start:

- Adafruit repos with CircuitPython topic
- Adafruit driver bundle

1.8.9 MicroPython libraries

Python standard libraries and micro-libraries

These libraries are inherited from MicroPython. They are similar to the standard Python libraries with the same name or with the "u" prefix dropped. They implement a subset of or a variant of the corresponding standard Python library.

Warning: Though these MicroPython-based libraries are available in CircuitPython, their functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, new versions of these libraries will be created that are more compliant with the standard Python libraries. You may need to change your code later if you rely on any non-standard functionality they currently provide.

CircuitPython's goal long-term goalis that code written in CircuitPython using Python standard libraries will be runnable on CPython without changes.

Some libraries below are not enabled on CircuitPython builds with limited flash memory, usually on non-Express builds: `uerrno`, `ure`.

Some libraries are not currently enabled in any CircuitPython build, but may be in the future: `uio`, `ujson`, `uzlib`.

Some libraries are only enabled only WiFi-capable ports (ESP8266, nRF) because they are typically used for network software: `binascii`, `hashlib`, `uheapq`, `uselect`, `ussl`. Not all of these are enabled on all WiFi-capable ports.

Builtin functions and exceptions

All builtin functions and exceptions are described here. They are also available via `builtins` module.

Functions and types

Not all of these functions and types are turned on in all CircuitPython ports, for space reasons.

abs()

all()

any()

bin()

class bool

class bytearray

class bytes
See CPython documentation: *bytes*.

callable()

chr()

classmethod()

compile()

class complex

delattr(*obj*, *name*)
The argument *name* should be a string, and this function deletes the named attribute from the object given by *obj*.

class dict

dir()

divmod()

enumerate()

eval()

exec()

filter()

class float

class frozenset

frozenset() is not enabled on non-Express CircuitPython boards.

getattr()

globals()

hasattr()

hash()

hex()

id()

input()

class int

classmethod from_bytes(*bytes*, *byteorder*)
In CircuitPython, *byteorder* parameter must be positional (this is compatible with CPython).

to_bytes(*size*, *byteorder*)
In CircuitPython, *byteorder* parameter must be positional (this is compatible with CPython).

isinstance()

issubclass()

iter()

len()

class list

locals()

map()

max()

class memoryview

min()

next()

class object

oct()

open()

ord()

pow()

print()

property()

range()

repr()

reversed()

reversed() is not enabled on non-Express CircuitPython boards.

round()

class set

setattr()

class slice

The *slice* builtin is the type that slice objects have.

sorted()

staticmethod()

class str

sum()

super()

class tuple

type()

zip()

Exceptions

exception AssertionError

exception AttributeError

exception Exception

exception ImportError

exception IndexError

exception KeyboardInterrupt

exception KeyError

exception MemoryError

exception NameError

exception NotImplementedError

exception OSError
See CPython documentation: *OSError*. CircuitPython doesn't implement the `errno` attribute, instead use the standard way to access exception arguments: `exc.args[0]`.

exception RuntimeError

exception ReloadException
ReloadException is used internally to deal with soft restarts.

exception StopIteration

exception SyntaxError

exception SystemExit
See CPython documentation: `python:SystemExit`.

exception TypeError
See CPython documentation: `python:TypeError`.

exception ValueError

exception ZeroDivisionError

uheapq – heap queue algorithm

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `heapq`.

This module implements the heap queue algorithm.

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

Functions

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

`uheapq.`**heappop**(*heap*)
Pop the first item from the `heap`, and return it. Raises IndexError if heap is empty.

`uheapq.`**heapify**(*x*)
Convert the list `x` into a heap. This is an in-place operation.

`array` – arrays of numeric data

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `array`.

Supported format codes: `b`, `B`, `h`, `H`, `i`, `I`, `l`, `L`, `q`, `Q`, `f`, `d` (the latter 2 depending on the floating-point support).

Classes

class `array.`**`array`**(*typecode*[, *iterable*])

Create array with elements of given type. Initial contents of the array are given by an `iterable`. If it is not provided, an empty array is created.

`append`(*val*)

Append new element to the end of array, growing it.

`extend`(*iterable*)

Append new elements as contained in an iterable to the end of array, growing it.

`binascii` – binary/ASCII conversions

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `binascii`.

This module implements conversions between binary data and various encodings of it in ASCII form (in both directions).

Functions

`binascii.`**`hexlify`**(*data*[, *sep*])

Convert binary data to hexadecimal representation. Returns bytes string.

Difference to CPython

If additional argument, *sep* is supplied, it is used as a separator between hexadecimal values.

`binascii.`**`unhexlify`**(*data*)

Convert hexadecimal data to binary representation. Returns bytes string. (i.e. inverse of hexlify)

`binascii.`**`a2b_base64`**(*data*)

Decode base64-encoded data, ignoring invalid characters in the input. Conforms to RFC 2045 s.6.8. Returns a bytes object.

`binascii.`**`b2a_base64`**(*data*)

Encode binary data in base64 format, as in RFC 3548. Returns the encoded data followed by a newline character, as a bytes object.

`collections` – collection and container types

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `collections`.

This module implements advanced collection and container types to hold/accumulate various objects.

Classes

`collections`.**`namedtuple`**(*name*, *fields*)

This is factory function to create a new namedtuple type with a specific name and set of fields. A namedtuple is a subclass of tuple which allows to access its fields not just by numeric index, but also with an attribute access syntax using symbolic field names. Fields is a sequence of strings specifying field names. For compatibility with CPython it can also be a a string with space-separated field named (but this is less efficient). Example of use:

```
from collections import namedtuple

MyTuple = namedtuple("MyTuple", ("id", "name"))
t1 = MyTuple(1, "foo")
t2 = MyTuple(2, "bar")
print(t1.name)
assert t2.name == t2[1]
```

`collections`.**`OrderedDict`**(...)

`dict` type subclass which remembers and preserves the order of keys added. When ordered dict is iterated over, keys/items are returned in the order they were added:

```
from collections import OrderedDict

# To make benefit of ordered keys, OrderedDict should be initialized
# from sequence of (key, value) pairs.
d = OrderedDict([("z", 1), ("a", 2)])
# More items can be added as usual
d["w"] = 5
d["b"] = 3
for k, v in d.items():
    print(k, v)
```

Output:

```
z 1
a 2
w 5
b 3
```

gc – control the garbage collector

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `gc`.

Functions

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

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

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

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

Difference to CPython

This function is a MicroPython extension.

`gc.`**`mem_free`**`()`
Return the number of bytes of available heap RAM, or -1 if this amount is not known.

Difference to CPython

This function is a MicroPython extension.

`gc.`**`threshold`**`(`[*amount*]`)`
Set or query the additional GC allocation threshold. Normally, a collection is triggered only when a new allocation cannot be satisfied, i.e. on an out-of-memory (OOM) condition. If this function is called, in addition to OOM, a collection will be triggered each time after *amount* bytes have been allocated (in total, since the previous time such an amount of bytes have been allocated). *amount* is usually specified as less than the full heap size, with the intention to trigger a collection earlier than when the heap becomes exhausted, and in the hope that an early collection will prevent excessive memory fragmentation. This is a heuristic measure, the effect of which will vary from application to application, as well as the optimal value of the *amount* parameter.

Calling the function without argument will return the current value of the threshold. A value of -1 means a disabled allocation threshold.

Difference to CPython

This function is a a MicroPython extension. CPython has a similar function - `set_threshold()`, but due to different GC implementations, its signature and semantics are different.

`hashlib` – hashing algorithms

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `hashlib`.

This module implements binary data hashing algorithms. The exact inventory of available algorithms depends on a board. Among the algorithms which may be implemented:

- SHA256 - The current generation, modern hashing algorithm (of SHA2 series). It is suitable for cryptographically-secure purposes. Included in the MicroPython core and any board is recommended to provide this, unless it has particular code size constraints.
- SHA1 - A previous generation algorithm. Not recommended for new usages, but SHA1 is a part of number of Internet standards and existing applications, so boards targeting network connectivity and interoperatiability will try to provide this.
- MD5 - A legacy algorithm, not considered cryptographically secure. Only selected boards, targeting interoperatibility with legacy applications, will offer this.

Constructors

class hashlib.**sha256**([*data*])
Create an SHA256 hasher object and optionally feed `data` into it.

class hashlib.**sha1**([*data*])
Create an SHA1 hasher object and optionally feed `data` into it.

class hashlib.**md5**([*data*])
Create an MD5 hasher object and optionally feed `data` into it.

Methods

hash.**update**(*data*)
Feed more binary data into hash.

hash.**digest**()
Return hash for all data passed through hash, as a bytes object. After this method is called, more data cannot be fed into the hash any longer.

hash.**hexdigest**()
This method is NOT implemented. Use `binascii.hexlify(hash.digest())` to achieve a similar effect.

sys – system specific functions

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `sys`.

Functions

`sys.`**`exit`**(*retval=0*)
Terminate current program with a given exit code. Underlyingly, this function raise as `SystemExit` exception. If an argument is given, its value given as an argument to `SystemExit`.

`sys.`**`print_exception`**(*exc*, *file=sys.stdout*)
Print exception with a traceback to a file-like object *file* (or `sys.stdout` by default).

Difference to CPython

This is simplified version of a function which appears in the `traceback` module in CPython. Unlike `traceback.print_exception()`, this function takes just exception value instead of exception type, exception value, and traceback object; *file* argument should be positional; further arguments are not supported.

Constants

`sys.`**`argv`**
A mutable list of arguments the current program was started with.

`sys.`**`byteorder`**
The byte order of the system (`"little"` or `"big"`).

`sys.`**`implementation`**
Object with information about the current Python implementation. For CircuitPython, it has following attributes:

- *name* - string "circuitpython"
- *version* - tuple (major, minor, micro), e.g. (1, 7, 0)

This object is the recommended way to distinguish CircuitPython from other Python implementations (note that it still may not exist in the very minimal ports).

Difference to CPython

CPython mandates more attributes for this object, but the actual useful bare minimum is implemented in CircuitPython.

`sys.`**`maxsize`**
Maximum value which a native integer type can hold on the current platform, or maximum value representable

by CircuitPython integer type, if it's smaller than platform max value (that is the case for CircuitPython ports without long int support).

This attribute is useful for detecting "bitness" of a platform (32-bit vs 64-bit, etc.). It's recommended to not compare this attribute to some value directly, but instead count number of bits in it:

```
bits = 0
v = sys.maxsize
while v:
    bits += 1
    v >>= 1
if bits > 32:
    # 64-bit (or more) platform
    ...
else:
    # 32-bit (or less) platform
    # Note that on 32-bit platform, value of bits may be less than 32
    # (e.g. 31) due to peculiarities described above, so use "> 16",
    # "> 32", "> 64" style of comparisons.
```

`sys.`**`modules`**
Dictionary of loaded modules. On some ports, it may not include builtin modules.

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

`sys.`**`platform`**
The platform that CircuitPython is running on. For OS/RTOS ports, this is usually an identifier of the OS, e.g. `"linux"`. For baremetal ports it is an identifier of the chip on a board, e.g. `"MicroChip SAMD51"`. It thus can be used to distinguish one board from another. If you need to check whether your program runs on CircuitPython (vs other Python implementation), use *`sys.implementation`* instead.

`sys.`**`stderr`**
Standard error stream.

`sys.`**`stdin`**
Standard input stream.

`sys.`**`stdout`**
Standard output stream.

`sys.`**`version`**
Python language version that this implementation conforms to, as a string.

`sys.`**`version_info`**
Python language version that this implementation conforms to, as a tuple of ints.

`uerrno` – system error codes

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `errno`.

This module provides access to symbolic error codes for *OSError* exception.

Constants

EEXIST, EAGAIN, etc.
Error codes, based on ANSI C/POSIX standard. All error codes start with "E". Errors are usually accessible as `exc.args[0]` where `exc` is an instance of *OSError*. Usage example:

```
try:
    os.mkdir("my_dir")
except OSError as exc:
    if exc.args[0] == uerrno.EEXIST:
        print("Directory already exists")
```

uerrno.**errorcode**
Dictionary mapping numeric error codes to strings with symbolic error code (see above):

```
>>> print(uerrno.errorcode[uerrno.EEXIST])
EEXIST
```

uio – input/output streams

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `io`.

This module contains additional types of stream (file-like) objects and helper functions.

Conceptual hierarchy

Difference to CPython

Conceptual hierarchy of stream base classes is simplified in MicroPython, as described in this section.

(Abstract) base stream classes, which serve as a foundation for behavior of all the concrete classes, adhere to few dichotomies (pair-wise classifications) in CPython. In MicroPython, they are somewhat simplified and made implicit to achieve higher efficiencies and save resources.

An important dichotomy in CPython is unbuffered vs buffered streams. In MicroPython, all streams are currently unbuffered. This is because all modern OSes, and even many RTOSes and filesystem drivers already perform buffering on their side. Adding another layer of buffering is counter- productive (an issue known as "bufferbloat") and takes precious memory. Note that there still cases where buffering may be useful, so we may introduce optional buffering support at a later time.

But in CPython, another important dichotomy is tied with "bufferedness" - it's whether a stream may incur short read/writes or not. A short read is when a user asks e.g. 10 bytes from a stream, but gets less, similarly for writes. In

CPython, unbuffered streams are automatically short operation susceptible, while buffered are guarantee against them. The no short read/writes is an important trait, as it allows to develop more concise and efficient programs - something which is highly desirable for MicroPython. So, while MicroPython doesn't support buffered streams, it still provides for no-short-operations streams. Whether there will be short operations or not depends on each particular class' needs, but developers are strongly advised to favor no-short-operations behavior for the reasons stated above. For example, MicroPython sockets are guaranteed to avoid short read/writes. Actually, at this time, there is no example of a short-operations stream class in the core, and one would be a port-specific class, where such a need is governed by hardware peculiarities.

The no-short-operations behavior gets tricky in case of non-blocking streams, blocking vs non-blocking behavior being another CPython dichotomy, fully supported by MicroPython. Non-blocking streams never wait for data either to arrive or be written - they read/write whatever possible, or signal lack of data (or ability to write data). Clearly, this conflicts with "no-short-operations" policy, and indeed, a case of non-blocking buffered (and this no-short-ops) streams is convoluted in CPython - in some places, such combination is prohibited, in some it's undefined or just not documented, in some cases it raises verbose exceptions. The matter is much simpler in MicroPython: non-blocking stream are important for efficient asynchronous operations, so this property prevails on the "no-short-ops" one. So, while blocking streams will avoid short reads/writes whenever possible (the only case to get a short read is if end of file is reached, or in case of error (but errors don't return short data, but raise exceptions)), non-blocking streams may produce short data to avoid blocking the operation.

The final dichotomy is binary vs text streams. MicroPython of course supports these, but while in CPython text streams are inherently buffered, they aren't in MicroPython. (Indeed, that's one of the cases for which we may introduce buffering support.)

Note that for efficiency, MicroPython doesn't provide abstract base classes corresponding to the hierarchy above, and it's not possible to implement, or subclass, a stream class in pure Python.

Functions

`uio.`**`open`**(*name*, *mode='r'*, ***kwargs*)
Open a file. Builtin `open()` function is aliased to this function. All ports (which provide access to file system) are required to support `mode` parameter, but support for other arguments vary by port.

Classes

class `uio.`**`FileIO`**(...)
This is type of a file open in binary mode, e.g. using `open(name, "rb")`. You should not instantiate this class directly.

class `uio.`**`TextIOWrapper`**(...)
This is type of a file open in text mode, e.g. using `open(name, "rt")`. You should not instantiate this class directly.

class `uio.`**`StringIO`**([*string*])

class `uio.`**`BytesIO`**([*string*])
In-memory file-like objects for input/output. *`StringIO`* is used for text-mode I/O (similar to a normal file opened with "t" modifier). *`BytesIO`* is used for binary-mode I/O (similar to a normal file opened with "b" modifier). Initial contents of file-like objects can be specified with `string` parameter (should be normal string for *`StringIO`* or bytes object for *`BytesIO`*). All the usual file methods like `read()`, `write()`, `seek()`, `flush()`, `close()` are available on these objects, and additionally, a following method:

`getvalue`()
Get the current contents of the underlying buffer which holds data.

ujson – JSON encoding and decoding

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `json`.

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

Functions

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

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

ure – simple regular expressions

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `re`.

This module implements regular expression operations. Regular expression syntax supported is a subset of CPython `re` module (and actually is a subset of POSIX extended regular expressions).

Supported operators are:

`'.'` Match any character.

`'[]'` Match set of characters. Individual characters and ranges are supported.

`'^'`

`'$'`

`'?'`

`'*'`

`'+'`

`'??'`

`'*?'`

`'+?'`

`'|'`

`'()'` Grouping. Each group is capturing (a substring it captures can be accessed with *`match.group()`* method).

Counted repetitions (`{m,n}`), more advanced assertions, named groups, etc. are not supported.

Functions

`ure.`**`compile`**(*regex_str*)
Compile regular expression, return *regex* object.

`ure.`**`match`**(*regex_str*, *string*)
Compile *regex_str* and match against *string*. Match always happens from starting position in a string.

`ure.`**`search`**(*regex_str*, *string*)
Compile *regex_str* and search it in a *string*. Unlike *`match`*, this will search string for first position which matches regex (which still may be 0 if regex is anchored).

`ure.`**`DEBUG`**
Flag value, display debug information about compiled expression.

Regex objects

Compiled regular expression. Instances of this class are created using *`ure.compile()`*.

`regex.`**`match`**(*string*)
`regex.`**`search`**(*string*)
Similar to the module-level functions *`match()`* and *`search()`*. Using methods is (much) more efficient if the same regex is applied to multiple strings.

`regex.`**`split`**(*string*, *max_split=-1*)
Split a *string* using regex. If *max_split* is given, it specifies maximum number of splits to perform. Returns list of strings (there may be up to *max_split+1* elements if it's specified).

Match objects

Match objects as returned by *`match()`* and *`search()`* methods.

`match.`**`group`**([*index*])
Return matching (sub)string. *index* is 0 for entire match, 1 and above for each capturing group. Only numeric groups are supported.

`uselect` – wait for events on a set of streams

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `select`.

This module provides functions to efficiently wait for events on multiple streams (select streams which are ready for operations).

Functions

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

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

This function is provided by some MicroPython ports for compatibility and is not efficient. Usage of `Poll` is recommended instead.

class `Poll`

Methods

`poll.`**`register`**`(`*obj*[, *eventmask*]`)`
Register *obj* for polling. *eventmask* is logical OR of:

- `select.POLLIN` - data available for reading
- `select.POLLOUT` - more data can be written
- `select.POLLERR` - error occurred
- `select.POLLHUP` - end of stream/connection termination detected

eventmask defaults to `select.POLLIN | select.POLLOUT`.

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

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

`poll.`**`poll`**`(`[*timeout*]`)`
Wait for at least one of the registered objects to become ready. Returns list of (`obj`, `event`, ...) tuples, `event` element specifies which events happened with a stream and is a combination of `select.POLL*` constants described above. There may be other elements in tuple, depending on a platform and version, so don't assume that its size is 2. In case of timeout, an empty list is returned.

Timeout is in milliseconds.

Difference to CPython

Tuples returned may contain more than 2 elements as described above.

`poll.`**`ipoll`**`(`*timeout=-1*, *flags=0*`)`
Like *poll.poll()*, but instead returns an iterator which yields `callee-owned tuples`. This function provides efficient, allocation-free way to poll on streams.

If *flags* is 1, one-shot behavior for events is employed: streams for which events happened, event mask will be automatically reset (equivalent to `poll.modify(obj, 0)`), so new events for such a stream won't be processed until new mask is set with *poll.modify()*. This behavior is useful for asynchronous I/O schedulers.

Difference to CPython

This function is a MicroPython extension.

`usocket` – socket module

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `socket`.

This module provides access to the BSD socket interface.

Difference to CPython

For efficiency and consistency, socket objects in MicroPython implement a stream (file-like) interface directly. In CPython, you need to convert a socket to a file-like object using *`makefile()`* method. This method is still supported by MicroPython (but is a no-op), so where compatibility with CPython matters, be sure to use it.

Socket address format(s)

The native socket address format of the `usocket` module is an opaque data type returned by *`getaddrinfo`* function, which must be used to resolve textual address (including numeric addresses):

```
sockaddr = usocket.getaddrinfo('www.micropython.org', 80)[0][-1]
# You must use getaddrinfo() even for numeric addresses
sockaddr = usocket.getaddrinfo('127.0.0.1', 80)[0][-1]
# Now you can use that address
sock.connect(addr)
```

Using *`getaddrinfo`* is the most efficient (both in terms of memory and processing power) and portable way to work with addresses.

However, `socket` module (note the difference with native MicroPython `usocket` module described here) provides CPython-compatible way to specify addresses using tuples, as described below.

Summing up:

- Always use *`getaddrinfo`* when writing portable applications.
- Tuple addresses described below can be used as a shortcut for quick hacks and interactive use, if your port supports them.

Tuple address format for `socket` module:

- IPv4: *(ipv4_address, port)*, where *ipv4_address* is a string with dot-notation numeric IPv4 address, e.g. `"8.8.8.8"`, and *port* is and integer port number in the range 1-65535. Note the domain names are not accepted as *ipv4_address*, they should be resolved first using *`usocket.getaddrinfo()`*.

- IPv6: *(ipv6_address, port, flowinfo, scopeid)*, where *ipv6_address* is a string with colon-notation numeric IPv6 address, e.g. `"2001:db8::1"`, and *port* is an integer port number in the range 1-65535. *flowinfo* must be 0. *scopeid* is the interface scope identifier for link-local addresses. Note the domain names are not accepted as *ipv6_address*, they should be resolved first using *usocket.getaddrinfo()*.

Functions

usocket.**socket**(*af=AF_INET*, *type=SOCK_STREAM*, *proto=IPPROTO_TCP*)

Create a new socket using the given address family, socket type and protocol number. Note that specifying *proto* in most cases is not required (and not recommended, as some MicroPython ports may omit `IPPROTO_*` constants). Instead, *type* argument will select needed protocol automatically:

```
# Create STREAM TCP socket
socket(AF_INET, SOCK_STREAM)
# Create DGRAM UDP socket
socket(AF_INET, SOCK_DGRAM)
```

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

Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. The list of 5-tuples has following structure:

```
(family, type, proto, canonname, sockaddr)
```

The following example shows how to connect to a given url:

```
s = usocket.socket()
s.connect(usocket.getaddrinfo('www.micropython.org', 80)[0][-1])
```

Difference to CPython

CPython raises a `socket.gaierror` exception (*OSError* subclass) in case of error in this function. MicroPython doesn't have `socket.gaierror` and raises OSError directly. Note that error numbers of *getaddrinfo()* form a separate namespace and may not match error numbers from *uerrno* module. To distinguish *getaddrinfo()* errors, they are represented by negative numbers, whereas standard system errors are positive numbers (error numbers are accessible using `e.args[0]` property from an exception object). The use of negative values is a provisional detail which may change in the future.

---

usocket.**inet_ntop**(*af*, *bin_addr*)

Convert a binary network address *bin_addr* of the given address family *af* to a textual representation:

```
>>> usocket.inet_ntop(usocket.AF_INET, b"\x7f\0\0\1")
'127.0.0.1'
```

usocket.**inet_pton**(*af*, *txt_addr*)

Convert a textual network address *txt_addr* of the given address family *af* to a binary representation:

```
>>> usocket.inet_pton(usocket.AF_INET, "1.2.3.4")
b'\x01\x02\x03\x04'
```

Constants

usocket.**AF_INET**

usocket.**AF_INET6**
Address family types. Availability depends on a particular MicroPython port.

usocket.**SOCK_STREAM**
usocket.**SOCK_DGRAM**
Socket types.

usocket.**IPPROTO_UDP**
usocket.**IPPROTO_TCP**
IP protocol numbers. Availability depends on a particular MicroPython port. Note that you don't need to specify these in a call to *usocket.socket()*, because *SOCK_STREAM* socket type automatically selects *IPPROTO_TCP*, and *SOCK_DGRAM* - *IPPROTO_UDP*. Thus, the only real use of these constants is as an argument to *setsockopt()*.

usocket.SOL_*
Socket option levels (an argument to *setsockopt()*). The exact inventory depends on a MicroPython port.

usocket.SO_*
Socket options (an argument to *setsockopt()*). The exact inventory depends on a MicroPython port.

Constants specific to WiPy:

usocket.**IPPROTO_SEC**
Special protocol value to create SSL-compatible socket.

class socket

Methods

socket.**close**()
Mark the socket closed and release all resources. Once that happens, all future operations on the socket object will fail. The remote end will receive EOF indication if supported by protocol.

Sockets are automatically closed when they are garbage-collected, but it is recommended to *close()* them explicitly as soon you finished working with them.

socket.**bind**(*address*)
Bind the socket to *address*. The socket must not already be bound.

socket.**listen**([*backlog*])
Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it's lower, it will be set to 0); and specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

socket.**accept**()
Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

socket.**connect**(*address*)
Connect to a remote socket at *address*.

socket.**send**(*bytes*)
Send data to the socket. The socket must be connected to a remote socket. Returns number of bytes sent, which may be smaller than the length of data ("short write").

socket.**sendall**(*bytes*)
Send all data to the socket. The socket must be connected to a remote socket. Unlike *send()*, this method will try to send all of data, by sending data chunk by chunk consecutively.

The behavior of this method on non-blocking sockets is undefined. Due to this, on MicroPython, it's recommended to use *write()* method instead, which has the same "no short writes" policy for blocking sockets, and will return number of bytes sent on non-blocking sockets.

socket.**recv**(*bufsize*)
Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by bufsize.

socket.**sendto**(*bytes*, *address*)
Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*.

socket.**recvfrom**(*bufsize*)
Receive data from the socket. The return value is a pair *(bytes, address)* where *bytes* is a bytes object representing the data received and *address* is the address of the socket sending the data.

socket.**setsockopt**(*level*, *optname*, *value*)
Set the value of the given socket option. The needed symbolic constants are defined in the socket module (SO_* etc.). The *value* can be an integer or a bytes-like object representing a buffer.

socket.**settimeout**(*value*)
Note: Not every port supports this method, see below.

Set a timeout on blocking socket operations. The value argument can be a nonnegative floating point number expressing seconds, or None. If a non-zero value is given, subsequent socket operations will raise an *OSError* exception if the timeout period value has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If None is given, the socket is put in blocking mode.

Not every `MicroPython port` supports this method. A more portable and generic solution is to use *uselect.poll* object. This allows to wait on multiple objects at the same time (and not just on sockets, but on generic stream objects which support polling). Example:

```
# Instead of:
s.settimeout(1.0)  # time in seconds
s.read(10)  # may timeout

# Use:
poller = uselect.poll()
poller.register(s, uselect.POLLIN)
res = poller.poll(1000)  # time in milliseconds
if not res:
    # s is still not ready for input, i.e. operation timed out
```

Difference to CPython

CPython raises a `socket.timeout` exception in case of timeout, which is an *OSError* subclass. MicroPython raises an OSError directly instead. If you use `except OSError:` to catch the exception, your code will work both in MicroPython and CPython.

socket.**setblocking**(*flag*)
Set blocking or non-blocking mode of the socket: if flag is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain *settimeout()* calls:

- `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
- `sock.setblocking(False)` is equivalent to `sock.settimeout(0)`

`socket.`**`makefile`**(*mode='rb'*, *buffering=0*)

Return a file object associated with the socket. The exact returned type depends on the arguments given to makefile(). The support is limited to binary modes only ('rb', 'wb', and 'rwb'). CPython's arguments: *encoding*, *errors* and *newline* are not supported.

Difference to CPython

As MicroPython doesn't support buffered streams, values of *buffering* parameter is ignored and treated as if it was 0 (unbuffered).

Difference to CPython

Closing the file object returned by makefile() WILL close the original socket as well.

`socket.`**`read`**(*[size]*)

Read up to size bytes from the socket. Return a bytes object. If *size* is not given, it reads all data available from the socket until EOF; as such the method will not return until the socket is closed. This function tries to read as much data as requested (no "short reads"). This may be not possible with non-blocking socket though, and then less data will be returned.

`socket.`**`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. Just as *read()*, this method follows "no short reads" policy.

Return value: number of bytes read and stored into *buf*.

`socket.`**`readline`**()

Read a line, ending in a newline character.

Return value: the line read.

`socket.`**`write`**(*buf*)

Write the buffer of bytes to the socket. This function will try to write all data to a socket (no "short writes"). This may be not possible with a non-blocking socket though, and returned value will be less than the length of *buf*.

Return value: number of bytes written.

`exception` `usocket.`**`error`**

MicroPython does NOT have this exception.

Difference to CPython

CPython used to have a `socket.error` exception which is now deprecated, and is an alias of *OSError*. In MicroPython, use *OSError* directly.

---

ussl – SSL/TLS module

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `ssl`.

This module provides access to Transport Layer Security (previously and widely known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side.

Functions

`ussl.`**`wrap_socket`**(*sock*, *server_side=False*, *keyfile=None*, *certfile=None*, *cert_reqs=CERT_NONE*, *ca_certs=None*)

Takes a stream *sock* (usually usocket.socket instance of `SOCK_STREAM` type), and returns an instance of ssl.SSLSocket, which wraps the underlying stream in an SSL context. Returned object has the usual stream interface methods like `read()`, `write()`, etc. In MicroPython, the returned object does not expose socket interface and methods like `recv()`, `send()`. In particular, a server-side SSL socket should be created from a normal socket returned from *`accept()`* on a non-SSL listening server socket.

Depending on the underlying module implementation in a particular `MicroPython port`, some or all keyword arguments above may be not supported.

Warning: Some implementations of `ussl` module do NOT validate server certificates, which makes an SSL connection established prone to man-in-the-middle attacks.

Exceptions

`ssl.`**`SSLError`**

This exception does NOT exist. Instead its base class, OSError, is used.

Constants

`ussl.`**`CERT_NONE`**
`ussl.`**`CERT_OPTIONAL`**
`ussl.`**`CERT_REQUIRED`**

Supported values for *cert_reqs* parameter.

uzlib – zlib decompression

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply

more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements a subset of the corresponding `CPython` *module, as described below. For more information, refer to the original* `CPython` *documentation:* `zlib`.

This module allows to decompress binary data compressed with DEFLATE algorithm (commonly used in zlib library and gzip archiver). Compression is not yet implemented.

Functions

`uzlib.`**`decompress`**(*data*, *wbits=0*, *bufsize=0*)

Return decompressed *data* as bytes. *wbits* is DEFLATE dictionary window size used during compression (8-15, the dictionary size is power of 2 of that value). Additionally, if value is positive, *data* is assumed to be zlib stream (with zlib header). Otherwise, if it's negative, it's assumed to be raw DEFLATE stream. *bufsize* parameter is for compatibility with CPython and is ignored.

`class` `uzlib.`**`DecompIO`**(*stream*, *wbits=0*)

Create a stream wrapper which allows transparent decompression of compressed data in another *stream*. This allows to process compressed streams with data larger than available heap size. In addition to values described in *`decompress()`*, *wbits* may take values 24..31 (16 + 8..15), meaning that input stream has gzip header.

Difference to CPython

This class is MicroPython extension. It's included on provisional basis and may be changed considerably or removed in later versions.

Omitted functions in the `string` library

A few string operations are not enabled on CircuitPython M0 non-Express builds, due to limited flash memory: `string.center()`, `string.partition()`, `string.splitlines()`, `string.reversed()`.

CircuitPython/MicroPython-specific libraries

Functionality specific to the CircuitPython/MicroPython implementation is available in the following libraries. These libraries may change signficantly or be removed in future versions of CircuitPtyon.

`btree` – simple BTree database

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

The `btree` module implements a simple key-value database using external storage (disk files, or in general case, a random-access stream). Keys are stored sorted in the database, and besides efficient retrieval by a key value, a database also supports efficient ordered range scans (retrieval of values with the keys in a given range). On the application

interface side, BTree database work as close a possible to a way standard *dict* type works, one notable difference is that both keys and values must be *bytes* objects (so, if you want to store objects of other types, you need to serialize them to *bytes* first).

The module is based on the well-known BerkelyDB library, version 1.xx.

Example:

```
import btree

# First, we need to open a stream which holds a database
# This is usually a file, but can be in-memory database
# using uio.BytesIO, a raw flash partition, etc.
# Oftentimes, you want to create a database file if it doesn't
# exist and open if it exists. Idiom below takes care of this.
# DO NOT open database with "a+b" access mode.
try:
    f = open("mydb", "r+b")
except OSError:
    f = open("mydb", "w+b")

# Now open a database itself
db = btree.open(f)

# The keys you add will be sorted internally in the database
db[b"3"] = b"three"
db[b"1"] = b"one"
db[b"2"] = b"two"

# Assume that any changes are cached in memory unless
# explicitly flushed (or database closed). Flush database
# at the end of each "transaction".
db.flush()

# Prints b'two'
print(db[b"2"])

# Iterate over sorted keys in the database, starting from b"2"
# until the end of the database, returning only values.
# Mind that arguments passed to values() method are *key* values.
# Prints:
#   b'two'
#   b'three'
for word in db.values(b"2"):
    print(word)

del db[b"2"]

# No longer true, prints False
print(b"2" in db)

# Prints:
#  b"1"
#  b"3"
for key in db:
    print(key)

db.close()
```

(continues on next page)

(continued from previous page)

```
# Don't forget to close the underlying stream!
f.close()
```

Functions

btree.**open**(*stream*, *, *flags=0*, *pagesize=0*, *cachesize=0*, *minkeypage=0*)

Open a database from a random-access `stream` (like an open file). All other parameters are optional and keyword-only, and allow to tweak advanced parameters of the database operation (most users will not need them):

- *flags* - Currently unused.
- *pagesize* - Page size used for the nodes in BTree. Acceptable range is 512-65536. If 0, a port-specific default will be used, optimized for port's memory usage and/or performance.
- *cachesize* - Suggested memory cache size in bytes. For a board with enough memory using larger values may improve performance. Cache policy is as follows: entire cache is not allocated at once; instead, accessing a new page in database will allocate a memory buffer for it, until value specified by *cachesize* is reached. Then, these buffers will be managed using LRU (least recently used) policy. More buffers may still be allocated if needed (e.g., if a database contains big keys and/or values). Allocated cache buffers aren't reclaimed.
- *minkeypage* - Minimum number of keys to store per page. Default value of 0 equivalent to 2.

Returns a BTree object, which implements a dictionary protocol (set of methods), and some additional methods described below.

Methods

btree.**close**()

Close the database. It's mandatory to close the database at the end of processing, as some unwritten data may be still in the cache. Note that this does not close underlying stream with which the database was opened, it should be closed separately (which is also mandatory to make sure that data flushed from buffer to the underlying storage).

btree.**flush**()

Flush any data in cache to the underlying stream.

btree.**__getitem__**(*key*)
btree.**get**(*key*, *default=None*)
btree.**__setitem__**(*key*, *val*)
btree.**__delitem__**(*key*)
btree.**__contains__**(*key*)

Standard dictionary methods.

btree.**__iter__**()

A BTree object can be iterated over directly (similar to a dictionary) to get access to all keys in order.

btree.**keys**([*start_key*[, *end_key*[, *flags*]]])
btree.**values**([*start_key*[, *end_key*[, *flags*]]])
btree.**items**([*start_key*[, *end_key*[, *flags*]]])

These methods are similar to standard dictionary methods, but also can take optional parameters to iterate over a key sub-range, instead of the entire database. Note that for all 3 methods, *start_key* and *end_key* arguments represent key values. For example, `values()` method will iterate over values corresponding to they key range given. None values for *start_key* means "from the first key", no *end_key* or its value of None means "until the

end of database". By default, range is inclusive of *start_key* and exclusive of *end_key*, you can include *end_key* in iteration by passing *flags* of *btree.INCL*. You can iterate in descending key direction by passing *flags* of *btree.DESC*. The flags values can be ORed together.

Constants

btree.**INCL**
A flag for *keys()*, *values()*, *items()* methods to specify that scanning should be inclusive of the end key.

btree.**DESC**
A flag for *keys()*, *values()*, *items()* methods to specify that scanning should be in descending direction of keys.

framebuf — Frame buffer manipulation

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module provides a general frame buffer which can be used to create bitmap images, which can then be sent to a display.

class FrameBuffer

The FrameBuffer class provides a pixel buffer which can be drawn upon with pixels, lines, rectangles, text and even other FrameBuffer's. It is useful when generating output for displays.

For example:

```
import framebuf

# FrameBuffer needs 2 bytes for every RGB565 pixel
fbuf = FrameBuffer(bytearray(10 * 100 * 2), 10, 100, framebuf.RGB565)

fbuf.fill(0)
fbuf.text('MicroPython!', 0, 0, 0xffff)
fbuf.hline(0, 10, 96, 0xffff)
```

Constructors

class framebuf.**FrameBuffer**(*buffer*, *width*, *height*, *format*, *stride=width*)
Construct a FrameBuffer object. The parameters are:

- *buffer* is an object with a buffer protocol which must be large enough to contain every pixel defined by the width, height and format of the FrameBuffer.
- *width* is the width of the FrameBuffer in pixels
- *height* is the height of the FrameBuffer in pixels

- *format* specifies the type of pixel used in the FrameBuffer; permissible values are listed under Constants below. These set the number of bits used to encode a color value and the layout of these bits in *buffer*. Where a color value c is passed to a method, c is a small integer with an encoding that is dependent on the format of the FrameBuffer.
- *stride* is the number of pixels between each horizontal line of pixels in the FrameBuffer. This defaults to *width* but may need adjustments when implementing a FrameBuffer within another larger FrameBuffer or screen. The *buffer* size must accommodate an increased step size.

One must specify valid *buffer*, *width*, *height*, *format* and optionally *stride*. Invalid *buffer* size or dimensions may lead to unexpected errors.

Drawing primitive shapes

The following methods draw shapes onto the FrameBuffer.

`FrameBuffer.`**`fill`**(*c*)
Fill the entire FrameBuffer with the specified color.

`FrameBuffer.`**`pixel`**(*x*, *y*[, *c*])
If *c* is not given, get the color value of the specified pixel. If *c* is given, set the specified pixel to the given color.

`FrameBuffer.`**`hline`**(*x*, *y*, *w*, *c*)

`FrameBuffer.`**`vline`**(*x*, *y*, *h*, *c*)

`FrameBuffer.`**`line`**(*x1*, *y1*, *x2*, *y2*, *c*)
Draw a line from a set of coordinates using the given color and a thickness of 1 pixel. The `line` method draws the line up to a second set of coordinates whereas the `hline` and `vline` methods draw horizontal and vertical lines respectively up to a given length.

`FrameBuffer.`**`rect`**(*x*, *y*, *w*, *h*, *c*)

`FrameBuffer.`**`fill_rect`**(*x*, *y*, *w*, *h*, *c*)
Draw a rectangle at the given location, size and color. The `rect` method draws only a 1 pixel outline whereas the `fill_rect` method draws both the outline and interior.

Drawing text

`FrameBuffer.`**`text`**(*s*, *x*, *y*[, *c*])
Write text to the FrameBuffer using the the coordinates as the upper-left corner of the text. The color of the text can be defined by the optional argument but is otherwise a default value of 1. All characters have dimensions of 8x8 pixels and there is currently no way to change the font.

Other methods

`FrameBuffer.`**`scroll`**(*xstep*, *ystep*)
Shift the contents of the FrameBuffer by the given vector. This may leave a footprint of the previous colors in the FrameBuffer.

`FrameBuffer.`**`blit`**(*fbuf*, *x*, *y*[, *key*])
Draw another FrameBuffer on top of the current one at the given coordinates. If *key* is specified then it should be a color integer and the corresponding color will be considered transparent: all pixels with that color value will not be drawn.

This method works between FrameBuffer instances utilising different formats, but the resulting colors may be unexpected due to the mismatch in color formats.

Constants

framebuf.**MONO_VLSB**
Monochrome (1-bit) color format This defines a mapping where the bits in a byte are vertically mapped with bit 0 being nearest the top of the screen. Consequently each byte occupies 8 vertical pixels. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered at locations starting at the leftmost edge, 8 pixels lower.

framebuf.**MONO_HLSB**
Monochrome (1-bit) color format This defines a mapping where the bits in a byte are horizontally mapped. Each byte occupies 8 horizontal pixels with bit 0 being the leftmost. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered on the next row, one pixel lower.

framebuf.**MONO_HMSB**
Monochrome (1-bit) color format This defines a mapping where the bits in a byte are horizontally mapped. Each byte occupies 8 horizontal pixels with bit 7 being the leftmost. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered on the next row, one pixel lower.

framebuf.**RGB565**
Red Green Blue (16-bit, 5+6+5) color format

framebuf.**GS4_HMSB**
Grayscale (4-bit) color format

micropython – access and control MicroPython internals

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

Functions

micropython.**const**(*expr*)
Used to declare that the expression is a constant so that the compile can optimise it. The use of this function should be as follows:

```
from micropython import const

CONST_X = const(123)
CONST_Y = const(2 * CONST_X + 1)
```

Constants declared this way are still accessible as global variables from outside the module they are declared in. On the other hand, if a constant begins with an underscore then it is hidden, it is not available as a global variable, and does not take up any memory during execution.

This *const* function is recognised directly by the MicroPython parser and is provided as part of the *micropython* module mainly so that scripts can be written which run under both CPython and MicroPython, by following the above pattern.

micropython.**opt_level**([*level*])
If *level* is given then this function sets the optimisation level for subsequent compilation of scripts, and returns `None`. Otherwise it returns the current optimisation level.

micropython.**alloc_emergency_exception_buf**(*size*)
Allocate *size* bytes of RAM for the emergency exception buffer (a good size is around 100 bytes). The buffer is used to create exceptions in cases when normal RAM allocation would fail (eg within an interrupt handler) and therefore give useful traceback information in these situations.

A good way to use this function is to put it at the start of your main script (eg `boot.py` or `main.py`) and then the emergency exception buffer will be active for all the code following it.

micropython.**mem_info**([*verbose*])
Print information about currently used memory. If the *verbose* argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the amount of stack and heap used. In verbose mode it prints out the entire heap indicating which blocks are used and which are free.

micropython.**qstr_info**([*verbose*])
Print information about currently interned strings. If the *verbose* argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the number of interned strings and the amount of RAM they use. In verbose mode it prints out the names of all RAM-interned strings.

micropython.**stack_use**()
Return an integer representing the current amount of stack that is being used. The absolute value of this is not particularly useful, rather it should be used to compute differences in stack usage at different points.

micropython.**heap_lock**()

micropython.**heap_unlock**()
Lock or unlock the heap. When locked no memory allocation can occur and a *MemoryError* will be raised if any heap allocation is attempted.

These functions can be nested, ie *heap_lock()* can be called multiple times in a row and the lock-depth will increase, and then *heap_unlock()* must be called the same number of times to make the heap available again.

micropython.**kbd_intr**(*chr*)
Set the character that will raise a *KeyboardInterrupt* exception. By default this is set to 3 during script execution, corresponding to Ctrl-C. Passing -1 to this function will disable capture of Ctrl-C, and passing 3 will restore it.

This function can be used to prevent the capturing of Ctrl-C on the incoming stream of characters that is usually used for the REPL, in case that stream is used for other purposes.

micropython.**schedule**(*func*, *arg*)
Schedule the function *func* to be executed "very soon". The function is passed the value *arg* as its single argument. "Very soon" means that the MicroPython runtime will do its best to execute the function at the earliest possible time, given that it is also trying to be efficient, and that the following conditions hold:

- A scheduled function will never preempt another scheduled function.
- Scheduled functions are always executed "between opcodes" which means that all fundamental Python operations (such as appending to a list) are guaranteed to be atomic.
- A given port may define "critical regions" within which scheduled functions will never be executed. Functions may be scheduled within a critical region but they will not be executed until that region is exited. An example of a critical region is a preempting interrupt handler (an IRQ).

A use for this function is to schedule a callback from a preempting IRQ. Such an IRQ puts restrictions on the code that runs in the IRQ (for example the heap may be locked) and scheduling a function to call later will lift those restrictions.

There is a finite stack to hold the scheduled functions and *schedule* will raise a *RuntimeError* if the stack is full.

network — network configuration

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module provides network drivers and routing configuration. To use this module, a MicroPython variant/build with network capabilities must be installed. Network drivers for specific hardware are available within this module and are used to configure hardware network interface(s). Network services provided by configured interfaces are then available for use via the *usocket* module.

For example:

```
# connect/ show IP config a specific network interface
# see below for examples of specific drivers
import network
import utime
nic = network.Driver(...)
if not nic.isconnected():
    nic.connect()
    print("Waiting for connection...")
    while not nic.isconnected():
        utime.sleep(1)
print(nic.ifconfig())

# now use usocket as usual
import usocket as socket
addr = socket.getaddrinfo('micropython.org', 80)[0][-1]
s = socket.socket()
s.connect(addr)
s.send(b'GET / HTTP/1.1\r\nHost: micropython.org\r\n\r\n')
data = s.recv(1000)
s.close()
```

Common network adapter interface

This section describes an (implied) abstract base class for all network interface classes implemented by MicroPython ports <MicroPython port> for different hardware. This means that MicroPython does not actually provide AbstractNIC class, but any actual NIC class, as described in the following sections, implements methods as described here.

class network.**AbstractNIC**(*id=None*, ...)

Instantiate a network interface object. Parameters are network interface dependent. If there are more than one interface of the same type, the first parameter should be *id*.

network.**active**([*is_active*])
Activate ("up") or deactivate ("down") the network interface, if a boolean argument is passed. Otherwise, query current state if no argument is provided. Most other methods require an active interface (behavior of calling them on inactive interface is undefined).

network.**connect**([*service_id*, *key=None*, *, ...])
Connect the interface to a network. This method is optional, and available only for interfaces which are not "always connected". If no parameters are given, connect to the default (or the only) service. If a single parameter is given, it is the primary identifier of a service to connect to. It may be accompanied by a key (password) required to access said service. There can be further arbitrary keyword-only parameters, depending on the networking medium type and/or particular device. Parameters can be used to: a) specify alternative service identifer types; b) provide additional connection parameters. For various medium types, there are different sets of predefined/recommended parameters, among them:

- WiFi: *bssid* keyword to connect to a specific BSSID (MAC address)

network.**disconnect**()
Disconnect from network.

network.**isconnected**()
Returns `True` if connected to network, otherwise returns `False`.

network.**scan**(*, ...)
Scan for the available network services/connections. Returns a list of tuples with discovered service parameters. For various network media, there are different variants of predefined/ recommended tuple formats, among them:

- WiFi: (ssid, bssid, channel, RSSI, authmode, hidden). There may be further fields, specific to a particular device.

The function may accept additional keyword arguments to filter scan results (e.g. scan for a particular service, on a particular channel, for services of a particular set, etc.), and to affect scan duration and other parameters. Where possible, parameter names should match those in connect().

network.**status**()
Return detailed status of the interface, values are dependent on the network medium/technology.

network.**ifconfig**([*(ip*, *subnet*, *gateway*, *dns)*])
Get/set IP-level network interface parameters: IP address, subnet mask, gateway and DNS server. When called with no arguments, this method returns a 4-tuple with the above information. To set the above values, pass a 4-tuple with the required information. For example:

```
nic.ifconfig(('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
```

network.**config**(*'param'*)
network.**config**(*param=value*, ...)
Get or set general network interface parameters. These methods allow to work with additional parameters beyond standard IP configuration (as dealt with by *`ifconfig()`*). These include network-specific and hardware-specific parameters and status values. For setting parameters, the keyword argument syntax should be used, and multiple parameters can be set at once. For querying, a parameter name should be quoted as a string, and only one parameter can be queried at a time:

```
# Set WiFi access point name (formally known as ESSID) and WiFi channel
ap.config(essid='My AP', channel=11)
# Query params one by one
print(ap.config('essid'))
print(ap.config('channel'))
```

(continues on next page)

(continued from previous page)

```
# Extended status information also available this way
print(sta.config('rssi'))
```

Functions

network.**phy_mode**([*mode*])
Get or set the PHY mode.

If the *mode* parameter is provided, sets the mode to its value. If the function is called without parameters, returns the current mode.

The possible modes are defined as constants:

- MODE_11B – IEEE 802.11b,
- MODE_11G – IEEE 802.11g,
- MODE_11N – IEEE 802.11n.

class WLAN

This class provides a driver for WiFi network processor in the ESP8266. Example usage:

```
import network
# enable station interface and connect to WiFi access point
nic = network.WLAN(network.STA_IF)
nic.active(True)
nic.connect('your-ssid', 'your-password')
# now use sockets as usual
```

Constructors

class network.**WLAN**(*interface_id*)

Create a WLAN network interface object. Supported interfaces are network.STA_IF (station aka client, connects to upstream WiFi access points) and network.AP_IF (access point, allows other WiFi clients to connect). Availability of the methods below depends on interface type. For example, only STA interface may *connect()* to an access point.

Methods

wlan.**active**([*is_active*])
Activate ("up") or deactivate ("down") network interface, if boolean argument is passed. Otherwise, query current state if no argument is provided. Most other methods require active interface.

wlan.**connect**(*ssid=None*, *password=None*, *, *bssid=None*)
Connect to the specified wireless network, using the specified password. If *bssid* is given then the connection will be restricted to the access-point with that MAC address (the *ssid* must also be specified in this case).

wlan.**disconnect**()
Disconnect from the currently connected wireless network.

wlan.**scan**()
Scan for the available wireless networks.

Scanning is only possible on STA interface. Returns list of tuples with the information about WiFi access points:

(ssid, bssid, channel, RSSI, authmode, hidden)

bssid is hardware address of an access point, in binary form, returned as bytes object. You can use *binascii.hexlify()* to convert it to ASCII form.

There are five values for authmode:

- 0 – open
- 1 – WEP
- 2 – WPA-PSK
- 3 – WPA2-PSK
- 4 – WPA/WPA2-PSK

and two for hidden:

- 0 – visible
- 1 – hidden

wlan.**status**()
Return the current status of the wireless connection.

The possible statuses are defined as constants:

- STAT_IDLE – no connection and no activity,
- STAT_CONNECTING – connecting in progress,
- STAT_WRONG_PASSWORD – failed due to incorrect password,
- STAT_NO_AP_FOUND – failed because no access point replied,
- STAT_CONNECT_FAIL – failed due to other problems,
- STAT_GOT_IP – connection successful.

wlan.**isconnected**()
In case of STA mode, returns True if connected to a WiFi access point and has a valid IP address. In AP mode returns True when a station is connected. Returns False otherwise.

wlan.**ifconfig**([*(ip, subnet, gateway, dns)*])
Get/set IP-level network interface parameters: IP address, subnet mask, gateway and DNS server. When called with no arguments, this method returns a 4-tuple with the above information. To set the above values, pass a 4-tuple with the required information. For example:

```
nic.ifconfig(('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
```

wlan.**config**(*'param'*)

wlan.**config**(*param=value, ...*)
Get or set general network interface parameters. These methods allow to work with additional parameters beyond standard IP configuration (as dealt with by *wlan.ifconfig()*). These include network-specific and hardware-specific parameters. For setting parameters, keyword argument syntax should be used, multiple parameters can be set at once. For querying, parameters name should be quoted as a string, and only one parameter can be queries at time:

```
# Set WiFi access point name (formally known as ESSID) and WiFi channel
ap.config(essid='My AP', channel=11)
# Query params one by one
print(ap.config('essid'))
print(ap.config('channel'))
```

Following are commonly supported parameters (availability of a specific parameter depends on network technology type, driver, and `MicroPython port`).

Parameter	Description
mac	MAC address (bytes)
essid	WiFi access point name (string)
channel	WiFi channel (integer)
hidden	Whether ESSID is hidden (boolean)
authmode	Authentication mode supported (enumeration, see module constants)
password	Access password (string)
dhcp_hostname	The DHCP hostname to use

`uctypes` – access binary data in a structured way

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

This module implements "foreign data interface" for MicroPython. The idea behind it is similar to CPython's `ctypes` modules, but the actual API is different, streamlined and optimized for small size. The basic idea of the module is to define data structure layout with about the same power as the C language allows, and the access it using familiar dot-syntax to reference sub-fields.

See also:

Module *`struct`* Standard Python way to access binary data structures (doesn't scale well to large and complex structures).

Defining structure layout

Structure layout is defined by a "descriptor" - a Python dictionary which encodes field names as keys and other properties required to access them as associated values. Currently, uctypes requires explicit specification of offsets for each field. Offset are given in bytes from a structure start.

Following are encoding examples for various field types:

- Scalar types:

```
"field_name": uctypes.UINT32 | 0
```

  in other words, value is scalar type identifier ORed with field offset (in bytes) from the start of the structure.

- Recursive structures:

```
"sub": (2, {
    "b0": uctypes.UINT8 | 0,
    "b1": uctypes.UINT8 | 1,
})
```

i.e. value is a 2-tuple, first element of which is offset, and second is a structure descriptor dictionary (note: offsets in recursive descriptors are relative to a structure it defines).

- Arrays of primitive types:

```
"arr": (uctypes.ARRAY | 0, uctypes.UINT8 | 2),
```

i.e. value is a 2-tuple, first element of which is ARRAY flag ORed with offset, and second is scalar element type ORed number of elements in array.

- Arrays of aggregate types:

```
"arr2": (uctypes.ARRAY | 0, 2, {"b": uctypes.UINT8 | 0}),
```

i.e. value is a 3-tuple, first element of which is ARRAY flag ORed with offset, second is a number of elements in array, and third is descriptor of element type.

- Pointer to a primitive type:

```
"ptr": (uctypes.PTR | 0, uctypes.UINT8),
```

i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, and second is scalar element type.

- Pointer to an aggregate type:

```
"ptr2": (uctypes.PTR | 0, {"b": uctypes.UINT8 | 0}),
```

i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, second is descriptor of type pointed to.

- Bitfields:

```
"bitf0": uctypes.BFUINT16 | 0 | 0 << uctypes.BF_POS | 8 << uctypes.BF_LEN,
```

i.e. value is type of scalar value containing given bitfield (typenames are similar to scalar types, but prefixes with "BF"), ORed with offset for scalar value containing the bitfield, and further ORed with values for bit offset and bit length of the bitfield within scalar value, shifted by BF_POS and BF_LEN positions, respectively. Bitfield position is counted from the least significant bit, and is the number of right-most bit of a field (in other words, it's a number of bits a scalar needs to be shifted right to extra the bitfield).

In the example above, first UINT16 value will be extracted at offset 0 (this detail may be important when accessing hardware registers, where particular access size and alignment are required), and then bitfield whose rightmost bit is least-significant bit of this UINT16, and length is 8 bits, will be extracted - effectively, this will access least-significant byte of UINT16.

Note that bitfield operations are independent of target byte endianness, in particular, example above will access least-significant byte of UINT16 in both little- and big-endian structures. But it depends on the least significant bit being numbered 0. Some targets may use different numbering in their native ABI, but `uctypes` always uses normalized numbering described above.

Module contents

class uctypes.**struct**(*addr*, *descriptor*, *layout_type=NATIVE*)
Instantiate a "foreign data structure" object based on structure address in memory, descriptor (encoded as a dictionary), and layout type (see below).

uctypes.**LITTLE_ENDIAN**
Layout type for a little-endian packed structure. (Packed means that every field occupies exactly as many bytes as defined in the descriptor, i.e. the alignment is 1).

uctypes.**BIG_ENDIAN**
Layout type for a big-endian packed structure.

uctypes.**NATIVE**
Layout type for a native structure - with data endianness and alignment conforming to the ABI of the system on which MicroPython runs.

uctypes.**sizeof**(*struct*)
Return size of data structure in bytes. Argument can be either structure class or specific instantiated structure object (or its aggregate field).

uctypes.**addressof**(*obj*)
Return address of an object. Argument should be bytes, bytearray or other object supporting buffer protocol (and address of this buffer is what actually returned).

uctypes.**bytes_at**(*addr*, *size*)
Capture memory at the given address and size as bytes object. As bytes object is immutable, memory is actually duplicated and copied into bytes object, so if memory contents change later, created object retains original value.

uctypes.**bytearray_at**(*addr*, *size*)
Capture memory at the given address and size as bytearray object. Unlike bytes_at() function above, memory is captured by reference, so it can be both written too, and you will access current value at the given memory address.

Structure descriptors and instantiating structure objects

Given a structure descriptor dictionary and its layout type, you can instantiate a specific structure instance at a given memory address using *uctypes.struct()* constructor. Memory address usually comes from following sources:

- Predefined address, when accessing hardware registers on a baremetal system. Lookup these addresses in datasheet for a particular MCU/SoC.
- As a return value from a call to some FFI (Foreign Function Interface) function.
- From uctypes.addressof(), when you want to pass arguments to an FFI function, or alternatively, to access some data for I/O (for example, data read from a file or network socket).

Structure objects

Structure objects allow accessing individual fields using standard dot notation: `my_struct.substruct1.field1`. If a field is of scalar type, getting it will produce a primitive value (Python integer or float) corresponding to the value contained in a field. A scalar field can also be assigned to.

If a field is an array, its individual elements can be accessed with the standard subscript operator `[]` - both read and assigned to.

If a field is a pointer, it can be dereferenced using `[0]` syntax (corresponding to C `*` operator, though `[0]` works in C too). Subscripting a pointer with other integer values but 0 are supported too, with the same semantics as in C.

Summing up, accessing structure fields generally follows C syntax, except for pointer dereference, when you need to use `[0]` operator instead of `*`.

Limitations

Accessing non-scalar fields leads to allocation of intermediate objects to represent them. This means that special care should be taken to layout a structure which needs to be accessed when memory allocation is disabled (e.g. from an interrupt). The recommendations are:

- Avoid nested structures. For example, instead of `mcu_registers.peripheral_a.register1`, define separate layout descriptors for each peripheral, to be accessed as `peripheral_a.register1`.
- Avoid other non-scalar data, like array. For example, instead of `peripheral_a.register[0]` use `peripheral_a.register0`.

Note that these recommendations will lead to decreased readability and conciseness of layouts, so they should be used only if the need to access structure fields without allocation is anticipated (it's even possible to define 2 parallel layouts - one for normal usage, and a restricted one to use when memory allocation is prohibited).

Libraries specific to the ESP8266

The following libraries are specific to the ESP8266.

`esp` — functions related to the ESP8266

Warning: Though this MicroPython-based library is available for use in CircuitPython, its functionality may change in the future, perhaps significantly. As CircuitPython continues to develop, it may be changed to comply more closely with the corresponding standard Python library. You may need to change your code later if you rely on any non-standard functionality it currently provides.

The `esp` module contains specific functions related to the ESP8266 module.

Functions

`esp.`**`sleep_type`**`(`[*sleep_type*]`)`
Get or set the sleep type.

If the *sleep_type* parameter is provided, sets the sleep type to its value. If the function is called without parameters, returns the current sleep type.

The possible sleep types are defined as constants:

- `SLEEP_NONE` – all functions enabled,
- `SLEEP_MODEM` – modem sleep, shuts down the WiFi Modem circuit.
- `SLEEP_LIGHT` – light sleep, shuts down the WiFi Modem circuit and suspends the processor periodically.

The system enters the set sleep mode automatically when possible.

`esp.`**`deepsleep`**`(`*time=0*`)`
Enter deep sleep.

The whole module powers down, except for the RTC clock circuit, which can be used to restart the module after the specified time if the pin 16 is connected to the reset pin. Otherwise the module will sleep until manually reset.

esp.**flash_id**()
Read the device ID of the flash memory.

esp.**flash_read**(*byte_offset*, *length_or_buffer*)

esp.**flash_write**(*byte_offset*, *bytes*)

esp.**flash_erase**(*sector_no*)

esp.**set_native_code_location**(*start*, *length*)
Set the location that native code will be placed for execution after it is compiled. Native code is emitted when the `@micropython.native`, `@micropython.viper` and `@micropython.asm_xtensa` decorators are applied to a function. The ESP8266 must execute code from either iRAM or the lower 1MByte of flash (which is memory mapped), and this function controls the location.

If *start* and *length* are both `None` then the native code location is set to the unused portion of memory at the end of the iRAM1 region. The size of this unused portion depends on the firmware and is typically quite small (around 500 bytes), and is enough to store a few very small functions. The advantage of using this iRAM1 region is that it does not get worn out by writing to it.

If neither *start* nor *length* are `None` then they should be integers. *start* should specify the byte offset from the beginning of the flash at which native code should be stored. *length* specifies how many bytes of flash from *start* can be used to store native code. *start* and *length* should be multiples of the sector size (being 4096 bytes). The flash will be automatically erased before writing to it so be sure to use a region of flash that is not otherwise used, for example by the firmware or the filesystem.

When using the flash to store native code *start+length* must be less than or equal to 1MByte. Note that the flash can be worn out if repeated erasures (and writes) are made so use this feature sparingly. In particular, native code needs to be recompiled and rewritten to flash on each boot (including wake from deepsleep).

In both cases above, using iRAM1 or flash, if there is no more room left in the specified region then the use of a native decorator on a function will lead to *MemoryError* exception being raised during compilation of that function.

1.8.10 Adafruit CircuitPython

Status | *Supported Boards* | *Download* | *Documentation* | *Contributing* | *Differences from Micropython* | *Project Structure*

CircuitPython is an *education friendly* open source derivative of MicroPython. CircuitPython supports use on educational development boards designed and sold by Adafruit. Adafruit CircuitPython features unified Python core APIs and a growing list of Adafruit libraries and drivers of that work with it.

Status

This project is stable. Most APIs should be stable going forward. Those that change will change on major version numbers such as 2.0.0 and 3.0.0.

Supported Boards

Designed for CircuitPython

- Adafruit CircuitPlayground Express (CircuitPython Guide)
- Adafruit Feather M0 Express (CircuitPython Guide)
- Adafruit Metro M0 Express (CircuitPython Guide)
- Adafruit Gemma M0 (CircuitPython Guide)
- Adafruit ItsyBitsy M0 Express (CircuitPython Guide)
- Adafruit Trinket M0 (CircuitPython Guide)
- Adafruit Metro M4 (CircuitPython Guide)

Other

- Adafruit Feather HUZZAH
- Adafruit Feather M0 Basic
- Adafruit Feather M0 Bluefruit LE (uses M0 Basic binaries)
- Adafruit Feather M0 Adalogger (MicroSD card supported using the Adafruit CircuitPython SD library)
- Arduino Zero

Download

Official binaries are available through the latest GitHub releases. Continuous (one per commit) builds are available here and includes experimental hardware support.

Documentation

Guides and videos are available through the Adafruit Learning System under the CircuitPython category and MicroPython category. An API reference is also available on Read the Docs. A collection of awesome resources can be found at Awesome CircuitPython.

Specifically useful documentation when starting out:

- Welcome to CircuitPython
- CircuitPython Essentials
- Example Code

Contributing

See CONTRIBUTING.md for full guidelines but please be aware that by contributing to this project you are agreeing to the Code of Conduct. Contributors who follow the Code of Conduct are welcome to submit pull requests and they will be promptly reviewed by project admins. Please join the Discord too.

Differences from MicroPython

CircuitPython:

- includes a ports for MicroChip SAMD21 (Commonly known as M0 in Adafruit product names) and SAMD51 (M4).
- supports only SAMD21, SAMD51, and ESP8266 ports. An nRF port is under development.
- tracks MicroPython's releases (not master).
- Longints (arbitrary-length integers) are enabled for most M0 Express boards (those boards with SPI flash chips external to the microcontroller), and for all M4 builds. Longints are disabled on other boards due to lack of flash space.

Behavior

- The order that files are run and the state that is shared between them. CircuitPython's goal is to clarify the role of each file and make each file independent from each other.
- `boot.py` (or `settings.py`) runs only once on start up before USB is initialized. This lays the ground work for configuring USB at startup rather than it being fixed. Since serial is not available, output is written to `boot_out.txt`.
- `code.py` (or `main.py`) is run after every reload until it finishes or is interrupted. After it is done running, the vm and hardware is reinitialized. **This means you cannot read state from ‘‘code.py‘‘ in the REPL anymore.** CircuitPython's goal for this change includes reduce confusion about pins and memory being used.
- After `code.py` the REPL can be entered by pressing any key. It no longer shares state with `code.py` so it is a fresh vm.
- Autoreload state will be maintained across reload.
- Adds a safe mode that does not run user code after a hard crash or brown out. The hope is that this will make it easier to fix code that causes nasty crashes by making it available through mass storage after the crash. A reset (the button) is needed after its fixed to get back into normal mode.

API

- Unified hardware APIs: audioio, analogio, busio, digitalio, pulseio, touchio, microcontroller, board, bitbangio
- No `machine` API on Atmel SAMD21 port.

Modules

- No module aliasing. (`uos` and `utime` are not available as `os` and `time` respectively.) Instead `os`, `time`, and `random` are CPython compatible.
- New `storage` module which manages file system mounts. (Functionality from `uos` in MicroPython.)
- Modules with a CPython counterpart, such as `time`, `os` and `random`, are strict subsets of their CPython version. Therefore, code from CircuitPython is runnable on CPython but not necessarily the reverse.
- tick count is available as time.monotonic()

atmel-samd21 features

- RGB status LED
- Auto-reload after file write over mass storage. (Disable with `samd.disable_autoreload()`)
- Wait state after boot and main run, before REPL.
- Main is one of these: `code.txt`, `code.py`, `main.py`, `main.txt`
- Boot is one of these: `settings.txt`, `settings.py`, `boot.py`, `boot.txt`

Project Structure

Here is an overview of the top-level source code directories.

Core

The core code of MicroPython is shared amongst ports including CircuitPython:

- `docs` High level user documentation in Sphinx reStructuredText format.
- `drivers` External device drivers written in Python.
- `examples` A few example Python scripts.
- `extmod` Shared C code used in multiple ports' modules.
- `lib` Shared core C code including externally developed libraries such as FATFS.
- `logo` The MicroPython logo.
- `mpy-cross` A cross compiler that converts Python files to byte code prior to being run in MicroPython. Useful for reducing library size.
- `py` Core Python implementation, including compiler, runtime, and core library.
- `shared-bindings` Shared definition of Python modules, their docs and backing C APIs. Ports must implement the C API to support the corresponding module.
- `shared-module` Shared implementation of Python modules that may be based on `common-hal`.
- `tests` Test framework and test scripts.
- `tools` Various tools, including the pyboard.py module.

Ports

Ports include the code unique to a microcontroller line and also variations based on the board.

- `atmel-samd` Support for SAMD21 based boards such as Arduino Zero, Adafruit Feather M0 Basic, and Adafruit Feather M0 Bluefruit LE.
- `bare-arm` A bare minimum version of MicroPython for ARM MCUs.
- `cc3200` Support for boards based CC3200 from TI such as the WiPy 1.0.
- `esp8266` Support for boards based on ESP8266 WiFi modules such as the Adafruit Feather HUZZAH.

- `minimal` A minimal MicroPython port. Start with this if you want to port MicroPython to another microcontroller.
- `pic16bit` Support for 16-bit PIC microcontrollers.
- `qemu-arm` Support for ARM emulation through QEMU.
- `stmhal` Support for boards based on STM32 microcontrollers including the MicroPython flagship PyBoard.
- `teensy` Support for the Teensy line of boards such as the Teensy 3.1.
- `unix` Support for UNIX.
- `windows` Support for Windows.
- `zephyr` Support for Zephyr, a real-time operating system by the Linux Foundation.

CircuitPython only maintains the `atmel-samd` and `esp8266` ports. The rest are here to maintain compatibility with the MicroPython parent project.

back to top

1.8.11 Contributing

Please note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms. Participation covers any forum used to converse about CircuitPython including unofficial and official spaces. Failure to do so will result in corrective actions such as time out or ban from the project.

Licensing

By contributing to this repository you are certifying that you have all necessary permissions to license the code under an MIT License. You still retain the copyright but are granting many permissions under the MIT License.

If you have an employment contract with your employer please make sure that they don't automatically own your work product. Make sure to get any necessary approvals before contributing. Another term for this contribution off-hours is moonlighting.

Getting started

CircuitPython developer Dan Halbert (@dhalbert) has written up build instructions using native build tools here.

For SAMD21 debugging workflow tips check out this learn guide from Scott (@tannewt).

Developer contacts

Scott Shawcroft (@tannewt) is the lead developer of CircuitPython and is sponsored by Adafruit Industries LLC. Scott is usually available during US West Coast working hours. Dan Halbert (@dhalbert) and Kattni Rembor (@kattni) are also sponsored by Adafruit Industries LLC and are usually available during US East Coast daytime hours including some weekends.

They are all reachable on Discord, GitHub issues and the Adafruit support forum.

Code guidelines

We aim to keep our code and commit style compatible with MicroPython upstream. Please review their code conventions to do so. Familiarity with their design philosophy is also useful though not always applicable to CircuitPython.

Furthermore, CircuitPython has a design guide that covers a variety of different topics. Please read it as well.

1.8.12 Contributor Covenant Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at support@adafruit.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4

1.8.13 MicroPython & CircuitPython license information

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

P

Q

R

S

T

U

V

W

Z