CircuitPython Documentation

Release 7.0.0-alpha.4

CircuitPython Contributors

Jul 09, 2021

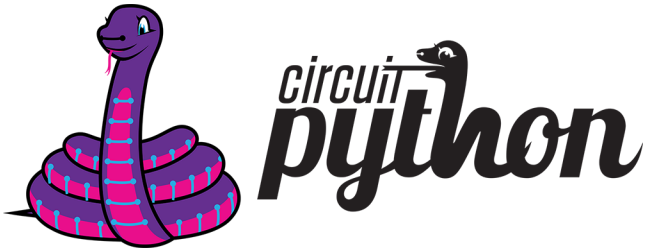
API AND USAGE

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

ONE

CIRCUITPYTHON



circuitpython.org | *Get CircuitPython* | *Documentation* | *Contributing* | *Branding* | *Differences from Micropython* | *Project Structure*

CircuitPython is a *beginner friendly*, open source version of Python for tiny, inexpensive computers called microcontrollers. Microcontrollers are the brains of many electronics including a wide variety of development boards used to build hobby projects and prototypes. CircuitPython in electronics is one of the best ways to learn to code because it connects code to reality. Simply install CircuitPython on a supported board via drag and drop and then edit a `code.py` file on the CIRCUITPY drive. The code will automatically reload. No software installs are needed besides a text editor (we recommend Mu for beginners.)

CircuitPython features unified Python core APIs and a growing list of 150+ device libraries and drivers that work with it. These libraries also work on single board computers with regular Python via the Adafruit Blinka Library.

CircuitPython is based on MicroPython. See *below* for differences. CircuitPython development is sponsored by Adafruit and is available on their educational development boards. Please support both MicroPython and Adafruit.

1.1 Get CircuitPython

Official binaries for all supported boards are available through circuitpython.org/downloads. The site includes stable, unstable and continuous builds. Full release notes and assets are available through GitHub releases as well.

1.2 Documentation

Guides and videos are available through the Adafruit Learning System under the CircuitPython 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

1.3 Code Search

GitHub doesn't currently support code search on forks. Therefore, CircuitPython doesn't have code search through GitHub because it is a fork of MicroPython. Luckily, SourceGraph has free code search for public repos like CircuitPython. So, visit sourcegraph.com/github.com/adafruit/circuitpython to search the CircuitPython codebase online.

1.4 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.

1.5 Branding

While we are happy to see CircuitPython forked and modified, we'd appreciate it if forked releases not use the name "CircuitPython" or the Blinka logo. "CircuitPython" means something special to us and those who learn about it. As a result, we'd like to make sure products referring to it meet a common set of requirements.

If you'd like to use the term "CircuitPython" and Blinka for your product here is what we ask:

- Your product is supported by the primary "adafruit/circuitpython" repo. This way we can update any custom code as we update the CircuitPython internals.
- Your product is listed on circuitpython.org (source here). This is to ensure that a user of your product can always download the latest version of CircuitPython from the standard place.
- Your product has a user accessible USB plug which appears as a CIRCUITPY drive when plugged in.

If you choose not to meet these requirements, then we ask you call your version of CircuitPython something else (for example, SuperDuperPython) and not use the Blinka logo. You can say it is "CircuitPython-compatible" if most CircuitPython drivers will work with it.

1.6 Differences from MicroPython

CircuitPython:

- Supports native USB on all boards, allowing file editing without special tools.
- Floats (aka decimals) are enabled for all builds.
- Error messages are translated into 10+ languages.
- Does not support concurrency within Python (including interrupts and threading). Some concurrency is achieved with native modules for tasks that require it such as audio file playback.

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, as the REPL is a fresh vm.** CircuitPython's goal for this change includes reducing confusion about pins and memory being used.
- After the main code is finished the REPL can be entered by pressing any key.
- 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 it's fixed to get back into normal mode.
- RGB status LED indicating CircuitPython state, and errors through a sequence of colored flashes.
- Re-runs `code.py` or other main file after file system writes over USB mass storage. (Disable with `supervisor.disable_autoreload()`)
- Autoreload is disabled while the REPL is active.
- 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`

1.6.2 API

- Unified hardware APIs. Documented on ReadTheDocs.
- API docs are rST within the C files in `shared-bindings`.
- No `machine` API.

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()

1.7 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 CircuitPython 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.

Supported	Support status
atmel-samd	`SAMD21` stable | `SAMD51` stable
cxd56	stable
esp32s2	stable
litex	alpha
mimxrt10xx	alpha
nrf	stable
raspberrypi	stable
stm	`F4` stable | `others` beta
unix	alpha

- `stable` Highly unlikely to have bugs or missing functionality.
- `beta` Being actively improved but may be missing functionality and have bugs.
- `alpha` Will have bugs and missing functionality.

1.7.3 Boards

- Each `port` has a `boards` directory containing variations of boards which belong to a specific microcontroller line.
- A list of native modules supported by a particular board can be found here.

Back to Top

1.8 Full Table of Contents

1.8.1 Core Modules

These core modules are intended on being consistent across ports and boards. A module may not exist on a port/board if no underlying hardware support is present or if flash space is limited. For example, a microcontroller without analog features will not have *analogio*. See the *Module Support Matrix - Which Modules Are Available on Which Boards* page for a list of modules supported on each board.

Module Support Matrix - Which Modules Are Available on Which Boards

The following table lists the available built-in modules for each CircuitPython capable board.

Board	Modules Available
8086 Commander	*adafruit_bus_device*, *analogio*, *board*, *busio*, *digitalio*, *gamepad*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
@sarfata shIRtty	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Adafruit BLM Badge	*analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *rainbow*, *random*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*
Adafruit Circuit Playground Bluefruit	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Adafruit Circuit Playground Express 4-H	*_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *bitbangio*, *board*, *busio*, *countio*, *digitalio*, *errno*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Circuit-Playground Express	*_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *bitbangio*, *board*, *busio*, *countio*, *digitalio*, *errno*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Circuit-Playground Express with Crickit libraries	*_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *errno*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Circuit-Playground Express with displayio	*analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*
Adafruit CLUE nRF52840 Express	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Adafruit Edge-Badge	*_bleio*, *_pixelbuf*, *_stage*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *gamepad*, *gamepadshift*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit Feather Bluefruit Sense	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Adafruit Feather M0 Adalogger	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Feather M0 Basic	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Feather M0 Express	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Feather M0 Express with Crickit libraries	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *errno*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Feather M0 RFM69	*adafruit_bus_device*, *analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *storage*, *struct*, *supervisor*, *time*, *usb_cdc*
Adafruit Feather M0 RFM9x	*adafruit_bus_device*, *analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *storage*, *struct*, *supervisor*, *time*, *usb_cdc*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Adafruit Feather M4 CAN	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit Feather M4 Express	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit Feather MIMXRT1011	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit Feather nRF52840 Express	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Adafruit Feather RP2040	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Adafruit Feather STM32F405 Express	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiocore*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *sdcardio*, *sdioio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit FunHouse	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
Adafruit Gemma M0	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Gemma M0 PyCon 2018	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Grand Central M4 Express	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sdioio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit Hallowing M4 Express	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit ItsyBitsy M0 Express	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Adafruit ItsyBitsy M4 Express	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit ItsyBitsy nRF52840 Express	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Adafruit ItsyBitsy RP2040	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Adafruit Macropad RP2040	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Adafruit MagTag	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Adafruit Matrix Portal M4	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit Metro ESP32S2	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
Adafruit Metro M0 Express	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Metro M4 Airlift Lite	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit Metro M4 Express	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Adafruit Metro nRF52840 Express	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Adafruit Monster M4SK	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit NeoKey Trinkey M0	*_pixelbuf*, *board*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *rainbow*, *random*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit NeoPixel Trinkey M0	*_pixelbuf*, *board*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *rainbow*, *random*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit ProxLight Trinkey M0	*_pixelbuf*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *rainbow*, *random*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*
Adafruit Pybadge	*_bleio*, *_pixelbuf*, *_stage*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *gamepad*, *gamepadshift*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit PyGamer	*_bleio*, *_pixelbuf*, *_stage*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *gamepad*, *gamepadshift*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Adafruit PyPortal	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit PyPortal Pynt	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit PyPortal Titano	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit PyRuler	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit QT Py M0	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit QT Py M0 Haxpress	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit QT Py RP2040	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Adafruit QT2040 Trinkey	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Adafruit Rotary Trinkey M0	*_pixelbuf*, *board*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *rainbow*, *random*, *rotaryio*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Slide Trinkey M0	*_pixelbuf*, *analogio*, *board*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *rainbow*, *random*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Adafruit Trellis M4 Express	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Adafruit Trinket M0	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
AlorimTech Evo M51	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
ARAMCON Badge 2019	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
ARAM-CON2 Badge	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Arduino MKR Zero	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Arduino MKR1300	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Arduino Nano 33 BLE	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Arduino Nano 33 IoT	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Arduino Nano RP2040 Connect	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Arduino Zero	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Artisense Reference Design RD00	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
Atelier-DuMaker nRF52840 Breakout	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
AT-MegaZero ESP32-S2	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
BastBLE	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
BastWiFi	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
BDMI-CRO VINA-D21	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
BDMI-CRO VINA-D51	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
BLE-SS dev board Multi Sensor	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Capable Robot Programmable USB Hub	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Cedar Grove String-Car M0 Express	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Circuit Playground Express Digi-Key PyCon 2019	*_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *bitbangio*, *board*, *busio*, *countio*, *digitalio*, *errno*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Circuit-Brains Basic	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Circuit-Brains Deluxe	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
CP Sapling M0	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
CP Sapling M0 w/ SPI Flash	*_pixelbuf*, *analogio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
CP32-M4	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Cytron Maker Pi RP2040	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
DynaLoRa_USB	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *sdcardio*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
DynOSSAT-EDU-EPS	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *usb_cdc*, *usb_hid*, *usb_midi*
DynOSSAT-EDU-OBC	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Electronic Cats Bast Pro Mini M0	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Electronic Cats Cat-WAN USBStick	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Electronic Cats Hunter Cat NFC	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Electronic Cats NFC Copy Cat	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Electronut Labs Blip	*_bleio*, *_pixelbuf*, *_stage*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *gamepad*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Electronut Labs Papyr	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Escornabot Makech	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*
Espruino Pico	*_bleio*, *_pixelbuf*, *analogio*, *binascii*, *bitbangio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *json*, *math*, *microcontroller*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *sdcardio*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Espruino Wifi	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Feather ESP32S2 without PSRAM	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
Feather MIMXRT1011	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Feather MIMXRT1062	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
FeatherS2	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
FeatherS2 PreRelease	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
Fluff M0	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Fomu	*_pixelbuf*, *binascii*, *digitalio*, *errno*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *rainbow*, *random*, *re*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*
Franzininho WIFI w/Wroom	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
Franzininho WIFI w/Wrover	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
Hacked Feather M0 Express with 8Mbyte SPI flash	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
HalloWing M0 Express	*_pixelbuf*, *analogio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
HiiBot BlueFi	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
IkigaiSense Vita nRF52840	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
iMX RT 1020 EVK	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
iMX RT 1060 EVK	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
IMXRT1010-EVK	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
J&J Studios datum-Distance	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
J&J Studios datum-IMU	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
J&J Studios datum-Light	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
J&J Studios datum-Weather	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Kaluga 1	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
keithp.com snekboard	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
LILYGO TTGO T8 ESP32-S2 w/Display	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
LoC BeR M4 base board	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Maker-diary M60 Keyboard	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Makerdiary nRF52840 M.2 Developer Kit	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
MakerDiary nRF52840 MDK	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
MakerDiary nRF52840 MDK USB Dongle	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Makerdiary Pitaya Go	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
MDBT50Q-DB-40	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Meow Meow	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
MEOW-BIT	*_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiocore*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Metro MIMXRT1011	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
microS2	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
Mini SAM M4	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
nanoESP32-S2 w/Wrover	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
nanoESP32-S2 w/Wroom	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
ndGarage[n°] Bit6: FeatherSnow-v2	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
ndGarage[n°] Bit6:FeatherSnow	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
nice!nano	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
NUCLEO STM32F746	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
NUCLEO STM32F767	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
NUCLEO STM32H743	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *os*, *rainbow*, *random*, *re*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Open Hardware Summit 2020 Badge	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
OPENMV-H7 R1	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *os*, *rainbow*, *random*, *re*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Particle Argon	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Particle Boron	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Particle Xenon	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
PCA10056 nRF52840-DK	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
PCA10059 nRF52840 Dongle	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
PCA10100 nRF52833 Dongle	*_bleio*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiopwmio*, *board*, *busio*, *digitalio*, *errno*, *math*, *microcontroller*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *synthio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *watchdog*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
PewPew 10.2	*_pew*, *analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*
PewPew 13	*_pew*, *analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*
PewPew M4	*_stage*, *analogio*, *audiocore*, *audioio*, *audiomixer*, *board*, *busio*, *digitalio*, *displayio*, *fontio*, *keypad*, *math*, *microcontroller*, *nvm*, *os*, *rainbow*, *random*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *usb_cdc*
PicoPlanet	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Pimoroni Keybow 2040	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Pimoroni PGA2040	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Pimoroni Pico LiPo (16MB)	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Pimoroni Pico LiPo (4MB)	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Pimoroni PicoSystem	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
Pimoroni Tiny 2040	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
PYB LR Nano V2	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *math*, *microcontroller*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
PyboardV1_1	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiocore*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *sdcardio*, *sdioio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
PyCubedv04	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *board*, *busio*, *countio*, *digitalio*, *errno*, *frequencyio*, *i2cperipheral*, *json*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rotaryio*, *rtc*, *sdcardio*, *storage*, *struct*, *supervisor*, *synthio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
PyCubedv04-MRAM	_bleio, _pixelbuf, adafruit_bus_device, analogio, audiocore, audioio, audiomixer, audiomp3, binascii, bitbangio, board, busio, countio, digitalio, errno, frequencyio, i2cperipheral, json, math, microcontroller, msgpack, neopixel_write, nvm, os, pulseio, pwmio, rainbow, random, re, rotaryio, rtc, sdcardio, storage, struct, supervisor, synthio, time, touchio, ulab, usb_cdc, usb_hid, usb_midi
Raspberry Pi Pico	_bleio, _pixelbuf, adafruit_bus_device, alarm, analogio, audiobusio, audiocore, audiomixer, audiomp3, audiopwmio, binascii, bitbangio, bitmaptools, bitops, board, busio, countio, digitalio, displayio, errno, fontio, framebufferio, imagecapture, json, keypad, math, microcontroller, msgpack, neopixel_write, nvm, os, pulseio, pwmio, rainbow, random, re, rgbmatrix, rotaryio, rtc, sdcardio, sharpdisplay, storage, struct, supervisor, synthio, terminalio, time, touchio, ulab, usb_cdc, usb_hid, usb_midi, vectorio, watchdog
Robo HAT MM1 M4	_bleio, _pixelbuf, adafruit_bus_device, analogio, audiocore, audioio, audiomixer, audiomp3, binascii, bitbangio, board, busio, countio, digitalio, errno, i2cperipheral, json, keypad, math, microcontroller, msgpack, neopixel_write, nvm, os, ps2io, pulseio, pwmio, rainbow, random, re, rotaryio, rtc, sdcardio, storage, struct, supervisor, synthio, time, touchio, ulab, usb_cdc, usb_hid, usb_midi
SAM E54 Xplained Pro	_bleio, _pixelbuf, adafruit_bus_device, analogio, audiobusio, audiocore, audioio, audiomixer, audiomp3, binascii, bitbangio, bitmaptools, board, busio, canio, countio, digitalio, displayio, errno, fontio, framebufferio, frequencyio, i2cperipheral, json, keypad, math, microcontroller, msgpack, neopixel_write, nvm, os, pulseio, pwmio, rainbow, random, re, rotaryio, rtc, sdcardio, sdioio, sharpdisplay, storage, struct, supervisor, synthio, terminalio, time, touchio, ulab, usb_cdc, usb_hid, usb_midi, vectorio
SAM32v26	_bleio, _pixelbuf, adafruit_bus_device, analogio, audiobusio, audiocore, audioio, audiomixer, audiomp3, binascii, bitbangio, bitmaptools, board, busio, countio, digitalio, displayio, errno, fontio, framebufferio, frequencyio, i2cperipheral, json, keypad, math, microcontroller, msgpack, neopixel_write, nvm, os, ps2io, pulseio, pwmio, rainbow, random, re, rgbmatrix, rotaryio, rtc, sdcardio, sharpdisplay, storage, struct, supervisor, synthio, terminalio, time, touchio, ulab, usb_cdc, usb_hid, usb_midi, vectorio
Saola 1 w/Wroom	_bleio, _pixelbuf, adafruit_bus_device, alarm, analogio, audiobusio, audiocore, audiomixer, binascii, bitbangio, bitmaptools, board, busio, canio, countio, digitalio, displayio, dualbank, errno, fontio, framebufferio, frequencyio, imagecapture, ipaddress, json, keypad, math, microcontroller, msgpack, neopixel_write, nvm, os, ps2io, pulseio, pwmio, rainbow, random, re, rgbmatrix, rotaryio, rtc, sdcardio, sharpdisplay, socketpool, ssl, storage, struct, supervisor, synthio, terminalio, time, touchio, ulab, usb_cdc, usb_hid, usb_midi, vectorio, watchdog, wifi

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Saola 1 w/Wrover	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
Seeeduino Wio Terminal	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Seeeduino XIAO	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
senseBox MCU	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Serpente	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
Silicognition LLC M4-Shim	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Simmel	*_bleio*, *aesio*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiopwmio*, *binascii*, *bitbangio*, *board*, *busio*, *digitalio*, *json*, *math*, *microcontroller*, *os*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *usb_hid*, *watchdog*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
SparkFun LUMIDrive	*_pixelbuf*, *analogio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
SparkFun MicroMod ATP - RP2040	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
SparkFun MicroMod nRF52840	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
SparkFun MicroMod SAMD51	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
SparkFun Pro Micro RP2040	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
SparkFun Pro nRF52840 Mini	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
SparkFun Qwiic Micro	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
SparkFun RedBoard Turbo	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
SparkFun SAMD21 Dev Breakout	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
SparkFun SAMD21 Mini Breakout	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
SparkFun Thing Plus - RP2040	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *bitops*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *imagecapture*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
SparkFun Thing Plus - SAMD51	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
SPRE-SENSE	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *board*, *busio*, *camera*, *digitalio*, *errno*, *gnss*, *json*, *keypad*, *math*, *microcontroller*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *sdcardio*, *sdioio*, *storage*, *struct*, *supervisor*, *time*, *ulab*, *usb_cdc*
Sprite_v2b	*_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiocore*, *audioio*, *audiomixer*, *binascii*, *bitbangio*, *board*, *busio*, *countio*, *digitalio*, *errno*, *frequencyio*, *i2cperipheral*, *json*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *sdcardio*, *storage*, *struct*, *supervisor*, *synthio*, *time*, *usb_cdc*, *usb_hid*, *usb_midi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
ST STM32F746G Discovery	_bleio, _pixelbuf, adafruit_bus_device, binascii, bitbangio, bitmaptools, board, busio, digitalio, displayio, errno, fontio, framebufferio, json, keypad, math, microcontroller, msgpack, os, pulseio, pwmio, rainbow, random, re, sdcardio, sharpdisplay, storage, struct, supervisor, terminalio, time, touchio, ulab, usb_cdc, usb_hid, usb_midi, vectorio
StackR-duino M0 PRO	_pixelbuf, analogio, audiobusio, audiocore, audioio, board, busio, digitalio, displayio, errno, fontio, math, microcontroller, neopixel_write, nvm, os, pulseio, pwmio, rainbow, random, rotaryio, rtc, storage, struct, supervisor, terminalio, time, touchio, usb_cdc, usb_hid, usb_midi
stm32f411ce-blackpill	_bleio, _pixelbuf, adafruit_bus_device, analogio, binascii, bitbangio, bitmaptools, board, busio, digitalio, displayio, errno, fontio, framebufferio, json, keypad, math, microcontroller, msgpack, neopixel_write, os, pulseio, pwmio, rainbow, random, re, sdcardio, sharpdisplay, storage, struct, supervisor, terminalio, time, touchio, usb_cdc, usb_hid, usb_midi, vectorio
stm32f411ce-blackpill-with-flash	_bleio, _pixelbuf, adafruit_bus_device, analogio, binascii, bitbangio, board, busio, digitalio, displayio, errno, fontio, framebufferio, json, keypad, math, microcontroller, msgpack, neopixel_write, os, pulseio, pwmio, rainbow, random, re, sdcardio, sharpdisplay, storage, struct, supervisor, terminalio, time, touchio, ulab, usb_cdc, usb_hid, usb_midi
STM32F411E_DISCO	_bleio, _pixelbuf, adafruit_bus_device, analogio, binascii, bitbangio, bitmaptools, board, busio, digitalio, displayio, errno, fontio, framebufferio, json, math, microcontroller, neopixel_write, os, pulseio, pwmio, rainbow, random, re, sdcardio, sharpdisplay, storage, struct, supervisor, terminalio, time, touchio, ulab, usb_cdc, usb_hid, usb_midi, vectorio
STM32F412G_DISCO	_bleio, _pixelbuf, adafruit_bus_device, analogio, audiocore, audiomp3, audiopwmio, binascii, bitbangio, bitmaptools, board, busio, digitalio, displayio, errno, fontio, framebufferio, json, keypad, math, microcontroller, msgpack, neopixel_write, os, pulseio, pwmio, rainbow, random, re, sdcardio, sharpdisplay, storage, struct, supervisor, synthio, terminalio, time, touchio, ulab, usb_cdc, usb_hid, usb_midi, vectorio
STM32F4_DISCO	_bleio, _pixelbuf, adafruit_bus_device, alarm, analogio, audiocore, audiomp3, audiopwmio, binascii, bitbangio, bitmaptools, board, busio, canio, digitalio, displayio, errno, fontio, framebufferio, json, keypad, math, microcontroller, msgpack, neopixel_write, os, pulseio, pwmio, rainbow, random, re, sdcardio, sdioio, sharpdisplay, storage, struct, supervisor, synthio, terminalio, time, touchio, ulab, usb_cdc, usb_hid, usb_midi, vectorio

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Targett Module Clip w/Wroom	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
Targett Module Clip w/Wrover	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*
Teensy 4.0	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Teensy 4.1	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
Teknikio Bluebird	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
TG-Boards' Datalore IP M4	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
TG-Watch	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
The Open Book Feather	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *gamepadshift*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
THUNDERPACK_v11	*_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiocore*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
THUNDERPACK_v12	*_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiocore*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_midi*, *vectorio*
TinkeringTech Scout-Makes Azul	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *aesio*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *audiomp3*, *audiopwmio*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*
TinyS2	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *alarm*, *analogio*, *audiobusio*, *audiocore*, *audiomixer*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *canio*, *countio*, *digitalio*, *displayio*, *dualbank*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *imagecapture*, *ipaddress*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *socketpool*, *ssl*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*, *watchdog*, *wifi*

continues on next page

Table 1 – continued from previous page

Board	Modules Available
Trinket M0 Haxpress	*_pixelbuf*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
UART-Logger II	*_bleio*, *_pixelbuf*, *adafruit_bus_device*, *analogio*, *audiobusio*, *audiocore*, *audioio*, *audiomixer*, *audiomp3*, *binascii*, *bitbangio*, *bitmaptools*, *board*, *busio*, *countio*, *digitalio*, *displayio*, *errno*, *fontio*, *framebufferio*, *frequencyio*, *i2cperipheral*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *ps2io*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rgbmatrix*, *rotaryio*, *rtc*, *sdcardio*, *sharpdisplay*, *storage*, *struct*, *supervisor*, *synthio*, *terminalio*, *time*, *touchio*, *ulab*, *usb_cdc*, *usb_hid*, *usb_midi*, *vectorio*
uChip	*analogio*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *touchio*, *usb_cdc*, *usb_hid*, *usb_midi*
uGame10	*_stage*, *analogio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *displayio*, *errno*, *fontio*, *gamepad*, *math*, *microcontroller*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rotaryio*, *storage*, *struct*, *supervisor*, *terminalio*, *time*, *usb_cdc*
Winterbloom Big Honking Button	*_pixelbuf*, *analogio*, *audiocore*, *audioio*, *board*, *busio*, *digitalio*, *errno*, *math*, *microcontroller*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *rotaryio*, *storage*, *struct*, *supervisor*, *time*, *usb_cdc*
Winterbloom Sol	*_pixelbuf*, *adafruit_bus_device*, *analogio*, *binascii*, *bitbangio*, *board*, *busio*, *countio*, *digitalio*, *errno*, *frequencyio*, *json*, *keypad*, *math*, *microcontroller*, *msgpack*, *neopixel_write*, *nvm*, *os*, *pulseio*, *pwmio*, *rainbow*, *random*, *re*, *rotaryio*, *rtc*, *sdcardio*, *storage*, *struct*, *supervisor*, *time*, *ulab*, *usb_cdc*, *usb_midi*
XinaBox CC03	*adafruit_bus_device*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *rtc*, *storage*, *struct*, *supervisor*, *time*, *usb_cdc*, *usb_hid*, *usb_midi*
XinaBox CS11	*adafruit_bus_device*, *board*, *busio*, *digitalio*, *math*, *microcontroller*, *nvm*, *os*, *pwmio*, *rainbow*, *random*, *sdcardio*, *storage*, *struct*, *supervisor*, *time*, *usb_cdc*, *usb_hid*

Modules

_bleio – Bluetooth Low Energy (BLE) communication

The *_bleio* module provides necessary low-level functionality for communicating using Bluetooth Low Energy (BLE). The '_' prefix indicates this module is meant for internal use by libraries but not by the end user. Its API may change incompatibly between minor versions of CircuitPython. Please use the adafruit_ble CircuitPython library instead, which builds on *_bleio*, and provides higher-level convenience functionality, including predefined beacons, clients, servers.

_bleio.**adapter** :**Adapter**
BLE Adapter used to manage device discovery and connections. This object is the sole instance of *_bleio.Adapter*.

exception _bleio.**BluetoothError**
Bases: *Exception*

Catchall exception for Bluetooth related errors.

Initialize self. See help(type(self)) for accurate signature.

exception _bleio.**RoleError**
Bases: *BluetoothError*

Raised when a resource is used as the mismatched role. For example, if a local CCCD is attempted to be set but they can only be set when remote.

Initialize self. See help(type(self)) for accurate signature.

exception _bleio.**SecurityError**
Bases: *BluetoothError*

Raised when a security related error occurs.

Initialize self. See help(type(self)) for accurate signature.

_bleio.**set_adapter**(*adapter: Optional[*Adapter*]*) → None
Set the adapter to use for BLE, such as when using an HCI adapter. Raises *NotImplementedError* when the adapter is a singleton and cannot be set.

class _bleio.**Adapter**(*, *uart:* busio.UART, *rts:* digitalio.DigitalInOut, *cts:* digitalio.DigitalInOut)
The BLE Adapter object manages the discovery and connection to other nearby Bluetooth Low Energy devices. This part of the Bluetooth Low Energy Specification is known as Generic Access Profile (GAP).

Discovery of other devices happens during a scanning process that listens for small packets of information, known as advertisements, that are broadcast unencrypted. The advertising packets have two different uses. The first is to broadcast a small piece of data to anyone who cares and and nothing more. These are known as beacons. The second class of advertisement is to promote additional functionality available after the devices establish a connection. For example, a BLE heart rate monitor would advertise that it provides the standard BLE Heart Rate Service.

The Adapter can do both parts of this process: it can scan for other device advertisements and it can advertise its own data. Furthermore, Adapters can accept incoming connections and also initiate connections.

On boards that do not have native BLE, you can use an HCI co-processor. Pass the uart and pins used to communicate with the co-processor, such as an Adafruit AirLift. The co-processor must have been reset and put into BLE mode beforehand by the appropriate pin manipulation. The `uart`, `rts`, and `cts` objects are used to communicate with the HCI co-processor in HCI mode. The *Adapter* object is enabled during this call.

After instantiating an Adapter, call *_bleio.set_adapter()* to set *_bleio.adapter*

On boards with native BLE, you cannot create an instance of *_bleio.Adapter*; this constructor will raise *NotImplementedError*. Use *_bleio.adapter* to access the sole instance already available.

enabled :bool
State of the BLE adapter.

address :Address
MAC address of the BLE adapter.

name :str
name of the BLE adapter used once connected. The name is "CIRCUITPY" + the last four hex digits of `adapter.address`, to make it easy to distinguish multiple CircuitPython boards.

advertising :bool
True when the adapter is currently advertising. (read-only)

connected :bool
True when the adapter is connected to another device regardless of who initiated the connection. (read-only)

connections :Tuple[Connection]
Tuple of active connections including those initiated through *_bleio.Adapter.connect()*. (read-only)

start_advertising(*self*, *data:* _typing.ReadableBuffer, *, *scan_response: Optional[_typing.ReadableBuffer] = None*, *connectable:* bool *= True*, *anonymous:* bool *= False*, *timeout:* int *= 0*, *interval:* float *= 0.1*, *tx_power:* int *= 0*, *directed_to: Optional[*Address*] = None*) → None
Starts advertising until *stop_advertising* is called or if connectable, another device connects to us.

Parameters

- **data** (`ReadableBuffer`) – advertising data packet bytes
- **scan_response** (`ReadableBuffer`) – scan response data packet bytes. `None` if no scan response is needed.
- **connectable** (`bool`) – If `True` then other devices are allowed to connect to this peripheral.
- **anonymous** (`bool`) – If `True` then this device's MAC address is randomized before advertising.
- **timeout** (`int`) – If set, we will only advertise for this many seconds. Zero means no timeout.
- **interval** (`float`) – advertising interval, in seconds
- **int** (*tx_power*) – transmitter power while advertising in dBm
- **Address** (*directed_to*) – peer to advertise directly to

stop_advertising(*self*) → None
Stop sending advertising packets.

start_scan(*self*, *prefixes:* _typing.ReadableBuffer *= b''*, *, *buffer_size:* int *= 512*, *extended:* bool *= False*, *timeout: Optional[*float*] = None*, *interval:* float *= 0.1*, *window:* float *= 0.1*, *minimum_rssi:* int *= - 80*, *active:* bool *= True*) → Iterable[*ScanEntry*]
Starts a BLE scan and returns an iterator of results. Advertisements and scan responses are filtered and returned separately.

Parameters

- **prefixes** (ReadableBuffer) – Sequence of byte string prefixes to filter advertising packets with. A packet without an advertising structure that matches one of the prefixes is ignored. Format is one byte for length (n) and n bytes of prefix and can be repeated.
- **buffer_size** (int) – the maximum number of advertising bytes to buffer.
- **extended** (bool) – When True, support extended advertising packets. Increasing buffer_size is recommended when this is set.
- **timeout** (float) – the scan timeout in seconds. If None or zero, will scan until *stop_scan* is called.
- **interval** (float) – the interval (in seconds) between the start of two consecutive scan windows Must be in the range 0.0025 - 40.959375 seconds.
- **window** (float) – the duration (in seconds) to scan a single BLE channel. window must be <= interval.
- **minimum_rssi** (int) – the minimum rssi of entries to return.
- **active** (bool) – retrieve scan responses for scannable advertisements.

Returns an iterable of *_bleio.ScanEntry* objects

Return type iterable

stop_scan(*self*) → None
Stop the current scan.

connect(*self*, *address:* Address, *, *timeout:* float) → *Connection*
Attempts a connection to the device with the given address.

Parameters

- **address** (Address) – The address of the peripheral to connect to
- **timeout** (*float/int*) – Try to connect for timeout seconds.

erase_bonding(*self*) → None
Erase all bonding information stored in flash memory.

class _bleio.**Address**(*address:* _typing.ReadableBuffer, *address_type:* int)
Encapsulates the address of a BLE device.

Create a new Address object encapsulating the address value. The value itself can be one of:

Parameters

- **address** (ReadableBuffer) – The address value to encapsulate. A buffer object (bytearray, bytes) of 6 bytes.
- **address_type** (int) – one of the integer values: *PUBLIC*, *RANDOM_STATIC*, *RANDOM_PRIVATE_RESOLVABLE*, or *RANDOM_PRIVATE_NON_RESOLVABLE*.

address_bytes :bytes
The bytes that make up the device address (read-only).

Note that the bytes object returned is in little-endian order: The least significant byte is address_bytes[0]. So the address will appear to be reversed if you print the raw bytes object. If you print or use *str()* on the *Attribute* object itself, the address will be printed in the expected order. For example:

```
>>> import _bleio
>>> _bleio.adapter.address
<Address c8:1d:f5:ed:a8:35>
>>> _bleio.adapter.address.address_bytes
b'5\xa8\xed\xf5\x1d\xc8'
```

type :int
The address type (read-only).

One of the integer values: *PUBLIC*, *RANDOM_STATIC*, *RANDOM_PRIVATE_RESOLVABLE*, or *RANDOM_PRIVATE_NON_RESOLVABLE*.

PUBLIC :int
A publicly known address, with a company ID (high 24 bits)and company-assigned part (low 24 bits).

RANDOM_STATIC :int
A randomly generated address that does not change often. It may never change or may change after a power cycle.

RANDOM_PRIVATE_RESOLVABLE :int
An address that is usable when the peer knows the other device's secret Identity Resolving Key (IRK).

RANDOM_PRIVATE_NON_RESOLVABLE :int
A randomly generated address that changes on every connection.

__eq__ (*self*, *other:* object) → *bool*
Two Address objects are equal if their addresses and address types are equal.

__hash__ (*self*) → *int*
Returns a hash for the Address data.

class _bleio.**Attribute**
Definitions associated with all BLE attributes: characteristics, descriptors, etc.

Attribute is, notionally, a superclass of *Characteristic* and *Descriptor*, but is not defined as a Python superclass of those classes.

You cannot create an instance of *Attribute*.

NO_ACCESS :int
security mode: access not allowed

OPEN :int
security_mode: no security (link is not encrypted)

ENCRYPT_NO_MITM :int
security_mode: unauthenticated encryption, without man-in-the-middle protection

ENCRYPT_WITH_MITM :int
security_mode: authenticated encryption, with man-in-the-middle protection

LESC_ENCRYPT_WITH_MITM :int
security_mode: LESC encryption, with man-in-the-middle protection

SIGNED_NO_MITM :int
security_mode: unauthenticated data signing, without man-in-the-middle protection

SIGNED_WITH_MITM :int
security_mode: authenticated data signing, without man-in-the-middle protection

class _bleio.**Characteristic**
Stores information about a BLE service characteristic and allows reading and writing of the characteristic's value.

There is no regular constructor for a Characteristic. A new local Characteristic can be created and attached to a Service by calling *add_to_service()*. Remote Characteristic objects are created by *Connection.discover_remote_services()* as part of remote Services.

properties :int
An int bitmask representing which properties are set, specified as bitwise or'ing of of these possible values. *BROADCAST*, *INDICATE*, *NOTIFY*, *READ*, *WRITE*, *WRITE_NO_RESPONSE*.

uuid :Optional[UUID]
The UUID of this characteristic. (read-only)

Will be `None` if the 128-bit UUID for this characteristic is not known.

value :bytearray
The value of this characteristic.

max_length :int
The max length of this characteristic.

descriptors :Descriptor
A tuple of *Descriptor* objects related to this characteristic. (read-only)

service :Service
The Service this Characteristic is a part of.

BROADCAST :int
property: allowed in advertising packets

INDICATE :int
property: server will indicate to the client when the value is set and wait for a response

NOTIFY :int
property: server will notify the client when the value is set

READ :int
property: clients may read this characteristic

WRITE :int
property: clients may write this characteristic; a response will be sent back

WRITE_NO_RESPONSE :int
property: clients may write this characteristic; no response will be sent back

add_to_service(*self*, *service:* Service, *uuid:* UUID, *, *properties:* int = *0*, *read_perm:* int = *Attribute.OPEN*, *write_perm:* int = *Attribute.OPEN*, *max_length:* int = *20*, *fixed_length:* bool = *False*, *initial_value:* *Optional[*_typing.ReadableBuffer*]* = *None*, *user_description: Optional[*str*]* = *None*) → *Characteristic*
Create a new Characteristic object, and add it to this Service.

Parameters

- **service** (Service) – The service that will provide this characteristic
- **uuid** (UUID) – The uuid of the characteristic
- **properties** (int) – The properties of the characteristic, specified as a bitmask of these values bitwise-or'd together: *BROADCAST*, *INDICATE*, *NOTIFY*, *READ*, *WRITE*, *WRITE_NO_RESPONSE*.
- **read_perm** (int) – Specifies whether the characteristic can be read by a client, and if so, which security mode is required. Must be one of the integer values *Attribute.NO_ACCESS*, *Attribute.OPEN*, *Attribute.ENCRYPT_NO_MITM*, *Attribute.ENCRYPT_WITH_MITM*, *Attribute.*

LESC_ENCRYPT_WITH_MITM, *Attribute.SIGNED_NO_MITM*, or *Attribute.SIGNED_WITH_MITM*.

- **write_perm** (int) – Specifies whether the characteristic can be written by a client, and if so, which security mode is required. Values allowed are the same as read_perm.
- **max_length** (int) – Maximum length in bytes of the characteristic value. The maximum allowed is is 512, or possibly 510 if fixed_length is False. The default, 20, is the maximum number of data bytes that fit in a single BLE 4.x ATT packet.
- **fixed_length** (bool) – True if the characteristic value is of fixed length.
- **initial_value** (ReadableBuffer) – The initial value for this characteristic. If not given, will be filled with zeros.
- **user_description** (str) – User friendly description of the characteristic

Returns the new Characteristic.

set_cccd(*self*, *, *notify:* bool = *False*, *indicate:* bool = *False*) → None
Set the remote characteristic's CCCD to enable or disable notification and indication.

Parameters

- **notify** (bool) – True if Characteristic should receive notifications of remote writes
- **indicate** (float) – True if Characteristic should receive indications of remote writes

class _bleio.**CharacteristicBuffer**(*characteristic:* Characteristic, *, *timeout:* int = *1*, *buffer_size:* int = *64*)
Accumulates a Characteristic's incoming values in a FIFO buffer.

Monitor the given Characteristic. Each time a new value is written to the Characteristic add the newly-written bytes to a FIFO buffer.

Parameters

- **characteristic** (Characteristic) – The Characteristic to monitor. It may be a local Characteristic provided by a Peripheral Service, or a remote Characteristic in a remote Service that a Central has connected to.
- **timeout** (int) – the timeout in seconds to wait for the first character and between subsequent characters.
- **buffer_size** (int) – Size of ring buffer that stores incoming data coming from client. Must be >= 1.

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

read(*self*, *nbytes: Optional[*int*] = None*) → Optional[*bytes*]
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(*self*, *buf:* _typing.WriteableBuffer) → Optional[*int*]
Read bytes into the buf. Read at most len(buf) bytes.

Returns number of bytes read and stored into buf

Return type *int* or None (on a non-blocking error)

readline(*self*) → *bytes*
Read a line, ending in a newline character.

Returns the line read

Return type *int* or None

reset_input_buffer(*self*) → None
Discard any unread characters in the input buffer.

deinit(*self*) → None
Disable permanently.

class _bleio.**Connection**
A BLE connection to another device. Used to discover and interact with services on the other device.

Usage:

```
import _bleio

my_entry = None
for entry in _bleio.adapter.scan(2.5):
    if entry.name is not None and entry.name == 'InterestingPeripheral':
        my_entry = entry
        break

if not my_entry:
    raise Exception("'InterestingPeripheral' not found")

connection = _bleio.adapter.connect(my_entry.address, timeout=10)
```

Connections cannot be made directly. Instead, to initiate a connection use *Adapter.connect*. Connections may also be made when another device initiates a connection. To use a Connection created by a peer, read the *Adapter.connections* property.

connected :bool
True if connected to the remote peer.

paired :bool
True if paired to the remote peer.

connection_interval :float
Time between transmissions in milliseconds. Will be multiple of 1.25ms. Lower numbers increase speed and decrease latency but increase power consumption.

When setting connection_interval, the peer may reject the new interval and *connection_interval* will then remain the same.

Apple has additional guidelines that dictate should be a multiple of 15ms except if HID is available. When HID is available Apple devices may accept 11.25ms intervals.

max_packet_length :int
The maximum number of data bytes that can be sent in a single transmission, not including overhead bytes.

This is the maximum number of bytes that can be sent in a notification, which must be sent in a single packet. But for a regular characteristic read or write, may be sent in multiple packets, so this limit does not apply.

disconnect(*self*) → None
Disconnects from the remote peripheral. Does nothing if already disconnected.

pair(*self*, *, *bond:* bool = *True*) → None
Pair to the peer to improve security.

discover_remote_services(*self*, *service_uuids_whitelist: Optional[Iterable[*UUID*]] = None*) → Tuple[*Service*, Ellipsis]

Do BLE discovery for all services or for the given service UUIDS, to find their handles and characteristics, and return the discovered services. *Connection.connected* must be True.

Parameters **service_uuids_whitelist** (*iterable*) – an iterable of *UUID* objects for the services provided by the peripheral that you want to use.

The peripheral may provide more services, but services not listed are ignored and will not be returned.

If service_uuids_whitelist is None, then all services will undergo discovery, which can be slow.

If the service UUID is 128-bit, or its characteristic UUID's are 128-bit, you you must have already created a *UUID* object for that UUID in order for the service or characteristic to be discovered. Creating the UUID causes the UUID to be registered for use. (This restriction may be lifted in the future.)

Returns A tuple of *_bleio.Service* objects provided by the remote peripheral.

class _bleio.**Descriptor**

Stores information about a BLE descriptor.

Descriptors are attached to BLE characteristics and provide contextual information about the characteristic.

There is no regular constructor for a Descriptor. A new local Descriptor can be created and attached to a Characteristic by calling *add_to_characteristic()*. Remote Descriptor objects are created by *Connection.discover_remote_services()* as part of remote Characteristics in the remote Services that are discovered.

uuid :UUID

The descriptor uuid. (read-only)

characteristic :Characteristic

The Characteristic this Descriptor is a part of.

value :bytearray

The value of this descriptor.

classmethod add_to_characteristic(*cls*, *characteristic:* Characteristic, *uuid:* UUID, ***, *read_perm:* int = *Attribute.OPEN*, *write_perm:* int = *Attribute.OPEN*, *max_length:* int = *20*, *fixed_length:* bool = *False*, *initial_value:* _typing.ReadableBuffer = *b''*) → *Descriptor*

Create a new Descriptor object, and add it to this Service.

Parameters

- **characteristic** (Characteristic) – The characteristic that will hold this descriptor
- **uuid** (UUID) – The uuid of the descriptor
- **read_perm** (int) – Specifies whether the descriptor can be read by a client, and if so, which security mode is required. Must be one of the integer values *Attribute.NO_ACCESS*, *Attribute.OPEN*, *Attribute.ENCRYPT_NO_MITM*, *Attribute.ENCRYPT_WITH_MITM*, *Attribute.LESC_ENCRYPT_WITH_MITM*, *Attribute.SIGNED_NO_MITM*, or *Attribute.SIGNED_WITH_MITM*.
- **write_perm** (int) – Specifies whether the descriptor can be written by a client, and if so, which security mode is required. Values allowed are the same as read_perm.

- **max_length** (int) – Maximum length in bytes of the descriptor value. The maximum allowed is is 512, or possibly 510 if fixed_length is False. The default, 20, is the maximum number of data bytes that fit in a single BLE 4.x ATT packet.
- **fixed_length** (bool) – True if the descriptor value is of fixed length.
- **initial_value** (ReadableBuffer) – The initial value for this descriptor.

Returns the new Descriptor.

class _bleio.**PacketBuffer**(*characteristic: Characteristic*, *, *buffer_size: int*, *max_packet_size: Optional[int] = None*)

Accumulates a Characteristic's incoming packets in a FIFO buffer and facilitates packet aware outgoing writes. A packet's size is either the characteristic length or the maximum transmission unit (MTU) minus overhead, whichever is smaller. The MTU can change so check *incoming_packet_length* and *outgoing_packet_length* before creating a buffer to store data.

When we're the server, we ignore all connections besides the first to subscribe to notifications.

Monitor the given Characteristic. Each time a new value is written to the Characteristic add the newly-written bytes to a FIFO buffer.

Monitor the given Characteristic. Each time a new value is written to the Characteristic add the newly-written packet of bytes to a FIFO buffer.

Parameters

- **characteristic** (Characteristic) – The Characteristic to monitor. It may be a local Characteristic provided by a Peripheral Service, or a remote Characteristic in a remote Service that a Central has connected to.
- **buffer_size** (int) – Size of ring buffer (in packets of the Characteristic's maximum length) that stores incoming packets coming from the peer.
- **max_packet_size** (int) – Maximum size of packets. Overrides value from the characteristic. (Remote characteristics may not have the correct length.)

incoming_packet_length :int

Maximum length in bytes of a packet we are reading.

outgoing_packet_length :int

Maximum length in bytes of a packet we are writing.

readinto(*self*, *buf: _typing.WriteableBuffer*) → *int*

Reads a single BLE packet into the buf. Raises an exception if the next packet is longer than the given buffer. Use *incoming_packet_length* to read the maximum length of a single packet.

Returns number of bytes read and stored into buf

Return type *int*

write(*self*, *data: _typing.ReadableBuffer*, *, *header: Optional[bytes] = None*) → *int*

Writes all bytes from data into the same outgoing packet. The bytes from header are included before data when the pending packet is currently empty.

This does not block until the data is sent. It only blocks until the data is pending.

Returns number of bytes written. May include header bytes when packet is empty.

Return type *int*

deinit(*self*) → None

Disable permanently.

class _bleio.**ScanEntry**

Encapsulates information about a device that was received during scanning. It can be advertisement or scan response data. This object may only be created by a *_bleio.ScanResults*: it has no user-visible constructor.

Cannot be instantiated directly. Use *_bleio.Adapter.start_scan*.

address :Address

The address of the device (read-only), of type *_bleio.Address*.

advertisement_bytes :bytes

All the advertisement data present in the packet, returned as a `bytes` object. (read-only)

rssi :int

The signal strength of the device at the time of the scan, in integer dBm. (read-only)

connectable :bool

True if the device can be connected to. (read-only)

scan_response :bool

True if the entry was a scan response. (read-only)

matches(*self*, *prefixes:* ScanEntry, *, *all:* bool = *True*) → *bool*

Returns True if the ScanEntry matches all prefixes when `all` is True. This is stricter than the scan filtering which accepts any advertisements that match any of the prefixes where all is False.

class _bleio.**ScanResults**

Iterates over advertising data received while scanning. This object is always created by a *_bleio.Adapter*: it has no user-visible constructor.

Cannot be instantiated directly. Use *_bleio.Adapter.start_scan*.

__iter__(*self*) → Iterator[*ScanEntry*]

Returns itself since it is the iterator.

__next__(*self*) → *ScanEntry*

Returns the next *_bleio.ScanEntry*. Blocks if none have been received and scanning is still active. Raises *StopIteration* if scanning is finished and no other results are available.

class _bleio.**Service**(*uuid:* UUID, *, *secondary:* bool = *False*)

Stores information about a BLE service and its characteristics.

Create a new Service identified by the specified UUID. It can be accessed by all connections. This is known as a Service server. Client Service objects are created via *Connection.discover_remote_services*.

To mark the Service as secondary, pass `True` as *secondary*.

Parameters

- **uuid** (`UUID`) – The uuid of the service
- **secondary** (`bool`) – If the service is a secondary one

Returns the new Service

characteristics :Tuple[Characteristic, Ellipsis]

A tuple of *Characteristic* designating the characteristics that are offered by this service. (read-only)

remote :bool

True if this is a service provided by a remote device. (read-only)

secondary :bool

True if this is a secondary service. (read-only)

uuid :Optional[UUID]
The UUID of this service. (read-only)

Will be `None` if the 128-bit UUID for this service is not known.

class _bleio.**UUID**(*value: Union[*int, _typing.ReadableBuffer, str*]*)
A 16-bit or 128-bit UUID. Can be used for services, characteristics, descriptors and more.

Create a new UUID or UUID object encapsulating the uuid value. The value can be one of:

- an *int* value in range 0 to 0xFFFF (Bluetooth SIG 16-bit UUID)
- a buffer object (bytearray, bytes) of 16 bytes in little-endian order (128-bit UUID)
- a string of hex digits of the form 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'

Creating a 128-bit UUID registers the UUID with the onboard BLE software, and provides a temporary 16-bit UUID that can be used in place of the full 128-bit UUID.

Parameters **value** (*int, ReadableBuffer or str*) – The uuid value to encapsulate

uuid16 :int
The 16-bit part of the UUID. (read-only)

Type *int*

uuid128 :bytes
The 128-bit value of the UUID Raises AttributeError if this is a 16-bit UUID. (read-only)

Type *bytes*

size :int
128 if this UUID represents a 128-bit vendor-specific UUID. 16 if this UUID represents a 16-bit Bluetooth SIG assigned UUID. (read-only) 32-bit UUIDs are not currently supported.

Type *int*

pack_into(*self*, *buffer:* _typing.WriteableBuffer, *offset:* int = *0*) → None
Packs the UUID into the given buffer at the given offset.

__eq__(*self*, *other:* object) → *bool*
Two UUID objects are equal if their values match and they are both 128-bit or both 16-bit.

_eve – Low-level BridgeTek EVE bindings

The *_eve* module provides a class _EVE which contains methods for constructing EVE command buffers and appending basic graphics commands.

class _eve.**_EVE**

register(*self*, *o:* object) → None

flush(*self*) → None
Send any queued drawing commands directly to the hardware.

Parameters **width** (*int*) – The width of the grid in tiles, or 1 for sprites.

cc(*self*, *b:* _typing.ReadableBuffer) → None
Append bytes to the command FIFO.

Parameters **b** (*ReadableBuffer*) – The bytes to add

AlphaFunc (*self*, *func:* int, *ref:* int) → None
Set the alpha test function

Parameters

- **func** (int) – specifies the test function, one of NEVER, LESS, LEQUAL, GREATER, GEQUAL, EQUAL, NOTEQUAL, or ALWAYS. Range 0-7. The initial value is ALWAYS(7)
- **ref** (int) – specifies the reference value for the alpha test. Range 0-255. The initial value is 0

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

Begin (*self*, *prim:* int) → None
Begin drawing a graphics primitive

Parameters **prim** (int) – graphics primitive.

Valid primitives are BITMAPS, POINTS, LINES, LINE_STRIP, EDGE_STRIP_R, EDGE_STRIP_L, EDGE_STRIP_A, EDGE_STRIP_B and RECTS.

BitmapExtFormat (*self*, *format:* int) → None
Set the bitmap format

Parameters **format** (int) – bitmap pixel format.

BitmapHandle (*self*, *handle:* int) → None
Set the bitmap handle

Parameters **handle** (int) – bitmap handle. Range 0-31. The initial value is 0

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

BitmapLayoutH (*self*, *linestride:* int, *height:* int) → None
Set the source bitmap memory format and layout for the current handle. high bits for large bitmaps

Parameters

- **linestride** (int) – high part of bitmap line stride, in bytes. Range 0-7
- **height** (int) – high part of bitmap height, in lines. Range 0-3

BitmapLayout (*self*, *format:* int, *linestride:* int, *height:* int) → None
Set the source bitmap memory format and layout for the current handle

Parameters

- **format** (int) – bitmap pixel format, or GLFORMAT to use BITMAP_EXT_FORMAT instead. Range 0-31
- **linestride** (int) – bitmap line stride, in bytes. Range 0-1023
- **height** (int) – bitmap height, in lines. Range 0-511

BitmapSizeH (*self*, *width:* int, *height:* int) → None
Set the screen drawing of bitmaps for the current handle. high bits for large bitmaps

Parameters

- **width** (int) – high part of drawn bitmap width, in pixels. Range 0-3
- **height** (int) – high part of drawn bitmap height, in pixels. Range 0-3

BitmapSize (*self*, *filter:* int, *wrapx:* int, *wrapy:* int, *width:* int, *height:* int) → None
Set the screen drawing of bitmaps for the current handle

Parameters

- **filter** (int) – bitmap filtering mode, one of NEAREST or BILINEAR. Range 0-1
- **wrapx** (int) – bitmap x wrap mode, one of REPEAT or BORDER. Range 0-1
- **wrapy** (int) – bitmap y wrap mode, one of REPEAT or BORDER. Range 0-1
- **width** (int) – drawn bitmap width, in pixels. Range 0-511
- **height** (int) – drawn bitmap height, in pixels. Range 0-511

BitmapSource(*self*, *addr:* int) → None
Set the source address for bitmap graphics

Parameters **addr** (int) – Bitmap start address, pixel-aligned. May be in SRAM or flash. Range 0-16777215

BitmapSwizzle(*self*, *r:* int, *g:* int, *b:* int, *a:* int) → None
Set the source for the r,g,b and a channels of a bitmap

Parameters

- **r** (int) – red component source channel. Range 0-7
- **g** (int) – green component source channel. Range 0-7
- **b** (int) – blue component source channel. Range 0-7
- **a** (int) – alpha component source channel. Range 0-7

BitmapTransformA(*self*, *p:* int, *v:* int) → None
Set the a component of the bitmap transform matrix

Parameters

- **p** (int) – precision control: 0 is 8.8, 1 is 1.15. Range 0-1. The initial value is 0
- **v** (int) – The a component of the bitmap transform matrix, in signed 8.8 or 1.15 bit fixed-point form. Range 0-131071. The initial value is 256

The initial value is **p** = 0, **v** = 256. This represents the value 1.0.

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

BitmapTransformB(*self*, *p:* int, *v:* int) → None
Set the b component of the bitmap transform matrix

Parameters

- **p** (int) – precision control: 0 is 8.8, 1 is 1.15. Range 0-1. The initial value is 0
- **v** (int) – The b component of the bitmap transform matrix, in signed 8.8 or 1.15 bit fixed-point form. Range 0-131071. The initial value is 0

The initial value is **p** = 0, **v** = 0. This represents the value 0.0.

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

BitmapTransformC(*self*, *v:* int) → None
Set the c component of the bitmap transform matrix

Parameters **v** (int) – The c component of the bitmap transform matrix, in signed 15.8 bit fixed-point form. Range 0-16777215. The initial value is 0

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

BitmapTransformD (*self*, *p:* int, *v:* int) → None
Set the d component of the bitmap transform matrix

Parameters

- **p** (int) – precision control: 0 is 8.8, 1 is 1.15. Range 0-1. The initial value is 0
- **v** (int) – The d component of the bitmap transform matrix, in signed 8.8 or 1.15 bit fixed-point form. Range 0-131071. The initial value is 0

The initial value is **p** = 0, **v** = 0. This represents the value 0.0.

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

BitmapTransformE (*self*, *p:* int, *v:* int) → None
Set the e component of the bitmap transform matrix

Parameters

- **p** (int) – precision control: 0 is 8.8, 1 is 1.15. Range 0-1. The initial value is 0
- **v** (int) – The e component of the bitmap transform matrix, in signed 8.8 or 1.15 bit fixed-point form. Range 0-131071. The initial value is 256

The initial value is **p** = 0, **v** = 256. This represents the value 1.0.

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

BitmapTransformF (*self*, *v:* int) → None
Set the f component of the bitmap transform matrix

Parameters **v** (int) – The f component of the bitmap transform matrix, in signed 15.8 bit fixed-point form. Range 0-16777215. The initial value is 0

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

BlendFunc (*self*, *src:* int, *dst:* int) → None
Set pixel arithmetic

Parameters

- **src** (int) – specifies how the source blending factor is computed. One of ZERO, ONE, SRC_ALPHA, DST_ALPHA, ONE_MINUS_SRC_ALPHA or ONE_MINUS_DST_ALPHA. Range 0-7. The initial value is SRC_ALPHA(2)
- **dst** (int) – specifies how the destination blending factor is computed, one of the same constants as **src**. Range 0-7. The initial value is ONE_MINUS_SRC_ALPHA(4)

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

Call (*self*, *dest:* int) → None
Execute a sequence of commands at another location in the display list

Parameters **dest** (int) – display list address. Range 0-65535

Cell (*self*, *cell:* int) → None
Set the bitmap cell number for the vertex2f command

Parameters **cell** (int) – bitmap cell number. Range 0-127. The initial value is 0

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

ClearColorA(*self*, *alpha:* int) → None
Set clear value for the alpha channel

Parameters **alpha** (int) – alpha value used when the color buffer is cleared. Range 0-255. The initial value is 0

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

ClearColorRGB(*self*, *red:* int, *green:* int, *blue:* int) → None
Set clear values for red, green and blue channels

Parameters

- **red** (int) – red value used when the color buffer is cleared. Range 0-255. The initial value is 0
- **green** (int) – green value used when the color buffer is cleared. Range 0-255. The initial value is 0
- **blue** (int) – blue value used when the color buffer is cleared. Range 0-255. The initial value is 0

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

Clear(*self*, *c:* int, *s:* int, *t:* int) → None
Clear buffers to preset values

Parameters

- **c** (int) – clear color buffer. Range 0-1
- **s** (int) – clear stencil buffer. Range 0-1
- **t** (int) – clear tag buffer. Range 0-1

ClearStencil(*self*, *s:* int) → None
Set clear value for the stencil buffer

Parameters **s** (int) – value used when the stencil buffer is cleared. Range 0-255. The initial value is 0

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

ClearTag(*self*, *s:* int) → None
Set clear value for the tag buffer

Parameters **s** (int) – value used when the tag buffer is cleared. Range 0-255. The initial value is 0

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

ColorA(*self*, *alpha:* int) → None
Set the current color alpha

Parameters **alpha** (int) – alpha for the current color. Range 0-255. The initial value is 255

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

ColorMask (*self*, *r:* int, *g:* int, *b:* int, *a:* int) → None
Enable and disable writing of frame buffer color components

Parameters

- **r** (int) – allow updates to the frame buffer red component. Range 0-1. The initial value is 1
- **g** (int) – allow updates to the frame buffer green component. Range 0-1. The initial value is 1
- **b** (int) – allow updates to the frame buffer blue component. Range 0-1. The initial value is 1
- **a** (int) – allow updates to the frame buffer alpha component. Range 0-1. The initial value is 1

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

ColorRGB (*self*, *red:* int, *green:* int, *blue:* int) → None
Set the drawing color

Parameters

- **red** (int) – red value for the current color. Range 0-255. The initial value is 255
- **green** (int) – green for the current color. Range 0-255. The initial value is 255
- **blue** (int) – blue for the current color. Range 0-255. The initial value is 255

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

Display (*self*) → None
End the display list

End (*self*) → None
End drawing a graphics primitive

Vertex2ii() and *Vertex2f()* calls are ignored until the next *Begin()*.

Jump (*self*, *dest:* int) → None
Execute commands at another location in the display list

Parameters **dest** (int) – display list address. Range 0-65535

Macro (*self*, *m:* int) → None
Execute a single command from a macro register

Parameters **m** (int) – macro register to read. Range 0-1

Nop (*self*) → None
No operation

PaletteSource (*self*, *addr:* int) → None
Set the base address of the palette

Parameters **addr** (int) – Address in graphics SRAM, 2-byte aligned. Range 0-4194303. The initial value is 0

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

RestoreContext (*self*) → None
Restore the current graphics context from the context stack

Return (*self*) → None
Return from a previous call command

SaveContext (*self*) → None
Push the current graphics context on the context stack

ScissorSize (*self*, *width:* int, *height:* int) → None
Set the size of the scissor clip rectangle

Parameters

- **width** (int) – The width of the scissor clip rectangle, in pixels. Range 0-4095. The initial value is hsize
- **height** (int) – The height of the scissor clip rectangle, in pixels. Range 0-4095. The initial value is 2048

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

ScissorXY (*self*, *x:* int, *y:* int) → None
Set the top left corner of the scissor clip rectangle

Parameters

- **x** (int) – The x coordinate of the scissor clip rectangle, in pixels. Range 0-2047. The initial value is 0
- **y** (int) – The y coordinate of the scissor clip rectangle, in pixels. Range 0-2047. The initial value is 0

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

StencilFunc (*self*, *func:* int, *ref:* int, *mask:* int) → None
Set function and reference value for stencil testing

Parameters

- **func** (int) – specifies the test function, one of NEVER, LESS, LEQUAL, GREATER, GEQUAL, EQUAL, NOTEQUAL, or ALWAYS. Range 0-7. The initial value is ALWAYS(7)
- **ref** (int) – specifies the reference value for the stencil test. Range 0-255. The initial value is 0
- **mask** (int) – specifies a mask that is ANDed with the reference value and the stored stencil value. Range 0-255. The initial value is 255

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

StencilMask (*self*, *mask:* int) → None
Control the writing of individual bits in the stencil planes

Parameters **mask** (int) – the mask used to enable writing stencil bits. Range 0-255. The initial value is 255

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

StencilOp (*self*, *sfail:* int, *spass:* int) → None
Set stencil test actions

Parameters

- **sfail** (int) – specifies the action to take when the stencil test fails, one of KEEP, ZERO, REPLACE, INCR, INCR_WRAP, DECR, DECR_WRAP, and INVERT. Range 0-7. The initial value is KEEP(1)
- **spass** (int) – specifies the action to take when the stencil test passes, one of the same constants as **sfail**. Range 0-7. The initial value is KEEP(1)

These values are part of the graphics context and are saved and restored by *SaveContext()* and *RestoreContext()*.

TagMask (*self*, *mask:* int) → None
Control the writing of the tag buffer

Parameters **mask** (int) – allow updates to the tag buffer. Range 0-1. The initial value is 1

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

Tag (*self*, *s:* int) → None
Set the current tag value

Parameters **s** (int) – tag value. Range 0-255. The initial value is 255

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

Vertex2ii (*self*, *x:* int, *y:* int, *handle:* int, *cell:* int) → None

Parameters

- **x** (int) – x-coordinate in pixels. Range 0-511
- **y** (int) – y-coordinate in pixels. Range 0-511
- **handle** (int) – bitmap handle. Range 0-31
- **cell** (int) – cell number. Range 0-127

This method is an alternative to *Vertex2f()*.

Vertex2f (*self*, *b:* float) → None
Draw a point.

Parameters

- **x** (float) – pixel x-coordinate
- **y** (float) – pixel y-coordinate

LineWidth (*self*, *width:* float) → None
Set the width of rasterized lines

Parameters **width** (float) – line width in pixels. Range 0-511. The initial value is 1

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

PointSize (*self*, *size:* float) → None
Set the diameter of rasterized points

Parameters **size** (float) – point diameter in pixels. Range 0-1023. The initial value is 1

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

VertexTranslateX (*self*, *x:* float) → None
Set the vertex transformation's x translation component

Parameters **x** (float) – signed x-coordinate in pixels. Range ±4095. The initial value is 0

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

VertexTranslateY(*self*, *y:* float) → None
Set the vertex transformation's y translation component

Parameters **y** (float) – signed y-coordinate in pixels. Range ±4095. The initial value is 0

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

VertexFormat(*self*, *frac:* int) → None
Set the precision of vertex2f coordinates

Parameters **frac** (int) – Number of fractional bits in X,Y coordinates, 0-4. Range 0-7. The initial value is 4

This value is part of the graphics context and is saved and restored by *SaveContext()* and *RestoreContext()*.

cmd0(*self*, *n:* int) → None
Append the command word n to the FIFO

Parameters **n** (int) – The command code

This method is used by the eve module to efficiently add commands to the FIFO.

cmd(*self*, *n:* int, *fmt:* str, *args: Tuple[*str, *Ellipsis]*) → None
Append a command packet to the FIFO.

Parameters

- **n** (int) – The command code
- **fmt** (str) – The command format *struct* layout
- **args** (tuple *(str, ..)*) – The command's arguments

Supported format codes: h, H, i, I.

This method is used by the eve module to efficiently add commands to the FIFO.

_pew – LED matrix driver

class _pew.**PewPew**(*buffer:* _typing.ReadableBuffer, *rows: List[*digitalio.DigitalInOut*]*, *cols: List[*digitalio.DigitalInOut*]*, *buttons:* digitalio.DigitalInOut)
This is an internal module to be used by the pew.py library from https://github.com/pewpew-game/pew-pewpew-standalone-10.x to handle the LED matrix display and buttons on the pewpew10 board.

Usage:

```
This singleton class is instantiated by the ``pew`` library, and
used internally by it. All user-visible interactions are done through
that library.
```

Initializes matrix scanning routines.

The buffer is a 64 byte long bytearray that stores what should be displayed on the matrix. rows and cols are both lists of eight DigitalInputOutput objects that are connected to the matrix rows and columns. buttons is a DigitalInputOutput object that is connected to the common side of all buttons (the other sides of the buttons are connected to rows of the matrix).

`_pixelbuf` – A fast RGB(W) pixel buffer library for like NeoPixel and DotStar

The *_pixelbuf* module provides the *PixelBuf* class to accelerate RGB(W) strip/matrix manipulation, such as DotStar and Neopixel.

Byteorders are configured with strings, such as "RGB" or "RGBD".

class _pixelbuf.**PixelBuf**(*size:* int, *, *byteorder:* str = *'BGR'*, *brightness:* float = *0*, *auto_write:* bool = *False*, *header:* _typing.ReadableBuffer = *b''*, *trailer:* _typing.ReadableBuffer = *b''*)

A fast RGB[W] pixel buffer for LED and similar devices.

Create a PixelBuf object of the specified size, byteorder, and bits per pixel.

When brightness is less than 1.0, a second buffer will be used to store the color values before they are adjusted for brightness.

When `P` (PWM duration) is present as the 4th character of the byteorder string, the 4th value in the tuple/list for a pixel is the individual pixel brightness (0.0-1.0) and will enable a Dotstar compatible 1st byte for each pixel.

Parameters

- **size** (`int`) – Number of pixels
- **byteorder** (`str`) – Byte order string (such as "RGB", "RGBW" or "PBGR")
- **brightness** (`float`) – Brightness (0 to 1.0, default 1.0)
- **auto_write** (`bool`) – Whether to automatically write pixels (Default False)
- **header** (`ReadableBuffer`) – Sequence of bytes to always send before pixel values.
- **trailer** (`ReadableBuffer`) – Sequence of bytes to always send after pixel values.

bpp :int
The number of bytes per pixel in the buffer (read-only)

brightness :float
Float value between 0 and 1. Output brightness.

When brightness is less than 1.0, a second buffer will be used to store the color values before they are adjusted for brightness.

auto_write :bool
Whether to automatically write the pixels after each update.

byteorder :str
byteorder string for the buffer (read-only)

show(*self*) → None
Transmits the color data to the pixels so that they are shown. This is done automatically when *auto_write* is True.

fill(*self*, *color: Union[*int, *Tuple[*int, int, int*], Tuple[*int, int, int, float*]]*) → None
Fills the given pixelbuf with the given color.

__getitem__(*self*, *index:* slice) → Union[Tuple[Tuple[*int*, *int*, *int*], Ellipsis], Tuple[Tuple[*int*, *int*, *int*, *float*], Ellipsis]]
__getitem__(*self*, *index:* int) → Union[Tuple[*int*, *int*, *int*], Tuple[*int*, *int*, *int*, *float*]]
Returns the pixel value at the given index as a tuple of (Red, Green, Blue[, White]) values between 0 and 255. When in PWM (DotStar) mode, the 4th tuple value is a float of the pixel intensity from 0-1.0.

__setitem__(*self*, *index:* slice, *value: Tuple[Union[*int, *Tuple[*float, *Ellipsis], List[*float*]], Ellipsis]*) → None
__setitem__(*self*, *index:* slice, *value: List[Union[*int, *Tuple[*float, *Ellipsis], List[*float*]]]*) → None

__setitem__(*self*, *index:* int, *value: Union[*int, *Tuple[*float, *Ellipsis], List[*float*]]*) → None
Sets the pixel value at the given index. Value can either be a tuple or integer. Tuples are The individual (Red, Green, Blue[, White]) values between 0 and 255. If given an integer, the red, green and blue values are packed into the lower three bytes (0xRRGGBB). For RGBW byteorders, if given only RGB values either as an int or as a tuple, the white value is used instead when the red, green, and blue values are the same.

_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>`_.

_stage.**render**(*x0:* int, *y0:* int, *x1:* int, *y1:* int, *layers: List[*Layer*]*, *buffer:* _typing.WriteableBuffer, *display:* displayio.Display, *scale:* int, *background:* int) → None
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[Layer]) – A list of the *Layer* objects.
- **buffer** (WriteableBuffer) – A buffer to use for rendering.
- **display** (Display) – The display to use.
- **scale** (int) – How many times should the image be scaled up.
- **background** (int) – What color to display when nothing is there.

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.

class _stage.**Layer**(*width:* int, *height:* int, *graphic:* _typing.ReadableBuffer, *palette:* _typing.ReadableBuffer, *grid:* _typing.ReadableBuffer)
Keep information about a single layer of graphics

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** (ReadableBuffer) – The graphic data of the tiles.
- **palette** (ReadableBuffer) – The color palette to be used.
- **grid** (ReadableBuffer) – 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(*self*, *x:* int, *y:* int) → None
Set the offset of the layer to the specified values.

frame(*self*, *frame:* int, *rotation:* int) → None
Set the animation frame of the sprite, and optionally rotation its graphic.

class _stage.**Text**(*width:* int, *height:* int, *font:* _typing.ReadableBuffer, *palette:* _typing.ReadableBuffer, *chars:* _typing.ReadableBuffer)
Keep information about a single grid of text

Keep internal information about a grid 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** (`ReadableBuffer`) – The font data of the characters.
- **palette** (`ReadableBuffer`) – The color palette to be used.
- **chars** (`ReadableBuffer`) – 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(*self*, *x:* int, *y:* int) → None
Set the offset of the text to the specified values.

_typing – Types for the C-level protocols

_typing.**ReadableBuffer**
Classes that implement the readable buffer protocol

- *bytes*
- *bytearray*
- *memoryview*
- *array.array*
- *ulab.ndarray*
- *rgbmatrix.RGBMatrix*

_typing.**WriteableBuffer**
Classes that implement the writeable buffer protocol

- *bytearray*
- *memoryview*
- *array.array*
- *ulab.ndarray*
- *rgbmatrix.RGBMatrix*

_typing.**AudioSample**
Classes that implement the audiosample protocol

- *audiocore.WaveFile*
- *audiocore.RawSample*
- *audiomixer.Mixer*
- *audiomp3.MP3Decoder*

- *synthio.MidiTrack*

You can play these back with *audioio.AudioOut*, *audiobusio.I2SOut* or *audiopwmio.PWMAudioOut*.

_typing.**FrameBuffer**
Classes that implement the framebuffer protocol

- *rgbmatrix.RGBMatrix*

_typing.**Alarm**
Classes that implement alarms for sleeping and asynchronous notification.

- *alarm.pin.PinAlarm*
- *alarm.time.TimeAlarm*

You can use these alarms to wake up from light or deep sleep.

adafruit_bus_device – Hardware accelerated external bus access

The I2CDevice and SPIDevice helper classes make managing transaction state on a bus easy. For example, they manage locking the bus to prevent other concurrent access. For SPI devices, it manages the chip select and protocol changes such as mode. For I2C, it manages the device address.

class adafruit_bus_device.**I2CDevice**(*i2c:* busio.I2C, *device_address:* int, *probe:* bool = *True*)
I2C Device Manager

Represents a single I2C device and manages locking the bus and the device address.

Parameters

- **i2c** (I2C) – The I2C bus the device is on
- **device_address** (int) – The 7 bit device address
- **probe** (bool) – Probe for the device upon object creation, default is true

Example:

```
import busio
from board import *
from adafruit_bus_device.i2c_device import I2CDevice
with busio.I2C(SCL, SDA) as i2c:
    device = I2CDevice(i2c, 0x70)
    bytes_read = bytearray(4)
    with device:
        device.readinto(bytes_read)
    # A second transaction
    with device:
        device.write(bytes_read)
```

__enter__(*self*) → *I2CDevice*
Context manager entry to lock bus.

__exit__(*self*) → None
Automatically unlocks the bus on exit.

readinto(*self*, *buf:* _typing.WriteableBuffer, *, *start:* int = *0*, *end: Optional[*int*]* = *None*) → None
Read into `buf` from the device. The number of bytes read will be the length of `buf`. If `start` or `end` is provided, then the buffer will be sliced as if `buf[start:end]`. This will not cause an allocation like `buf[start:end]` will so it saves memory.

Parameters

- **buf** (bytearray) – buffer to write into
- **start** (int) – Index to start writing at
- **end** (int) – Index to write up to but not include; if None, use len(buf)

write(*self*, *buf:* _typing.ReadableBuffer, *, *start:* int = *0*, *end: Optional[*int*] = None*) → None

Write the bytes from buffer to the device, then transmit a stop bit. 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.

Parameters

- **buf** (bytearray) – buffer containing the bytes to write
- **start** (int) – Index to start writing from
- **end** (int) – Index to read up to but not include; if None, use len(buf)

write_then_readinto(*self*, *out_buffer:* _typing.WriteableBuffer, *in_buffer:* _typing.ReadableBuffer, *, *out_start:* int = *0*, *out_end: Optional[*int*] = None*, *in_start:* int = *0*, *in_end: Optional[*int*] = None*) → None

Write the bytes from out_buffer to the device, then immediately reads into in_buffer from the device. The number of bytes read will be the length of in_buffer. If out_start or out_end is provided, then the output buffer will be sliced as if out_buffer[out_start:out_end]. This will not cause an allocation like buffer[out_start:out_end] will so it saves memory. If in_start or in_end is provided, then the input buffer will be sliced as if in_buffer[in_start:in_end]. This will not cause an allocation like in_buffer[in_start:in_end] will so it saves memory.

Parameters

- **out_buffer** (bytearray) – buffer containing the bytes to write
- **in_buffer** (bytearray) – buffer containing the bytes to read into
- **out_start** (int) – Index to start writing from
- **out_end** (int) – Index to read up to but not include; if None, use len(out_buffer)
- **in_start** (int) – Index to start writing at
- **in_end** (int) – Index to write up to but not include; if None, use len(in_buffer)

class adafruit_bus_device.**SPIDevice**(*spi:* busio.SPI, *chip_select:* microcontroller.Pin, *, *baudrate:* int = *100000*, *polarity:* int = *0*, *phase:* int = *0*, *extra_clocks:* int = *0*)

SPI Device Manager

Represents a single SPI device and manages locking the bus and the device address.

Parameters

- **spi** (SPI) – The SPI bus the device is on
- **chip_select** (DigitalInOut) – The chip select pin object that implements the DigitalInOut API.
- **extra_clocks** (int) – The minimum number of clock cycles to cycle the bus after CS is high. (Used for SD cards.)

Example:

```
import busio
import digitalio
from board import *
from adafruit_bus_device.spi_device import SPIDevice
with busio.SPI(SCK, MOSI, MISO) as spi_bus:
    cs = digitalio.DigitalInOut(D10)
    device = SPIDevice(spi_bus, cs)
    bytes_read = bytearray(4)
    # The object assigned to spi in the with statements below
    # is the original spi_bus object. We are using the busio.SPI
    # operations busio.SPI.readinto() and busio.SPI.write().
    with device as spi:
        spi.readinto(bytes_read)
    # A second transaction
    with device as spi:
        spi.write(bytes_read)
```

__enter__(*self*) → *busio.SPI*
Starts a SPI transaction by configuring the SPI and asserting chip select.

__exit__(*self*) → None
Ends a SPI transaction by deasserting chip select. See *Lifetime and ContextManagers* for more info.

aesio – AES encryption routines

The *AES* module contains classes used to implement encryption and decryption. It aims to be low overhead in terms of memory.

class aesio.**AES**(*key:* _typing.ReadableBuffer, *mode:* int = *0*, *iv: Optional[*_typing.ReadableBuffer*]* = *None*, *segment_size:* int = *8*)
Encrypt and decrypt AES streams

Create a new AES state with the given key.

Parameters

- **key** (ReadableBuffer) – A 16-, 24-, or 32-byte key
- **mode** (int) – AES mode to use. One of: AES.MODE_ECB, AES.MODE_CBC, or AES.MODE_CTR
- **iv** (ReadableBuffer) – Initialization vector to use for CBC or CTR mode

Additional arguments are supported for legacy reasons.

Encrypting a string:

```
import aesio
from binascii import hexlify

key = b'Sixteen byte key'
inp = b'CircuitPython!!!' # Note: 16-bytes long
outp = bytearray(len(inp))
cipher = aesio.AES(key, aesio.mode.MODE_ECB)
cipher.encrypt_into(inp, outp)
hexlify(outp)
```

encrypt_into(*self*, *src:* _typing.ReadableBuffer, *dest:* _typing.WriteableBuffer) → None
Encrypt the buffer from `src` into `dest`.

For ECB mode, the buffers must be 16 bytes long. For CBC mode, the buffers must be a multiple of 16 bytes, and must be equal length. For CTX mode, there are no restrictions.

decrypt_into(*self*, *src:* _typing.ReadableBuffer, *dest:* _typing.WriteableBuffer) → None
Decrypt the buffer from `src` into `dest`. For ECB mode, the buffers must be 16 bytes long. For CBC mode, the buffers must be a multiple of 16 bytes, and must be equal length. For CTX mode, there are no restrictions.

alarm – Alarms and sleep

Provides alarms that trigger based on time intervals or on external events, such as pin changes. The program can simply wait for these alarms, or go to sleep and be awoken when they trigger.

There are two supported levels of sleep: light sleep and deep sleep.

Light sleep keeps sufficient state so the program can resume after sleeping. It does not shut down WiFi, BLE, or other communications, or ongoing activities such as audio playback. It reduces power consumption to the extent possible that leaves these continuing activities running. In some cases there may be no decrease in power consumption.

Deep sleep shuts down power to nearly all of the microcontroller including the CPU and RAM. This can save a more significant amount of power, but CircuitPython must restart `code.py` from the beginning when awakened.

For both light sleep and deep sleep, if CircuitPython is connected to a host computer, maintaining the connection takes priority and power consumption may not be reduced.

alarm.pin – Trigger an alarm when a pin changes state.

class alarm.pin.**PinAlarm**(*pin:* microcontroller.Pin, *value:* bool, *edge:* bool = *False*, *pull:* bool = *False*)
Create an alarm triggered by a *microcontroller.Pin* level. The alarm is not active until it is passed to an *alarm*-enabling function, such as *alarm.light_sleep_until_alarms()* or *alarm.exit_and_deep_sleep_until_alarms()*.

Parameters

- **pin** (microcontroller.Pin) – The pin to monitor. On some ports, the choice of pin may be limited due to hardware restrictions, particularly for deep-sleep alarms.
- **value** (bool) – When active, trigger when the pin value is high (`True`) or low (`False`). On some ports, multiple *PinAlarm* objects may need to have coordinated values for deep-sleep alarms.
- **edge** (bool) – If `True`, trigger only when there is a transition to the specified value of `value`. If `True`, if the alarm becomes active when the pin value already matches `value`, the alarm is not triggered: the pin must transition from `not value` to `value` to trigger the alarm. On some ports, edge-triggering may not be available, particularly for deep-sleep alarms.
- **pull** (bool) – Enable a pull-up or pull-down which pulls the pin to the level opposite that of `value`. For instance, if `value` is set to `True`, setting `pull` to `True` will enable a pull-down, to hold the pin low normally until an outside signal pulls it high.

pin :microcontroller.Pin
The trigger pin.

value :bool
The value on which to trigger.

alarm.time – Trigger an alarm when the specified time is reached.

class alarm.time.**TimeAlarm**(*monotonic_time: Optional[float] = None*, *epoch_time: Optional[int] = None*)

Create an alarm that will be triggered when *time.monotonic()* would equal monotonic_time, or when *time.time()* would equal epoch_time. Only one of the two arguments can be given. The alarm is not active until it is passed to an *alarm*-enabling function, such as *alarm.light_sleep_until_alarms()* or *alarm.exit_and_deep_sleep_until_alarms()*.

If the given time is in the past when sleep occurs, the alarm will be triggered immediately.

monotonic_time :float

When this time is reached, the alarm will trigger, based on the *time.monotonic()* clock. The time may be given as epoch_time in the constructor, but it is returned by this property only as a *time.monotonic()* time.

alarm.touch – Trigger an alarm when touch is detected.

class alarm.touch.**TouchAlarm**(**pin:* microcontroller.Pin)

Create an alarm that will be triggered when the given pin is touched. The alarm is not active until it is passed to an *alarm*-enabling function, such as *alarm.light_sleep_until_alarms()* or *alarm.exit_and_deep_sleep_until_alarms()*.

Parameters **pin** (microcontroller.Pin) – The pin to monitor. On some ports, the choice of pin may be limited due to hardware restrictions, particularly for deep-sleep alarms.

pin :microcontroller.Pin

The trigger pin.

alarm.**sleep_memory :SleepMemory**

Memory that persists during deep sleep. This object is the sole instance of *alarm.SleepMemory*.

alarm.**wake_alarm :_typing.Alarm**

The most recently triggered alarm. If CircuitPython was sleeping, the alarm the woke it from sleep.

alarm.**light_sleep_until_alarms**(**alarms:* _typing.Alarm) → *_typing.Alarm*

Go into a light sleep until awakened one of the alarms. The alarm causing the wake-up is returned, and is also available as *alarm.wake_alarm*.

If no alarms are specified, return immediately.

If CircuitPython is connected to a host computer, the connection will be maintained, and the microcontroller may not actually go into a light sleep. This allows the user to interrupt an existing program with ctrl-C, and to edit the files in CIRCUITPY, which would not be possible in true light sleep. Thus, to use light sleep and save significant power, it may be necessary to disconnect from the host.

alarm.**exit_and_deep_sleep_until_alarms**(**alarms:* _typing.Alarm) → None

Exit the program and go into a deep sleep, until awakened by one of the alarms. This function does not return.

When awakened, the microcontroller will restart and will run boot.py and code.py from the beginning.

After restart, an alarm *equivalent* to the one that caused the wake-up will be available as *alarm.wake_alarm*. Its type and/or attributes may not correspond exactly to the original alarm. For time-base alarms, currently, an *alarm.time.TimeAlarm()* is created.

If no alarms are specified, the microcontroller will deep sleep until reset.

If CircuitPython is connected to a host computer, the connection will be maintained, and the system will not go into deep sleep. This allows the user to interrupt an existing program with ctrl-C, and to edit the files

in CIRCUITPY, which would not be possible in true deep sleep. Thus, to use deep sleep and save significant power, you will need to disconnect from the host.

Here is skeletal example that deep-sleeps and restarts every 60 seconds:

```
import alarm
import time

print("Waking up")

# Set an alarm for 60 seconds from now.
time_alarm = alarm.time.TimeAlarm(monotonic_time=time.monotonic() + 60)

# Deep sleep until the alarm goes off. Then restart the program.
alarm.exit_and_deep_sleep_until_alarms(time_alarm)
```

class alarm.**SleepMemory**

Store raw bytes in RAM that persists during deep sleep. The class acts as a `bytearray`. If power is lost, the memory contents are lost.

Note that this class can't be imported and used directly. The sole instance of *SleepMemory* is available at *alarm.sleep_memory*.

Usage:

```
import alarm
alarm.sleep_memory[0] = True
alarm.sleep_memory[1] = 12
```

Not used. Access the sole instance through *alarm.sleep_memory*.

__bool__(*self*) → *bool*

`sleep_memory` is `True` if its length is greater than zero. This is an easy way to check for its existence.

__len__(*self*) → *int*

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

__getitem__(*self*, *index:* slice) → *bytearray*
__getitem__(*self*, *index:* int) → *int*

Returns the value at the given index.

__setitem__(*self*, *index:* slice, *value:* _typing.ReadableBuffer) → None
__setitem__(*self*, *index:* int, *value:* int) → None

Set the value at the given index.

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.

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 *
```

(continues on next page)

(continued from previous page)

```
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.

class analogio.**AnalogIn**(*pin:* microcontroller.Pin)

Read analog voltage levels

Usage:

```
import analogio
from board import *

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

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

value :int

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 :float

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

deinit(*self*) → None

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

__enter__(*self*) → *AnalogIn*

No-op used by Context Managers.

__exit__(*self*) → None

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

class analogio.**AnalogOut**(*pin:* microcontroller.Pin)

Output analog values (a specific voltage).

Example usage:

```
import analogio
from board import *

dac = analogio.AnalogOut(A2)                # output on pin A2
dac.value = 32768                           # makes A2 1.65V
```

Use the AnalogOut on the given pin.

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

value :int

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.

deinit(*self*) → None
Turn off the AnalogOut and release the pin for other use.

__enter__(*self*) → *AnalogOut*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

audiobusio – Support for audio input and output over digital buses

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.

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.

class audiobusio.**I2SOut**(*bit_clock:* microcontroller.Pin, *word_select:* microcontroller.Pin, *data:* microcontroller.Pin, *, *left_justified:* bool)
Output an I2S audio signal

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 audiocore
import board
import array
import time
import math

# Generate one period of sine wave.
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 / length) * (2 ** 15) + 2 ** 15)

sine_wave = audiocore.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 audiocore
import audiobusio
import digitalio


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

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

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

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

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

deinit(*self*) → None
Deinitialises the I2SOut and releases any hardware resources for reuse.

__enter__(*self*) → *I2SOut*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

play(*self*, *sample:* _typing.AudioSample, *, *loop:* bool = *False*) → None
Plays the sample once when loop=False and continuously when loop=True. Does not block. Use *playing* to block.

Sample must be an *audiocore.WaveFile*, *audiocore.RawSample*, *audiomixer.Mixer* or *audiomp3.MP3Decoder*.

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

stop(*self*) → None
Stops playback.

pause(*self*) → None
Stops playback temporarily while remembering the position. Use *resume* to resume playback.

resume(*self*) → None
Resumes sample playback after *pause()*.

class audiobusio.**PDMIn**(*clock_pin:* microcontroller.Pin, *data_pin:* microcontroller.Pin, *, *sample_rate:* int = *16000*, *bit_depth:* int = *8*, *mono:* bool = *True*, *oversample:* int = *64*, *startup_delay:* float = *0.11*)
Record an input PDM audio stream

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.

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

deinit(*self*) → None
Deinitialises the PDMIn and releases any hardware resources for reuse.

__enter__(*self*) → *PDMIn*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context.

record(*self*, *destination:* _typing.WriteableBuffer, *destination_length:* int) → None
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.

audiocore – Support for audio samples

class audiocore.**RawSample**(*buffer:* _typing.ReadableBuffer, *, *channel_count:* int = *1*, *sample_rate:* int = *8000*)
A raw audio sample buffer in memory

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** (ReadableBuffer) – A buffer 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 audiocore
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 / length) * (2 ** 15))

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

sample_rate :Optional[int]
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.

deinit(*self*) → None
Deinitialises the RawSample and releases any hardware resources for reuse.

__enter__(*self*) → *RawSample*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

class audiocore.**WaveFile**(*file: BinaryIO*, *buffer:* _typing.WriteableBuffer)
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. If a buffer is provided, it will be used instead of allocating an internal buffer.

Load a .wav file for playback with *audioio.AudioOut* or *audiobusio.I2SOut*.

Parameters

- **file** (*typing.BinaryIO*) – Already opened wave file
- **buffer** (WriteableBuffer) – Optional pre-allocated buffer, that will be split in half and used for double-buffering of the data. If not provided, two 256 byte buffers are allocated internally.

Playing a wave file from flash:

```
import board
import audiocore
import audioio
import digitalio

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

(continues on next page)

(continued from previous page)

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

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

sample_rate :int
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.

bits_per_sample :int
Bits per sample. (read only)

channel_count :int
Number of audio channels. (read only)

deinit(*self*) → None
Deinitialises the WaveFile and releases all memory resources for reuse.

__enter__(*self*) → *WaveFile*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

audioio – Support for audio output

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

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.

Since CircuitPython 5, *RawSample* and *WaveFile* are moved to *audiocore*, and *Mixer* is moved to *audiomixer*.

For compatibility with CircuitPython 4.x, some builds allow the items in *audiocore* to be imported from *audioio*. This will be removed for all boards in a future build of CircuitPython.

class audioio.**AudioOut**(*left_channel:* microcontroller.Pin, *, *right_channel:* Optional[microcontroller.Pin] = *None*, *quiescent_value:* int = *32768*)
Output an analog audio signal

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
- **quiescent_value** (int) – The output value when no signal is present. Samples should start and end with this value to prevent audible popping.

Simple 8ksps 440 Hz sin wave:

```
import audiocore
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 / length) * (2 ** 15) + 2 ** 15)

dac = audioio.AudioOut(board.SPEAKER)
sine_wave = audiocore.RawSample(sine_wave, sample_rate=8000)
dac.play(sine_wave, loop=True)
time.sleep(1)
dac.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 = audiocore.WaveFile(data)
a = audioio.AudioOut(board.A0)

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

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

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

deinit(*self*) → None
: Deinitialises the AudioOut and releases any hardware resources for reuse.

__enter__(*self*) → *AudioOut*
: No-op used by Context Managers.

__exit__(*self*) → None
: Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

play(*self*, *sample:* _typing.AudioSample, *, *loop:* bool = *False*) → None
: Plays the sample once when loop=False and continuously when loop=True. Does not block. Use *playing* to block.

Sample must be an *audiocore.WaveFile*, *audiocore.RawSample*, *audiomixer.Mixer* or *audiomp3.MP3Decoder*.

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(*self*) → None
Stops playback and resets to the start of the sample.

pause(*self*) → None
Stops playback temporarily while remembering the position. Use *resume* to resume playback.

resume(*self*) → None
Resumes sample playback after *pause()*.

audiomixer – Support for audio mixing

class audiomixer.**Mixer**(*voice_count:* int = *2*, *buffer_size:* int = *1024*, *channel_count:* int = *2*, *bits_per_sample:* int = *16*, *samples_signed:* bool = *True*, *sample_rate:* int = *8000*)
Mixes one or more audio samples together into one sample.

Create a Mixer object that can mix multiple channels with the same sample rate. Samples are accessed and controlled with the mixer's *audiomixer.MixerVoice* objects.

Parameters

- **voice_count** (int) – The maximum number of voices to mix
- **buffer_size** (int) – The total size in bytes of the buffers to mix into
- **channel_count** (int) – The number of channels the source samples contain. 1 = mono; 2 = stereo.
- **bits_per_sample** (int) – The bits per sample of the samples being played
- **samples_signed** (bool) – Samples are signed (True) or unsigned (False)
- **sample_rate** (int) – The sample rate to be used for all samples

Playing a wave file from flash:

```
import board
import audioio
import audiocore
import audiomixer
import digitalio

a = audioio.AudioOut(board.A0)
music = audiocore.WaveFile(open("cplay-5.1-16bit-16khz.wav", "rb"))
drum = audiocore.WaveFile(open("drum.wav", "rb"))
mixer = audiomixer.Mixer(voice_count=2, sample_rate=16000, channel_count=1,
                         bits_per_sample=16, samples_signed=True)

print("playing")
# Have AudioOut play our Mixer source
a.play(mixer)
# Play the first sample voice
mixer.voice[0].play(music)
while mixer.playing:
```

(continues on next page)

(continued from previous page)

```
  # Play the second sample voice
  mixer.voice[1].play(drum)
  time.sleep(1)
print("stopped")
```

playing :bool
True when any voice is being output. (read-only)

sample_rate :int
32 bit value that dictates how quickly samples are played in Hertz (cycles per second).

voice :Tuple[MixerVoice, Ellipsis]
A tuple of the mixer's *audiomixer.MixerVoice* object(s).

```
>>> mixer.voice
(<MixerVoice>,)
```

deinit (*self*) → None
Deinitialises the Mixer and releases any hardware resources for reuse.

__enter__ (*self*) → *Mixer*
No-op used by Context Managers.

__exit__ (*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

play (*self*, *sample:* _typing.AudioSample, *, *voice:* int = *0*, *loop:* bool = *False*) → None
Plays the sample once when loop=False and continuously when loop=True. Does not block. Use *playing* to block.

Sample must be an *audiocore.WaveFile*, *audiocore.RawSample*, *audiomixer.Mixer* or *audiomp3.MP3Decoder*.

The sample must match the Mixer's encoding settings given in the constructor.

stop_voice (*self*, *voice:* int = *0*) → None
Stops playback of the sample on the given voice.

class audiomixer.**MixerVoice**
Voice objects used with Mixer

Used to access and control samples with *audiomixer.Mixer*.

MixerVoice instance object(s) created by *audiomixer.Mixer*.

level :float
The volume level of a voice, as a floating point number between 0 and 1.

playing :bool
True when this voice is being output. (read-only)

play (*self*, *sample:* _typing.AudioSample, *, *loop:* bool = *False*) → None
Plays the sample once when `loop=False`, and continuously when `loop=True`. Does not block. Use *playing* to block.

Sample must be an *audiocore.WaveFile*, *audiocore.RawSample*, *audiomixer.Mixer* or *audiomp3.MP3Decoder*.

The sample must match the *audiomixer.Mixer*'s encoding settings given in the constructor.

stop(*self*) → None
Stops playback of the sample on this voice.

audiomp3 – Support for MP3-compressed audio files

class audiomp3.**MP3Decoder**(*file: BinaryIO*, *buffer:* _typing.WriteableBuffer)
Load a mp3 file for audio playback

Load a .mp3 file for playback with *audioio.AudioOut* or *audiobusio.I2SOut*.

Parameters

- **file** (*typing.BinaryIO*) – Already opened mp3 file
- **buffer** (WriteableBuffer) – Optional pre-allocated buffer, that will be split in half and used for double-buffering of the data. If not provided, two buffers are allocated internally. The specific buffer size required depends on the mp3 file.

Playing a mp3 file from flash:

```
import board
import audiomp3
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-16bit-16khz-64kbps.mp3", "rb")
mp3 = audiomp3.MP3Decoder(data)
a = audioio.AudioOut(board.A0)

print("playing")
a.play(mp3)
while a.playing:
  pass
print("stopped")
```

file :BinaryIO
File to play back.

sample_rate :int
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.

bits_per_sample :int
Bits per sample. (read only)

channel_count :int
Number of audio channels. (read only)

rms_level :float
The RMS audio level of a recently played moment of audio. (read only)

deinit(*self*) → None
Deinitialises the MP3 and releases all memory resources for reuse.

__enter__(*self*) → *MP3Decoder*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

audiopwmio – Audio output via digital PWM

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

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.

Since CircuitPython 5, *Mixer*, *RawSample* and *WaveFile* are moved to *audiocore*.

class audiopwmio.**PWMAudioOut**(*left_channel:* microcontroller.Pin, *, *right_channel:* *Optional[*microcontroller.Pin*]* = *None*, *quiescent_value:* int = *32768*)
Output an analog audio signal by varying the PWM duty cycle.

Create a PWMAudioOut object associated with the given pin(s). This allows you to play audio signals out on the given pin(s). In contrast to mod:*audioio*, the pin(s) specified are digital pins, and are driven with a device-dependent PWM signal.

Parameters

- **left_channel** (Pin) – The pin to output the left channel to
- **right_channel** (Pin) – The pin to output the right channel to
- **quiescent_value** (int) – The output value when no signal is present. Samples should start and end with this value to prevent audible popping.

Simple 8ksps 440 Hz sin wave:

```
import audiocore
import audiopwmio
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 / length) * (2 ** 15) + 2 ** 15)

dac = audiopwmio.PWMAudioOut(board.SPEAKER)
sine_wave = audiocore.RawSample(sine_wave, sample_rate=8000)
dac.play(sine_wave, loop=True)
time.sleep(1)
dac.stop()
```

Playing a wave file from flash:

```
import board
import audiocore
import audiopwmio
import digitalio
```

(continues on next page)

(continued from previous page)

```
# 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 = audiocore.WaveFile(data)
a = audiopwmio.PWMAudioOut(board.SPEAKER)

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

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

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

deinit(*self*) → None
Deinitialises the PWMAudioOut and releases any hardware resources for reuse.

__enter__(*self*) → *PWMAudioOut*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

play(*self*, *sample:* _typing.AudioSample, *, *loop:* bool = *False*) → None
Plays the sample once when loop=False and continuously when loop=True. Does not block. Use *playing* to block.

Sample must be an *audiocore.WaveFile*, *audiocore.RawSample*, *audiomixer.Mixer* or *audiomp3.MP3Decoder*.

The sample itself should consist of 16 bit samples. Microcontrollers with a lower output resolution will use the highest order bits to output.

stop(*self*) → None
Stops playback and resets to the start of the sample.

pause(*self*) → None
Stops playback temporarily while remembering the position. Use *resume* to resume playback.

resume(*self*) → None
Resumes sample playback after *pause()*.

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.

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.

class bitbangio.**I2C**(*scl:* microcontroller.Pin, *sda:* microcontroller.Pin, **, frequency:* int = *400000*, *timeout:* int = *255*)

Two wire serial protocol

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 of the bus
- **timeout** (int) – The maximum clock stretching timeout in microseconds

deinit(*self*) → None

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

__enter__(*self*) → *I2C*

No-op used in Context Managers.

__exit__(*self*) → None

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

scan(*self*) → List[*int*]

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(*self*) → *bool*
Attempts to grab the I2C lock. Returns True on success.

unlock(*self*) → None
Releases the I2C lock.

readfrom_into(*self*, *address:* int, *buffer:* _typing.WriteableBuffer, *, *start:* int = *0*, *end:* *Optional[*int*]* = *None*) → None
Read into `buffer` from the device selected 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** (`WriteableBuffer`) – buffer to write into
- **start** (`int`) – Index to start writing at
- **end** (`int`) – Index to write up to but not include

writeto(*self*, *address:* int, *buffer:* _typing.ReadableBuffer, *, *start:* int = *0*, *end:* *Optional[*int*]* = *None*, *stop:* bool = *True*) → None
Write the bytes from `buffer` to the device selected by `address` and then transmits a stop bit. Use *`writeto_then_readfrom`* when needing a write, no stop and repeated start before a 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 `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** (`ReadableBuffer`) – buffer containing the bytes to write
- **start** (`int`) – Index to start writing from
- **end** (`int`) – Index to read up to but not include

writeto_then_readfrom(*self*, *address:* int, *out_buffer:* _typing.ReadableBuffer, *in_buffer:* _typing.ReadableBuffer, *, *out_start:* int = *0*, *out_end:* *Optional[*int*]* = *None*, *in_start:* int = *0*, *in_end:* *Optional[*int*]* = *None*) → None
Write the bytes from `out_buffer` to the device selected by `address`, generate no stop bit, generate a repeated start and read into `in_buffer`. `out_buffer` and `in_buffer` can be the same buffer because they are used sequentially.

If `start` or `end` is provided, then the corresponding 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
- **out_buffer** (`ReadableBuffer`) – buffer containing the bytes to write
- **in_buffer** (`WriteableBuffer`) – buffer to write into
- **out_start** (`int`) – Index to start writing from
- **out_end** (`int`) – Index to read up to but not include. Defaults to `len(buffer)`
- **in_start** (`int`) – Index to start writing at

- **in_end** (int) – Index to write up to but not include. Defaults to len(buffer)

class bitbangio.**OneWire**(*pin:* microcontroller.Pin)

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

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(*self*) → None

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

__enter__(*self*) → *OneWire*

No-op used by Context Managers.

__exit__(*self*) → None

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

reset(*self*) → *bool*

Reset the OneWire bus

read_bit(*self*) → *bool*

Read in a bit

Returns bit state read

Return type *bool*

write_bit(*self*, *value:* bool) → None

Write out a bit based on value.

class bitbangio.**SPI**(*clock:* microcontroller.Pin, *MOSI: Optional[*microcontroller.Pin*] = None*, *MISO: Optional[*microcontroller.Pin*] = None*)

A 3-4 wire serial protocol

SPI is a serial protocol that has exclusive pins for data in and out of the main device. It is typically faster than *I2C* because a separate pin is used to select a device 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 select line, often abbreviated CS or SS. (This is common because multiple secondaries can share the clock, MOSI and MISO lines and therefore the hardware.)

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 Main Out Selected In pin.
- **MISO** (Pin) – the Main In Selected Out pin.

deinit (*self*) → None
Turn off the SPI bus.

__enter__ (*self*) → *SPI*
No-op used by Context Managers.

__exit__ (*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

configure (*self*, *, *baudrate:* int = *100000*, *polarity:* int = *0*, *phase:* int = *0*, *bits:* int = *8*) → None
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 (*self*) → *bool*
Attempts to grab the SPI lock. Returns True on success.

Returns True when lock has been grabbed

Return type *bool*

unlock (*self*) → None
Releases the SPI lock.

write (*self*, *buf:* _typing.ReadableBuffer) → None
Write the data contained in `buf`. Requires the SPI being locked. If the buffer is empty, nothing happens.

readinto (*self*, *buffer:* _typing.WriteableBuffer, *, *start:* int = *0*, *end:* *Optional[*int*]* = *None*, *write_value:* int = *0*) → None
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. Defaults to `len(buffer)`
- **write_value** (int) – Value to write while reading.

write_readinto(*self*, *buffer_out:* _typing.ReadableBuffer, *buffer_in:* _typing.ReadableBuffer, ***, *out_start:* int = *0*, *out_end: Optional[*int*]* = *None*, *in_start:* int = *0*, *in_end: Optional[*int*]* = *None*) → None

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** (ReadableBuffer) – Write out the data in this buffer
- **buffer_in** (WriteableBuffer) – 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. Defaults to `len(buffer_out)`
- **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. Defaults to `len(buffer_in)`

bitmaptools – Collection of bitmap manipulation tools

bitmaptools.**rotozoom**(*dest_bitmap:* displayio.Bitmap, *source_bitmap:* displayio.Bitmap, ***, *ox:* int, *oy:* int, *dest_clip0: Tuple[*int, int*]*, *dest_clip1: Tuple[*int, int*]*, *px:* int, *py:* int, *source_clip0: Tuple[*int, int*]*, *source_clip1: Tuple[*int, int*]*, *angle:* float, *scale:* float, *skip_index:* int) → None

Inserts the source bitmap region into the destination bitmap with rotation (angle), scale and clipping (both on source and destination bitmaps).

Parameters

- **dest_bitmap** (*bitmap*) – Destination bitmap that will be copied into
- **source_bitmap** (*bitmap*) – Source bitmap that contains the graphical region to be copied
- **ox** (int) – Horizontal pixel location in destination bitmap where source bitmap point (px,py) is placed
- **oy** (int) – Vertical pixel location in destination bitmap where source bitmap point (px,py) is placed
- **dest_clip0** (*Tuple[int, int]*) – First corner of rectangular destination clipping region that constrains region of writing into destination bitmap
- **dest_clip1** (*Tuple[int, int]*) – Second corner of rectangular destination clipping region that constrains region of writing into destination bitmap
- **px** (int) – Horizontal pixel location in source bitmap that is placed into the destination bitmap at (ox,oy)
- **py** (int) – Vertical pixel location in source bitmap that is placed into the destination bitmap at (ox,oy)
- **source_clip0** (*Tuple[int, int]*) – First corner of rectangular source clipping region that constrains region of reading from the source bitmap

- **source_clip1** (*Tuple[int, int]*) – Second corner of rectangular source clipping region that constrains region of reading from the source bitmap
- **angle** (float) – Angle of rotation, in radians (positive is clockwise direction)
- **scale** (float) – Scaling factor
- **skip_index** (int) – Bitmap palette index in the source that will not be copied, set to None to copy all pixels

bitmaptools.**fill_region**(*dest_bitmap:* displayio.Bitmap, *x1:* int, *y1:* int, *x2:* int, *y2:* int, *value:* int) → None

Draws the color value into the destination bitmap within the rectangular region bounded by (x1,y1) and (x2,y2), exclusive.

Parameters

- **dest_bitmap** (*bitmap*) – Destination bitmap that will be written into
- **x1** (int) – x-pixel position of the first corner of the rectangular fill region
- **y1** (int) – y-pixel position of the first corner of the rectangular fill region
- **x2** (int) – x-pixel position of the second corner of the rectangular fill region (exclusive)
- **y2** (int) – y-pixel position of the second corner of the rectangular fill region (exclusive)
- **value** (int) – Bitmap palette index that will be written into the rectangular fill region in the destination bitmap

bitmaptools.**draw_line**(*dest_bitmap:* displayio.Bitmap, *x1:* int, *y1:* int, *x2:* int, *y2:* int, *value:* int) → None

Draws a line into a bitmap specified two endpoints (x1,y1) and (x2,y2).

Parameters

- **dest_bitmap** (*bitmap*) – Destination bitmap that will be written into
- **x1** (int) – x-pixel position of the line's first endpoint
- **y1** (int) – y-pixel position of the line's first endpoint
- **x2** (int) – x-pixel position of the line's second endpoint
- **y2** (int) – y-pixel position of the line's second endpoint
- **value** (int) – Bitmap palette index that will be written into the line in the destination bitmap

bitmaptools.**arrayblit**(*bitmap:* displayio.Bitmap, *data:* _typing.ReadableBuffer, *x1:* int = *0*, *y1:* int = *0*, *x2: Optional[*int*] = None*, *y2: Optional[*int*] = None*, *skip_index: Optional[*int*] = None*) → None

Inserts pixels from `data` into the rectangle of width×height pixels with the upper left corner at `(x, y)`

The values from `data` are taken modulo the number of color values avalable in the destination bitmap.

If x1 or y1 are not specified, they are taken as 0. If x2 or y2 are not specified, or are given as -1, they are taken as the width and height of the image.

The coordinates affected by the blit are `x1 <= x < x2` and `y1 <= y < y2`.

`data` must contain at least as many elements as required. If it contains excess elements, they are ignored.

The blit takes place by rows, so the first elements of `data` go to the first row, the next elements to the next row, and so on.

Parameters

- **bitmap** (displayio.Bitmap) – A writable bitmap
- **data** (*ReadableBuffer*) – Buffer containing the source pixel values
- **x1** (int) – The left corner of the area to blit into (inclusive)
- **y1** (int) – The top corner of the area to blit into (inclusive)
- **x2** (int) – The right of the area to blit into (exclusive)
- **y2** (int) – The bottom corner of the area to blit into (exclusive)
- **skip_index** (int) – Bitmap palette index in the source that will not be copied, set to None to copy all pixels

bitmaptools.**readinto**(*bitmap:* displayio.Bitmap, *file: BinaryIO*, *bits_per_pixel:* int, *element_size:* int = *1*, *reverse_pixels_in_element:* bool = *False*, *swap_bytes_in_element:* bool = *False*, *reverse_rows:* bool = *False*) → None

Reads from a binary file into a bitmap.

The file must be positioned so that it consists of `bitmap.height` rows of pixel data, where each row is the smallest multiple of `element_size` bytes that can hold `bitmap.width` pixels.

The bytes in an element can be optionally swapped, and the pixels in an element can be reversed. Also, the row loading direction can be reversed, which may be requires for loading certain bitmap files.

This function doesn't parse image headers, but is useful to speed up loading of uncompressed image formats such as PCF glyph data.

Parameters

- **bitmap** (displayio.Bitmap) – A writable bitmap
- **file** (*typing.BinaryIO*) – A file opened in binary mode
- **bits_per_pixel** (int) – Number of bits per pixel. Values 1, 2, 4, 8, 16, 24, and 32 are supported;
- **element_size** (int) – Number of bytes per element. Values of 1, 2, and 4 are supported, except that 24 `bits_per_pixel` requires 1 byte per element.
- **reverse_pixels_in_element** (bool) – If set, the first pixel in a word is taken from the Most Signficant Bits; otherwise, it is taken from the Least Significant Bits.
- **swap_bytes_in_element** (bool) – If the `element_size` is not 1, then reverse the byte order of each element read.
- **reverse_rows** (bool) – Reverse the direction of the row loading (required for some bitmap images).

bitops – Routines for low-level manipulation of binary data

bitops.**bit_transpose**(*input:* _typing.ReadableBuffer, *output:* _typing.WriteableBuffer, *width:* int = *8*) → *_typing.WriteableBuffer*

"Transpose" a buffer by assembling each output byte with bits taken from each of `width` different input bytes.

This can be useful to convert a sequence of pixel values into a single stream of bytes suitable for sending via a parallel conversion method.

The number of bytes in the input buffer must be a multiple of the width, and the width can be any value from 2 to 8. If the width is fewer than 8, then the remaining (less significant) bits of the output are set to zero.

Let `stride = len(input)//width`. Then the first byte is made out of the most significant bits of `[input[0], input[stride], input[2*stride], ...]`. The second byte is made out of

the second bits, and so on until the 8th output byte which is made of the first bits of `input[1]`, `input[1+stride]`, `input[2*stride]`, `...`.

The required output buffer size is `len(input) * 8 // width`.

Returns the output buffer.

`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.

Warning: The board module varies by board. The APIs documented here may or may not be available on a specific board.

`board.`**`I2C`**`()` → *busio.I2C*
Returns the *`busio.I2C`* object for the board designated SDA and SCL pins. It is a singleton.

`board.`**`SPI`**`()` → *busio.SPI*
Returns the *`busio.SPI`* object for the board designated SCK, MOSI and MISO pins. It is a singleton.

`board.`**`UART`**`()` → *busio.UART*
Returns the *`busio.UART`* object for the board designated TX and RX pins. It is a singleton.

The object created uses the default parameter values for *`busio.UART`*. If you need to set parameters that are not changeable after creation, such as `receiver_buffer_size`, do not use *`board.UART()`*; instead create a *`busio.UART`* object explicitly with the desired parameters.

`busio` – Hardware accelerated external bus access

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.

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.

`class` `busio.`**`I2C`**`(`*scl:* microcontroller.Pin, *sda:* microcontroller.Pin, **, frequency:* int = *100000*, *timeout:* int = *255*`)`
Two wire serial protocol

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 timeout - (used only for `bitbangio.I2C`; ignored for `busio.I2C`)

Note: On the nRF52840, only one I2C object may be created, except on the Circuit Playground Bluefruit, which allows two, one for the onboard accelerometer, and one for offboard use.

---

deinit(*self*) → None
Releases control of the underlying hardware so other classes can use it.

__enter__(*self*) → *I2C*
No-op used in Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware on context exit. See *Lifetime and ContextManagers* for more info.

scan(*self*) → List[*int*]
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(*self*) → *bool*
Attempts to grab the I2C lock. Returns True on success.

Returns True when lock has been grabbed

Return type *bool*

unlock(*self*) → None
Releases the I2C lock.

readfrom_into(*self*, *address:* int, *buffer:* _typing.WriteableBuffer, *, *start:* int = *0*, *end:* *Optional[*int*] = None*) → None
Read into `buffer` from the device selected 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** (WriteableBuffer) – buffer to write into
- **start** (int) – Index to start writing at
- **end** (int) – Index to write up to but not include. Defaults to len(buffer)

writeto(*self*, *address:* int, *buffer:* _typing.ReadableBuffer, *, *start:* int = *0*, *end:* *Optional[*int*]* = *None*, *stop:* bool = *True*) → None

Write the bytes from buffer to the device selected by address and then transmit a stop bit.

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** (ReadableBuffer) – buffer containing the bytes to write
- **start** (int) – Index to start writing from
- **end** (int) – Index to read up to but not include. Defaults to len(buffer)

writeto_then_readfrom(*self*, *address:* int, *out_buffer:* _typing.ReadableBuffer, *in_buffer:* _typing.WriteableBuffer, *, *out_start:* int = *0*, *out_end:* *Optional[*int*]* = *None*, *in_start:* int = *0*, *in_end:* *Optional[*int*]* = *None*) → None

Write the bytes from out_buffer to the device selected by address, generate no stop bit, generate a repeated start and read into in_buffer. out_buffer and in_buffer can be the same buffer because they are used sequentially.

If start or end is provided, then the corresponding 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
- **out_buffer** (ReadableBuffer) – buffer containing the bytes to write
- **in_buffer** (WriteableBuffer) – buffer to write into
- **out_start** (int) – Index to start writing from
- **out_end** (int) – Index to read up to but not include. Defaults to len(buffer)
- **in_start** (int) – Index to start writing at
- **in_end** (int) – Index to write up to but not include. Defaults to len(buffer)

class busio.**OneWire**(*pin:* microcontroller.Pin)

Lowest-level of the Maxim OneWire protocol

(formerly Dallas Semi) OneWire protocol.

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

class 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(*self*) → None
Deinitialize the OneWire bus and release any hardware resources for reuse.

__enter__(*self*) → *OneWire*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

reset(*self*) → *bool*
Reset the OneWire bus and read presence

Returns False when at least one device is present

Return type *bool*

read_bit(*self*) → *bool*
Read in a bit

Returns bit state read

Return type *bool*

write_bit(*self*, *value:* bool) → None
Write out a bit based on value.

class busio.**SPI**(*clock:* microcontroller.Pin, *MOSI: Optional[*microcontroller.Pin*] = None*, *MISO: Optional[*microcontroller.Pin*] = None*)
A 3-4 wire serial protocol

SPI is a serial protocol that has exclusive pins for data in and out of the main device. It is typically faster than *I2C* because a separate pin is used to select a device 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 select line, often abbreviated `CS` or `SS`. (This is common because multiple secondaries can share the `clock`, `MOSI` and `MISO` lines and therefore the hardware.)

Construct an SPI object on the given pins.

Note: The SPI peripherals allocated in order of desirability, if possible, such as highest speed and not shared use first. For instance, on the nRF52840, there is a single 32MHz SPI peripheral, and multiple 8MHz peripherals, some of which may also be used for I2C. The 32MHz SPI peripheral is returned first, then the exclusive 8MHz SPI peripheral, and finally the shared 8MHz peripherals.

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 Main Out Selected In pin.
- **MISO** (Pin) – the Main In Selected Out pin.

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

deinit (*self*) → None
Turn off the SPI bus.

__enter__ (*self*) → *SPI*
No-op used by Context Managers. Provided by context manager helper.

__exit__ (*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

configure (*self*, *, *baudrate:* int = *100000*, *polarity:* int = *0*, *phase:* int = *0*, *bits:* int = *8*) → None
Configures the SPI bus. The SPI object must be 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.
- **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

Note: On the SAMD21, it is possible to set the baudrate 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.

Note: On the nRF52840, these baudrates are available: 125kHz, 250kHz, 1MHz, 2MHz, 4MHz, and 8MHz. If you pick a a baudrate other than one of these, the nearest lower baudrate will be chosen, with a minimum of 125kHz. Two SPI objects may be created, except on the Circuit Playground Bluefruit, which allows only one (to allow for an additional I2C object).

---

try_lock (*self*) → *bool*
Attempts to grab the SPI lock. Returns True on success.

Returns True when lock has been grabbed

Return type *bool*

unlock (*self*) → None
Releases the SPI lock.

write (*self*, *buffer:* _typing.ReadableBuffer, *, *start:* int = *0*, *end: Optional[*int*]* = *None*) → None
Write the data contained in buffer. The SPI object must be locked. If the buffer is empty, nothing happens.

Parameters

- **buffer** (ReadableBuffer) – 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. Defaults to `len(buffer)`

readinto(*self*, *buffer:* _typing.WriteableBuffer, *, *start:* int = *0*, *end:* *Optional[*int*]* = *None*, *write_value:* int = *0*) → None

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** (WriteableBuffer) – 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. Defaults to `len(buffer)`
- **write_value** (int) – Value to write while reading. (Usually ignored.)

write_readinto(*self*, *buffer_out:* _typing.ReadableBuffer, *buffer_in:* _typing.WriteableBuffer, *, *out_start:* int = *0*, *out_end:* *Optional[*int*]* = *None*, *in_start:* int = *0*, *in_end:* *Optional[*int*]* = *None*) → None

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** (ReadableBuffer) – Write out the data in this buffer
- **buffer_in** (WriteableBuffer) – 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. Defaults to `len(buffer_out)`
- **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. Defaults to `len(buffer_in)`

class busio.**UART**(*tx:* microcontroller.Pin, *rx:* microcontroller.Pin, *, *baudrate:* int = *9600*, *bits:* int = *8*, *parity:* *Optional[*Parity*]* = *None*, *stop:* int = *1*, *timeout:* float = *1*, *receiver_buffer_size:* int = *64*)

A bidirectional serial protocol

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.
- **rts** (Pin) – the pin for rts, or `None` if rts not in use.
- **cts** (Pin) – the pin for cts, or `None` if cts not in use.
- **rs485_dir** (Pin) – the output pin for rs485 direction setting, or `None` if rs485 not in use.
- **rs485_invert** (bool) – rs485_dir pin active high when set. Active low otherwise.
- **baudrate** (int) – the transmit and receive speed.

- **bits** (int) – the number of bits per byte, 5 to 9.
- **parity** (Parity) – the parity used for error checking.
- **stop** (int) – the number of stop bits, 1 or 2.
- **timeout** (float) – the timeout in seconds to wait for the first character and between subsequent characters when reading. Raises ValueError if timeout >100 seconds.
- **receiver_buffer_size** (int) – the character length of the read buffer (0 to disable). (When a character is 9 bits the buffer will be 2 * receiver_buffer_size bytes.)

New in CircuitPython 4.0: timeout has incompatibly changed units from milliseconds to seconds. The new upper limit on timeout is meant to catch mistaken use of milliseconds.

baudrate :int
The current baudrate.

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

timeout :float
The current timeout, in seconds (float).

deinit(*self*) → None
Deinitialises the UART and releases any hardware resources for reuse.

__enter__(*self*) → *UART*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

read(*self*, *nbytes: Optional[*int*] = None*) → Optional[*bytes*]
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(*self*, *buf:* _typing.WriteableBuffer) → Optional[*int*]
Read bytes into the buf. Read at most len(buf) bytes.

Returns number of bytes read and stored into buf

Return type *int* or None (on a non-blocking error)

New in CircuitPython 4.0: No length parameter is permitted.

readline(*self*) → *bytes*

Read a line, ending in a newline character, or return None if a timeout occurs sooner, or return everything readable if no newline is found and timeout=0

Returns the line read

Return type *bytes* or None

write(*self*, *buf:* _typing.WriteableBuffer) → Optional[*int*]
Write the buffer of bytes to the bus.

New in CircuitPython 4.0: buf must be bytes, not a string.

return the number of bytes written

rtype int or None

reset_input_buffer(*self*) → None
Discard any unread characters in the input buffer.

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

ODD :int
Total number of ones should be odd.

EVEN :int
Total number of ones should be even.

camera – Support for camera input

The *camera* module contains classes to control the camera and take pictures.

class camera.**Camera**
The class to control camera.

Usage:

```
import board
import sdioio
import storage
import camera

sd = sdioio.SDCard(
    clock=board.SDIO_CLOCK,
    command=board.SDIO_COMMAND,
    data=board.SDIO_DATA,
    frequency=25000000)
vfs = storage.VfsFat(sd)
storage.mount(vfs, '/sd')

cam = camera.Camera()

buffer = bytearray(512 * 1024)
file = open("/sd/image.jpg","wb")
size = cam.take_picture(buffer, width=1920, height=1080, format=camera.
↪ImageFormat.JPG)
file.write(buffer, size)
file.close()
```

Initialize camera.

deinit(*self*) → None
De-initialize camera.

take_picture(*self*, *buf:* _typing.WriteableBuffer, *format:* ImageFormat) → *int*
Take picture and save to `buf` in the given `format`. The size of the picture taken is `width` by `height` in pixels.

Returns the number of bytes written into buf

Return type *int*

class camera.**ImageFormat**
Image format

Enum-like class to define the image format.

JPG :ImageFormat
JPG format.

RGB565 :ImageFormat
RGB565 format.

canio – CAN bus access

The *canio* module contains low level classes to support the CAN bus protocol.

CAN and Listener 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 canio
from board import *

can = canio.CAN(board.CAN_RX, board.CAN_TX, baudrate=1000000)
message = canio.Message(id=0x0408, data=b"adafruit")
can.send(message)
can.deinit()
```

This example will write the data 'adafruit' onto the CAN bus to any device listening for message id 0x0408.

A CAN bus involves a transceiver, which is often a separate chip with a "standby" pin. If your board has a CAN_STANDBY pin, ensure to set it to an output with the value False to enable the transceiver.

Other implementations of the CAN device may exist (for instance, attached via an SPI bus). If so their constructor arguments may differ, but otherwise we encourage implementors to follow the API that the core uses.

class canio.**BusState**
The state of the CAN bus

ERROR_ACTIVE :object
The bus is in the normal (active) state

ERROR_WARNING :object
The bus is in the normal (active) state, but a moderate number of errors have occurred recently.

NOTE: Not all implementations may use ERROR_WARNING. Do not rely on seeing ERROR_WARNING before ERROR_PASSIVE.

ERROR_PASSIVE :object
The bus is in the passive state due to the number of errors that have occurred recently.

This device will acknowledge packets it receives, but cannot transmit messages. If additional errors occur, this device may progress to BUS_OFF. If it successfully acknowledges other packets on the bus, it can return to ERROR_WARNING or ERROR_ACTIVE and transmit packets.

BUS_OFF :object
The bus has turned off due to the number of errors that have occurred recently. It must be restarted before it will send or receive packets. This device will neither send or acknowledge packets on the bus.

class canio.**CAN**(*tx:* microcontroller.Pin, *rx:* microcontroller.Pin, *, *baudrate:* int = *250000*, *loopback:* bool = *False*, *silent:* bool = *False*, *auto_restart:* bool = *False*)

CAN bus protocol

A common shared-bus protocol. The rx and tx pins are generally connected to a transceiver which controls the H and L pins on a shared bus.

Parameters

- **rx** (Pin) – the pin to receive with
- **tx** (Pin) – the pin to transmit with
- **baudrate** (int) – The bit rate of the bus in Hz. All devices on the bus must agree on this value.
- **loopback** (bool) – When True the rx pin's value is ignored, and the device receives the packets it sends.
- **silent** (bool) – When True the tx pin is always driven to the high logic level. This mode can be used to "sniff" a CAN bus without interfering.
- **auto_restart** (bool) – If True, will restart communications after entering bus-off state

auto_restart :bool
If True, will restart communications after entering bus-off state

baudrate :int
The baud rate (read-only)

transmit_error_count :int
The number of transmit errors (read-only). Increased for a detected transmission error, decreased for successful transmission. Limited to the range from 0 to 255 inclusive. Also called TEC.

receive_error_count :int
The number of receive errors (read-only). Increased for a detected reception error, decreased for successful reception. Limited to the range from 0 to 255 inclusive. Also called REC.

state :BusState
The current state of the bus. (read-only)

loopback :bool
True if the device was created in loopback mode, False otherwise (read-only)

silent :bool
True if the device was created in silent mode, False otherwise (read-only)

restart(*self*) → None
If the device is in the bus off state, restart it.

listen(*self*, *matches: Optional[Sequence[*Match*]] = None*, *, *timeout:* float = *10*) → *Listener*
Start receiving messages that match any one of the filters.

Creating a listener is an expensive operation and can interfere with reception of messages by other listeners.

There is an implementation-defined maximum number of listeners and limit to the complexity of the filters.

If the hardware cannot support all the requested matches, a ValueError is raised. Note that generally there are some number of hardware filters shared among all fifos.

A message can be received by at most one Listener. If more than one listener matches a message, it is undefined which one actually receives it.

An empty filter list causes all messages to be accepted.

Timeout dictates how long receive() and next() will block.

Platform specific notes:

SAM E5x supports two Listeners. Filter blocks are shared between the two listeners. There are 4 standard filter blocks and 4 extended filter blocks. Each block can either match 2 single addresses or a mask of addresses. The number of filter blocks can be increased, up to a hardware maximum, by rebuilding CircuitPython, but this decreases the CircuitPython free memory even if canio is not used.

STM32F405 supports two Listeners. Filter blocks are shared between the two listeners. There are 14 filter blocks. Each block can match 2 standard addresses with mask or 1 extended address with mask.

ESP32S2 supports one Listener. There is a single filter block, which can either match a standard address with mask or an extended address with mask.

send(*self*, *message: Union[*RemoteTransmissionRequest, Message*]*) → None
Send a message on the bus with the given data and id. If the message could not be sent due to a full fifo or a bus error condition, RuntimeError is raised.

deinit(*self*) → None
Deinitialize this object, freeing its hardware resources

__enter__(*self*) → *CAN*
Returns self, to allow the object to be used in a `The with statement` statement for resource control

__exit__(*self*, *unused1: Optional[Type[BaseException]]*, *unused2: Optional[BaseException]*, *unused3: Optional[types.TracebackType]*) → None
Calls deinit()

class canio.**Listener**
Listens for CAN message

canio.Listener is not constructed directly, but instead by calling *canio.CAN.listen*.

In addition to using the *receive* method to retrieve a message or the *in_waiting* method to check for an available message, a listener can be used as an iterable, yielding messages until no message arrives within `self.timeout` seconds.

timeout :float

receive(*self*) → Optional[Union[*RemoteTransmissionRequest*, *Message*]]
Reads a message, after waiting up to `self.timeout` seconds

If no message is received in time, `None` is returned. Otherwise, a *Message* or *RemoteTransmissionRequest* is returned.

in_waiting(*self*) → *int*
Returns the number of messages (including remote transmission requests) waiting

__iter__(*self*) → *Listener*
Returns self

This method exists so that *Listener* can be used as an iterable

__next__(*self*) → Union[*RemoteTransmissionRequest*, *Message*]
Reads a message, after waiting up to self.timeout seconds

If no message is received in time, raises StopIteration. Otherwise, a Message or is returned.

This method enables the *Listener* to be used as an iterable, for instance in a for-loop.

deinit(*self*) → None
Deinitialize this object, freeing its hardware resources

__enter__(*self*) → *CAN*
Returns self, to allow the object to be used in a `The with statement` statement for resource control

__exit__(*self*, *unused1: Optional[Type[BaseException]]*, *unused2: Optional[BaseException]*, *unused3: Optional[types.TracebackType]*) → None
Calls deinit()

class canio.**Match**(*id:* int, *, *mask: Optional[*int*] = None*, *extended:* bool = *False*)
Describe CAN bus messages to match

Construct a Match with the given properties.

If mask is not None, then the filter is for any id which matches all the nonzero bits in mask. Otherwise, it matches exactly the given id. If extended is true then only extended ids are matched, otherwise only standard ids are matched.

id :int
The id to match

mask :int
The optional mask of ids to match

extended :bool
True to match extended ids, False to match standard ides

class canio.**Message**(*id:* int, *data:* bytes, *, *extended:* bool = *False*)
Construct a Message to send on a CAN bus.

Parameters

- **id** (int) – The numeric ID of the message
- **data** (bytes) – The content of the message
- **extended** (bool) – True if the message has an extended identifier, False if it has a standard identifier

In CAN, messages can have a length from 0 to 8 bytes.

id :int
The numeric ID of the message

data :bytes
The content of the message

extended :bool
True if the message's id is an extended id

class canio.**RemoteTransmissionRequest**(*id:* int, *length:* int, *, *extended:* bool = *False*)
Construct a RemoteTransmissionRequest to send on a CAN bus.

Parameters

- **id** (int) – The numeric ID of the requested message
- **length** (int) – The length of the requested message
- **extended** (bool) – True if the message has an extended identifier, False if it has a standard identifier

In CAN, messages can have a length from 0 to 8 bytes.

id :int
The numeric ID of the message

extended :bool
True if the message's id is an extended id

length :int
The length of the requested message.

countio – Support for edge counting

The *countio* module contains logic to read and count edge transistions

Warning: This module is not available in some SAMD21 (aka M0) builds. See the *Module Support Matrix - Which Modules Are Available on Which Boards* 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.

class countio.**Counter**(*pin_a:* microcontroller.Pin)
Counter will keep track of the number of falling edge transistions (pulses) on a given pin

Create a Counter object associated with the given pin. It tracks the number of falling pulses relative when the object is constructed.

Parameters **pin_a** (Pin) – Pin to read pulses from.

For example:

```
import board
import countio

pin_counter = countio.Counter(board.D1)
#reset the count after 100 counts
while True:
    if pin_counter.count == 100:
        pin_counter.reset()
    print(pin_counter.count)
```

count :int
The current count in terms of pulses.

deinit(*self*) → None
Deinitializes the Counter and releases any hardware resources for reuse.

__enter__(*self*) → *Counter*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

reset(*self*) → None
Resets the count back to 0.

digitalio – Basic digital pin support

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

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)
```

class digitalio.**DriveMode**
Defines the drive mode of a digital pin

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

PUSH_PULL :DriveMode
Output both high and low digital values

OPEN_DRAIN :DriveMode
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.

class digitalio.**DigitalInOut**(*pin:* microcontroller.Pin)
Digital input and output

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

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

direction :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 :bool
The digital logic level of the pin.

drive_mode :DriveMode
The pin drive mode. One of:

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

pull :Optional[Pull]
The pin pull direction. One of:

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

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

deinit (*self*) → None
Turn off the DigitalInOut and release the pin for other use.

__enter__ (*self*) → *DigitalInOut*
No-op used by Context Managers.

__exit__ (*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

switch_to_output (*self*, *value:* bool = *False*, *drive_mode:* DriveMode = *DriveMode.PUSH_PULL*) → None
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 (*self*, *pull: Optional[*Pull*]* = *None*) → 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)
```

class digitalio.**Direction**
Defines the direction of a digital pin

Enum-like class to define which direction the digital values are going.

INPUT :Direction
Read digital data in

OUTPUT :Direction
Write digital data out

class digitalio.**Pull**
Defines the pull of a digital input pin

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

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

displayio – Native helpers for driving displays

The *displayio* module contains classes to manage display output including synchronizing with refresh rates and partial updating.

displayio.**release_displays**() → None
Releases any actively used displays so their busses and pins can be used again. This will also release the builtin display on boards that have one. You will need to reinitialize it yourself afterwards. This may take seconds to complete if an active EPaperDisplay is refreshing.

Use this once in your code.py if you initialize a display. Place it right before the initialization so the display is active as long as possible.

class displayio.**Colorspace**
The colorspace for a *ColorConverter* to operate in

RGB888 :Colorspace
The standard 24-bit colorspace. Bits 0-7 are blue, 8-15 are green, and 16-24 are red. (0xRRGGBB)

RGB565 :Colorspace
The standard 16-bit colorspace. Bits 0-4 are blue, bits 5-10 are green, and 11-15 are red (0bRRRRRGGGGGGBBBBB)

RGB565_SWAPPED :Colorspace
The swapped 16-bit colorspace. First, the high and low 8 bits of the number are swapped, then they are interpreted as for RGB565

RGB555 :Colorspace
The standard 15-bit colorspace. Bits 0-4 are blue, bits 5-9 are green, and 11-14 are red. The top bit is ignored. (0bxRRRRRGGGGGBBBBB)

RGB555_SWAPPED :Colorspace
The swapped 15-bit colorspace. First, the high and low 8 bits of the number are swapped, then they are interpreted as for RGB555

class displayio.**Bitmap**(*width:* int, *height:* int, *value_count:* int)
Stores values of a certain size in a 2D array

Bitmaps can be treated as read-only buffers. If the number of bits in a pixel is 8, 16, or 32; and the number of bytes per row is a multiple of 4, then the resulting memoryview will correspond directly with the bitmap's contents. Otherwise, the bitmap data is packed into the memoryview with unspecified padding.

A Bitmap can be treated as a buffer, allowing its content to be viewed and modified using e.g., with ulab.numpy.frombuffer, but the *displayio.Bitmap.dirty* method must be used to inform displayio when a bitmap was modified through the buffer interface.

bitmaptools.arrayblit can also be useful to move data efficiently into a Bitmap.

Create a Bitmap object with the given fixed size. Each pixel stores a value that is used to index into a corresponding palette. This enables differently colored sprites to share the underlying Bitmap. value_count is used to minimize the memory used to store the Bitmap.

Parameters

- **width** (int) – The number of values wide
- **height** (int) – The number of values high
- **value_count** (int) – The number of possible pixel values.

width :int
Width of the bitmap. (read only)

height :int
Height of the bitmap. (read only)

__getitem__(*self*, *index: Union[Tuple[*int, int*], *int*]*) → *int*
Returns the value at the given index. The index can either be an x,y tuple or an int equal to `y * width + x`.

This allows you to:

```
print(bitmap[0,1])
```

__setitem__(*self*, *index: Union[Tuple[*int, int*], *int*]*, *value:* int) → None
Sets the value at the given index. The index can either be an x,y tuple or an int equal to `y * width + x`.

This allows you to:

```
bitmap[0,1] = 3
```

blit(*self*, *x:* int, *y:* int, *source_bitmap:* Bitmap, *, *x1:* int, *y1:* int, *x2:* int, *y2:* int, *skip_index:* int) → None
Inserts the source_bitmap region defined by rectangular boundaries (x1,y1) and (x2,y2) into the bitmap at the specified (x,y) location.

Parameters

- **x** (int) – Horizontal pixel location in bitmap where source_bitmap upper-left corner will be placed
- **y** (int) – Vertical pixel location in bitmap where source_bitmap upper-left corner will be placed
- **source_bitmap** (*bitmap*) – Source bitmap that contains the graphical region to be copied
- **x1** (int) – Minimum x-value for rectangular bounding box to be copied from the source bitmap
- **y1** (int) – Minimum y-value for rectangular bounding box to be copied from the source bitmap
- **x2** (int) – Maximum x-value (exclusive) for rectangular bounding box to be copied from the source bitmap
- **y2** (int) – Maximum y-value (exclusive) for rectangular bounding box to be copied from the source bitmap

- **skip_index** (int) – bitmap palette index in the source that will not be copied, set to None to copy all pixels

fill(*self*, *value:* int) → None
Fills the bitmap with the supplied palette index value.

dirty(*self*, *x1:* int = *0*, *y1:* int = *0*, *x2:* int = *- 1*, *y2:* int = *- 1*) → None
Inform displayio of bitmap updates done via the buffer protocol.

Parameters

- **x1** (int) – Minimum x-value for rectangular bounding box to be considered as modified
- **y1** (int) – Minimum y-value for rectangular bounding box to be considered as modified
- **x2** (int) – Maximum x-value (exclusive) for rectangular bounding box to be considered as modified
- **y2** (int) – Maximum y-value (exclusive) for rectangular bounding box to be considered as modified

If x1 or y1 are not specified, they are taken as 0. If x2 or y2 are not specified, or are given as -1, they are taken as the width and height of the image. Thus, calling dirty() with the default arguments treats the whole bitmap as modified.

When a bitmap is modified through the buffer protocol, the display will not be properly updated unless the bitmap is notified of the "dirty rectangle" that encloses all modified pixels.

class displayio.**ColorConverter**(*, *colorspace:* Colorspace = *Colorspace.RGB888*, *dither:* bool = *False*)
Converts one color format to another.

Create a ColorConverter object to convert color formats.

Parameters

- **colorspace** (Colorspace) – The source colorspace, one of the Colorspace constants
- **dither** (bool) – Adds random noise to dither the output image

dither :bool
When true the color converter dithers the output by adding random noise when truncating to display bitdepth

convert(*self*, *color:* int) → *int*
Converts the given color to RGB565 according to the Colorspace

make_transparent(*self*, *pixel:* int) → None
Sets a pixel to not opaque.

make_opaque(*self*, *pixel:* int) → None
Sets a pixel to opaque.

displayio.**_DisplayBus**
FourWire, *ParallelBus* or *I2CDisplay*

class displayio.**Display**(*display_bus:* _DisplayBus, *init_sequence:* _typing.ReadableBuffer, *, *width:* int, *height:* int, *colstart:* int = *0*, *rowstart:* int = *0*, *rotation:* int = *0*, *color_depth:* int = *16*, *grayscale:* bool = *False*, *pixels_in_byte_share_row:* bool = *True*, *bytes_per_cell:* int = *1*, *reverse_pixels_in_byte:* bool = *False*, *set_column_command:* int = *42*, *set_row_command:* int = *43*, *write_ram_command:* int = *44*, *set_vertical_scroll:* int = *0*, *backlight_pin:* *Optional[*microcontroller.Pin*]* = *None*, *brightness_command:* *Optional[*int*]* = *None*, *brightness:* float = *1.0*, *auto_brightness:* bool = *False*, *single_byte_bounds:* bool = *False*, *data_as_commands:* bool = *False*, *auto_refresh:* bool = *True*, *native_frames_per_second:* int = *60*, *backlight_on_high:* bool = *True*, *SH1107_addressing:* bool = *False*)

Manage updating a display over a display bus

This initializes a display and connects it into CircuitPython. Unlike other objects in CircuitPython, Display objects live until *displayio.release_displays()* is called. This is done so that CircuitPython can use the display itself.

Most people should not use this class directly. Use a specific display driver instead that will contain the initialization sequence at minimum.

Create a Display object on the given display bus (*FourWire*, *ParallelBus* or *I2CDisplay*).

The `init_sequence` is bitpacked to minimize the ram impact. Every command begins with a command byte followed by a byte to determine the parameter count and delay. When the top bit of the second byte is 1 (0x80), a delay will occur after the command parameters are sent. The remaining 7 bits are the parameter count excluding any delay byte. The bytes following are the parameters. When the delay bit is set, a single byte after the parameters specifies the delay duration in milliseconds. The value 0xff will lead to an extra long 500 ms delay instead of 255 ms. The next byte will begin a new command definition. Here is an example:

```
init_sequence = (b"\xe1\x0f\x00\x0E\x14\x03\x11\x07\x31\xC1\x48\x08\x0F\x0C\x31\
↪x36\x0F" # Set Gamma
                 b"\x11\x80\x78"# Exit Sleep then delay 0x78 (120ms)
                 b"\x29\x81\xaa\x78"# Display on then delay 0x78 (120ms)
                )
display = displayio.Display(display_bus, init_sequence, width=320, height=240)
```

The first command is 0xe1 with 15 (0xf) parameters following. The second is 0x11 with 0 parameters and a 120ms (0x78) delay. The third command is 0x29 with one parameter 0xaa and a 120ms delay (0x78). Multiple byte literals (b"") are merged together on load. The parens are needed to allow byte literals on subsequent lines.

The initialization sequence should always leave the display memory access inline with the scan of the display to minimize tearing artifacts.

Parameters

- **display_bus** – The bus that the display is connected to
- **init_sequence** (ReadableBuffer) – Byte-packed initialization sequence.
- **width** (int) – Width in pixels
- **height** (int) – Height in pixels
- **colstart** (int) – The index if the first visible column
- **rowstart** (int) – The index if the first visible row
- **rotation** (int) – The rotation of the display in degrees clockwise. Must be in 90 degree increments (0, 90, 180, 270)
- **color_depth** (int) – The number of bits of color per pixel transmitted. (Some displays support 18 bit but 16 is easier to transmit. The last bit is extrapolated.)

- **grayscale** (bool) – True if the display only shows a single color.
- **pixels_in_byte_share_row** (bool) – True when pixels are less than a byte and a byte includes pixels from the same row of the display. When False, pixels share a column.
- **bytes_per_cell** (int) – Number of bytes per addressable memory location when color_depth < 8. When greater than one, bytes share a row or column according to pixels_in_byte_share_row.
- **reverse_pixels_in_byte** (bool) – Reverses the pixel order within each byte when color_depth < 8. Does not apply across multiple bytes even if there is more than one byte per cell (bytes_per_cell.)
- **reverse_bytes_in_word** (bool) – Reverses the order of bytes within a word when color_depth == 16
- **set_column_command** (int) – Command used to set the start and end columns to update
- **set_row_command** (int) – Command used so set the start and end rows to update
- **write_ram_command** (int) – Command used to write pixels values into the update region. Ignored if data_as_commands is set.
- **set_vertical_scroll** (int) – Command used to set the first row to show
- **backlight_pin** (microcontroller.Pin) – Pin connected to the display's backlight
- **brightness_command** (int) – Command to set display brightness. Usually available in OLED controllers.
- **brightness** (float) – Initial display brightness. This value is ignored if auto_brightness is True.
- **auto_brightness** (bool) – If True, brightness is controlled via an ambient light sensor or other mechanism.
- **single_byte_bounds** (bool) – Display column and row commands use single bytes
- **data_as_commands** (bool) – Treat all init and boundary data as SPI commands. Certain displays require this.
- **auto_refresh** (bool) – Automatically refresh the screen
- **native_frames_per_second** (int) – Number of display refreshes per second that occur with the given init_sequence.
- **backlight_on_high** (bool) – If True, pulling the backlight pin high turns the backlight on.
- **SH1107_addressing** (bool) – Special quirk for SH1107, use upper/lower column set and page set

auto_refresh :bool
True when the display is refreshed automatically.

brightness :float
The brightness of the display as a float. 0.0 is off and 1.0 is full brightness. When *auto_brightness* is True, the value of *brightness* will change automatically. If *brightness* is set, *auto_brightness* will be disabled and will be set to False.

auto_brightness :bool
True when the display brightness is adjusted automatically, based on an ambient light sensor or other

method. Note that some displays may have this set to True by default, but not actually implement automatic brightness adjustment. *auto_brightness* is set to False if *brightness* is set manually.

width :int
Gets the width of the board

height :int
Gets the height of the board

rotation :int
The rotation of the display as an int in degrees.

bus :_DisplayBus
The bus being used by the display

show(*self*, *group:* Group) → None
Switches to displaying the given group of layers. When group is None, the default CircuitPython terminal will be shown.

Parameters **group** (Group) – The group to show.

refresh(*self*, *, *target_frames_per_second: Optional[*int*] = None*, *minimum_frames_per_second:* int *= 1*) → *bool*
When auto refresh is off, waits for the target frame rate and then refreshes the display, returning True. If the call has taken too long since the last refresh call for the given target frame rate, then the refresh returns False immediately without updating the screen to hopefully help getting caught up.

If the time since the last successful refresh is below the minimum frame rate, then an exception will be raised. Set `minimum_frames_per_second` to 0 to disable.

When auto refresh is off, `display.refresh()` or `display.refresh(target_frames_per_second=None)` will update the display immediately.

When auto refresh is on, updates the display immediately. (The display will also update without calls to this.)

Parameters

- **target_frames_per_second** (int) – How many times a second *refresh* should be called and the screen updated. Set to None for immediate refresh.
- **minimum_frames_per_second** (int) – The minimum number of times the screen should be updated per second.

fill_row(*self*, *y:* int, *buffer:* _typing.WriteableBuffer) → *_typing.WriteableBuffer*
Extract the pixels from a single row

Parameters

- **y** (int) – The top edge of the area
- **buffer** (WriteableBuffer) – The buffer in which to place the pixel data

class displayio.**EPaperDisplay**(*display_bus:* _DisplayBus, *start_sequence:* _typing.ReadableBuffer, *stop_sequence:* _typing.ReadableBuffer, *, *width:* int, *height:* int, *ram_width:* int, *ram_height:* int, *colstart:* int = *0*, *rowstart:* int = *0*, *rotation:* int = *0*, *set_column_window_command:* Optional[int] = *None*, *set_row_window_command:* Optional[int] = *None*, *single_byte_bounds:* bool = *False*, *write_black_ram_command:* int, *black_bits_inverted:* bool = *False*, *write_color_ram_command:* Optional[int] = *None*, *color_bits_inverted:* bool = *False*, *highlight_color:* int = *0*, *refresh_display_command:* int, *refresh_time:* float = *40*, *busy_pin:* Optional[microcontroller.Pin] = *None*, *busy_state:* bool = *True*, *seconds_per_frame:* float = *180*, *always_toggle_chip_select:* bool = *False*, *grayscale:* bool = *False*)

Manage updating an epaper display over a display bus

This initializes an epaper display and connects it into CircuitPython. Unlike other objects in CircuitPython, EPaperDisplay objects live until *displayio.release_displays()* is called. This is done so that CircuitPython can use the display itself.

Most people should not use this class directly. Use a specific display driver instead that will contain the startup and shutdown sequences at minimum.

Create a EPaperDisplay object on the given display bus (*displayio.FourWire* or *displayio.ParallelBus*).

The `start_sequence` and `stop_sequence` are bitpacked to minimize the ram impact. Every command begins with a command byte followed by a byte to determine the parameter count and delay. When the top bit of the second byte is 1 (0x80), a delay will occur after the command parameters are sent. The remaining 7 bits are the parameter count excluding any delay byte. The bytes following are the parameters. When the delay bit is set, a single byte after the parameters specifies the delay duration in milliseconds. The value 0xff will lead to an extra long 500 ms delay instead of 255 ms. The next byte will begin a new command definition.

Parameters

- **display_bus** – The bus that the display is connected to
- **start_sequence** (ReadableBuffer) – Byte-packed initialization sequence.
- **stop_sequence** (ReadableBuffer) – Byte-packed initialization sequence.
- **width** (int) – Width in pixels
- **height** (int) – Height in pixels
- **ram_width** (int) – RAM width in pixels
- **ram_height** (int) – RAM height in pixels
- **colstart** (int) – The index if the first visible column
- **rowstart** (int) – The index if the first visible row
- **rotation** (int) – The rotation of the display in degrees clockwise. Must be in 90 degree increments (0, 90, 180, 270)
- **set_column_window_command** (int) – Command used to set the start and end columns to update
- **set_row_window_command** (int) – Command used so set the start and end rows to update

- **set_current_column_command** (int) – Command used to set the current column location
- **set_current_row_command** (int) – Command used to set the current row location
- **write_black_ram_command** (int) – Command used to write pixels values into the update region
- **black_bits_inverted** (bool) – True if 0 bits are used to show black pixels. Otherwise, 1 means to show black.
- **write_color_ram_command** (int) – Command used to write pixels values into the update region
- **color_bits_inverted** (bool) – True if 0 bits are used to show the color. Otherwise, 1 means to show color.
- **highlight_color** (int) – RGB888 of source color to highlight with third ePaper color.
- **refresh_display_command** (int) – Command used to start a display refresh
- **refresh_time** (float) – Time it takes to refresh the display before the stop_sequence should be sent. Ignored when busy_pin is provided.
- **busy_pin** (microcontroller.Pin) – Pin used to signify the display is busy
- **busy_state** (bool) – State of the busy pin when the display is busy
- **seconds_per_frame** (float) – Minimum number of seconds between screen refreshes
- **always_toggle_chip_select** (bool) – When True, chip select is toggled every byte
- **grayscale** (bool) – When true, the color ram is the low bit of 2-bit grayscale

time_to_refresh :float
Time, in fractional seconds, until the ePaper display can be refreshed.

busy :bool
True when the display is refreshing. This uses the busy_pin when available or the refresh_time otherwise.

width :int
Gets the width of the display in pixels

height :int
Gets the height of the display in pixels

rotation :int
The rotation of the display as an int in degrees.

bus :_DisplayBus
The bus being used by the display

show(*self*, *group:* Group) → None
Switches to displaying the given group of layers. When group is None, the default CircuitPython terminal will be shown.

Parameters **group** (Group) – The group to show.

refresh(*self*) → None
Refreshes the display immediately or raises an exception if too soon. Use time.sleep(display.time_to_refresh) to sleep until a refresh can occur.

class displayio.**FourWire**(*spi_bus:* busio.SPI, *, *command:* microcontroller.Pin, *chip_select:* microcontroller.Pin, *reset: Optional[*microcontroller.Pin*] = None*, *baudrate:* int *= 24000000*, *polarity:* int *= 0*, *phase:* int *= 0*)

Manage updating a display over SPI four wire protocol in the background while Python code runs. It doesn't handle display initialization.

Create a FourWire object associated with the given pins.

The SPI bus and pins are then in use by the display until *displayio.release_displays()* is called even after a reload. (It does this so CircuitPython can use the display after your code is done.) So, the first time you initialize a display bus in code.py you should call *displayio.release_displays()* first, otherwise it will error after the first code.py run.

Parameters

- **spi_bus** (busio.SPI) – The SPI bus that make up the clock and data lines
- **command** (microcontroller.Pin) – Data or command pin
- **chip_select** (microcontroller.Pin) – Chip select pin
- **reset** (microcontroller.Pin) – Reset pin. When None only software reset can be used
- **baudrate** (int) – Maximum baudrate in Hz for the display on the bus
- **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.

reset(*self*) → None

Performs a hardware reset via the reset pin. Raises an exception if called when no reset pin is available.

send(*self*, *command:* int, *data:* FourWire, *, *toggle_every_byte:* bool *= False*) → None

Sends the given command value followed by the full set of data. Display state, such as vertical scroll, set via send may or may not be reset once the code is done.

class displayio.**Group**(*, *max_size:* int *= 4*, *scale:* int *= 1*, *x:* int *= 0*, *y:* int *= 0*)

Manage a group of sprites and groups and how they are inter-related.

Create a Group of a given size and scale. Scale is in one dimension. For example, scale=2 leads to a layer's pixel being 2x2 pixels when in the group.

Parameters

- **max_size** (int) – Ignored. Will be removed in 7.x.
- **scale** (int) – Scale of layer pixels in one dimension.
- **x** (int) – Initial x position within the parent.
- **y** (int) – Initial y position within the parent.

hidden :bool

True when the Group and all of it's layers are not visible. When False, the Group's layers are visible if they haven't been hidden.

scale :int

Scales each pixel within the Group in both directions. For example, when scale=2 each pixel will be represented by 2x2 pixels.

x :int

X position of the Group in the parent.

y :int
Y position of the Group in the parent.

append(*self*, *layer: Union[*vectorio.VectorShape, Group, TileGrid*]*) → None
Append a layer to the group. It will be drawn above other layers.

insert(*self*, *index:* int, *layer: Union[*vectorio.VectorShape, Group, TileGrid*]*) → None
Insert a layer into the group.

index(*self*, *layer: Union[*vectorio.VectorShape, Group, TileGrid*]*) → *int*
Returns the index of the first copy of layer. Raises ValueError if not found.

pop(*self*, *i:* int = *- 1*) → Union[*vectorio.VectorShape*, *Group*, *TileGrid*]
Remove the ith item and return it.

remove(*self*, *layer: Union[*vectorio.VectorShape, Group, TileGrid*]*) → None
Remove the first copy of layer. Raises ValueError if it is not present.

__bool__(*self*) → *bool*

__len__(*self*) → *int*
Returns the number of layers in a Group

__getitem__(*self*, *index:* int) → Union[*vectorio.VectorShape*, *Group*, *TileGrid*]
Returns the value at the given index.

This allows you to:

```
print(group[0])
```

__setitem__(*self*, *index:* int, *value: Union[*vectorio.VectorShape, Group, TileGrid*]*) → None
Sets the value at the given index.

This allows you to:

```
group[0] = sprite
```

__delitem__(*self*, *index:* int) → None
Deletes the value at the given index.

This allows you to:

```
del group[0]
```

sort(*self*, *key: function*, *reverse:* bool) → None
Sort the members of the group.

class displayio.**I2CDisplay**(*i2c_bus:* busio.I2C, *, *device_address:* int, *reset: Optional[*microcontroller.Pin*] = None*)
Manage updating a display over I2C in the background while Python code runs. It doesn't handle display initialization.

Create a I2CDisplay object associated with the given I2C bus and reset pin.

The I2C bus and pins are then in use by the display until *displayio.release_displays()* is called even after a reload. (It does this so CircuitPython can use the display after your code is done.) So, the first time you initialize a display bus in code.py you should call *displayio.release_displays()* first, otherwise it will error after the first code.py run.

Parameters

- **i2c_bus** (busio.I2C) – The I2C bus that make up the clock and data lines

- **device_address** (int) – The I2C address of the device
- **reset** (microcontroller.Pin) – Reset pin. When None only software reset can be used

reset(*self*) → None
Performs a hardware reset via the reset pin. Raises an exception if called when no reset pin is available.

send(*self*, *command:* int, *data:* _typing.ReadableBuffer) → None
Sends the given command value followed by the full set of data. Display state, such as vertical scroll, set via send may or may not be reset once the code is done.

class displayio.**OnDiskBitmap**(*file: BinaryIO*)
Loads values straight from disk. This minimizes memory use but can lead to much slower pixel load times. These load times may result in frame tearing where only part of the image is visible.

It's easiest to use on a board with a built in display such as the Hallowing M0 Express.

```
import board
import displayio
import time
import pulseio

board.DISPLAY.auto_brightness = False
board.DISPLAY.brightness = 0
splash = displayio.Group()
board.DISPLAY.show(splash)

with open("/sample.bmp", "rb") as f:
    odb = displayio.OnDiskBitmap(f)
    face = displayio.TileGrid(odb, pixel_shader=odb.pixel_shader)
    splash.append(face)
    # Wait for the image to load.
    board.DISPLAY.refresh(target_frames_per_second=60)

    # Fade up the backlight
    for i in range(100):
        board.DISPLAY.brightness = 0.01 * i
        time.sleep(0.05)

    # Wait forever
    while True:
        pass
```

Create an OnDiskBitmap object with the given file.

Parameters **file** (*file*) – The open bitmap file

width :int
Width of the bitmap. (read only)

height :int
Height of the bitmap. (read only)

pixel_shader :Union[ColorConverter, Palette]
The image's pixel_shader. The type depends on the underlying bitmap's structure. The pixel shadder can be modified (e.g., to set the transparent pixel or, for paletted images, to update the palette

class displayio.**Palette**(*color_count:* int)
Map a pixel palette_index to a full color. Colors are transformed to the display's format internally to save memory.

Create a Palette object to store a set number of colors.

Parameters **color_count** (int) – The number of colors in the Palette

__bool__(*self*) → *bool*

__len__(*self*) → *int*
Returns the number of colors in a Palette

__getitem__(*self*, *index:* int) → Optional[*int*]
Return the pixel color at the given index as an integer.

__setitem__(*self*, *index:* int, *value: Union[*int, _typing.ReadableBuffer, *Tuple[*int, int, int*]]*) → None
Sets the pixel color at the given index. The index should be an integer in the range 0 to color_count-1.

The value argument represents a color, and can be from 0x000000 to 0xFFFFFF (to represent an RGB value). Value can be an int, bytes (3 bytes (RGB) or 4 bytes (RGB + pad byte)), bytearray, or a tuple or list of 3 integers.

This allows you to:

```
palette[0] = 0xFFFFFF                       # set using an integer
palette[1] = b'\xff\xff\x00'                # set using 3 bytes
palette[2] = b'\xff\xff\x00\x00'            # set using 4 bytes
palette[3] = bytearray(b'\x00\x00\xFF')     # set using a bytearay of 3 or 4␣
↪bytes
palette[4] = (10, 20, 30)                   # set using a tuple of 3 integers
```

make_transparent(*self*, *palette_index:* int) → None

make_opaque(*self*, *palette_index:* int) → None

is_transparent(*self*, *palette_index:* int) → *bool*
Returns `True` if the palette index is transparent. Returns `False` if opaque.

class displayio.**ParallelBus**(*, *data0:* microcontroller.Pin, *command:* microcontroller.Pin, *chip_select:* microcontroller.Pin, *write:* microcontroller.Pin, *read:* microcontroller.Pin, *reset:* microcontroller.Pin)
Manage updating a display over 8-bit parallel bus in the background while Python code runs. This protocol may be refered to as 8080-I Series Parallel Interface in datasheets. It doesn't handle display initialization.

Create a ParallelBus object associated with the given pins. The bus is inferred from data0 by implying the next 7 additional pins on a given GPIO port.

The parallel bus and pins are then in use by the display until *displayio.release_displays()* is called even after a reload. (It does this so CircuitPython can use the display after your code is done.) So, the first time you initialize a display bus in code.py you should call *displayio.release_displays()* first, otherwise it will error after the first code.py run.

Parameters

- **data0** (microcontroller.Pin) – The first data pin. The rest are implied
- **command** (microcontroller.Pin) – Data or command pin
- **chip_select** (microcontroller.Pin) – Chip select pin
- **write** (microcontroller.Pin) – Write pin
- **read** (microcontroller.Pin) – Read pin
- **reset** (microcontroller.Pin) – Reset pin

reset(*self*) → None
Performs a hardware reset via the reset pin. Raises an exception if called when no reset pin is available.

send(*self*, *command:* int, *data:* _typing.ReadableBuffer) → None
Sends the given command value followed by the full set of data. Display state, such as vertical scroll, set via `send` may or may not be reset once the code is done.

class displayio.**Shape**(*width:* int, *height:* int, *, *mirror_x:* bool = *False*, *mirror_y:* bool = *False*)
Represents a shape made by defining boundaries that may be mirrored.

Create a Shape object with the given fixed size. Each pixel is one bit and is stored by the column boundaries of the shape on each row. Each row's boundary defaults to the full row.

Parameters

- **width** (int) – The number of pixels wide
- **height** (int) – The number of pixels high
- **mirror_x** (bool) – When true the left boundary is mirrored to the right.
- **mirror_y** (bool) – When true the top boundary is mirrored to the bottom.

set_boundary(*self*, *y:* int, *start_x:* int, *end_x:* int) → None
Loads pre-packed data into the given row.

class displayio.**TileGrid**(*bitmap:* Bitmap, *, *pixel_shader: Union[*ColorConverter, Palette*]*, *width:* int = *1*, *height:* int = *1*, *tile_width: Optional[*int*] = None*, *tile_height: Optional[*int*] = None*, *default_tile:* int = *0*, *x:* int = *0*, *y:* int = *0*)
A grid of tiles sourced out of one bitmap

Position a grid of tiles sourced from a bitmap and pixel_shader combination. Multiple grids can share bitmaps and pixel shaders.

A single tile grid is also known as a Sprite.

Create a TileGrid object. The bitmap is source for 2d pixels. The pixel_shader is used to convert the value and its location to a display native pixel color. This may be a simple color palette lookup, a gradient, a pattern or a color transformer.

tile_width and tile_height match the height of the bitmap by default.

Parameters

- **bitmap** (Bitmap) – The bitmap storing one or more tiles.
- **pixel_shader** (ColorConverter, Palette) – The pixel shader that produces colors from values
- **width** (int) – Width of the grid in tiles.
- **height** (int) – Height of the grid in tiles.
- **tile_width** (int) – Width of a single tile in pixels. Defaults to the full Bitmap and must evenly divide into the Bitmap's dimensions.
- **tile_height** (int) – Height of a single tile in pixels. Defaults to the full Bitmap and must evenly divide into the Bitmap's dimensions.
- **default_tile** (int) – Default tile index to show.
- **x** (int) – Initial x position of the left edge within the parent.
- **y** (int) – Initial y position of the top edge within the parent.

hidden :bool
True when the TileGrid is hidden. This may be False even when a part of a hidden Group.

x :int
X position of the left edge in the parent.

y :int
Y position of the top edge in the parent.

flip_x :bool
If true, the left edge rendered will be the right edge of the right-most tile.

flip_y :bool
If true, the top edge rendered will be the bottom edge of the bottom-most tile.

transpose_xy :bool
If true, the TileGrid's axis will be swapped. When combined with mirroring, any 90 degree rotation can be achieved along with the corresponding mirrored version.

pixel_shader :Union[ColorConverter, Palette]
The pixel shader of the tilegrid.

__getitem__(*self*, *index: Union[Tuple[*int, int*], *int*]*) → *int*
Returns the tile index at the given index. The index can either be an x,y tuple or an int equal to `y * width + x`.

This allows you to:

```
print(grid[0])
```

__setitem__(*self*, *index: Union[Tuple[*int, int*], *int*]*, *value:* int) → None
Sets the tile index at the given index. The index can either be an x,y tuple or an int equal to `y * width + x`.

This allows you to:

```
grid[0] = 10
```

or:

```
grid[0,0] = 10
```

dualbank – DUALBANK Module

The *dualbank* module adds ability to update and switch between the two app partitions.

There are two identical partitions, these contain different firmware versions. Having two partitions enables rollback functionality.

The two partitions are defined as boot partition and next-update partition. Calling *dualbank.flash()* writes the next-update partition.

After the next-update partition is written a validation check is performed and on a successful validation this partition is set as the boot partition. On next reset, firmware will be loaded from this partition.

Here is the sequence of commands to follow:

```
import dualbank

dualbank.flash(buffer, offset)
dualbank.switch()
```

dualbank.**flash**(**buffer:* _typing.ReadableBuffer, *offset:* int = *0*) → None
Writes one of two app partitions at the given offset.

This can be called multiple times when flashing the firmware in small chunks.

dualbank.**switch**() → None
Switches the boot partition.

On next reset, firmware will be loaded from the partition just switched over to.

fontio – Core font related data structures

class fontio.**BuiltinFont**
A font built into CircuitPython

Creation not supported. Available fonts are defined when CircuitPython is built. See the Adafruit_CircuitPython_Bitmap_Font library for dynamically loaded fonts.

bitmap :displayio.Bitmap
Bitmap containing all font glyphs starting with ASCII and followed by unicode. Use *get_glyph* in most cases. This is useful for use with *displayio.TileGrid* and *terminalio.Terminal*.

get_bounding_box(*self*) → Tuple[*int*, *int*]
Returns the maximum bounds of all glyphs in the font in a tuple of two values: width, height.

get_glyph(*self*, *codepoint:* int) → *Glyph*
Returns a *fontio.Glyph* for the given codepoint or None if no glyph is available.

class fontio.**Glyph**(*bitmap:* displayio.Bitmap, *tile_index:* int, *width:* int, *height:* int, *dx:* int, *dy:* int, *shift_x:* int, *shift_y:* int)
Storage of glyph info

Named tuple used to capture a single glyph and its attributes.

Parameters

- **bitmap** – the bitmap including the glyph
- **tile_index** – the tile index within the bitmap
- **width** – the width of the glyph's bitmap
- **height** – the height of the glyph's bitmap
- **dx** – x adjustment to the bitmap's position
- **dy** – y adjustment to the bitmap's position
- **shift_x** – the x difference to the next glyph
- **shift_y** – the y difference to the next glyph

framebufferio – Native framebuffer display driving

The *framebufferio* module contains classes to manage display output including synchronizing with refresh rates and partial updating. It is used in conjunction with classes from *displayio* to actually place items on the display; and classes like *RGBMatrix* to actually drive the display.

class framebufferio.**FramebufferDisplay**(*framebuffer:* _typing.FrameBuffer, *, *rotation:* int = *0*, *auto_refresh:* bool = *True*)
Manage updating a display with framebuffer in RAM

This initializes a display and connects it into CircuitPython. Unlike other objects in CircuitPython, Display objects live until *displayio.release_displays()* is called. This is done so that CircuitPython can use the display itself.

Create a Display object with the given framebuffer (a buffer, array, ulab.array, etc)

Parameters

- **framebuffer** (FrameBuffer) – The framebuffer that the display is connected to
- **auto_refresh** (bool) – Automatically refresh the screen
- **rotation** (int) – The rotation of the display in degrees clockwise. Must be in 90 degree increments (0, 90, 180, 270)

auto_refresh :bool
True when the display is refreshed automatically.

brightness :float
The brightness of the display as a float. 0.0 is off and 1.0 is full brightness. When *auto_brightness* is True, the value of *brightness* will change automatically. If *brightness* is set, *auto_brightness* will be disabled and will be set to False.

auto_brightness :bool
True when the display brightness is adjusted automatically, based on an ambient light sensor or other method. Note that some displays may have this set to True by default, but not actually implement automatic brightness adjustment. *auto_brightness* is set to False if *brightness* is set manually.

width :int
Gets the width of the framebuffer

height :int
Gets the height of the framebuffer

rotation :int
The rotation of the display as an int in degrees.

framebuffer :_typing.FrameBuffer
The framebuffer being used by the display

show(*self*, *group:* displayio.Group) → None
Switches to displaying the given group of layers. When group is None, the default CircuitPython terminal will be shown.

Parameters **group** (*Group*) – The group to show.

refresh(*self*, *, *target_frames_per_second:* int = *60*, *minimum_frames_per_second:* int = *1*) → *bool*
When auto refresh is off, waits for the target frame rate and then refreshes the display, returning True. If the call has taken too long since the last refresh call for the given target frame rate, then the refresh returns False immediately without updating the screen to hopefully help getting caught up.

If the time since the last successful refresh is below the minimum frame rate, then an exception will be raised. Set minimum_frames_per_second to 0 to disable.

When auto refresh is on, updates the display immediately. (The display will also update without calls to this.)

Parameters

- **target_frames_per_second** (int) – How many times a second *refresh* should be called and the screen updated.
- **minimum_frames_per_second** (int) – The minimum number of times the screen should be updated per second.

fill_row(*self*, *y:* int, *buffer:* _typing.WriteableBuffer) → *_typing.WriteableBuffer*
Extract the pixels from a single row

Parameters

- **y** (int) – The top edge of the area
- **buffer** (WriteableBuffer) – The buffer in which to place the pixel data

frequencyio – Support for frequency based protocols

Warning: This module is not available in SAMD21 builds. See the *Module Support Matrix - Which Modules Are Available on Which Boards* 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 frequencyio
import time
from board import *

frequency = frequencyio.FrequencyIn(D13)
frequency.capture_period = 15
time.sleep(0.1)
```

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

class frequencyio.**FrequencyIn**(*pin:* microcontroller.Pin, *capture_period:* int = *10*)

Read a frequency signal

FrequencyIn is used to measure the frequency, in hertz, of a digital signal on an incoming pin. Accuracy has shown to be within 10%, if not better. It is recommended to utilize an average of multiple samples to smooth out readings.

Frequencies below 1KHz are not currently detectable.

FrequencyIn will not determine pulse width (use `PulseIn`).

Create a FrequencyIn object associated with the given pin.

Parameters

- **pin** (Pin) – Pin to read frequency from.
- **capture_period** (int) – Keyword argument to set the measurement period, in milliseconds. Default is 10ms; range is 1ms - 500ms.

Read the incoming frequency from a pin:

```
import frequencyio
import board

frequency = frequencyio.FrequencyIn(board.D11)

# Loop while printing the detected frequency
while True:
    print(frequency.value)
```

(continues on next page)

(continued from previous page)

```
    # Optional clear() will reset the value
    # to zero. Without this, if the incoming
    # signal stops, the last reading will remain
    # as the value.
    frequency.clear()
```

capture_period :int
The capture measurement period. Lower incoming frequencies will be measured more accurately with longer capture periods. Higher frequencies are more accurate with shorter capture periods.

Note: When setting a new `capture_period`, all previous capture information is cleared with a call to `clear()`.

---

deinit(*self*) → None
Deinitialises the FrequencyIn and releases any hardware resources for reuse.

__enter__(*self*) → *FrequencyIn*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

pause(*self*) → None
Pause frequency capture.

resume(*self*) → None
Resumes frequency capture.

clear(*self*) → None
Clears the last detected frequency capture value.

__get__(*self*, *index:* int) → *int*
Returns the value of the last frequency captured.

gamepad – Button handling in the background

class gamepad.**GamePad**(*b1:* digitalio.DigitalInOut, *b2:* digitalio.DigitalInOut, *b3:* digitalio.DigitalInOut, *b4:* digitalio.DigitalInOut, *b5:* digitalio.DigitalInOut, *b6:* digitalio.DigitalInOut, *b7:* digitalio.DigitalInOut, *b8:* digitalio.DigitalInOut)
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),
```

(continues on next page)

(continued from previous page)

```
    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)
```

Initializes button scanning routines.

The `b1`-`b8` parameters are `DigitalInOut` objects, which immediately get switched to input with a pull-up, (unless they already were set to pull-down, in which case they remain so), 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(*self*) → *int*

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(*self*) → None

Disable button scanning.

gamepadshift – Tracks button presses read through a shift register

class gamepadshift.**GamePadShift**(*clock:* digitalio.DigitalInOut, *data:* digitalio.DigitalInOut, *latch:* digitalio.DigitalInOut)

Scan buttons for presses through a shift register

Initializes button scanning routines.

The `clock`, `data` and `latch` parameters are `DigitalInOut` objects connected to the shift register controlling the buttons.

The 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.

Only one gamepad (*gamepad.GamePad* or *gamepadshift.GamePadShift*) may be used at a time.

get_pressed(*self*) → *int*
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(*self*) → None
Disable button scanning.

gnss – Global Navigation Satellite System

The *gnss* module contains classes to control the GNSS and acquire positioning information.

class gnss.**GNSS**(*system: Union[*SatelliteSystem*, List[*SatelliteSystem*]]*)
Get updated positioning information from Global Navigation Satellite System (GNSS)

Usage:

```
import gnss
import time

nav = gnss.GNSS([gnss.SatelliteSystem.GPS, gnss.SatelliteSystem.GLONASS])
last_print = time.monotonic()
while True:
    nav.update()
    current = time.monotonic()
    if current - last_print >= 1.0:
        last_print = current
        if nav.fix is gnss.PositionFix.INVALID:
            print("Waiting for fix...")
            continue
        print("Latitude: {0:.6f} degrees".format(nav.latitude))
        print("Longitude: {0:.6f} degrees".format(nav.longitude))
```

Turn on the GNSS.

Parameters **system** – satellite system to use

latitude :float
Latitude of current position in degrees (float).

longitude :float
Longitude of current position in degrees (float).

altitude :float
Altitude of current position in meters (float).

timestamp :time.struct_time
Time when the position data was updated.

fix :PositionFix
Fix mode.

deinit(*self*) → None
Turn off the GNSS.

update(*self*) → None
Update GNSS positioning information.

class gnss.**PositionFix**
Position fix mode

Enum-like class to define the position fix mode.

INVALID :PositionFix
No measurement.

FIX_2D :PositionFix
2D fix.

FIX_3D :PositionFix
3D fix.

class gnss.**SatelliteSystem**
Satellite system type

Enum-like class to define the satellite system type.

GPS :SatelliteSystem
Global Positioning System.

GLONASS :SatelliteSystem
GLObal NAvigation Satellite System.

SBAS :SatelliteSystem
Satellite Based Augmentation System.

QZSS_L1CA :SatelliteSystem
Quasi-Zenith Satellite System L1C/A.

QZSS_L1S :SatelliteSystem
Quasi-Zenith Satellite System L1S.

i2cperipheral – Two wire serial protocol peripheral

The *i2cperipheral* module contains classes to support an I2C peripheral.

Example emulating a peripheral with 2 addresses (read and write):

```
import board
from i2cperipheral import I2CPeripheral

regs = [0] * 16
index = 0

with I2CPeripheral(board.SCL, board.SDA, (0x40, 0x41)) as device:
    while True:
        r = device.request()
        if not r:
            # Maybe do some housekeeping
            continue
        with r:  # Closes the transfer if necessary by sending a NACK or feeding␣
↪dummy bytes
            if r.address == 0x40:
                if not r.is_read:  # Main write which is Selected read
                    b = r.read(1)
                    if not b or b[0] > 15:
                        break
                    index = b[0]
```

(continues on next page)

(continued from previous page)

```
                    b = r.read(1)
                    if b:
                        regs[index] = b[0]
                elif r.is_restart:  # Combined transfer: This is the Main read message
                    n = r.write(bytes([regs[index]]))
                #else:
                    # A read transfer is not supported in this example
                    # If the microcontroller tries, it will get 0xff byte(s) by the␣
↪ctx manager (r.close())
            elif r.address == 0x41:
                if not r.is_read:
                    b = r.read(1)
                    if b and b[0] == 0xde:
                        # do something
                        pass
```

This example sets up an I2C device that can be accessed from Linux like this:

```
$ i2cget -y 1 0x40 0x01
0x00
$ i2cset -y 1 0x40 0x01 0xaa
$ i2cget -y 1 0x40 0x01
0xaa
```

Warning: I2CPeripheral makes use of clock stretching in order to slow down the host. Make sure the I2C host supports this.

Raspberry Pi in particular does not support this with its I2C hw block. This can be worked around by using the `i2c-gpio` bit banging driver. Since the RPi firmware uses the hw i2c, it's not possible to emulate a HAT eeprom.

class i2cperipheral.**I2CPeripheral**(*scl:* microcontroller.Pin, *sda:* microcontroller.Pin, *addresses: Sequence[*int*]*, *smbus:* bool = *False*)

Two wire serial protocol peripheral

I2C is a two-wire protocol for communicating between devices. This implements the peripheral (sensor, secondary) side.

Parameters

- **scl** (Pin) – The clock pin
- **sda** (Pin) – The data pin
- **addresses** (list[int]) – The I2C addresses to respond to (how many is hw dependent).
- **smbus** (bool) – Use SMBUS timings if the hardware supports it

deinit(*self*) → None

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

__enter__(*self*) → *I2CPeripheral*

No-op used in Context Managers.

__exit__(*self*) → None

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

request (*self*, *timeout:* float = - *1*) → *I2CPeripheralRequest*
Wait for an I2C request.

Parameters **timeout** (float) – Timeout in seconds. Zero means wait forever, a negative value means check once

Returns I2C Slave Request or None if timeout=-1 and there's no request

Return type *I2CPeripheralRequest*

class i2cperipheral.**I2CPeripheralRequest** (*peripheral:* I2CPeripheral, *address:* int, *is_read:* bool, *is_restart:* bool)
Information about an I2C transfer request This cannot be instantiated directly, but is returned by *I2CPeripheral.request()*.

Parameters

- **peripheral** – The I2CPeripheral object receiving this request
- **address** – I2C address
- **is_read** – True if the main peripheral is requesting data
- **is_restart** – Repeated Start Condition

address :int
The I2C address of the request.

is_read :bool
The I2C main controller is reading from this peripheral.

is_restart :bool
Is Repeated Start Condition.

__enter__ (*self*) → *I2CPeripheralRequest*
No-op used in Context Managers.

__exit__ (*self*) → None
Close the request.

read (*self*, *n:* int = - *1*, *ack:* bool = *True*) → *bytearray*
Read data. If ack=False, the caller is responsible for calling *I2CPeripheralRequest.ack()*.

Parameters

- **n** – Number of bytes to read (negative means all)
- **ack** – Whether or not to send an ACK after the n'th byte

Returns Bytes read

write (*self*, *buffer:* _typing.ReadableBuffer) → *int*
Write the data contained in buffer.

Parameters **buffer** (ReadableBuffer) – Write out the data in this buffer

Returns Number of bytes written

ack (*self*, *ack:* bool = *True*) → None
Acknowledge or Not Acknowledge last byte received. Use together with *I2CPeripheralRequest.read()* ack=False.

Parameters **ack** – Whether to send an ACK or NACK

imagecapture – Support for "Parallel capture" interfaces

class imagecapture.**ParallelImageCapture**(*, *data_pins:* List[microcontroller.Pin], *clock:* microcontroller.Pin, *vsync:* Optional[microcontroller.Pin], *href:* Optional[microcontroller.Pin])

Capture image frames from a camera with parallel data interface

Create a parallel image capture object

Parameters

- **data_pins** (*List[microcontroller.Pin]*) – The data pins.
- **clock** (microcontroller.Pin) – The pixel clock input.
- **vsync** (microcontroller.Pin) – The vertical sync input, which has a negative-going pulse at the beginning of each frame.
- **href** (microcontroller.Pin) – The horizontal reference input, which is high whenever the camera is transmitting valid pixel information.

capture(*self*, *buffer:* _typing.WriteableBuffer, *width:* int, *height:* int, *bpp:* int = *16*) → None
Capture a single frame into the given buffer

deinit(*self*) → None
Deinitialize this instance

__enter__(*self*) → *ParallelImageCapture*
No-op used in Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware on context exit. See *Lifetime and ContextManagers* for more info.

ipaddress

The *ipaddress* module provides types for IP addresses. It is a subset of CPython's ipaddress module.

ipaddress.**ip_address**(*obj: Union[int, str]*) → *IPv4Address*
Return a corresponding IP address object or raise ValueError if not possible.

class ipaddress.**IPv4Address**(*address: Union[int, str, bytes]*)
Encapsulates an IPv4 address.

Create a new IPv4Address object encapsulating the address value.

The value itself can either be bytes or a string formatted address.

packed :bytes
The bytes that make up the address (read-only).

version :int
4 for IPv4, 6 for IPv6

__eq__(*self*, *other:* object) → *bool*
Two Address objects are equal if their addresses and address types are equal.

__hash__(*self*) → *int*
Returns a hash for the IPv4Address data.

keypad – Support for scanning keys and key matrices

The *keypad* module provides native support to scan sets of keys or buttons, connected independently to individual pins, connected to a shift register, or connected in a row-and-column matrix.

class keypad.**Event**(*key_number:* int = *0*, *pressed:* bool = *True*)
A key transition event.

Create a key transition event, which reports a key-pressed or key-released transition.

Parameters

- **key_number** (int) – the key number
- **pressed** (bool) – True if the key was pressed; False if it was released.

key_number :int
The key number.

pressed :bool
True if the event represents a key down (pressed) transition. The opposite of *released*.

released :bool
True if the event represents a key up (released) transition. The opposite of *pressed*.

__eq__(*self*, *other:* object) → *bool*
Two *Event* objects are equal if their *key_number* and *pressed*/*released* values are equal.

__hash__(*self*) → *int*
Returns a hash for the *Event*, so it can be used in dictionaries, etc..

class keypad.**EventQueue**
A queue of *Event* objects, filled by a *keypad* scanner such as *Keys* or *KeyMatrix*.

You cannot create an instance of *EventQueue* directly. Each scanner creates an instance when it is created.

overflowed :bool
True if an event could not be added to the event queue because it was full. (read-only) Set to False by *clear()*.

get(*self*) → Optional[*Event*]
Return the next key transition event. Return None if no events are pending.

Note that the queue size is limited; see max_events in the constructor of a scanner such as *Keys* or *KeyMatrix*. If a new event arrives when the queue is full, the event is discarded, and *overflowed* is set to True.

Returns the next queued key transition *Event*

Return type Optional[*Event*]

get_into(*self*, *event:* Event) → *bool*
Store the next key transition event in the supplied event, if available, and return True. If there are no queued events, do not touch event and return False.

The advantage of this method over get() is that it does not allocate storage. Instead you can reuse an existing Event object.

Note that the queue size is limited; see max_events in the constructor of a scanner such as *Keys* or *KeyMatrix*.

:return True if an event was available and stored, False if not. :rtype: bool

clear(*self*) → None
Clear any queued key transition events. Also sets *overflowed* to False.

__bool__(*self*) → *bool*
`True` if *`len()`* is greater than zero. This is an easy way to check if the queue is empty.

__len__(*self*) → *int*
Return the number of events currently in the queue. Used to implement `len()`.

class keypad.**KeyMatrix**(*row_pins: Sequence[microcontroller.Pin], column_pins: Sequence[microcontroller.Pin], columns_to_anodes: bool = True, interval: float = 0.02, max_events: int = 64*)
Manage a 2D matrix of keys with row and column pins.

Create a *`Keys`* object that will scan the key matrix attached to the given row and column pins. There should not be any external pull-ups or pull-downs on the matrix: `KeyMatrix` enables internal pull-ups or pull-downs on the pins as necessary.

The keys are numbered sequentially from zero. A key number can be computed by `row * len(column_pins) + column`.

An *`EventQueue`* is created when this object is created and is available in the *`events`* attribute.

Parameters

- **row_pins** (*`Sequence[microcontroller.Pin]`*) – The pins attached to the rows.
- **column_pins** (*`Sequence[microcontroller.Pin]`*) – The pins attached to the colums.
- **columns_to_anodes** (`bool`) – Default `True`. If the matrix uses diodes, the diode anodes are typically connected to the column pins, and the cathodes should be connected to the row pins. If your diodes are reversed, set `columns_to_anodes` to `False`.
- **interval** (`float`) – Scan keys no more often than `interval` to allow for debouncing. `interval` is in float seconds. The default is 0.020 (20 msecs).
- **max_events** (`int`) – maximum size of *`events EventQueue`*: maximum number of key transition events that are saved. Must be >= 1. If a new event arrives when the queue is full, the oldest event is discarded.

key_count :int
The number of keys that are being scanned. (read-only)

events :EventQueue
The *`EventQueue`* associated with this *`Keys`* object. (read-only)

deinit(*self*) → None
Stop scanning and release the pins.

__enter__(*self*) → *KeyMatrix*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes when exiting a context. See *Lifetime and ContextManagers* for more info.

reset(*self*) → None
Reset the internal state of the scanner to assume that all keys are now released. Any key that is already pressed at the time of this call will therefore immediately cause a new key-pressed event to occur.

key_number_to_row_column(*self*, *row: int*, *column: int*) → Tuple[*int*]
Return the row and column for the given key number. The row is `key_number // len(column_pins)`. The column is `key_number % len(column_pins)`.

Returns `(row, column)`

Return type Tuple[*int*]

row_column_to_key_number(*self*, *row:* int, *column:* int) → *int*
Return the key number for a given row and column. The key number is `row * len(column_pins) + column`.

class keypad.**Keys**(*pins: Sequence[microcontroller.Pin]*, ***, *value_when_pressed:* bool, *pull:* bool = *True*, *interval:* float = *0.02*, *max_events:* int = *64*)
Manage a set of independent keys.

Create a *Keys* object that will scan keys attached to the given sequence of pins. Each key is independent and attached to its own pin.

An *EventQueue* is created when this object is created and is available in the *events* attribute.

Parameters

- **pins** (*Sequence[microcontroller.Pin]*) – The pins attached to the keys. The key numbers correspond to indices into this sequence.
- **value_when_pressed** (bool) – `True` if the pin reads high when the key is pressed. `False` if the pin reads low (is grounded) when the key is pressed. All the pins must be connected in the same way.
- **pull** (bool) – `True` if an internal pull-up or pull-down should be enabled on each pin. A pull-up will be used if `value_when_pressed` is `False`; a pull-down will be used if it is `True`. If an external pull is already provided for all the pins, you can set `pull` to `False`. However, enabling an internal pull when an external one is already present is not a problem; it simply uses slightly more current.
- **interval** (float) – Scan keys no more often than `interval` to allow for debouncing. `interval` is in float seconds. The default is 0.020 (20 msecs).
- **max_events** (int) – maximum size of *events EventQueue*: maximum number of key transition events that are saved. Must be >= 1. If a new event arrives when the queue is full, the oldest event is discarded.

key_count :int
The number of keys that are being scanned. (read-only)

events :EventQueue
The *EventQueue* associated with this *Keys* object. (read-only)

deinit(*self*) → None
Stop scanning and release the pins.

__enter__(*self*) → *Keys*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes when exiting a context. See *Lifetime and ContextManagers* for more info.

reset(*self*) → None
Reset the internal state of the scanner to assume that all keys are now released. Any key that is already pressed at the time of this call will therefore immediately cause a new key-pressed event to occur.

class keypad.**ShiftRegisterKeys**(***, *clock:* microcontroller.Pin, *data:* microcontroller.Pin, *latch:* microcontroller.Pin, *value_to_latch:* bool = *True*, *key_count:* int, *value_when_pressed:* bool, *interval:* float = *0.02*, *max_events:* int = *64*)
Manage a set of keys attached to an incoming shift register.

Create a *Keys* object that will scan keys attached to a parallel-in serial-out shift register like the 74HC165 or CD4021. Note that you may chain shift registers to load in as many values as you need.

Key number 0 is the first (or more properly, the zero-th) bit read. In the 74HC165, this bit is labeled `Q7`. Key number 1 will be the value of `Q6`, etc.

An *`EventQueue`* is created when this object is created and is available in the *`events`* attribute.

Parameters

- **`clock`** (`microcontroller.Pin`) – The shift register clock pin. The shift register should clock on a low-to-high transition.
- **`data`** (`microcontroller.Pin`) – the incoming shift register data pin
- **`latch`** (`microcontroller.Pin`) – Pin used to latch parallel data going into the shift register.
- **`value_to_latch`** (`bool`) – Pin state to latch data being read. `True` if the data is latched when `latch` goes high `False` if the data is latched when `latch goes low. The default is ``True`, which is how the 74HC165 operates. The CD4021 latch is the opposite. Once the data is latched, it will be shifted out by toggling the clock pin.
- **`key_count`** (`int`) – number of data lines to clock in
- **`value_when_pressed`** (`bool`) – `True` if the pin reads high when the key is pressed. `False` if the pin reads low (is grounded) when the key is pressed.
- **`interval`** (`float`) – Scan keys no more often than `interval` to allow for debouncing. `interval` is in float seconds. The default is 0.020 (20 msecs).
- **`max_events`** (`int`) – maximum size of *`events`* *`EventQueue`*: maximum number of key transition events that are saved. Must be >= 1. If a new event arrives when the queue is full, the oldest event is discarded.

`key_count :int`
The number of keys that are being scanned. (read-only)

`events :EventQueue`
The *`EventQueue`* associated with this *`Keys`* object. (read-only)

`deinit`(*self*) → None
Stop scanning and release the pins.

`__enter__`(*self*) → *Keys*
No-op used by Context Managers.

`__exit__`(*self*) → None
Automatically deinitializes when exiting a context. See *Lifetime and ContextManagers* for more info.

`reset`(*self*) → None
Reset the internal state of the scanner to assume that all keys are now released. Any key that is already pressed at the time of this call will therefore immediately cause a new key-pressed event to occur.

`math` – mathematical functions

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

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

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

math.**acos**(*x:* float) → *float*
Return the inverse cosine of x.

math.**asin**(*x:* float) → *float*
Return the inverse sine of x.

math.**atan**(*x:* float) → *float*
Return the inverse tangent of x.

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

math.**ceil**(*x:* float) → *int*
Return an integer, being x rounded towards positive infinity.

math.**copysign**(*x:* float, *y:* float) → *float*
Return x with the sign of y.

math.**cos**(*x:* float) → *float*
Return the cosine of x.

math.**degrees**(*x:* float) → *float*
Return radians x converted to degrees.

math.**exp**(*x:* float) → *float*
Return the exponential of x.

math.**fabs**(*x:* float) → *float*
Return the absolute value of x.

math.**floor**(*x:* float) → *int*
Return an integer, being x rounded towards negative infinity.

math.**fmod**(*x:* float, *y:* float) → *int*
Return the remainder of x/y.

math.**frexp**(*x:* float) → Tuple[*int*, *int*]
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:* float) → *bool*
Return `True` if x is finite.

math.**isinf**(*x:* float) → *bool*
Return `True` if x is infinite.

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

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

math.**modf**(*x:* float) → Tuple[*float*, *float*]
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:* float, *y:* float) → *float*
Returns x to the power of y.

math.**radians**(*x:* float) → *float*
Return degrees x converted to radians.

math.**sin**(*x:* float) → *float*
Return the sine of x.

math.**sqrt**(*x:* float) → *float*
Returns the square root of x.

math.**tan**(*x:* float) → *float*
Return the tangent of x.

math.**trunc**(*x:* float) → *int*
Return an integer, being x rounded towards 0.

math.**expm1**(*x:* float) → *float*
Return `exp(x) - 1`.

math.**log2**(*x:* float) → *float*
Return the base-2 logarithm of x.

math.**log10**(*x:* float) → *float*
Return the base-10 logarithm of x.

math.**cosh**(*x:* float) → *float*
Return the hyperbolic cosine of x.

math.**sinh**(*x:* float) → *float*
Return the hyperbolic sine of x.

math.**tanh**(*x:* float) → *float*
Return the hyperbolic tangent of x.

math.**acosh**(*x:* float) → *float*
Return the inverse hyperbolic cosine of x.

math.**asinh**(*x:* float) → *float*
Return the inverse hyperbolic sine of x.

math.**atanh**(*x:* float) → *float*
Return the inverse hyperbolic tangent of x.

math.**erf**(*x:* float) → *float*
Return the error function of x.

math.**erfc**(*x:* float) → *float*
Return the complementary error function of x.

math.**gamma**(*x:* float) → *float*
Return the gamma function of x.

math.**lgamma**(*x:* float) → *float*
Return the natural logarithm of the gamma function of x.

memorymonitor – Memory monitoring helpers

exception memorymonitor.**AllocationError**
Bases: *Exception*

Catchall exception for allocation related errors.

Initialize self. See help(type(self)) for accurate signature.

class memorymonitor.**AllocationAlarm**(*, *minimum_block_count:* int = *1*)

Throw an exception when an allocation of `minimum_block_count` or more blocks occurs while active.

Track allocations:

```
import memorymonitor

aa = memorymonitor.AllocationAlarm(minimum_block_count=2)
x = 2
# Should not allocate any blocks.
with aa:
    x = 5

# Should throw an exception when allocating storage for the 20 bytes.
with aa:
    x = bytearray(20)
```

ignore(*self*, *count:* int) → *AllocationAlarm*
Sets the number of applicable allocations to ignore before raising the exception. Automatically set back to zero at context exit.

Use it within a `with` block:

```
# Will not alarm because the bytearray allocation will be ignored.
with aa.ignore(2):
    x = bytearray(20)
```

__enter__(*self*) → *AllocationAlarm*
Enables the alarm.

__exit__(*self*) → None
Automatically disables the allocation alarm when exiting a context. See *Lifetime and ContextManagers* for more info.

class memorymonitor.**AllocationSize**
Tracks the number of allocations in power of two buckets.

It will have 16 16-bit buckets to track allocation counts. It is total allocations meaning frees are ignored. Reallocated memory is counted twice, at allocation and when reallocated with the larger size.

The buckets are measured in terms of blocks which is the finest granularity of the heap. This means bucket 0 will count all allocations less than or equal to the number of bytes per block, typically 16. Bucket 2 will be less than or equal to 4 blocks. See *bytes_per_block* to convert blocks to bytes.

Multiple AllocationSizes can be used to track different code boundaries.

Track allocations:

```
import memorymonitor

mm = memorymonitor.AllocationSize()
with mm:
  print("hello world" * 3)

for bucket, count in enumerate(mm):
    print("<", 2 ** bucket, count)
```

bytes_per_block :int
Number of bytes per block

__enter__(*self*) → *AllocationSize*
Clears counts and resumes tracking.

__exit__(*self*) → None
Automatically pauses allocation tracking when exiting a context. See *Lifetime and ContextManagers* for more info.

__len__(*self*) → *int*
Returns the number of allocation buckets.

This allows you to:

```
mm = memorymonitor.AllocationSize()
print(len(mm))
```

__getitem__(*self*, *index:* int) → Optional[*int*]
Returns the allocation count for the given bucket.

This allows you to:

```
mm = memorymonitor.AllocationSize()
print(mm[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.

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

microcontroller.**cpus :Processor**
CPU information and control, such as `cpus[0].temperature` and `cpus[1].frequency` (clock frequency) on chips with more than 1 cpu. The index selects which cpu. This object is an instance of *microcontroller.Processor*.

microcontroller.**delay_us**(*delay:* int) → None
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**() → None
Disable all interrupts. Be very careful, this can stall everything.

microcontroller.**enable_interrupts**() → None
Enable the interrupts that were enabled at the last disable.

microcontroller.**on_next_reset**(*run_mode:* RunMode) → None
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**() → None
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 :Optional[nvm.ByteArray]**
Available non-volatile memory. This object is the sole instance of *nvm.ByteArray* when available or None otherwise.

Type *nvm.ByteArray* or None

microcontroller.**watchdog :Optional[watchdog.WatchDogTimer]**
Available watchdog timer. This object is the sole instance of *watchdog.WatchDogTimer* when available or None otherwise.

class microcontroller.**Pin**
Identifies an IO pin on the microcontroller.

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.

class microcontroller.**Processor**
Microcontroller CPU information and control

Usage:

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

Note that on chips with more than one cpu (such as the RP2040)
microcontroller.cpu will return the value for CPU 0.
To get values from other CPUs use microcontroller.cpus indexed by
the number of the desired cpu. i.e.

print(microcontroller.cpus[0].temperature)
print(microcontroller.cpus[1].frequency)
```

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

frequency :int
The CPU operating frequency in Hertz. (read-only)

reset_reason :ResetReason
The reason the microcontroller started up from reset state.

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

Is None if the temperature is not available.

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

voltage :Optional[float]
The input voltage to the microcontroller, as a float. (read-only)

Is None if the voltage is not available.

class microcontroller.**ResetReason**
The reason the microntroller was last reset

POWER_ON :object
The microntroller was started from power off.

BROWNOUT :object
The microntroller was reset due to too low a voltage.

SOFTWARE :object
The microntroller was reset from software.

DEEP_SLEEP_ALARM :object
The microntroller was reset for deep sleep and restarted by an alarm.

RESET_PIN :object
The microntroller was reset by a signal on its reset pin. The pin might be connected to a reset button.

WATCHDOG :object
The microcontroller was reset by its watchdog timer.

UNKNOWN :object
The microntroller restarted for an unknown reason.

class microcontroller.**RunMode**
run state of the microcontroller

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

NORMAL :RunMode
Run CircuitPython as normal.

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

BOOTLOADER :RunMode
Run the bootloader.

msgpack – Pack object in msgpack format

The msgpack format is similar to json, except that the encoded data is binary. See https://msgpack.org for details. The module implements a subset of the cpython module msgpack-python.

Not implemented: 64-bit int, uint, float.

Example 1:

```
import msgpack
from io import BytesIO

b = BytesIO()
msgpack.pack({'list': [True, False, None, 1, 3.14], 'str': 'blah'}, b)
b.seek(0)
print(msgpack.unpack(b))
```

Example 2: handling objects:

```
from msgpack import pack, unpack, ExtType
from io import BytesIO

class MyClass:
    def __init__(self, val):
        self.value = val
    def __str__(self):
```

(continues on next page)

(continued from previous page)

```
        return str(self.value)

data = MyClass(b'my_value')

def encoder(obj):
    if isinstance(obj, MyClass):
        return ExtType(1, obj.value)
    return f"no encoder for {obj}"

def decoder(code, data):
    if code == 1:
        return MyClass(data)
    return f"no decoder for type {code}"

buffer = BytesIO()
pack(data, buffer, default=encoder)
buffer.seek(0)
decoded = unpack(buffer, ext_hook=decoder)
print(f"{data} -> {buffer.getvalue()} -> {decoded}")
```

msgpack.**pack**(*obj:* object, *buffer:* _typing.WriteableBuffer, *, *default:* *Union[Callable[[*object*], *None*], *None*] = *None*) → None

Ouput object to buffer in msgpack format.

Parameters

- **obj** (*object*) – Object to convert to msgpack format.
- **buffer** (*WriteableBuffer*) – buffer to write into
- **None]] default** (*Optional[Callable[[object],*) – function called for python objects that do not have a representation in msgpack format.

msgpack.**unpack**(*buffer:* _typing.ReadableBuffer, *, *ext_hook:* *Union[Callable[[*int, bytes*], *object*], *None*] = *None*, *use_list:* bool = *True*) → *object*

Unpack and return one object from buffer.

Parameters

- **buffer** (*ReadableBuffer*) – buffer to read from
- **bytes], object]] ext_hook** (*Optional[Callable[[int,*) – function called for objects in msgpack ext format.
- **use_list** (*Optional[bool]*) – return array as list or tuple (use_list=False).

Return object object read from buffer.

class msgpack.**ExtType**(*code:* int, *data:* bytes)

ExtType represents ext type in msgpack.

Constructor :param int code: type code in range 0~127. :param bytes data: representation.

code :int

The type code, in range 0~127.

data :bytes

Data.

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**() → Optional[BinaryIO]
Returns the current secondary terminal.

multiterminal.**set_secondary_terminal**(*stream: BinaryIO*) → None
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**() → None
Clears the secondary terminal.

multiterminal.**schedule_secondary_terminal_read**(*socket: schedule_secondary_terminal_read.socket*) → None
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: digitalio.DigitalInOut, buf: _typing.ReadableBuffer*) → None
Write buf out on the given DigitalInOut.

Parameters

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

network – Network Interface Management

Warning: This module is disabled in 6.x and will removed in 7.x. Please use networking libraries instead.

This module provides a registry of configured NICs. It is used by the 'socket' module to look up a suitable NIC when a socket is created.

network.**route**() → List[*object*]
Returns a list of all configured NICs.

nvm – Non-volatile memory

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

Note that this module can't be imported and used directly. The sole instance of *ByteArray* is available at *microcontroller.nvm*.

class nvm.**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"
```

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

__bool__(*self*) → *bool*

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

__getitem__(*self*, *index:* slice) → *bytearray*
__getitem__(*self*, *index:* int) → *int*
Returns the value at the given index.

__setitem__(*self*, *index:* slice, *value:* _typing.ReadableBuffer) → None
__setitem__(*self*, *index:* int, *value:* int) → None
Set the value at the given index.

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**() → *_Uname*
Returns a named tuple of operating specific and CircuitPython port specific information.

class os.**_Uname**
Bases: NamedTuple

The type of values that *uname()* returns

sysname :str

nodename :str

release :str

version :str

machine :str

os.**chdir**(*path:* str) → None
Change current directory.

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

os.**listdir**(*dir:* str) → *str*
With no argument, list the current directory. Otherwise list the given directory.

os.**mkdir**(*path:* str) → None
Create a new directory.

os.**remove**(*path:* str) → None
Remove a file.

os.**rmdir**(*path:* str) → None
Remove a directory.

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

os.**stat**(*path:* str) → Tuple[*int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*]
Get the status of a file or directory.

Note: On builds without long integers, the number of seconds for contemporary dates will not fit in a small integer. So the time fields return 946684800, which is the number of seconds corresponding to 1999-12-31.

---

os.**statvfs**(*path:* str) → Tuple[*int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*]
Get the status of a filesystem.

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 unprivileged users
- `f_files` – number of inodes
- `f_ffree` – number of free inodes
- `f_favail` – number of free inodes for unprivileged 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**() → None
Sync all filesystems.

os.**urandom**(*size:* int) → *str*
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** :**str**
Separator used to delineate path components such as folder and file names.

ps2io – Support for PS/2 protocol

The *ps2io* module contains classes to provide PS/2 communication.

Warning: This module is not available in some SAMD21 builds. See the *Module Support Matrix - Which Modules Are Available on Which Boards* 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.

class ps2io.**Ps2**(*data_pin:* microcontroller.Pin, *clock_pin:* microcontroller.Pin)
Communicate with a PS/2 keyboard or mouse

Ps2 implements the PS/2 keyboard/mouse serial protocol, used in legacy devices. It is similar to UART but there are only two lines (Data and Clock). PS/2 devices are 5V, so bidirectional level converters must be used to connect the I/O lines to pins of 3.3V boards.

Create a Ps2 object associated with the given pins.

Parameters

- **data_pin** (Pin) – Pin tied to data wire.
- **clock_pin** (Pin) – Pin tied to clock wire. This pin must support interrupts.

Read one byte from PS/2 keyboard and turn on Scroll Lock LED:

```
import ps2io
import board

kbd = ps2io.Ps2(board.D10, board.D11)

while len(kbd) == 0:
    pass

print(kbd.popleft())
print(kbd.sendcmd(0xed))
print(kbd.sendcmd(0x01))
```

deinit(*self*) → None
Deinitialises the Ps2 and releases any hardware resources for reuse.

__enter__(*self*) → *Ps2*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

popleft (*self*) → *int*
Removes and returns the oldest received byte. When buffer is empty, raises an IndexError exception.

sendcmd (*self*, *byte:* int) → *int*
Sends a command byte to PS/2. Returns the response byte, typically the general ack value (0xFA). Some commands return additional data which is available through *popleft()*.

Raises a RuntimeError in case of failure. The root cause can be found by calling *clear_errors()*. It is advisable to call *clear_errors()* before *sendcmd()* to flush any previous errors.

Parameters **byte** (int) – byte value of the command

clear_errors (*self*) → None
Returns and clears a bitmap with latest recorded communication errors.

Reception errors (arise asynchronously, as data is received):

0x01: start bit not 0

0x02: timeout

0x04: parity bit error

0x08: stop bit not 1

0x10: buffer overflow, newest data discarded

Transmission errors (can only arise in the course of sendcmd()):

0x100: clock pin didn't go to LO in time

0x200: clock pin didn't go to HI in time

0x400: data pin didn't ACK

0x800: clock pin didn't ACK

0x1000: device didn't respond to RTS

0x2000: device didn't send a response byte in time

__bool__ (*self*) → *bool*

__len__ (*self*) → *int*
Returns the number of received bytes in buffer, available to *popleft()*.

pulseio – Support for individual pulse based protocols

The *pulseio* module contains classes to provide access to basic pulse IO. Individual pulses are commonly used in infrared remotes and in DHT temperature sensors.

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.

class pulseio.**PulseIn** (*pin:* microcontroller.Pin, *maxlen:* int = *2*, ***, *idle_state:* bool = *False*)
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.

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)
```

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

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

deinit(*self*) → None
Deinitialises the PulseIn and releases any hardware resources for reuse.

__enter__(*self*) → *PulseIn*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

pause(*self*) → None
Pause pulse capture

resume(*self*, *trigger_duration:* int = *0*) → None
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(*self*) → None
Clears all captured pulses

popleft(*self*) → *int*
Removes and returns the oldest read pulse.

__bool__(*self*) → *bool*

__len__(*self*) → *int*
Returns the number of pulse durations currently stored.

This allows you to:

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

__getitem__(*self*, *index:* int) → Optional[*int*]
Returns the value at the given index or values in slice.

This allows you to:

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

class pulseio.**PulseOut**(*carrier:* pwmio.PWMOut)
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.

Create a PulseOut object associated with the given PWMout object.

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

Send a short series of pulses:

```
import array
import pulseio
import pwmio
import board

# 50% duty cycle at 38kHz.
pwm = pwmio.PWMOut(board.D13, frequency=38000, duty_cycle=32768)
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(*self*) → None
Deinitialises the PulseOut and releases any hardware resources for reuse.

__enter__(*self*) → *PulseOut*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

send(*self*, *pulses:* _typing.ReadableBuffer) → None
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

pwmio – Support for PWM based protocols

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

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 pwmio
import time
from board import *

pwm = pwmio.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.

class pwmio.**PWMOut**(*pin:* microcontroller.Pin, *, *duty_cycle:* int = *0*, *frequency:* int = *500*, *variable_frequency:* bool = *False*)
Output a Pulse Width Modulated signal on a given pin.

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 pwmio
import board
```

(continues on next page)

(continued from previous page)

```
pwm = pwmio.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 pwmio
import board

pwm = pwmio.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 pwmio
import board
import time

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

duty_cycle :int

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.

Depending on how PWM is implemented on a specific board, the internal representation for duty cycle might have less than 16 bits of resolution. Reading this property will return the value from the internal representation, so it may differ from the value set.

frequency :int

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

Depending on how PWM is implemented on a specific board, the internal value for the PWM's duty cycle may need to be recalculated when the frequency changes. In these cases, the duty cycle is automatically recalculated from the original duty cycle value. This should happen without any need to manually re-set the duty cycle.

deinit(*self*) → None

Deinitialises the PWMOut and releases any hardware resources for reuse.

__enter__(*self*) → *PWMOut*

No-op used by Context Managers.

__exit__(*self*) → None

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

rainbow

rainbow module.

Provides the *colorwheel()* function.

rainbow.**colorwheel**(*n:* float) → *int*
C implementation of the common colorwheel() function found in many examples. Returns the colorwheel RGB value as an integer value for n (usable in neopixel and dotstar).

random – pseudo-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.**_T**

random.**seed**(*seed:* int) → None
Sets the starting seed of the random number generation. Further calls to *random* will return deterministic results afterwards.

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

random.**randrange**(*stop: Tuple[*int, int, int*]*) → *int*
Returns a randomly selected integer from `range(start, stop, step)`.

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

random.**choice**(*seq: Sequence[*_T*]*) → *_T*
Returns a randomly selected element from the given sequence. Raises IndexError when the sequence is empty.

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

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

rgbmatrix – Low-level routines for bitbanged LED matrices

class rgbmatrix.**RGBMatrix**(*, *width:* int, *bit_depth:* int, *rgb_pins: Sequence[*digitalio.DigitalInOut*]*, *addr_pins: Sequence[*digitalio.DigitalInOut*]*, *clock_pin:* digitalio.DigitalInOut, *latch_pin:* digitalio.DigitalInOut, *output_enable_pin:* digitalio.DigitalInOut, *doublebuffer:* bool = *True*, *framebuffer: Optional[*_typing.WriteableBuffer*]* = *None*, *height:* int = *0*, *tile:* int = *1*, *serpentine:* bool = *True*)
Displays an in-memory framebuffer to a HUB75-style RGB LED matrix.

Create a RGBMatrix object with the given attributes. The height of the display is determined by the number of rgb and address pins and the number of tiles: `len(rgb_pins) // 3 * 2 ** len(address_pins) * abs(tile)`. With 6 RGB pins, 4 address lines, and a single matrix, the display will be 32 pixels tall. If the optional height parameter is specified and is not 0, it is checked against the calculated height.

Up to 30 RGB pins and 8 address pins are supported.

The RGB pins must be within a single "port" and performance and memory usage are best when they are all within "close by" bits of the port. The clock pin must also be on the same port as the RGB pins. See the documentation of the underlying protomatter C library for more information. Generally, Adafruit's interface boards are designed so that these requirements are met when matched with the intended microcontroller board. For instance, the Feather M4 Express works together with the RGB Matrix Feather.

The framebuffer is in "RGB565" format.

"RGB565" means that it is organized as a series of 16-bit numbers where the highest 5 bits are interpreted as red, the next 6 as green, and the final 5 as blue. The object can be any buffer, but *array.array* and `ulab.ndarray` objects are most often useful. To update the content, modify the framebuffer and call refresh.

If a framebuffer is not passed in, one is allocated and initialized to all black. In any case, the framebuffer can be retrieved by passing the RGBMatrix object to memoryview().

If doublebuffer is False, some memory is saved, but the display may flicker during updates.

A RGBMatrix is often used in conjunction with a *framebufferio.FramebufferDisplay*.

brightness :float
In the current implementation, 0.0 turns the display off entirely and any other value up to 1.0 turns the display on fully.

width :int
The width of the display, in pixels

height :int
The height of the display, in pixels

deinit(*self*) → None
Free the resources (pins, timers, etc.) associated with this rgbmatrix instance. After deinitialization, no further operations may be performed.

refresh(*self*) → None
Transmits the color data in the buffer to the pixels so that they are shown.

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.

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.

class rotaryio.**IncrementalEncoder**(*pin_a:* microcontroller.Pin, *pin_b:* microcontroller.Pin)
IncrementalEncoder determines the relative rotational position based on two series of pulses.

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
```

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

deinit(*self*) → None
Deinitializes the IncrementalEncoder and releases any hardware resources for reuse.

__enter__(*self*) → *IncrementalEncoder*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

rp2pio – Hardware interface to RP2 series' programmable IO (PIO) peripheral.

rp2pio.**pins_are_sequential**(*pins: List[*microcontroller.Pin*]*) → *bool*
Return True if the pins have sequential GPIO numbers, False otherwise

class rp2pio.**StateMachine**(*program:* _typing.ReadableBuffer, *frequency:* int, *, *init:* *Optional[*_typing.ReadableBuffer*]* = *None*, *first_out_pin:* *Optional[*microcontroller.Pin*]* = *None*, *out_pin_count:* int = *1*, *initial_out_pin_state:* int = *0*, *initial_out_pin_direction:* int = *4294967295*, *first_in_pin:* *Optional[*microcontroller.Pin*]* = *None*, *in_pin_count:* int = *1*, *pull_in_pin_up:* int = *0*, *pull_in_pin_down:* int = *0*, *first_set_pin:* *Optional[*microcontroller.Pin*]* = *None*, *set_pin_count:* int = *1*, *initial_set_pin_state:* int = *0*, *initial_set_pin_direction:* int = *31*, *first_sideset_pin:* *Optional[*microcontroller.Pin*]* = *None*, *sideset_pin_count:* int = *1*, *initial_sideset_pin_state:* int = *0*, *initial_sideset_pin_direction:* int = *31*, *exclusive_pin_use:* bool = *True*, *auto_pull:* bool = *False*, *pull_threshold:* int = *32*, *out_shift_right:* bool = *True*, *wait_for_txstall:* bool = *True*, *auto_push:* bool = *False*, *push_threshold:* int = *32*, *in_shift_right:* bool = *True*)
A single PIO StateMachine

The programmable I/O peripheral on the RP2 series of microcontrollers is unique. It is a collection of generic state machines that can be used for a variety of protocols. State machines may be independent or coordinated. Program memory and IRQs are shared between the state machines in a particular PIO instance. They are independent otherwise.

This class is designed to facilitate sharing of PIO resources. By default, it is assumed that the state machine is used on its own and can be placed in either PIO. State machines with the same program will be placed in the same PIO if possible.

Construct a StateMachine object on the given pins with the given program.

Parameters

- **program** (*ReadableBuffer*) – the program to run with the state machine
- **frequency** (int) – the target clock frequency of the state machine. Actual may be less.
- **init** (*ReadableBuffer*) – a program to run once at start up. This is run after program is started so instructions may be intermingled
- **first_out_pin** (Pin) – the first pin to use with the OUT instruction
- **out_pin_count** (int) – the count of consecutive pins to use with OUT starting at first_out_pin
- **initial_out_pin_state** (int) – the initial output value for out pins starting at first_out_pin
- **initial_out_pin_direction** (int) – the initial output direction for out pins starting at first_out_pin
- **first_in_pin** (Pin) – the first pin to use with the IN instruction
- **in_pin_count** (int) – the count of consecutive pins to use with IN starting at first_in_pin
- **pull_in_pin_up** (int) – a 1-bit in this mask sets pull up on the corresponding in pin
- **pull_in_pin_down** (int) – a 1-bit in this mask sets pull down on the corresponding in pin. Setting both pulls enables a "bus keep" function, i.e. a weak pull to whatever is current high/low state of GPIO.
- **first_set_pin** (Pin) – the first pin to use with the SET instruction
- **set_pin_count** (int) – the count of consecutive pins to use with SET starting at first_set_pin
- **initial_set_pin_state** (int) – the initial output value for set pins starting at first_set_pin
- **initial_set_pin_direction** (int) – the initial output direction for set pins starting at first_set_pin
- **first_sideset_pin** (Pin) – the first pin to use with a side set
- **sideset_pin_count** (int) – the count of consecutive pins to use with a side set starting at first_sideset_pin
- **initial_sideset_pin_state** (int) – the initial output value for sideset pins starting at first_sideset_pin
- **initial_sideset_pin_direction** (int) – the initial output direction for sideset pins starting at first_sideset_pin
- **exclusive_pin_use** (bool) – When True, do not share any pins with other state machines. Pins are never shared with other peripherals
- **auto_pull** (bool) – When True, automatically load data from the tx FIFO into the output shift register (OSR) when an OUT instruction shifts more than pull_threshold bits

- **pull_threshold** (int) – Number of bits to shift before loading a new value into the OSR from the tx FIFO
- **out_shift_right** (bool) – When True, data is shifted out the right side (LSB) of the OSR. It is shifted out the left (MSB) otherwise. NOTE! This impacts data alignment when the number of bytes is not a power of two (1, 2 or 4 bytes).
- **wait_for_txstall** (bool) – When True, writing data out will block until the TX FIFO and OSR are empty and an instruction is stalled waiting for more data. When False, data writes won't wait for the OSR to empty (only the TX FIFO) so make sure you give enough time before deiniting or stopping the state machine.
- **auto_push** (bool) – When True, automatically save data from input shift register (ISR) into the rx FIFO when an IN instruction shifts more than push_threshold bits
- **push_threshold** (int) – Number of bits to shift before saving the ISR value to the RX FIFO
- **in_shift_right** (bool) – When True, data is shifted into the right side (LSB) of the ISR. It is shifted into the left (MSB) otherwise. NOTE! This impacts data alignment when the number of bytes is not a power of two (1, 2 or 4 bytes).

frequency :int
The actual state machine frequency. This may not match the frequency requested due to internal limitations.

rxstall :bool
True when the state machine has stalled due to a full RX FIFO since the last *clear_rxfifo* call.

in_waiting :int
The number of words available to readinto

deinit (*self*) → None
Turn off the state machine and release its resources.

__enter__ (*self*) → *StateMachine*
No-op used by Context Managers. Provided by context manager helper.

__exit__ (*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

restart (*self*) → None
Resets this state machine, runs any init and enables the clock.

run (*self*, *instructions:* _typing.ReadableBuffer) → None
Runs all given instructions. They will likely be interleaved with in-memory instructions. Make sure this doesn't wait for input!

This can be used to output internal state to the RX FIFO and then read with *readinto*.

stop (*self*) → None
Stops the state machine clock. Use *restart* to enable it.

write (*self*, *buffer:* _typing.ReadableBuffer, *, *start:* int = *0*, *end: Optional[*int*] = None*) → None
Write the data contained in `buffer` to the state machine. If the buffer is empty, nothing happens.

Parameters

- **buffer** (ReadableBuffer) – 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. Defaults to `len(buffer)`

readinto(*self*, *buffer:* _typing.WriteableBuffer, *, *start:* int = *0*, *end: Optional[*int*] = None*) → None

Read into `buffer`. If the number of bytes to read is 0, nothing happens. The buffer include any data added to the fifo even if it was added before this was called.

Parameters

- **buffer** (`WriteableBuffer`) – 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. Defaults to `len(buffer)`

write_readinto(*self*, *buffer_out:* _typing.ReadableBuffer, *buffer_in:* _typing.WriteableBuffer, *, *out_start:* int = *0*, *out_end: Optional[*int*] = None*, *in_start:* int = *0*, *in_end: Optional[*int*] = None*) → None

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]` may be different. The function will return once both are filled. If buffer slice lengths are both 0, nothing happens.

Parameters

- **buffer_out** (`ReadableBuffer`) – Write out the data in this buffer
- **buffer_in** (`WriteableBuffer`) – 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. Defaults to `len(buffer_out)`
- **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. Defaults to `len(buffer_in)`

clear_rxfifo(*self*) → None

Clears any unread bytes in the rxfifo.

`rtc` – Real Time Clock

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

rtc.**set_time_source**(*rtc:* RTC) → None

Sets the RTC time source used by *`time.localtime()`*. The default is *`rtc.RTC`*, but it's useful to use this to override the time source for testing purposes. For example:

```
import rtc
import time

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

(continues on next page)

(continued from previous page)

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

class rtc.**RTC**

Real Time Clock

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

datetime :time.struct_time

The current date and time of the RTC as a *time.struct_time*.

This must be set to the current date and time whenever the board loses power:

```
import rtc
import time

r = rtc.RTC()
r.datetime = time.struct_time((2019, 5, 29, 15, 14, 15, 0, -1, -1))
```

Once set, the RTC will automatically update this value as time passes. You can read this property to get a snapshot of the current time:

```
current_time = r.datetime
print(current_time)
# struct_time(tm_year=2019, tm_month=5, ...)
```

calibration :int

The RTC calibration value as an *int*.

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 approximately 1 ppm:

```
import rtc
import time

r = rtc.RTC()
r.calibration = 1
```

samd – SAMD implementation settings

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 :bool

Is the clock enabled? (read-only)

parent :Union[Clock, None]

Clock parent. (read-only)

frequency :int

Clock frequency in Herz. (read-only)

calibration :int

Clock calibration. Not all clocks can be calibrated.

sdcardio – Interface to an SD card via the SPI bus

class sdcardio.**SDCard**(*bus:* busio.SPI, *cs:* microcontroller.Pin, *baudrate:* int = *8000000*)

SD Card Block Interface

Controls an SD card over SPI. This built-in module has higher read performance than the library adafruit_sdcard, but it is only compatible with *busio.SPI*, not *bitbangio.SPI*. Usually an SDCard object is used with `storage.VfsFat` to allow file I/O to an SD card.

Construct an SPI SD Card object with the given properties

Parameters

- **spi** (busio.SPI) – The SPI bus
- **cs** (microcontroller.Pin) – The chip select connected to the card
- **baudrate** (int) – The SPI data rate to use after card setup

Note that during detection and configuration, a hard-coded low baudrate is used. Data transfers use the specified baurate (rounded down to one that is supported by the microcontroller)

Important: If the same SPI bus is shared with other peripherals, it is important that the SD card be initialized before accessing any other peripheral on the bus. Failure to do so can prevent the SD card from being recognized until it is powered off or re-inserted.

Example usage:

```
import os

import board
import sdcardio
import storage

sd = sdcardio.SDCard(board.SPI(), board.SD_CS)
vfs = storage.VfsFat(sd)
storage.mount(vfs, '/sd')
os.listdir('/sd')
```

count(*self*) → *int*

Returns the total number of sectors

Due to technical limitations, this is a function and not a property.

Returns The number of 512-byte blocks, as a number

deinit(*self*) → None

Disable permanently.

Returns None

readblocks(*self*, *start_block:* int, *buf:* _typing.WriteableBuffer) → None

Read one or more blocks from the card

Parameters

- **start_block** (int) – The block to start reading from
- **buf** (WriteableBuffer) – The buffer to write into. Length must be multiple of 512.

Returns None

writeblocks(*self*, *start_block:* int, *buf:* _typing.ReadableBuffer) → None
Write one or more blocks to the card

Parameters

- **start_block** (int) – The block to start writing from
- **buf** (ReadableBuffer) – The buffer to read from. Length must be multiple of 512.

Returns None

sdioio – Interface to an SD card via the SDIO bus

class sdioio.**SDCard**(*clock:* microcontroller.Pin, *command:* microcontroller.Pin, *data:* Sequence[microcontroller.Pin], *frequency:* int)
SD Card Block Interface with SDIO

Controls an SD card over SDIO. SDIO is a parallel protocol designed for SD cards. It uses a clock pin, a command pin, and 1 or 4 data pins. It can be operated at a high frequency such as 25MHz. Usually an SDCard object is used with `storage.VfsFat` to allow file I/O to an SD card.

Construct an SDIO SD Card object with the given properties

Parameters

- **clock** (Pin) – the pin to use for the clock.
- **command** (Pin) – the pin to use for the command.
- **data** – A sequence of pins to use for data.
- **frequency** – The frequency of the bus in Hz

Example usage:

```
import os

import board
import sdioio
import storage

sd = sdioio.SDCard(
    clock=board.SDIO_CLOCK,
    command=board.SDIO_COMMAND,
    data=board.SDIO_DATA,
    frequency=25000000)
vfs = storage.VfsFat(sd)
storage.mount(vfs, '/sd')
os.listdir('/sd')
```

configure(*self*, *frequency:* int = *0*, *width:* int = *0*) → None
Configures the SDIO bus.

Parameters

- **frequency** (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.
- **width** (int) – the number of data lines to use. Must be 1 or 4 and must also not exceed the number of data lines at construction

Note: Leaving a value unspecified or 0 means the current setting is kept

count (*self*) → *int*
Returns the total number of sectors

Due to technical limitations, this is a function and not a property.

Returns The number of 512-byte blocks, as a number

readblocks (*self*, *start_block:* int, *buf:* _typing.WriteableBuffer) → None
Read one or more blocks from the card

Parameters

- **start_block** (int) – The block to start reading from
- **buf** (WriteableBuffer) – The buffer to write into. Length must be multiple of 512.

Returns None

writeblocks (*self*, *start_block:* int, *buf:* _typing.ReadableBuffer) → None
Write one or more blocks to the card

Parameters

- **start_block** (int) – The block to start writing from
- **buf** (ReadableBuffer) – The buffer to read from. Length must be multiple of 512.

Returns None

property frequency (*self*) → *int*
The actual SDIO bus frequency. This may not match the frequency requested due to internal limitations.

property width (*self*) → *int*
The actual SDIO bus width, in bits

deinit (*self*) → None
Disable permanently.

Returns None

__enter__ (*self*) → *SDCard*
No-op used by Context Managers. Provided by context manager helper.

__exit__ (*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

sharpdisplay – Support for Sharp Memory Display framebuffers

socket – TCP, UDP and RAW socket support

Warning: This module is disabled in 6.x and will removed in 7.x. Please use networking libraries instead. (Native networking will provide a socket compatible class.)

Create TCP, UDP and RAW sockets for communicating over the Internet.

class socket.**socket**(*family:* int = *AF_INET*, *type:* int = *SOCK_STREAM*, *proto:* int = *IPPROTO_TCP*)

Create a new socket

Parameters

- **family** (int) – AF_INET or AF_INET6
- **type** (int) – SOCK_STREAM, SOCK_DGRAM or SOCK_RAW
- **proto** (int) – IPPROTO_TCP, IPPROTO_UDP or IPPROTO_RAW (ignored)

AF_INET :int

AF_INET6 :int

SOCK_STREAM :int

SOCK_DGRAM :int

SOCK_RAW :int

IPPROTO_TCP :int

bind(*self*, *address: Tuple[*str, int*]*) → None

Bind a socket to an address

Parameters **address** (tuple(str, int)) – tuple of (remote_address, remote_port)

listen(*self*, *backlog:* int) → None

Set socket to listen for incoming connections

Parameters **backlog** (int) – length of backlog queue for waiting connetions

accept(*self*) → Tuple[*socket*, *str*]

Accept a connection on a listening socket of type SOCK_STREAM, creating a new socket of type SOCK_STREAM. Returns a tuple of (new_socket, remote_address)

connect(*self*, *address: Tuple[*str, int*]*) → None

Connect a socket to a remote address

Parameters **address** (tuple(str, int)) – tuple of (remote_address, remote_port)

send(*self*, *bytes:* _typing.ReadableBuffer) → *int*

Send some bytes to the connected remote address. Suits sockets of type SOCK_STREAM

Parameters **bytes** (ReadableBuffer) – some bytes to send

recv_into(*self*, *buffer:* _typing.WriteableBuffer, *bufsize:* int) → *int*

Reads some bytes from the connected remote address, writing into the provided buffer. If bufsize <= len(buffer) is given, a maximum of bufsize bytes will be read into the buffer. If no valid value is given for bufsize, the default is the length of the given buffer.

Suits sockets of type SOCK_STREAM Returns an int of number of bytes read.

Parameters

- **buffer** (WriteableBuffer) – buffer to receive into
- **bufsize** (int) – optionally, a maximum number of bytes to read.

recv(*self*, *bufsize:* int) → *bytes*

Reads some bytes from the connected remote address. Suits sockets of type SOCK_STREAM Returns a bytes() of length <= bufsize

Parameters **bufsize** (int) – maximum number of bytes to receive

sendto(*self*, *bytes:* _typing.ReadableBuffer, *address: Tuple[*str, int*]*) → *int*
Send some bytes to a specific address. Suits sockets of type SOCK_DGRAM

Parameters

- **bytes** (ReadableBuffer) – some bytes to send
- **address** (tuple *(str, int)*) – tuple of (remote_address, remote_port)

recvfrom(*self*, *bufsize:* int) → Tuple[*bytes*, Tuple[*str*, *int*]]
Reads some bytes from the connected remote address. Suits sockets of type SOCK_STREAM

Returns a tuple containing * a bytes() of length <= bufsize * a remote_address, which is a tuple of ip address and port number

Parameters **bufsize** (int) – maximum number of bytes to receive

setsockopt(*self*, *level:* int, *optname:* int, *value:* int) → None
Sets socket options

settimeout(*self*, *value:* int) → None
Set the timeout value for this socket.

Parameters **value** (int) – timeout in seconds. 0 means non-blocking. None means block indefinitely.

setblocking(*self*, *flag:* bool) → Optional[*int*]
Set the blocking behaviour of this socket.

Parameters **flag** (bool) – False means non-blocking, True means block indefinitely.

socket.**getaddrinfo**(*host:* str, *port:* int) → Tuple[*int*, *int*, *int*, *str*, *str*]
Gets the address information for a hostname and port

Returns the appropriate family, socket type, socket protocol and address information to call socket.socket() and socket.connect() with, as a tuple.

socketpool

The *socketpool* module provides sockets through a pool. The pools themselves act like CPython's *socket* module.

class socketpool.**Socket**
TCP, UDP and RAW socket. Cannot be created directly. Instead, call *SocketPool.socket()*.

Provides a subset of CPython's *socket.socket* API. It only implements the versions of recv that do not allocate bytes objects.

__enter__(*self*) → *Socket*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically closes the Socket when exiting a context. See *Lifetime and ContextManagers* for more info.

accept(*self*) → Tuple[*Socket*, Tuple[*str*, *int*]]
Accept a connection on a listening socket of type SOCK_STREAM, creating a new socket of type SOCK_STREAM. Returns a tuple of (new_socket, remote_address)

bind(*self*, *address: Tuple[*str, int*]*) → None
Bind a socket to an address

Parameters **address** (~tuple) – tuple of (remote_address, remote_port)

close (*self*) → None
Closes this Socket and makes its resources available to its SocketPool.

connect (*self*, *address: Tuple[*str, int*]*) → None
Connect a socket to a remote address

Parameters **address** (~tuple) – tuple of (remote_address, remote_port)

listen (*self*, *backlog:* int) → None
Set socket to listen for incoming connections

Parameters **backlog** (~int) – length of backlog queue for waiting connetions

recvfrom_into (*self*, *buffer:* _typing.WriteableBuffer) → Tuple[*int*, Tuple[*str*, *int*]]
Reads some bytes from a remote address.

Returns a tuple containing * the number of bytes received into the given buffer * a remote_address, which is a tuple of ip address and port number

Parameters **buffer** (object) – buffer to read into

recv_into (*self*, *buffer:* _typing.WriteableBuffer, *bufsize:* int) → *int*
Reads some bytes from the connected remote address, writing into the provided buffer. If bufsize <= len(buffer) is given, a maximum of bufsize bytes will be read into the buffer. If no valid value is given for bufsize, the default is the length of the given buffer.

Suits sockets of type SOCK_STREAM Returns an int of number of bytes read.

Parameters

- **buffer** (bytearray) – buffer to receive into
- **bufsize** (int) – optionally, a maximum number of bytes to read.

send (*self*, *bytes:* _typing.ReadableBuffer) → *int*
Send some bytes to the connected remote address. Suits sockets of type SOCK_STREAM

Parameters **bytes** (~bytes) – some bytes to send

sendto (*self*, *bytes:* _typing.ReadableBuffer, *address: Tuple[*str, int*]*) → *int*
Send some bytes to a specific address. Suits sockets of type SOCK_DGRAM

Parameters

- **bytes** (~bytes) – some bytes to send
- **address** (~tuple) – tuple of (remote_address, remote_port)

setblocking (*self*, *flag:* bool) → Optional[*int*]
Set the blocking behaviour of this socket.

Parameters **flag** (~bool) – False means non-blocking, True means block indefinitely.

settimeout (*self*, *value:* int) → None
Set the timeout value for this socket.

Parameters **value** (~int) – timeout in seconds. 0 means non-blocking. None means block indefinitely.

__hash__ (*self*) → *int*
Returns a hash for the Socket.

class socketpool.**SocketPool**
A pool of socket resources available for the given radio. Only one SocketPool can be created for each radio.

SocketPool should be used in place of CPython's socket which provides a pool of sockets provided by the underlying OS.

AF_INET :int

AF_INET6 :int

SOCK_STREAM :int

SOCK_DGRAM :int

SOCK_RAW :int

socket (*self*, *family:* int = *AF_INET*, *type:* int = *SOCK_STREAM*) → *Socket*
Create a new socket

Parameters

- **family** (~int) – AF_INET or AF_INET6
- **type** (~int) – SOCK_STREAM, SOCK_DGRAM or SOCK_RAW

socketpool.**getaddrinfo** (*host:* str, *port:* int, *family:* int = *0*, *type:* int = *0*, *proto:* int = *0*, *flags:* int = *0*) → Tuple[*int*, *int*, *int*, *str*, Tuple[*str*, *int*]]
Gets the address information for a hostname and port

Returns the appropriate family, socket type, socket protocol and address information to call socket.socket() and socket.connect() with, as a tuple.

ssl

The *ssl* module provides SSL contexts to wrap sockets in.

ssl.**create_default_context** () → *SSLContext*
Return the default SSLContext.

class ssl.**SSLContext**
Settings related to SSL that can be applied to a socket by wrapping it. This is useful to provide SSL certificates to specific connections rather than all of them.

ssl.**wrap_socket** (*sock:* socketpool.Socket, *, *server_side:* bool = *False*, *server_hostname: Optional[*str*]* = *None*) → *SSLSocket*
Wraps the socket into a socket-compatible class that handles SSL negotiation. The socket must be of type SOCK_STREAM.

class ssl.**SSLSocket**
Implements TLS security on a subset of *socketpool.Socket* functions. Cannot be created directly. Instead, call *wrap_socket* on an existing socket object.

Provides a subset of CPython's *ssl.SSLSocket* API. It only implements the versions of recv that do not allocate bytes objects.

__enter__ (*self*) → *SSLSocket*
No-op used by Context Managers.

__exit__ (*self*) → None
Automatically closes the Socket when exiting a context. See *Lifetime and ContextManagers* for more info.

accept (*self*) → Tuple[*SSLSocket*, Tuple[*str*, *int*]]
Accept a connection on a listening socket of type SOCK_STREAM, creating a new socket of type SOCK_STREAM. Returns a tuple of (new_socket, remote_address)

bind (*self*, *address: Tuple[*str, int*]*) → None
Bind a socket to an address

Parameters **address** (~tuple) – tuple of (remote_address, remote_port)

close (*self*) → None
Closes this Socket

connect (*self*, *address: Tuple[*str, int*]*) → None
Connect a socket to a remote address

Parameters **address** (~tuple) – tuple of (remote_address, remote_port)

listen (*self*, *backlog:* int) → None
Set socket to listen for incoming connections

Parameters **backlog** (~int) – length of backlog queue for waiting connetions

recv_into (*self*, *buffer:* _typing.WriteableBuffer, *bufsize:* int) → *int*
Reads some bytes from the connected remote address, writing into the provided buffer. If bufsize <= len(buffer) is given, a maximum of bufsize bytes will be read into the buffer. If no valid value is given for bufsize, the default is the length of the given buffer.

Suits sockets of type SOCK_STREAM Returns an int of number of bytes read.

Parameters

- **buffer** (bytearray) – buffer to receive into
- **bufsize** (int) – optionally, a maximum number of bytes to read.

send (*self*, *bytes:* _typing.ReadableBuffer) → *int*
Send some bytes to the connected remote address. Suits sockets of type SOCK_STREAM

Parameters **bytes** (~bytes) – some bytes to send

settimeout (*self*, *value:* int) → None
Set the timeout value for this socket.

Parameters **value** (~int) – timeout in seconds. 0 means non-blocking. None means block indefinitely.

setblocking (*self*, *flag:* bool) → Optional[*int*]
Set the blocking behaviour of this socket.

Parameters **flag** (~bool) – False means non-blocking, True means block indefinitely.

__hash__ (*self*) → *int*
Returns a hash for the Socket.

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:* VfsFat, *mount_path:* str, *, *readonly:* bool = *False*) → None
Mounts the given filesystem object at the given path.

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

Parameters **readonly** (bool) – True when the filesystem should be readonly to CircuitPython.

storage.**umount**(*mount: Union[*str, VfsFat*]*) → None
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:* str, *readonly:* bool = *False*, *, *disable_concurrent_write_protection:* bool = *False*) → None
Remounts the given path with new parameters.

Parameters

- **readonly** (bool) – True when the filesystem should be readonly to CircuitPython.
- **disable_concurrent_write_protection** (bool) – When True, the check that makes sure the underlying filesystem data is written by one computer is disabled. Disabling the protection allows CircuitPython and a host to write to the same filesystem with the risk that the filesystem will be corrupted.

storage.**getmount**(*mount_path:* str) → *VfsFat*
Retrieves the mount object associated with the mount path

storage.**erase_filesystem**() → None
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.

storage.**disable_usb_drive**() → None
Disable presenting `CIRCUITPY` as a USB mass storage device. By default, the device is enabled and `CIRCUITPY` is visible. Can be called in `boot.py`, before USB is connected.

storage.**enable_usb_drive**() → None
Enabled presenting `CIRCUITPY` as a USB mass storage device. By default, the device is enabled and `CIRCUITPY` is visible, so you do not normally need to call this function. Can be called in `boot.py`, before USB is connected.

If you enable too many devices at once, you will run out of USB endpoints. The number of available endpoints varies by microcontroller. CircuitPython will go into safe mode after running boot.py to inform you if not enough endpoints are available.

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

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

label :str
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(*self*) → None
Format the block device, deleting any data that may have been there

open(*self*, *path:* str, *mode:* str) → None
Like builtin `open()`

ilistdir(*self*, *path:* str) → Iterator[Union[Tuple[AnyStr, *int*, *int*, *int*], Tuple[AnyStr, *int*, *int*]]]
Return an iterator whose values describe files and folders within `path`

mkdir (*self*, *path:* str) → None
Like *os.mkdir*

rmdir (*self*, *path:* str) → None
Like *os.rmdir*

stat (*self*, *path:* str) → Tuple[*int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*]
Like *os.stat*

statvfs (*self*, *path:* int) → Tuple[*int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*, *int*]
Like *os.statvfs*

mount (*self*, *readonly:* bool, *mkfs:* VfsFat) → None
Don't call this directly, call *storage.mount*.

umount (*self*) → None
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*, *x*, *h*, *H*, *i*, *I*, *l*, *L*, *q*, *Q*, *s*, *P*, *f*, *d* (the latter 2 depending on the floating-point support).

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

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

`struct.`**pack_into** (*fmt:* str, *buffer:* _typing.WriteableBuffer, *offset:* int, **values: Any*) → None
Pack the values 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:* str, *data:* _typing.ReadableBuffer) → Tuple[Any, Ellipsis]
Unpack from the data according to the format string fmt. The return value is a tuple of the unpacked values. The buffer size must match the size required by the format.

`struct.`**unpack_from** (*fmt:* str, *data:* _typing.ReadableBuffer, *offset:* int = *0*) → Tuple[Any, Ellipsis]
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. The buffer size must be at least as big as the size required by the form.

`supervisor` – Supervisor settings

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

`supervisor.`**enable_autoreload**`()` → None
Enable autoreload based on USB file write activity.

`supervisor.`**disable_autoreload**`()` → None
Disable autoreload based on USB file write activity until *enable_autoreload* is called.

supervisor.**set_rgb_status_brightness**(*brightness:* int) → None
Set brightness of status neopixel from 0-255 *set_rgb_status_brightness* is called.

supervisor.**reload**() → None
Reload the main Python code and run it (equivalent to hitting Ctrl-D at the REPL).

supervisor.**set_next_stack_limit**(*size:* int) → None
Set the size of the stack for the next vm run. If its too large, the default will be used.

supervisor.**set_next_code_file**(*filename: Optional[str], *, reload_on_success:* bool = *False*, *reload_on_error:* bool = *False*, *sticky_on_success:* bool = *False*, *sticky_on_error:* bool = *False*, *sticky_on_reload:* bool = *False*) → None
Set what file to run on the next vm run.

When not `None`, the given `filename` is inserted at the front of the usual ['code.py', 'main.py'] search sequence.

The optional keyword arguments specify what happens after the specified file has run:

`sticky_on_...` determine whether the newly set filename and options stay in effect: If True, further runs will continue to run that file (unless it says otherwise by calling `set_next_code_filename()` itself). If False, the settings will only affect one run and revert to the standard code.py/main.py afterwards.

`reload_on_...` determine how to continue: If False, wait in the usual "Code done running. Waiting for reload. / Press any key to enter the REPL. Use CTRL-D to reload." state. If True, reload immediately as if CTRL-D was pressed.

`..._on_success` take effect when the program runs to completion or calls `sys.exit()`.

`..._on_error` take effect when the program exits with an exception, including the KeyboardInterrupt caused by CTRL-C.

`..._on_reload` take effect when the program is interrupted by files being written to the USB drive (auto-reload) or when it calls `supervisor.reload()`.

These settings are stored in RAM, not in persistent memory, and will therefore only affect soft reloads. Powering off or resetting the device will always revert to standard settings.

When called multiple times in the same run, only the last call takes effect, replacing any settings made by previous ones. This is the main use of passing `None` as a filename: to reset to the standard search sequence.

class supervisor.**RunReason**
The reason that CircuitPython started running.

STARTUP :object
CircuitPython started the microcontroller started up. See *microcontroller.Processor.reset_reason* for more detail on why the microcontroller was started.

AUTO_RELOAD :object
CircuitPython restarted due to an external write to the filesystem.

SUPERVISOR_RELOAD :object
CircuitPython restarted due to a call to *supervisor.reload()*.

REPL_RELOAD :object
CircuitPython started due to the user typing CTRL-D in the REPL.

class supervisor.**Runtime**
Current status of runtime objects.

Usage:

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

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

usb_connected :bool
Returns the USB enumeration status (read-only).

serial_connected :bool
Returns the USB serial communication status (read-only).

serial_bytes_available :int
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)

run_reason :RunReason
Returns why CircuitPython started running this particular time.

synthio – Support for MIDI synthesis

synthio.**from_file**(*file: BinaryIO*, *, *sample_rate:* int = *11025*) → *MidiTrack*
Create an AudioSample from an already opened MIDI file. Currently, only single-track MIDI (type 0) is supported.

Parameters

- **file** (*typing.BinaryIO*) – Already opened MIDI file
- **sample_rate** (int) – The desired playback sample rate; higher sample rate requires more memory

Playing a MIDI file from flash:

```
import audioio
import board
import synthio

data = open("single-track.midi", "rb")
midi = synthio.from_file(data)
a = audioio.AudioOut(board.A0)

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

class synthio.**MidiTrack**(*buffer:* _typing.ReadableBuffer, *tempo:* int, *, *sample_rate:* int = *11025*)
Simple square-wave MIDI synth

Create a MidiTrack from the given stream of MIDI events. Only "Note On" and "Note Off" events are supported; channel numbers and key velocities are ignored. Up to two notes may be on at the same time.

Parameters

- **buffer** (ReadableBuffer) – Stream of MIDI events, as stored in a MIDI file track chunk

- **tempo** (int) – Tempo of the streamed events, in MIDI ticks per second
- **sample_rate** (int) – The desired playback sample rate; higher sample rate requires more memory

Simple melody:

```
import audioio
import board
import synthio

dac = audioio.AudioOut(board.SPEAKER)
melody = synthio.MidiTrack(b"\0\x90H\0*\x80H\0\6\x90J\0*\x80J\0\6\x90L\0*\x80L\0\
↪6\x90J\0" +
                           b"*\x80J\0\6\x90H\0*\x80H\0\6\x90J\0*\x80J\0\6\x90L\0T\
↪x80L\0" +
                           b"\x0c\x90H\0T\x80H\0\x0c\x90H\0T\x80H\0", tempo=640)
dac.play(melody)
print("playing")
while dac.playing:
  pass
print("stopped")
```

sample_rate :Optional[int]
32 bit value that tells how quickly samples are played in Hertz (cycles per second).

deinit(*self*) → None
Deinitialises the MidiTrack and releases any hardware resources for reuse.

__enter__(*self*) → *MidiTrack*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

terminalio – Displays text in a TileGrid

The *terminalio* module contains classes to display a character stream on a display. The built in font is available as terminalio.FONT.

terminalio.**FONT :fontio.BuiltinFont**
The built in font

class terminalio.**Terminal**(*tilegrid:* displayio.TileGrid, *font:* fontio.BuiltinFont)
Display a character stream with a TileGrid

Terminal manages tile indices and cursor position based on VT100 commands. The font should be a *fontio.BuiltinFont* and the TileGrid's bitmap should match the font's bitmap.

write(*self*, *buf:* _typing.ReadableBuffer) → Optional[*int*]
Write the buffer of bytes to the bus.

Returns the number of bytes written

Return type *int* or None

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**() → *float*
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:* float) → None
Sleep for a given number of seconds.

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

class time.**struct_time**(*time_tuple: Sequence[*int*]*)
Structure used to capture a date and time. Can be constructed from a *struct_time*, *tuple*, *list*, or *namedtuple* with 9 elements.

Parameters **time_tuple** (*Sequence*) – Sequence of time info: (tm_year, tm_mon, tm_mday, tm_hour, tm_min, tm_sec, tm_wday, tm_yday, tm_isdst)

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

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

Returns the current time

Return type *int*

time.**monotonic_ns**() → *int*
Return the time of the monotonic clock, cannot go backward, in nanoseconds.

Returns the current time

Return type *int*

time.**localtime**(*secs:* int) → *struct_time*
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:* struct_time) → *int*

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.

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*.

class touchio.**TouchIn**(*pin:* microcontroller.Pin)

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!")
```

Use the TouchIn on the given pin.

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

value :bool

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

True when *raw_value* > *threshold*.

raw_value :int

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

threshold :Optional[int]

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:

```
import board
import touchio

touch = touchio.TouchIn(board.A1)
touch.threshold = 7300
```

deinit(*self*) → None
Deinitialises the TouchIn and releases any hardware resources for reuse.

__enter__(*self*) → *TouchIn*
No-op used by Context Managers.

__exit__(*self*) → None
Automatically deinitializes the hardware when exiting a context. See *Lifetime and ContextManagers* for more info.

uheap – Heap size analysis

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

ulab – Manipulate numeric data similar to numpy

ulab is a numpy-like module for micropython, meant to simplify and speed up common mathematical operations on arrays. The primary goal was to implement a small subset of numpy that might be useful in the context of a microcontroller. This means low-level data processing of linear (array) and two-dimensional (matrix) data.

ulab is adapted from micropython-ulab, and the original project's documentation can be found at https://micropython-ulab.readthedocs.io/en/latest/

ulab is modeled after numpy, and aims to be a compatible subset where possible. Numpy's documentation can be found at https://docs.scipy.org/doc/numpy/index.html

ulab.numpy – Compatibility layer for numpy

ulab.numpy.approx – Numerical approximation methods

ulab.numpy.approx.**interp**(*x:* ulab.ndarray, *xp:* ulab.ndarray, *fp:* ulab.ndarray, **, left: Optional[*float*] = None, right: Optional[*float*] = None*) → *ulab.ndarray*

Parameters

- **x** (ulab.ndarray) – The x-coordinates at which to evaluate the interpolated values.
- **xp** (ulab.ndarray) – The x-coordinates of the data points, must be increasing
- **fp** (ulab.ndarray) – The y-coordinates of the data points, same length as xp
- **left** – Value to return for `x < xp[0]`, default is `fp[0]`.
- **right** – Value to return for `x > xp[-1]`, default is `fp[-1]`.

Returns the one-dimensional piecewise linear interpolant to a function with given discrete data points (xp, fp), evaluated at x.

ulab.numpy.approx.**trapz**(*y:* ulab.ndarray, *x: Optional[*ulab.ndarray*] = None, dx:* float *= 1.0*) → *float*

Parameters

- **ulab.ndarray y** (*1D*) – the values of the dependent variable
- **ulab.ndarray x** (*1D*) – optional, the coordinates of the independent variable. Defaults to uniformly spaced values.
- **dx** (float) – the spacing between sample points, if x=None

Returns the integral of y(x) using the trapezoidal rule.

ulab.numpy.fft – Frequency-domain functions

ulab.numpy.fft.**fft**(*r:* ulab.ndarray, *c:* *Optional[*ulab.ndarray*]* = *None*) → Tuple[*ulab.ndarray*, *ulab.ndarray*]

Parameters

- **r** (ulab.ndarray) – A 1-dimension array of values whose size is a power of 2
- **c** (ulab.ndarray) – An optional 1-dimension array of values whose size is a power of 2, giving the complex part of the value

Return tuple (r, c) The real and complex parts of the FFT

Perform a Fast Fourier Transform from the time domain into the frequency domain

See also ~ulab.extras.spectrum, which computes the magnitude of the fft, rather than separately returning its real and imaginary parts.

ulab.numpy.fft.**ifft**(*r:* ulab.ndarray, *c:* *Optional[*ulab.ndarray*]* = *None*) → Tuple[*ulab.ndarray*, *ulab.ndarray*]

Parameters

- **r** (ulab.ndarray) – A 1-dimension array of values whose size is a power of 2
- **c** (ulab.ndarray) – An optional 1-dimension array of values whose size is a power of 2, giving the complex part of the value

Return tuple (r, c) The real and complex parts of the inverse FFT

Perform an Inverse Fast Fourier Transform from the frequeny domain into the time domain

ulab.numpy.linalg

ulab.numpy.linalg.**cholesky**(*A:* ulab.numpy.ndarray) → *ulab.numpy.ndarray*

Parameters **A** (ndarray) – a positive definite, symmetric square matrix

Return ~ulab.numpy.ndarray L a square root matrix in the lower triangular form

Raises ***ValueError*** – If the input does not fulfill the necessary conditions

The returned matrix satisfies the equation m=LL*

ulab.numpy.linalg.**det**(*m:* ulab.numpy.ndarray) → *float*

Param m, a square matrix

Return float The determinant of the matrix

Computes the eigenvalues and eigenvectors of a square matrix

ulab.numpy.linalg.**eig**(*m:* ulab.numpy.ndarray) → Tuple[*ulab.numpy.ndarray*, *ulab.numpy.ndarray*]

Parameters **m** – a square matrix

Return tuple (eigenvectors, eigenvalues)

Computes the eigenvalues and eigenvectors of a square matrix

ulab.numpy.linalg.**inv**(*m:* ulab.numpy.ndarray) → *ulab.numpy.ndarray*

Parameters **m** (ndarray) – a square matrix

Returns The inverse of the matrix, if it exists

Raises ***ValueError*** – if the matrix is not invertible

Computes the inverse of a square matrix

ulab.numpy.linalg.**norm**(*x:* ulab.numpy.ndarray) → *float*

Parameters **x** (ndarray) – a vector or a matrix

Computes the 2-norm of a vector or a matrix, i.e., `sqrt(sum(x*x))`, however, without the RAM overhead.

ulab.numpy.numerical – Numerical and Statistical functions

Most of these functions take an "axis" argument, which indicates whether to operate over the flattened array (None), or a particular axis (integer).

ulab.numpy.numerical.**argmax**(*array:* ulab._ArrayLike, *, *axis: Optional[*int*] = None*) → *int*
Return the index of the maximum element of the 1D array

ulab.numpy.numerical.**argmin**(*array:* ulab._ArrayLike, *, *axis: Optional[*int*] = None*) → *int*
Return the index of the minimum element of the 1D array

ulab.numpy.numerical.**argsort**(*array:* ulab.ndarray, *, *axis:* int = *- 1*) → *ulab.ndarray*
Returns an array which gives indices into the input array from least to greatest.

ulab.numpy.numerical.**cross**(*a:* ulab.ndarray, *b:* ulab.ndarray) → *ulab.ndarray*
Return the cross product of two vectors of length 3

ulab.numpy.numerical.**diff**(*array:* ulab.ndarray, *, *n:* int = *1*, *axis:* int = *- 1*) → *ulab.ndarray*
Return the numerical derivative of successive elements of the array, as an array. axis=None is not supported.

ulab.numpy.numerical.**flip**(*array:* ulab.ndarray, *, *axis: Optional[*int*] = None*) → *ulab.ndarray*
Returns a new array that reverses the order of the elements along the given axis, or along all axes if axis is None.

ulab.numpy.numerical.**max**(*array:* ulab._ArrayLike, *, *axis: Optional[*int*] = None*) → *float*
Return the maximum element of the 1D array

ulab.numpy.numerical.**mean**(*array:* ulab._ArrayLike, *, *axis: Optional[*int*] = None*) → *float*
Return the mean element of the 1D array, as a number if axis is None, otherwise as an array.

ulab.numpy.numerical.**median**(*array:* ulab.ndarray, *, *axis:* int = *- 1*) → *ulab.ndarray*
Find the median value in an array along the given axis, or along all axes if axis is None.

ulab.numpy.numerical.**min**(*array:* ulab._ArrayLike, *, *axis: Optional[*int*] = None*) → *float*
Return the minimum element of the 1D array

ulab.numpy.numerical.**roll**(*array:* ulab.ndarray, *distance:* int, *, *axis: Optional[*int*] = None*) → None
Shift the content of a vector by the positions given as the second argument. If the `axis` keyword is supplied, the shift is applied to the given axis. The array is modified in place.

ulab.numpy.numerical.**sort**(*array:* ulab.ndarray, *, *axis:* int = *- 1*) → *ulab.ndarray*
Sort the array along the given axis, or along all axes if axis is None. The array is modified in place.

ulab.numpy.numerical.**std**(*array:* ulab._ArrayLike, *, *axis: Optional[*int*] = None*, *ddof:* int = *0*) → *float*
Return the standard deviation of the array, as a number if axis is None, otherwise as an array.

ulab.numpy.numerical.**sum**(*array:* ulab._ArrayLike, *, *axis: Optional[*int*] = None*) → Union[*float*, *int*, *ulab.ndarray*]
Return the sum of the array, as a number if axis is None, otherwise as an array.

ulab.numpy.stats

ulab.numpy.stats.**trace**(*m:* ulab.numpy.ndarray) → *float*

Parameters **m** – a square matrix

Compute the trace of the matrix, the sum of its diagonal elements.

ulab.numpy.transform

ulab.numpy.transform.**dot**(*m1:* ulab.numpy.ndarray, *m2:* ulab.numpy.ndarray) → Union[*ulab.numpy.ndarray*, *float*]

Parameters

- **m1** (ndarray) – a matrix, or a vector
- **m2** (ndarray) – a matrix, or a vector

Computes the product of two matrices, or two vectors. In the letter case, the inner product is returned.

ulab.numpy.vector – Element-by-element functions

These functions can operate on numbers, 1-D iterables, and arrays of 1 to 4 dimensions by applying the function to every element in the array. This is typically much more efficient than expressing the same operation as a Python loop.

ulab.numpy.vector.**acos**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the inverse cosine function

ulab.numpy.vector.**acosh**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the inverse hyperbolic cosine function

ulab.numpy.vector.**asin**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the inverse sine function

ulab.numpy.vector.**asinh**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the inverse hyperbolic sine function

ulab.numpy.vector.**around**(*a:* ulab._ArrayLike, *, *decimals:* int = *0*) → *ulab.ndarray*
Returns a new float array in which each element is rounded to decimals places.

ulab.numpy.vector.**atan**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the inverse tangent function; the return values are in the range [-pi/2,pi/2].

ulab.numpy.vector.**arctan2**(*ya:* ulab._ArrayLike, *xa:* ulab._ArrayLike) → *ulab.ndarray*
Computes the inverse tangent function of y/x; the return values are in the range [-pi, pi].

ulab.numpy.vector.**atanh**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the inverse hyperbolic tangent function

ulab.numpy.vector.**ceil**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Rounds numbers up to the next whole number

ulab.numpy.vector.**cos**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the cosine function

ulab.numpy.vector.**cosh**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the hyperbolic cosine function

ulab.numpy.vector.**degrees**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Converts angles from radians to degrees

ulab.numpy.vector.**erf**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the error function, which has applications in statistics

ulab.numpy.vector.**erfc**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the complementary error function, which has applications in statistics

ulab.numpy.vector.**exp**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the exponent function.

ulab.numpy.vector.**expm1**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes e^x-1. In certain applications, using this function preserves numeric accuracy better than the *exp* function.

ulab.numpy.vector.**floor**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Rounds numbers up to the next whole number

ulab.numpy.vector.**gamma**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the gamma function

ulab.numpy.vector.**lgamma**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the natural log of the gamma function

ulab.numpy.vector.**log**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the natural log

ulab.numpy.vector.**log10**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the log base 10

ulab.numpy.vector.**log2**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the log base 2

ulab.numpy.vector.**radians**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Converts angles from degrees to radians

ulab.numpy.vector.**sin**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the sine function

ulab.numpy.vector.**sinh**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the hyperbolic sine

ulab.numpy.vector.**sqrt**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the square root

ulab.numpy.vector.**tan**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the tangent

ulab.numpy.vector.**tanh**(*a:* ulab._ArrayLike) → *ulab.ndarray*
Computes the hyperbolic tangent

ulab.numpy.vector.**vectorize**(*f: Union[Callable[[*int*], float], Callable[[*float*], float]]*, *, *otypes: Optional[*ulab._DType*] = None*) → Callable[[*ulab._ArrayLike*], *ulab.ndarray*]

Parameters

- **f** (*callable*) – The function to wrap

- **otypes** – List of array types that may be returned by the function. None is interpreted to mean the return value is float.

Wrap a Python function `f` so that it can be applied to arrays. The callable must return only values of the types specified by `otypes`, or the result is undefined.

class ulab.numpy.**ndarray**

ulab.scipy – Compatibility layer for scipy

ulab.scipy.linalg

ulab.scipy.linalg.**solve_triangular**(*A:* ulab.numpy.ndarray, *b:* ulab.numpy.ndarray, *lower:* bool) → *ulab.numpy.ndarray*

Parameters

- **A** (ndarray) – a matrix
- **b** (ndarray) – a vector
- **lower** (~bool) – if true, use only data contained in lower triangle of A, else use upper triangle of A

Returns solution to the system A x = b. Shape of return matches b

Raises

- ***TypeError*** – if A and b are not of type ndarray and are not dense
- ***ValueError*** – if A is a singular matrix

Solve the equation A x = b for x, assuming A is a triangular matrix

ulab.scipy.linalg.**cho_solve**(*L:* ulab.numpy.ndarray, *b:* ulab.numpy.ndarray) → *ulab.numpy.ndarray*

Parameters

- **L** (ndarray) – the lower triangular, Cholesky factorization of A
- **b** (ndarray) – right-hand-side vector b

Returns solution to the system A x = b. Shape of return matches b

Raises ***TypeError*** – if L and b are not of type ndarray and are not dense

Solve the linear equations A x = b, given the Cholesky factorization of A as input

ulab.scipy.optimize

ulab.scipy.optimize.**bisect**(*fun: Callable[[*float*]*, float*]*, *a:* float, *b:* float, **, *xtol:* float = *2.4e-07*, *maxiter:* int = *100*) → *float*

Parameters

- **f** (callable) – The function to bisect
- **a** (float) – The left side of the interval
- **b** (float) – The right side of the interval
- **xtol** (float) – The tolerance value

- **maxiter** (float) – The maximum number of iterations to perform

Find a solution (zero) of the function `f(x)` on the interval (`a..``b``) using the bisection method. The result is accurate to within `xtol` unless more than `maxiter` steps are required.

ulab.scipy.optimize.**fmin**(*fun: Callable[[*float*], *float*], *x0:* float, *, *xatol:* float = *2.4e-07*, *fatol:* float = *2.4e-07*, *maxiter:* int = *200*) → *float*

Parameters

- **f** (callable) – The function to bisect
- **x0** (float) – The initial x value
- **xatol** (float) – The absolute tolerance value
- **fatol** (float) – The relative tolerance value

Find a minimum of the function `f(x)` using the downhill simplex method. The located `x` is within `fxtol` of the actual minimum, and `f(x)` is within `fatol` of the actual minimum unless more than `maxiter` steps are requried.

ulab.scipy.optimize.**newton**(*fun: Callable[[*float*], *float*], *x0:* float, *, *xtol:* float = *2.4e-07*, *rtol:* float = *0.0*, *maxiter:* int = *50*) → *float*

Parameters

- **f** (callable) – The function to bisect
- **x0** (float) – The initial x value
- **xtol** (float) – The absolute tolerance value
- **rtol** (float) – The relative tolerance value
- **maxiter** (float) – The maximum number of iterations to perform

Find a solution (zero) of the function `f(x)` using Newton's Method. The result is accurate to within `xtol * rtol * |f(x)|` unless more than `maxiter` steps are requried.

ulab.scipy.signal

ulab.scipy.signal.**spectrogram**(*r:* ulab.ndarray) → *ulab.ndarray*

Parameters **r** (ulab.ndarray) – A 1-dimension array of values whose size is a power of 2

Computes the spectrum of the input signal. This is the absolute value of the (complex-valued) fft of the signal. This function is similar to scipy's `scipy.signal.spectrogram`.

ulab.user – This module should hold arbitrary user-defined functions.

ulab.**_DType**
ulab.int8, *ulab.uint8*, *ulab.int16*, *ulab.uint16*, *ulab.float* or *ulab.bool*

ulab.**_float**
Type alias of the bulitin float

ulab.**_bool**
Type alias of the bulitin bool

ulab.**_Index**

class ulab.**ndarray**(*values: Union[*ndarray*, Iterable[Union[*_float*, *_bool*, Iterable[Any]]]]*, ***, *dtype:* _DType = *ulab.float*)

1- and 2- dimensional ndarray

Parameters

- **values** (*sequence*) – Sequence giving the initial content of the ndarray.
- **dtype** (_DType) – The type of ndarray values, *ulab.int8*, *ulab.uint8*, *ulab.int16*, *ulab.uint16*, *ulab.float* or *ulab.bool*

The `values` sequence can either be another ~ulab.ndarray, sequence of numbers (in which case a 1-dimensional ndarray is created), or a sequence where each subsequence has the same length (in which case a 2-dimensional ndarray is created).

Passing a *ulab.ndarray* and a different dtype can be used to convert an ndarray from one dtype to another.

In many cases, it is more convenient to create an ndarray from a function like *zeros* or *linspace*.

ulab.ndarray implements the buffer protocol, so it can be used in many places an *array.array* can be used.

shape :Tuple[int, Ellipsis]
The size of the array, a tuple of length 1 or 2

size :int
The number of elements in the array

itemsize :int
The size of a single item in the array

strides :Tuple[int, Ellipsis]
Tuple of bytes to step in each dimension, a tuple of length 1 or 2

copy(*self*) → *ndarray*
Return a copy of the array

dtype(*self*) → *_DType*
Returns the dtype of the array

flatten(*self*, ***, *order:* str = *'C'*) → *ndarray*

Parameters **order** – Whether to flatten by rows ('C') or columns ('F')

Returns a new *ulab.ndarray* object which is always 1 dimensional. If order is 'C' (the default", then the data is ordered in rows; If it is 'F', then the data is ordered in columns. "C" and "F" refer to the typical storage organization of the C and Fortran languages.

reshape(*self*, *shape: Tuple[*int, *Ellipsis]*) → *ndarray*
Returns an ndarray containing the same data with a new shape.

sort(*self*, ***, *axis: Optional[*int*] = 1*) → None

Parameters **axis** – Whether to sort elements within rows (0), columns (1), or elements (None)

tobytes(*self*) → *bytearray*
Return the raw data bytes in the ndarray

transpose(*self*) → *ndarray*
Swap the rows and columns of a 2-dimensional ndarray

__add__(*self*, *other: Union[*ndarray, _float*]*) → *ndarray*
Adds corresponding elements of the two ndarrays, or adds a number to all elements of the ndarray. If both arguments are ndarrays, their sizes must match.

__radd__(*self*, *other:* _float) → *ndarray*

__sub__ (*self*, *other: Union[*ndarray, _float*]*) → *ndarray*
Subtracts corresponding elements of the two ndarrays, or subtracts a number from all elements of the ndarray. If both arguments are ndarrays, their sizes must match.

__rsub__ (*self*, *other:* _float) → *ndarray*

__mul__ (*self*, *other: Union[*ndarray, _float*]*) → *ndarray*
Multiplies corresponding elements of the two ndarrays, or multiplies all elements of the ndarray by a number. If both arguments are ndarrays, their sizes must match.

__rmul__ (*self*, *other:* _float) → *ndarray*

__div__ (*self*, *other: Union[*ndarray, _float*]*) → *ndarray*
Multiplies corresponding elements of the two ndarrays, or divides all elements of the ndarray by a number. If both arguments are ndarrays, their sizes must match.

__rdiv__ (*self*, *other:* _float) → *ndarray*

__pow__ (*self*, *other: Union[*ndarray, _float*]*) → *ndarray*
Computes the power (x**y) of corresponding elements of the the two ndarrays, or one number and one ndarray. If both arguments are ndarrays, their sizes must match.

__rpow__ (*self*, *other:* _float) → *ndarray*

__inv__ (*self*) → *ndarray*

__neg__ (*self*) → *ndarray*

__pos__ (*self*) → *ndarray*

__abs__ (*self*) → *ndarray*

__len__ (*self*) → *int*

__lt__ (*self*, *other: Union[*ndarray, _float*]*) → *ndarray*
Return self<value.

__le__ (*self*, *other: Union[*ndarray, _float*]*) → *ndarray*
Return self<=value.

__gt__ (*self*, *other: Union[*ndarray, _float*]*) → *ndarray*
Return self>value.

__ge__ (*self*, *other: Union[*ndarray, _float*]*) → *ndarray*
Return self>=value.

__iter__ (*self*) → Union[Iterator[*ndarray*], Iterator[*_float*]]

__getitem__ (*self*, *index:* _Index) → Union[*ndarray*, *_float*]
Retrieve an element of the ndarray.

__setitem__ (*self*, *index:* _Index, *value: Union[*ndarray, _float*]*) → None
Set an element of the ndarray.

ulab.**_ArrayLike**
ulab.ndarray, `List[float]`, `Tuple[float]` or *range*

ulab.**int8 :_DType**
Type code for signed integers in the range -128 .. 127 inclusive, like the 'b' typecode of *array.array*

ulab.**int16 :_DType**
Type code for signed integers in the range -32768 .. 32767 inclusive, like the 'h' typecode of *array.array*

ulab.**float :_DType**
Type code for floating point values, like the 'f' typecode of *array.array*

ulab.**uint8** :_DType
Type code for unsigned integers in the range 0 .. 255 inclusive, like the 'H' typecode of *array.array*

ulab.**uint16** :_DType
Type code for unsigned integers in the range 0 .. 65535 inclusive, like the 'h' typecode of *array.array*

ulab.**bool** :_DType
Type code for boolean values

ulab.**get_printoptions**() → Dict[*str*, *int*]
Get printing options

ulab.**set_printoptions**(*threshold: Optional[*int*] = None*, *edgeitems: Optional[*int*] = None*) → None
Set printing options

ulab.**ndinfo**(*array:* ndarray) → None

ulab.**array**(*values: Union[*ndarray*, Iterable[Union[*_float*, *_bool*, Iterable[Any]]]]*, *, *dtype:* _DType = *ulab.float*) → *ndarray*
alternate constructor function for *ulab.ndarray*. Mirrors numpy.array

ulab.**arange**(*stop:* _float, *step:* _float = *1*, *, *dtype:* _DType = *ulab.float*) → *ndarray*
ulab.**arange**(*start:* _float, *stop:* _float, *step:* _float = *1*, *, *dtype:* _DType = *ulab.float*) → *ndarray*
Return a new 1-D array with elements ranging from `start` to `stop`, with step size `step`.

ulab.**concatenate**(*arrays: Tuple[*ndarray*]*, *, *axis:* int = *0*) → *ndarray*
Join a sequence of arrays along an existing axis.

ulab.**diag**(*a:* ndarray, *, *k:* int = *0*) → *ndarray*
Return specified diagonals.

ulab.**eye**(*size:* int, *, *M: Optional[*int*] = None*, *k:* int = *0*, *dtype:* _DType = *ulab.float*) → *ndarray*
Return a new square array of size, with the diagonal elements set to 1 and the other elements set to 0.

ulab.**full**(*shape: Union[*int*, Tuple[*int*, Ellipsis]]*, *fill_value: Union[*_float*, *_bool*]*, *, *dtype:* _DType = *ulab.float*) → *ndarray*
Return a new array of the given shape with all elements set to 0.

ulab.**linspace**(*start:* _float, *stop:* _float, *, *dtype:* _DType = *ulab.float*, *num:* int = *50*, *endpoint:* _bool = *True*, *retstep:* _bool = *False*) → *ndarray*
Return a new 1-D array with `num` elements ranging from `start` to `stop` linearly.

ulab.**logspace**(*start:* _float, *stop:* _float, *, *dtype:* _DType = *ulab.float*, *num:* int = *50*, *endpoint:* _bool = *True*, *base:* _float = *10.0*) → *ndarray*
Return a new 1-D array with `num` evenly spaced elements on a log scale. The sequence starts at `base ** start`, and ends with `base ** stop`.

ulab.**ones**(*shape: Union[*int*, Tuple[*int*, Ellipsis]]*, *, *dtype:* _DType = *ulab.float*) → *ndarray*
Return a new array of the given shape with all elements set to 1.

ulab.**zeros**(*shape: Union[*int*, Tuple[*int*, Ellipsis]]*, *, *dtype:* _DType = *ulab.float*) → *ndarray*
Return a new array of the given shape with all elements set to 0.

usb_cdc – USB CDC Serial streams

The *usb_cdc* module allows access to USB CDC (serial) communications.

On Windows, each *Serial* is visible as a separate COM port. The ports will often be assigned consecutively, *console* first, but this is not always true.

On Linux, the ports are typically `/dev/ttyACM0` and `/dev/ttyACM1`. The *console* port will usually be first.

On MacOS, the ports are typically `/dev/cu.usbmodem<something>`. The something varies based on the USB bus and port used. The *console* port will usually be first.

usb_cdc.**console** :Optional[Serial]

The *console* *Serial* object is used for the REPL, and for *sys.stdin* and *sys.stdout*. *console* is `None` if disabled.

However, note that *sys.stdin* and *sys.stdout* are text-based streams, and the *console* object is a binary stream. You do not normally need to write to *console* unless you want to write binary data.

usb_cdc.**data** :Optional[Serial]
A *Serial* object that can be used to send and receive binary data to and from the host. Note that *data* is *disabled* by default. `data` is `None` if disabled.

usb_cdc.**disable**() → None
Do not present any USB CDC device to the host. Can be called in `boot.py`, before USB is connected. Equivalent to `usb_cdc.enable(console=False, data=False)`.

usb_cdc.**enable**(*, *console:* bool = *True*, *data:* bool = *False*) → None
Enable or disable each CDC device. Can be called in `boot.py`, before USB is connected.

Parameters

- **bool** (`data`) – Enable or disable the *console* USB serial device. True to enable; False to disable. Enabled by default.
- **bool** – Enable or disable the *data* USB serial device. True to enable; False to disable. *Disabled* by default.

If you enable too many devices at once, you will run out of USB endpoints. The number of available endpoints varies by microcontroller. CircuitPython will go into safe mode after running boot.py to inform you if not enough endpoints are available.

class usb_cdc.**Serial**
Receives cdc commands over USB

You cannot create an instance of *usb_cdc.Serial*. The available instances are in the `usb_cdc.serials` tuple.

connected :bool
True if this Serial is connected to a host. (read-only)

Note: The host is considered to be connected if it is asserting DTR (Data Terminal Ready). Most terminal programs and `pyserial` assert DTR when opening a serial connection. However, the C# `SerialPort` API does not. You must set `SerialPort.DtrEnable`.

in_waiting :int
Returns the number of bytes waiting to be read on the USB serial input. (read-only)

out_waiting :int
Returns the number of bytes waiting to be written on the USB serial output. (read-only)

timeout :Optional[float]
The initial value of *timeout* is None. If None, wait indefinitely to satisfy the conditions of a read operation. If 0, do not wait. If > 0, wait only timeout seconds.

write_timeout :Optional[float]
The initial value of *write_timeout* is None. If None, wait indefinitely to finish writing all the bytes passed to write().If 0, do not wait. If > 0, wait only write_timeout seconds.

read(*self*, *size:* int = *1*) → *bytes*
Read at most size bytes. If size exceeds the internal buffer size only the bytes in the buffer will be read. If *timeout* is > 0 or None, and fewer than size bytes are available, keep waiting until the timeout expires or size bytes are available.

Returns Data read

Return type *bytes*

readinto(*self*, *buf:* _typing.WriteableBuffer) → *int*
Read bytes into the buf. If nbytes is specified then read at most that many bytes, subject to *timeout*. Otherwise, read at most len(buf) bytes.

Returns number of bytes read and stored into buf

Return type *bytes*

readline(*self*, *size:* int = - *1*) → Optional[*bytes*]
Read a line ending in a newline character ("\n"), including the newline. Return everything readable if no newline is found and timeout is 0. Return None in case of error.

This is a binary stream: the newline character "\n" cannot be changed. If the host computer transmits "\r" it will also be included as part of the line.

Parameters **size** (int) – maximum number of characters to read. -1 means as many as possible.

Returns the line read

Return type *bytes* or None

readlines(*self*) → List[Optional[*bytes*]]
Read multiple lines as a list, using *readline()*.

> **Warning:** If timeout is None, *readlines()* will never return, because there is no way to indicate end of stream.

Returns a list of the line read

Return type *list*

write(*self*, *buf:* _typing.ReadableBuffer) → *int*
Write as many bytes as possible from the buffer of bytes.

Returns the number of bytes written

Return type *int*

flush(*self*) → None
Force out any unwritten bytes, waiting until they are written.

reset_input_buffer(*self*) → None
Clears any unread bytes.

reset_output_buffer(*self*) → None
Clears any unwritten bytes.

usb_hid – USB Human Interface Device

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

usb_hid.**devices :Tuple[Device, Ellipsis]**
Tuple of all active HID device interfaces. The default set of devices is `Device.KEYBOARD, Device.MOUSE, Device.CONSUMER_CONTROL`, On boards where *usb_hid* is disabled by default, *devices* is an empty tuple.

usb_hid.**disable**() → None
Do not present any USB HID devices to the host computer. Can be called in `boot.py`, before USB is connected. The HID composite device is normally enabled by default, but on some boards with limited endpoints, including STM32F4, it is disabled by default. You must turn off another USB device such as *usb_cdc* or *storage* to free up endpoints for use by *usb_hid*.

usb_hid.**enable**(*devices: Optional[Sequence[*Device*]]*) → None
Specify which USB HID devices that will be available. Can be called in `boot.py`, before USB is connected.

Parameters **devices** (*Sequence*) – *Device* objects. If *devices* is empty, HID is disabled. The order of the `Devices` may matter to the host. For instance, for MacOS, put the mouse device before any Gamepad or Digitizer HID device or else it will not work.

If you enable too many devices at once, you will run out of USB endpoints. The number of available endpoints varies by microcontroller. CircuitPython will go into safe mode after running boot.py to inform you if not enough endpoints are available.

class usb_hid.**Device**(*, *descriptor:* _typing.ReadableBuffer, *usage_page:* int, *usage:* int, *in_report_length:* int, *out_report_length:* int = *0*, *report_id_index: Optional[*int*]*)
HID Device specification

Create a description of a USB HID device. The actual device is created when you pass a *Device* to *usb_hid.enable()*.

Parameters

- **report_descriptor** (*ReadableBuffer*) – The USB HID Report descriptor bytes. The descriptor is not not verified for correctness; it is up to you to make sure it is not malformed.
- **usage_page** (int) – The Usage Page value from the descriptor. Must match what is in the descriptor.
- **usage** (int) – The Usage value from the descriptor. Must match what is in the descriptor.
- **in_report_length** (int) – Size in bytes of the HID report sent to the host. "In" is with respect to the host.
- **out_report_length** (int) – Size in bytes of the HID report received from the host. "Out" is with respect to the host. If no reports are expected, use 0.
- **report_id_index** (int) – position of byte in descriptor that contains the Report ID. A Report ID will be assigned when the device is created. If there is no Report ID, use `None`.

KEYBOARD :Device
Standard keyboard device supporting keycodes 0x00-0xDD, modifiers 0xE-0xE7, and five LED indicators.

MOUSE :Device
Standard mouse device supporting five mouse buttons, X and Y relative movements from -127 to 127 in each report, and a relative mouse wheel change from -127 to 127 in each report.

CONSUMER_CONTROL :Device
Consumer Control device supporting sent values from 1-652, with no rollover.

last_received_report :bytes
The HID OUT report as a *bytes*. (read-only). None if nothing received.

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

usage :int
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.

send_report(*self*, *buf:* _typing.ReadableBuffer) → None
Send a HID report.

usb_midi – MIDI over USB

The *usb_midi* module contains classes to transmit and receive MIDI messages over USB.

usb_midi.**ports :Tuple[Union[PortIn, PortOut], Ellipsis]**
Tuple of all MIDI ports. Each item is ether *PortIn* or *PortOut*.

usb_midi.**disable**() → None
Disable presenting a USB MIDI device to the host. The device is normally enabled by default, but on some boards with limited endpoints including ESP32-S2 and certain STM boards, it is disabled by default. Can be called in `boot.py`, before USB is connected.

usb_midi.**enable**() → None
Enable presenting a USB MIDI device to the host. The device is enabled by default, so you do not normally need to call this function. Can be called in `boot.py`, before USB is connected.

If you enable too many devices at once, you will run out of USB endpoints. The number of available endpoints varies by microcontroller. CircuitPython will go into safe mode after running boot.py to inform you if not enough endpoints are available.

class usb_midi.**PortIn**
Receives midi commands over USB

You cannot create an instance of *usb_midi.PortIn*.

PortIn objects are constructed for every corresponding entry in the USB descriptor and added to the `usb_midi.ports` tuple.

read(*self*, *nbytes: Optional[*int*] = None*) → Optional[*bytes*]
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(*self*, *buf:* _typing.WriteableBuffer, *nbytes: Optional[*int*] = None*) → Optional[*bytes*]
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

class usb_midi.**PortOut**
Sends midi messages to a computer over USB

You cannot create an instance of *usb_midi.PortOut*.

PortOut objects are constructed for every corresponding entry in the USB descriptor and added to the `usb_midi.ports` tuple.

write(*self*, *buf:* _typing.ReadableBuffer) → Optional[*int*]
Write the buffer of bytes to the bus.

Returns the number of bytes written

Return type *int* or None

ustack – Stack information and analysis

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

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

vectorio – Lightweight 2d shapes for displays

class vectorio.**Circle**(*radius:* int)
Circle is positioned on screen by its center point.

Parameters **radius** – The radius of the circle in pixels

radius :int
The radius of the circle in pixels.

class vectorio.**Polygon**(*points: List[Tuple[*int, int*]]*)
Represents a closed shape by ordered vertices

Parameters **points** – Vertices for the polygon

points :List[Tuple[int, int]]
Set a new look and shape for this polygon

class vectorio.**Rectangle**(*width:* int, *height:* int)
Represents a rectangle by defining its bounds

Parameters

- **width** – The number of pixels wide
- **height** – The number of pixels high

class vectorio.**VectorShape**(*shape: Union[Polygon, Rectangle, Circle]*, *pixel_shader: Union[displayio.ColorConverter, displayio.Palette]*, *x: int = 0*, *y: int = 0*)

Binds a vector shape to a location and pixel color

Parameters

- **shape** – The shape to draw.
- **pixel_shader** – The pixel shader that produces colors from values
- **x** – Initial x position of the center axis of the shape within the parent.
- **y** – Initial y position of the center axis of the shape within the parent.

x :int
X position of the center point of the shape in the parent.

y :int
Y position of the center point of the shape in the parent.

pixel_shader :Union[displayio.ColorConverter, displayio.Palette]
The pixel shader of the shape.

watchdog – Watchdog Timer

The *watchdog* module provides support for a Watchdog Timer. This timer will reset the device if it hasn't been fed after a specified amount of time. This is useful to ensure the board has not crashed or locked up. Note that on some platforms the watchdog timer cannot be disabled once it has been enabled.

The *WatchDogTimer* is used to restart the system when the application crashes and ends up into a non recoverable state. Once started it cannot be stopped or reconfigured in any way. After enabling, the application must "feed" the watchdog periodically to prevent it from expiring and resetting the system.

Example usage:

```
from microcontroller import watchdog as w
from watchdog import WatchDogMode
w.timeout=2.5 # Set a timeout of 2.5 seconds
w.mode = WatchDogMode.RAISE
w.feed()
```

class watchdog.**WatchDogMode**
run state of the watchdog timer

Enum-like class to define the run mode of the watchdog timer.

RAISE :WatchDogMode
Raise an exception when the WatchDogTimer expires.

RESET :WatchDogMode
Reset the system if the WatchDogTimer expires.

class watchdog.**WatchDogTimer**
Timer that is used to detect code lock ups and automatically reset the microcontroller when one is detected.

A lock up is detected when the watchdog hasn't been fed after a given duration. So, make sure to call *feed* within the timeout.

Not currently dynamically supported. Access the sole instance through *microcontroller.watchdog*.

timeout :float
The maximum number of seconds that can elapse between calls to feed()

mode :WatchDogMode
The current operating mode of the WatchDogTimer *watchdog.WatchDogMode*.

Setting a WatchDogMode activates the WatchDog:

```
import microcontroller
import watchdog

w = microcontroller.watchdog
w.timeout = 5
w.mode = watchdog.WatchDogMode.RAISE
```

Once set, the WatchDogTimer will perform the specified action if the timer expires.

feed(*self*) → None
Feed the watchdog timer. This must be called regularly, otherwise the timer will expire.

deinit(*self*) → None
Stop the watchdog timer. This may raise an error if the watchdog timer cannot be disabled on this platform.

wifi

The *wifi* module provides necessary low-level functionality for managing wifi connections. Use *socketpool* for communicating over the network.

wifi.**radio :Radio**
Wifi radio used to manage both station and AP modes. This object is the sole instance of *wifi.Radio*.

class wifi.**AuthMode**
The authentication protocols used by WiFi.

OPEN :object
Open network. No authentication required.

WEP :object
Wired Equivalent Privacy.

WPA :object
Wireless Protected Access.

WPA2 :object
Wireless Protected Access 2.

WPA3 :object
Wireless Protected Access 3.

PSK :object
Pre-shared Key. (password)

ENTERPRISE :object
Each user has a unique credential.

class wifi.**Network**
A wifi network provided by a nearby access point.

You cannot create an instance of *wifi.Network*. They are returned by *wifi.Radio.start_scanning_networks*.

ssid :str
String id of the network

bssid :bytes
BSSID of the network (usually the AP's MAC address)

rssi :int
Signal strength of the network

channel :int
Channel number the network is operating on

country :str
String id of the country code

authmode :str
String id of the authmode

class wifi.**Radio**
Native wifi radio.

This class manages the station and access point functionality of the native Wifi radio.

You cannot create an instance of *wifi.Radio*. Use *wifi.radio* to access the sole instance available.

enabled :bool
`True` when the wifi radio is enabled. If you set the value to `False`, any open sockets will be closed.

mac_address :bytes
MAC address of the wifi radio station. (read-only)

mac_address_ap :bytes
MAC address of the wifi radio access point. (read-only)

hostname :_typing.ReadableBuffer
Hostname for wifi interface. When the hostname is altered after interface started/connected the changes would only be reflected once the interface restarts/reconnects.

ipv4_gateway :Optional[ipaddress.IPv4Address]
IP v4 Address of the station gateway when connected to an access point. None otherwise.

ipv4_gateway_ap :Optional[ipaddress.IPv4Address]
IP v4 Address of the access point gateway, when enabled. None otherwise.

ipv4_subnet :Optional[ipaddress.IPv4Address]
IP v4 Address of the station subnet when connected to an access point. None otherwise.

ipv4_subnet_ap :Optional[ipaddress.IPv4Address]
IP v4 Address of the access point subnet, when enabled. None otherwise.

ipv4_address :Optional[ipaddress.IPv4Address]
IP v4 Address of the station when connected to an access point. None otherwise.

ipv4_address_ap :Optional[ipaddress.IPv4Address]
IP v4 Address of the access point, when enabled. None otherwise.

ipv4_dns :Optional[ipaddress.IPv4Address]
IP v4 Address of the DNS server in use when connected to an access point. None otherwise.

ap_info :Optional[Network]
Network object containing BSSID, SSID, authmode, channel, country and RSSI when connected to an access point. None otherwise.

start_scanning_networks(*self*, *, *start_channel:* int = *1*, *stop_channel:* int = *11*) → Iterable[*Network*]
Scans for available wifi networks over the given channel range. Make sure the channels are allowed in your country.

stop_scanning_networks(*self*) → None
Stop scanning for Wifi networks and free any resources used to do it.

start_station(*self*) → None
Starts a Station.

stop_station(*self*) → None
Stops the Station.

start_ap(*self*, *ssid:* _typing.ReadableBuffer, *password:* _typing.ReadableBuffer = *b''*, *, *channel: Optional[*int*] = 1*, *authmode: Optional[*AuthMode*]*) → None
Starts an Access Point with the specified ssid and password.

If `channel` is given, the access point will use that channel unless a station is already operating on a different channel.

If `authmode` is given, the access point will use that Authentication mode. If a password is given, `authmode` must not be `OPEN`. If `authmode` isn't given, `OPEN` will be used when password isn't provided, otherwise `WPA_WPA2_PSK`.

stop_ap(*self*) → None
Stops the Access Point.

connect(*self*, *ssid:* _typing.ReadableBuffer, *password:* _typing.ReadableBuffer = *b''*, *, *channel: Optional[*int*] = 0*, *bssid: Optional[*_typing.ReadableBuffer*] = b''*, *timeout: Optional[*float*] = None*) → *bool*
Connects to the given ssid and waits for an ip address. Reconnections are handled automatically once one connection succeeds.

By default, this will scan all channels and connect to the access point (AP) with the given `ssid` and greatest signal strength (rssi).

If `channel` is given, the scan will begin with the given channel and connect to the first AP with the given `ssid`. This can speed up the connection time significantly because a full scan doesn't occur.

If `bssid` is given, the scan will start at the first channel or the one given and connect to the AP with the given `bssid` and `ssid`.

ping(*self*, *ip:* ipaddress.IPv4Address, *, *timeout: Optional[*float*] = 0.5*) → *float*
Ping an IP to test connectivity. Returns echo time in seconds. Returns None when it times out.

class wifi.**ScannedNetworks**
Iterates over all *wifi.Network* objects found while scanning. This object is always created by a *wifi.Radio*: it has no user-visible constructor.

Cannot be instantiated directly. Use *wifi.Radio.start_scanning_networks*.

__iter__(*self*) → Iterator[*Network*]
Returns itself since it is the iterator.

__next__(*self*) → *Network*
Returns the next *wifi.Network*. Raises *StopIteration* if scanning is finished and no other results are available.

`wiznet` – Support for WizNet hardware, including the WizNet 5500 Ethernet adaptor.

Warning: This module is disabled in 6.x and will removed in 7.x. Please use networking libraries instead.

class wiznet.**WIZNET5K**(*spi:* busio.SPI, *cs:* microcontroller.Pin, *rst:* microcontroller.Pin, *dhcp:* bool = *True*)

Wrapper for Wiznet 5500 Ethernet interface

Create a new WIZNET5500 interface using the specified pins

Parameters

- **spi** (SPI) – spi bus to use
- **cs** (Pin) – pin to use for Chip Select
- **rst** (Pin) – pin to use for Reset (optional)
- **dhcp** (bool) – boolean flag, whether to start DHCP automatically (optional, keyword only, default True)

- The reset pin is optional: if supplied it is used to reset the wiznet board before initialization.
- The SPI bus will be initialized appropriately by this library.
- At present, the WIZNET5K object is a singleton, so only one WizNet interface is supported at a time.

connected :bool

(boolean, readonly) is this device physically connected?

dhcp :bool

(boolean, readwrite) is DHCP active on this device?

- set to True to activate DHCP, False to turn it off

ifconfig(*self*, *params: Optional[Tuple[*str, str, str, str*]] = None*) → Optional[Tuple[*str*, *str*, *str*, *str*]]

Called without parameters, returns a tuple of (ip_address, subnet_mask, gateway_address, dns_server)

Or can be called with the same tuple to set those parameters. Setting ifconfig parameters turns DHCP off, if it was on.

`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

CircuitPython supports a number of microcontroller families. Support quality for each varies depending on the active contributors for each port.

Adafruit sponsored developers are actively contributing to atmel-samd, mimxrt10xx, nrf and stm ports. They also maintain the other ports in order to ensure the boards build. Additional testing is limited.

SAMD21 and SAMD51

This port supports many development boards that utilize SAMD21 and SAMD51 chips. See https://circuitpython.org/downloads for all supported boards.

Building

For build instructions see this guide: https://learn.adafruit.com/building-circuitpython/

Debugging

For debugging instructions see this guide: https://learn.adafruit.com/debugging-the-samd21-with-gdb

Port Specific modules

CircuitPython port to Spresense

This directory contains the port of CircuitPython to Spresense. It is a compact development board based on Sony's power-efficient multicore microcontroller CXD5602.

Board features:

- Integrated GPS
  - The embedded GNSS with support for GPS, QZSS and GLONASS enables applications where tracking is required.
- Hi-res audio output and multi mic inputs
  - Advanced 192kHz/24 bit audio codec and amplifier for audio output, and support for up to 8 mic input channels.
- Multicore microcontroller
  - Spresense is powered by Sony's CXD5602 microcontroller (ARM® Cortex®-M4F × 6 cores), with a clock speed of 156 MHz.

Currently, Spresense port does not support Audio and Multicore.

Refer to developer.sony.com/develop/spresense/ for further information about this board.

Prerequisites

Linux

Add user to `dialout` group:

```
$ sudo usermod -a -G dialout <user-name>
```

Windows

Download and install USB serial driver

- CP210x USB to serial driver for Windows 7/8/8.1
- CP210x USB to serial driver for Windows 10

macOS

Download and install USB serial driver

- CP210x USB to serial driver for Mac OS X

Build instructions

Pull all submodules into your clone:

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

Build the MicroPython cross-compiler:

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

Change directory to cxd56:

```
$ cd ports/cxd56
```

To build circuitpython image run:

```
$ make BOARD=spresense
```

USB connection

Connect the `Spresense main board` to the PC via the USB cable.

Flash the bootloader

The correct bootloader is required for the Spresense board to function.

Bootloader information:

- The bootloader has to be flashed the very first time the board is used.
- You have to accept the End User License Agreement to be able to download and use the Spresense bootloader binary.

Download the spresense binaries zip archive from: Spresense firmware v2-0-002

Extract spresense binaries in your PC to ports/spresense/spresense-exported-sdk/firmware/

To flash the bootloader run the command:

```
$ make BOARD=spresense flash-bootloader
```

Flash the circuitpython image

To flash the firmware run the command:

```
$ make BOARD=spresense flash
```

Accessing the board

Connect the `Spresense extension board` to the PC via the USB cable.

Once built and deployed, access the CircuitPython REPL (the Python prompt) via USB. You can run:

```
$ screen /dev/ttyACM0 115200
```

Circuitpython on ESP32-S2

This port adds the ESP32-S2 line of modules from Espressif to Circuitpython. ESP32-S2 modules are low power, single-core Wi-Fi microcontroller SoCs designed for IoT applications.

How this port is organized:

- **bindings/** contains some required bindings to the ESP-IDF for exceptions and memory.
- **boards/** contains the configuration files for each development board and breakout available on the port.
- **common-hal/** contains the port-specific module implementations, used by shared-module and shared-bindings.
- **esp-idf/** contains the Espressif IoT development framework installation, including all the drivers for the port.
- **modules/** contains information specific to certain ESP32-S2 hardware modules, such as the pins used for flash and RAM on the WROVER and WROOM.
- **peripherals/** contains peripheral setup files and peripheral mapping information, sorted by family and sub-variant. Most files in this directory can be generated with the python scripts in **tools/**.
- **supervisor/** contains port-specific implementations of internal flash, serial and USB, as well as the **port.c** file, which initializes the port at startup.
- **tools/** includes useful python scripts for debugging and other purposes.

At the root level, refer to **mpconfigboard.h** and **mpconfigport.mk** for port specific settings and a list of enabled circuitpython modules.

Connecting to the ESP32-S2

The USB port built into ESP32-S2 boards such as the Saola is not the native USB of the board, but a debugging and programming interface. The actual ESP32-S2 native USB which exposes the Circuitpython drive and CDC connection is located on IO pins 19 and 20:

GPIO	USB
20	D+ (green)
19	D- (white)
GND	GND (black)
5V	+5V (red)

Connect these pins using a USB adapter or breakout cable to access the Circuitpython drive.

Building and flashing

Before building or flashing the ESP32-S2, you must install the esp-idf. This must be re-done every time the esp-idf is updated, but not every time you build. Run `cd ports/esp32s2` from `circuitpython/` to move to the esp32s2 port root, and run:

```
./esp-idf/install.sh
```

After this initial installation, you must add the esp-idf tools to your path. You must also do this **any time you open a new bash environment for building or flashing**:

```
. esp-idf/export.sh
```

When Circuitpython updates the ESP-IDF to a new release, you may need to run this installation process again. The exact commands used may also vary based on your bash environment.

Building boards such as the Saola is typically done through `make flash`. The default port is `tty.SLAB_USBtoUART`, which will only work on certain Mac setups. On most machines, both Mac and Linux, you will need to set the port yourself by running `ls /dev/tty.usb*` and selecting the one that only appears when your development board is plugged in. An example make command with the port setting is as follows:

```
make BOARD=espressif_saola_1_wrover flash PORT=/dev/tty.usbserial-1421120
```

Debugging

The ESP32-S2 supports JTAG debugging over OpenOCD using a JLink or other probe hardware. The official tutorials can be found on the Espressif website here, but they are mostly for the ESP32-S2 Kaluga, which has built-in debugging.

OpenOCD is automatically installed and added to your bash environment during the esp-idf installation and setup process. You can double check that it is installed by using `openocd --version`, as per the tutorial. Attach the JTAG probe pins according to the instructions for JTAG debugging on boards that do not contain an integrated debugger.

Once the debugger is connected physically, you must run OpenOCD with attached configuration files specifying the **interface** (your debugger probe) and either a **target** or a **board** (targets are for SoCs only, and can be used when a full board configuration file doesn't exist). You can find the path location of these files by checking the `OPENOCD_SCRIPTS` environmental variable by running `echo $OPENOCD_SCRIPTS` in bash. Interfaces will be in the `interface/` directory, and targets and boards in the `target/` and `board/` directories, respectively.

Note: Unfortunately, there are no board files for the esp32-s2 other than the Kaluga, and the included `target/esp32s2.cfg` target file will not work by default on the Jlink for boards like the Saola 1, as the default speed is incorrect. In addition, these files are covered under the GPL and cannot be included in Circuitpython. Thus, you must make a copy of the esp32s2.cfg file yourself and add the following line manually, under `transport select jtag` at the start of the file:

```
adapter_khz 1000
```

Once this is complete, your final OpenOCD command may look something like this:

```
openocd -f interface/jlink.cfg -f SOMEPATH/copied-esp32s2-saola-1.cfg
```

Where `SOMEPATH` is the location of your copied configuration file (this can be placed in the port/boards directory with a prefix to ignore it with `.gitignore`, for instance). Interface, target and board config files sourced from

espressif only need their paths from the $OPENOCD_SCRIPTS location, you don't need to include their full path. Once OpenOCD is running, connect to GDB with:

```
xtensa-esp32s2-elf-gdb build-espressif_saola_1_wrover/firmware.elf
```

And follow the Espressif GDB tutorial instructions for connecting, or add them to your `gdbinit`:

```
target remote :3333
set remote hardware-watchpoint-limit 2
mon reset halt
flushregs
thb app_main
c
```

LiteX (FPGA)

LiteX is a Python-based System on a Chip (SoC) designer for open source supported Field Programmable Gate Array (FPGA) chips. This means that the CPU core(s) and peripherals are not defined by the physical chip. Instead, they are loaded as separate "gateware". Once this gateware is loaded, CircuitPython can be loaded on top of it to work as expected.

Installation

You'll need `dfu-util` to install CircuitPython on the Fomu.

Make sure the foboot bootloader is updated. Instructions are here: https://github.com/im-tomu/fomu-workshop/blob/master/docs/bootloader.rst

Once you've updated the bootloader, you should know how to use `dfu-util`. It's pretty easy!

To install CircuitPython do:

```
dfu-util -D adafruit-circuitpython-fomu-en_US-<version>.dfu
```

It will install and then restart. CIRCUITPY should appear as it usually does and work the same.

CircuitPython Port To The NXP i.MX RT10xx Series

This is a port of CircuitPython to the i.MX RT10xx series of chips.

CircuitPython Port To The Nordic Semiconductor nRF52 Series

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

> **NOTE**: There are board-specific READMEs that may be more up to date than the generic board-neutral documentation below.

Flash

Some boards have UF2 bootloaders and can simply be flashed in the normal way, by copying firmware.uf2 to the BOOT drive.

For some boards, you can use the `flash` target:

```
make BOARD=pca10056 flash
```

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

run follow command to install adafruit-nrfutil from PyPi

```
$ pip3 install --user adafruit-nrfutil
```

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.

When enabled you have different options to test it:

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

RP2040

This port supports many development boards that utilize RP2040 chips. See https://circuitpython.org/downloads for all supported boards.

Building

For build instructions see this guide: https://learn.adafruit.com/building-circuitpython/

Port Specific modules

Circuitpython on STM32

This port brings the ST Microelectronics STM32 series of MCUs to Circuitpython. STM32 chips have a wide range of capability, from <$1 low power STM32F0s to dual-core STM32H7s running at 400+ MHz. Currently, only the F4, F7, and H7 families are supported, powered by the ARM Cortex M4 and M7 processors.

Refer to the ST Microelectronics website for more information on features sorted by family and individual chip lines: st.com/en/microcontrollers-microprocessors/stm32-high-performance-mcus.html

STM32 SoCs vary product-by-product in clock speed, peripheral capability, pin assignments, and their support within this port. Refer to **mpconfigport.mk** for a full list of enabled modules sorted by family.

How this port is organized:

- **boards/** contains the configuration files for each development board and breakout available on the port, as well as system files and both shared and SoC-specific linker files. Board configuration includes a pin mapping of the board, oscillator information, board-specific build flags, and setup for OLED or TFT screens where applicable.
- **common-hal/** contains the port-specific module implementations, used by shared-module and shared-bindings.
- **packages/** contains package-specific pin bindings (LQFP100, BGA216, etc)
- **peripherals/** contains peripheral setup files and peripheral mapping information, sorted by family and sub-variant. Most files in this directory can be generated with the python scripts in **tools/**.
- **st-driver/** submodule for ST HAL and LL files generated via CubeMX. Shared with TinyUSB.
- **supervisor/** contains port-specific implementations of internal flash, serial and USB, as well as the **port.c** file, which initializes the port at startup.
- **tools/** python scripts for generating peripheral and pin mapping files in **peripherals/** and **board/**.

At the root level, refer to **mpconfigboard.h** and **mpconfigport.mk** for port specific settings and a list of enabled modules.

Build instructions

Ensure your clone of Circuitpython is ready to build by following the guide on the Adafruit Website. This includes installing the toolchain, synchronizing submodules, and running `mpy-cross`.

Once the one-time build tasks are complete, you can build at any time by navigating to the port directory:

```
$ cd ports/stm
```

To build for a specific circuitpython board, run:

```
$ make BOARD=feather_stm32f405_express
```

You may also build with certain flags available in the makefile, depending on your board and development goals. The following flags would enable debug information and correct flash locations for a pre-flashed UF2 bootloader:

```
$ make BOARD=feather_stm32f405_express DEBUG=1 UF2_BOOTLOADER=1
```

USB connection

Connect your development board of choice to the host PC via the USB cable. Note that for most ST development boards such as the Nucleo and Discovery series, you must use a secondary OTG USB connector to access circuitpython, as the primary USB connector will be connected to a built-in ST-Link debugger rather than the chip itself.

In many cases, this ST-Link USB connector will **still need to be connected to power** for the chip to turn on - refer to your specific product manual for details.

Flash the bootloader

Most ST development boards come with a built-in STLink programming and debugging probe accessible via USB. This programmer may show up as an `MBED` drive on the host PC, enabling simple drag and drop programming with a .bin file, or they may require a tool like OpenOCD or StLink-org/stlink to run flashing and debugging commands.

Many hobbyist and 3rd party development boards also expose SWD pins. These can be used with a cheap stlink debugger or other common programmers.

For non-ST products or users without a debugger, all STM32 boards in the high performance families (F4, F7 and H7) include a built-in DFU bootloader stored in ROM. This bootloader is accessed by ensuring the BOOT0 pin is held to a logic 1 and the BOOT1 pin is held to a logic 0 when the chip is reset (ST Appnote AN2606). Most chips hold BOOT low by default, so this can usually be achieved by running a jumper wire from 3.3V power to the BOOT0 pin, if it is exposed, or by flipping the appropriate switch or button as the chip is reset. Once the chip is started in DFU mode, BOOT0 no longer needs to be held high and can be released. An example is available in the Feather STM32F405 guide.

Windows users will need to install stm32cubeprog, while Mac and Linux users will need to install `dfu-util` with `brew install dfu-util` or `sudo apt-get install dfu-util`. More details are available in the Feather F405 guide.

Flashing the circuitpython image with DFU-Util

Ensure the board is in dfu mode by following the steps in the previous section. Then run:

```
$ make BOARD=feather_stm32F405_express flash
```

Alternatively, you can navigate to the build directory and run the raw `dfu-util` command:

```
dfu-util -a 0 --dfuse-address 0x08000000 -D firmware.bin
```

Accessing the board

Connecting the board to the PC via the USB cable will allow code to be uploaded to the `CIRCUITPY` volume.

Circuitpython exposes a CDC virtual serial connection for REPL access and debugging. Connecting to it from OSX will look something like this:

```
screen /dev/tty.usbmodem14111201 115200
```

You may also use a program like mu to assist with REPL access.

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 it's 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 it's 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, 2.x and 3.x, and will change again between 6.x and 7.x.

So, for instance, if you just upgraded to CircuitPython 7.x from 6.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 7.0.0 or higher in the filename.

1.8.4 Additional CircuitPython 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.

Adafruit CircuitPython Library 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, it's 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 Adafruit bundles are available on GitHub: <https://github.com/adafruit/Adafruit_CircuitPython_Bundle/releases>.
- Documentation for the bundle, which includes links to documentation for all libraries, is available here: <https://circuitpython.readthedocs.io/projects/bundle/en/latest/>.

CircuitPython Community Library Bundle

This bundle contains non-Adafruit sponsored libraries, that are written and submitted by members of the community.

- The Community bundles are available on GitHub: <https://github.com/adafruit/CircuitPython_Community_Bundle/releases>.
- Documentation is not available on ReadTheDocs at this time. See each library for any included documentation.

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, readme among other files.

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.

Terminology

As our Code of Conduct states, we strive to use "welcoming and inclusive language." Whether it is in documentation or in code, the words we use matter. This means we disfavor language that due to historical and social context can make community members and potential community members feel unwelcome.

There are specific terms to avoid except where technical limitations require it. While specific cases may call for other terms, consider using these suggested terms first:

Preferred	Deprecated
Main (device)	Master
Peripheral	Slave
Sensor	
Secondary (device)	
Denylist	Blacklist
Allowlist	Whitelist

Note that "technical limitations" refers e.g., to the situation where an upstream library or URL has to contain those substrings in order to work. However, when it comes to documentation and the names of parameters and properties in CircuitPython, we will use alternate terms even if this breaks tradition with past practice.

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
import time

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
import time
```

(continues on next page)

(continued from previous page)

```
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.

Exceptions and asserts

Raise an appropriate Exception, along with a useful message, whenever a critical test or other condition fails.

Example:

```
if not 0 <= pin <= 7:
    raise ValueError("Pin number must be 0-7.")
```

If memory is constrained and a more compact method is needed, use `The assert statement` instead.

Example:

```
assert 0 <= pin <= 7, "Pin number must be 0-7."
```

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>`
================================================

<Longer description>

* Author(s):

Implementation Notes
--------------------


**Hardware:**
```

(continues on next page)

(continued from previous page)

```
* `Adafruit Device Description
  <hyperlink>`_ (Product ID: <Product Number>)

**Software and Dependencies:**

* Adafruit CircuitPython firmware for the supported boards:
  https://circuitpython.org/downloads

* Adafruit's Bus Device library:
  https://github.com/adafruit/Adafruit_CircuitPython_BusDevice

* Adafruit's Register library:
  https://github.com/adafruit/Adafruit_CircuitPython_Register

"""
```

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. Defaults to :const:`0x40`
    """

    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. Defaults to 0x40

Documenting Parameters

Although there are different ways to document class and functions definitions in Python, the following is the prevalent method of documenting parameters for CircuitPython libraries. When documenting class parameters you should use the following structure:

```
:param param_type param_name: Parameter_description
```

param_type

The type of the parameter. This could be among other *int*, *float*, *str bool*, etc. To document an object in the CircuitPython domain, you need to include a ~ before the definition as shown in the following example:

```
:param ~busio.I2C i2c_bus: The I2C bus the DS3231 is connected to.
```

To include references to CircuitPython modules, cookiecutter creates an entry in the intersphinx_mapping section in the `conf.py` file located within the `docs` directory. To add different types outside CircuitPython you need to include them in the intersphinx_mapping:

```
intersphinx_mapping = {
    "python": ("https://docs.python.org/3.4", None),
    "BusDevice":("https://circuitpython.readthedocs.io/projects/busdevice/en/latest/",
↪ None,),
    "CircuitPython": ("https://circuitpython.readthedocs.io/en/latest/", None),
}
```

The intersphinx_mapping above includes references to Python, BusDevice and CircuitPython Documentation

When the parameter have two different types, you should reference them as follows:

```
class Character_LCD:
    """Base class for character LCD

       :param ~digitalio.DigitalInOut rs: The reset data line
       :param ~pwmio.PWMOut,~digitalio.DigitalInOut blue: Blue RGB Anode

    """

    def __init__(self, rs, blue):
        self._rc = rs
        self.blue = blue
```

Renders as:

class Character_LCD(*rs*, *blue*)
Base class for character LCD

Parameters

- **rs** (DigitalInOut) – The reset data line
- **blue** (PWMOut,DigitalInOut) – Blue RGB Anode

param_name

Parameter name used in the class or method definition

Parameter_description

Parameter description. When the parameter defaults to a particular value, it is good practice to include the default:

```
:param int pitch: Pitch value for the servo. Defaults to :const:`4500`
```

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

Documentation References to other Libraries

When you need to make references to documentation in other libraries you should refer the class using single backticks `:class:`~adafruit_motor.servo.Servo``. You must also add the reference in the `conf.py` file in the `intersphinx_mapping section` by adding a new entry:

```
"adafruit_motor": ("https://circuitpython.readthedocs.io/projects/motor/en/latest/", None),
```

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):
        # 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):
```

(continues on next page)

(continued from previous page)

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

Class documentation example template

When documenting classes, you should use the following template to illustrate basic usage. It is similar with the simpletest example, however this will display the information in the Read The Docs documentation. The advantage of using this template is it makes the documentation consistent across the libraries.

This is an example for a AHT20 temperature sensor. Include the following after the class parameter:

```
"""

**Quickstart: Importing and using the AHT10/AHT20 temperature sensor**

    Here is an example of using the :class:`AHTx0` class.
    First you will need to import the libraries to use the sensor

    .. code-block:: python

        import board
        import adafruit_ahtx0

    Once this is done you can define your `board.I2C` object and define your sensor␣
↪object

    .. code-block:: python

        i2c = board.I2C()  # uses board.SCL and board.SDA
        aht = adafruit_ahtx0.AHTx0(i2c)

    Now you have access to the temperature and humidity using
    the :attr:`temperature` and :attr:`relative_humidity` attributes

    .. code-block:: python

        temperature = aht.temperature
        relative_humidity = aht.relative_humidity

"""
```

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, it's 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.

It's 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*.

Use of MicroPython `const()`

The MicroPython `const()` feature, as discussed in this forum post, and in this issue thread, provides some optimizations that can be useful on smaller, memory constrained devices. However, when using `const()`, keep in mind these general guide lines:

- Always use via an import, ex: `from micropython import const`
- Limit use to global (module level) variables only.
- If user will not need access to variable, prefix name with a leading underscore, ex: `_SOME_CONST`.

Libraries Examples

When adding examples, cookiecutter will add a `<name>_simpletest.py` file in the examples directory for you. Be sure to include code with the library minimal functionalities to work on a device. You could other examples if needed featuring different functionalities of the library. If you add additional examples, be sure to include them in the `examples.rst`. Naming of the examples files should use the name of the library followed by a description, using underscore to separate them. When using print statements you should use the `" ".format()` format, as there are particular boards that are not capable to use f-strings.

```
text_to_display = "World!"

print("Hello {}".format(text_to_display))
```

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 Celsius
`CO2`	float	measured CO2 in ppm
`eCO2`	float	equivalent/estimated CO2 in ppm (estimated from some other measurement)
`TVOC`	float	Total Volatile Organic Compounds in ppb
`distance`	float	centimeters (cm)
`proximity`	int	non-unit-specific proximity values (monotonic but not actual distance)
`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 (Hz)
`value`	bool	Digital logic
`value`	int	16-bit Analog value, unit-less
`weight`	float	grams (g)
`sound_level`	float	non-unit-specific sound level (monotonic but not actual decibels)

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):

Build CircuitPython

MicroPython compatibility

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

We love CircuitPython and would love to see it come to more microcontroller platforms. Since 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 complete, 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!

Circuitpython has a number of modules enabled by default in `py/circuitpy_mpconfig.mk`. Most of these modules will need to be disabled in `mpconfigboard.mk` during the early stages of a port in order for it to compile. As the port progresses in module support, this list can be pruned down as a natural "TODO" list. An example minimal build list is shown below:

```
# These modules are implemented in ports/<port>/common-hal:

# Typically the first module to create
CIRCUITPY_MICROCONTROLLER = 0
# Typically the second module to create
CIRCUITPY_DIGITALIO = 0
# Other modules:
CIRCUITPY_ANALOGIO = 0
CIRCUITPY_BUSIO = 0
CIRCUITPY_COUNTIO = 0
CIRCUITPY_NEOPIXEL_WRITE = 0
CIRCUITPY_PULSEIO = 0
CIRCUITPY_OS = 0
CIRCUITPY_NVM = 0
CIRCUITPY_AUDIOBUSIO = 0
CIRCUITPY_AUDIOIO = 0
CIRCUITPY_ROTARYIO = 0
CIRCUITPY_RTC = 0
CIRCUITPY_SDCARDIO = 0
CIRCUITPY_FRAMEBUFFERIO = 0
CIRCUITPY_FREQUENCYIO = 0
CIRCUITPY_I2CPERIPHERAL = 0
# Requires SPI, PulseIO (stub ok):
CIRCUITPY_DISPLAYIO = 0

# These modules are implemented in shared-module/ - they can be included in
# any port once their prerequisites in common-hal are complete.
# Requires DigitalIO:
CIRCUITPY_BITBANGIO = 0
# Requires DigitalIO
CIRCUITPY_GAMEPAD = 0
# Requires neopixel_write or SPI (dotstar)
CIRCUITPY_PIXELBUF = 0
# Requires OS
CIRCUITPY_RANDOM = 0
# Requires OS, filesystem
CIRCUITPY_STORAGE = 0
# Requires Microcontroller
CIRCUITPY_TOUCHIO = 0
# Requires USB
CIRCUITPY_USB_HID = 0
CIRCUITPY_USB_MIDI = 0
# Does nothing without I2C
CIRCUITPY_REQUIRE_I2C_PULLUPS = 0
# No requirements, but takes extra flash
CIRCUITPY_ULAB = 0
```

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. It's 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 \
        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 \
```

(continues on next page)

(continued from previous page)

```
    { 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

The libraries below are inherited from MicroPython. They are similar to the standard Python libraries with the same name. They implement a subset of or a variant of the corresponding standard Python library.

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

These libraries are not enabled on CircuitPython builds with limited flash memory, usually on non-Express builds: `binascii`, `errno`, `json`, `re`.

These libraries are not currently enabled in any CircuitPython build, but may be in the future, with the `u` prefix dropped: `uctypes`, `uhashlib`, `uzlib`.

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

exception RuntimeError

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

exception StopIteration

exception SyntaxError

exception SystemExit
See CPython documentation: SystemExit.

exception TypeError
See CPython documentation: TypeError.

exception ValueError

exception ZeroDivisionError

`uheapq` – heap queue algorithm

Warning: Though this MicroPython-based library may be available for use in some builds of CircuitPython, it is unsupported and 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 will likely 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 min heap queue algorithm.

A heap queue is essentially a list that has its elements stored in such a way that the first item of the list is always the smallest.

Functions

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

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

The returned item will be the smallest item in the `heap`.

`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 `val` to the end of array, growing it.

`extend`(*iterable*)
Append new elements as contained in `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 the bytes in the *data* object to a hexadecimal representation. Returns a bytes object.

If the 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 may be available for use in some builds of CircuitPython, it is unsupported and 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 will likely 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.`**`deque`**(*iterable*, *maxlen*[, *flags*])
Deques (double-ended queues) are a list-like container that support O(1) appends and pops from either side of the deque. New deques are created using the following arguments:

- *iterable* must be the empty tuple, and the new deque is created empty.
- *maxlen* must be specified and the deque will be bounded to this maximum length. Once the deque is full, any new items added will discard items from the opposite end.
- The optional *flags* can be 1 to check for overflow when adding items.

As well as supporting `bool` and `len`, deque objects have the following methods:

deque.**append**(*x*)
Add *x* to the right side of the deque. Raises IndexError if overflow checking is enabled and there is no more room left.

deque.**popleft**()
Remove and return an item from the left side of the deque. Raises IndexError if no items are present.

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
```

errno – system error codes

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. A particular inventory of codes depends on *MicroPython port*.

Constants

EEXIST, EAGAIN, etc.
Error codes, based on ANSI C/POSIX standard. All error codes start with "E". As mentioned above, inventory of the codes depends on *MicroPython port*. 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] == errno.EEXIST:
        print("Directory already exists")
```

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

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

gc – control the garbage collector

Warning: Though this MicroPython-based library may be available for use in some builds of CircuitPython, it is unsupported and 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 will likely 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 may be available for use in some builds of CircuitPython, it is unsupported and 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 will likely 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 interoperability will try to provide this.
- MD5 - A legacy algorithm, not considered cryptographically secure. Only selected boards, targeting interoperability 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.

io – input/output streams

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

`io.`**`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 `io.`**`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 `io.`**`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 `io.`**`StringIO`**([*string*])

class `io.`**`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.

`json` – JSON encoding and decoding

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

`json.`**`dump`**(*obj*, *stream*)
Serialise `obj` to a JSON string, writing it to the given *stream*.

`json.`**`dumps`**(*obj*)
Return `obj` represented as a JSON string.

`json.`**`load`**(*stream*)
Parse the given `stream`, interpreting it as a JSON string and deserialising the data to a Python object. The resulting object is returned.

Parsing continues until end-of-file is encountered. A *`ValueError`* is raised if the data in `stream` is not correctly formed.

`json.`**`loads`**(*str*)
Parse the JSON *str* and return an object. Raises *`ValueError`* if the string is not correctly formed.

`re` – simple regular expressions

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 and special sequences are:

`.` Match any character.

`[...]` Match set of characters. Individual characters and ranges are supported, including negated sets (e.g. `[^a-c]`).

`^` Match the start of the string.

`$` Match the end of the string.

`?` Match zero or one of the previous sub-pattern.

`*` Match zero or more of the previous sub-pattern.

`+` Match one or more of the previous sub-pattern.

`??` Non-greedy version of `?`, match zero or one, with the preference for zero.

`*?` Non-greedy version of `*`, match zero or more, with the preference for the shortest match.

`+?` Non-greedy version of `+`, match one or more, with the preference for the shortest match.

`|` Match either the left-hand side or the right-hand side sub-patterns of this operator.

`(...)` Grouping. Each group is capturing (a substring it captures can be accessed with *`match.group()`* method).

`\d` Matches digit. Equivalent to `[0-9]`.

\D Matches non-digit. Equivalent to `[^0-9]`.

\s Matches whitespace. Equivalent to `[ \t-\r]`.

\S Matches non-whitespace. Equivalent to `[^ \t-\r]`.

\w Matches "word characters" (ASCII only). Equivalent to `[A-Za-z0-9_]`.

\W Matches non "word characters" (ASCII only). Equivalent to `[^A-Za-z0-9_]`.

**** Escape character. Any other character following the backslash, except for those listed above, is taken literally. For example, `\*` is equivalent to literal `*` (not treated as the `*` operator). Note that `\r`, `\n`, etc. are not handled specially, and will be equivalent to literal letters `r`, `n`, etc. Due to this, it's not recommended to use raw Python strings (`r""`) for regular expressions. For example, `r"\r\n"` when used as the regular expression is equivalent to `"rn"`. To match CR character followed by LF, use `"\r\n"`.

NOT SUPPORTED:

- counted repetitions (`{m,n}`)
- named groups (`(?P<name>...)`)
- non-capturing groups (`(?:...)`)
- more advanced assertions (`\b`, `\B`)
- special character escapes like `\r`, `\n` - use Python's own escaping instead
- etc.

Example:

```
import ure

# As ure doesn't support escapes itself, use of r"" strings is not
# recommended.
regex = ure.compile("[\r\n]")

regex.split("line1\rline2\nline3\r\n")

# Result:
# ['line1', 'line2', 'line3', '', '']
```

Functions

re.**compile**(*regex_str*[, *flags*])
: Compile regular expression, return *regex* object.

re.**match**(*regex_str*, *string*)
: Compile *regex_str* and match against *string*. Match always happens from starting position in a string.

re.**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).

re.**sub**(*regex_str*, *replace*, *string*, *count=0*, *flags=0*, /)
: Compile *regex_str* and search for it in *string*, replacing all matches with *replace*, and returning the new string.

  replace can be a string or a function. If it is a string then escape sequences of the form `\<number>` and `\g<number>` can be used to expand to the corresponding group (or an empty string for unmatched groups). If *replace* is a function then it must take a single argument (the match) and should return a replacement string.

If *count* is specified and non-zero then substitution will stop after this many substitutions are made. The *flags* argument is ignored.

Note: availability of this function depends on *MicroPython port*.

`re.`**`DEBUG`**
Flag value, display debug information about compiled expression. (Availability depends on *MicroPython port*.)

Regex objects

Compiled regular expression. Instances of this class are created using `re.compile()`.

`regex.`**`match`**`(`*string*`)`
`regex.`**`search`**`(`*string*`)`
`regex.`**`sub`**`(`*replace*, *string*, *count=0*, *flags=0*, `/)`
Similar to the module-level functions `match()`, `search()` and `sub()`. 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, and passed to the replacement function in `sub()`.

`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.

`match.`**`groups`**`()`
Return a tuple containing all the substrings of the groups of the match.

Note: availability of this method depends on *MicroPython port*.

`match.`**`start`**`(`[*index*]`)`
`match.`**`end`**`(`[*index*]`)`
Return the index in the original string of the start or end of the substring group that was matched. *index* defaults to the entire group, otherwise it will select a group.

Note: availability of these methods depends on *MicroPython port*.

`match.`**`span`**`(`[*index*]`)`
Returns the 2-tuple `(match.start(index), match.end(index))`.

Note: availability of this method depends on *MicroPython port*.

`sys` – system specific functions

Warning: Though this MicroPython-based library may be available for use in some builds of CircuitPython, it is unsupported and 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 will likely 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`*.

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
```

(continues on next page)

(continued from previous page)

```
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.

Difference to CPython

Only the first three version numbers (major, minor, micro) are supported and they can be referenced only by index, not by name.

`uasyncio` — asynchronous I/O scheduler

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

Example:

```
import uasyncio

async def blink(led, period_ms):
    while True:
        led.on()
        await uasyncio.sleep_ms(5)
        led.off()
```

(continues on next page)

(continued from previous page)

```
        await uasyncio.sleep_ms(period_ms)

async def main(led1, led2):
    uasyncio.create_task(blink(led1, 700))
    uasyncio.create_task(blink(led2, 400))
    await uasyncio.sleep_ms(10_000)

# Running on a pyboard
from pyb import LED
uasyncio.run(main(LED(1), LED(2)))

# Running on a generic board
from machine import Pin
uasyncio.run(main(Pin(1), Pin(2)))
```

Core functions

uasyncio.**create_task**(*coro*)
Create a new task from the given coroutine and schedule it to run.

Returns the corresponding *Task* object.

uasyncio.**current_task**()
Return the *Task* object associated with the currently running task.

uasyncio.**run**(*coro*)
Create a new task from the given coroutine and run it until it completes.

Returns the value returned by *coro*.

uasyncio.**sleep**(*t*)
Sleep for *t* seconds (can be a float).

This is a coroutine.

uasyncio.**sleep_ms**(*t*)
Sleep for *t* milliseconds.

This is a coroutine, and a MicroPython extension.

Additional functions

uasyncio.**wait_for**(*awaitable*, *timeout*)
Wait for the *awaitable* to complete, but cancel it if it takes longer that *timeout* seconds. If *awaitable* is not a task then a task will be created from it.

If a timeout occurs, it cancels the task and raises `asyncio.TimeoutError`: this should be trapped by the caller.

Returns the return value of *awaitable*.

This is a coroutine.

uasyncio.**wait_for_ms**(*awaitable*, *timeout*)
Similar to *wait_for* but *timeout* is an integer in milliseconds.

This is a coroutine, and a MicroPython extension.

uasyncio.**gather**(**awaitables*, *return_exceptions=False*)
Run all *awaitables* concurrently. Any *awaitables* that are not tasks are promoted to tasks.

Returns a list of return values of all *awaitables*.

This is a coroutine.

class Task

class uasyncio.**Task**
This object wraps a coroutine into a running task. Tasks can be waited on using `await task`, which will wait for the task to complete and return the return value of the task.

Tasks should not be created directly, rather use *create_task* to create them.

Task.**cancel**()
Cancel the task by injecting a `CancelledError` into it. The task may or may not ignore this exception.

class Event

class uasyncio.**Event**
Create a new event which can be used to synchronise tasks. Events start in the cleared state.

Event.**is_set**()
Returns `True` if the event is set, `False` otherwise.

Event.**set**()
Set the event. Any tasks waiting on the event will be scheduled to run.

Event.**clear**()
Clear the event.

Event.**wait**()
Wait for the event to be set. If the event is already set then it returns immediately.

This is a coroutine.

class Lock

class uasyncio.**Lock**
Create a new lock which can be used to coordinate tasks. Locks start in the unlocked state.

In addition to the methods below, locks can be used in an `async with` statement.

Lock.**locked**()
Returns `True` if the lock is locked, otherwise `False`.

Lock.**acquire**()
Wait for the lock to be in the unlocked state and then lock it in an atomic way. Only one task can acquire the lock at any one time.

This is a coroutine.

Lock.**release**()
Release the lock. If any tasks are waiting on the lock then the next one in the queue is scheduled to run and the lock remains locked. Otherwise, no tasks are waiting an the lock becomes unlocked.

TCP stream connections

uasyncio.**open_connection**(*host*, *port*)
Open a TCP connection to the given *host* and *port*. The *host* address will be resolved using *socket.getaddrinfo*, which is currently a blocking call.

Returns a pair of streams: a reader and a writer stream. Will raise a socket-specific `OSError` if the host could not be resolved or if the connection could not be made.

This is a coroutine.

uasyncio.**start_server**(*callback*, *host*, *port*, *backlog=5*)
Start a TCP server on the given *host* and *port*. The *callback* will be called with incoming, accepted connections, and be passed 2 arguments: reader and writer streams for the connection.

Returns a *Server* object.

This is a coroutine.

class uasyncio.**Stream**
This represents a TCP stream connection. To minimise code this class implements both a reader and a writer, and both `StreamReader` and `StreamWriter` alias to this class.

Stream.**get_extra_info**(*v*)
Get extra information about the stream, given by *v*. The valid values for *v* are: `peername`.

Stream.**close**()
Close the stream.

Stream.**wait_closed**()
Wait for the stream to close.

This is a coroutine.

Stream.**read**(*n*)
Read up to *n* bytes and return them.

This is a coroutine.

Stream.**readinto**(*buf*)
Read up to n bytes into *buf* with n being equal to the length of *buf*.

Return the number of bytes read into *buf*.

This is a coroutine, and a MicroPython extension.

Stream.**readline**()
Read a line and return it.

This is a coroutine.

Stream.**write**(*buf*)
Accumulated *buf* to the output buffer. The data is only flushed when *Stream.drain* is called. It is recommended to call *Stream.drain* immediately after calling this function.

Stream.**drain**()
Drain (write) all buffered output data out to the stream.

This is a coroutine.

class uasyncio.**Server**
This represents the server class returned from *start_server*. It can be used in an `async with` statement to close the server upon exit.

Server.**close**()
Close the server.

Server.**wait_closed**()
Wait for the server to close.

This is a coroutine.

Event Loop

uasyncio.**get_event_loop**()
Return the event loop used to schedule and run tasks. See *Loop*.

uasyncio.**new_event_loop**()
Reset the event loop and return it.

Note: since MicroPython only has a single event loop this function just resets the loop's state, it does not create a new one.

class uasyncio.**Loop**
This represents the object which schedules and runs tasks. It cannot be created, use *get_event_loop* instead.

Loop.**create_task**(*coro*)
Create a task from the given *coro* and return the new *Task* object.

Loop.**run_forever**()
Run the event loop until *stop()* is called.

Loop.**run_until_complete**(*awaitable*)
Run the given *awaitable* until it completes. If *awaitable* is not a task then it will be promoted to one.

Loop.**stop**()
Stop the event loop.

Loop.**close**()
Close the event loop.

Loop.**set_exception_handler**(*handler*)
Set the exception handler to call when a Task raises an exception that is not caught. The *handler* should accept two arguments: `(loop, context)`.

Loop.**get_exception_handler**()
Get the current exception handler. Returns the handler, or `None` if no custom handler is set.

Loop.**default_exception_handler**(*context*)
The default exception handler that is called.

Loop.**call_exception_handler**(*context*)
Call the current exception handler. The argument *context* is passed through and is a dictionary containing keys: `'message'`, `'exception'`, `'future'`.

`uctypes` – access binary data in a structured way

Warning: Though this MicroPython-based library may be available for use in some builds of CircuitPython, it is unsupported and 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 will likely 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 then access it using familiar dot-syntax to reference sub-fields.

Warning: `uctypes` module allows access to arbitrary memory addresses of the machine (including I/O and control registers). Uncareful usage of it may lead to crashes, data loss, and even hardware malfunction.

See also:

Module *`struct`* Standard Python way to access binary data structures (doesn't scale well to large and complex structures).

Usage examples:

```
import uctypes

# Example 1: Subset of ELF file header
# https://wikipedia.org/wiki/Executable_and_Linkable_Format#File_header
ELF_HEADER = {
    "EI_MAG": (0x0 | uctypes.ARRAY, 4 | uctypes.UINT8),
    "EI_DATA": 0x5 | uctypes.UINT8,
    "e_machine": 0x12 | uctypes.UINT16,
}

# "f" is an ELF file opened in binary mode
buf = f.read(uctypes.sizeof(ELF_HEADER, uctypes.LITTLE_ENDIAN))
header = uctypes.struct(uctypes.addressof(buf), ELF_HEADER, uctypes.LITTLE_ENDIAN)
assert header.EI_MAG == b"\x7fELF"
assert header.EI_DATA == 1, "Oops, wrong endianness. Could retry with uctypes.BIG_
↪ENDIAN."
print("machine:", hex(header.e_machine))


# Example 2: In-memory data structure, with pointers
COORD = {
    "x": 0 | uctypes.FLOAT32,
    "y": 4 | uctypes.FLOAT32,
}

STRUCT1 = {
    "data1": 0 | uctypes.UINT8,
    "data2": 4 | uctypes.UINT32,
    "ptr": (8 | uctypes.PTR, COORD),
}
```

(continues on next page)

(continued from previous page)

```
# Suppose you have address of a structure of type STRUCT1 in "addr"
# uctypes.NATIVE is optional (used by default)
struct1 = uctypes.struct(addr, STRUCT1, uctypes.NATIVE)
print("x:", struct1.ptr[0].x)


# Example 3: Access to CPU registers. Subset of STM32F4xx WWDG block
WWDG_LAYOUT = {
    "WWDG_CR": (0, {
        # BFUINT32 here means size of the WWDG_CR register
        "WDGA": 7 << uctypes.BF_POS | 1 << uctypes.BF_LEN | uctypes.BFUINT32,
        "T": 0 << uctypes.BF_POS | 7 << uctypes.BF_LEN | uctypes.BFUINT32,
    }),
    "WWDG_CFR": (4, {
        "EWI": 9 << uctypes.BF_POS | 1 << uctypes.BF_LEN | uctypes.BFUINT32,
        "WDGTB": 7 << uctypes.BF_POS | 2 << uctypes.BF_LEN | uctypes.BFUINT32,
        "W": 0 << uctypes.BF_POS | 7 << uctypes.BF_LEN | uctypes.BFUINT32,
    }),
}

WWDG = uctypes.struct(0x40002c00, WWDG_LAYOUT)

WWDG.WWDG_CFR.WDGTB = 0b10
WWDG.WWDG_CR.WDGA = 1
print("Current counter:", WWDG.WWDG_CR.T)
```

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:

```
{
    "field1": <properties>,
    "field2": <properties>,
    ...
}
```

Currently, `uctypes` requires explicit specification of offsets for each field. Offset are given in bytes from the structure start.

Following are encoding examples for various field types:

- Scalar types:

  ```
  "field_name": offset | uctypes.UINT32
  ```

  in other words, the value is a scalar type identifier ORed with a field offset (in bytes) from the start of the structure.

- Recursive structures:

  ```
  "sub": (offset, {
      "b0": 0 | uctypes.UINT8,
      "b1": 1 | uctypes.UINT8,
  })
  ```

i.e. value is a 2-tuple, first element of which is an offset, and second is a structure descriptor dictionary (note: offsets in recursive descriptors are relative to the structure it defines). Of course, recursive structures can be specified not just by a literal dictionary, but by referring to a structure descriptor dictionary (defined earlier) by name.

- Arrays of primitive types:

```
"arr": (offset | uctypes.ARRAY, size | uctypes.UINT8),
```

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 the array.

- Arrays of aggregate types:

```
"arr2": (offset | uctypes.ARRAY, size, {"b": 0 | uctypes.UINT8}),
```

i.e. value is a 3-tuple, first element of which is ARRAY flag ORed with offset, second is a number of elements in the array, and third is a descriptor of element type.

- Pointer to a primitive type:

```
"ptr": (offset | uctypes.PTR, uctypes.UINT8),
```

i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, and second is a scalar element type.

- Pointer to an aggregate type:

```
"ptr2": (offset | uctypes.PTR, {"b": 0 | uctypes.UINT8}),
```

i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, second is a descriptor of type pointed to.

- Bitfields:

```
"bitf0": offset | uctypes.BFUINT16 | lsbit << uctypes.BF_POS | bitsize << uctypes.
↪BF_LEN,
```

i.e. value is a 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 position and bit length of the bitfield within the scalar value, shifted by BF_POS and BF_LEN bits, respectively. A bitfield position is counted from the least significant bit of the scalar (having position of 0), 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 extract the bitfield).

In the example above, first a 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 *lsbit* bit of this UINT16, and length is *bitsize* bits, will be extracted. For example, if *lsbit* is 0 and *bitsize* is 8, then effectively it 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 the 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*, *layout_type=NATIVE*, */*)
Return size of data structure in bytes. The *struct* argument can be either a structure class or a 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.

uctypes.**UINT8**
uctypes.**INT8**
uctypes.**UINT16**
uctypes.**INT16**
uctypes.**UINT32**
uctypes.**INT32**
uctypes.**UINT64**
uctypes.**INT64**
Integer types for structure descriptors. Constants for 8, 16, 32, and 64 bit types are provided, both signed and unsigned.

uctypes.**FLOAT32**
uctypes.**FLOAT64**
Floating-point types for structure descriptors.

uctypes.**VOID**
`VOID` is an alias for `UINT8`, and is provided to conveniently define C's void pointers: `(uctypes.PTR, uctypes.VOID)`.

uctypes.**PTR**
uctypes.**ARRAY**
Type constants for pointers and arrays. Note that there is no explicit constant for structures, it's implicit: an aggregate type without `PTR` or `ARRAY` flags is a structure.

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 also supported, with the same semantics as in C.

Summing up, accessing structure fields generally follows the C syntax, except for pointer dereference, when you need to use `[0]` operator instead of `*`.

Limitations

1. 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 accessing 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`. Or just cache a particular peripheral: `peripheral_a = mcu_registers.peripheral_a`. If a register consists of multiple bitfields, you would need to cache references to a particular register: `reg_a = mcu_registers.peripheral_a.reg_a`.
- Avoid other non-scalar data, like arrays. For example, instead of `peripheral_a.register[0]` use `peripheral_a.register0`. Again, an alternative is to cache intermediate values, e.g. `register0 = peripheral_a.register[0]`.

2. Range of offsets supported by the `uctypes` module is limited. The exact range supported is considered an implementation detail, and the general suggestion is to split structure definitions to cover from a few kilobytes to a few dozen of kilobytes maximum. In most cases, this is a natural situation anyway, e.g. it doesn't make sense to define all registers of an MCU (spread over 32-bit address space) in one structure, but rather a peripheral block by peripheral block. In some extreme cases, you may need to split a structure in several parts artificially (e.g. if accessing native data structure with multi-megabyte array in the middle, though that would be a very synthetic case).

`uselect` – wait for events on a set of streams

Warning: Though this MicroPython-based library may be available for use in some builds of CircuitPython, it is unsupported and 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 will likely 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 `stream` objects (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 `stream` *obj* for polling. *eventmask* is logical OR of:

- `uselect.POLLIN` - data available for reading
- `uselect.POLLOUT` - more data can be written

Note that flags like `uselect.POLLHUP` and `uselect.POLLERR` are *not* valid as input eventmask (these are unsolicited events which will be returned from *`poll()`* regardless of whether they are asked for). This semantics is per POSIX.

eventmask defaults to `uselect.POLLIN | uselect.POLLOUT`.

It is OK to call this function multiple times for the same *obj*. Successive calls will update *obj*'s eventmask to the value of *eventmask* (i.e. will behave as *`modify()`*).

`poll.`**`unregister`**`(`*obj*`)`
Unregister *obj* from polling.

`poll.`**`modify`**`(`*obj*, *eventmask*`)`
Modify the *eventmask* for *obj*. If *obj* is not registered, *`OSError`* is raised with error of ENOENT.

`poll.`**`poll`**`(`*timeout=-1*, */*`)`
Wait for at least one of the registered objects to become ready or have an exceptional condition, with optional timeout in milliseconds (if *timeout* arg is not specified or -1, there is no timeout).

Returns list of (`obj`, `event`, ...) tuples. There may be other elements in tuple, depending on a platform and version, so don't assume that its size is 2. The `event` element specifies which events happened with a stream and is a combination of `uselect.POLL*` constants described above. Note that flags `uselect.POLLHUP` and `uselect.POLLERR` can be returned at any time (even if were not asked for), and must be acted on accordingly (the corresponding stream unregistered from poll and likely closed), because otherwise all further invocations of *poll()* may return immediately with these flags set for this stream again.

In case of timeout, an empty list is returned.

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 a `callee-owned tuples`. This function provides efficient, allocation-free way to poll on streams.

If *flags* is 1, one-shot behaviour for events is employed: streams for which events happened will have their event masks 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 behaviour is useful for asynchronous I/O schedulers.

Difference to CPython

This function is a MicroPython extension.

`uzlib` – zlib decompression

Warning: Though this MicroPython-based library may be available for use in some builds of CircuitPython, it is unsupported and 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 will likely 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 small builds (usually non-Express), 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 significantly or be removed in future versions of CircuitPython.

`btree` – simple BTree database

Warning: Though this MicroPython-based library may be available for use in some builds of CircuitPython, it is unsupported and 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 will likely 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
```

(continues on next page)

(continued from previous page)

```
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()

# 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 may be available for use in some builds of CircuitPython, it is unsupported and 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 will likely 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 = framebuf.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 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.**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 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.**RGB565**
Red Green Blue (16-bit, 5+6+5) color format

framebuf.**GS2_HMSB**
Grayscale (2-bit) color format

framebuf.**GS4_HMSB**
Grayscale (4-bit) color format

framebuf.**GS8**
Grayscale (8-bit) color format

micropython – access and control MicroPython internals

Warning: Though this MicroPython-based library may be available for use in some builds of CircuitPython, it is unsupported and 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 will likely 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.

The optimisation level controls the following compilation features:

- Assertions: at level 0 assertion statements are enabled and compiled into the bytecode; at levels 1 and higher assertions are not compiled.
- Built-in `__debug__` variable: at level 0 this variable expands to `True`; at levels 1 and higher it expands to `False`.
- Source-code line numbers: at levels 0, 1 and 2 source-code line number are stored along with the bytecode so that exceptions can report the line number they occurred at; at levels 3 and higher line numbers are not stored.

The default optimisation level is usually level 0.

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**()

micropython.**heap_locked**()
Lock or unlock the heap. When locked no memory allocation can occur and a *MemoryError* will be raised if any heap allocation is attempted. *heap_locked()* returns a true value if the heap is currently locked.

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.

Both *heap_unlock()* and *heap_locked()* return the current lock depth (after unlocking for the former) as a non-negative integer, with 0 meaning the heap is not locked.

If the REPL becomes active with the heap locked then it will be forcefully unlocked.

Note: *heap_locked()* is not enabled on most ports by default, requires `MICROPY_PY_MICROPYTHON_HEAP_LOCKED`.

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.

Note: If *schedule()* is called from a preempting IRQ, when memory allocation is not allowed and the callback to be passed to *schedule()* is a bound method, passing this directly will fail. This is because creating a reference to a bound method causes memory allocation. A solution is to create a reference to the method in the class constructor and to pass that reference to *schedule()*.

There is a finite queue to hold the scheduled functions and *schedule()* will raise a *RuntimeError* if the queue is full.

1.8.10 Glossary

baremetal A system without a (full-fledged) operating system, for example an *MCU*-based system. When running on a baremetal system, MicroPython effectively functions like a small operating system, running user programs and providing a command interpreter (*REPL*).

buffer protocol Any Python object that can be automatically converted into bytes, such as `bytes`, `bytearray`, `memoryview` and `str` objects, which all implement the "buffer protocol".

board Typically this refers to a printed circuit board (PCB) containing a *microcontroller* and supporting components. MicroPython firmware is typically provided per-board, as the firmware contains both MCU-specific functionality but also board-level functionality such as drivers or pin names.

bytecode A compact representation of a Python program that generated by compiling the Python source code. This is what the VM actually executes. Bytecode is typically generated automatically at runtime and is invisible to the user. Note that while *CPython* and MicroPython both use bytecode, the format is different. You can also pre-compile source code offline using the *cross-compiler*.

callee-owned tuple This is a MicroPython-specific construct where, for efficiency reasons, some built-in functions or methods may re-use the same underlying tuple object to return data. This avoids having to allocate a new tuple for every call, and reduces heap fragmentation. Programs should not hold references to callee-owned tuples and instead only extract data from them (or make a copy).

CircuitPython A variant of MicroPython developed by Adafruit Industries.

CPython CPython is the reference implementation of the Python programming language, and the most well-known one. It is, however, one of many implementations (including Jython, IronPython, PyPy, and MicroPython). While MicroPython's implementation differs substantially from CPython, it aims to maintain as much compatibility as possible.

cross-compiler Also known as `mpy-cross`. This tool runs on your PC and converts a *.py file* containing MicroPython code into a *.mpy file* containing MicroPython bytecode. This means it loads faster (the board doesn't have to compile the code), and uses less space on flash (the bytecode is more space efficient).

driver A MicroPython library that implements support for a particular component, such as a sensor or display.

FFI Acronym for Foreign Function Interface. A mechanism used by the *MicroPython Unix port* to access operating system functionality. This is not available on *baremetal* ports.

filesystem Most MicroPython ports and boards provide a filesystem stored in flash that is available to user code via the standard Python file APIs such as `open()`. Some boards also make this internal filesystem accessible to the host via USB mass-storage.

frozen module A Python module that has been cross compiled and bundled into the firmware image. This reduces RAM requirements as the code is executed directly from flash.

Garbage Collector A background process that runs in Python (and MicroPython) to reclaim unused memory in the *heap*.

GPIO General-purpose input/output. The simplest means to control electrical signals (commonly referred to as "pins") on a microcontroller. GPIO typically allows pins to be either input or output, and to set or get their digital value (logical "0" or "1"). MicroPython abstracts GPIO access using the `machine.Pin` and `machine.Signal` classes.

GPIO port A group of *GPIO* pins, usually based on hardware properties of these pins (e.g. controllable by the same register).

heap A region of RAM where MicroPython stores dynamic data. It is managed automatically by the *Garbage Collector*. Different MCUs and boards have vastly different amounts of RAM available for the heap, so this will affect how complex your program can be.

interned string An optimisation used by MicroPython to improve the efficiency of working with strings. An interned string is referenced by its (unique) identity rather than its address and can therefore be quickly compared just by its identifier. It also means that identical strings can be de-duplicated in memory. String interning is almost always invisible to the user.

MCU Microcontroller. Microcontrollers usually have much less resources than a desktop, laptop, or phone, but are smaller, cheaper and require much less power. MicroPython is designed to be small and optimized enough to run on an average modern microcontroller.

MicroPython port MicroPython supports different *boards*, RTOSes, and OSes, and can be relatively easily adapted to new systems. MicroPython with support for a particular system is called a "port" to that system. Different ports may have widely different functionality. This documentation is intended to be a reference of the generic APIs available across different ports ("MicroPython core"). Note that some ports may still omit some APIs described here (e.g. due to resource constraints). Any such differences, and port-specific extensions beyond the MicroPython core functionality, would be described in the separate port-specific documentation.

MicroPython Unix port The unix port is one of the major *MicroPython ports*. It is intended to run on POSIX-compatible operating systems, like Linux, MacOS, FreeBSD, Solaris, etc. It also serves as the basis of Windows port. The Unix port is very useful for quick development and testing of the MicroPython language and machine-independent features. It can also function in a similar way to *CPython*'s `python` executable.

.mpy file The output of the *cross-compiler*. A compiled form of a *.py file* that contains MicroPython bytecode instead of Python source code.

native Usually refers to "native code", i.e. machine code for the target microcontroller (such as ARM Thumb, Xtensa, x86/x64). The `@native` decorator can be applied to a MicroPython function to generate native code instead of bytecode for that function, which will likely be faster but use more RAM.

port Usually short for *MicroPython port*, but could also refer to *GPIO port*.

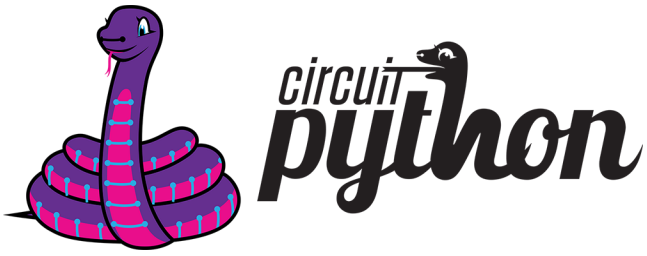
.py file A file containing Python source code.

REPL An acronym for "Read, Eval, Print, Loop". This is the interactive Python prompt, useful for debugging or testing short snippets of code. Most MicroPython boards make a REPL available over a UART, and this is typically accessible on a host PC via USB.

stream Also known as a "file-like object". An Python object which provides sequential read-write access to the underlying data. A stream object implements a corresponding interface, which consists of methods like `read()`, `write()`, `readinto()`, `seek()`, `flush()`, `close()`, etc. A stream is an important concept in MicroPython; many I/O objects implement the stream interface, and thus can be used consistently and interchangeably in different contexts. For more information on streams in MicroPython, see the *io* module.

UART Acronym for "Universal Asynchronous Receiver/Transmitter". This is a peripheral that sends data over a pair of pins (TX & RX). Many boards include a way to make at least one of the UARTs available to a host PC as a serial port over USB.

1.8.11 CircuitPython



circuitpython.org | *Get CircuitPython* | *Documentation* | *Contributing* | *Branding* | *Differences from Micropython* | *Project Structure*

CircuitPython is a *beginner friendly*, open source version of Python for tiny, inexpensive computers called microcontrollers. Microcontrollers are the brains of many electronics including a wide variety of development boards used to build hobby projects and prototypes. CircuitPython in electronics is one of the best ways to learn to code because it connects code to reality. Simply install CircuitPython on a supported board via drag and drop and then edit a `code.py` file on the CIRCUITPY drive. The code will automatically reload. No software installs are needed besides a text editor (we recommend Mu for beginners.)

CircuitPython features unified Python core APIs and a growing list of 150+ device libraries and drivers that work with it. These libraries also work on single board computers with regular Python via the Adafruit Blinka Library.

CircuitPython is based on MicroPython. See *below* for differences. CircuitPython development is sponsored by Adafruit and is available on their educational development boards. Please support both MicroPython and Adafruit.

Get CircuitPython

Official binaries for all supported boards are available through circuitpython.org/downloads. The site includes stable, unstable and continuous builds. Full release notes and assets are available through GitHub releases as well.

Documentation

Guides and videos are available through the Adafruit Learning System under the CircuitPython 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

Code Search

GitHub doesn't currently support code search on forks. Therefore, CircuitPython doesn't have code search through GitHub because it is a fork of MicroPython. Luckily, SourceGraph has free code search for public repos like CircuitPython. So, visit sourcegraph.com/github.com/adafruit/circuitpython to search the CircuitPython codebase online.

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.

Branding

While we are happy to see CircuitPython forked and modified, we'd appreciate it if forked releases not use the name "CircuitPython" or the Blinka logo. "CircuitPython" means something special to us and those who learn about it. As a result, we'd like to make sure products referring to it meet a common set of requirements.

If you'd like to use the term "CircuitPython" and Blinka for your product here is what we ask:

- Your product is supported by the primary "adafruit/circuitpython" repo. This way we can update any custom code as we update the CircuitPython internals.
- Your product is listed on circuitpython.org (source here). This is to ensure that a user of your product can always download the latest version of CircuitPython from the standard place.
- Your product has a user accessible USB plug which appears as a CIRCUITPY drive when plugged in.

If you choose not to meet these requirements, then we ask you call your version of CircuitPython something else (for example, SuperDuperPython) and not use the Blinka logo. You can say it is "CircuitPython-compatible" if most CircuitPython drivers will work with it.

Differences from MicroPython

CircuitPython:

- Supports native USB on all boards, allowing file editing without special tools.
- Floats (aka decimals) are enabled for all builds.
- Error messages are translated into 10+ languages.
- Does not support concurrency within Python (including interrupts and threading). Some concurrency is achieved with native modules for tasks that require it such as audio file playback.

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, as the REPL is a fresh vm.** CircuitPython's goal for this change includes reducing confusion about pins and memory being used.
- After the main code is finished the REPL can be entered by pressing any key.
- 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 it's fixed to get back into normal mode.
- RGB status LED indicating CircuitPython state, and errors through a sequence of colored flashes.
- Re-runs `code.py` or other main file after file system writes over USB mass storage. (Disable with `supervisor.disable_autoreload()`)
- Autoreload is disabled while the REPL is active.
- 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`

API

- Unified hardware APIs. Documented on ReadTheDocs.
- API docs are rST within the C files in `shared-bindings`.
- No `machine` API.

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()

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 CircuitPython 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.

Supported	Support status
atmel-samd	`SAMD21` stable | `SAMD51` stable
cxd56	stable
esp32s2	stable
litex	alpha
mimxrt10xx	alpha
nrf	stable
raspberrypi	stable
stm	`F4` stable | `others` beta
unix	alpha

- `stable` Highly unlikely to have bugs or missing functionality.
- `beta` Being actively improved but may be missing functionality and have bugs.
- `alpha` Will have bugs and missing functionality.

Boards

- Each `port` has a `boards` directory containing variations of boards which belong to a specific microcontroller line.
- A list of native modules supported by a particular board can be found here.

Back to Top

1.8.12 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.

Ways to contribute

As CircuitPython grows, there are more and more ways to contribute. Here are some ideas:

- Build a project with CircuitPython and share how to do it online.
- Test the latest libraries and CircuitPython versions with your projects and file issues for any bugs you find.
- Contribute Python code to CircuitPython libraries that support new devices or features of an existing device.
- Contribute C code to CircuitPython which fixes an open issue or adds a new feature.

Getting started with C

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.

1.8.13 Building CircuitPython

Welcome to CircuitPython!

This document is a quick-start guide only.

Detailed guides on how to build CircuitPython can be found in the Adafruit Learn system at https://learn.adafruit.com/building-circuitpython/

Setup

Please ensure you setup your build environment appropriately, as per the guide. You will need:

- Linux: https://learn.adafruit.com/building-circuitpython/linux
- MacOS: https://learn.adafruit.com/building-circuitpython/macos
- Windows Subsystem for Linux (WSL): https://learn.adafruit.com/building-circuitpython/windows-subsystem-for-linux

Submodules

This project has a bunch of git submodules. You will need to update them regularly.

```
git submodule sync
git submodule update --init
```

mpy-cross

As part of the build process, mpy-cross is needed to compile .py files into .mpy files. To compile (or recompile) mpy-cross:

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

1.8.14 Building

There a number of ports of CircuitPython! To build for your board, change to the appropriate ports directory and build.

Examples:

```
cd ports/atmel-samd
make BOARD=circuitplayground_express

cd ports/nrf
make BOARD=circuitplayground_bluefruit
```

If you aren't sure what boards exist, have a peek in the boards subdirectory of your port. If you have a fast computer with many cores, consider adding `-j` to your build flags, such as `-j17` on a 6-core 12-thread machine.

1.8.15 Testing

If you are working on changes to the core language, you might find it useful to run the test suite. The test suite in the top level `tests` directory. It needs the unix port to run.

```
cd ports/unix
make axtls
make micropython
```

Then you can run the test suite:

```
cd ../../tests
./run-tests
```

A successful run will say something like

```
676 tests performed (19129 individual testcases)
676 tests passed
30 tests skipped: buffered_writer builtin_help builtin_range_binop class_delattr_
↪setattr cmd_parsetree extra_coverage framebuf1 framebuf16 framebuf2 framebuf4␣
↪framebuf8 framebuf_subclass mpy_invalid namedtuple_asdict non_compliant resource_
↪stream schedule sys_getsizeof urandom_extra ure_groups ure_span ure_sub ure_sub_
↪unmatched vfs_basic vfs_fat_fileio1 vfs_fat_fileio2 vfs_fat_more vfs_fat_oldproto␣
↪vfs_fat_ramdisk vfs_userfs
```

1.8.16 Debugging

The easiest way to debug CircuitPython on hardware is with a JLink device, JLinkGDBServer, and an appropriate GDB. Instructions can be found at https://learn.adafruit.com/debugging-the-samd21-with-gdb

If using JLink, you'll need both the `JLinkGDBServer` and `arm-none-eabi-gdb` running.

Example:

```
JLinkGDBServer -if SWD -device ATSAMD51J19
arm-none-eabi-gdb build-metro_m4_express/firmware.elf -iex "target extended-remote␣
↪:2331"
```

If your port/build includes `arm-none-eabi-gdb-py`, consider using it instead, as it can be used for better register debugging with https://github.com/bnahill/PyCortexMDebug

1.8.17 Code Quality Checks

We apply code quality checks using pre-commit. Install pre-commit once per system with

```
python3 -mpip install pre-commit
```

Activate it once per git clone with

```
pre-commit --install
```

Pre-commit also requires some additional programs to be installed through your package manager:

- Standard Unix tools such as make, find, etc
- The gettext package, any modern version
- uncrustify version 0.71 (0.72 is also tested)

Each time you create a git commit, the pre-commit quality checks will be run. You can also run them e.g., with `pre-commit run foo.c` or `pre-commit run --all` to run on all files whether modified or not.

Some pre-commit quality checks require your active attention to resolve, others (such as the formatting checks of uncrustify) are made automatically and must simply be incorporated into your code changes by committing them.

1.8.18 Adafruit Community Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and leaders 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 or type of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

We are committed to providing a friendly, safe and welcoming environment for all.

Examples of behavior that contributes to creating a positive environment include:

- Be kind and courteous to others
- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Collaborating with other community members
- 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 sexual attention or advances
- The use of inappropriate images, including in a community member's avatar
- The use of inappropriate language, including in a community member's nickname
- Any spamming, flaming, baiting or other attention-stealing behavior
- Excessive or unwelcome helping; answering outside the scope of the question asked
- Trolling, insulting/derogatory comments, and personal or political attacks
- Promoting or spreading disinformation, lies, or conspiracy theories against a person, group, organisation, project, or community
- 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

The goal of the standards and moderation guidelines outlined here is to build and maintain a respectful community. We ask that you don't just aim to be "technically unimpeachable", but rather try to be your best self.

We value many things beyond technical expertise, including collaboration and supporting others within our community. Providing a positive experience for other community members can have a much more significant impact than simply providing the correct answer.

Our Responsibilities

Project leaders 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 leaders have the right and responsibility to remove, edit, or reject messages, comments, commits, code, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any community member for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Moderation

Instances of behaviors that violate the Adafruit Community Code of Conduct may be reported by any member of the community. Community members are encouraged to report these situations, including situations they witness involving other community members.

You may report in the following ways:

In any situation, you may send an email to mailto:support@adafruit.com.

On the Adafruit Discord, you may send an open message from any channel to all Community Moderators by tagging @community moderators. You may also send an open message from any channel, or a direct message to @kattni#1507, @tannewt#4653, @danh#1614, @cater#2442, @sommersoft#0222, @Mr. Certainly#0472 or @Andon#8175.

Email and direct message reports will be kept confidential.

In situations on Discord where the issue is particularly egregious, possibly illegal, requires immediate action, or violates the Discord terms of service, you should also report the message directly to Discord.

These are the steps for upholding our community's standards of conduct.

1. Any member of the community may report any situation that violates the Adafruit Community Code of Conduct. All reports will be reviewed and investigated.
2. If the behavior is an egregious violation, the community member who committed the violation may be banned immediately, without warning.
3. Otherwise, moderators will first respond to such behavior with a warning.
4. Moderators follow a soft "three strikes" policy - the community member may be given another chance, if they are receptive to the warning and change their behavior.
5. If the community member is unreceptive or unreasonable when warned by a moderator, or the warning goes unheeded, they may be banned for a first or second offense. Repeated offenses will result in the community member being banned.

Scope

This Code of Conduct and the enforcement policies listed above apply to all Adafruit Community venues. This includes but is not limited to any community spaces (both public and private), the entire Adafruit Discord server, and Adafruit GitHub repositories. Examples of Adafruit Community spaces include but are not limited to meet-ups, audio chats on the Adafruit Discord, or interaction at a conference.

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. As a community member, you are representing our community, and are expected to behave accordingly.

Attribution

This Code of Conduct is adapted from the [Contributor Covenant][homepage], version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html, and the Rust Code of Conduct.

For other projects adopting the Adafruit Community Code of Conduct, please contact the maintainers of those projects for enforcement. If you wish to use this code of conduct for your own project, consider explicitly mentioning your moderation policy or making a copy with your own moderation policy so as to avoid confusion.

1.8.19 MicroPython & CircuitPython license information

The MIT License (MIT)

Copyright (c) 2013-2017 Damien P. George, and others

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.8.20 WebUSB Serial Support

To date, this has only been tested on one port (esp32s2), on one board (espressif_kaluga_1).

What it does

If you have ever used CircuitPython on a platform with a graphical LCD display, you have probably already seen multiple "consoles" in use (although the LCD console is "output only").

New compile-time option CIRCUITPY_USB_VENDOR enables an additional "console" that can be used in parallel with the original (CDC) serial console.

Web pages that support the WebUSB standard can connect to the "vendor" interface and activate this WebUSB serial console at any time.

You can type into either console, and CircuitPython output is sent to all active consoles.

One example of a web page you can use to test drive this feature can be found at:

https://adafruit.github.io/Adafruit_TinyUSB_Arduino/examples/webusb-serial/index.html

How to enable

Update your platform's mpconfigboard.mk file to enable and disable specific types of USB interfaces.

CIRCUITPY_USB_HID = xxx CIRCUITPY_USB_MIDI = xxx CIRCUITPY_USB_VENDOR = xxx

On at least some of the hardware platforms, the maximum number of USB endpoints is fixed. For example, on the ESP32S2, you must pick only one of the above 3 interfaces to be enabled.

Original espressif_kaluga_1 mpconfigboard.mk settings:

CIRCUITPY_USB_HID = 1 CIRCUITPY_USB_MIDI = 0 CIRCUITPY_USB_VENDOR = 0

Settings to enable WebUSB instead:

CIRCUITPY_USB_HID = 0 CIRCUITPY_USB_MIDI = 0 CIRCUITPY_USB_VENDOR = 1

Notice that to enable VENDOR on ESP32-S2, we had to give up HID. There may be platforms that can have both, or even all three.

Implementation Notes

CircuitPython uses the tinyusb library.

The tinyusb library already has support for WebUSB serial. The tinyusb examples already include a "WebUSB serial" example.

```
Sidenote - The use of the term "vendor" instead of "WebUSB" was done to match tinyusb.
```

Basically, this feature was ported into CircuitPython by pulling code snippets out of the tinyusb example, and putting them where they best belonged in the CircuitPython codebase.

TODO: This needs to be reworked for dynamic USB descriptors.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z