CircuitPython Documentation

Release 9.0.4

CircuitPython Contributors

Apr 24, 2024

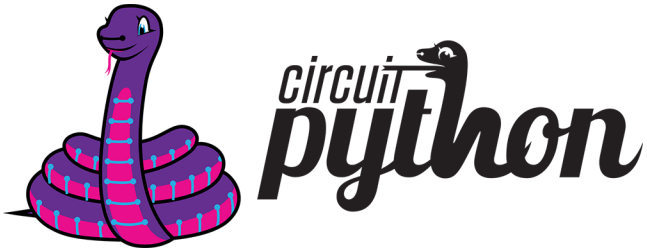
CONTENTS

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 USB board usually 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.)

Starting with CircuitPython 7.0.0, some boards may only be connectable over Bluetooth Low Energy (BLE). Those boards provide serial and file access over BLE instead of USB using open protocols. (Some boards may use both USB and BLE.) BLE access can be done from a variety of apps including code.circuitpython.org.

CircuitPython features unified Python core APIs and a growing list of 300+ 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. Most, but not all, 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 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 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.

1.4 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 supports at least one standard "Workflow" for serial and file access:
  - With a user accessible USB plug which appears as a CIRCUITPY drive when plugged in.
  - With file and serial access over Bluetooth Low Energy using the BLE Workflow.
  - With file access over WiFi using the WiFi Workflow with serial access over USB and/or WebSocket.
- Boards that do not support the USB Workflow should be clearly marked.

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.

1.5 Differences from MicroPython

CircuitPython:

- Supports native USB on most boards and BLE otherwise, allowing file editing without special tools.
- Floats (aka decimals) are enabled for all builds.
- Error messages are translated into 10+ languages.
- Concurrency within Python is not well supported. Interrupts and threading are disabled. async/await keywords are available on some boards for cooperative multitasking. Some concurrency is achieved with native modules for tasks that require it such as audio file playback.

1.5.1 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` runs only once on start up before workflows are 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. - If the file `repl.py` exists, it is executed before the REPL Prompt is shown - In safe mode this functionality is disabled, to ensure the REPL Prompt can always be reached
  - Autoreload state will be maintained across reload.
- Adds a safe mode that does not run user code after a hard crash or brown out. This makes it possible 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.
- Safe mode may be handled programmatically by providing a `safemode.py`. `safemode.py` is run if the board has reset due to entering safe mode, unless the safe mode initiated by the user by pressing button(s). USB is not available so nothing can be printed. `safemode.py` can determine why the safe mode occurred using `supervisor.runtime.safe_mode_reason`, and take appropriate action. For instance, if a hard crash occurred, `safemode.py` may do a `microcontroller.reset()` to automatically restart despite the crash. If the battery is low, but is being charged, `safemode.py` may put the board in deep sleep for a while. Or it may simply reset, and have `code.py` check the voltage and do the sleep.
- RGB status LED indicating CircuitPython state. - One green flash - code completed without error. - Two red flashes - code ended due to an exception. - Three yellow flashes - safe mode. May be due to CircuitPython internal error.
- Re-runs `code.py` or other main file after file system writes by a workflow. (Disable with `supervisor.disable_autoreload()`)
- Autoreload is disabled while the REPL is active.
- `code.py` may also be named `code.txt`, `main.py`, or `main.txt`.
- `boot.py` may also be named `boot.txt`.
- `safemode.py` may also be named `safemode.txt`.

1.5.2 API

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

1.5.3 Modules

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

1.6 Project Structure

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

1.6.1 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.

1.6.2 Ports

Ports include the code unique to a microcontroller line.

Supported	Support status
atmel-samd	`SAMD21` stable | `SAMD51` stable
cxd56	stable
espressif	`ESP32` beta | `ESP32-C3` beta | `ESP32-S2` stable | `ESP32-S3` beta
litex	alpha
mimxrt10xx	alpha
nrf	stable
raspberrypi	stable
silabs (efr32)	alpha
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.

1.6.3 Boards

- Each `port` has a `boards` directory containing 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

CHAPTER

TWO

ADAFRUIT CIRCUITPYTHON LIBRARIES

Documentation for all Adafruit-sponsored CircuitPython libraries is at: <https://docs.circuitpython.org/projects/bundle/en/latest/drivers.html>.

CHAPTER

THREE

CIRCUITPYTHON LIBRARY BUNDLES

Many Python libraries, including device drivers, have been written for use with CircuitPython. They are maintained in separate GitHub repos, one per library.

Libraries are packaged in *bundles*, which are ZIP files that are snapshots in time of a group of libraries.

Adafruit sponsors and maintains several hundred libraries, packaged in the **Adafruit Library Bundle**. Adafruit-sponsored libraries are also available on <https://pypi.org>.

Yet other libraries are maintained by members of the CircuitPython community, and are packaged in the **CircuitPython Community Library Bundle**.

The Adafruit bundles are available on GitHub: <https://github.com/adafruit/Adafruit_CircuitPython_Bundle/releases>. The Community bundles are available at: <https://github.com/adafruit/CircuitPython_Community_Bundle/releases>.

More detailed information about the bundles, and download links for the latest bundles are at <https://circuitpython.org/libraries>.

Documentation about bundle construction is at: <https://circuitpython.readthedocs.io/projects/bundle/en/latest/>.

Documentation for Community Libraries is not available on ReadTheDocs at this time. See the GitHub repository for each library for any included documentation.

CHAPTER

FOUR

WORKFLOWS

Workflows are the process used to 1) manipulate files on the CircuitPython device and 2) interact with the serial connection to CircuitPython. The serial connection is usually used to access the REPL.

Starting with CircuitPython 3.x we moved to a USB-only workflow. Prior to that, we used the serial connection alone to do the whole workflow. In CircuitPython 7.x, a BLE workflow was added with the advantage of working with mobile devices. CircuitPython 8.x added a web workflow that works over the local network (usually Wi-Fi) and a web browser. Other clients can also use the Web REST API. Boards should clearly document which workflows are supported.

Code for workflows lives in `supervisor/shared`.

The workflow APIs are documented here.

4.1 USB

These USB interfaces are enabled by default on boards with USB support. They are usable once the device has been plugged into a host.

4.1.1 CIRCUITPY drive

CircuitPython exposes a standard mass storage (MSC) interface to enable file manipulation over a standard interface. This interface works underneath the file system at the block level so using it excludes other types of workflows from manipulating the file system at the same time.

4.1.2 CDC serial

CircuitPython exposes one CDC USB interface for CircuitPython serial. This is a standard serial USB interface.

TODO: Document how it designates itself from the user CDC.

Setting baudrate 1200 and disconnecting will reboot into a bootloader. (Used by Arduino to trigger a reset into bootloader.)

4.2 BLE

The BLE workflow is enabled for nRF boards. By default, to prevent malicious access, it is disabled. To connect to the BLE workflow, press the reset button while the status led blinks blue quickly after the safe mode blinks. The board will restart and broadcast the file transfer service UUID (`0xfebb`) along with the board's Creation IDs. This public broadcast is done at a lower transmit level so the devices must be closer. On connection, the device will need to pair and bond. Once bonded, the device will broadcast whenever disconnected using a rotating key rather than a static one. Non-bonded devices won't be able to resolve it. After connection, the central device can discover two default services. One for file transfer and one for CircuitPython specifically that includes serial characteristics.

To change the default BLE advertising name without (or before) running user code, the desired name can be put in the `settings.toml` file. The key is `CIRCUITPY_BLE_NAME`. It's limited to approximately 30 characters depending on the port's settings and will be truncated if longer.

4.2.1 File Transfer API

CircuitPython uses an open File Transfer API to enable file system access.

4.2.2 CircuitPython Service

The base UUID for the CircuitPython service is `ADAFXXXX-4369-7263-7569-7450794686e`. The `XXXX` is replaced by the four specific digits below. The service itself is `0001`.

TX - `0002` / RX - `0003`

These characteristic work just like the Nordic Uart Service (NUS) but have different UUIDs to prevent conflicts with user created NUS services.

Version - `0100`

Read-only characteristic that returns the UTF-8 encoded version string.

4.3 Web

If the keys `CIRCUITPY_WIFI_SSID` and `CIRCUITPY_WIFI_PASSWORD` are set in `settings.toml`, CircuitPython will automatically connect to the given Wi-Fi network on boot and upon reload.

If `CIRCUITPY_WEB_API_PASSWORD` is set, MDNS and the http server for the web workflow will also start.

The webserver is on port 80 unless overridden by `CIRCUITPY_WEB_API_PORT`. It also enables MDNS. The name of the board as advertised to the network can be overridden by `CIRCUITPY_WEB_INSTANCE_NAME`.

Here is an example `/settings.toml`:

```
# To auto-connect to Wi-Fi
CIRCUITPY_WIFI_SSID="scottswifi"
CIRCUITPY_WIFI_PASSWORD="secretpassword"

# To enable the web workflow. Change this too!
# Leave the User field blank in the browser.
```

(continues on next page)

(continued from previous page)

```
CIRCUITPY_WEB_API_PASSWORD="passw0rd"

CIRCUITPY_WEB_API_PORT=80
CIRCUITPY_WEB_INSTANCE_NAME=""
```

MDNS is used to resolve `circuitpython.local` to a device specific hostname of the form `cpy-XXXXXX.local`. The `XXXXXX` is based on network MAC address. The device also provides the MDNS service with service type `_circuitpython` and protocol `_tcp`.

Since port 80 (or the port assigned to `CIRCUITPY_WEB_API_PORT`) is used for web workflow, the `mdns` module can't advertise an additional service on that port.

4.3.1 HTTP

The web server is HTTP 1.1 and may use chunked responses so that it doesn't need to precompute content length.

The API generally consists of an HTTP method such as GET or PUT and a path. Requests and responses also have headers. Responses will contain a status code and status text such as `404 Not Found`. This API tries to use standard status codes to encode the status of the various operations. The Mozilla Developer Network HTTP docs are a great reference.

Examples

The examples use `curl`, a common command line program for issuing HTTP requests. The examples below use `circuitpython.local` as the easiest way to work. If you have multiple active devices, you'll want to use the specific `cpy-XXXXXX.local` version.

The examples also use `passw0rd` as the password placeholder. Replace it with your password before running the example.

4.3.2 /

The root welcome page links to the file system page and also displays other CircuitPython devices found using MDNS service discovery. This allows web browsers to find other devices from one. (All devices will respond to `circuitpython.local` so the device redirected to may vary.)

4.3.3 CORS

The web server will allow requests from `cpy-XXXXXX.local`, `127.0.0.1`, the device's IP and `code.circuitpython.org`. (`circuitpython.local` requests will be redirected to `cpy-XXXXXX.local`.)

4.3.4 File REST API

All file system related APIs are protected by HTTP basic authentication. It is *NOT* secure but will hopefully prevent some griefing in shared settings. The password is sent unencrypted so do not reuse a password with something important. The user field is left blank.

The password is taken from `settings.toml` with the key `CIRCUITPY_WEB_API_PASSWORD`. If this is unset, the server will respond with `403 Forbidden`. When a password is set, but not provided in a request, it will respond `401 Unauthorized`.

/fs/

The `/fs/` page will respond with a directory browsing HTML once authenticated. This page is always gzipped. If the `Accept: application/json` header is provided, then the JSON representation of the root will be returned.

OPTIONS

When requested with the `OPTIONS` method, the server will respond with CORS related headers. Most aren't needed for API use. They are there for the web browser.

- `Access-Control-Allow-Methods` - Varies with USB state. `GET, OPTIONS` when USB is active. `GET, OPTIONS, PUT, DELETE, MOVE` otherwise.

Example:

```
curl -v -u :passw0rd -X OPTIONS -L --location-trusted http://circuitpython.local/fs/
```

/fs/<directory path>/

Directory paths must end with a /. Otherwise, the path is assumed to be a file.

GET

Returns a JSON representation of the directory.

- `200 OK` - Directory exists and JSON returned
- `401 Unauthorized` - Incorrect password
- `403 Forbidden` - No `CIRCUITPY_WEB_API_PASSWORD` set
- `404 Not Found` - Missing directory

Returns directory information:

- `free`: Count of free blocks on the disk holding this directory.
- `total`: Total blocks that make up the disk holding this directory.
- `block_size`: Size of a block in bytes.
- `writable`: True when CircuitPython and the web workflow can write to the disk. USB may claim a disk instead.
- `files`: Array of objects. One for each file.

Returns information about each file in the directory:

- `name` - File name. No trailing `/` on directory names

- `directory` - `true` when a directory. `false` otherwise
- `modified_ns` - File modification time in nanoseconds since January 1st, 1970. May not use full resolution
- `file_size` - File size in bytes. `0` for directories

Example:

```
curl -v -u :passw0rd -H "Accept: application/json" -L --location-trusted http://
↪circuitpython.local/fs/lib/hello/
```

```
{
        "free": 451623,
        "total": 973344,
        "block_size": 32768,
        "writable": true,
        "files": [
                {
                        "name": "world.txt",
                        "directory": false,
                        "modified_ns": 946934328000000000,
                        "file_size": 12
                }
        ]
}
```

PUT

Tries to make a directory at the given path. Request body is ignored. The custom `X-Timestamp` header can provide a timestamp in milliseconds since January 1st, 1970 (to match JavaScript's file time resolution) used for the directories modification time. The RTC time will used otherwise.

Returns:

- `204 No Content` - Directory or file exists
- `201 Created` - Directory created
- `401 Unauthorized` - Incorrect password
- `403 Forbidden` - No `CIRCUITPY_WEB_API_PASSWORD` set
- `409 Conflict` - USB is active and preventing file system modification
- `404 Not Found` - Missing parent directory
- `500 Server Error` - Other, unhandled error

Example:

```
curl -v -u :passw0rd -X PUT -L --location-trusted http://circuitpython.local/fs/lib/
↪hello/world/
```

Move

Moves the directory at the given path to `X-Destination`. Also known as rename.

The custom `X-Destination` header stores the destination path of the directory.

- `201 Created` - Directory renamed
- `401 Unauthorized` - Incorrect password
- `403 Forbidden` - No `CIRCUITPY_WEB_API_PASSWORD` set
- `404 Not Found` - Source directory not found or destination path is missing
- `409 Conflict` - USB is active and preventing file system modification
- `412 Precondition Failed` - The destination path is already in use

Example:

```
curl -v -u :passw0rd -X MOVE -H "X-Destination: /fs/lib/hello2/" -L --location-trusted␣
↪http://circuitpython.local/fs/lib/hello/
```

DELETE

Deletes the directory and all of its contents.

- `204 No Content` - Directory and its contents deleted
- `401 Unauthorized` - Incorrect password
- `403 Forbidden` - No `CIRCUITPY_WEB_API_PASSWORD` set
- `404 Not Found` - No directory
- `409 Conflict` - USB is active and preventing file system modification

Example:

```
curl -v -u :passw0rd -X DELETE -L --location-trusted http://circuitpython.local/fs/lib/
↪hello2/world/
```

/fs/<file path>

PUT

Stores the provided content to the file path.

The custom `X-Timestamp` header can provide a timestamp in milliseconds since January 1st, 1970 (to match JavaScript's file time resolution) used for the directories modification time. The RTC time will used otherwise.

Returns:

- `201 Created` - File created and saved
- `204 No Content` - File existed and overwritten
- `401 Unauthorized` - Incorrect password
- `403 Forbidden` - No `CIRCUITPY_WEB_API_PASSWORD` set

- `404 Not Found` - Missing parent directory
- `409 Conflict` - USB is active and preventing file system modification
- `413 Payload Too Large` - `Expect` header not sent and file is too large
- `417 Expectation Failed` - `Expect` header sent and file is too large
- `500 Server Error` - Other, unhandled error

If the client sends the `Expect` header, the server will reply with `100 Continue` when ok.

Example:

```
echo "Hello world" >> test.txt
curl -v -u :passw0rd -T test.txt -L --location-trusted http://circuitpython.local/fs/lib/
↪hello/world.txt
```

GET

Returns the raw file contents. `Content-Type` will be set based on extension:

- `text/plain` - `.py`, `.txt`
- `text/javascript` - `.js`
- `text/html` - `.html`
- `application/json` - `.json`
- `application/octet-stream` - Everything else

Will return:

- `200 OK` - File exists and file returned
- `401 Unauthorized` - Incorrect password
- `403 Forbidden` - No `CIRCUITPY_WEB_API_PASSWORD` set
- `404 Not Found` - Missing file

Example:

```
curl -v -u :passw0rd -L --location-trusted http://circuitpython.local/fs/lib/hello/world.
↪txt
```

Move

Moves the file at the given path to the `X-Destination`. Also known as rename.

The custom `X-Destination` header stores the destination path of the file.

- `201 Created` - File renamed
- `401 Unauthorized` - Incorrect password
- `403 Forbidden` - No `CIRCUITPY_WEB_API_PASSWORD` set
- `404 Not Found` - Source file not found or destination path is missing
- `409 Conflict` - USB is active and preventing file system modification

- `412 Precondition Failed` - The destination path is already in use

Example:

```
curl -v -u :passw0rd -X MOVE -H "X-Destination: /fs/lib/hello/world2.txt" -L --location-
↪trusted http://circuitpython.local/fs/lib/hello/world.txt
```

DELETE

Deletes the file.

- `204 No Content` - File existed and deleted
- `401 Unauthorized` - Incorrect password
- `403 Forbidden` - No `CIRCUITPY_WEB_API_PASSWORD` set
- `404 Not Found` - File not found
- `409 Conflict` - USB is active and preventing file system modification

Example:

```
curl -v -u :passw0rd -X DELETE -L --location-trusted http://circuitpython.local/fs/lib/
↪hello/world2.txt
```

4.3.5 /cp/

`/cp/` serves basic info about the CircuitPython device and others discovered through MDNS. It is not protected by basic auth in case the device is someone elses.

Only `GET` requests are supported and will return `405 Method Not Allowed` otherwise.

/cp/devices.json

Returns information about other devices found on the network using MDNS.

- `total`: Total MDNS response count. May be more than in `devices` if internal limits were hit.
- `devices`: List of discovered devices.
  - `hostname`: MDNS hostname
  - `instance_name`: MDNS instance name. Defaults to human readable board name.
  - `port`: Port of CircuitPython Web API
  - `ip`: IP address

Example:

```
curl -v -L http://circuitpython.local/cp/devices.json
```

```
{
        "total": 1,
        "devices": [
                {
```

(continues on next page)

(continued from previous page)

```
                        "hostname": "cpy-951032",
                        "instance_name": "Adafruit Feather ESP32-S2 TFT",
                        "port": 80,
                        "ip": "192.168.1.235"
                }
        ]
}
```

/cp/diskinfo.json

Returns information about the attached disk(s). A list of objects, one per disk.

- `root`: Filesystem path to the root of the disk.
- `free`: Count of free blocks on the disk.
- `total`: Total blocks that make up the disk.
- `block_size`: Size of a block in bytes.
- `writable`: True when CircuitPython and the web workflow can write to the disk. USB may claim a disk instead.

Example:

```
curl -v -L http://circuitpython.local/cp/diskinfo.json
```

```
[{
        "root": "/",
        "free": 2964992,
        "block_size": 512,
        "writable": true,
        "total": 2967552
}]
```

/cp/serial/

Serves a basic serial terminal program when a `GET` request is received without the `Upgrade: websocket` header. Otherwise the socket is upgraded to a WebSocket. See WebSockets below for more detail.

This is an authenticated endpoint in both modes.

/cp/version.json

Returns information about the device.

- `web_api_version`: Between 1 and 4. This versions the rest of the API and new versions may not be backwards compatible. See below for more info.
- `version`: CircuitPython build version.
- `build_date`: CircuitPython build date.
- `board_name`: Human readable name of the board.
- `mcu_name`: Human readable name of the microcontroller.

- `board_id`: Board id used in code and on circuitpython.org.
- `creator_id`: Creator ID for the board.
- `creation_id`: Creation ID for the board, set by the creator.
- `hostname`: MDNS hostname.
- `port`: Port of CircuitPython Web Service.
- `ip`: IP address of the device.

Example:

```
curl -v -L http://circuitpython.local/cp/version.json
```


```
{
        "web_api_version": 1,
        "version": "8.0.0-alpha.1-20-ge1d4518a9-dirty",
        "build_date": "2022-06-24",
        "board_name": "ESP32-S3-USB-OTG-N8",
        "mcu_name": "ESP32S3",
        "board_id": "espressif_esp32s3_usb_otg_n8",
        "creator_id": 12346,
        "creation_id": 28683,
        "hostname": "cpy-f57ce8",
        "port": 80,
        "ip": "192.168.1.94"
}
```


`/code/`

The `/code/` page returns a small static html page that will pull in and load the full code editor from code.circuitpython.org for a full code editor experience. Because most of the resources reside online instead of the device, an active internet connection is required.

4.3.6 Static files

- `/favicon.ico` - Blinka
- `/directory.js` - JavaScript for `/fs/`
- `/welcome.js` - JavaScript for `/`

4.3.7 WebSocket

The CircuitPython serial interactions are available over a WebSocket. A WebSocket begins as a special HTTP request that gets upgraded to a WebSocket. Authentication happens before upgrading.

WebSockets are *not* bare sockets once upgraded. Instead they have their own framing format for data. CircuitPython can handle PING and CLOSE opcodes. All others are treated as TEXT. Data to CircuitPython is expected to be masked UTF-8, as the spec requires. Data from CircuitPython to the client is unmasked. It is also unbuffered so the client will get a variety of frame sizes.

Only one WebSocket at a time is supported.

4.3.8 Versions

- `1` - Initial version.
- `2` - Added `/cp/diskinfo.json`.
- `3` - Changed `/cp/diskinfo.json` to return a list in preparation for multi-disk support.
- `4` - Changed directory json to an object with additional data. File list is under `files` and is the same as the old format.

CHAPTER

FIVE

ENVIRONMENT VARIABLES

CircuitPython 8.0.0 introduces support for environment variables. Environment variables are commonly used to store "secrets" such as Wi-Fi passwords and API keys. This method *does not* make them secure. It only separates them from the code.

CircuitPython uses a file called `settings.toml` at the drive root (no folder) as the environment. User code can access the values from the file using *os.getenv()*. It is recommended to save any values used repeatedly in a variable because *os.getenv()* will parse the `settings.toml` file contents on every access.

CircuitPython only supports a subset of the full toml specification, see below for more details. The subset is very "Python-like", which is a key reason we selected the format.

Due to technical limitations it probably also accepts some files that are not valid TOML files; bugs of this nature are subject to change (i.e., be fixed) without the usual deprecation period for incompatible changes.

File format example:

```
str_key="Hello world" # with trailing comment
int_key = 7
unicode_key="œuvre"
unicode_key2="\\u0153uvre" # same as above
unicode_key3="\\U00000153uvre" # same as above
escape_codes="supported, including \\r\\n\\"\\\\"
# comment
[subtable]
subvalue="cannot retrieve this using getenv"
```

5.1 Details of the toml language subset

- The content is required to be in UTF-8 encoding
- The supported data types are string and integer
- Only basic strings are supported, not triple-quoted strings
- Only integers supported by strtol. (no 0o, no 0b, no underscores 1_000, 011 is 9, not 11)
- Only bare keys are supported
- Duplicate keys are not diagnosed.
- Comments are supported
- Only values from the "root table" can be retrieved
- due to technical limitations, the content of multi-line strings can erroneously be parsed as a value.

5.2 CircuitPython behavior

CircuitPython will also read the environment to configure its behavior. Other keys are ignored by CircuitPython. Here are the keys it uses:

5.2.1 CIRCUITPY_BLE_NAME

Default BLE name the board advertises as, including for the BLE workflow.

5.2.2 CIRCUITPY_HEAP_START_SIZE

Sets the initial size of the python heap, allocated from the outer heap. Must be a multiple of 4. The default is currently 8192. The python heap will grow by doubling and redoubling this initial size until it cannot fit in the outer heap. Larger values will reserve more RAM for python use and prevent the supervisor and SDK from large allocations of their own. Smaller values will likely grow sooner than large start sizes.

5.2.3 CIRCUITPY_PYSTACK_SIZE

Sets the size of the python stack. Must be a multiple of 4. The default value is currently 1536. Increasing the stack reduces the size of the heap available to python code. Used to avoid "Pystack exhausted" errors when the code can't be reworked to avoid it.

5.2.4 CIRCUITPY_WEB_API_PASSWORD

Password required to make modifications to the board from the Web Workflow.

5.2.5 CIRCUITPY_WEB_API_PORT

TCP port number used for the web HTTP API. Defaults to 80 when omitted.

5.2.6 CIRCUITPY_WEB_INSTANCE_NAME

Name the board advertises as for the WEB workflow. Defaults to human readable board name if omitted.

5.2.7 CIRCUITPY_WIFI_PASSWORD

Wi-Fi password used to auto connect to CIRCUITPY_WIFI_SSID.

5.2.8 CIRCUITPY_WIFI_SSID

Wi-Fi SSID to auto-connect to even if user code is not running.

CHAPTER

SIX

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.

6.1 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.

Corruption 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. Back up your code if possible before continuing!**

6.1.1 REPL Erase Method

This is the recommended method of erasing your board. If you are having trouble accessing the `CIRCUITPY` drive or the REPL, consider first putting your board into safe mode.

To erase any board if you have access to the REPL:

1. Connect to the CircuitPython REPL using a terminal program.
2. Type `import storage` into the REPL.
3. Then, type `storage.erase_filesystem()` into the REPL.
4. The `CIRCUITPY` drive will be erased and the board will restart with an empty `CIRCUITPY` drive.

6.1.2 Erase File Method

If you do not have access to the REPL, you may still have options to erase your board.

The Erase CIRCUITPY Without Access to the REPL section of the Troubleshooting page in the Welcome to CircuitPython guide covers the non-REPL erase process for various boards. Visit the guide, find the process that applies to your board, and follow the instructions to erase your board.

6.2 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.

CHAPTER

SEVEN

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.

7.1 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.

7.2 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.

7.3 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).

7.4 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 Jeff Epler (@jepler) are also sponsored by Adafruit Industries LLC and are usually available during US daytime hours including some weekends.

They are all reachable on Discord, GitHub issues and the Adafruit support forum.

7.5 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.

CHAPTER

EIGHT

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/

8.1 Setup

Please ensure you set up 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

8.1.1 Submodules

This project has a bunch of git submodules. You will need to update them regularly.

In the root folder of the CircuitPython repository, execute the following:

```
make fetch-all-submodules
```

Or, in the ports directory for the particular port you are building, do:

```
make fetch-port-submodules
```

8.1.2 Required Python Packages

Failing to install these will prevent from properly building.

```
pip3 install -r requirements-dev.txt
```

If you run into an error installing minify_html, you may need to install `rust`.

8.1.3 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
```

8.2 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.

8.3 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.py
```

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
```

8.4 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

8.5 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 and OK; 0.75 is not OK)

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.

CHAPTER

NINE

WEBUSB SERIAL SUPPORT

To date, this has only been tested on one port (espressif), on one board (espressif_kaluga_1).

9.1 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

9.2 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.

9.3 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.

9.3.1 TODO: This needs to be reworked for dynamic USB descriptors.

CHAPTER

TEN

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.

10.1 SAMD21 and SAMD51

This port supports many development boards that utilize SAMD21 and SAMD51 chips. See https://circuitpython.org/downloads for all supported boards.

10.1.1 Building

For build instructions see this guide: https://learn.adafruit.com/building-circuitpython/

10.1.2 Debugging

For debugging instructions see this guide: https://learn.adafruit.com/debugging-the-samd21-with-gdb

10.1.3 Port Specific modules

`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: *Clock* | None

Clock parent. (read-only)

frequency: *int*

Clock frequency in Herz. (read-only)

calibration: *int*

Clock calibration. Not all clocks can be calibrated.

10.2 Broadcom

This port supports running CircuitPython bare-metal on Raspberry Pi single board computers that utilize Broadcom system-on-chips.

10.3 CXD56 (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.

10.3.1 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

10.3.2 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
```

10.3.3 USB connection

Connect the `Spresense main board` to the PC via the USB cable.

10.3.4 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 v3-0-0

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
```

10.3.5 Flash the circuitpython image

To flash the firmware run the command:

```
$ make BOARD=spresense flash
```

10.3.6 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
```

10.4 Espressif

This port adds the Espressif line of SoCs to CircuitPython.

10.4.1 Support Status:

SoC	Status
ESP32	beta
ESP32-H2	alpha
ESP32-C3	beta
ESP32-C6	alpha
ESP32-S2	stable
ESP32-S3	stable

10.4.2 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.
- **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.

10.4.3 Connecting to the ESP32

The ESP32 chip itself has no USB support. On many boards there is a USB-serial adapter chip, such as a CP2102N, CP2104 or CH9102F, usually connected to the ESP32 TXD0 (GPIO1)and RXD0 (GPIO3) pins, for access to the bootloader. CircuitPython also uses this serial channel for the REPL.

10.4.4 Connecting to the ESP32-C3

USB Connection:

On ESP32-C3 REV3 chips, a USB Serial/JTAG Controller is available. Note: This USB connection cannot be used for a `CIRCUITPY` drive.

Depending on the board you have, the USB port may or may not be connected to native USB.

The following connections need to be made if native USB isn't available on the USB port:

GPIO	USB
19	D+ (green)
18	D- (white)
GND	GND (black)
5V	5V (red)

Connect these pins using a USB adapter or breakout cable.

UART Connection:

A USB to UART converter can be used for connecting to ESP32-C3 to get access to the serial console and REPL and for flashing CircuitPython.

The following connections need to be made in this case:

GPIO	UART
21	RX
20	TX
GND	GND
5V	5V

BLE Connection:

This feature is not yet available and currently under development.

10.4.5 Connecting to the ESP32-S2

USB Connection:

Depending on the board you have, the USB port may or may not be connected to native USB.

The following connections need to be made if native USB isn't available on the USB port:

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.

UART Connection:

A USB to UART converter can be used for connecting to ESP32-S2 to get access to the serial console and REPL and for flashing CircuitPython.

The following connections need to be made in this case:

GPIO	UART
43	RX
44	TX
GND	GND
5V	5V

BLE Connection:

This feature isn't available on ESP32-S2.

10.4.6 Connecting to the ESP32-S3

USB Connection:

Depending on the board you have, the USB port may or may not be connected to native USB.

The following connections need to be made if native USB isn't available on the USB port:

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.

UART Connection:

A USB to UART converter can be used for connecting to ESP32-S3 to get access to the serial console and REPL and for flashing CircuitPython.

The following connections need to be made in this case:

GPIO	UART
43	RX
44	TX
GND	GND
5V	5V

BLE Connection:

This feature is not yet available and currently under development.

10.4.7 Building and flashing

Before building or flashing the, you must install the ESP-IDF.

Note: This must be re-done every time the ESP-IDF is updated, but not every time you build.

Run `cd ports/espressif` from `circuitpython/` to move to the espressif port root, and run:

```
./esp-idf/install.sh
```

After this initial installation, you must add the ESP-IDF tools to your path.

Note: This must be re-done every time you open a new shell environment for building or flashing.

Run `cd ports/espressif` from `circuitpython/` to move to the espressif port root, and run:

```
source ./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 shell environment.

Building boards is typically done through `make BOARD=board_id`. 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=board_id PORT=/dev/tty.usbserial-1421120 flash
```

`board_id` is the unique board identifier in CircuitPython. It is the same as the name of the board in the `boards` directory.

10.4.8 Debugging

TODO: Add documentation for ESP32-C3/S3 JTAG feature.

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 location of these files by checking the `OPENOCD_SCRIPTS`

environmental variable by running `echo $OPENOCD_SCRIPTS`. 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
```

10.5 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.

10.5.1 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.

10.6 NXP i.MX RT10xx Series

This is a port of CircuitPython to the i.MX RT10xx series of chips.

10.7 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.

10.7.1 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
```

10.7.2 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.

10.7.3 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)

10.8 RP2040

This port supports many development boards that utilize RP2040 chips. See https://circuitpython.org/downloads for all supported boards.

10.8.1 Building

For build instructions see this guide: https://learn.adafruit.com/building-circuitpython/

10.8.2 Port Specific modules

cyw43 – A class that represents a GPIO pin attached to the wifi chip.

class cyw43.**CywPin**

Cannot be constructed at runtime, but may be the type of a pin object in *board*. A *CywPin* can be used as a DigitalInOut, but not with other peripherals such as *PWMOut*.

cyw43.**PM_STANDARD: int**

The standard power management mode

cyw43.**PM_AGGRESSIVE: int**

Aggressive power management mode for optimal power usage at the cost of performance

cyw43.**PM_PERFORMANCE: int**

Performance power management mode where more power is used to increase performance

cyw43.**PM_DISABLED: int**

Disable power management and always use highest power mode. CircuitPython sets this value at reset time, because it provides the best connectivity reliability.

cyw43.**set_power_management**(*value: int*) → None

Set the power management register

For transmitter power, see `wifi.Radio.txpower`. This controls software power saving features inside the cyw43 chip. it does not control transmitter power.

The value is interpreted as a 24-bit hexadecimal number of the form `0x00adbrrm`.

The low 4 bits, `m`, are the power management mode:

- 0: disabled
- 1: aggressive power saving which reduces wifi throughput
- 2: Power saving with high throughput

The next 8 bits, `r`, specify "the maximum time to wait before going back to sleep" for power management mode 2. The units of `r` are 10ms.

The next 4 bits, `b`, are the "wake period is measured in beacon periods".

The next 4 bits, `d`, specify the "wake interval measured in DTIMs. If this is set to 0, the wake interval is measured in beacon periods".

The top 4 bits, `a`, specifies the "wake interval sent to the access point"

Several PM_ constants gathered from various sources are included in this module. According to Raspberry Pi documentation, the value 0xa11140 (called *cyw43.PM_DISABLED* here) increases responsiveness at the cost of higher power usage.

cyw43.**get_power_management**() → int

Retrieve the power management register

picodvi – Low-level routines for interacting with PicoDVI Output

class picodvi.**Framebuffer**(*width: int*, *height: int*, *, *clk_dp:* microcontroller.Pin, *clk_dn:* microcontroller.Pin, *red_dp:* microcontroller.Pin, *red_dn:* microcontroller.Pin, *green_dp:* microcontroller.Pin, *green_dn:* microcontroller.Pin, *blue_dp:* microcontroller.Pin, *blue_dn:* microcontroller.Pin, *color_depth: int = 8*)

Create a Framebuffer object with the given dimensions. Memory is allocated outside of onto the heap and then moved outside on VM end.

> **Warning:** This will change the system clock speed to match the DVI signal. Make sure to initialize other objects after this one so they account for the changed clock.

This allocates a very large framebuffer and is most likely to succeed the earlier it is attempted.

Each dp and dn pair of pins must be neighboring, such as 19 and 20. They must also be ordered the same way. In other words, dp must be less than dn for all pairs or dp must be greater than dn for all pairs.

The framebuffer pixel format varies depending on color_depth:

- 1 - Each bit is a pixel. Either white (1) or black (0).
- 2 - Each 2 bits is a pixels. Grayscale between white (0x3) and black (0x0).
- 8 - Each byte is a pixels in RGB332 format.
- 16 - Each two bytes are a pixel in RGB565 format.

Two output resolutions are currently supported, 640x480 and 800x480. Monochrome framebuffers (color_depth=1 or 2) must be full resolution. Color framebuffers must be half resolution (320x240 or 400x240) and pixels will be duplicated to create the signal.

A Framebuffer is often used in conjunction with a *framebufferio.FramebufferDisplay*.

Parameters

- **width** (*int*) – the width of the target display signal. Only 320, 400, 640 or 800 is currently supported depending on color_depth.
- **height** (*int*) – the height of the target display signal. Only 240 or 480 is currently supported depending on color_depth.
- **clk_dp** (Pin) – the positive clock signal pin
- **clk_dn** (Pin) – the negative clock signal pin
- **red_dp** (Pin) – the positive red signal pin
- **red_dn** (Pin) – the negative red signal pin
- **green_dp** (Pin) – the positive green signal pin
- **green_dn** (Pin) – the negative green signal pin
- **blue_dp** (Pin) – the positive blue signal pin

- **blue_dn** (Pin) – the negative blue signal pin
- **color_depth** (*int*) – the color depth of the framebuffer in bits. 1, 2 for grayscale and 8 or 16 for color

width: int

The width of the framebuffer, in pixels. It may be doubled for output.

height: int

The width of the framebuffer, in pixels. It may be doubled for output.

deinit() → None

Free the resources (pins, timers, etc.) associated with this *picodvi.Framebuffer* instance. After deinitialization, no further operations may be performed.

rp2pio – Hardware interface to RP2 series' programmable IO (PIO) peripheral.

Note: This module is intended to be used with the adafruit_pioasm library. For an introduction and guide to working with PIO in CircuitPython, see this Learn guide.

rp2pio.**pins_are_sequential**(*pins: List[*microcontroller.Pin*]*) → bool

Return True if the pins have sequential GPIO numbers, False otherwise

class rp2pio.**StateMachine**(*program: circuitpython_typing.ReadableBuffer*, *frequency: int*, **, may_exec: circuitpython_typing.ReadableBuffer | None = None*, *init: circuitpython_typing.ReadableBuffer | None = None*, *first_out_pin:* microcontroller.Pin *| None = None*, *out_pin_count: int = 1*, *initial_out_pin_state: int = 0*, *initial_out_pin_direction: int = 4294967295*, *first_in_pin:* microcontroller.Pin *| None = None*, *in_pin_count: int = 1*, *pull_in_pin_up: int = 0*, *pull_in_pin_down: int = 0*, *first_set_pin:* microcontroller.Pin *| None = None*, *set_pin_count: int = 1*, *initial_set_pin_state: int = 0*, *initial_set_pin_direction: int = 31*, *first_sideset_pin:* microcontroller.Pin *| None = None*, *sideset_pin_count: int = 1*, *initial_sideset_pin_state: int = 0*, *initial_sideset_pin_direction: int = 31*, *sideset_enable: bool = False*, *jmp_pin:* microcontroller.Pin *| None = None*, *jmp_pin_pull:* digitalio.Pull *| None = None*, *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*, *user_interruptible: bool = True*, *wrap_target: int = 0*, *wrap: int = -1*, *offset: int = -1*)

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. Use 0 for system clock speed.
- **init** (*ReadableBuffer*) – a program to run once at start up. This is run after program is started so instructions may be intermingled
- **may_exec** (*ReadableBuffer*) – Instructions that may be executed via *StateMachine.run* calls. Some elements of the *StateMachine*'s configuration are inferred from the instructions used; for instance, if there is no `in` or `push` instruction, then the *StateMachine* is configured without a receive FIFO. In this case, passing a `may_exec` program containing an `in` instruction such as `in x`, a receive FIFO will be configured.
- **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. Does not include sideset enable
- **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
- **sideset_enable** (*bool*) – True when the top sideset bit is to enable. This should be used with the ".side_set # opt" directive
- **jmp_pin** (Pin) – the pin which determines the branch taken by JMP PIN instructions
- **jmp_pin_pull** (Pull) – The pull value for the jmp pin, default is no pull.
- **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).
- **user_interruptible** (*bool*) – When True (the default), *write()*, *readinto()*, and *write_readinto()* can be interrupted by a ctrl-C. This is useful when developing a PIO program: if there is an error in the program that causes an infinite loop, you will be able to interrupt the loop. However, if you are writing to a device that can get into a bad state if a read or write is interrupted, you may want to set this to False after your program has been vetted.
- **wrap_target** (*int*) – The target instruction number of automatic wrap. Defaults to the first instruction of the program.
- **wrap** (*int*) – The instruction after which to wrap to the `wrap` instruction. As a special case, -1 (the default) indicates the last instruction of the program.
- **offset** (*int*) – A specific offset in the state machine's program memory where the program must be loaded. The default value, -1, allows the program to be loaded at any offset. This is appropriate for most programs.

writing: bool

Returns True if a background write is in progress

pending: int

Returns the number of pending buffers for background writing.

If the number is 0, then a *StateMachine.background_write* call will not block.

frequency: int

The actual state machine frequency. This may not match the frequency requested due to internal limitations.

txstall: bool

True when the state machine has stalled due to a full TX FIFO since the last *clear_txstall* call.

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() → None

Turn off the state machine and release its resources.

__enter__() → *StateMachine*

No-op used by Context Managers. Provided by context manager helper.

__exit__() → None

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

restart() → None

Resets this state machine, runs any init and enables the clock.

run(*instructions: circuitpython_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() → None

Stops the state machine clock. Use `restart` to enable it.

write(*buffer: circuitpython_typing.ReadableBuffer*, *, *start: int = 0*, *end: int | None = None*, *swap: bool = False*) → None

Write the data contained in `buffer` to the state machine. If the buffer is empty, nothing happens.

Writes to the FIFO will match the input buffer's element size. For example, bytearray elements will perform 8 bit writes to the PIO FIFO. The RP2040's memory bus will duplicate the value into the other byte positions. So, pulling more data in the PIO assembly will read the duplicated values.

To perform 16 or 32 bits writes into the FIFO use an `array.array` with a type code of the desired size.

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)`
- **swap** (`bool`) – For 2- and 4-byte elements, swap (reverse) the byte order

background_write(*once: circuitpython_typing.ReadableBuffer | None = None*, *, *loop: circuitpython_typing.ReadableBuffer | None = None*, *swap: bool = False*) → None

Write data to the TX fifo in the background, with optional looping.

First, if any previous `once` or `loop` buffer has not been started, this function blocks until they have been started. This means that any `once` or `loop` buffer will be written at least once. Then the `once` and/or `loop` buffers are queued. and the function returns. The `once` buffer (if specified) will be written just once. Finally, the `loop` buffer (if specified) will continue being looped indefinitely.

Writes to the FIFO will match the input buffer's element size. For example, bytearray elements will perform 8 bit writes to the PIO FIFO. The RP2040's memory bus will duplicate the value into the other byte positions. So, pulling more data in the PIO assembly will read the duplicated values.

To perform 16 or 32 bits writes into the FIFO use an `array.array` with a type code of the desired size, or use `memoryview.cast` to change the interpretation of an existing buffer. To send just part of a larger buffer, slice a `memoryview` of it.

If a buffer is modified while it is being written out, the updated values will be used. However, because of interactions between CPU writes, DMA and the PIO FIFO are complex, it is difficult to predict the result of modifying multiple values. Instead, alternate between a pair of buffers.

Having both a `once` and a `loop` parameter is to support a special case in PWM generation where a change in duty cycle requires a special transitional buffer to be used exactly once. Most use cases will probably only use one of `once` or `loop`.

Having neither `once` nor `loop` terminates an existing background looping write after exactly a whole loop. This is in contrast to *stop_background_write*, which interrupts an ongoing DMA operation.

Parameters

- **once** (*~Optional[circuitpython_typing.ReadableBuffer]*) – Data to be written once
- **loop** (*~Optional[circuitpython_typing.ReadableBuffer]*) – Data to be written repeatedly
- **swap** (*bool*) – For 2- and 4-byte elements, swap (reverse) the byte order

stop_background_write() → None

Immediately stop a background write, if one is in progress. Any DMA in progress is halted, but items already in the TX FIFO are not affected.

readinto(*buffer: circuitpython_typing.WriteableBuffer*, *, *start: int = 0*, *end: int | None = None*, *swap: bool = False*) → None

Read into `buffer`. If the number of bytes to read is 0, nothing happens. The buffer includes any data added to the fifo even if it was added before this was called.

Reads from the FIFO will match the input buffer's element size. For example, bytearray elements will perform 8 bit reads from the PIO FIFO. The alignment within the 32 bit value depends on `in_shift_right`. When `in_shift_right` is True, the upper N bits will be read. The lower bits will be read when `in_shift_right` is False.

To perform 16 or 32 bits writes into the FIFO use an *array.array* with a type code of the desired size.

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)`
- **swap** (*bool*) – For 2- and 4-byte elements, swap (reverse) the byte order

write_readinto(*buffer_out: circuitpython_typing.ReadableBuffer*, *buffer_in: circuitpython_typing.WriteableBuffer*, *, *out_start: int = 0*, *out_end: int | None = None*, *in_start: int = 0*, *in_end: int | None = 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.

Data transfers to and from the FIFOs will match the corresponding buffer's element size. See *write* and *readinto* for details.

To perform 16 or 32 bits writes into the FIFO use an *array.array* with a type code of the desired size.

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)`
- **swap_out** (*bool*) – For 2- and 4-byte elements, swap (reverse) the byte order for the buffer being transmitted (written)
- **swap_in** (*bool*) – For 2- and 4-rx elements, swap (reverse) the byte order for the buffer being received (read)

clear_rxfifo() → None

Clears any unread bytes in the rxfifo.

clear_txstall() → None

Clears the txstall flag.

10.9 Silicon Labs EFR32

This port brings the Silicon Labs EFR32 series of MCUs to Circuitpython.

Refer to **mpconfigport.mk** for a full list of enabled modules sorted by family.

10.9.1 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 other peripherals where applicable.
- **common-hal/** contains the port-specific module implementations, used by shared-module and shared-bindings.
- **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 and serial, as well as the **port.c** file, which initializes the port at startup.
- **tools/** contains the Silicon Labs Configurator (SLC) tool, 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.

10.9.2 Prerequisites

Please ensure you set up your build environment appropriately, as per the guide. You will need:

- Linux: https://learn.adafruit.com/building-circuitpython/linux
- Windows Subsystem for Linux (WSL): https://learn.adafruit.com/building-circuitpython/windows-subsystem-for-linux
- macOS: Not supported yet

Install necessary packages

```
sudo apt install default-jre gcc-arm-none-eabi wget python3 python3-pip git git-lfs␣
↪gettext uncrustify
sudo python -m pip install --upgrade pip
```

Note that this uses git lfs and will not link without it. The error is something like "Unknown file format" because git lfs has a text placeholder file.

10.9.3 Supported boards

Board	Code	Build CMD
xG24 Dev Kit	brd2601b	devkit_xg24_brd2601b
xG24 Explorer Kit	brd2703a	explorerkit_xg24_brd2703a
SparkFun Thing Plus MGM240P	brd2704a	sparkfun_thingplus_matter_mgm240p_brd2704a

10.9.4 Build instructions

Ensure your clone of CircuitPython is ready to build by following the guide on the Adafruit Learning System. This includes installing the toolchain, synchronizing submodules, and running `mpy-cross`.

Clone the source code of CircuitPython from GitHub:

```
$  git clone https://github.com/SiliconLabs/circuitpython.git
$  cd circuitpython/ports/silabs
$  make fetch-port-submodules
```

Checkout the branch or tag you want to build. For example:

```
git checkout main
```

Follow the guide below to install the required packages for the Silicon Labs Configurator (SLC): https://www.silabs.com/documents/public/user-guides/ug520-software-project-generation-configuration-with-slc-cli.pdf

Once the one-time build tasks are complete, you can build at any time by navigating to the port directory:

```
make BOARD=explorerkit_xg24_brd2703a
```

You may also build with certain flags available in the makefile, depending on your board and development goals:

```
make BOARD=explorerkit_xg24_brd2703a DEBUG=1
```

Clean the project by using:

```
make BOARD=explorerkit_xg24_brd2703a clean
```

10.9.5 Flashing CircuitPython

Flash the project by using Simplicity Commander:

```
make BOARD=explorerkit_xg24_brd2703a flash
```

10.9.6 Running CircuitPython

Connecting to the Serial Console

Connect the devkit to the PC via the USB cable. The board uses serial for REPL access and debugging because the EFR32 chips has no USB support.

Windows

On Windows, we need to install a serial console e.g., PuTTY, MobaXterm. The JLink CDC UART Port can be found in the Device Manager.

Linux

Open a terminal and issue the following command:

```
ls /dev/ttyACM*
```

Then note down the correct name and substitute `com-port-name` in the following command with it:

```
screen /dev/'com-port-name'
```

Using the REPL prompt

After flashing the firmware to the board, at your first connecting to the board, you might see a blank screen. Press enter and you should be presented with a Circuitpython prompt,>>>. If not, try to reset the board (see instructions below).

You can now type in simple commands such as:

```
>>> print("Hello world!")
Hello world!
```

If something goes wrong with the board, you can reset it. Pressing CTRL+D when the prompt is open performs a soft reset.

Recommended editors

Thonny is a simple code editor that works with the Adafruit CircuitPython boards.

Config serial: Tools > Options > Interpreter > Select MicroPython > Select Port Jlink CDC UART Port

Running CircuitPython scripts

At the boot stage, two scripts will be run (if not booting in safe mode). First, the file `boot.py` will be executed. The file `boot.py` can be used to perform the initial setup. Then, after `boot.py` has been completed, the file `code.py` will be executed.

After code.py has finished executing, a REPL prompt will be presented on the serial port. Other files can also be executed by using the **Thonny** editors or using **Ampy** tool.

With the boards which support USB mass storage, we can drag the files to the board file system. However, because the EFR32 boards don't support USB mass storage, we need to use a tool like **Ampy** to copy the file to the board. You can use the latest version of **Ampy** and its command to copy the module directories to the board.

Refer to the guide below for installing the **Ampy** tool:

https://learn.adafruit.com/micropython-basics-load-files-and-run-code/install-ampy

10.10 ST Microelectronics 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.

10.10.1 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.

10.10.2 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
```

10.10.3 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.

10.10.4 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.

10.10.5 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
```

10.10.6 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.

10.11 The Unix version

The "unix" port requires a standard Unix-like environment with gcc and GNU make. This includes Linux, BSD, macOS, and Windows Subsystem for Linux. The x86 and x64 architectures are supported (i.e. x86 32- and 64-bit), as well as ARM and MIPS. Making a full-featured port to another architecture requires writing some assembly code for the exception handling and garbage collection. Alternatively, a fallback implementation based on setjmp/longjmp can be used.

To build (see section below for required dependencies):

```
$ cd ports/unix
$ make submodules
$ make
```

Then to give it a try:

```
$ ./build-standard/micropython
>>> list(5 * x + y for x in range(10) for y in [4, 2, 1])
```

Use `CTRL-D` (i.e. EOF) to exit the shell.

Learn about command-line options (in particular, how to increase heap size which may be needed for larger applications):

```
$ ./build-standard/micropython -h
```

To run the complete testsuite, use:

```
$ make test
```

The Unix port comes with a built-in package manager called `mip`, e.g.:

```
$ ./build-standard/micropython -m mip install hmac
```

or

```
$ ./build-standard/micropython
>>> import mip
>>> mip.install("hmac")
```

Browse available modules at micropython-lib. See Package management for more information about `mip`.

10.12 External dependencies

The `libffi` library and `pkg-config` tool are required. On Debian/Ubuntu/Mint derivative Linux distros, install `build-essential`(includes toolchain and make), `libffi-dev`, and `pkg-config` packages.

Other dependencies can be built together with MicroPython. This may be required to enable extra features or capabilities, and in recent versions of MicroPython, these may be enabled by default. To build these additional dependencies, in the unix port directory first execute:

```
$ make submodules
```

This will fetch all the relevant git submodules (sub repositories) that the port needs. Use the same command to get the latest versions of submodules as they are updated from time to time. After that execute:

```
$ make deplibs
```

This will build all available dependencies (regardless whether they are used or not). If you intend to build MicroPython with additional options (like cross-compiling), the same set of options should be passed to `make deplibs`. To actually enable/disable use of dependencies, edit the `ports/unix/mpconfigport.mk` file, which has inline descriptions of the options. For example, to build the SSL module, `MICROPY_PY_SSL` should be set to 1.

10.12.1 Debug Symbols

By default, builds are stripped of symbols and debug information to save size.

To build a debuggable version of the Unix port, there are two options

1. Run `make [other arguments] DEBUG=1`. Note setting `DEBUG` also reduces the optimisation level, so it's not a good option for builds that also want the best performance.
2. Run `make [other arguments] STRIP=`. Note that the value of `STRIP` is empty. This will skip the build step that strips symbols and debug information, but changes nothing else in the build configuration.

CHAPTER

ELEVEN

DESIGN AND PORTING REFERENCE

11.1 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.

11.1.1 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.

11.1.2 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.

11.1.3 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.

11.1.4 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.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)
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

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

(continues on next page)

(continued from previous page)

```
    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.

11.1.5 Verify your device

Whenever possible, make sure the 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.

11.1.6 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.

11.1.7 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."
```

11.1.8 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. 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 unmounting 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.

11.1.9 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

"""
```

Version description

After the import statements:

```
__version__ = "0.0.0+auto.0"
__repo__ = "<repo github link>"
```

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 others, `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
```

(continues on next page)

(continued from previous page)

```
@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),
```

11.1.10 Use `adafruit_register` when possible

Register is a foundational library that manages packing and unpacking data from I2C device registers. There is also Register SPI for SPI devices. When possible, use one of these libraries for unpacking and packing registers. This ensures the packing code is shared amongst all registers (even across drivers). Furthermore, it simplifies device definitions by making them declarative (only data.)

Values with non-consecutive bits in a register or that represent FIFO endpoints may not map well to existing register classes. In unique cases like these, it is ok to read and write the register directly.

Do not add all registers from a datasheet upfront. Instead, only add the ones necessary for the functionality the driver exposes. Adding them all will lead to unnecessary file size and API clutter. See this video about outside-in design from @tannewt.

I2C Example

```
from adafruit_register import i2c_bit
from adafruit_bus_device import i2c_device

class HelloWorldDevice:
    """Device with two bits to control when the words 'hello' and 'world' are lit."""

    hello = i2c_bit.RWBit(0x0, 0x0)
    """Bit to indicate if hello is lit."""

    world = i2c_bit.RWBit(0x1, 0x0)
    """Bit to indicate if world is lit."""

    def __init__(self, i2c, device_address=0x0):
        self.i2c_device = i2c_device.I2CDevice(i2c, device_address)
```

11.1.11 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):
        """Widget's one register."""
        with self.spi_device as spi:
            spi.write(b'0x00')
            spi.readinto(self.buf)
        return self.buf[0]
```

11.1.12 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

"""
```

11.1.13 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.

11.1.14 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.

11.1.15 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.

11.1.16 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*.

11.1.17 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.
- Only used when the user will not need access to variable and prefix name with a leading underscore, ex: `_SOME_CONST`.

Example

```
from adafruit_bus_device import i2c_device
from micropython import const

_DEFAULT_I2C_ADDR = const(0x42)

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

    def __init__(self, i2c, address=_DEFAULT_I2C_ADDR):
        self.i2c_device = i2c_device.I2CDevice(i2c, address)
```

11.1.18 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.

11.1.19 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)

11.1.20 Driver constant naming

When adding variables for constant values for a driver. Do not include the device's name in the variable name. For example, in `adafruit_fancy123.py`, variables should not start with `FANCY123_`. Adding this prefix increases RAM usage and .mpy file size because variable names are preserved. User code should refer to these constants as `adafruit_fancy123.HELLO_WORLD` for clarity. `adafruit_fancy123.FANCY123_HELLO_WORLD` would be overly verbose.

11.1.21 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

11.1.22 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.

11.2 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.

11.3 Porting

11.3.1 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
```

(continues on next page)

(continued from previous page)

```
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_I2CTARGET = 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 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
```

11.3.2 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.

11.3.3 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.

11.4 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.

11.4.1 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-module` 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`).

11.4.2 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 \
```

(continues on next page)

(continued from previous page)

```
        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

Modules are registered by the macro `MP_REGISTER_MODULE` from `py/obj.h`. The macro takes two arguments: the module name as a QSTR and the module object itself. The `board` module is registered like so:

```
MP_REGISTER_MODULE(MP_QSTR_board, board_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

CHAPTER

TWELVE

API REFERENCE

12.1 Standard Libraries

12.1.1 Python standard libraries

The libraries below implement a subset of the corresponding standard Python (CPython) library. They are implemented in C, not Python.

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: `binascii`, `errno`, `json`, `re`.

These libraries are not currently enabled in any CircuitPython build, but may be in the future: `ctypes`, `platform`

`builtins` – builtin functions and exceptions

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

For more information about built-ins, see the following CPython documentation:

- Builtin CPython Functions
- Builtin CPython Exceptions
- Builtin CPython Constants

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

Functions and types

builtins.**abs**()

builtins.**all**()

builtins.**any**()

builtins.**bin**()

class builtins.**bool**

class builtins.**bytearray**

class builtins.**bytes**

See CPython documentation: bytes.

builtins.**callable**()

builtins.**chr**()

builtins.**classmethod**()

builtins.**compile**()

class builtins.**complex**

builtins.**delattr**(*obj*, *name*)

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

class builtins.**dict**

builtins.**dir**()

builtins.**divmod**()

builtins.**enumerate**()

builtins.**eval**()

builtins.**exec**()

builtins.**filter**()

class builtins.**float**

class builtins.**frozenset**

frozenset() is not enabled on the smallest CircuitPython boards for space reasons.

builtins.**getattr**()

builtins.**globals**()

builtins.**hasattr**()

builtins.**hash**()

builtins.**hex**()

builtins.**id**()

builtins.**input**()

class builtins.**int**

classmethod from_bytes(*bytes*, *byteorder*)

In CircuitPython, the `byteorder` parameter must be positional (this is compatible with CPython).

to_bytes(*size*, *byteorder*)

In CircuitPython, the `byteorder` parameter must be positional (this is compatible with CPython).

builtins.**isinstance**()

builtins.**issubclass**()

builtins.**iter**()

builtins.**len**()

class builtins.**list**

builtins.**locals**()

builtins.**map**()

builtins.**max**()

class builtins.**memoryview**

builtins.**min**()

builtins.**next**()

class builtins.**object**

builtins.**oct**()

builtins.**open**()

builtins.**ord**()

builtins.**pow**()

builtins.**print**()

builtins.**property**()

builtins.**range**()

builtins.**repr**()

builtins.**reversed**()

reversed() is not enabled on the smallest CircuitPython boards for space reasons.

builtins.**round**()

class builtins.**set**

builtins.**setattr**()

class builtins.**slice**

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

builtins.**sorted**()

builtins.**staticmethod**()

class builtins.**str**

builtins.**sum**()

builtins.**super**()

class builtins.**tuple**

builtins.**type**()

builtins.**zip**()

Exceptions

exception builtins.**ArithmeticError**

exception builtins.**AssertionError**

exception builtins.**AttributeError**

exception builtins.**BaseException**

exception builtins.**BrokenPipeError**

exception builtins.**ConnectionError**

exception builtins.**EOFError**

exception builtins.**Exception**

exception builtins.**ImportError**

exception builtins.**IndentationError**

exception builtins.**IndexError**

exception builtins.**KeyboardInterrupt**

exception builtins.**KeyError**

exception builtins.**LookupError**

exception builtins.**MemoryError**

exception builtins.**NameError**

exception builtins.**NotImplementedError**

exception builtins.**OSError**

exception builtins.**OverflowError**

exception builtins.**RuntimeError**

exception builtins.**ReloadException**

ReloadException is used internally to deal with soft restarts.

Not a part of the CPython standard library

exception builtins.**StopAsyncIteration**

exception builtins.**StopIteration**

exception builtins.**SyntaxError**

exception builtins.**SystemExit**

See CPython documentation: SystemExit.

exception builtins.**TimeoutError**

exception builtins.**TypeError**

See CPython documentation: TypeError.

exception builtins.**UnicodeError**

exception builtins.**ValueError**

exception builtins.**ZeroDivisionError**

Constants

builtins.**Ellipsis**

builtins.**NotImplemented**

heapq – 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

heapq.**heappush**(*heap*, *item*)

Push the item onto the heap.

heapq.**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.

heapq.**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.

__getitem__(*index*)

Indexed read of the array, called as a[index] (where a is an array). Returns a value if *index* is an int and an array if *index* is a slice. Negative indices count from the end and IndexError is thrown if the index is out of range.

Note: __getitem__ cannot be called directly (a.__getitem__(index) fails) and is not present in __dict__, however a[index] does work.

__setitem__(*index*, *value*)

Indexed write into the array, called as a[index] = value (where a is an array). value is a single value if *index* is an int and an array if *index* is a slice. Negative indices count from the end and IndexError is thrown if the index is out of range.

Note: __setitem__ cannot be called directly (a.__setitem__(index, value) fails) and is not present in __dict__, however a[index] = value does work.

__len__()

Returns the number of items in the array, called as len(a) (where a is an array).

Note: __len__ cannot be called directly (a.__len__() fails) and the method is not present in __dict__, however len(a) does work.

__add__(*other*)

Return a new array that is the concatenation of the array with *other*, called as a + other (where a and *other* are both arrays).

Note: __add__ cannot be called directly (a.__add__(other) fails) and is not present in __dict__, however a + other does work.

__iadd__(*other*)

Concatenates the array with *other* in-place, called as a += other (where a and *other* are both arrays). Equivalent to extend(other).

Note: __iadd__ cannot be called directly (a.__iadd__(other) fails) and is not present in __dict__, however a += other does work.

`__repr__()`

Returns the string representation of the array, called as `str(a)` or `repr(a)` ` (where `a` is an `array`). Returns the string `"array(<type>, [<elements>])"`, where `<type>` is the type code letter for the array and `<elements>` is a comma separated list of the elements of the array.

Note: `__repr__` cannot be called directly (`a.__repr__()` fails) and is not present in `__dict__`, however `str(a)` and `repr(a)` both work.

`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*, *, *newline=True*`)`

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

`binascii.`**`crc32`**`(`*data*, *value=0*, */*`)`

Compute CRC-32, the 32-bit checksum of the bytes in *data* starting with an initial CRC of *value*. The default initial CRC is 0. The algorithm is consistent with the ZIP file checksum.

`collections` – collection and container types

Limitations: Not implemented on the smallest CircuitPython boards for space reasons. *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

class 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:

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.

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]
```

class 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
```

(continues on next page)

(continued from previous page)

```
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. The codes available may vary per CircuitPython build.

Constants

EEXIST, EAGAIN, etc.

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

```
try:
    os.mkdir("my_dir")
except OSError as exc:
    if exc.errno == 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

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 by Python code.

Difference to CPython

This function is a MicroPython extension.

gc.**mem_free**()

Return the number of bytes of heap RAM that is available for Python code to allocate, 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 MicroPython extension. CPython has a similar function - `set_threshold()`, but due to different GC implementations, its signature and semantics are different.

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.

class io.StringIO(*alloc_size*)

class io.BytesIO(*alloc_size*)

Create an empty *StringIO*/*BytesIO* object, preallocated to hold up to *alloc_size* number of bytes. That means that writing that amount of bytes won't lead to reallocation of the buffer, and thus won't hit out-of-memory situation or lead to memory fragmentation. These constructors are a MicroPython extension and are recommended for usage only in special cases and in system-level libraries, not for end-user applications.

Difference to CPython

These constructors are a MicroPython extension.

---

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*, *separators=None*)

Serialise `obj` to a JSON string, writing it to the given *stream*.

If specified, separators should be an `(item_separator, key_separator)` tuple. The default is `(', ', ': ')`. To get the most compact JSON representation, you should specify `(',', ':')` to eliminate whitespace.

json.**dumps**(*obj*, *separators=None*)

Return `obj` represented as a JSON string.

The arguments have the same meaning as in *dump*.

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.

platform – access to underlying platform's identifying data

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

This module tries to retrieve as much platform-identifying data as possible. It makes this information available via function APIs.

Functions

platform.**platform**()

Returns a string identifying the underlying platform. This string is composed of several substrings in the following order, delimited by dashes (-):

- the name of the platform system (e.g. Unix, Windows or MicroPython)
- the MicroPython version
- the architecture of the platform
- the version of the underlying platform
- the concatenation of the name of the libc that MicroPython is linked to and its corresponding version.

For example, this could be "MicroPython-1.20.0-xtensa-IDFv4.2.4-with-newlib3.0.0".

platform.**python_compiler**()

Returns a string identifying the compiler used for compiling MicroPython.

platform.**libc_ver**()

Returns a tuple of strings *(lib, version)*, where *lib* is the name of the libc that MicroPython is linked to, and *version* the corresponding version of this libc.

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 re

# As re doesn't support escapes itself, use of r"" strings is not
# recommended.
```

(continues on next page)

(continued from previous page)

```
regex = re.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

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)
- *_machine* - string describing the underlying machine
- *_mpy* - supported mpy file-format version (optional attribute)

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

Difference to CPython

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

---

sys.**maxsize**

Maximum value which a native integer type can hold on the current platform, or maximum value representable by CircuitPython integer type, if it's smaller than platform max value (that is the case for CircuitPython ports without long int support).

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

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

sys.**modules**

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

sys.**path**

A mutable list of directories to search for imported modules.

Difference to CPython

On MicroPython, an entry with the value `".frozen"` will indicate that import should search *frozen modules* at that point in the search. If no frozen module is found then search will *not* look for a directory called `.frozen`, instead it will continue with the next entry in `sys.path`.

---

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.**ps1**

sys.**ps2**

Mutable attributes holding strings, which are used for the REPL prompt. The defaults give the standard Python prompt of >>> and `...`.

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

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

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

sys.**tracebacklimit**
: A mutable attribute holding an integer value which is the maximum number of traceback entries to store in an exception. Set to 0 to disable adding tracebacks. Defaults to 1000.

  Note: this is not available on all ports.

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.

---

uctypes – access binary data in a structured way

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,
}
```

(continues on next page)

(continued from previous page)

```
# "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),
}

# 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).

`select` – wait for events on a set of streams

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

This module provides functions to efficiently wait for events on multiple `stream` objects (select streams which are ready for operations).

Functions

`select.`**`poll`**`()`

Create an instance of the Poll class.

`select.`**`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:

- `select.POLLIN` - data available for reading
- `select.POLLOUT` - more data can be written

Note that flags like `select.POLLHUP` and `select.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 `select.POLLIN | select.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 `select.POLL*` constants described above. Note that flags `select.POLLHUP` and `select.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 tuple". This function provides an 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.

12.1.2 Omitted `string` functions

A few string operations are not enabled on small builds due to limited flash memory: `string.center()`, `string.partition()`, `string.splitlines()`, `string.reversed()`.

12.1.3 CircuitPython/MicroPython-specific libraries

Functionality specific to the CircuitPython/MicroPython implementations is available in the following libraries.

`micropython` – MicroPython extensions and internals

Functions

micropython.**const**(*expr*)

Used to declare that the expression is a constant so that the compiler 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.

12.2 `_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:* Adapter | *None*) → 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(*data: circuitpython_typing.ReadableBuffer*, *, *scan_response: circuitpython_typing.ReadableBuffer | None = None*, *connectable: bool = True*, *anonymous: bool = False*, *timeout: int = 0*, *interval: float = 0.1*, *tx_power: int = 0*, *directed_to:* Address | *None = None*) → None

Starts advertising until *stop_advertising* is called or if connectable, another device connects to us.

Warning: If data is longer than 31 bytes, then this will automatically advertise as an extended advertisement that older BLE 4.x clients won't be able to scan for.

Note: If you set `anonymous=True`, then a timeout must be specified. If no timeout is specified, then the maximum allowed timeout will be selected automatically.

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() → None

Stop sending advertising packets.

start_scan(*prefixes: circuitpython_typing.ReadableBuffer = b''*, *, *buffer_size: int = 512*, *extended: bool = False*, *timeout: float | None = 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() → None

Stop the current scan.

connect(*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() → None

Erase all bonding information stored in flash memory.

class _bleio.**Address**(*address: circuitpython_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__(*other: object*) → bool

Two Address objects are equal if their addresses and address types are equal.

__hash__() → 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: *UUID* | None

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(*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:* *circuitpython_typing.ReadableBuffer* | *None* = *None*, *user_description:* *str* | *None* = *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`(*, *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`(*nbytes: int | None = None*) → bytes | None

Read characters. If `nbytes` is specified then read at most that many bytes. Otherwise, read everything that arrives until the connection times out. Providing the number of bytes expected is highly recommended because it will be faster.

Returns
Data read

Return type
bytes or None

`readinto`(*buf: circuitpython_typing.WriteableBuffer*) → int | None

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() → bytes

Read a line, ending in a newline character.

Returns
the line read

Return type
int or None

reset_input_buffer() → None

Discard any unread characters in the input buffer.

deinit() → 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() → None

Disconnects from the remote peripheral. Does nothing if already disconnected.

pair(*, *bond: bool = True*) → None

Pair to the peer to improve security.

discover_remote_services(*service_uuids_whitelist: Iterable[*UUID*] | None = 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(*characteristic:* Characteristic, *uuid:* UUID, *, *read_perm: int = Attribute.OPEN*, *write_perm: int = Attribute.OPEN*, *max_length: int = 20*, *fixed_length: bool = False*, *initial_value: circuitpython_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:* *int* | *None* = *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(*buf:* *circuitpython_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(*data: circuitpython_typing.ReadableBuffer*, *, *header: bytes | None = 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() → 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(*prefixes:* ScanEntry, *, *match_all: bool = True*) → bool

Returns True if the ScanEntry matches all prefixes when `match_all` is True. This is stricter than the scan filtering which accepts any advertisements that match any of the prefixes where `match_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__() → Iterator[*ScanEntry*]

Returns itself since it is the iterator.

__next__() → *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: *UUID* | None

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: int | circuitpython_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(*buffer: circuitpython_typing.WriteableBuffer, offset: int = 0*) → None

Packs the UUID into the given buffer at the given offset.

__eq__(*other: object*) → bool

Two UUID objects are equal if their values match and they are both 128-bit or both 16-bit.

12.3 _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**

Create an _EVE object

register(*o: object*) → None

flush() → 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(*b: circuitpython_typing.ReadableBuffer*) → None

Append bytes to the command FIFO.

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

AlphaFunc(*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(*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(*format: int*) → None

Set the bitmap format

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

BitmapHandle(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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(*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() → None

End the display list

End() → None

End drawing a graphics primitive

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

Jump(*dest: int*) → None

Execute commands at another location in the display list

Parameters

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

Macro(*m: int*) → None

Execute a single command from a macro register

Parameters

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

Nop() → None

No operation

PaletteSource(*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() → None

Restore the current graphics context from the context stack

Return() → None

Return from a previous call command

SaveContext() → None

Push the current graphics context on the context stack

ScissorSize(*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(*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(*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(*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(*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(*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(*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(*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(*b: float*) → None

Draw a point.

Parameters

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

LineWidth(*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(*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(*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(*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(*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(*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(*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.

12.4 _pew – LED matrix driver

class _pew.**PewPew**(*buffer: circuitpython_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).

12.5 _pixelmap – A fast pixel mapping library

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

_pixelmap.**PixelReturnType**

_pixelmap.**PixelReturnSequence**

_pixelmap.**PixelType**

_pixelmap.**PixelSequence**

class _pixelmap.**PixelMap**(*pixelbuf:* adafruit_pixelbuf.PixelBuf, *indices: Tuple[int | Tuple[int]]*)

Construct a PixelMap object that uses the given indices of the underlying pixelbuf

auto_write: bool

True if updates should be automatically written

bpp: int

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

byteorder: str

byteorder string for the buffer (read-only)

fill(*color:* adafruit_pixelbuf.PixelType) → None

Fill all the pixels in the map with the given color

indices(*index: int*) → Tuple[int]

Return the PixelBuf indices for a PixelMap index

__getitem__(*index: slice*) → *PixelReturnSequence*

__getitem__(*index: int*) → *adafruit_pixelbuf.PixelReturnType*

Retrieve the value of one of the underlying pixels at 'index'.

__setitem__(*index: slice*, *value:* adafruit_pixelbuf.PixelSequence) → None

__setitem__(*index: int*, *value:* adafruit_pixelbuf.PixelType) → 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.

__len__() → int

Length of the map

show() → None

Transmits the color data to the pixels so that they are shown. This is done automatically when *auto_write* is True.

12.6 _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: circuitpython_typing.WriteableBuffer*, *display:* busdisplay.BusDisplay, *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** (BusDisplay) – 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: circuitpython_typing.ReadableBuffer*, *palette: circuitpython_typing.ReadableBuffer*, *grid: circuitpython_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(*x: int*, *y: int*) → None

Set the offset of the layer to the specified values.

frame(*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: circuitpython_typing.ReadableBuffer*, *palette: circuitpython_typing.ReadableBuffer*, *chars: circuitpython_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(*x: int*, *y: int*) → None

Set the offset of the text to the specified values.

12.7 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.

12.7.1 adafruit_bus_device.i2c_device – I2C Device Manager

class adafruit_bus_device.i2c_device.**I2CDevice**(*i2c: busio.I2C*, *device_address: int*, *probe: bool = True*)

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)
```

(continues on next page)

(continued from previous page)

```
    # A second transaction
    with device:
        device.write(bytes_read)
```

__enter__() → *I2CDevice*

Context manager entry to lock bus.

__exit__() → None

Automatically unlocks the bus on exit.

readinto(*buffer: circuitpython_typing.WriteableBuffer*, *, *start: int = 0*, *end: int = sys.maxsize*) → None

Read into `buffer` from the device.

If `start` or `end` is provided, then the buffer will be sliced as if `buffer[start:end]` were passed. The number of bytes read will be the length of `buffer[start:end]`.

Parameters

- **buffer** (*WriteableBuffer*) – read bytes into this buffer
- **start** (*int*) – beginning of buffer slice
- **end** (*int*) – end of buffer slice; if not specified, use `len(buffer)`

write(*buffer: circuitpython_typing.ReadableBuffer*, *, *start: int = 0*, *end: int = sys.maxsize*) → 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]` were passed, but without copying the data. The number of bytes written will be the length of `buffer[start:end]`.

Parameters

- **buffer** (*ReadableBuffer*) – write out bytes from this buffer
- **start** (*int*) – beginning of buffer slice
- **end** (*int*) – end of buffer slice; if not specified, use `len(buffer)`

write_then_readinto(*out_buffer: circuitpython_typing.ReadableBuffer*, *in_buffer: circuitpython_typing.WriteableBuffer*, *, *out_start: int = 0*, *out_end: int = sys.maxsize*, *in_start: int = 0*, *in_end: int = sys.maxsize*) → None

Write the bytes from `out_buffer` to the device, then immediately reads into `in_buffer` from the device.

If `out_start` or `out_end` is provided, then the buffer will be sliced as if `out_buffer[out_start:out_end]` were passed, but without copying the data. The number of bytes written will be the length of `out_buffer[out_start:out_end]`.

If `in_start` or `in_end` is provided, then the input buffer will be sliced as if `in_buffer[in_start:in_end]` were passed, The number of bytes read will be the length of `out_buffer[in_start:in_end]`.

Parameters

- **out_buffer** (*ReadableBuffer*) – write out bytes from this buffer
- **in_buffer** (*WriteableBuffer*) – read bytes into this buffer
- **out_start** (*int*) – beginning of `out_buffer` slice
- **out_end** (*int*) – end of `out_buffer` slice; if not specified, use `len(out_buffer)`
- **in_start** (*int*) – beginning of `in_buffer` slice

- **in_end** (*int*) – end of `in_buffer slice`; if not specified, use `len(in_buffer)`

12.7.2 adafruit_bus_device.spi_device – SPI Device Manager

class adafruit_bus_device.spi_device.**SPIDevice**(*spi:* busio.SPI, *chip_select:* digitalio.DigitalInOut | *None* = *None*, *, *baudrate:* *int* = *100000*, *polarity:* *int* = *0*, *phase:* *int* = *0*, *extra_clocks:* *int* = *0*)

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. `None` if a chip select pin is not being used.
- **cs_active_value** (*bool*) – Set to true if your device requires CS to be active high. Defaults to false.
- **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__() → *busio.SPI*

Starts a SPI transaction by configuring the SPI and asserting chip select.

__exit__() → None

Ends a SPI transaction by deasserting chip select. See *Lifetime and ContextManagers* for more info.

12.8 adafruit_pixelbuf – A fast RGB(W) pixel buffer library for like NeoPixel and DotStar

The *adafruit_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".

adafruit_pixelbuf.**PixelReturnType**

adafruit_pixelbuf.**PixelReturnSequence**

adafruit_pixelbuf.**PixelType**

adafruit_pixelbuf.**PixelSequence**

class adafruit_pixelbuf.**PixelBuf**(*size: int*, ***, *byteorder: str = 'BGR'*, *brightness: float = 0*, *auto_write: bool = False*, *header: circuitpython_typing.ReadableBuffer = b''*, *trailer: circuitpython_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() → None

Transmits the color data to the pixels so that they are shown. This is done automatically when *auto_write* is True.

fill(*color:* PixelType) → None

Fills the given pixelbuf with the given color.

__getitem__(*index: slice*) → *PixelReturnSequence*

__getitem__(*index: int*) → *PixelReturnType*

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__(*index: slice*, *value:* PixelSequence) → None

__setitem__(*index: int*, *value:* PixelType) → 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.

12.9 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.

For more information on AES, refer to the Wikipedia entry.

aesio.**MODE_ECB**: int

aesio.**MODE_CBC**: int

aesio.**MODE_CTR**: int

class aesio.**AES**(*key: circuitpython_typing.ReadableBuffer*, *mode: int = 0*, *IV: circuitpython_typing.ReadableBuffer | None = 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: *MODE_ECB*, *MODE_CBC*, or *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_ECB)
cipher.encrypt_into(inp, outp)
hexlify(outp)
```

rekey(*key: circuitpython_typing.ReadableBuffer*, *IV: circuitpython_typing.ReadableBuffer | None = None*) → None

Update the AES state with the given key.

Parameters

- **key** (*ReadableBuffer*) – A 16-, 24-, or 32-byte key
- **IV** (*ReadableBuffer*) – Initialization vector to use for CBC or CTR mode

encrypt_into(*src: circuitpython_typing.ReadableBuffer*, *dest: circuitpython_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 CTR mode, there are no restrictions.

decrypt_into(*src: circuitpython_typing.ReadableBuffer*, *dest: circuitpython_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 CTR mode, there are no restrictions.

12.10 `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.

For more information about working with alarms and light/deep sleep in CircuitPython, see this Learn guide.

12.10.1 `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.

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

class alarm.time.**TimeAlarm**(*, *monotonic_time: float | None = None*, *epoch_time: int | None = 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 sleep function, such as *alarm.light_sleep_until_alarms()* or *alarm.exit_and_deep_sleep_until_alarms()*.

If the given time is already in the past, then an exception is raised. If the sleep happens after the given time, then it will wake immediately due to this time alarm.

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.

12.10.3 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.

Limitations: Not available on SAMD, nRF, or RP2040.

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:** circuitpython_typing.Alarm | None

The most recently triggered alarm. If CircuitPython was sleeping, the alarm that woke it from sleep. If no alarm occurred since the last hard reset or soft restart, value is `None`.

alarm.**light_sleep_until_alarms**(**alarms: circuitpython_typing.Alarm*) → circuitpython_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: circuitpython_typing.Alarm*, *preserve_dios: Sequence[*digitalio.DigitalInOut*] = ()*) → 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.

Parameters

- **alarms** (*circuitpython_typing.Alarm*) – the alarms that can wake the microcontroller.
- **preserve_dios** (*Sequence[*digitalio.DigitalInOut*]*) – A sequence of *DigitalInOut* objects whose state should be preserved during deep sleep. If a *DigitalInOut* in the sequence is set to be an output, its current *DigitalInOut.value* (`True` or `False`) will be preserved during the deep sleep. If a *DigitalInOut* in the sequence is set to be an input, its current *DigitalInOut.pull* value (`DOWN`, `UP`, or `None`) will be preserved during deep sleep.

Preserving *DigitalInOut* states during deep sleep can be used to ensure that external or on-board devices are powered or unpowered during sleep, among other purposes.

On some microcontrollers, some pins cannot remain in their original state for hardware reasons.

Limitations: `preserve_dios` is currently only available on Espressif.

Note: On Espressif chips, preserving pin settings during deep sleep may consume extra current. On ESP32, this was measured to be 250 uA or more. Consider not preserving pins unless you need to. Measure power consumption carefully both with no pins preserved and with the pins you might want to preserve to achieve the lowest consumption.

If CircuitPython is connected to a host computer via USB or BLE the first time a deep sleep is requested, 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.

If CircuitPython goes into a true deep sleep, and USB or BLE is reconnected, the next deep sleep will still be a true deep sleep. You must do a hard reset or power-cycle to exit a true deep sleep loop.

Here is a skeletal example:

```
import alarm
import time
import board
```

(continues on next page)

(continued from previous page)

```
print("Waking up")

# Create an alarm for 60 seconds from now, and also a pin alarm.
time_alarm = alarm.time.TimeAlarm(monotonic_time=time.monotonic() + 60)
pin_alarm = alarm.pin.PinAlarm(board.D7, False)

# Deep sleep until one of the alarm goes off. Then restart the program.
alarm.exit_and_deep_sleep_until_alarms(time_alarm, pin_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*.

Limitations: Not supported on RP2040.

Usage:

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

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

__bool__() → bool

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

__len__() → int

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

__getitem__(*index: slice*) → bytearray

__getitem__(*index: int*) → int

Returns the value at the given index.

__setitem__(*index: slice*, *value: circuitpython_typing.ReadableBuffer*) → None

__setitem__(*index: int*, *value: int*) → None

Set the value at the given index.

12.11 analogbufio – Analog Buffered IO Hardware Support

The *analogbufio* module contains classes to provide access to analog-to-digital conversion and digital-to-analog (DAC) for multiple value transfer.

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.

TODO: For the essentials of *analogbufio*, see the CircuitPython Essentials Learn guide

TODO: For more information on using *analogbufio*, see this additional Learn guide

class analogbufio.**BufferedIn**(*pin:* microcontroller.Pin, *, *sample_rate: int*)

Capture multiple analog voltage levels to the supplied buffer

Usage:

```
import board
import analogbufio
import array

length = 1000
mybuffer = array.array("H", [0x0000] * length)
rate = 500000
adcbuf = analogbufio.BufferedIn(board.GP26, sample_rate=rate)
adcbuf.readinto(mybuffer)
adcbuf.deinit()
for i in range(length):
    print(i, mybuffer[i])

(TODO) The reference voltage varies by platform so use
``reference_voltage`` to read the configured setting.
(TODO) Provide mechanism to read CPU Temperature.
```

Create a *BufferedIn* on the given pin and given sample rate.

Parameters

- **pin** (Pin) – the pin to read from
- **sample_rate** (*~int*) – rate: sampling frequency, in samples per second

deinit() → None

Shut down the *BufferedIn* and release the pin for other use.

__enter__() → *BufferedIn*

No-op used by Context Managers.

__exit__() → None

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

readinto(*buffer: circuitpython_typing.WriteableBuffer*) → int

Fills the provided buffer with ADC voltage values.

ADC values will be read into the given buffer at the supplied sample_rate. Depending on the buffer type-code, 'B', 'H', samples are 8-bit byte-arrays or 16-bit half-words and are always unsigned. The ADC most significant bits of the ADC are kept. (See https://docs.circuitpython.org/en/latest/docs/library/array.html)

Parameters

buffer (*WriteableBuffer*) – buffer: A buffer for samples

12.12 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 *

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.

For the essentials of *analogio*, see the CircuitPython Essentials Learn guide

For more information on using *analogio*, see this additional Learn guide

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. Note the ADC value may not scale to the actual voltage linearly at ends of the analog range.

deinit() → None

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

__enter__() → *AnalogIn*

No-op used by Context Managers.

__exit__() → 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).

Limitations: Not available on nRF, RP2040, Spresense, as there is no on-chip DAC. On Espressif, available only on ESP32 and ESP32-S2; other chips do not have a DAC.

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() → None

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

__enter__() → *AnalogOut*

No-op used by Context Managers.

__exit__() → None

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

12.13 atexit – Atexit Module

This module defines functions to register and unregister cleanup functions. Functions thus registered are automatically executed upon normal vm termination.

These functions are run in the reverse order in which they were registered; if you register A, B, and C, they will be run in the order C, B, A.

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

atexit.**register**(*func: Callable[Ellipsis, Any]*, **args: Any | None*, ***kwargs: Any | None*) → Callable[Ellipsis, Any]

Register func as a function to be executed at termination.

Any optional arguments that are to be passed to func must be passed as arguments to *register()*. It is possible to register the same function and arguments more than once.

At normal program termination (for instance, if *sys.exit()* is called or the vm execution completes), all functions registered are called in last in, first out order.

If an exception is raised during execution of the exit handler, a traceback is printed (unless *SystemExit* is raised) and the execution stops.

This function returns func, which makes it possible to use it as a decorator.

atexit.**unregister**(*func: Callable[Ellipsis, Any]*) → None

Remove func from the list of functions to be run at termination.

unregister() silently does nothing if func was not previously registered. If func has been registered more than once, every occurrence of that function in the atexit call stack will be removed.

12.14 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 classes 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, *, *main_clock:* microcontroller.Pin | *None* = *None*, *left_justified: bool* = *False*)

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
- **main_clock** (Pin) – The main clock 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 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() → None

Deinitialises the I2SOut and releases any hardware resources for reuse.

__enter__() → *I2SOut*

No-op used by Context Managers.

__exit__() → None

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

play(*sample: circuitpython_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() → None

Stops playback.

pause() → None

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

resume() → 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.

Limitations: On SAMD and RP2040, supports only 8 or 16 bit mono input, with 64x oversampling. On nRF52840, supports only 16 bit mono input at 16 kHz; oversampling is fixed at 64x. Not provided on nRF52833 for space reasons. Not available on Espressif.

For example, to record 8-bit unsigned samples to a buffer:

```
import audiobusio
import board

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

To record 16-bit unsigned samples to a buffer:

```
import audiobusio
import board

# Prep a buffer to record into.
b = array.array("H", [0] * 200)
with audiobusio.PDMIn(board.MICROPHONE_CLOCK, board.MICROPHONE_DATA, sample_
↪rate=16000, bit_depth=16) as mic:
    mic.record(b, len(b))
```

sample_rate: int

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

deinit() → None

Deinitialises the PDMIn and releases any hardware resources for reuse.

__enter__() → *PDMIn*

No-op used by Context Managers.

__exit__() → None

Automatically deinitializes the hardware when exiting a context.

record(*destination: circuitpython_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.

12.15 audiocore – Support for audio samples

class audiocore.**RawSample**(*buffer: circuitpython_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 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: int | None

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() → None

Deinitialises the RawSample and releases any hardware resources for reuse.

__enter__() → *RawSample*

No-op used by Context Managers.

__exit__() → None

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

class audiocore.**WaveFile**(*file: str | BinaryIO*, *buffer: circuitpython_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, which can prevent memory fragmentation.

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

Parameters

- **file** (*Union[str, BinaryIO]*) – The name of a wave file (preferred) or an already opened wave file
- **buffer** (*WriteableBuffer*) – Optional pre-allocated buffer, that will be split in half and used for double-buffering of the data. The buffer must be 8 to 1024 bytes long. If not provided, two 256 byte buffers are initially 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)

wav = audiocore.WaveFile("cplay-5.1-16bit-16khz.wav")
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() → None

Deinitialises the WaveFile and releases all memory resources for reuse.

__enter__() → *WaveFile*

No-op used by Context Managers.

__exit__() → None

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

12.16 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.

For more information on working with this module, refer to the CircuitPython Essentials Learn Guide.

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:* microcontroller.Pin | *None* = *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)
```

(continues on next page)

(continued from previous page)

```
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() → None

Deinitialises the AudioOut and releases any hardware resources for reuse.

__enter__() → *AudioOut*

No-op used by Context Managers.

__exit__() → None

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

play(*sample: circuitpython_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() → None

Stops playback and resets to the start of the sample.

pause() → None

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

resume() → None

Resumes sample playback after *pause()*.

12.17 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:
  # 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() → None

Deinitialises the Mixer and releases any hardware resources for reuse.

__enter__() → *Mixer*

No-op used by Context Managers.

__exit__() → None

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

play(*sample: circuitpython_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(*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(*sample: circuitpython_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() → None

Stops playback of the sample on this voice.

12.18 audiomp3 – Support for MP3-compressed audio files

For more information about working with MP3 files in CircuitPython, see this CircuitPython Essentials Learn guide page.

class audiomp3.**MP3Decoder**(*file: str | BinaryIO*, *buffer: circuitpython_typing.WriteableBuffer*)

Load a mp3 file for audio playback

Note: MP3Decoder uses a lot of contiguous memory, so care should be given to optimizing memory usage. More information and recommendations can be found here: https://learn.adafruit.com/Memory-saving-tips-for-CircuitPython/reducing-memory-fragmentation

---

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

Parameters

- **file** (*Union[str, BinaryIO]*) – The name of a mp3 file (preferred) or an 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.

Playback of mp3 audio is CPU intensive, and the exact limit depends on many factors such as the particular microcontroller, SD card or flash performance, and other code in use such as displayio. If playback is garbled, skips, or plays as static, first try using a "simpler" mp3:

- Use constant bit rate (CBR) not VBR or ABR (variable or average bit rate) when encoding your mp3 file
- Use a lower sample rate (e.g., 11.025kHz instead of 48kHz)
- Use a lower bit rate (e.g., 32kbit/s instead of 256kbit/s)

Reduce activity taking place at the same time as mp3 playback. For instance, only update small portions of a displayio screen if audio is playing. Disable auto-refresh and explicitly call refresh.

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)

mp3 = audiomp3.MP3Decoder("cplay-16bit-16khz-64kbps.mp3")
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)

samples_decoded: int

The number of audio samples decoded from the current file. (read only)

deinit() → None

Deinitialises the MP3 and releases all memory resources for reuse.

__enter__() → *MP3Decoder*

No-op used by Context Managers.

__exit__() → None

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

open(*filepath: str*) → None

Takes in the name of a mp3 file, opens it, and replaces the old playback file.

12.19 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:* microcontroller.Pin | *None* = *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.

Limitations: On mimxrt10xx, low sample rates may have an audible "carrier" frequency. The manufacturer datasheet states that the "MQS" peripheral is intended for 44 kHz or 48kHz input signals.

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

# 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() → None

Deinitialises the PWMAudioOut and releases any hardware resources for reuse.

__enter__() → *PWMAudioOut*

No-op used by Context Managers.

__exit__() → None

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

play(*sample: circuitpython_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() → None

Stops playback and resets to the start of the sample.

pause() → None

Stops playback temporarily while remembering the position. Use `resume` to resume playback.

resume() → None

Resumes sample playback after `pause()`.

12.20 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() → None

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

__enter__() → *I2C*

No-op used in Context Managers.

__exit__() → None

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

scan() → 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() → bool

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

unlock() → None

Releases the I2C lock.

readfrom_into(*address: int*, *buffer: circuitpython_typing.WriteableBuffer*, ***, *start: int = 0*, *end: int = sys.maxsize*) → 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(*address: int*, *buffer: circuitpython_typing.ReadableBuffer*, ***, *start: int = 0*, *end: int = sys.maxsize*) → 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]` were passed, but without copying the data. The number of bytes written will be the length of `buffer[start:end]`.

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*) – beginning of buffer slice
- **end** (*int*) – end of buffer slice; if not specified, use `len(buffer)`

writeto_then_readfrom(*address: int*, *out_buffer: circuitpython_typing.ReadableBuffer*, *in_buffer: circuitpython_typing.ReadableBuffer*, *, *out_start: int = 0*, *out_end: int = sys.maxsize*, *in_start: int = 0*, *in_end: int = sys.maxsize*) → 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 `out_start` or `out_end` is provided, then the buffer will be sliced as if `out_buffer[out_start:out_end]` were passed, but without copying the data. The number of bytes written will be the length of `out_buffer[start:end]`.

If `in_start` or `in_end` is provided, then the input buffer will be sliced as if `in_buffer[in_start:in_end]` were passed, The number of bytes read will be the length of `out_buffer[in_start:in_end]`.

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*) – beginning of `out_buffer` slice
- **out_end** (*int*) – end of `out_buffer` slice; if not specified, use `len(out_buffer)`
- **in_start** (*int*) – beginning of `in_buffer` slice
- **in_end** (*int*) – end of `in_buffer slice`; if not specified, use `len(in_buffer)`

class bitbangio.**SPI**(*clock:* microcontroller.Pin, *MOSI:* microcontroller.Pin | *None* = *None*, *MISO:* microcontroller.Pin | *None* = *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() → None

Turn off the SPI bus.

__enter__() → *SPI*

No-op used by Context Managers.

__exit__() → None

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

configure(*, *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() → bool

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

Returns
True when lock has been grabbed

Return type
bool

unlock() → None

Releases the SPI lock.

write(*buf: circuitpython_typing.ReadableBuffer*, *, *start: int = 0*, *end: int = sys.maxsize*) → None

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

If `start` or `end` is provided, then the buffer will be sliced as if `buffer[start:end]` were passed, but without copying the data. The number of bytes written will be the length of `buffer[start:end]`.

Parameters

- **buffer** (*ReadableBuffer*) – buffer containing the bytes to write
- **start** (*int*) – beginning of buffer slice
- **end** (*int*) – end of buffer slice; if not specified, use `len(buffer)`

readinto(*buffer: circuitpython_typing.WriteableBuffer*, *, *start: int = 0*, *end: int = sys.maxsize*, *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.

If `start` or `end` is provided, then the buffer will be sliced as if `buffer[start:end]` were passed. The number of bytes read will be the length of `buffer[start:end]`.

Parameters

- **buffer** (*WriteableBuffer*) – read bytes into this buffer
- **start** (*int*) – beginning of buffer slice
- **end** (*int*) – end of buffer slice; if not specified, use `len(buffer)`
- **write_value** (*int*) – value to write while reading

write_readinto(*out_buffer: circuitpython_typing.ReadableBuffer*, *in_buffer: circuitpython_typing.WriteableBuffer*, ***, *out_start: int = 0*, *out_end: int = sys.maxsize*, *in_start: int = 0*, *in_end: int = sys.maxsize*) → None

Write out the data in `out_buffer` while simultaneously reading data into `in_buffer`. The SPI object must be locked.

If `out_start` or `out_end` is provided, then the buffer will be sliced as if `out_buffer[out_start:out_end]` were passed, but without copying the data. The number of bytes written will be the length of `out_buffer[out_start:out_end]`.

If `in_start` or `in_end` is provided, then the input buffer will be sliced as if `in_buffer[in_start:in_end]` were passed, The number of bytes read will be the length of `out_buffer[in_start:in_end]`.

The lengths of the slices defined by `out_buffer[out_start:out_end]` and `in_buffer[in_start:in_end]` must be equal. If buffer slice lengths are both 0, nothing happens.

Parameters

- **out_buffer** (*ReadableBuffer*) – write out bytes from this buffer
- **in_buffer** (*WriteableBuffer*) – read bytes into this buffer
- **out_start** (*int*) – beginning of `out_buffer` slice
- **out_end** (*int*) – end of `out_buffer` slice; if not specified, use `len(out_buffer)`
- **in_start** (*int*) – beginning of `in_buffer` slice
- **in_end** (*int*) – end of `in_buffer slice`; if not specified, use `len(in_buffer)`

12.21 `bitmapfilter` – Convolve an image with a kernel

`bitmapfilter`.**morph**(*bitmap:* displayio.Bitmap, *weights: Sequence[int]*, *mul: float | None = None*, *add: float = 0*, *mask:* displayio.Bitmap *| None = None*, *threshold=False*, *offset: int = 0*, *invert: bool = False*) → *displayio.Bitmap*

The name of the function comes from OpenMV. ImageMagick calls this "-morphology" ("-morph" is an unrelated image blending algorithm). PIL calls this "kernel".

For background on how this kind of image processing, including some useful `weights` values, see wikipedia's article on the subject.

The `bitmap`, which must be in RGB565_SWAPPED format, is modified according to the `weights`. Then a scaling factor `mul` and an offset factor `add` are applied.

The `weights` must be a sequence of integers. The length of the tuple must be the square of an odd number, usually 9 and sometimes 25. Specific weights create different effects. For instance, these weights represent a 3x3 gaussian blur: `[1, 2, 1, 2, 4, 2, 1, 2, 1]`

`mul` is number to multiply the convolution pixel results by. If `None` (the default) is passed, the value of `1/sum(weights)` is used (or `1` if `sum(weights)` is `0`). For most weights, his default value will preserve the overall image brightness.

`add` is a value to add to each convolution pixel result.

`mul` basically allows you to do a global contrast adjustment and add allows you to do a global brightness adjustment. Pixels that go outside of the image mins and maxes for color channels will be clipped.

If you'd like to adaptive threshold the image on the output of the filter you can pass `threshold=True` which will enable adaptive thresholding of the image which sets pixels to one or zero based on a pixel's brightness in relation to the brightness of the kernel of pixels around them. A negative `offset` value sets more pixels to 1 as you make it more negative while a positive value only sets the sharpest contrast changes to 1. Set `invert` to invert the binary image resulting output.

`mask` is another image to use as a pixel level mask for the operation. The mask should be an image the same size as the image being operated on. Only pixels set to a non-zero value in the mask are modified.

```
kernel_gauss_3 = [
    1, 2, 1,
    2, 4, 2,
    1, 2, 1]

def blur(bitmap):
    """Blur the bitmap with a 3x3 gaussian kernel"""
    bitmapfilter.morph(bitmap, kernel_gauss_3, 1/sum(kernel_gauss_3))
```

class bitmapfilter.**ChannelScale**(*r: float*, *g: float*, *b: float*)

A weight object to use with mix() that scales each channel independently

This is useful for global contrast and brightness adjustment on a per-component basis. For instance, to cut red contrast in half (while keeping the minimum value as black or 0.0),

```
reduce_red_contrast = bitmapfilter.ChannelScale(0.5, 1, 1)
```

Construct a ChannelScale object

The `r` parameter gives the scale factor for the red channel of pixels, and so forth.

class bitmapfilter.**ChannelScaleOffset**(*r: float*, *r_add: float*, *g: float*, *g_add: float*, *b: float*, *b_add: float*)

A weight object to use with mix() that scales and offsets each channel independently

The `r`, `g`, and `b` parameters give a scale factor for each color component, while the `r_add`, `g_add` and `b_add` give offset values added to each component.

This is useful for global contrast and brightness adjustment on a per-component basis. For instance, to cut red contrast in half while adjusting the brightness so that the middle value is still 0.5:

```
reduce_red_contrast = bitmapfilter.ChannelScaleOffset(
        0.5, 0.25,
        1, 0,
        1, 0)
```

Construct a ChannelScaleOffset object

class bitmapfilter.**ChannelMixer**(*rr: float*, *rg: float*, *rb: float*, *gr: float*, *gg: float*, *gb: float*, *br: float*, *bg: float*, *bb: float*)

A weight object to use with mix() that mixes different channels together

The parameters with names like `rb` give the fraction of each channel to mix into every other channel. For instance, `rb` gives the fraction of blue to mix into red, and `gg` gives the fraction of green to mix into green.

Conversion to sepia is an example where a ChannelMixer is appropriate, because the sepia conversion is defined as mixing a certain fraction of R, G, and B input values into each output value:

```
sepia_weights = bitmapfilter.ChannelMixer(
    .393,  .769,   .189,
    .349,  .686,   .168,
    .272,  .534,   .131)

def sepia(bitmap):
    """Convert the bitmap to sepia"""
    bitmapfilter.mix(bitmap, sepia_weights)
mix_into_red = ChannelMixer(
        0.5, 0.25, 0.25,
        0,   1,    0,
        0,   1,    0)
```

Construct a ChannelMixer object

class bitmapfilter.**ChannelMixerOffset**(*rr: float*, *rg: float*, *rb: float*, *r_add: float*, *gr: float*, *gg: float*, *gb: float*, *g_add: float*, *br: float*, *bg: float*, *bb: float*, *b_add: float*)

A weight object to use with mix() that mixes different channels together, plus an offset value

The parameters with names like `rb` give the fraction of each channel to mix into every other channel. For instance, `rb` gives the fraction of blue to mix into red, and `gg` gives the fraction of green to mix into green. The `r_add`, `g_add` and `b_add` parameters give offsets applied to each component.

For instance, to perform sepia conversion but also increase the overall brightness by 10%:

```
sepia_weights_brighten = bitmapfilter.ChannelMixerOffset(
    .393,  .769,   .189, .1
    .349,  .686,   .168, .1
    .272,  .534,   .131, .1)
```

Construct a ChannelMixerOffset object

bitmapfilter.**mix**(*bitmap:* displayio.Bitmap, *weights:* ChannelScale | ChannelScaleOffset | ChannelMixer | ChannelMixerOffset, *mask:* displayio.Bitmap | *None* = *None*) → *displayio.Bitmap*

Perform a channel mixing operation on the bitmap

This is similar to the "channel mixer" tool in popular photo editing software. Imagemagick calls this "-color-matrix". In PIL, this is accomplished with the `convert` method's `matrix` argument.

The `bitmap`, which must be in RGB565_SWAPPED format, is modified according to the `weights`.

The `weights` must be one of the above types: *ChannelScale*, *ChannelScaleOffset*, *ChannelMixer*, or *ChannelMixerOffset*. For the effect of each different kind of weights object, see the type documentation.

After computation, any out of range values are clamped to the greatest or smallest valid value.

`mask` is another image to use as a pixel level mask for the operation. The mask should be an image the same size as the image being operated on. Only pixels set to a non-zero value in the mask are modified.

bitmapfilter.**solarize**(*bitmap*, *threshold: float* = *0.5*, *mask:* displayio.Bitmap | *None* = *None*)

Create a "solarization" effect on an image

This filter inverts pixels with brightness values above `threshold`, while leaving lower brightness pixels alone.

This effect is similar to an effect observed in real life film which can also be produced during the printmaking process

PIL and ImageMagic both call this "solarize".

bitmapfilter.**LookupFunction**

Any function which takes a number and returns a number. The input and output values should be in the range from 0 to 1 inclusive.

bitmapfilter.**ThreeLookupFunctions**

Any sequenceof three *LookupFunction* objects

bitmapfilter.**lookup**(*bitmap:* displayio.Bitmap, *lookup:* LookupFunction | ThreeLookupFunctions, *mask:* displayio.Bitmap | *None*) → *displayio.Bitmap*

Modify the channels of a bitmap according to a look-up table

This can be used to implement non-linear transformations of color values, such as gamma curves.

This is similar to, but more limiting than, PIL's "LUT3D" facility. It is not directly available in OpenMV or ImageMagic.

The `bitmap`, which must be in RGB565_SWAPPED format, is modified according to the values of the `lookup` function or functions.

If one `lookup` function is supplied, the same function is used for all 3 image channels. Otherwise, it must be a tuple of 3 functions. The first function is used for R, the second function for G, and the third for B.

Each lookup function is called for each possible channel value from 0 to 1 inclusive (64 times for green, 32 times for red or blue), and the return value (also from 0 to 1) is used whenever that color value is returned.

`mask` is another image to use as a pixel level mask for the operation. The mask should be an image the same size as the image being operated on. Only pixels set to a non-zero value in the mask are modified.

bitmapfilter.**false_color**(*bitmap:* displayio.Bitmap, *palette:* displayio.Palette, *mask:* displayio.Bitmap | *None*) → *displayio.Bitmap*

Convert the image to false color using the given palette

In OpenMV this is accomplished via the `ironbow` function, which uses a default palette known as "ironbow". Imagemagic produces a similar effect with `-clut`. PIL can accomplish this by converting an image to "L" format, then applying a palette to convert it into "P" mode.

The `bitmap`, which must be in RGB565_SWAPPED format, is converted into false color.

The `palette`, which must be of length 256, is used as a look-up table.

Each pixel is converted to a luminance (brightness/greyscale) value in the range 0..255, then the corresponding palette entry is looked up and stored in the bitmap.

`mask` is another image to use as a pixel level mask for the operation. The mask should be an image the same size as the image being operated on. Only pixels set to a non-zero value in the mask are modified.

bitmapfilter.**BlendFunction**

A function used to blend two images

bitmapfilter.**BlendTable**

A precomputed blend table

There is not actually a BlendTable type. The real type is actually any buffer 4096 bytes in length.

bitmapfilter.**blend_precompute**(*lookup:* BlendFunction, *table:* BlendTable | *None* = *None*) → *BlendTable*

Precompute a BlendTable from a BlendFunction

If the optional `table` argument is provided, an existing *BlendTable* is updated with the new function values.

The function's two arguments will range from 0 to 1. The returned value should also range from 0 to 1.

A function to do a 33% blend of each source image could look like this:

```
def blend_one_third(a, b):
    return a * .33 + b * .67
```

bitmapfilter.**blend**(*dest:* displayio.Bitmap, *src1:* displayio.Bitmap, *src2:* displayio.Bitmap, *lookup:* BlendFunction | BlendTable, *mask:* displayio.Bitmap | *None* = *None*) → *displayio.Bitmap*

Blend the 'src1' and 'src2' images according to lookup function or table 'lookup'

If `lookup` is a function, it is converted to a *BlendTable* by internally calling blend_precompute. If a blend function is used repeatedly it can be more efficient to compute it once with *blend_precompute*.

If the mask is supplied, pixels from `src1` are taken unchanged in masked areas.

The source and destination bitmaps may be the same bitmap.

The destination bitmap is returned.

12.22 `bitmaptools` – Collection of bitmap manipulation tools

Note: If you're looking for information about displaying bitmaps on screens in CircuitPython, see this Learn guide for information about using the *displayio* module.

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. Defaults to None which causes it to use the horizontal midway point of the destination bitmap.
- **oy** (*int*) – Vertical pixel location in destination bitmap where source bitmap point (px,py) is placed. Defaults to None which causes it to use the vertical midway point of the destination bitmap.
- **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). Defaults to None which causes it to use the horizontal midway point in the source bitmap.
- **py** (*int*) – Vertical pixel location in source bitmap that is placed into the destination bitmap at (ox,oy). Defaults to None which causes it to use the vertical midway point in the source bitmap.

- **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). Defaults to None which gets treated as 0.0 radians or no rotation.
- **scale** (*float*) – Scaling factor. Defaults to None which gets treated as 1.0 or same as original source size.
- **skip_index** (*int*) – Bitmap palette index in the source that will not be copied, set to None to copy all pixels

class bitmaptools.**BlendMode**

The blend mode for *alphablend* to operate use

Normal: *BlendMode*

Blend with equal parts of the two source bitmaps

Screen: *BlendMode*

Blend based on the value in each color channel. The result keeps the lighter colors and discards darker colors.

bitmaptools.**alphablend**(*dest_bitmap:* displayio.Bitmap, *source_bitmap_1:* displayio.Bitmap, *source_bitmap_2:* displayio.Bitmap, *colorspace:* displayio.Colorspace, *factor1: float = 0.5*, *factor2: float | None = None*, *blendmode:* BlendMode | *None = BlendMode.Normal*, *skip_source1_index: int | None = None*, *skip_source2_index: int | None = None*) → None

Alpha blend the two source bitmaps into the destination.

It is permitted for the destination bitmap to be one of the two source bitmaps.

Parameters

- **dest_bitmap** (*bitmap*) – Destination bitmap that will be written into
- **source_bitmap_1** (*bitmap*) – The first source bitmap
- **source_bitmap_2** (*bitmap*) – The second source bitmap
- **factor1** (*float*) – The proportion of bitmap 1 to mix in
- **factor2** (*float*) – The proportion of bitmap 2 to mix in. If specified as `None`, `1-factor1` is used. Usually the proportions should sum to 1.
- **colorspace** (displayio.Colorspace) – The colorspace of the bitmaps. They must all have the same colorspace. Only the following colorspaces are permitted: `L8`, `RGB565`, `RGB565_SWAPPED`, `BGR565` and `BGR565_SWAPPED`.
- **blendmode** (bitmaptools.BlendMode) – The blend mode to use. Default is Normal.
- **skip_source1_index** (*int*) – Bitmap palette or luminance index in source_bitmap_1 that will not be blended, set to None to blend all pixels
- **skip_source2_index** (*int*) – Bitmap palette or luminance index in source_bitmap_2 that will not be blended, set to None to blend all pixels

For the L8 colorspace, the bitmaps must have a bits-per-value of 8. For the RGB colorspaces, they must have a bits-per-value of 16.

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.**boundary_fill**(*dest_bitmap:* displayio.Bitmap, *x: int*, *y: int*, *fill_color_value: int*, *replaced_color_value: int*) → None

Draws the color value into the destination bitmap enclosed area of pixels of the background_value color. Like "Paint Bucket" fill tool.

Parameters

- **dest_bitmap** (*bitmap*) – Destination bitmap that will be written into
- **x** (*int*) – x-pixel position of the first pixel to check and fill if needed
- **y** (*int*) – y-pixel position of the first pixel to check and fill if needed
- **fill_color_value** (*int*) – Bitmap palette index that will be written into the enclosed area in the destination bitmap
- **replaced_color_value** (*int*) – Bitmap palette index that will filled with the value color in the enclosed area 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.**draw_polygon**(*dest_bitmap:* displayio.Bitmap, *xs: circuitpython_typing.ReadableBuffer*, *ys: circuitpython_typing.ReadableBuffer*, *value: int*, *close: bool | None = True*) → None

Draw a polygon connecting points on provided bitmap with provided value

Parameters

- **dest_bitmap** (*bitmap*) – Destination bitmap that will be written into
- **xs** (*ReadableBuffer*) – x-pixel position of the polygon's vertices

- **ys** (*ReadableBuffer*) – y-pixel position of the polygon's vertices
- **value** (*int*) – Bitmap palette index that will be written into the line in the destination bitmap
- **close** (*bool*) – (Optional) Whether to connect first and last point. (True)

```
import board
import displayio
import bitmaptools

display = board.DISPLAY
main_group = displayio.Group()
display.root_group = main_group

palette = displayio.Palette(3)
palette[0] = 0xffffff
palette[1] = 0x0000ff
palette[2] = 0xff0000

bmp = displayio.Bitmap(128,128, 3)
bmp.fill(0)

xs = bytes([4, 101, 101, 19])
ys = bytes([4, 19,  121, 101])
bitmaptools.draw_polygon(bmp, xs, ys, 1)

xs = bytes([14, 60, 110])
ys = bytes([14, 24,  90])
bitmaptools.draw_polygon(bmp, xs, ys, 2)

tilegrid = displayio.TileGrid(bitmap=bmp, pixel_shader=palette)
main_group.append(tilegrid)

while True:
    pass
```

bitmaptools.**arrayblit**(*bitmap:* displayio.Bitmap, *data:* *circuitpython_typing.ReadableBuffer*, *x1:* *int* = *0*, *y1:* *int* = *0*, *x2:* *int* | *None* = *None*, *y2:* *int* | *None* = *None*, *skip_index:* *int* | *None* = *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 available 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** (*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 Significant 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).

class bitmaptools.**DitherAlgorithm**

Identifies the algorithm for dither to use

Atkinson: *DitherAlgorithm*

The classic Atkinson dither, often associated with the Hypercard esthetic

FloydStenberg: *DitherAlgorithm*

The Floyd-Stenberg dither

bitmaptools.**dither**(*dest_bitmap:* displayio.Bitmap, *source_bitmapp:* displayio.Bitmap, *source_colorspace:* displayio.Colorspace, *algorithm:* DitherAlgorithm = *DitherAlgorithm.Atkinson*) → None

Convert the input image into a 2-level output image using the given dither algorithm.

Parameters

- **dest_bitmap** (*bitmap*) – Destination bitmap. It must have a value_count of 2 or 65536. The stored values are 0 and the maximum pixel value.

- **source_bitmap** (*bitmap*) – Source bitmap that contains the graphical region to be dithered. It must have a value_count of 65536.
- **colorspace** – The colorspace of the image. The supported colorspaces are `RGB565`, `BGR565`, `RGB565_SWAPPED`, and `BGR565_SWAPPED`
- **algorithm** – The dither algorithm to use, one of the *DitherAlgorithm* values.

bitmaptools.**draw_circle**(*dest_bitmap:* displayio.Bitmap, *x: int*, *y: int*, *radius: int*, *value: int*) → None

Draws a circle into a bitmap specified using a center (x0,y0) and radius r.

Parameters

- **dest_bitmap** (*bitmap*) – Destination bitmap that will be written into
- **x** (*int*) – x-pixel position of the circle's center
- **y** (*int*) – y-pixel position of the circle's center
- **radius** (*int*) – circle's radius
- **value** (*int*) – Bitmap palette index that will be written into the circle in the destination bitmap

```
import board
import displayio
import bitmaptools

display = board.DISPLAY
main_group = displayio.Group()
display.root_group = main_group

palette = displayio.Palette(2)
palette[0] = 0xffffff
palette[1] = 0x440044

bmp = displayio.Bitmap(128,128, 2)
bmp.fill(0)

bitmaptools.circle(64,64, 32, 1)

tilegrid = displayio.TileGrid(bitmap=bmp, pixel_shader=palette)
main_group.append(tilegrid)

while True:
    pass
```

bitmaptools.**blit**(*dest_bitmap:* displayio.Bitmap, *source_bitmap:* displayio.Bitmap, *x: int*, *y: int*, ***, *x1: int = 0*, *y1: int = 0*, *x2: int | None = None*, *y2: int | None = None*, *skip_source_index: int | None = None*, *skip_dest_index: int | None = None*) → 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

- **dest_bitmap** (*bitmap*) – Destination bitmap that the area will be copied into.
- **source_bitmap** (*bitmap*) – Source bitmap that contains the graphical region to be copied

- **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
- **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. If unspecified or `None`, the source bitmap width is used.
- **y2** (*int*) – Maximum y-value (exclusive) for rectangular bounding box to be copied from the source bitmap. If unspecified or `None`, the source bitmap height is used.
- **skip_source_index** (*int*) – bitmap palette index in the source that will not be copied, set to None to copy all pixels
- **skip_dest_index** (*int*) – bitmap palette index in the destination bitmap that will not get overwritten by the pixels from the source

12.23 `bitops` – Routines for low-level manipulation of binary data

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

12.24 `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.

Another common use of this module is to use serial communication buses with the default pins and settings. For more information about serial communcication in CircuitPython, see the *busio*.

For more information regarding the typical usage of *board*, refer to the CircuitPython Essentials Learn guide

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

board.**board_id**: str

Board ID string. The unique identifier for the board model in circuitpython, as well as on circuitpython.org. Example: "hallowing_m0_express".

board.**I2C**() → *busio.I2C*

Returns the busio.I2C object for the board's designated I2C bus(es). The object created is a singleton, and uses the default parameter values for busio.I2C.

board.**SPI**() → *busio.SPI*

Returns the busio.SPI object for the board's designated SPI bus(es). The object created is a singleton, and uses the default parameter values for busio.SPI.

board.**UART**() → *busio.UART*

Returns the busio.UART object for the board's designated UART bus(es). The object created is a singleton, and uses the default parameter values for busio.UART.

12.25 busdisplay

Displays a displayio object tree on an external device with a built-in framebuffer

busdisplay.**_DisplayBus**

fourwire.FourWire, paralleldisplaybus.ParallelBus or i2cdisplaybus.I2CDisplayBus

class busdisplay.**BusDisplay**(*display_bus:* _DisplayBus, *init_sequence:* *circuitpython_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*, *backlight_pin:* microcontroller.Pin | *None* = *None*, *brightness_command:* *int* | *None* = *None*, *brightness:* *float* = *1.0*, *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, paralleldisplaybus.ParallelBus or I2CDisplayBus).

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 = busdisplay.BusDisplay(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.
- **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.

- **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
- **set_vertical_scroll** (*int*) – This parameter is accepted but ignored for backwards compatibility. It will be removed in a future release.
- **backlight_pwm_frequency** (*int*) – The frequency to use to drive the PWM for backlight brightness control. Default is 50000.

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.

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

root_group: *displayio.Group*

The root group on the display. If the root group is set to *displayio.CIRCUITPYTHON_TERMINAL*, the default CircuitPython terminal will be shown. If the root group is set to None, no output will be shown.

refresh(*, *target_frames_per_second: int | None = None*, *minimum_frames_per_second: int = 0*) → bool

When auto_refresh is off, and target_frames_per_second is not None this 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. The default minimum_frames_per_second of 0 disables this behavior.

When auto_refresh is off, and target_frames_per_second is None this 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** (*Optional[int]*) – The target frame rate that *refresh()* should try to achieve. 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(*y: int*, *buffer: circuitpython_typing.WriteableBuffer*) → circuitpython_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

12.26 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)
i2c.try_lock()
print(i2c.scan())
i2c.unlock()
i2c.deinit()
```

This example will initialize the the device, lock the I2C bus, run *scan()*, unlock the bus, and then *deinit()* the hardware. The last step is optional because CircuitPython automatically resets hardware after a program finishes.

Note that drivers will typically handle communication if provided the bus instance (such as `busio.I2C(board.SCL, board.SDA)`), and that many of the methods listed here are lower level functionalities that are needed for working with custom drivers.

Tutorial for I2C and SPI: https://learn.adafruit.com/circuitpython-basics-i2c-and-spi

Tutorial for UART: https://learn.adafruit.com/circuitpython-essentials/circuitpython-uart-serial

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*)

deinit() → None

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

__enter__() → *I2C*

No-op used in Context Managers.

__exit__() → None

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

scan() → 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() → bool

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

Returns
True when lock has been grabbed

Return type
bool

unlock() → None

Releases the I2C lock.

readfrom_into(*address: int*, *buffer: circuitpython_typing.WriteableBuffer*, ***, *start: int = 0*, *end: int = sys.maxsize*) → None

Read into `buffer` from the device selected by `address`. At least one byte must be read.

If `start` or end is provided, then the buffer will be sliced as if `buffer[start:end]` were passed, but without copying the data. The number of bytes read will be the length of `buffer[start:end]`.

Parameters

- **address** (*int*) – 7-bit device address
- **buffer** (*WriteableBuffer*) – buffer to write into
- **start** (*int*) – beginning of buffer slice
- **end** (*int*) – end of buffer slice; if not specified, use `len(buffer)`

writeto(*address: int*, *buffer: circuitpython_typing.ReadableBuffer*, ***, *start: int = 0*, *end: int = sys.maxsize*) → 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]` were passed, but without copying the data. The number of bytes written will be the length of `buffer[start:end]`.

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*) – beginning of buffer slice
- **end** (*int*) – end of buffer slice; if not specified, use `len(buffer)`

writeto_then_readfrom(*address: int*, *out_buffer: circuitpython_typing.ReadableBuffer*, *in_buffer: circuitpython_typing.WriteableBuffer*, ***, *out_start: int = 0*, *out_end: int = sys.maxsize*, *in_start: int = 0*, *in_end: int = sys.maxsize*) → 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 `out_start` or `out_end` is provided, then the buffer will be sliced as if `out_buffer[out_start:out_end]` were passed, but without copying the data. The number of bytes written will be the length of `out_buffer[start:end]`.

If `in_start` or `in_end` is provided, then the input buffer will be sliced as if `in_buffer[in_start:in_end]` were passed, The number of bytes read will be the length of `out_buffer[in_start:in_end]`.

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*) – beginning of `out_buffer` slice
- **out_end** (*int*) – end of `out_buffer` slice; if not specified, use `len(out_buffer)`
- **in_start** (*int*) – beginning of `in_buffer` slice
- **in_end** (*int*) – end of `in_buffer slice`; if not specified, use `len(in_buffer)`

class busio.**SPI**(*clock:* microcontroller.Pin, *MOSI:* microcontroller.Pin | *None = None*, *MISO:* microcontroller.Pin | *None = None*, *half_duplex: bool = False*)

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.
- **half_duplex** (*bool*) – True when MOSI is used for bidirectional data. False when SPI is full-duplex or simplex.

Limitations: `half_duplex` is available only on STM; other chips do not have the hardware support.

frequency: int

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

deinit() → None

Turn off the SPI bus.

__enter__() → *SPI*

No-op used by Context Managers. Provided by context manager helper.

__exit__() → None

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

configure(*, *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()` → bool

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

Returns
True when lock has been grabbed

Return type
bool

`unlock()` → None

Releases the SPI lock.

`write`(*buffer: circuitpython_typing.ReadableBuffer*, *, *start: int = 0*, *end: int = sys.maxsize*) → None

Write the data contained in `buffer`. The SPI object must be locked. If the buffer is empty, nothing happens.

If `start` or `end` is provided, then the buffer will be sliced as if `buffer[start:end]` were passed, but without copying the data. The number of bytes written will be the length of `buffer[start:end]`.

Parameters

- **buffer** (*ReadableBuffer*) – write out bytes from this buffer
- **start** (*int*) – beginning of buffer slice
- **end** (*int*) – end of buffer slice; if not specified, use `len(buffer)`

`readinto`(*buffer: circuitpython_typing.WriteableBuffer*, *, *start: int = 0*, *end: int = sys.maxsize*, *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.

If `start` or `end` is provided, then the buffer will be sliced as if `buffer[start:end]` were passed. The number of bytes read will be the length of `buffer[start:end]`.

Parameters

- **buffer** (*WriteableBuffer*) – read bytes into this buffer
- **start** (*int*) – beginning of buffer slice
- **end** (*int*) – end of buffer slice; if not specified, it will be the equivalent value of `len(buffer)` and for any value provided it will take the value of `min(end, len(buffer))`
- **write_value** (*int*) – value to write while reading

`write_readinto`(*out_buffer: circuitpython_typing.ReadableBuffer*, *in_buffer: circuitpython_typing.WriteableBuffer*, *, *out_start: int = 0*, *out_end: int = sys.maxsize*, *in_start: int = 0*, *in_end: int = sys.maxsize*) → None

Write out the data in `out_buffer` while simultaneously reading data into `in_buffer`. The SPI object must be locked.

If `out_start` or `out_end` is provided, then the buffer will be sliced as if `out_buffer[out_start:out_end]` were passed, but without copying the data. The number of bytes written will be the length of `out_buffer[out_start:out_end]`.

If `in_start` or `in_end` is provided, then the input buffer will be sliced as if `in_buffer[in_start:in_end]` were passed, The number of bytes read will be the length of `out_buffer[in_start:in_end]`.

The lengths of the slices defined by `out_buffer[out_start:out_end]` and `in_buffer[in_start:in_end]` must be equal. If buffer slice lengths are both 0, nothing happens.

Parameters

- **out_buffer** (*ReadableBuffer*) – write out bytes from this buffer
- **in_buffer** (*WriteableBuffer*) – read bytes into this buffer
- **out_start** (*int*) – beginning of `out_buffer` slice
- **out_end** (*int*) – end of `out_buffer` slice; if not specified, use `len(out_buffer)`
- **in_start** (*int*) – beginning of `in_buffer` slice
- **in_end** (*int*) – end of `in_buffer slice`; if not specified, use `len(in_buffer)`

class busio.**UART**(*tx:* microcontroller.Pin | *None* = *None*, *rx:* microcontroller.Pin | *None* = *None*, **, rts:* microcontroller.Pin | *None* = *None*, *cts:* microcontroller.Pin | *None* = *None*, *rs485_dir:* microcontroller.Pin | *None* = *None*, *rs485_invert: bool* = *False*, *baudrate: int* = *9600*, *bits: int* = *8*, *parity:* Parity | *None* = *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.)

`tx` and `rx` cannot both be `None`.

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.

Limitations: RS485 is not supported on SAMD, nRF, Broadcom, Spresense, or STM. On i.MX and Raspberry Pi RP2040, RS485 support is implemented in software: The timing for the `rs485_dir` pin signal is done on a best-effort basis, and may not meet RS485 specifications intermittently.

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() → None

Deinitialises the UART and releases any hardware resources for reuse.

__enter__() → *UART*

No-op used by Context Managers.

__exit__() → None

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

read(*nbytes: int | None = None*) → bytes | None

Read bytes. 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. If no bytes are read, return `None`.

Note: When no bytes are read due to a timeout, this function returns `None`. This matches the behavior of `io.RawIOBase.read` in Python 3, but differs from pyserial which returns `b''` in that situation.

Returns
Data read

Return type
bytes or None

readinto(*buf: circuitpython_typing.WriteableBuffer*) → int | None

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() → 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(*buf: circuitpython_typing.ReadableBuffer*) → int | None

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() → 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.

12.27 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() → None

De-initialize camera.

take_picture(*buf: circuitpython_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.

12.28 canio – CAN bus access

The *canio* module contains low level classes to support the CAN bus protocol on microcontrollers that have built-in CAN peripherals.

Boards like the Adafruit RP2040 CAN Bus Feather that use an MCP2515 or compatible chip use the `adafruit_mcp2515` module instead.

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.

For more information on working with this module, refer to this Learn Guide on using it.

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() → None

If the device is in the bus off state, restart it.

listen(*matches: Sequence[*Match*] | None = 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(*message:* 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() → None

Deinitialize this object, freeing its hardware resources

__enter__() → *CAN*

Returns self, to allow the object to be used in a `The with statement` statement for resource control

__exit__(*unused1: Type[BaseException] | None*, *unused2: BaseException | None*, *unused3: types.TracebackType | None*) → 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() → *RemoteTransmissionRequest* | *Message* | None

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() → int

Returns the number of messages (including remote transmission requests) waiting

__iter__() → *Listener*

Returns self

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

__next__() → *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() → None

Deinitialize this object, freeing its hardware resources

__enter__() → *CAN*

Returns self, to allow the object to be used in a `The with statement` statement for resource control

__exit__(*unused1: Type[BaseException]* | *None*, *unused2: BaseException* | *None*, *unused3: types.TracebackType* | *None*) → None

Calls deinit()

class canio.**Match**(*id: int*, *, *mask: int* | *None* = *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.

12.29 codeop – Utilities to compile possibly incomplete Python source code.

codeop.**compile_command**(*source: str*, *filename: str = '<input>'*, *symbol: str = 'single'*)

Compile a command and determine whether it is incomplete

The 'completeness' determination is slightly different than in standard Python (it's whatever the internal function `mp_repl_continue_with_input` does). In particular, it's important that the code not end with a newline character or it is likely to be treated as a complete command.

12.30 countio – Support for edge counting

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

For more information on the applications of counting edges, see this Learn Guide on sequential circuits.

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.**Edge**

Enumerates which signal transitions can be counted.

Enum-like class to define which signal transitions to count.

RISE: *Edge*

Count the rising edges.

FALL: *Edge*

Count the falling edges.

RISE_AND_FALL: *Edge*

Count the rising and falling edges.

Limitations: RISE_AND_FALL is not available to RP2040 due to hardware limitations.

class countio.**Counter**(*pin:* microcontroller.Pin, *, *edge:* Edge = *Edge.FALL*, *pull:* digitalio.Pull | *None* = *None*)

Count the number of rising- and/or falling-edge transitions on a given pin.

Create a Counter object associated with the given pin that counts rising- and/or falling-edge transitions. At least one of `rise` and `fall` must be True. The default is to count only falling edges, and is for historical backward compatibility.

Parameters

- **pin** (Pin) – pin to monitor
- **edge** (Edge) – which edge transitions to count
- **pull** (*Optional*[digitalio.Pull]) – enable a pull-up or pull-down if not None

For example:

```
import board
import countio

# Count rising edges only.
pin_counter = countio.Counter(board.D1, edge=countio.Edge.RISE)
# Reset the count after 100 counts.
while True:
    if pin_counter.count >= 100:
        pin_counter.reset()
    print(pin_counter.count)
```

Limitations: On RP2040, *Counter* uses the PWM peripheral, and is limited to using PWM channel B pins due to hardware restrictions. See the pin assignments for your board to see which pins can be used.

count: int

The current count in terms of pulses.

deinit() → None

Deinitializes the Counter and releases any hardware resources for reuse.

__enter__() → *Counter*

No-op used by Context Managers.

__exit__() → None

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

reset() → None

Resets the count back to 0.

12.31 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
import board

pin = digitalio.DigitalInOut(board.LED)
print(pin.value)
```

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

Here is blinky:

```
import time
import digitalio
import board

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

For the essentials of *digitalio*, see the CircuitPython Essentials Learn guide

For more information on using *digitalio*, see this additional Learn guide

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: *Pull* | None

The pin pull direction. One of:

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

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

deinit() → None

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

__enter__() → *DigitalInOut*

No-op used by Context Managers.

__exit__() → None

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

switch_to_output(*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(*pull:* Pull | *None* = *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.

12.32 displayio – High level, display object compositing system

The *displayio* module contains classes to define what objects to display. It is optimized for low memory use and, therefore, computes final pixel values for dirty regions as needed.

Separate modules manage transmitting the display contents to a display.

For more a more thorough explanation and guide for using *displayio*, please refer to this Learn guide.

displayio.**CIRCUITPYTHON_TERMINAL:** *Group*

The *displayio.Group* that is the displayed serial terminal (REPL).

displayio.**release_displays**() → None

Releases any actively used displays so their buses 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)

bits_per_value: int

Bits per Pixel of the bitmap. (read only)

__getitem__(*index: 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__(*index: 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
```

fill(*value: int*) → None

Fills the bitmap with the supplied palette index value.

dirty(*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.

deinit() → None

Release resources allocated by Bitmap.

class displayio.**ColorConverter**(*, *input_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 ColorConverter dithers the output by adding random noise when truncating to display bitdepth

convert(*color: int*) → int

Converts the given color to RGB565 according to the Colorspace

make_transparent(*color: int*) → None

Set the transparent color or index for the ColorConverter. This will raise an Exception if there is already a selected transparent index.

Parameters

color (*int*) – The color to be transparent

make_opaque(*color: int*) → None

Make the ColorConverter be opaque and have no transparent pixels.

Parameters
color (*int*) – [IGNORED] Use any value

class displayio.**Group**(*, *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

- **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 its 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(*layer:* vectorio.Circle | vectorio.Rectangle | vectorio.Polygon | Group | TileGrid) → None

Append a layer to the group. It will be drawn above other layers.

insert(*index: int*, *layer:* vectorio.Circle | vectorio.Rectangle | vectorio.Polygon | Group | TileGrid) → None

Insert a layer into the group.

index(*layer:* vectorio.Circle | vectorio.Rectangle | vectorio.Polygon | Group | TileGrid) → int

Returns the index of the first copy of layer. Raises ValueError if not found.

pop(*i: int = -1*) → *vectorio.Circle* | *vectorio.Rectangle* | *vectorio.Polygon* | *Group* | *TileGrid*

Remove the ith item and return it.

remove(*layer:* vectorio.Circle | vectorio.Rectangle | vectorio.Polygon | Group | TileGrid) → None

Remove the first copy of layer. Raises ValueError if it is not present.

__bool__() → bool

__contains__(*item:* vectorio.Circle | vectorio.Rectangle | vectorio.Polygon | Group | TileGrid) → bool

__iter__() → Iterator[*vectorio.Circle* | *vectorio.Rectangle* | *vectorio.Polygon* | *Group* | *TileGrid*]

__len__() → int

Returns the number of layers in a Group

__getitem__(*index: int*) → *vectorio.Circle* | *vectorio.Rectangle* | *vectorio.Polygon* | *Group* | *TileGrid*

Returns the value at the given index.

This allows you to:

```
print(group[0])
```

__setitem__(*index: int*, *value:* vectorio.Circle | vectorio.Rectangle | vectorio.Polygon | Group | TileGrid) → None

Sets the value at the given index.

This allows you to:

```
group[0] = sprite
```

__delitem__(*index: int*) → None

Deletes the value at the given index.

This allows you to:

```
del group[0]
```

sort(*key: function*, *reverse: bool*) → None

Sort the members of the group.

class displayio.**OnDiskBitmap**(*file: str* | *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.brightness = 0
splash = displayio.Group()
board.DISPLAY.root_group = splash

odb = displayio.OnDiskBitmap('/sample.bmp')
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 name of the bitmap file. For backwards compatibility, a file opened in binary mode may also be passed.

Older versions of CircuitPython required a file opened in binary mode. CircuitPython 7.0 modified OnDiskBitmap so that it takes a filename instead, and opens the file internally. A future version of CircuitPython will remove the ability to pass in an opened file.

width: int

Width of the bitmap. (read only)

height: int

Height of the bitmap. (read only)

pixel_shader: *ColorConverter* | *Palette*

The image's pixel_shader. The type depends on the underlying bitmap's structure. The pixel shader can be modified (e.g., to set the transparent pixel or, for palette shaded images, to update the palette.)

class displayio.**Palette**(*color_count: int*, *, *dither: bool = False*)

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
- **dither** (*bool*) – When true, dither the RGB color before converting to the display's color space

dither: bool

When True the Palette dithers the output color by adding random noise when truncating to display bitdepth

__bool__() → bool

__len__() → int

Returns the number of colors in a Palette

__getitem__(*index: int*) → int | None

Return the pixel color at the given index as an integer.

__setitem__(*index: int*, *value: int | circuitpython_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(*palette_index: int*) → None

make_opaque(*palette_index: int*) → None

is_transparent(*palette_index: int*) → bool

Returns `True` if the palette index is transparent. Returns `False` if opaque.

class displayio.**TileGrid**(*bitmap: Bitmap | OnDiskBitmap*, *, *pixel_shader: ColorConverter | Palette*, *width: int = 1*, *height: int = 1*, *tile_width: int | None = None*, *tile_height: int | None = 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.

To save RAM usage, tile values are only allowed in the range from 0 to 255 inclusive (single byte values).

tile_width and tile_height match the height of the bitmap by default.

Parameters

- **bitmap** (`Bitmap`,`OnDiskBitmap`) – 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.

width: `int`

Width of the tilegrid in tiles.

height: `int`

Height of the tilegrid in tiles.

tile_width: `int`

Width of a single tile in pixels.

tile_height: *int*

Height of a single tile in pixels.

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: *ColorConverter* | *Palette*

The pixel shader of the tilegrid.

bitmap: *Bitmap* | *OnDiskBitmap*

The bitmap of the tilegrid.

contains(*touch_tuple: tuple*) → bool

Returns True if the first two values in `touch_tuple` represent an x,y coordinate inside the tilegrid rectangle bounds.

__getitem__(*index: 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__(*index: 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
```

12.33 dotclockframebuffer – Native helpers for driving parallel displays

dotclockframebuffer.**Length**

dotclockframebuffer.**ioexpander_send_init_sequence**(*bus:* busio.I2C, *init_sequence: circuitpython_typing.ReadableBuffer*, **, i2c_init_sequence: circuitpython_typing.ReadableBuffer, i2c_address: int, gpio_address: int, gpio_data_len:* Length, *gpio_data: int, cs_bit: int, mosi_bit: int, clk_bit: int, reset_bit: int | None*)

Send a displayio-style initialization sequence over an I2C I/O expander

This function is highly generic in order to support various I/O expanders. What's assumed is that all the GPIO can be updated by writing to a single I2C register. Normal output polarity is assumed (CS and CLK are active low, data is not inverted). Only 8-bit I2C addresses are supported. 8- and 16-bit I2C addresses and data registers are supported. The Data/Command bit is sent as part of the serial data.

Normally this function is used via a convenience library that is specific to the display & I/O expander in use.

If the board has an integrated I/O expander, `**board.TFT_IO_EXPANDER` expands to the proper arguments starting with `gpio_address`. Note that this may include the `i2c_init_sequence` argument which can change the direction & value of I/O expander pins. If this is undesirable, take a copy of `TFT_IO_EXPANDER` and change or remove the `i2c_init_sequence` key.

If the board has an integrated display that requires an initialization sequence, `board.TFT_INIT_SEQUENCE` is the initialization string for the display.

Parameters

- **bus** (busio.I2C) – The I2C bus where the I/O expander resides
- **busio.i2c_address** (*int*) – The I2C bus address of the I/O expander
- **init_sequence** (*ReadableBuffer*) – The initialization sequence to send to the display
- **gpio_address** (*int*) – The address portion of the I2C transaction (1 byte)
- **gpio_data_len** (*int*) – The size of the data portion of the I2C transaction, 1 or 2 bytes
- **gpio_data** (*int*) – The output value for all GPIO bits other than cs, mosi, and clk (needed because GPIO expanders may be unable to read back the current output value)
- **cs_bit** (*int*) – The bit number (from 0 to 7, or from 0 to 15) of the chip select bit in the GPIO register
- **mosi_value** (*int*) – The bit number (from 0 to 7, or from 0 to 15) of the data out bit in the GPIO register
- **clk_value** (*int*) – The bit number (from 0 to 7, or from 0 to 15) of the clock out bit in the GPIO register
- **reset_value** (*Optional[int]*) – The bit number (from 0 to 7, or from 0 to 15) of the display reset bit in the GPIO register
- **i2c_init_sequence** (*Optional[ReadableBuffer]*) – An initialization sequence to send to the I2C expander

class dotclockframebuffer.**DotClockFramebuffer**(*, *de:* microcontroller.Pin, *vsync:* microcontroller.Pin, *hsync:* microcontroller.Pin, *dclk:* microcontroller.Pin, *red: Tuple[*microcontroller.Pin*]*, *green: Tuple[*microcontroller.Pin*]*, *blue: Tuple[*microcontroller.Pin*]*, *frequency: int*, *width: int*, *height: int*, *hsync_pulse_width: int*, *hsync_back_porch: int*, *hsync_front_porch: int*, *hsync_idle_low: bool*, *vsync_back_porch: int*, *vsync_front_porch: int*, *vsync_idle_low: bool*, *de_idle_high: bool*, *pclk_active_high: bool*, *pclk_idle_high: bool*, *overscan_left: int = 0*)

Manage updating a 'dot-clock' framebuffer in the background while Python code runs. It doesn't handle display initialization.

Create a DotClockFramebuffer object associated with the given pins.

The 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.

When a board has dedicated dot clock framebuffer pins and/or timings, they are intended to be used in the constructor with ** dictionary unpacking like so: `DotClockFramebuffer(**board.TFT_PINS, **board.TFT_TIMINGS)`

On Espressif-family microcontrollers, this driver requires that the `CIRCUITPY_RESERVED_PSRAM` in `settings.toml` be large enough to hold the framebuffer. Generally, boards with built-in displays or display connectors will have a default setting that is large enough for typical use. If the constructor raises a MemoryError or an IDFError, this probably indicates the setting is too small and should be increased.

TFT connection parameters:

Parameters

- **de** (microcontroller.Pin) – The "data enable" input to the display
- **vsync** (microcontroller.Pin) – The "vertical sync" input to the display
- **hsync** (microcontroller.Pin) – The "horizontal sync" input to the display
- **dclk** (microcontroller.Pin) – The "data clock" input to the display
- **red** (*~tuple*) – The red data pins, most significant pin first.
- **green** (*~tuple*) – The green data pins, most significant pin first.
- **blue** (*~tuple*) – The blue data pins, most significant pin first.

TFT timing parameters:

Parameters

- **frequency** (*int*) – The requested data clock frequency in Hz.
- **width** (*int*) – The visible width of the display, in pixels
- **height** (*int*) – The visible height of the display, in pixels
- **hsync_pulse_width** (*int*) – Horizontal sync width in pixels
- **hsync_back_porch** (*int*) – Horizontal back porch, number of pixels between hsync and start of line active data
- **hsync_front_porch** (*int*) – Horizontal front porch, number of pixels between the end of active data and the next hsync
- **vsync_back_porch** (*int*) – Vertical back porch, number of lines between vsync and start of frame
- **vsync_front_porch** (*int*) – Vertical front porch, number of lines between the end of frame and the next vsync
- **hsync_idle_low** (*bool*) – True if the hsync signal is low in IDLE state
- **vsync_idle_low** (*bool*) – True if the vsync signal is low in IDLE state
- **de_idle_high** (*bool*) – True if the de signal is high in IDLE state
- **pclk_active_high** (*bool*) – True if the display data is clocked out at the rising edge of dclk
- **pclk_idle_high** (*bool*) – True if the dclk stays at high level in IDLE phase

- **overscan_left** (*int*) – Allocate additional non-visible columns left of the first display column

refresh_rate: float

The pixel refresh rate of the display, in Hz

frequency: int

The pixel frequency of the display, in Hz

width: int

The width of the display, in pixels

height: int

The height of the display, in pixels

row_stride: int

The row_stride of the display, in bytes

Due to overscan or alignment requirements, the memory address for row N+1 may not be exactly `2*width` bytes after the memory address for row N. This property gives the stride in **bytes**.

On Espressif this value is **guaranteed** to be a multiple of the 2 (i.e., it is a whole number of pixels)

first_pixel_offset: int

The first_pixel_offset of the display, in bytes

Due to overscan or alignment requirements, the memory address for row N+1 may not be exactly `2*width` bytes after the memory address for row N. This property gives the stride in **bytes**.

On Espressif this value is **guaranteed** to be a multiple of the 2 (i.e., it is a whole number of pixels)

refresh() → None

Transmits the color data in the buffer to the pixels so that they are shown.

If this function is not called, the results are unpredictable; updates may be partially shown.

12.34 dualbank – Dualbank Module

The *dualbank* module adds ability to update and switch between the two identical app partitions, which can contain different firmware versions.

Having two partitions enables rollback functionality.

The two partitions are defined as the boot partition and the 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.

Use cases:

- Can be used for `OTA` Over-The-Air updates.
- Can be used for `dual-boot` of different firmware versions or platforms.

Note:

Boards with flash =2MB:
This module is unavailable as the flash is only large enough for one app partition.

Boards with flash >2MB:
This module is enabled/disabled at runtime based on whether the CIRCUITPY drive is extended or not. See *storage.erase_filesystem()* for more information.

```
import dualbank

dualbank.flash(buffer, offset)
dualbank.switch()
```

dualbank.**flash**(*buffer: circuitpython_typing.ReadableBuffer*, *offset: int = 0*) → None

Writes one of the two app partitions at the given offset.

This can be called multiple times when flashing the firmware in smaller chunks.

Parameters

- **buffer** (*ReadableBuffer*) – The entire firmware or a partial chunk.
- **offset** (*int*) – Start writing at this offset in the app partition.

dualbank.**switch**() → None

Switches to the next-update partition.

On next reset, firmware will be loaded from the partition just switched over to.

12.35 epaperdisplay

Displays a *displayio* object tree on an e-paper display

class epaperdisplay.**EPaperDisplay**(*display_bus: busdisplay._DisplayBus*, *start_sequence: circuitpython_typing.ReadableBuffer*, *stop_sequence: circuitpython_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: int | None = None*, *set_row_window_command: int | None = None*, *set_current_column_command: int | None = None*, *set_current_row_command: int | None = None*, *write_black_ram_command: int*, *black_bits_inverted: bool = False*, *write_color_ram_command: int | None = None*, *color_bits_inverted: bool = False*, *highlight_color: int = 0*, *refresh_display_command: int | circuitpython_typing.ReadableBuffer*, *refresh_time: float = 40*, *busy_pin: microcontroller.Pin | None = None*, *busy_state: bool = True*, *seconds_per_frame: float = 180*, *always_toggle_chip_select: bool = False*, *grayscale: bool = False*, *advanced_color_epaper: bool = False*, *two_byte_sequence_length: bool = False*, *start_up_time: float = 0*, *address_little_endian: 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 (*fourwire.FourWire* or *parallelbus.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 command sequence.
- **stop_sequence** (*ReadableBuffer*) – Byte-packed command 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. Single int or byte-packed command sequence
- **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
- **advanced_color_epaper** (*bool*) – When true, the display is a 7-color advanced color epaper (ACeP)
- **two_byte_sequence_length** (*bool*) – When true, use two bytes to define sequence length
- **start_up_time** (*float*) – Time to wait after reset before sending commands
- **address_little_endian** (*bool*) – Send the least significant byte (not bit) of multi-byte addresses first. Ignored when ram is addressed with one byte

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: *busdisplay._DisplayBus*

The bus being used by the display

root_group: *displayio.Group*

The root group on the epaper display. If the root group is set to *displayio.CIRCUITPYTHON_TERMINAL*, the default CircuitPython terminal will be shown. If the root group is set to `None`, no output will be shown.

update_refresh_mode(*start_sequence: circuitpython_typing.ReadableBuffer*, *seconds_per_frame: float = 180*) → None

Updates the `start_sequence` and `seconds_per_frame` parameters to enable varying the refresh mode of the display.

refresh() → 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.

12.36 espcamera – Wrapper for the espcamera library

This library enables access to any camera sensor supported by the library, including OV5640 and OV2640.

See also:

Non-Espressif microcontrollers use the *imagecapture* module together with wrapper libraries such as adafruit_ov5640.

class espcamera.**GrabMode**

Controls when a new frame is grabbed.

WHEN_EMPTY: *GrabMode*

Fills buffers when they are empty. Less resources but first `fb_count` frames might be old

LATEST: *GrabMode*

Except when 1 frame buffer is used, queue will always contain the last `fb_count` frames

class espcamera.**PixelFormat**

Format of data in the captured frames

RGB565: *PixelFormat*

A 16-bit format with 5 bits of Red and Blue and 6 bits of Green

GRAYSCALE: *PixelFormat*

An 8-bit format with 8-bits of luminance

JPEG: *PixelFormat*

A compressed format

class espcamera.**FrameSize**

The pixel size of the captured frames

R96X96: *FrameSize*

96x96

QQVGA: *FrameSize*

160x120

QCIF: *FrameSize*

176x144

HQVGA: *FrameSize*

240x176

R240X240: *FrameSize*

240x240

QVGA: *FrameSize*

320x240

CIF: *FrameSize*

400x296

HVGA: *FrameSize*

480x320

VGA: *FrameSize*

640x480

SVGA: *FrameSize*

800x600

XGA: *FrameSize*

1024x768

HD: *FrameSize*

1280x720

SXGA: *FrameSize*

1280x1024

UXGA: *FrameSize*
1600x1200

FHD: *FrameSize*
1920x1080

P_HD: *FrameSize*
720x1280

P_3MP: *FrameSize*
864x1536

QXGA: *FrameSize*
2048x1536

QHD: *FrameSize*
2560x1440

WQXGA: *FrameSize*
2560x1600

P_FHD: *FrameSize*
1080x1920

QSXGA: *FrameSize*
2560x1920

class espcamera.**GainCeiling**

The maximum amount of gain applied to raw sensor data.

Higher values are useful in darker conditions, but increase image noise.

GAIN_2X: *GainCeiling*

GAIN_4X: *GainCeiling*

GAIN_8X: *GainCeiling*

GAIN_16X: *GainCeiling*

GAIN_32X: *GainCeiling*

GAIN_64X: *GainCeiling*

GAIN_128X: *GainCeiling*

class espcamera.**Camera**(*, *data_pins: List[*microcontroller.Pin*]*, *pixel_clock_pin:* microcontroller.Pin, *vsync_pin:* microcontroller.Pin, *href_pin:* microcontroller.Pin, *i2c:* busio.I2C, *external_clock_pin:* microcontroller.Pin | *None* = *None*, *external_clock_frequency: int* = *20000000*, *powerdown_pin:* microcontroller.Pin | *None* = *None*, *reset_pin:* microcontroller.Pin | *None* = *None*, *pixel_format:* PixelFormat = *PixelFormat.RGB565*, *frame_size:* FrameSize = *FrameSize.QQVGA*, *jpeg_quality: int* = *15*, *framebuffer_count: int* = *1*, *grab_mode:* GrabMode = *GrabMode.WHEN_EMPTY*)

Configure and initialize a camera with the given properties

Important: Not all supported sensors have all of the properties listed below. For instance, the OV5640 supports *denoise*, but the OV2640 does not. The underlying esp32-camera library does not provide a reliable API

to check which settings are supported. CircuitPython makes a best effort to determine when an unsupported property is set and will raise an exception in that case.

Parameters

- **data_pins** – The 8 data data_pins used for image data transfer from the camera module, least significant bit first
- **pixel_clock_pin** – The pixel clock output from the camera module
- **vsync_pin** – The vertical sync pulse output from the camera module
- **href_pin** – The horizontal reference output from the camera module
- **i2c** – The I2C bus connected to the camera module
- **external_clock_pin** – The pin on which to generate the external clock
- **external_clock_frequency** – The frequency generated on the external clock pin
- **powerdown_pin** – The powerdown input to the camera module
- **reset_pin** – The reset input to the camera module
- **pixel_format** – The pixel format of the captured image
- **frame_size** – The size of captured image
- **jpeg_quality** – For *PixelFormat.JPEG*, the quality. Higher numbers increase quality. If the quality is too high, the JPEG data will be larger than the available buffer size and the image will be unusable or truncated. The exact range of appropriate values depends on the sensor and must be determined empirically.
- **framebuffer_count** – The number of framebuffers (1 for single-buffered and 2 for double-buffered)
- **grab_mode** – When to grab a new frame

frame_available: bool

True if a frame is available, False otherwise

pixel_format: *PixelFormat*

The pixel format of captured frames

frame_size: *FrameSize*

The size of captured frames

contrast: int

The sensor contrast. Positive values increase contrast, negative values lower it. The total range is device-specific but is often from -2 to +2 inclusive.

brightness: int

The sensor brightness. Positive values increase brightness, negative values lower it. The total range is device-specific but is often from -2 to +2 inclusive.

saturation: int

The sensor saturation. Positive values increase saturation (more vibrant colors), negative values lower it (more muted colors). The total range is device-specific but the value is often from -2 to +2 inclusive.

sharpness: int

The sensor sharpness. Positive values increase sharpness (more defined edges), negative values lower it (softer edges). The total range is device-specific but the value is often from -2 to +2 inclusive.

denoise: int

The sensor 'denoise' setting. Any camera sensor has inherent 'noise', especially in low brightness environments. Software algorithms can decrease noise at the expense of fine detail. A larger value increases the amount of software noise removal. The total range is device-specific but the value is often from 0 to 10.

gain_ceiling: *GainCeiling*

The sensor 'gain ceiling' setting. "Gain" is an analog multiplier applied to the raw sensor data. The 'ceiling' is the maximum gain value that the sensor will use. A higher gain means that the sensor has a greater response to light, but also makes sensor noise more visible.

quality: int

The 'quality' setting when capturing JPEG images. This is similar to the quality setting when exporting a jpeg image from photo editing software. Typical values range from 5 to 40, with higher numbers leading to larger image sizes and better overall image quality. However, when the quality is set to a high number, the total size of the JPEG data can exceed the size of an internal buffer, causing image capture to fail.

colorbar: bool

When `True`, a test pattern image is captured and the real sensor data is not used.

whitebal: bool

When `True`, the camera attempts to automatically control white balance. When `False`, the *wb_mode* setting is used instead.

gain_ctrl: bool

When `True`, the camera attempts to automatically control the sensor gain, up to the value in the *gain_ceiling* property. When `False`, the *agc_gain* setting is used instead.

exposure_ctrl: bool

When `True` the camera attempts to automatically control the exposure. When `False`, the *aec_value* setting is used instead.

hmirror: bool

When `True` the camera image is mirrored left-to-right

vflip: bool

When `True` the camera image is flipped top-to-bottom

aec2: bool

When `True` the sensor's "night mode" is enabled, extending the range of automatic gain control.

awb_gain: bool

Access the awb_gain property of the camera sensor

agc_gain: int

Access the gain level of the sensor. Higher values produce brighter images. Typical settings range from 0 to 30.

aec_value: int

Access the exposure value of the camera. Higher values produce brighter images. Typical settings range from 0 to 1200.

special_effect: int

Enable a "special effect". Zero is no special effect. On OV5640, special effects range from 0 to 6 inclusive and select various color modes.

wb_mode: *int*

The white balance mode. 0 is automatic white balance. Typical values range from 0 to 4 inclusive.

ae_level: *int*

The exposure offset for automatic exposure. Typical values range from -2 to +2.

dcw: *bool*

When `True` an advanced white balance mode is selected.

bpc: *bool*

When `True`, "black point compensation" is enabled. This can make black parts of the image darker.

wpc: *bool*

When `True`, "white point compensation" is enabled. This can make white parts of the image whiter.

raw_gma: *bool*

When `True`, raw gamma mode is enabled.

lenc: *bool*

Enable "lens correction". This can help compensate for light fall-off at the edge of the sensor area.

max_frame_size: *FrameSize*

The maximum frame size that can be captured

address: *int*

The I2C (SCCB) address of the sensor

sensor_name: *str*

The name of the sensor

supports_jpeg: *bool*

True if the sensor can capture images in JPEG format

height: *int*

The height of the image being captured

width: *int*

The width of the image being captured

grab_mode: *GrabMode*

The grab mode of the camera

framebuffer_count: *int*

True if double buffering is used

deinit() → None

Deinitialises the camera and releases all memory resources for reuse.

__enter__() → *Camera*

No-op used by Context Managers.

__exit__() → None

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

take(*timeout: float | None = 0.25*) → *displayio.Bitmap* | circuitpython_typing.ReadableBuffer | None

Record a frame. Wait up to 'timeout' seconds for a frame to be captured.

In the case of timeout, None is returned. If *pixel_format* is *PixelFormat.JPEG*, the returned value is a read-only *memoryview*. Otherwise, the returned value is a read-only *displayio.Bitmap*.

reconfigure(*frame_size:* FrameSize | *None = None*, *pixel_format:* PixelFormat | *None = None*, *grab_mode:* GrabMode | *None = None*, *framebuffer_count: int | None = None*) → None

Change multiple related camera settings simultaneously

Because these settings interact in complex ways, and take longer than the other properties to set, they are set together in a single function call.

If an argument is unspecified or None, then the setting is unchanged.

12.37 espidf – Return the total size of the ESP-IDF, which includes the CircuitPython heap.

espidf.**heap_caps_get_total_size**() → int

espidf.**heap_caps_get_free_size**() → int

Return total free memory in the ESP-IDF heap.

espidf.**heap_caps_get_largest_free_block**() → int

Return the size of largest free memory block in the ESP-IDF heap.

espidf.**erase_nvs**() → None

Erase all data in the non-volatile storage (nvs), including data stored by with *microcontroller.nvm*

This is necessary when upgrading from CircuitPython 6.3.0 or earlier to CircuitPython 7.0.0, because the layout of data in nvs has changed. The old data will be lost when you perform this operation.

exception espidf.**IDFError**

Bases: OSError

Raised when an ESP-IDF function returns an error code. esp_err_t

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

exception espidf.**MemoryError**

Bases: *MemoryError*

Raised when an ESP-IDF memory allocation fails.

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

espidf.**get_total_psram**() → int

Returns the number of bytes of psram detected, or 0 if psram is not present or not configured

12.38 espnow – ESP-NOW Module

The *espnow* module provides an interface to the ESP-NOW protocol provided by Espressif on its SoCs.

Sender

```
import espnow

e = espnow.ESPNow()
peer = espnow.Peer(mac=b'aaaaaa')
e.peers.append(peer)

e.send("Starting...")
for i in range(10):
    e.send(str(i)*20)
e.send(b'end')
```

Receiver

```
import espnow

e = espnow.ESPNow()
packets = []

while True:
    if e:
        packet = e.read()
        packets.append(packet)
        if packet.msg == b'end':
            break

print("packets:", f"length={len(packets)}")
for packet in packets:
    print(packet)
```

class espnow.**ESPNow**(*buffer_size: int = 526*, *phy_rate: int = 0*)

Provides access to the ESP-NOW protocol.

Allocate and initialize *ESPNow* instance as a singleton.

Parameters

- **buffer_size** (*int*) – The size of the internal ring buffer. Default: 526 bytes.
- **phy_rate** (*int*) – The ESP-NOW physical layer rate. Default: 1 Mbps. wifi_phy_rate_t

send_success: int

The number of tx packets received by the peer(s) ESP_NOW_SEND_SUCCESS. (read-only)

send_failure: int

The number of failed tx packets ESP_NOW_SEND_FAIL. (read-only)

read_success: int

The number of rx packets captured in the buffer. (read-only)

read_failure: int

The number of dropped rx packets due to buffer overflow. (read-only)

buffer_size: *int*

The size of the internal ring buffer. (read-only)

phy_rate: *int*

The ESP-NOW physical layer rate. wifi_phy_rate_t

peers: *Peers*

The peer info records for all registered *ESPNow* peers. (read-only)

deinit() → None

Deinitializes ESP-NOW and releases it for another program.

__enter__() → *ESPNow*

No-op used by Context Managers.

__exit__() → None

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

send(*message: circuitpython_typing.ReadableBuffer*, *peer:* Peer | *None* = *None*) → None

Send a message to the peer's mac address.

This blocks until a timeout of 2 seconds if the ESP-NOW internal buffers are full.

Parameters

- **message** (*ReadableBuffer*) – The message to send (length <= 250 bytes).
- **peer** (Peer) – Send message to this peer. If None, send to all registered peers.

read() → *ESPNowPacket* | None

Read a packet from the receive buffer.

This is non-blocking, the packet is received asynchronously from the peer(s).

Returns

An *ESPNowPacket* if available in the buffer, otherwise None.

set_pmk(*pmk: circuitpython_typing.ReadableBuffer*) → None

Set the ESP-NOW Primary Master Key (pmk) for encrypted communications.

Parameters

pmk (*ReadableBuffer*) – The ESP-NOW Primary Master Key (length = 16 bytes).

__bool__() → bool

True if *len()* is greater than zero. This is an easy way to check if the buffer is empty.

__len__() → int

Return the number of *bytes* available to read. Used to implement len().

class espnow.**ESPNowPacket**

A packet retrieved from ESP-NOW communication protocol. A namedtuple.

mac: *circuitpython_typing.ReadableBuffer*

The sender's mac address (length = 6 bytes).

msg: *circuitpython_typing.ReadableBuffer*

The message sent by the peer (length <= 250 bytes).

rssi: *int*

The received signal strength indication (in dBm from -127 to 0).

time: *int*

The time in milliseconds since the device last booted when the packet was received.

class espnow.**Peer**(*mac: bytes*, *, *lmk: bytes | None*, *channel: int = 0*, *interface: int = 0*, *encrypted: bool = False*)

A data class to store parameters specific to a peer.

Construct a new peer object.

Parameters

- **mac** (*bytes*) – The mac address of the peer.
- **lmk** (*bytes*) – The Local Master Key (lmk) of the peer.
- **channel** (*int*) – The peer's channel. Default: 0 ie. use the current channel.
- **interface** (*int*) – The WiFi interface to use. Default: 0 ie. STA.
- **encrypted** (*bool*) – Whether or not to use encryption.

mac: *circuitpython_typing.ReadableBuffer*

The WiFi mac to use.

lmk: *circuitpython_typing.ReadableBuffer*

The WiFi lmk to use.

channel: *int*

The WiFi channel to use.

interface: *int*

The WiFi interface to use.

encrypted: *bool*

Whether or not to use encryption.

class espnow.**Peers**

Maintains a *list* of *Peer* internally and only exposes a subset of *list* methods.

You cannot create an instance of *Peers*.

append(*peer:* Peer) → None

Append peer.

Parameters

peer (Peer) – The peer object to append.

remove(*peer:* Peer) → None

Remove peer.

Parameters

peer (Peer) – The peer object to remove.

12.39 espulp – ESP Ultra Low Power Processor Module

The *espulp* module adds ability to load and run programs on the ESP32-Sx's ultra-low-power RISC-V processor.

```
import espulp
import memorymap

shared_mem = memorymap.AddressRange(start=0x50000000, length=1024)
ulp = espulp.ULP()

with open("program.bin", "rb") as f:
    program = f.read()

ulp.run(program)
print(shared_mem[0])
# ulp.halt()
```

espulp.**get_rtc_gpio_number**(*pin:* microcontroller.Pin) → int | None

Return the RTC GPIO number of the given pin or None if not connected to RTC GPIO.

class espulp.**Architecture**

The ULP architectures available.

FSM: *Architecture*

The ULP Finite State Machine.

RISCV: *Architecture*

The ULP RISC-V Coprocessor.

class espulp.**ULP**(*arch:* Architecture = *Architecture.FSM*)

The ultra-low-power processor.

Raises an exception if another ULP has been instantiated. This ensures that is is only used by one piece of code at a time.

Parameters
arch (Architecture) – The ulp arch

arch: *Architecture*

The ulp architecture. (read-only)

deinit() → None

Deinitialises the ULP and releases it for another program.

__enter__() → *ULP*

No-op used by Context Managers.

__exit__() → None

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

run(*program: circuitpython_typing.ReadableBuffer*, *, *pins: Sequence[*microcontroller.Pin*] = ()*) → None

Loads the program into ULP memory and then runs the program. The given pins are claimed and not reset until *halt()* is called.

The program will continue to run even when the running Python is halted.

halt() → None

Halts the running program and releases the pins given in *run()*.

class espulp.**ULPAlarm**(*ulp:* ULP)

Trigger an alarm when the ULP requests wake-up.

Create an alarm that will be triggered when the ULP requests wake-up.

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

ulp (ULP) – The ulp instance

12.40 floppyio – Read flux transition information into the buffer.

floppyio.**flux_readinto**(*buffer:* *circuitpython_typing.WriteableBuffer*, *data:* digitalio.DigitalInOut, *index:* digitalio.DigitalInOut) → int

The function returns when the buffer has filled, or when the index input indicates that one full revolution of data has been recorded. Due to technical limitations, this process may not be interruptible by KeyboardInterrupt.

Parameters

- **buffer** – Read data into this buffer. Each element represents the time between successive zero-to-one transitions.
- **data** – Pin on which the flux data appears
- **index** – Pin on which the index pulse appears

Returns

The actual number of bytes of read

floppyio.**mfm_readinto**(*buffer:* *circuitpython_typing.WriteableBuffer*, *data:* digitalio.DigitalInOut, *index:* digitalio.DigitalInOut) → int

Read mfm blocks into the buffer.

The track is assumed to consist of 512-byte sectors.

The function returns when all sectors have been successfully read, or a number of index pulses have occurred. Due to technical limitations, this process may not be interruptible by KeyboardInterrupt.

Parameters

- **buffer** – Read data into this buffer. Must be a multiple of 512.
- **data** – Pin on which the mfm data appears
- **index** – Pin on which the index pulse appears

Returns

The actual number of sectors read

floppyio.**samplerate**: int

The approximate sample rate in Hz used by flux_readinto.

12.41 fontio – Core font related data structures

Note: This module is intended only for low-level usage. For working with fonts in CircuitPython see the adafruit_bitmap_font library. For information on creating custom fonts for use in CircuitPython, see this Learn guide

class fontio.**FontProtocol**

Bases: typing_extensions.Protocol

A protocol shared by *BuiltinFont* and classes in adafruit_bitmap_font

get_bounding_box() → Tuple[int, int] | Tuple[int, int, int, int]

Retrieve the maximum bounding box of any glyph in the font.

The four element version is (width, height, x_offset, y_offset). The two element version is (width, height), in which x_offset and y_offset are assumed to be zero.

get_glyph(*codepoint: int*) → *Glyph* | None

Retrieve the Glyph for a given code point

If the code point is not present in the font, None is returned.

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() → Tuple[int, int]

Returns the maximum bounds of all glyphs in the font in a tuple of two values: width, height.

get_glyph(*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

12.42 fourwire – Connects to a BusDisplay over a four wire bus

class fourwire.**FourWire**(*spi_bus:* busio.SPI, *, *command:* microcontroller.Pin | *None*, *chip_select:* microcontroller.Pin, *reset:* microcontroller.Pin | *None* = *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.

If the `command` pin is not specified, a 9-bit SPI mode will be simulated by adding a data/command bit to every bit being transmitted, and splitting the resulting data back into 8-bit bytes for transmission. The extra bits that this creates at the end are ignored by the receiving device.

Parameters

- **spi_bus** (busio.SPI) – The SPI bus that make up the clock and data lines
- **command** (microcontroller.Pin) – Data or command pin. When None, 9-bit SPI is simulated.
- **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() → None

Performs a hardware reset via the reset pin. Raises an exception if called when no reset pin is available.

send(*command:* *int*, *data:* *circuitpython_typing.ReadableBuffer*, *, *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.

12.43 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:* *circuitpython_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.

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: circuitpython_typing.FrameBuffer

The framebuffer being used by the display

root_group: *displayio.Group*

The root group on the display. If the root group is set to *displayio.CIRCUITPYTHON_TERMINAL*, the default CircuitPython terminal will be shown. If the root group is set to None, no output will be shown.

refresh(*, *target_frames_per_second: int | None = None*, *minimum_frames_per_second: int = 0*) → bool

When auto_refresh is off, and target_frames_per_second is not None this 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. The default minimum_frames_per_second of 0 disables this behavior.

When auto_refresh is off, and target_frames_per_second is None this 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** (*Optional[int]*) – The target frame rate that *refresh()* should try to achieve. 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(*y: int*, *buffer: circuitpython_typing.WriteableBuffer*) → circuitpython_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

12.44 frequencyio – Support for frequency based protocols

Warning: This module is not available in SAMD21 builds. See the module-support-matrix for more info.

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

For example:

```
import time
import frequencyio
import board

frequency = frequencyio.FrequencyIn(board.D11)
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)

    # Optional clear() will reset the value
    # to zero. Without this, if the incoming
    # signal stops, the last reading will remain
```

(continues on next page)

(continued from previous page)

```
    # 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() → None

Deinitialises the FrequencyIn and releases any hardware resources for reuse.

__enter__() → *FrequencyIn*

No-op used by Context Managers.

__exit__() → None

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

pause() → None

Pause frequency capture.

resume() → None

Resumes frequency capture.

clear() → None

Clears the last detected frequency capture value.

__get__(*index: int*) → int

Returns the value of the last frequency captured.

12.45 getpass – Getpass Module

This module provides a way to get input from user without echoing it.

getpass.**getpass**(*prompt: str | None = 'Password: '*, *stream:* io.FileIO | *None = None*) → str

Prompt the user without echoing.

Parameters

- **prompt** (*str*) – The user is prompted using the string `prompt`, which defaults to `'Password: '`.
- **stream** (`io.FileIO`) – The `prompt` is written to the file-like object `stream` if provided.

12.46 gifio – Access GIF-format images

class gifio.**GifWriter**(*file: BinaryIO | str*, *width: int*, *height: int*, *colorspace:* displayio.Colorspace, *loop: bool = True*, *dither: bool = False*)

Construct a GifWriter object

Parameters

- **file** – Either a file open in bytes mode, or the name of a file to open in bytes mode.
- **width** – The width of the image. All frames must have the same width.
- **height** – The height of the image. All frames must have the same height.
- **colorspace** – The colorspace of the image. All frames must have the same colorspace. The supported colorspaces are `RGB565`, `BGR565`, `RGB565_SWAPPED`, `BGR565_SWAPPED`, and `L8` (greyscale)
- **loop** – If True, the GIF is marked for looping playback
- **dither** – If True, and the image is in color, a simple ordered dither is applied.

__enter__() → *GifWriter*

No-op used by Context Managers.

__exit__() → None

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

deinit() → None

Close the underlying file.

add_frame(*bitmap: circuitpython_typing.ReadableBuffer*, *delay: float = 0.1*) → None

Add a frame to the GIF.

Parameters

- **bitmap** – The frame data
- **delay** – The frame delay in seconds. The GIF format rounds this to the nearest 1/100 second, and the largest permitted value is 655 seconds.

class gifio.**OnDiskGif**(*file: str*)

Loads one frame of a GIF into memory at a time.

The code can be used in cooperation with displayio but this mode is relatively slow:

```
import board
import gifio
import displayio
import time

display = board.DISPLAY
splash = displayio.Group()
display.root_group = splash

odg = gifio.OnDiskGif('/sample.gif')

start = time.monotonic()
```

(continues on next page)

(continued from previous page)

```
next_delay = odg.next_frame() # Load the first frame
end = time.monotonic()
overhead = end - start

face = displayio.TileGrid(
    odg.bitmap,
    pixel_shader=displayio.ColorConverter(
        input_colorspace=displayio.Colorspace.RGB565_SWAPPED
    ),
)
splash.append(face)
board.DISPLAY.refresh()

# Display repeatedly.
while True:
    # Sleep for the frame delay specified by the GIF,
    # minus the overhead measured to advance between frames.
    time.sleep(max(0, next_delay - overhead))
    next_delay = odg.next_frame()
```

The displayio Group and TileGrid layers can be bypassed and the image can be directly blitted to the full screen. This can give a speed-up of ~4x to ~6x depending on the GIF and display. This requires an LCD that uses standard codes to set the update area, and which accepts RGB565_SWAPPED pixel data directly:

```
# Initial set-up the same as above

# Take over display to drive directly
display.auto_refresh = False
display_bus = display.bus

# Display repeatedly & directly.
while True:
    # Sleep for the frame delay specified by the GIF,
    # minus the overhead measured to advance between frames.
    time.sleep(max(0, next_delay - overhead))
    next_delay = odg.next_frame()

    display_bus.send(42, struct.pack(">hh", 0, odg.bitmap.width - 1))
    display_bus.send(43, struct.pack(">hh", 0, odg.bitmap.height - 1))
    display_bus.send(44, odg.bitmap)

# The following optional code will free the OnDiskGif and allocated resources
# after use. This may be required before loading a new GIF in situations
# where RAM is limited and the first GIF took most of the RAM.
odg.deinit()
odg = None
gc.collect()
```

Create an *OnDiskGif* object with the given file. The GIF frames are decoded into RGB565 big-endian format. *displayio* expects little-endian, so the example above uses *Colorspace.RGB565_SWAPPED*.

Parameters

file (*file*) – The name of the GIF file.

If the image is too large it will be cropped at the bottom and right when displayed.

Limitations: The image width is limited to 320 pixels at present. *ValueError* will be raised if the image is too wide. The height is not limited but images that are too large will cause a memory exception.

width: int

Width of the gif. (read only)

height: int

Height of the gif. (read only)

bitmap: *displayio.Bitmap*

The bitmap used to hold the current frame.

palette: *displayio.Palette* | None

The palette for the current frame if it exists.

duration: float

Returns the total duration of the GIF in seconds. (read only)

frame_count: int

Returns the number of frames in the GIF. (read only)

min_delay: float

The minimum delay found between frames. (read only)

max_delay: float

The maximum delay found between frames. (read only)

__enter__() → *OnDiskGif*

No-op used by Context Managers.

__exit__() → None

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

next_frame() → float

Loads the next frame. Returns expected delay before the next frame in seconds.

deinit() → None

Release resources allocated by OnDiskGif.

12.47 gnss – Global Navigation Satellite System

The *gnss* module contains classes to control the GNSS and acquire positioning information.

class gnss.**GNSS**(*system:* 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()
```

(continues on next page)

(continued from previous page)

```
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() → None

Turn off the GNSS.

update() → 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.

12.48 hashlib – Hashing related functions

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

hashlib.**new**(*name: str*, *data: bytes = b''*) → *Hash*

Returns a Hash object setup for the named algorithm. Raises ValueError when the named algorithm is unsupported.

Returns
a hash object for the given algorithm

Return type
hashlib.Hash

class hashlib.**Hash**

In progress hash algorithm. This object is always created by a *hashlib.new()*. It has no user-visible constructor.

digest_size: *int*

Digest size in bytes

update(*data: circuitpython_typing.ReadableBuffer*) → None

Update the hash with the given bytes.

Parameters
data (*ReadableBuffer*) – Update the hash from data in this buffer

digest() → bytes

Returns the current digest as bytes() with a length of *hashlib.Hash.digest_size*.

12.49 i2cdisplaybus – Communicates to a display IC over I2C

class i2cdisplaybus.**I2CDisplayBus**(*i2c_bus:* busio.I2C, *, *device_address:* *int*, *reset:* microcontroller.Pin | *None* = *None*)

Manage updating a display over I2C in the background while Python code runs. It doesn't handle display initialization.

Create a I2CDisplayBus 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() → None

Performs a hardware reset via the reset pin. Raises an exception if called when no reset pin is available.

send(*command:* *int*, *data:* *circuitpython_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.

12.50 i2ctarget – Two wire serial protocol target

The *i2ctarget* module contains classes to support an I2C target.

Example emulating a target with 2 addresses (read and write):

```
import board
from i2ctarget import I2CTarget

regs = [0] * 16
index = 0

with I2CTarget(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: I2CTarget 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 i2ctarget.**I2CTarget**(*scl:* microcontroller.Pin, *sda:* microcontroller.Pin, *addresses: Sequence[int]*, *smbus: bool = False*)

Two wire serial protocol target

I2C is a two-wire protocol for communicating between devices. This implements the target (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 hardware dependent).
- **smbus** (*bool*) – Use SMBUS timings if the hardware supports it

deinit() → None

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

__enter__() → *I2CTarget*

No-op used in Context Managers.

__exit__() → None

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

request(*, *timeout: float = -1*) → *I2CTargetRequest*

Wait for an I2C request.

Parameters
timeout (*float*) – Timeout in seconds. Zero means wait forever, a negative value means check once

Returns
I2CTargetRequest or None if timeout=-1 and there's no request

Return type
I2CTargetRequest

class i2ctarget.**I2CTargetRequest**(*target:* I2CTarget, *address: int*, *is_read: bool*, *is_restart: bool*)

Information about an I2C transfer request This cannot be instantiated directly, but is returned by *I2CTarget.request()*.

Parameters

- **target** – The I2CTarget object receiving this request
- **address** – I2C address
- **is_read** – True if the main target 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 target.

is_restart: bool

Is Repeated Start Condition.

__enter__() → *I2CTargetRequest*

No-op used in Context Managers.

__exit__() → None

Close the request.

read(*n: int = -1*, *ack: bool = True*) → bytearray

Read data. If ack=False, the caller is responsible for calling *I2CTargetRequest.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(*buffer: circuitpython_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(*ack: bool = True*) → None

Acknowledge or Not Acknowledge last byte received. Use together with *I2CTargetRequest.read()* ack=False.

Parameters
ack – Whether to send an ACK or NACK

12.51 imagecapture – Support for "Parallel capture" interfaces

See also:

Espressif microcontrollers use the *espcamera* module together.

class imagecapture.**ParallelImageCapture**(*, *data_pins: List[*microcontroller.Pin*]*, *clock:* microcontroller.Pin, *vsync:* microcontroller.Pin | *None*, *href:* microcontroller.Pin | *None*)

Capture image frames from a camera with parallel data interface

Create a parallel image capture object

This object is usually used with a camera-specific wrapper library such as adafruit_ov5640.

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(*buffer: circuitpython_typing.WriteableBuffer*) → circuitpython_typing.WriteableBuffer

Capture a single frame into the given buffer.

This will stop a continuous-mode capture, if one is in progress.

continuous_capture_start(*buffer1: circuitpython_typing.WriteableBuffer*, *buffer2: circuitpython_typing.WriteableBuffer*, */*) → None

Begin capturing into the given buffers in the background.

Call *continuous_capture_get_frame* to get the next available frame, and *continuous_capture_stop* to stop capturing.

Until *continuous_capture_stop* (or *deinit*) is called, the *ParallelImageCapture* object keeps references to `buffer1` and `buffer2`, so the objects will not be garbage collected.

continuous_capture_get_frame() → circuitpython_typing.WriteableBuffer

Return the next available frame, one of the two buffers passed to *continuous_capture_start*

continuous_capture_stop() → None

Stop continuous capture.

Calling this method also causes the object to release its references to the buffers passed to *continuous_capture_start*, potentially allowing the objects to be garbage collected.

deinit() → None

Deinitialize this instance

__enter__() → *ParallelImageCapture*

No-op used in Context Managers.

__exit__() → None

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

12.52 ipaddress

The *ipaddress* module provides types for IP addresses. It is a subset of CPython's ipaddress module.

ipaddress.**ip_address**(*obj: int | str*) → *IPv4Address*

Return a corresponding IP address object or raise ValueError if not possible.

class ipaddress.**IPv4Address**(*address: 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__(*other: object*) → bool

Two Address objects are equal if their addresses and address types are equal.

__hash__() → int

Returns a hash for the IPv4Address data.

12.53 is31fl3741 – Creates an in-memory framebuffer for a IS31FL3741 device.

class is31fl3741.**IS31FL3741_FrameBuffer**(*is31: IS31FL3741*, *width: int*, *height: int*, *mapping: Tuple[int, Ellipsis]*, ***, *framebuffer: circuitpython_typing.WriteableBuffer | None = None*, *scale: bool = False*, *gamma: bool = False*)

Create a IS31FL3741_FrameBuffer object with the given attributes.

The framebuffer is in "RGB888" format using 4 bytes per pixel. Bits 24-31 are ignored. The format is in RGB order.

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 Is31fl3741 object to memoryview().

A Is31fl3741 is often used in conjunction with a *framebufferio.FramebufferDisplay*.

Parameters

- **is31** (*is31fl3741.IS31FL3741*) – base IS31FL3741 instance to drive the framebuffer
- **width** (*int*) – width of the display

- **height** (*int*) – height of the display
- **mapping** (*Tuple[int, ...]*) – mapping of matrix locations to LEDs
- **framebuffer** (*Optional[WriteableBuffer]*) – Optional buffer to hold the display
- **scale** (*bool*) – if True display is scaled down by 3 when displayed
- **gamma** (*bool*) – if True apply gamma correction to all LEDs

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() → None

Free the resources associated with this IS31FL3741 instance. After deinitialization, no further operations may be performed.

refresh() → None

Transmits the color data in the buffer to the pixels so that they are shown.

class is31fl3741.**IS31FL3741**(*i2c:* busio.I2C, *, *addr: int = 48*)

Driver for an IS31FL3741 device.

Create a IS31FL3741 object with the given attributes.

Designed to work low level or passed to and object such as *IS31FL3741_FrameBuffer*.

Parameters

- **i2c** (I2C) – I2C bus the IS31FL3741 is on
- **addr** (*int*) – device address

deinit() → None

Free the resources associated with this IS31FL3741 instance. After deinitialization, no further operations may be performed.

is31fl3741.**reset**(*self*) → None

Resets the IS31FL3741 chip.

is31fl3741.**enable**(*self*) → None

Enables the IS31FL3741 chip.

is31fl3741.**set_global_current**(*self*, *current: int*) → None

Sets the global current of the IS31FL3741 chip.

Parameters

current (*int*) – global current value 0x00 to 0xFF

is31fl3741.**set_led**(*self*, *led: int*, *value: int*, *page: int*) → None

Resets the IS31FL3741 chip.

Parameters

- **led** (*int*) – which LED to set

- **value** (*int*) – value to set the LED to 0x00 to 0xFF
- **page** (*int*) – page to write to 0 or 2. If the LED is a >= 180 the routine will automatically write to page 1 or 3 (instead of 0 or 2)

is31fl3741.**write**(*mapping: Tuple[int, Ellipsis]*, *buf: circuitpython_typing.ReadableBuffer*) → None

Write buf out on the I2C bus to the IS31FL3741.

Parameters

- **mapping** (*~Tuple[int, ...]*) – map the pixels in the buffer to the order addressed by the driver chip
- **buf** (*ReadableBuffer*) – The bytes to clock out. No assumption is made about color order

12.54 jpegio – Support for JPEG image decoding

class jpegio.**JpegDecoder**

A JPEG decoder

A JpegDecoder allocates a few thousand bytes of memory. To reduce memory fragmentation, create a single JpegDecoder object and use it anytime a JPEG image needs to be decoded.

Example:

```
from jpegio import JpegDecoder
from displayio import Bitmap

decoder = JpegDecoder()
width, height = decoder.open("/sd/example.jpg")
bitmap = Bitmap(width, height, 65535)
decoder.decode(bitmap)
# .. do something with bitmap
```

open(*filename: str*) → Tuple[int, int]

open(*buffer: circuitpython_typing.ReadableBuffer*) → Tuple[int, int]

open(*bytesio:* io.BytesIO) → Tuple[int, int]

Use the specified object as the JPEG data source.

The source may be a filename, a binary buffer in memory, or an opened binary stream.

The single parameter is positional-only (write `open(f)`, not `open(filename=f)` but due to technical limitations this is not shown in the function signature in the documentation.

Returns the image size as the tuple `(width, height)`.

decode(*bitmap:* displayio.Bitmap, *scale: int = 0*, *x: int = 0*, *y: int = 0*, **, x1: int*, *y1: int*, *x2: int*, *y2: int*, *skip_source_index: int*, *skip_dest_index: int*) → None

Decode JPEG data

The bitmap must be large enough to contain the decoded image. The pixel data is stored in the *displayio.Colorspace.RGB565_SWAPPED* colorspace.

The image is optionally downscaled by a factor of `2**scale`. Scaling by a factor of 8 (scale=3) is particularly efficient in terms of decoding time.

The remaining parameters are as for *bitmaptools.blit*. Because JPEG is a lossy data format, chroma keying based on the "source index" is not reliable, because the same original RGB value might end up being

decompressed as a similar but not equal color value. Using a higher JPEG encoding quality can help, but ultimately it will not be perfect.

After a call to `decode`, you must `open` a new JPEG. It is not possible to repeatedly `decode` the same jpeg data, even if it is to select different scales or crop regions from it.

Parameters

- **bitmap** (*Bitmap*) – Optional output buffer
- **scale** (*int*) – Scale factor from 0 to 3, inclusive.
- **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
- **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_source_index** (*int*) – bitmap palette index in the source that will not be copied, set to None to copy all pixels
- **skip_dest_index** (*int*) – bitmap palette index in the destination bitmap that will not get overwritten by the pixels from the source

12.55 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.

For more information about working with the *keypad* module in CircuitPython, see this Learn guide.

class keypad.**Event**(*key_number: int = 0*, *pressed: bool = True*, *timestamp: int | None = None*)

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.
- **timestamp** (*int*) – The time in milliseconds that the keypress occurred in the *supervisor.ticks_ms* time system. If specified as None, the current value of *supervisor.ticks_ms* is used.

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*.

timestamp: int

The timestamp.

__eq__(*other: object*) → bool

Two *Event* objects are equal if their *key_number* and *pressed*/*released* values are equal. Note that this does not compare the event timestamps.

__hash__() → int

Returns a hash for the *Event*, so it can be used in dictionaries, etc..

Note that as events with different timestamps compare equal, they also hash to the same value.

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() → *Event* | None

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(*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*.

Returns
`True` if an event was available and stored, `False` if not.

Return type
bool

clear() → None

Clear any queued key transition events. Also sets *overflowed* to `False`.

__bool__() → bool

True if *len()* is greater than zero. This is an easy way to check if the queue is empty.

__len__() → 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 columns.
- **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() → None

Stop scanning and release the pins.

__enter__() → *KeyMatrix*

No-op used by Context Managers.

__exit__() → None

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

reset() → 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(*key_number: 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`(*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`() → None

Stop scanning and release the pins.

`__enter__`() → *Keys*

No-op used by Context Managers.

`__exit__`() → None

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

`reset`() → 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 | *Sequence[*microcontroller.Pin*]*, *latch:* microcontroller.Pin, *value_to_latch:* *bool* = *True*, *key_count:* *int* | *Sequence[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. Furthermore, you can put multiple shift registers in parallel and share clock and latch.

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. With multiple data pins, key numbers of the next pin are sequentially to the current pin.

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** (*Union[*microcontroller.Pin, *Sequence[*microcontroller.Pin*]]*) – the incoming shift register data pin(s)
- **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** (*Union[int, Sequence[int]]*) – number of data lines to clock in (per data pin)
- **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 total number of keys that are being scanned. (read-only)

events: *EventQueue*

The *EventQueue* associated with this *Keys* object. (read-only)

deinit() → None

Stop scanning and release the pins.

__enter__() → *Keys*

No-op used by Context Managers.

__exit__() → None

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

reset() → 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.

12.56 locale – Locale support module

locale.**getlocale**() → None

Returns the current locale setting as a tuple `(language code, "utf-8")`

The language code comes from the installed translation of CircuitPython, specifically the "Language:" code specified in the translation metadata. This can be useful to allow modules coded in Python to show messages in the user's preferred language.

Differences from CPython: No `LC_*` argument is permitted.

12.57 math – mathematical functions

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

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

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.**log**(*x: float*, *base: float* = *e*) → float

Return the logarithm of x to the given base. If base is not specified, returns the natural logarithm (base e) of x

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`.

May not be available on some boards.

math.**log2**(*x: float*) → float
Return the base-2 logarithm of `x`.

May not be available on some boards.

math.**log10**(*x: float*) → float
Return the base-10 logarithm of `x`.

May not be available on some boards.

math.**cosh**(*x: float*) → float
Return the hyperbolic cosine of `x`.

May not be available on some boards.

math.**sinh**(*x: float*) → float
Return the hyperbolic sine of `x`.

May not be available on some boards.

math.**tanh**(*x: float*) → float
Return the hyperbolic tangent of `x`.

May not be available on some boards.

math.**acosh**(*x: float*) → float
Return the inverse hyperbolic cosine of `x`.

May not be available on some boards.

math.**asinh**(*x: float*) → float
Return the inverse hyperbolic sine of `x`.

May not be available on some boards.

math.**atanh**(*x: float*) → float
Return the inverse hyperbolic tangent of `x`.

May not be available on some boards.

math.**erf**(*x: float*) → float
Return the error function of `x`.

May not be available on some boards.

math.**erfc**(*x: float*) → float
Return the complementary error function of `x`.

May not be available on some boards.

math.**gamma**(*x: float*) → float
Return the gamma function of `x`.

May not be available on some boards.

math.**lgamma**(*x: float*) → float
Return the natural logarithm of the gamma function of `x`.

May not be available on some boards.

12.58 mdns – Multicast Domain Name Service

The *mdns* module provides basic support for multicast domain name services. Basic use provides hostname resolution under the .local TLD. This module also supports DNS Service Discovery that allows for discovering other hosts that provide a desired service.

class mdns.**RemoteService**

Encapsulates information about a remote service that was found during a search. This object may only be created by a *mdns.Server*. It has no user-visible constructor.

Cannot be instantiated directly. Use *mdns.Server.find*.

hostname: *str*

The hostname of the device (read-only),.

instance_name: *str*

The human readable instance name for the service. (read-only)

service_type: *str*

The service type string such as `_http`. (read-only)

protocol: *str*

The protocol string such as `_tcp`. (read-only)

port: *int*

Port number used for the service. (read-only)

ipv4_address: *ipaddress.IPv4Address* | None

IP v4 Address of the remote service. None if no A records are found.

__del__() → None

Deletes the RemoteService object.

class mdns.**Server**(*network_interface:* wifi.Radio)

The MDNS Server responds to queries for this device's information and allows for querying other devices.

Constructs or returns the mdns.Server for the given network_interface. (CircuitPython may already be using it.) Only native interfaces are currently supported.

hostname: *str*

Hostname resolvable as `<hostname>.local` in addition to `circuitpython.local`. Make sure this is unique across all devices on the network. It defaults to `cpy-######` where `######` is the hex digits of the last three bytes of the mac address.

instance_name: *str*

Human readable name to describe the device.

deinit() → None

Stops the server

find(*service_type: str*, *protocol: str*, *, *timeout: float = 1*) → Tuple[*RemoteService*]

Find all locally available remote services with the given service type and protocol.

This doesn't allow for direct hostname lookup. To do that, use *socketpool.SocketPool.getaddrinfo()*.

Parameters

- **service_type** (*str*) – The service type such as "_http"

- **protocol** (*str*) – The service protocol such as "_tcp"
- **timeout** (*float/int*) – Time to wait for responses

advertise_service(*, *service_type: str*, *protocol: str*, *port: int*) → None

Respond to queries for the given service with the given port.

`service_type` and `protocol` can only occur on one port. Any call after the first will update the entry's port.

If web workflow is active, the port it uses can't also be used to advertise a service.

Limitations: Publishing up to 32 TXT records is only supported on the RP2040 Pico W board at this time.

Parameters

- **service_type** (*str*) – The service type such as "_http"
- **protocol** (*str*) – The service protocol such as "_tcp"
- **port** (*int*) – The port used by the service
- **txt_records** (*Sequence[str]*) – An optional sequence of strings to serve as TXT records along with the service

12.59 memorymap – Raw memory map access

The *memorymap* module allows you to read and write memory addresses in the address space seen from the processor running CircuitPython. It is usually the physical address space.

class memorymap.**AddressRange**(*, *start*, *length*)

Presents a range of addresses as a bytearray.

The addresses may access memory or memory mapped peripherals.

Some address ranges may be protected by CircuitPython to prevent errors. An exception will be raised when constructing an AddressRange for an invalid or protected address.

Multiple AddressRanges may overlap. There is no "claiming" of addresses.

Example usage on ESP32-S2:

```
import memorymap
rtc_slow_mem = memorymap.AddressRange(start=0x50000000, length=0x2000)
rtc_slow_mem[0:3] = b"\xcc\x10\x00"
```

Example I/O register usage on RP2040:

```
import binascii
import board
import digitalio
import memorymap

def rp2040_set_pad_drive(p, d):
    pads_bank0 = memorymap.AddressRange(start=0x4001C000, length=0x4000)
    pad_ctrl = int.from_bytes(pads_bank0[p*4+4:p*4+8], "little")
    # Pad control register is updated using an MP-safe atomic XOR
    pad_ctrl ^= (d << 4)
    pad_ctrl &= 0x00000030
```

(continues on next page)

(continued from previous page)

```
    pads_bank0[p*4+0x3004:p*4+0x3008] = pad_ctrl.to_bytes(4, "little")

def rp2040_get_pad_drive(p):
    pads_bank0 = memorymap.AddressRange(start=0x4001C000, length=0x4000)
    pad_ctrl = int.from_bytes(pads_bank0[p*4+4:p*4+8], "little")
    return (pad_ctrl >> 4) & 0x3

# set GPIO16 pad drive strength to 12 mA
rp2040_set_pad_drive(16, 3)

# print GPIO16 pad drive strength
print(rp2040_get_pad_drive(16))
```

Constructs an address range starting at `start` and ending at `start + length`. An exception will be raised if any of the addresses are invalid or protected.

__bool__() → bool

__len__() → int

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

__getitem__(*index: slice*) → bytearray

__getitem__(*index: int*) → int

Returns the value(s) at the given index.

1, 2, 4 and 8 byte aligned reads will be done in one transaction when possible. All others may use multiple transactions.

__setitem__(*index: slice*, *value: circuitpython_typing.ReadableBuffer*) → None

__setitem__(*index: int*, *value: int*) → None

Set the value(s) at the given index.

1, 2, 4 and 8 byte aligned writes will be done in one transaction when possible. All others may use multiple transactions.

12.60 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)
```

(continues on next page)

(continued from previous page)

```
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(*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__() → *AllocationAlarm*

Enables the alarm.

__exit__() → 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__() → *AllocationSize*

Clears counts and resumes tracking.

__exit__() → None

Automatically pauses allocation tracking when exiting a context. See *Lifetime and ContextManagers* for more info.

__len__() → int

Returns the number of allocation buckets.

This allows you to:

```
mm = memorymonitor.AllocationSize()
print(len(mm))
```

__getitem__(*index: int*) → int | None

Returns the allocation count for the given bucket.

This allows you to:

```
mm = memorymonitor.AllocationSize()
print(mm[0])
```

12.61 microcontroller – Pin references and cpu functionality

The *microcontroller* module defines the pins and other bare-metal hardware 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:** *nvm.ByteArray* | None

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:** *watchdog.WatchDogTimer* | None

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.

__hash__() → int

Returns a hash for the 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.

Limitations: On most boards, frequency is read-only. Setting the frequency is possible on RP2040 boards and some i.MX boards.

> **Warning:** Overclocking likely voids your warranties and may reduce the lifetime of the chip.

> **Warning:** Changing the frequency may cause issues with other subsystems, such as USB, PWM, and PIO. To minimize issues, set the CPU frequency before initializing other systems.

reset_reason: *ResetReason*

The reason the microcontroller started up from reset state.

temperature: float | None

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

Is None if the temperature is not available.

Limitations: Not available on ESP32 or ESP32-S3. On small SAMD21 builds without external flash, the reported temperature has reduced accuracy and precision, to save code space.

uid: bytearray

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

voltage: float | None

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 microcontroller was last reset

POWER_ON: object

The microcontroller was started from power off.

BROWNOUT: object

The microcontroller was reset due to too low a voltage.

SOFTWARE: object

The microcontroller was reset from software.

DEEP_SLEEP_ALARM: object

The microcontroller was reset for deep sleep and restarted by an alarm.

RESET_PIN: object

The microcontroller 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 microcontroller restarted for an unknown reason.

RESCUE_DEBUG: object

The microcontroller was reset by the rescue debug port.

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 run and the file system will be writeable over USB.

UF2: *RunMode*

Run the uf2 bootloader.

BOOTLOADER: *RunMode*

Run the default bootloader.

12.62 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.

For more information about working with msgpack, see the CPython Library Documentation.

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):
        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)
```

(continues on next page)

(continued from previous page)

```
buffer.seek(0)
decoded = unpack(buffer, ext_hook=decoder)
print(f"{data} -> {buffer.getvalue()} -> {decoded}")
```

msgpack.**pack**(*obj: object*, *stream: circuitpython_typing.ByteStream*, *, *default: Callable[[object], None] | None = None*) → None

Output object to stream in msgpack format.

Parameters

- **obj** (*object*) – Object to convert to msgpack format.
- **stream** (*ByteStream*) – stream to write to
- **default** (*Optional[Callable[[object], None]]*) – function called for python objects that do not have a representation in msgpack format.

msgpack.**unpack**(*stream: circuitpython_typing.ByteStream*, *, *ext_hook: Callable[[int, bytes], object] | None = None*, *use_list: bool = True*) → object

Unpack and return one object from stream.

Parameters

- **stream** (*ByteStream*) – stream to read from
- **ext_hook** (*Optional[Callable[[int, bytes], object]]*) – 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 stream.

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.

12.63 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
```

(continues on next page)

(continued from previous page)

```
pixel_off = bytearray([0, 0, 0])
neopixel_write.neopixel_write(pin, pixel_off)
```

Note: This module is typically not used by user level code.

For more information on actually using NeoPixels, refer to the CircuitPython Essentials Learn guide

For a much more thorough guide about using NeoPixels, refer to the Adafruit NeoPixel Überguide.

---

neopixel_write.**neopixel_write**(*digitalinout:* digitalio.DigitalInOut, *buf:* *circuitpython_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

12.64 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__() → bool

__len__() → int

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

__getitem__(*index: slice*) → bytearray

__getitem__(*index: int*) → int

Returns the value at the given index.

__setitem__(*index: slice*, *value: circuitpython_typing.ReadableBuffer*) → None

__setitem__(*index: int*, *value: int*) → None

Set the value at the given index.

12.65 onewireio – Low-level bit primitives for Maxim (formerly Dallas Semi) one-wire protocol.

Protocol definition is here: https://www.analog.com/en/technical-articles/1wire-communication-through-software.html

class onewireio.**OneWire**(*pin:* microcontroller.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 onewireio
import board

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

deinit() → None

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

__enter__() → *OneWire*

No-op used by Context Managers.

__exit__() → None

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

reset() → bool

Reset the OneWire bus and read presence

Returns
False when at least one device is present

Return type
bool

read_bit() → bool

Read in a bit

Returns
bit state read

Return type
bool

write_bit(*value: bool*) → None

Write out a bit based on value.

12.66 os – functions that an OS normally provides

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

`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.getenv`(*key: str*, *default: str | None = None*) → str | None

Get the environment variable value for the given key or return `default`.

This may load values from disk so cache the result instead of calling this often.

On boards that do not support `settings.toml` reading in the core, this function will raise NotImplementedError.

`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.

Returns a tuple with the status of a file or directory in the following order:

- `st_mode` – File type, regular or directory
- `st_ino` – Set to 0
- `st_dev` – Set to 0

- `st_nlink` – Set to 0
- `st_uid` – Set to 0
- `st_gid` – Set to 0
- `st_size` – Size of the file in bytes
- `st_atime` – Time of most recent access expressed in seconds
- `st_mtime` – Time of most recent content modification expressed in seconds.
- `st_ctime` – Time of most recent content modification expressed in seconds.

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.

Limitations: Not available on SAMD21 due to lack of hardware.

`os.utime`(*path: str*, *times: Tuple[int, int]*) → None

Change the timestamp of a file.

`os.sep: str`

Separator used to delineate path components such as folder and file names.

12.67 paralleldisplaybus – Native helpers for driving parallel displays

class paralleldisplaybus.**ParallelBus**(*, *command:* microcontroller.Pin, *chip_select:* microcontroller.Pin, *write:* microcontroller.Pin, *data0:* microcontroller.Pin | *None* = *None*, *data_pins: Sequence[*microcontroller.Pin*]* | *None* = *None*, *read:* microcontroller.Pin | *None*, *reset:* microcontroller.Pin | *None* = *None*, *frequency: int = 30000000*)

Manage updating a display over 8-bit parallel bus in the background while Python code runs. This protocol may be referred 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

- **data_pins** (microcontroller.Pin) – A list of data pins. Specify exactly one of `data_pins` or `data0`.
- **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, optional
- **reset** (microcontroller.Pin) – Reset pin, optional
- **frequency** (*int*) – The communication frequency in Hz for the display on the bus

reset() → None

Performs a hardware reset via the reset pin. Raises an exception if called when no reset pin is available.

send(*command: int*, *data: circuitpython_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.

12.68 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 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() → None

Deinitialises the Ps2 and releases any hardware resources for reuse.

__enter__() → *Ps2*

No-op used by Context Managers.

__exit__() → None

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

popleft() → int

Removes and returns the oldest received byte. When buffer is empty, raises an IndexError exception.

sendcmd(*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() → 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__() → bool

__len__() → int

Returns the number of received bytes in buffer, available to *popleft()*.

12.69 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:
```

(continues on next page)

(continued from previous page)

```
    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() → None

Deinitialises the PulseIn and releases any hardware resources for reuse.

__enter__() → *PulseIn*

No-op used by Context Managers.

__exit__() → None

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

pause() → None

Pause pulse capture

resume(*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() → None

Clears all captured pulses

popleft() → int

Removes and returns the oldest read pulse duration in microseconds.

__bool__() → bool

__len__() → int

Returns the number of pulse durations currently stored.

This allows you to:

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

__getitem__(*index: int*) → int | None

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**(*pin:* microcontroller.Pin, *, *frequency: int = 38000*, *duty_cycle: int = 1 << 15*)

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 pin.

Parameters

- **pin** (Pin) – Signal output pin
- **frequency** (*int*) – Carrier signal frequency in Hertz
- **duty_cycle** (*int*) – 16-bit duty cycle of carrier frequency (0 - 65536)

Send a short series of pulses:

```
import array
import pulseio
import pwmio
import board

# 50% duty cycle at 38kHz.
pulse = pulseio.PulseOut(board.LED, frequency=38000, duty_cycle=32768)
#                                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() → None

Deinitialises the PulseOut and releases any hardware resources for reuse.

__enter__() → *PulseOut*

No-op used by Context Managers.

__exit__() → None

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

send(*pulses: circuitpython_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

12.70 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 time
import pwmio
import board

pwm = pwmio.PWMOut(board.LED)
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.

For the essentials of *pwmio*, see the CircuitPython Essentials Learn guide.

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.

Note: The exact frequencies possible depend on the specific microcontroller. If the requested frequency is within the available range, one of the two nearest possible frequencies to the requested one is selected.

If the requested frequency is outside the range, either (A) a ValueError may be raised or (B) the highest or lowest frequency is selected. This behavior is microcontroller-dependent, and may depend on whether it's the upper or lower bound that is exceeded.

In any case, the actual frequency (rounded to 1Hz) is available in the `frequency` property after construction.

Note: The frequency is calculated based on a nominal CPU frequency. However, depending on the board, the error between the nominal and actual CPU frequency can be large (several hundred PPM in the case of crystal oscillators and up to ten percent in the case of RC oscillators)

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 on:

```
import pwmio
import board

pwm = pwmio.PWMOut(board.LED)

while True:
    pwm.duty_cycle = 2 ** 15  # Cycles the pin with 50% duty cycle (half of 2 **␣
↪16) at the default 500hz
```

PWM LED fade:

```
import pwmio
import board

pwm = pwmio.PWMOut(board.LED)  # output on LED pin with default of 500Hz

while True:
    for cycle in range(0, 65535):  # Cycles through the full PWM range from 0 to␣
↪65535
        pwm.duty_cycle = cycle  # Cycles the LED pin duty cycle through the range␣
↪of values
    for cycle in range(65534, 0, -1):  # Cycles through the PWM range backwards␣
↪from 65534 to 0
        pwm.duty_cycle = cycle  # Cycles the LED pin duty cycle through the range␣
↪of values
```

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. However, an output glitch may occur during the adjustment.

deinit() → None

Deinitialises the PWMOut and releases any hardware resources for reuse.

__enter__() → *PWMOut*

No-op used by Context Managers.

__exit__() → None

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

12.71 qrio – Low-level QR code decoding

Provides the *QRDecoder* object used for decoding QR codes. For more information about working with QR codes, see this Learn guide.

Note: This module only handles decoding QR codes. If you are looking to generate a QR code, use the adafruit_miniqr library

class qrio.**PixelPolicy**

EVERY_BYTE: *PixelPolicy*

The input buffer to *QRDecoder.decode* consists of greyscale values in every byte

EVEN_BYTES: *PixelPolicy*

The input buffer to *QRDecoder.decode* consists of greyscale values in positions 0, 2, ..., and ignored bytes in positions 1, 3, ... This can decode directly from YUV images where the even bytes hold the Y (luminance) data.

ODD_BYTES: *PixelPolicy*

The input buffer to *QRDecoder.decode* consists of greyscale values in positions 1, 3, ..., and ignored bytes in positions 0, 2, ... This can decode directly from YUV images where the odd bytes hold the Y (luminance) data

RGB565_SWAPPED: *PixelPolicy*

The input buffer to *QRDecoder.decode* consists of RGB565 values in byte-swapped order. Most cameras produce data in byte-swapped order. The green component is used.

RGB565: *PixelPolicy*

The input buffer to *QRDecoder.decode* consists of RGB565 values in native order. The green component is used.

class qrio.**QRDecoder**(*width: int*, *height: int*)

Construct a QRDecoder object

Parameters

- **width** (*int*) – The pixel width of the image to decode
- **height** (*int*) – The pixel height of the image to decode

width: int

The width of image the decoder expects

height: int

The height of image the decoder expects

decode(*buffer: circuitpython_typing.ReadableBuffer*, *pixel_policy:* PixelPolicy = *PixelPolicy.EVERY_BYTE*) → List[*QRInfo*]

Decode zero or more QR codes from the given image. The size of the buffer must be at least length``×``width bytes for *EVERY_BYTE*, and 2×``length``×``width`` bytes for *EVEN_BYTES* or *ODD_BYTES*.

find(*buffer: circuitpython_typing.ReadableBuffer*, *pixel_policy:* PixelPolicy = *PixelPolicy.EVERY_BYTE*) → List[*QRPosition*]

Find all visible QR codes from the given image. The size of the buffer must be at least length``×``width bytes for *EVERY_BYTE*, and 2×``length``×``width`` bytes for *EVEN_BYTES* or *ODD_BYTES*.

class qrio.**QRInfo**

Information about a decoded QR code

payload: bytes

The content of the QR code

data_type: str | int

The encoding of the payload as a string (if a standard encoding) or int (if not standard)

class qrio.**QRPosition**

Information about a non-decoded QR code

top_left_x: int

X coordinate of the top left corner

top_left_y: *int*
Y coordinate of the top left corner

top_right_x: *int*
X coordinate of the top right corner

top_right_y: *int*
Y coordinate of the top right corner

bottom_right_x: *int*
X coordinate of the bottom right corner

bottom_right_y: *int*
Y coordinate of the bottom right corner

bottom_left_x: *int*
X coordinate of the bottom left corner

bottom_left_y: *int*
Y coordinate of the bottom left corner

size: *int*
The number of bits the code contains

12.72 rainbowio

rainbowio module.

Provides the *colorwheel()* function.

rainbowio.**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).

12.73 random – pseudo-random numbers and choices

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

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: int*) → int

random.**randrange**(*start: int*, *stop: int*) → int

random.**randrange**(*start: int*, *stop: int*, *step: 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.

12.74 `rgbmatrix` – Low-level routines for bitbanged LED matrices

For more information about working with RGB matrix panels in CircuitPython, see the dedicated learn guide.

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: circuitpython_typing.WriteableBuffer | None = 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.

Tiled matrices, those with more than one panel, must be laid out in a specific order, as detailed in the guide.

At least 6 RGB pins and 5 address pins are supported, for common panels with up to 64 rows of pixels. Some microcontrollers may support more, up to a soft limit of 30 RGB pins and 8 address pins.

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*.

On boards designed for use with RGBMatrix panels, `board.MTX_ADDRESS` is a tuple of all the address pins, and `board.MTX_COMMON` is a dictionary with `rgb_pins`, `clock_pin`, `latch_pin`, and `output_enable_pin`. For panels that use fewer than the maximum number of address pins, "slice" `MTX_ADDRESS` to get the correct number of address pins. Using these board properties makes calling the constructor simpler and more portable:

```
matrix = rgbmatrix.RGBMatrix(..., addr_pins=board.MTX_ADDRESS[:4], **board.MTX_
↪COMMON)
```

Parameters

- **width** (*int*) – The overall width of the whole matrix in pixels. For a matrix with multiple panels in row, this is the width of a single panel times the number of panels across.
- **tile** (*int*) – In a multi-row matrix, the number of rows of panels
- **bit_depth** (*int*) – The color depth of the matrix. A value of 1 gives 8 colors, a value of 2 gives 64 colors, and so on. Increasing bit depth increases the CPU and RAM usage of the RGBMatrix, and may lower the panel refresh rate. The framebuffer is always in RGB565 format regardless of the bit depth setting
- **serpentine** (*bool*) – In a multi-row matrix, True when alternate rows of panels are rotated 180°, which can reduce wiring length
- **rgb_pins** (*Sequence[*digitalio.DigitalInOut*]*) – The matrix's RGB pins in the order `(R1,G1,B1,R2,G2,B2...)`
- **addr_pins** (*Sequence[*digitalio.DigitalInOut*]*) – The matrix's address pins in the order `(A,B,C,D...)`
- **clock_pin** (digitalio.DigitalInOut) – The matrix's clock pin
- **latch_pin** (digitalio.DigitalInOut) – The matrix's latch pin
- **output_enable_pin** (digitalio.DigitalInOut) – The matrix's output enable pin
- **doublebuffer** (*bool*) – True if the output is double-buffered
- **framebuffer** (*Optional[WriteableBuffer]*) – A pre-allocated framebuffer to use. If unspecified, a framebuffer is allocated
- **height** (*int*) – The optional overall height of the whole matrix in pixels. This value is not required because it can be calculated as described above.

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() → None

Free the resources (pins, timers, etc.) associated with this rgbmatrix instance. After deinitialization, no further operations may be performed.

refresh() → None

Transmits the color data in the buffer to the pixels so that they are shown.

12.75 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.

For more information on working with rotary encoders using this library, see this Learn Guide.

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, *divisor:* *int* = *4*)

IncrementalEncoder determines the relative rotational position based on two series of pulses. It assumes that the encoder's common pin(s) are connected to ground,and enables pull-ups on pin_a and pin_b.

Create an IncrementalEncoder object associated with the given pins. It tracks the positional state of an incremental rotary encoder (also known as a quadrature encoder.) Position is relative to the position when the object is constructed.

Parameters

- **pin_a** (Pin) – First pin to read pulses from.
- **pin_b** (Pin) – Second pin to read pulses from.
- **divisor** (*int*) – The divisor of the quadrature signal.

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
```

divisor: *int*

The divisor of the quadrature signal. Use 1 for encoders without detents, or encoders with 4 detents per cycle. Use 2 for encoders with 2 detents per cycle. Use 4 for encoders with 1 detent per cycle.

position: *int*

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

deinit() → None

Deinitializes the IncrementalEncoder and releases any hardware resources for reuse.

__enter__() → *IncrementalEncoder*

No-op used by Context Managers.

__exit__() → None

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

12.76 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 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))

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.

Limitations: Calibration not supported on SAMD, nRF, RP240, Spresense, and STM.

Range and value is hardware specific, but one step is often approximately 1 ppm:

```
import rtc
import time
```

(continues on next page)

(continued from previous page)

```
r = rtc.RTC()
r.calibration = 1
```

12.77 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() → 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() → None

Disable permanently.

Returns
None

readblocks(*start_block: int*, *buf: circuitpython_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

sync() → None

Ensure all blocks written are actually committed to the SD card

Returns
None

writeblocks(*start_block: int*, *buf: circuitpython_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

12.78 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
```

(continues on next page)

(continued from previous page)

```
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')
```

frequency: *int*

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

width: *int*

The actual SDIO bus width, in bits

configure(*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() → 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(*start_block: int*, *buf: circuitpython_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(*start_block: int*, *buf: circuitpython_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

deinit() → None

Disable permanently.

Returns
None

__enter__() → *SDCard*

No-op used by Context Managers. Provided by context manager helper.

__exit__() → None

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

12.79 sharpdisplay – Support for Sharp Memory Display framebuffers

For more information about working with Sharp Memory Displays, see this Learn guide.

class sharpdisplay.**SharpMemoryFramebuffer**(*spi_bus:* busio.SPI, *chip_select:* microcontroller.Pin, *width:* *int*, *height:* *int*, *baudrate:* *int* = *2000000*, *jdi_display:* *bool* = *False*)

A framebuffer for a memory-in-pixel display. Sharp makes monochrome displays and JDI used to make 8-color displays.

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 framebuffer for the memory-in-pixel display.

Parameters

- **spi_bus** (busio.SPI) – The SPI bus that the display is connected to
- **chip_select** (microcontroller.Pin) – The pin connect to the display's chip select line
- **width** (*int*) – The width of the display in pixels
- **height** (*int*) – The height of the display in pixels
- **baudrate** (*int*) – The baudrate to communicate with the screen at
- **jdi_display** (*bool*) – When True, work with an 8-color JDI display. Otherwise, a monochrome Sharp display.

deinit() → None

Free the resources (pins, timers, etc.) associated with this SharpMemoryFramebuffer instance. After deinitialization, no further operations may be performed.

12.80 socketpool

The *socketpool* module provides sockets through a pool. The pools themselves act like CPython's *socket* module.

For more information about the *socket* module, see the CPython documentation: https://docs.python.org/3/library/socket.html

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.

__hash__() → int

Returns a hash for the Socket.

__enter__() → *Socket*

No-op used by Context Managers.

__exit__() → None

Automatically closes the Socket when exiting a context. See *Lifetime and ContextManagers* for more info.

accept() → 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(*address: Tuple[str, int]*) → None

Bind a socket to an address

Parameters
address (*~tuple*) – tuple of (remote_address, remote_port)

close() → None

Closes this Socket and makes its resources available to its SocketPool.

connect(*address: Tuple[str, int]*) → None

Connect a socket to a remote address

Parameters
address (*~tuple*) – tuple of (remote_address, remote_port)

listen(*backlog: int*) → None

Set socket to listen for incoming connections

Parameters
backlog (*~int*) – length of backlog queue for waiting connections

recvfrom_into(*buffer: circuitpython_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(*buffer: circuitpython_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(*bytes: circuitpython_typing.ReadableBuffer*) → int

Send some bytes to the connected remote address. Suits sockets of type SOCK_STREAM

Parameters

bytes (*~bytes*) – some bytes to send

sendall(*bytes: circuitpython_typing.ReadableBuffer*) → None

Send some bytes to the connected remote address. Suits sockets of type SOCK_STREAM

This calls send() repeatedly until all the data is sent or an error occurs. If an error occurs, it's impossible to tell how much data has been sent.

Parameters

bytes (*~bytes*) – some bytes to send

sendto(*bytes: circuitpython_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(*flag: bool*) → int | None

Set the blocking behaviour of this socket.

Parameters

flag (*~bool*) – False means non-blocking, True means block indefinitely.

setsockopt(*level: int*, *optname: int*, *value: int*) → None

Sets socket options

settimeout(*value: int*) → None

Set the timeout value for this socket.

Parameters

value (*~int*) – timeout in seconds. 0 means non-blocking. None means block indefinitely.

class socketpool.**SocketPool**(*radio:* wifi.Radio)

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.

Create a new SocketPool object for the provided radio

Parameters

radio (*wifi.Radio*) – The (connected) network hardware to associate with this SocketPool; currently, this will always be the object returned by *wifi.radio*

AF_INET: int

AF_INET6: int

SOCK_STREAM: int

SOCK_DGRAM: int

SOCK_RAW: int

EAI_NONAME: int

SOL_SOCKET: int

SO_REUSEADDR: int

TCP_NODELAY: int

IPPROTO_IP: int

IPPROTO_ICMP: int

IPPROTO_TCP: int

IPPROTO_UDP: int

IPPROTO_IPV6: int

IPPROTO_RAW: int

IP_MULTICAST_TTL: int

socket(*family: int = AF_INET*, *type: int = SOCK_STREAM*, *proto: int = IPPROTO_IP*) → *Socket*

Create a new socket

Parameters

- **family** (*~int*) – AF_INET or AF_INET6
- **type** (*~int*) – SOCK_STREAM, SOCK_DGRAM or SOCK_RAW
- **proto** (*~int*) – IPPROTO_IP, IPPROTO_ICMP, IPPROTO_TCP, IPPROTO_UDP, IPPROTO_IPV6or IPPROTO_RAW. Only works with SOCK_RAW

The `fileno` argument available in `socket.socket()` in CPython is not supported.

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.

12.81 ssl

The *ssl* module provides SSL contexts to wrap sockets in.

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

`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.

check_hostname: bool

Whether to match the peer certificate's hostname.

load_cert_chain(*certfile: str*, *keyfile: str*) → None

Load a private key and the corresponding certificate.

The certfile string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The keyfile string must point to a file containing the private key.

load_verify_locations(*cafile: str | None = None*, *capath: str | None = None*, *cadata: str | None = None*) → None

Load a set of certification authority (CA) certificates used to validate other peers' certificates.

Parameters

- **cafile** (*str*) – path to a file of contcatenated CA certificates in PEM format. **Not implemented**.
- **capath** (*str*) – path to a directory of CA certificate files in PEM format. **Not implemented**.
- **cadata** (*str*) – A single CA certificate in PEM format. **Limitation**: CPython allows one or more certificates, but this implementation is limited to one.

set_default_verify_paths() → None

Load a set of default certification authority (CA) certificates.

wrap_socket(*sock:* socketpool.Socket, *, *server_side: bool = False*, *server_hostname: str | None = 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.

__hash__() → int

Returns a hash for the Socket.

__enter__() → *SSLSocket*

No-op used by Context Managers.

__exit__() → None

Automatically closes the Socket when exiting a context. See *Lifetime and ContextManagers* for more info.

accept() → 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(*address: Tuple[str, int]*) → None

Bind a socket to an address

Parameters
address (*~tuple*) – tuple of (remote_address, remote_port)

close() → None

Closes this Socket

connect(*address: Tuple[str, int]*) → None

Connect a socket to a remote address

Parameters
address (*~tuple*) – tuple of (remote_address, remote_port)

listen(*backlog: int*) → None

Set socket to listen for incoming connections

Parameters
backlog (*~int*) – length of backlog queue for waiting connetions

recv_into(*buffer: circuitpython_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(*bytes: circuitpython_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(*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(*flag: bool*) → int | None

Set the blocking behaviour of this socket.

Parameters
flag (*~bool*) – False means non-blocking, True means block indefinitely.

12.82 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.

For more information regarding using the *storage* module, refer to the CircuitPython Essentials Learn guide.

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

- **filesystem** (*VfsFat*) – The filesystem to mount.
- **mount_path** (*str*) – Where to mount the filesystem.
- **readonly** (*bool*) – True when the filesystem should be readonly to CircuitPython.

storage.**umount**(*mount:* *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

- **mount_path** (*str*) – The path to remount.
- **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**(*extended:* *bool* | *None* = *None*) → 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.

Parameters

extended (*bool*) – On boards that support `dualbank` module and the `extended` parameter, the `CIRCUITPY` storage can be extended by setting this to `True`. If this isn't provided or set to `None` (default), the existing configuration will be used.

Note: New firmware starts with storage extended. In case of an existing filesystem (e.g. uf2 load), the existing extension setting is preserved.

> **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: circuitpython_typing.BlockDevice*)

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.

readonly: bool

True when the device is mounted as readonly by the microcontroller. This property cannot be changed, use *storage.remount* instead.

static mkfs(*block_device: circuitpython_typing.BlockDevice*) → None

Format the block device, deleting any data that may have been there.

Limitations: On SAMD21 builds, *mkfs()* will raise OSError(22) when attempting to format filesystems larger than 4GB. The extra code to format larger filesystems will not fit on these builds. You can still access larger filesystems, but you will need to format the filesystem on another device.

open(*path: str*, *mode: str*) → None

Like builtin open()

ilistdir(*path: str*) → Iterator[Tuple[AnyStr, int, int, int] | Tuple[AnyStr, int, int]]

Return an iterator whose values describe files and folders within path

mkdir(*path: str*) → None

Like *os.mkdir*

rmdir(*path: str*) → None

Like *os.rmdir*

stat(*path: str*) → Tuple[int, int, int, int, int, int, int, int, int, int]

Like *os.stat*

statvfs(*path: int*) → Tuple[int, int, int, int, int, int, int, int, int, int]

Like *os.statvfs*

mount(*readonly: bool*, *mkfs:* VfsFat) → None

Don't call this directly, call *storage.mount*.

umount() → None

Don't call this directly, call *storage.umount*.

12.83 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: `cpython: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: circuitpython_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: circuitpython_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: circuitpython_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.

12.84 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.`**`status_bar`**: *StatusBar*

The status bar, shown on an attached display, and also sent to an attached terminal via OSC escape codes over the REPL serial connection. The status bar reports the current IP or BLE connection, what file is running, the last exception name and location, and firmware version information. This object is the sole instance of *supervisor.StatusBar*.

`supervisor.`**`reload`**() → None

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

`supervisor.`**`set_next_code_file`**(*filename: str | None*, ***, *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.

`supervisor.`**`ticks_ms`**`()` → int

Return the time in milliseconds since an unspecified reference point, wrapping after 2**29ms.

The value is initialized so that the first overflow occurs about 65 seconds after power-on, making it feasible to check that your program works properly around an overflow.

The wrap value was chosen so that it is always possible to add or subtract two *ticks_ms* values without overflow on a board without long ints (or without allocating any long integer objects, on boards with long ints).

This ticks value comes from a low-accuracy clock internal to the microcontroller, just like *time.monotonic*. Due to its low accuracy and the fact that it "wraps around" every few days, it is intended for working with short term events like advancing an LED animation, not for long term events like counting down the time until a holiday.

Addition, subtraction, and comparison of ticks values can be done with routines like the following:

```
_TICKS_PERIOD = const(1<<29)
_TICKS_MAX = const(_TICKS_PERIOD-1)
_TICKS_HALFPERIOD = const(_TICKS_PERIOD//2)

def ticks_add(ticks, delta):
    "Add a delta to a base number of ticks, performing wraparound at 2**29ms."
    return (ticks + delta) % _TICKS_PERIOD

def ticks_diff(ticks1, ticks2):
    "Compute the signed difference between two ticks values, assuming that they are␣
↪within 2**28 ticks"
    diff = (ticks1 - ticks2) & _TICKS_MAX
    diff = ((diff + _TICKS_HALFPERIOD) & _TICKS_MAX) - _TICKS_HALFPERIOD
    return diff

def ticks_less(ticks1, ticks2):
    "Return true iff ticks1 is less than ticks2, assuming that they are within␣
↪2**28 ticks"
    return ticks_diff(ticks1, ticks2) < 0
```

`supervisor.`**`get_previous_traceback`**`()` → str | None

If the last vm run ended with an exception (including the KeyboardInterrupt caused by CTRL-C), returns the traceback as a string. Otherwise, returns `None`.

An exception traceback is only preserved over a soft reload, a hard reset clears it.

Only code (main or boot) runs are considered, not REPL runs.

supervisor.**reset_terminal**(*x_pixels: int*, *y_pixels: int*) → None

Reset the CircuitPython serial terminal with new dimensions.

supervisor.**set_usb_identification**(*manufacturer: str | None = None*, *product: str | None = None*, *vid: int = -1*, *pid: int = -1*) → None

Override identification constants in the USB Device Descriptor.

If passed, *manufacturer* and *product* must be ASCII strings (or buffers) of at most 126 characters. Any omitted arguments will be left at their default values.

This method must be called in boot.py to have any effect.

Not available on boards without native USB support.

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: bool

Returns 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*

Why CircuitPython started running this particular time (read-only).

safe_mode_reason: *SafeModeReason*

Why CircuitPython went into safe mode this particular time (read-only).

Limitations: Raises `NotImplementedError` on builds that do not implement `safemode.py`.

autoreload: bool

Whether CircuitPython may autoreload based on workflow writes to the filesystem.

ble_workflow: bool

Enable/Disable ble workflow until a reset. This prevents BLE advertising outside of the VM and the services used for it.

rgb_status_brightness: int

Set brightness of status RGB LED from 0-255. This will take effect after the current code finishes and the status LED is used to show the finish state.

class supervisor.**SafeModeReason**

The reason that CircuitPython went into safe mode.

Limitations: Class not available on builds that do not implement `safemode.py`.

NONE: object

CircuitPython is not in safe mode.

BROWNOUT: object

The microcontroller voltage dropped too low.

FLASH_WRITE_FAIL: object

Could not write to flash memory.

GC_ALLOC_OUTSIDE_VM: object

CircuitPython tried to allocate storage when its virtual machine was not running.

HARD_FAULT: object

The microcontroller detected a fault, such as an out-of-bounds memory write.

INTERRUPT_ERROR: object

Internal error related to interrupts.

NLR_JUMP_FAIL: object

An error occurred during exception handling, possibly due to memory corruption.

NO_CIRCUITPY: object

The CIRCUITPY drive was not available.

NO_HEAP: object

Heap storage was not present.

PROGRAMMATIC: object

The program entered safe mode using the *supervisor* module.

SDK_FATAL_ERROR: object

Third party firmware reported a fatal error.

STACK_OVERFLOW: object

The CircuitPython heap was corrupted because the stack was too small.

USB_BOOT_DEVICE_NOT_INTERFACE_ZERO: object

The USB HID boot device was not set up to be the first device, on interface #0.

USB_TOO_MANY_ENDPOINTS: object

USB devices need more endpoints than are available.

USB_TOO_MANY_INTERFACE_NAMES: object

USB devices specify too many interface names.

USER: object

The user pressed one or more buttons to enter safe mode. This safe mode does **not** cause `safemode.py` to be run, since its purpose is to prevent all user code from running. This allows errors in `safemode.py` to be corrected easily.

SAFE_MODE_WATCHDOG: object

An internal watchdog timer expired.

class supervisor.**StatusBar**

Current status of runtime objects.

Usage:

```
import supervisor

supervisor.status_bar.console = False
```

You cannot create an instance of *supervisor.StatusBar*. Use *supervisor.status_bar* to access the sole instance available.

console: bool

Whether status bar information is sent over the console (REPL) serial connection, using OSC terminal escape codes that change the terminal's title. Default is `True`. If set to `False`, status bar will be cleared and then disabled. May be set in `boot.py` or later. Persists across soft restarts.

display: bool

Whether status bar information is displayed on the top line of the display. Default is `True`. If set to `False`, status bar will be cleared and then disabled. May be set in `boot.py` or later. Persists across soft restarts. Not available if *terminalio* is not available.

12.85 synthio – Support for multi-channel audio synthesis

At least 2 simultaneous notes are supported. samd5x, mimxrt10xx and rp2040 platforms support up to 12 notes.

class synthio.**EnvelopeState**

ATTACK: *EnvelopeState*

The note is in its attack phase

DECAY: *EnvelopeState*

The note is in its decay phase

SUSTAIN: *EnvelopeState*

The note is in its sustain phase

RELEASE: *EnvelopeState*

The note is in its release phase

synthio.**BlockInput**

Blocks and Notes can take any of these types as inputs on certain attributes

A BlockInput can be any of the following types: *Math*, *LFO*, *builtins.float*, None (treated same as 0).

class synthio.**Envelope**(*, *attack_time: float | None = 0.1*, *decay_time: float | None = 0.05*, *release_time: float | None = 0.2*, *attack_level: float | None = 1.0*, *sustain_level: float | None = 0.8*)

Construct an Envelope object

The Envelope defines an ADSR (Attack, Decay, Sustain, Release) envelope with linear amplitude ramping. A note starts at 0 volume, then increases to `attack_level` over `attack_time` seconds; then it decays to `sustain_level` over `decay_time` seconds. Finally, when the note is released, it decreases to `0` volume over `release_time`.

If the `sustain_level` of an envelope is 0, then the decay and sustain phases of the note are always omitted. The note is considered to be released as soon as the envelope reaches the end of the attack phase. The `decay_time` is ignored. This is similar to how a plucked or struck instrument behaves.

If a note is released before it reaches its sustain phase, it decays with the same slope indicated by `sustain_level/release_time` (or `attack_level/release_time` for plucked envelopes)

Parameters

- **attack_time** (*float*) – The time in seconds it takes to ramp from 0 volume to attack_volume
- **decay_time** (*float*) – The time in seconds it takes to ramp from attack_volume to sustain_volume
- **release_time** (*float*) – The time in seconds it takes to ramp from sustain_volume to release_volume. When a note is released before it has reached the sustain phase, the release is done with the same slope indicated by `release_time` and `sustain_level`. If the `sustain_level` is `0.0` then the release slope calculations use the `attack_level` instead.
- **attack_level** (*float*) – The level, in the range `0.0` to `1.0` of the peak volume of the attack phase
- **sustain_level** (*float*) – The level, in the range `0.0` to `1.0` of the volume of the sustain phase relative to the attack level

attack_time: float

The time in seconds it takes to ramp from 0 volume to attack_volume

decay_time: float

The time in seconds it takes to ramp from attack_volume to sustain_volume

release_time: float

The time in seconds it takes to ramp from sustain_volume to release_volume. When a note is released before it has reached the sustain phase, the release is done with the same slope indicated by `release_time` and `sustain_level`

attack_level: float

The level, in the range `0.0` to `1.0` of the peak volume of the attack phase

sustain_level: float

The level, in the range `0.0` to `1.0` of the volume of the sustain phase relative to the attack level

synthio.**from_file**(*file: BinaryIO*, *, *sample_rate: int = 11025*, *waveform: circuitpython_typing.ReadableBuffer | None = None*, *envelope:* Envelope | *None = None*) → *MidiTrack*

Create an AudioSample from an already opened MIDI file. Currently, only single-track MIDI (type 0) is supported.

Parameters

- **file** (*BinaryIO*) – Already opened MIDI file
- **sample_rate** (*int*) – The desired playback sample rate; higher sample rate requires more memory
- **waveform** (*ReadableBuffer*) – A single-cycle waveform. Default is a 50% duty cycle square wave. If specified, must be a ReadableBuffer of type 'h' (signed 16 bit)
- **envelope** (Envelope) – An object that defines the loudness of a note over time. The default envelope provides no ramping, voices turn instantly on and off.

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")
```

synthio.**midi_to_hz**(*midi_note: float*) → float

Converts the given midi note (60 = middle C, 69 = concert A) to Hz

synthio.**voct_to_hz**(*ctrl: float*) → float

Converts a 1v/octave signal to Hz.

24/12 (2.0) corresponds to middle C, 33/12 (2.75) is concert A.

synthio.**waveform_max_length: int**

The maximum number of samples permitted in a waveform

class synthio.**Biquad**(*b0: float*, *b1: float*, *b2: float*, *a1: float*, *a2: float*)

Construct a normalized biquad filter object.

This implements the "direct form 1" biquad filter, where each coefficient has been pre-divided by a0.

Biquad objects are usually constructed via one of the related methods on a *Synthesizer* object rather than directly from coefficients.

https://github.com/WebAudio/Audio-EQ-Cookbook/blob/main/Audio-EQ-Cookbook.txt

class synthio.**LFO**(*waveform: circuitpython_typing.ReadableBuffer | None = None*, ***, *rate:* BlockInput *= 1.0*, *scale:* BlockInput *= 1.0*, *offset:* BlockInput *= 0.0*, *phase_offset:* BlockInput *= 0.0*, *once=False*, *interpolate=True*)

A low-frequency oscillator block

Every *rate* seconds, the output of the LFO cycles through its *waveform*. The output at any particular moment is `waveform[idx] * scale + offset`.

If *waveform* is None, a triangle waveform is used.

rate, *phase_offset*, *offset*, *scale*, and *once* can be changed at run-time. *waveform* may be mutated.

waveform must be a `ReadableBuffer` with elements of type `'h'` (16-bit signed integer). Internally, the elements of *waveform* are scaled so that the input range `[-32768,32767]` maps to `[-1.0, 0.99996]`.

An LFO only updates if it is actually associated with a playing *Synthesizer*, including indirectly via a *Note* or another intermediate LFO.

Using the same LFO as an input to multiple other LFOs or Notes is OK, but the result if an LFO is tied to multiple Synthtesizer objects is undefined.

In the current implementation, LFOs are updated every 256 samples. This should be considered an implementation detail, though it affects how LFOs behave for instance when used to implement an integrator (`l.offset = 1`).

waveform: circuitpython_typing.ReadableBuffer | None

The waveform of this lfo. (read-only, but the values in the buffer may be modified dynamically)

rate: *BlockInput*

The rate (in Hz) at which the LFO cycles through its waveform

offset: *BlockInput*

An additive value applied to the LFO's output

phase_offset: *BlockInput*

An additive value applied to the LFO's phase

scale: *BlockInput*

An multiplier value applied to the LFO's output

once: bool

True if the waveform should stop when it reaches its last output value, false if it should re-start at the beginning of its waveform

This applies to the `phase` *before* the addition of any `phase_offset`

interpolate: bool

True if the waveform should perform linear interpolation between values

phase: float

The phase of the oscillator, in the range 0 to 1 (read-only)

value: float

The value of the oscillator (read-only)

retrigger()

Reset the LFO's internal index to the start of the waveform. Most useful when it its *once* property is `True`.

class synthio.**MathOperation**

Operation for a Math block

SUM: *MathOperation*

Computes `a+b+c`. For 2-input sum, set one argument to `0.0`. To hold a control value for multiple subscribers, set two arguments to `0.0`.

ADD_SUB: *MathOperation*

Computes `a+b-c`. For 2-input subtraction, set `b` to `0.0`.

PRODUCT: *MathOperation*

Computes `a*b*c`. For 2-input product, set one argument to `1.0`.

MUL_DIV: *MathOperation*

Computes `a*b/c`. If `c` is zero, the output is `1.0`.

SCALE_OFFSET: *MathOperation*

Computes `(a*b)+c`.

OFFSET_SCALE: *MathOperation*

Computes `(a+b)*c`. For 2-input multiplication, set `b` to 0.

LERP: *MathOperation*

Computes `a * (1-c) + b * c`.

CONSTRAINED_LERP: *MathOperation*

Computes `a * (1-c') + b * c'`, where `c'` is constrained to be between `0.0` and `1.0`.

DIV_ADD: *MathOperation*

Computes `a/b+c`. If `b` is zero, the output is `c`.

ADD_DIV: *MathOperation*

Computes `(a+b)/c`. For 2-input product, set `b` to `0.0`.

MID: *MathOperation*

Returns the middle of the 3 input values.

MAX: *MathOperation*

Returns the biggest of the 3 input values.

MIN: *MathOperation*

Returns the smallest of the 3 input values.

ABS: *MathOperation*

Returns the absolute value of `a`.

__call__(*a:* BlockInput, *b:* BlockInput = *0.0*, *c:* BlockInput = *1.0*) → *Math*

A MathOperation enumeration value can be called to construct a Math block that performs that operation

class synthio.**Math**(*operation:* MathOperation, *a:* BlockInput, *b:* BlockInput = *0.0*, *c:* BlockInput = *1.0*)

An arithmetic block

Performs an arithmetic operation on up to 3 inputs. See the documentation of `MathOperation` for the specific functions available.

The properties can all be changed at run-time.

An Math only updates if it is actually associated with a playing *Synthesizer*, including indirectly via a *Note* or another intermediate Math.

Using the same Math as an input to multiple other Maths or Notes is OK, but the result if an Math is tied to multiple Synthtesizer objects is undefined.

In the current implementation, Maths are updated every 256 samples. This should be considered an implementation detail.

a: *BlockInput*

The first input to the operation

b: *BlockInput*

The second input to the operation

c: *BlockInput*

The third input to the operation

operation: *MathOperation*

The function to compute

value: float

The value of the oscillator (read-only)

class synthio.**MidiTrack**(*buffer: circuitpython_typing.ReadableBuffer*, *tempo: int*, *, *sample_rate: int = 11025*, *waveform: circuitpython_typing.ReadableBuffer | None = None*, *envelope: Envelope | None = None*)

Simple 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
- **waveform** (*ReadableBuffer*) – A single-cycle waveform. Default is a 50% duty cycle square wave. If specified, must be a ReadableBuffer of type 'h' (signed 16 bit)
- **envelope** (Envelope) – An object that defines the loudness of a note over time. The default envelope provides no ramping, voices turn instantly on and off.

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: int

32 bit value that tells how quickly samples are played in Hertz (cycles per second).

error_location: int | None

Offset, in bytes within the midi data, of a decoding error

deinit() → None

Deinitialises the MidiTrack and releases any hardware resources for reuse.

__enter__() → *MidiTrack*

No-op used by Context Managers.

__exit__() → None

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

class synthio.**Note**(*, *frequency: float*, *panning:* BlockInput = *0.0*, *waveform: circuitpython_typing.ReadableBuffer | None = None*, *waveform_loop_start: int = 0*, *waveform_loop_end: int = waveform_max_length*, *envelope:* Envelope | *None = None*, *amplitude:* BlockInput = *0.0*, *bend:* BlockInput = *0.0*, *filter:* Biquad | *None = None*, *ring_frequency: float = 0.0*, *ring_bend: float = 0.0*, *ring_waveform: circuitpython_typing.ReadableBuffer | None = None*, *ring_waveform_loop_start: int = 0*, *ring_waveform_loop_end: int = waveform_max_length*)

Construct a Note object, with a frequency in Hz, and optional panning, waveform, envelope, tremolo (volume change) and bend (frequency change).

If waveform or envelope are `None` the synthesizer object's default waveform or envelope are used.

If the same Note object is played on multiple Synthesizer objects, the result is undefined.

frequency: *float*

The base frequency of the note, in Hz.

filter: *Biquad* | **None**

If not None, the output of this Note is filtered according to the provided coefficients.

Construct an appropriate filter by calling a filter-making method on the *Synthesizer* object where you plan to play the note, as filter coefficients depend on the sample rate

panning: *BlockInput*

Defines the channel(s) in which the note appears.

-1 is left channel only, 0 is both channels, and 1 is right channel. For fractional values, the note plays at full amplitude in one channel and partial amplitude in the other channel. For instance -.5 plays at full amplitude in the left channel and 1/2 amplitude in the right channel.

amplitude: *BlockInput*

The relative amplitude of the note, from 0 to 1

An amplitude of 0 makes the note inaudible. It is combined multiplicatively with the value from the note's envelope.

To achieve a tremolo effect, attach an LFO here.

bend: *BlockInput*

The pitch bend depth of the note, from -12 to +12

A depth of 0 plays the programmed frequency. A depth of 1 corresponds to a bend of 1 octave. A depth of (1/12) = 0.0833 corresponds to a bend of 1 semitone, and a depth of .00833 corresponds to one musical cent.

To achieve a vibrato or sweep effect, attach an LFO here.

waveform: circuitpython_typing.ReadableBuffer | **None**

The waveform of this note. Setting the waveform to a buffer of a different size resets the note's phase.

waveform_loop_start: *int*

The sample index of where to begin looping waveform data.

Values outside the range `0` to `waveform_max_length-1` (inclusive) are rejected with a *ValueError*.

Values greater than or equal to the actual waveform length are treated as 0.

waveform_loop_end: *int*

The sample index of where to end looping waveform data.

Values outside the range `1` to `waveform_max_length` (inclusive) are rejected with a *ValueError*.

If the value is greater than the actual waveform length, or less than or equal to the loop start, the loop will occur at the end of the waveform.

Use the *synthio.waveform_max_length* constant to set the loop point at the end of the wave form, no matter its length.

envelope: *Envelope*

The envelope of this note

ring_frequency: *float*

The ring frequency of the note, in Hz. Zero disables.

For ring to take effect, both `ring_frequency` and `ring_waveform` must be set.

ring_bend: *float*

The pitch bend depth of the note's ring waveform, from -12 to +12

A depth of 0 plays the programmed frequency. A depth of 1 corresponds to a bend of 1 octave. A depth of (1/12) = 0.0833 corresponds to a bend of 1 semitone, and a depth of .00833 corresponds to one musical cent.

To achieve a vibrato or sweep effect on the ring waveform, attach an LFO here.

ring_waveform: *circuitpython_typing.ReadableBuffer | None*

The ring waveform of this note. Setting the ring_waveform to a buffer of a different size resets the note's phase.

For ring to take effect, both `ring_frequency` and `ring_waveform` must be set.

ring_waveform_loop_start: *int*

The sample index of where to begin looping waveform data.

Values outside the range `0` to `waveform_max_length-1` (inclusive) are rejected with a *ValueError*.

Values greater than or equal to the actual waveform length are treated as 0.

ring_waveform_loop_end: *int*

The sample index of where to end looping waveform data.

Values outside the range `1` to `waveform_max_length` (inclusive) are rejected with a *ValueError*.

If the value is greater than the actual waveform length, or less than or equal to the loop start, the loop will occur at the end of the waveform.

Use the *synthio.waveform_max_length* constant to set the loop point at the end of the wave form, no matter its length.

synthio.**NoteSequence**

A sequence of notes, which can each be integer MIDI note numbers or *Note* objects

synthio.**NoteOrNoteSequence**

A note or sequence of notes

synthio.**LFOOrLFOSequence**

An LFO or a sequence of LFOs

class synthio.**Synthesizer**(*, *sample_rate: int = 11025*, *channel_count: int = 1*, *waveform: circuitpython_typing.ReadableBuffer | None = None*, *envelope:* Envelope | *None = None*)

Create a synthesizer object.

This API is experimental.

Integer notes use MIDI note numbering, with 60 being C4 or Middle C, approximately 262Hz. Integer notes use the given waveform & envelope, and do not support advanced features like tremolo or vibrato.

Parameters

- **sample_rate** (*int*) – The desired playback sample rate; higher sample rate requires more memory
- **channel_count** (*int*) – The number of output channels (1=mono, 2=stereo)
- **waveform** (*ReadableBuffer*) – A single-cycle waveform. Default is a 50% duty cycle square wave. If specified, must be a ReadableBuffer of type 'h' (signed 16 bit)
- **envelope** (*Optional[Envelope]*) – An object that defines the loudness of a note over time. The default envelope, None provides no ramping, voices turn instantly on and off.

envelope: *Envelope* | **None**

The envelope to apply to all notes. None, the default envelope, instantly turns notes on and off. The envelope may be changed dynamically, but it affects all notes (even currently playing notes)

sample_rate: int

32 bit value that tells how quickly samples are played in Hertz (cycles per second).

pressed: *NoteSequence*

A sequence of the currently pressed notes (read-only property).

This does not include notes in the release phase of the envelope.

blocks: List[*BlockInput*]

A list of blocks to advance whether or not they are associated with a playing note.

This can be used to implement 'free-running' LFOs. LFOs associated with playing notes are advanced whether or not they are in this list.

This property is read-only but its contents may be modified by e.g., calling `synth.blocks.append()` or `synth.blocks.remove()`. It is initially an empty list.

max_polyphony: int

Maximum polyphony of the synthesizer (read-only class property)

press(*/*, *press=()*) → None

Turn some notes on.

Pressing a note that was already pressed has no effect.

Parameters

press (NoteOrNoteSequence) – Any sequence of notes.

release(*/*, *release=()*) → None

Turn some notes off.

Releasing a note that was already released has no effect.

Parameters
release (NoteOrNoteSequence) – Any sequence of notes.

change(*release:* NoteOrNoteSequence = *()*, *press:* NoteOrNoteSequence = *()*, *retrigger=LFOOrLFOSequence*) → None

Start notes, stop them, and/or re-trigger some LFOs.

The changes all happen atomically with respect to output generation.

It is OK to release note that was not actually turned on.

Pressing a note that was already pressed returns it to the attack phase but without resetting its amplitude. Releasing a note and immediately pressing it again returns it to the attack phase with an initial amplitude of 0.

At the same time, the passed LFOs (if any) are retriggered.

Parameters

- **release** (NoteOrNoteSequence) – Any sequence of notes.
- **press** (NoteOrNoteSequence) – Any sequence of notes.
- **retrigger** (LFOOrLFOSequence) – Any sequence of LFOs.

Note: for compatibility, `release_then_press` may be used as an alias for this function. This compatibility name will be removed in 9.0.

release_all_then_press(*/*, *press*) → None

Turn any currently-playing notes off, then turn on the given notes

Releasing a note and immediately pressing it again returns it to the attack phase with an initial amplitude of 0.

Parameters
press (NoteOrNoteSequence) – Any sequence of notes.

release_all() → None

Turn any currently-playing notes off

deinit() → None

Deinitialises the object and releases any memory resources for reuse.

__enter__() → *Synthesizer*

No-op used by Context Managers.

__exit__() → None

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

note_info(*note:* Note) → Tuple[*EnvelopeState* | None, float]

Get info about a note's current envelope state

If the note is currently playing (including in the release phase), the returned value gives the current envelope state and the current envelope value.

If the note is not playing on this synthesizer, returns the tuple `(None, 0.0)`.

low_pass_filter(*frequency: float*, *q_factor: float = 0.7071067811865475*) → *Biquad*

Construct a low-pass filter with the given parameters.

`frequency`, called f0 in the cookbook, is the corner frequency in Hz of the filter.

`q_factor`, called `Q` in the cookbook. Controls how peaked the response will be at the cutoff frequency. A large value makes the response more peaked.

high_pass_filter(*frequency: float*, *q_factor: float = 0.7071067811865475*) → *Biquad*

Construct a high-pass filter with the given parameters.

`frequency`, called f0 in the cookbook, is the corner frequency in Hz of the filter.

`q_factor`, called `Q` in the cookbook. Controls how peaked the response will be at the cutoff frequency. A large value makes the response more peaked.

band_pass_filter(*frequency: float*, *q_factor: float = 0.7071067811865475*) → *Biquad*

Construct a band-pass filter with the given parameters.

`frequency`, called f0 in the cookbook, is the center frequency in Hz of the filter.

`q_factor`, called `Q` in the cookbook. Controls how peaked the response will be at the cutoff frequency. A large value makes the response more peaked.

The coefficients are scaled such that the filter has a 0dB peak gain.

12.86 `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`.

Note: This module does not give access to the REPL.

terminalio.**FONT**: *fontio.BuiltinFont*

The built in font

class terminalio.**Terminal**(*scroll_area:* displayio.TileGrid, *font:* fontio.BuiltinFont, *, *status_bar:* displayio.TileGrid | *None* = *None*)

Display a character stream with a TileGrid

ASCII control: * `\r` - Move cursor to column 1 * `\n` - Move cursor down a row * `\b` - Move cursor left one if possible

OSC control sequences: * `ESC ] 0; <s> ESC \` - Set title bar to <s> * `ESC ] ####; <s> ESC \` - Ignored

VT100 control sequences: * `ESC [ K` - Clear the remainder of the line * `ESC [ #### D` - Move the cursor to the left by #### * `ESC [ 2 J` - Erase the entire display * `ESC [ nnnn ; mmmm H` - Move the cursor to mmmm, nnnn.

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(*buf: circuitpython_typing.ReadableBuffer*) → int | None

Write the buffer of bytes to the bus.

Returns
the number of bytes written

Return type
int or None

12.87 time – time and timing related functions

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

`time.monotonic()` → float

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

On most boards, *time.monotonic()* converts a 64-bit millisecond tick counter to a float. Floats on most boards are encoded in 30 bits internally, with effectively 22 bits of precision. The float returned by *time.monotonic()* will accurately represent time to millisecond precision only up to 2**22 milliseconds (about 1.165 hours). At that point it will start losing precision, and its value will change only every second millisecond. At 2**23 milliseconds it will change every fourth millisecond, and so forth.

If you need more consistent precision, use *time.monotonic_ns()*, or *supervisor.ticks_ms()*. *time.monotonic_ns()* is not available on boards without long integer support. *supervisor.ticks_ms()* uses intervals of a millisecond, but wraps around, and is not CPython-compatible.

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, which cannot go backward, in nanoseconds. Not available on boards without long integer support. Only use it to compare against other values from *time.monotonic()* during a single code run.

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

12.88 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 more information about working with the *touchio* module in CircuitPython, see this Learn guide page.

Example:

```
import touchio
from board import *
```

(continues on next page)

(continued from previous page)

```
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: int | None

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() → None

Deinitialises the TouchIn and releases any hardware resources for reuse.

__enter__() → *TouchIn*

No-op used by Context Managers.

__exit__() → None

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

12.89 traceback – Traceback Module

This module provides a standard interface to print stack traces of programs. This is useful when you want to print stack traces under program control.

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

`traceback.format_exception`(*exc: BaseException | Type[BaseException], /, value: BaseException | None = None, tb: types.TracebackType | None = None, limit: int | None = None, chain: bool | None = True*) → List[str]

Format a stack trace and the exception information.

If the exception value is passed in `exc`, then this exception value and its associated traceback are used. This is compatible with CPython 3.10 and newer.

If the exception value is passed in `value`, then any value passed in for `exc` is ignored. `value` is used as the exception value and the traceback in the `tb` argument is used. In this case, if `tb` is None, no traceback will be shown. This is compatible with CPython 3.5 and newer.

The arguments have the same meaning as the corresponding arguments to print_exception(). The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does print_exception().

Parameters

- **exc** – The exception. Must be an instance of *BaseException*. Unused if value is specified.
- **value** – If specified, is used in place of `exc`.
- **tb** (*TracebackType*) – When value is alsp specified, `tb` is used in place of the exception's own traceback. If `None`, the traceback will not be printed.
- **limit** (*int*) – Print up to limit stack trace entries (starting from the caller's frame) if limit is positive. Otherwise, print the last `abs(limit)` entries. If limit is omitted or None, all entries are printed.
- **chain** (*bool*) – If `True` then chained exceptions will be printed.

`traceback.print_exception`(*exc: BaseException | Type[BaseException], /, value: BaseException | None = None, tb: types.TracebackType | None = None, limit: int | None = None, file: io.FileIO | None = None, chain: bool | None = True*) → None

Prints exception information and stack trace entries.

If the exception value is passed in `exc`, then this exception value and its associated traceback are used. This is compatible with CPython 3.10 and newer.

If the exception value is passed in `value`, then any value passed in for `exc` is ignored. `value` is used as the exception value and the traceback in the `tb` argument is used. In this case, if `tb` is None, no traceback will be shown. This is compatible with CPython 3.5 and newer.

Parameters

- **exc** – The exception. Must be an instance of *BaseException*. Unused if value is specified.
- **value** – If specified, is used in place of `exc`.
- **tb** – When value is alsp specified, `tb` is used in place of the exception's own traceback. If `None`, the traceback will not be printed.

- **limit** (*int*) – Print up to limit stack trace entries (starting from the caller's frame) if limit is positive. Otherwise, print the last `abs(limit)` entries. If limit is omitted or None, all entries are printed.
- **file** (`io.FileIO`) – If file is omitted or `None`, the output goes to *sys.stderr*; otherwise it should be an open file or file-like object to receive the output.
- **chain** (*bool*) – If `True` then chained exceptions will be printed.

12.90 uheap – Heap size analysis

uheap.**info**(*object: info.object*) → int

Prints memory debugging info for the given object and returns the estimated size.

12.91 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

12.91.1 ulab.numpy – Numerical approximation methods

ulab.numpy.carray – Return the real part of the complex argument, which can be either an ndarray, or a scalar.

ulab.numpy.carray.**real**(*val*)

ulab.numpy.carray.**imag**(*val*)

Return the imaginary part of the complex argument, which can be either an ndarray, or a scalar.

ulab.numpy.carray.**conjugate**(*val*)

Return the conjugate of the complex argument, which can be either an ndarray, or a scalar.

ulab.numpy.carray.**sort_complex**(*a:* ulab.numpy.ndarray) → *ulab.numpy.ndarray*

Sort a complex array using the real part first, then the imaginary part. Always returns a sorted complex array, even if the input was real.

ulab.numpy.carray.**abs**(*a:* ulab.numpy.ndarray) → *ulab.numpy.ndarray*

Return the absolute value of complex ndarray.

ulab.numpy.fft – Frequency-domain functions

ulab.numpy.fft.**fft**(*r:* ulab.numpy.ndarray, *c:* ulab.numpy.ndarray | *None* = *None*) → Tuple[*ulab.numpy.ndarray*, *ulab.numpy.ndarray*]

Parameters

- **r** (ulab.numpy.ndarray) – A 1-dimension array of values whose size is a power of 2
- **c** (ulab.numpy.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.utils.spectrogram*, which computes the magnitude of the fft, rather than separately returning its real and imaginary parts.

ulab.numpy.fft.**ifft**(*r:* ulab.numpy.ndarray, *c:* ulab.numpy.ndarray | *None* = *None*) → Tuple[*ulab.numpy.ndarray*, *ulab.numpy.ndarray*]

Parameters

- **r** (ulab.numpy.ndarray) – A 1-dimension array of values whose size is a power of 2
- **c** (ulab.numpy.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.linalg.**qr**(*m:* ulab.numpy.ndarray) → Tuple[*ulab.numpy.ndarray*, *ulab.numpy.ndarray*]

Parameters
m – a matrix

Return tuple (Q, R)

Factor the matrix a as QR, where Q is orthonormal and R is upper-triangular.

ulab.numpy.**interp**(*x:* ndarray, *xp:* ndarray, *fp:* ndarray, *, *left: _float | None = None*, *right: _float | None = None*) → *ndarray*

Parameters

- **x** (ulab.numpy.ndarray) – The x-coordinates at which to evaluate the interpolated values.
- **xp** (ulab.numpy.ndarray) – The x-coordinates of the data points, must be increasing
- **fp** (ulab.numpy.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.**trapz**(*y:* ndarray, *x:* ndarray | *None = None*, *dx: _float = 1.0*) → _float

Parameters

- **y** (*1D ulab.numpy.ndarray*) – the values of the dependent variable
- **x** (*1D ulab.numpy.ndarray*) – 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.**arange**(*stop: _float*, *step: _float = 1*, *, *dtype: _DType = ulab.numpy.float*) → *ndarray*

ulab.numpy.**arange**(*start: _float*, *stop: _float*, *step: _float = 1*, *, *dtype: _DType = ulab.numpy.float*) → *ndarray*

Return a new 1-D array with elements ranging from `start` to `stop`, with step size `step`.

ulab.numpy.**concatenate**(*arrays: Tuple[*ndarray*]*, *, *axis: int = 0*) → *ndarray*

Join a sequence of arrays along an existing axis.

ulab.numpy.**diag**(*a:* ndarray, *, *k: int = 0*) → *ndarray*

Return specified diagonals.

ulab.numpy.**empty**(*shape: int | Tuple[int, Ellipsis]*, *, *dtype: _DType = ulab.numpy.float*) → *ndarray*

Return a new array of the given shape with all elements set to 0. An alias for numpy.zeros.

ulab.numpy.**eye**(*size: int*, *, *M: int | None = None*, *k: int = 0*, *dtype: _DType = ulab.numpy.float*) → *ndarray*

Return a new square array of size, with the diagonal elements set to 1 and the other elements set to 0. If k is given, the diagonal is shifted by the specified amount.

ulab.numpy.**full**(*shape: int | Tuple[int, Ellipsis]*, *fill_value: _float | _bool*, *, *dtype: _DType = ulab.numpy.float*) → *ndarray*

Return a new array of the given shape with all elements set to 0.

ulab.numpy.**linspace**(*start: _float*, *stop: _float*, *, *dtype: _DType = ulab.numpy.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.numpy.**logspace**(*start: _float*, *stop: _float*, *, *dtype: _DType = ulab.numpy.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.numpy.**ones**(*shape: int | Tuple[int, Ellipsis]*, *, *dtype: _DType = ulab.numpy.float*) → *ndarray*

Return a new array of the given shape with all elements set to 1.

ulab.numpy.**zeros**(*shape: int | Tuple[int, Ellipsis]*, *, *dtype: _DType = ulab.numpy.float*) → *ndarray*

Return a new array of the given shape with all elements set to 0.

ulab.numpy.**_ArrayLike**

ulab.numpy.**_DType**

ulab.numpy.int8, *ulab.numpy.uint8*, *ulab.numpy.int16*, *ulab.numpy.uint16*, *ulab.numpy.float* or *ulab.numpy.bool*

ulab.numpy.**int8**: ***_DType***

Type code for signed integers in the range -128 .. 127 inclusive, like the 'b' typecode of *array.array*

ulab.numpy.**int16**: ***_DType***

Type code for signed integers in the range -32768 .. 32767 inclusive, like the 'h' typecode of *array.array*

ulab.numpy.**float**: ***_DType***

Type code for floating point values, like the 'f' typecode of *array.array*

ulab.numpy.**uint8**: ***_DType***

Type code for unsigned integers in the range 0 .. 255 inclusive, like the 'H' typecode of *array.array*

ulab.numpy.**uint16**: ***_DType***

Type code for unsigned integers in the range 0 .. 65535 inclusive, like the 'h' typecode of *array.array*

ulab.numpy.**bool:** *_DType*

Type code for boolean values

ulab.numpy.**argmax**(*array:* _ArrayLike, *, *axis:* *int* | *None* = *None*) → int

Return the index of the maximum element of the 1D array

ulab.numpy.**argmin**(*array:* _ArrayLike, *, *axis:* *int* | *None* = *None*) → int

Return the index of the minimum element of the 1D array

ulab.numpy.**argsort**(*array:* ndarray, *, *axis:* *int* = *-1*) → *ndarray*

Returns an array which gives indices into the input array from least to greatest.

ulab.numpy.**cross**(*a:* ndarray, *b:* ndarray) → *ndarray*

Return the cross product of two vectors of length 3

ulab.numpy.**diff**(*array:* ndarray, *, *n:* *int* = *1*, *axis:* *int* = *-1*) → *ndarray*

Return the numerical derivative of successive elements of the array, as an array. axis=None is not supported.

ulab.numpy.**flip**(*array:* ndarray, *, *axis:* *int* | *None* = *None*) → *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.**max**(*array:* _ArrayLike, *, *axis:* *int* | *None* = *None*) → *float*

Return the maximum element of the 1D array

ulab.numpy.**mean**(*array:* _ArrayLike, *, *axis:* *int* | *None* = *None*) → *float*

Return the mean element of the 1D array, as a number if axis is None, otherwise as an array.

ulab.numpy.**median**(*array:* ndarray, *, *axis:* *int* = *-1*) → *ndarray*

Find the median value in an array along the given axis, or along all axes if axis is None.

ulab.numpy.**min**(*array:* _ArrayLike, *, *axis:* *int* | *None* = *None*) → *float*

Return the minimum element of the 1D array

ulab.numpy.**roll**(*array:* ndarray, *distance:* *int*, *, *axis:* *int* | *None* = *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.**sort**(*array:* ndarray, *, *axis:* *int* = *-1*) → *ndarray*

Sort the array along the given axis, or along all axes if axis is None. The array is modified in place.

ulab.numpy.**std**(*array:* _ArrayLike, *, *axis:* *int* | *None* = *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.**sum**(*array:* _ArrayLike, *, *axis:* *int* | *None* = *None*) → *float* | int | *ndarray*

Return the sum of the array, as a number if axis is None, otherwise as an array.

class ulab.numpy.**ndarray**

ulab.numpy.**get_printoptions**() → Dict[str, int]

Get printing options

ulab.numpy.**set_printoptions**(*threshold:* *int* | *None* = *None*, *edgeitems:* *int* | *None* = *None*) → None

Set printing options

ulab.numpy.**ndinfo**(*array:* ndarray) → None

ulab.numpy.**array**(*values:* ndarray | *Iterable[*float | bool | *Iterable[Any]]*, *, *dtype:* _DType = *ulab.numpy.float*) → *ndarray*

alternate constructor function for *ulab.numpy.ndarray*. Mirrors numpy.array

ulab.numpy.**trace**(*m:* ndarray) → *float*

Parameters
m – a square matrix

Compute the trace of the matrix, the sum of its diagonal elements.

ulab.numpy.**dot**(*m1:* ndarray, *m2:* ndarray) → *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.**acos**(*a:* _ArrayLike) → *ndarray*
Computes the inverse cosine function

ulab.numpy.**acosh**(*a:* _ArrayLike) → *ndarray*
Computes the inverse hyperbolic cosine function

ulab.numpy.**asin**(*a:* _ArrayLike) → *ndarray*
Computes the inverse sine function

ulab.numpy.**asinh**(*a:* _ArrayLike) → *ndarray*
Computes the inverse hyperbolic sine function

ulab.numpy.**around**(*a:* _ArrayLike, *, *decimals:* *int* = *0*) → *ndarray*
Returns a new float array in which each element is rounded to `decimals` places.

ulab.numpy.**atan**(*a:* _ArrayLike) → *ndarray*
Computes the inverse tangent function; the return values are in the range [-pi/2,pi/2].

ulab.numpy.**atanh**(*a:* _ArrayLike) → *ndarray*
Computes the inverse hyperbolic tangent function

ulab.numpy.**arctan2**(*ya:* _ArrayLike, *xa:* _ArrayLike) → *ndarray*
Computes the inverse tangent function of y/x; the return values are in the range [-pi, pi].

ulab.numpy.**ceil**(*a:* _ArrayLike) → *ndarray*
Rounds numbers up to the next whole number

ulab.numpy.**cos**(*a:* _ArrayLike) → *ndarray*
Computes the cosine function

ulab.numpy.**cosh**(*a:* _ArrayLike) → *ndarray*
Computes the hyperbolic cosine function

ulab.numpy.**degrees**(*a:* _ArrayLike) → *ndarray*
Converts angles from radians to degrees

ulab.numpy.**erf**(*a:* _ArrayLike) → *ndarray*
Computes the error function, which has applications in statistics

ulab.numpy.**erfc**(*a:* _ArrayLike) → *ndarray*
Computes the complementary error function, which has applications in statistics

ulab.numpy.**exp**(*a:* _ArrayLike) → *ndarray*
Computes the exponent function.

ulab.numpy.**expm1**(*a:* _ArrayLike) → *ndarray*

Computes e^x-1. In certain applications, using this function preserves numeric accuracy better than the *exp* function.

ulab.numpy.**floor**(*a:* _ArrayLike) → *ndarray*

Rounds numbers up to the next whole number

ulab.numpy.**gamma**(*a:* _ArrayLike) → *ndarray*

Computes the gamma function

ulab.numpy.**lgamma**(*a:* _ArrayLike) → *ndarray*

Computes the natural log of the gamma function

ulab.numpy.**log**(*a:* _ArrayLike) → *ndarray*

Computes the natural log

ulab.numpy.**log10**(*a:* _ArrayLike) → *ndarray*

Computes the log base 10

ulab.numpy.**log2**(*a:* _ArrayLike) → *ndarray*

Computes the log base 2

ulab.numpy.**radians**(*a:* _ArrayLike) → *ndarray*

Converts angles from degrees to radians

ulab.numpy.**sin**(*a:* _ArrayLike) → *ndarray*

Computes the sine function

ulab.numpy.**sinc**(*a:* _ArrayLike) → *ndarray*

Computes the normalized sinc function

ulab.numpy.**sinh**(*a:* _ArrayLike) → *ndarray*

Computes the hyperbolic sine

ulab.numpy.**sqrt**(*a:* _ArrayLike) → *ndarray*

Computes the square root

ulab.numpy.**tan**(*a:* _ArrayLike) → *ndarray*

Computes the tangent

ulab.numpy.**tanh**(*a:* _ArrayLike) → *ndarray*

Computes the hyperbolic tangent

ulab.numpy.**vectorize**(*f: Callable[[int], float] | Callable[[float], float]*, *, *otypes:* _DType | *None* = *None*) → Callable[[*_ArrayLike*], *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.

12.91.2 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.

12.91.3 ulab.user – This module should hold arbitrary user-defined functions.

12.91.4 ulab.utils

ulab.utils.**spectrogram**(*r:* ulab.numpy.ndarray) → *ulab.numpy.ndarray*

Parameters

r (ulab.numpy.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.welch` https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.welch.html.

12.92 usb – PyUSB-compatible USB host API

The *usb* is a subset of PyUSB that allows you to communicate to USB devices.

12.92.1 usb.core – USB Core

This is a subset of the PyUSB core module.

exception usb.core.**USBError**

Bases: OSError

Catchall exception for USB related errors.

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

exception usb.core.**USBTimeoutError**

Bases: *USBError*

Raised when a USB transfer times out.

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

usb.core.**find**(*find_all: bool = False*, ***, *idVendor: int | None = None*, *idProduct: int | None = None*) → *Device*

Find the first device that matches the given requirements or, if find_all is True, return a generator of all matching devices.

Returns None if no device matches.

class usb.core.**Device**

User code cannot create Device objects. Instead, get them from *usb.core.find*.

idVendor: int

The USB vendor ID of the device

idProduct: int

The USB product ID of the device

serial_number: str

The USB device's serial number string.

product: str

The USB device's product string.

manufacturer: str

The USB device's manufacturer string.

set_configuration(*configuration=1*)

Set the active configuration.

The configuration parameter is the bConfigurationValue field of the configuration you want to set as active. If you call this method without parameter, it will use the first configuration found. As a device hardly ever has more than one configuration, calling the method without arguments is enough to get the device ready.

write(*endpoint: int*, *data: circuitpython_typing.ReadableBuffer*, *timeout: int | None = None*) → int

Write data to a specific endpoint on the device.

Parameters

- **endpoint** (*int*) – the bEndpointAddress you want to communicate with.
- **data** (*ReadableBuffer*) – the data to send
- **timeout** (*int*) – Time to wait specified in milliseconds. (Different from most CircuitPython!)

Returns
the number of bytes written

read(*endpoint: int*, *size_or_buffer:* array.array, *timeout: int | None = None*) → int

Read data from the endpoint.

Parameters

- **endpoint** (*int*) – the bEndpointAddress you want to communicate with.
- **size_or_buffer** (*array.array*) – the array to read data into. PyUSB also allows size but CircuitPython only support array to force deliberate memory use.
- **timeout** (*int*) – Time to wait specified in milliseconds. (Different from most CircuitPython!)

Returns
the number of bytes read

ctrl_transfer(*bmRequestType: int*, *bRequest: int*, *wValue: int = 0*, *wIndex: int = 0*, *data_or_wLength:* array.array *| None = None*, *timeout: int | None = None*) → int

Do a control transfer on the endpoint 0. The parameters bmRequestType, bRequest, wValue and wIndex are the same of the USB Standard Control Request format.

Control requests may or may not have a data payload to write/read. In cases which it has, the direction bit of the bmRequestType field is used to infer the desired request direction.

For host to device requests (OUT), data_or_wLength parameter is the data payload to send, and it must be a sequence type convertible to an array object. In this case, the return value is the number of bytes written in the data payload.

For device to host requests (IN), data_or_wLength is an array object which the data will be read to, and the return value is the number of bytes read.

is_kernel_driver_active(*interface: int*) → bool

Determine if CircuitPython is using the interface. If it is, the object will be unable to perform I/O.

Parameters
interface (*int*) – the device interface number to check

detach_kernel_driver(*interface: int*) → None

Stop CircuitPython from using the interface. If successful, you will then be able to perform I/O. CircuitPython will automatically re-start using the interface on reload.

Parameters
interface (*int*) – the device interface number to stop CircuitPython on

attach_kernel_driver(*interface: int*) → None

Allow CircuitPython to use the interface if it wants to.

Parameters
interface (*int*) – the device interface number to allow CircuitPython to use

12.93 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**: *Serial* | None

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**: *Serial* | None

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: float | None

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: float | None

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(*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(*buf: circuitpython_typing.WriteableBuffer*) → int

Read bytes into the buf. Read at most len(buf) bytes. If *timeout* is > 0 or None, keep waiting until the timeout expires or len(buf) bytes are available.

Returns
number of bytes read and stored into buf

Return type
int

readline(*size: int = -1*) → bytes | None

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() → List[bytes | None]

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(*buf: circuitpython_typing.ReadableBuffer*) → int

Write as many bytes as possible from the buffer of bytes.

Returns
the number of bytes written

Return type
int

flush() → None

Force out any unwritten bytes, waiting until they are written.

reset_input_buffer() → None

Clears any unread bytes.

reset_output_buffer() → None

Clears any unwritten bytes.

12.94 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.

If a boot device is enabled by *usb_hid.enable()*, *and* the host has requested a boot device, the *devices* tuple is **replaced** when `code.py` starts with a single-element tuple containing a *Device* that describes the boot device chosen (keyboard or mouse). The request for a boot device overrides any other HID devices.

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: Sequence[*Device*] | None*, *boot_device: int = 0*) → 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.
- **boot_device** (*int*) – If non-zero, inform the host that support for a a boot HID device is available. If `boot_device=1`, a boot keyboard is available. If `boot_device=2`, a boot mouse is available. No other values are allowed. See below.

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.

Boot Devices

Boot devices implement a fixed, predefined report descriptor, defined in https://www.usb.org/sites/default/files/hid1_12.pdf, Appendix B. A USB host can request to use the boot device if the USB device says it is available. Usually only a BIOS or other kind of limited-functionality host needs boot keyboard support.

For example, to make a boot keyboard available, you can use this code:

```
usb_hid.enable((Device.KEYBOARD), boot_device=1)  # 1 for a keyboard
```

If the host requests the boot keyboard, the report descriptor provided by *Device.KEYBOARD* will be ignored, and the predefined report descriptor will be used. But if the host does not request the boot keyboard, the descriptor provided by *Device.KEYBOARD* will be used.

The HID boot device must usually be the first or only device presented by CircuitPython. The HID device will be USB interface number 0. To make sure it is the first device, disable other USB devices, including CDC and MSC (CIRCUITPY). If you specify a non-zero `boot_device`, and it is not the first device, CircuitPython will enter safe mode to report this error.

usb_hid.**get_boot_device**() → int

Returns

the boot device requested by the host, if any. Returns 0 if the host did not request a boot device, or if *usb_hid.enable()* was called with `boot_device=0`, the default, which disables boot device support. If the host did request a boot device, returns the value of `boot_device` set in *usb_hid.enable()*: 1 for a boot keyboard, or 2 for boot mouse. However, the standard devices provided by CircuitPython, *Device.KEYBOARD* and *Device.MOUSE*, describe reports that match the boot device reports, so you don't need to check this if you are using those devices.

Rtype int

usb_hid.**set_interface_name**(*interface_name: str*) → None

Override HID interface name in the USB Interface Descriptor.

`interface_name` must be an ASCII string (or buffer) of at most 126.

This method must be called in boot.py to have any effect.

Not available on boards without native USB support.

class usb_hid.**Device**(*, *report_descriptor: circuitpython_typing.ReadableBuffer*, *usage_page: int*, *usage: int*, *report_ids: Sequence[int]*, *in_report_lengths: Sequence[int]*, *out_report_lengths: Sequence[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.
- **report_ids** (*Sequence[int]*) – Sequence of report ids used by the descriptor. If the `report_descriptor` does not specify any report IDs, use `(0,)`.
- **in_report_lengths** (*Sequence[int]*) – Sequence of sizes in bytes of the HID reports sent to the host. The sizes are in order of the `report_ids`. Use a size of `0` for a report that is not an IN report. "IN" is with respect to the host.
- **out_report_lengths** (*int*) – Sequence of sizes in bytes of the HID reports received from the host. The sizes are in order of the `report_ids`. Use a size of `0` for a report that is not an OUT report. "OUT" is with respect to the host.

`report_ids`, `in_report_lengths`, and `out_report_lengths` must all have the same number of elements.

Here is an example of a *Device* with a descriptor that specifies two report IDs, 3 and 4. Report ID 3 sends an IN report of length 5, and receives an OUT report of length 6. Report ID 4 sends an IN report of length 2, and does not receive an OUT report:

```
device = usb_hid.Device(
    descriptor=b"...",          # Omitted for brevity.
    report_ids=(3, 4),
    in_report_lengths=(5, 2),
    out_report_lengths=(6, 0),
)
```

The HID device is able to wake up a suspended (sleeping) host computer. See *send_report()* for details.

KEYBOARD: *Device*

Standard keyboard device supporting keycodes 0x00-0xDD, modifiers 0xE-0xE7, and five LED indicators. Uses Report ID 1 for its IN and OUT reports.

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. Uses Report ID 2 for its IN report.

CONSUMER_CONTROL: *Device*

Consumer Control device supporting sent values from 1-652, with no rollover. Uses Report ID 3 for its IN report.

usage_page: int

The device usage page identifier, which designates a category of device. (read-only)

usage: int

The device usage identifier, which designates a specific kind of device. (read-only)

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

send_report(*report: circuitpython_typing.ReadableBuffer*, *report_id: int | None = None*) → None

Send an HID report. If the device descriptor specifies zero or one report id's, you can supply `None` (the default) as the value of `report_id`. Otherwise you must specify which report id to use when sending the report.

If the USB host is suspended (sleeping), then *send_report()* will request that the host wake up. The `report` itself will be discarded, to prevent unwanted extraneous characters, mouse clicks, etc.

Note: Host operating systems allow enabling and disabling specific devices and kinds of devices to do wakeup. The defaults are different for different operating systems. For instance, on Linux, only the primary keyboard may be enabled. In addition, there may be USB wakeup settings in the host computer BIOS/UEFI.

get_last_received_report(*report_id: int | None = None*) → bytes | None

Get the last received HID OUT or feature report for the given report ID. The report ID may be omitted if there is no report ID, or only one report ID. Return `None` if nothing received. After returning a report, subsequent calls will return `None` until next report is received.

12.95 usb_host – USB Host

The *usb_host* module allows you to manage USB host ports. To communicate with devices use the *usb* module that is a subset of PyUSB's API.

usb_host.**set_user_keymap**(*keymap: circuitpython_typing.ReadableBuffer*, /) → None

Set the keymap used by a USB HID keyboard in kernel mode

The keymap consists of 256 or 384 1-byte entries that map from USB keycodes to ASCII codes. The first 128 entries are for unmodified keys, the next 128 entries are for shifted keys,and the next optional 128 entries are for altgr-modified keys.

The values must all be ASCII (32 through 126 inclusive); other values are not valid.

The values at index 0, 128, and 256 (if the keymap has 384 entries) must be 0; other values are reserved for future expansion to indicate alternate keymap formats.

At other indices, the value 0 is used to indicate that the normal definition is still used. For instance, the entry for HID_KEY_ARROW_UP (0x52) is usually 0 so that the default behavior of sending an escape code is preserved.

This function is a CircuitPython extension not present in PyUSB

class usb_host.**Port**(*dp:* microcontroller.Pin, *dm:* microcontroller.Pin)

USB host port. Also known as a root hub port.

Create a USB host port on the given pins. Access attached devices through the *usb* module.

The resulting object lives longer than the CircuitPython VM so that USB devices such as keyboards can continue to be used. Subsequent calls to this constructor will return the same object and *not* reinitialize the USB host port. It will raise an exception when given different arguments from the first successful call.

Parameters

- **dp** (Pin) – The data plus pin
- **dm** (Pin) – The data minus pin

12.96 usb_midi – MIDI over USB

The *usb_midi* module contains classes to transmit and receive MIDI messages over USB.

usb_midi.**ports**: **Tuple[*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(*nbytes: int | None = None*) → bytes | None

Read characters. If `nbytes` is specified then read at most that many bytes. Otherwise, read everything that arrives until the connection times out. Providing the number of bytes expected is highly recommended because it will be faster.

Returns
Data read

Return type
bytes or None

readinto(*buf: circuitpython_typing.WriteableBuffer*, *nbytes: int | None = None*) → bytes | None

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

Returns
number of bytes read and stored into `buf`

Return type
bytes or None

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(*buf: circuitpython_typing.ReadableBuffer*) → int | None

Write the buffer of bytes to the bus.

Returns
the number of bytes written

Return type
int or None

12.97 usb_video – Allows streaming bitmaps to a host computer via USB

This makes your CircuitPython device identify to the host computer as a video camera. This mode is also known as "USB UVC".

This mode requires 1 IN endpoint. Generally, microcontrollers have a limit on the number of endpoints. If you exceed the number of endpoints, CircuitPython will automatically enter Safe Mode. Even in this case, you may be able to enable USB video by also disabling other USB functions, such as *usb_hid* or *usb_midi*.

To enable this mode, you must configure the framebuffer size in `boot.py` and then create a display in `code.py`.

```
# boot.py
import usb_video
usb_video.enable_framebuffer(128, 96)
```

```
# code.py
import usb_video
import framebufferio
import displayio

displayio.release_displays()
display = framebufferio.FramebufferDisplay(usb_video.USBFramebuffer())

# ... use the display object with displayio Group and TileGrid objects
```

This interface is experimental and may change without notice even in stable versions of CircuitPython.

usb_video.**enable_framebuffer**(*width: int*, *height: int*) → None

Enable a USB video framebuffer, setting the given width & height

This function may only be used from `boot.py`.

Width is rounded up to a multiple of 2.

After boot.py completes, the framebuffer will be allocated. Total storage of 4×``width``×``height`` bytes is required, reducing the amount available for Python objects. If the allocation fails, a MemoryError is raised. This message can be seen in `boot_out.txt`.

class usb_video.**USBFramebuffer**

Displays to a USB connected computer using the UVC protocol

The data in the framebuffer is in RGB565_SWAPPED format.

This object is most often used with *framebufferio.FramebufferDisplay*. However, it also supports the `WritableBuffer` protocol and can be accessed as an array of `H` (unsigned 16-bit values).

Returns the singleton framebuffer object, if USB video is enabled

width: int

The width of the display, in pixels

height: int

The height of the display, in pixels

refresh() → None

Transmits the color data in the buffer to the pixels so that they are shown.

12.98 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.

12.99 vectorio – Lightweight 2D shapes for displays

The *vectorio* module provide simple filled drawing primitives for use with *displayio*.

```
group = displayio.Group()

palette = displayio.Palette(1)
palette[0] = 0x125690

circle = vectorio.Circle(pixel_shader=palette, radius=25, x=70, y=40)
group.append(circle)

rectangle = vectorio.Rectangle(pixel_shader=palette, width=40, height=30, x=55, y=45)
group.append(rectangle)

points=[(5, 5), (100, 20), (20, 20), (20, 100)]
polygon = vectorio.Polygon(pixel_shader=palette, points=points, x=0, y=0)
group.append(polygon)
```

class vectorio.**Circle**(*pixel_shader:* displayio.ColorConverter | displayio.Palette, *radius: int*, *x: int*, *y: int*)

Circle is positioned on screen by its center point.

Parameters

- **pixel_shader** (*Union[ColorConverter,Palette]*) – The pixel shader that produces colors from values
- **radius** (*int*) – The radius of the circle in pixels
- **x** (*int*) – Initial x position of the axis.
- **y** (*int*) – Initial y position of the axis.
- **color_index** (*int*) – Initial color_index to use when selecting color from the palette.

radius: int

The radius of the circle in pixels.

color_index: int

The color_index of the circle as 0 based index of the palette.

x: int

X position of the center point of the circle in the parent.

y: int

Y position of the center point of the circle in the parent.

hidden: bool

Hide the circle or not.

location: Tuple[int, int]

(X,Y) position of the center point of the circle in the parent.

pixel_shader: *displayio.ColorConverter* | *displayio.Palette*

The pixel shader of the circle.

class vectorio.**Polygon**(*pixel_shader:* displayio.ColorConverter | displayio.Palette, *points: List[Tuple[int, int]]*, *x: int*, *y: int*)

Represents a closed shape by ordered vertices. The path will be treated as 'closed', the last point will connect to the first point.

Parameters

- **pixel_shader** (*Union[ColorConverter,Palette]*) – The pixel shader that produces colors from values
- **points** (*List[Tuple[int,int]]*) – Vertices for the polygon
- **x** (*int*) – Initial screen x position of the 0,0 origin in the points list.
- **y** (*int*) – Initial screen y position of the 0,0 origin in the points list.
- **color_index** (*int*) – Initial color_index to use when selecting color from the palette.

points: List[Tuple[int, int]]

Vertices for the polygon.

color_index: int

The color_index of the polygon as 0 based index of the palette.

x: int

X position of the 0,0 origin in the points list.

y: int

Y position of the 0,0 origin in the points list.

hidden: bool

Hide the polygon or not.

location: Tuple[int, int]

(X,Y) position of the 0,0 origin in the points list.

pixel_shader: *displayio.ColorConverter* | *displayio.Palette*

The pixel shader of the polygon.

class vectorio.**Rectangle**(*pixel_shader:* displayio.ColorConverter | displayio.Palette, *width: int*, *height: int*, *x: int*, *y: int*)

Represents a rectangle by defining its bounds

Parameters

- **pixel_shader** (*Union[ColorConverter,Palette]*) – The pixel shader that produces colors from values
- **width** (*int*) – The number of pixels wide
- **height** (*int*) – The number of pixels high
- **x** (*int*) – Initial x position of the top left corner.
- **y** (*int*) – Initial y position of the top left corner.
- **color_index** (*int*) – Initial color_index to use when selecting color from the palette.

width: int
The width of the rectangle in pixels.

height: int
The height of the rectangle in pixels.

color_index: int
The color_index of the rectangle in 1 based index of the palette.

x: int
X position of the top left corner of the rectangle in the parent.

y: int
Y position of the top left corner of the rectangle in the parent.

hidden: bool
Hide the rectangle or not.

location: Tuple[int, int]
(X,Y) position of the top left corner of the rectangle in the parent.

pixel_shader: *displayio.ColorConverter* | *displayio.Palette*
The pixel shader of the rectangle.

12.100 warnings – Warn about potential code issues.

This is a slimmed down version of the full CPython module. It defaults to the "always" action instead of "default", which prints once per occurrence. Only "error" and "ignore" are also supported. No filtering on category is available.

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

warnings.**warn**(*message: str*, *category: type* = *Warning*) → None
Issue a warning with an optional category. Use *simplefilter()* to set if warnings are ignored, printed or raise an exception.

warnings.**simplefilter**(*action: str*) → None
Set the action to take on all warnings. This is a subset of the CPython behavior because it allows for per-category changes.

12.101 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()
```

exception watchdog.**WatchDogTimeout**

Bases: Exception

Exception raised when the watchdog timer is set to `WatchDogMode.RAISE` and expires.

Example:

```
import microcontroller
import watchdog
import time

wdt = microcontroller.watchdog
wdt.timeout = 5

while True:
    wdt.mode = watchdog.WatchDogMode.RAISE
    print("Starting loop -- should exit after five seconds")
    try:
        while True:
            time.sleep(10)  # Also works with pass
    except watchdog.WatchDogTimeout as e:
        print("Watchdog expired")
    except Exception as e:
        print("Other exception")

print("Exited loop")
```

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

class watchdog.**WatchDogMode**

Run state of the watchdog timer.

RAISE: *WatchDogMode*

Raise an exception when the *WatchDogTimer* expires.

RESET: *WatchDogMode*

Reset the system when 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.

Access the sole instance through *microcontroller.watchdog*.

timeout: float

The maximum number of seconds that can elapse between calls to *feed()*. Setting the timeout will also feed the watchdog.

mode: *WatchDogMode* | None

The current operating mode of the WatchDogTimer *watchdog.WatchDogMode* or None when the timer is disabled.

Setting a *WatchDogMode* activates the WatchDog:

```
from microcontroller import watchdog
from watchdog import WatchDogMode

watchdog.timeout = 5
watchdog.mode = WatchDogMode.RESET
```

Once set, the *WatchDogTimer* will perform the specified action if the timer expires.

feed() → None

Feed the watchdog timer. This must be called regularly, otherwise the timer will expire. Silently does nothing if the watchdog isn't active.

deinit() → None

Stop the watchdog timer.

Raises

RuntimeError – if the watchdog timer cannot be disabled on this platform.

Note: This is deprecated in 9.0.0 and will be removed in 10.0.0. Set watchdog *mode* to None instead.

12.102 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.**Monitor**(*channel: int | None = 1*, *queue: int | None = 128*)

For monitoring WiFi packets.

Initialize *wifi.Monitor* singleton.

Parameters

- **channel** (*int*) – The WiFi channel to scan.
- **queue** (*int*) – The queue size for buffering the packet.

channel: *int*

The WiFi channel to scan.

queue: *int*

The queue size for buffering the packet.

deinit() → None

De-initialize *wifi.Monitor* singleton.

lost() → int

Returns the packet loss count. The counter resets after each poll.

queued() → int

Returns the packet queued count.

packet() → dict

Returns the monitor packet.

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.**Packet**

The packet parameters.

CH: object

The packet's channel.

LEN: object

The packet's length.

RAW: object

The packet's payload.

RSSI: object

The packet's rssi.

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.

hostname: str | circuitpython_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.

mac_address: circuitpython_typing.ReadableBuffer

MAC address for the station. When the address is altered after interface is connected
the changes would only be reflected once the interface reconnects.

Limitations: Not settable on RP2040 CYW43 boards, such as Pi Pico W.

tx_power: float

Wifi transmission power, in dBm.

mac_address_ap: circuitpython_typing.ReadableBuffer

MAC address for the AP. When the address is altered after interface is started
the changes would only be reflected once the interface restarts.

Limitations: Not settable on RP2040 CYW43 boards, such as Pi Pico W.

ap_active: bool

True if running as an access point. (read-only)

connected: bool

True if connected to an access point (read-only).

ipv4_gateway: *ipaddress.IPv4Address* | None

IP v4 Address of the station gateway when connected to an access point. None otherwise. (read-only)

ipv4_gateway_ap: *ipaddress.IPv4Address* | None

IP v4 Address of the access point gateway, when enabled. None otherwise. (read-only)

ipv4_subnet: *ipaddress.IPv4Address* | None

IP v4 Address of the station subnet when connected to an access point. None otherwise. (read-only)

ipv4_subnet_ap: *ipaddress.IPv4Address* | None

IP v4 Address of the access point subnet, when enabled. None otherwise. (read-only)

ipv4_address: *ipaddress.IPv4Address* | None

IP v4 Address of the station when connected to an access point. None otherwise. (read-only)

ipv4_address_ap: *ipaddress.IPv4Address* | None

IP v4 Address of the access point, when enabled. None otherwise. (read-only)

ipv4_dns: *ipaddress.IPv4Address*

IP v4 Address of the DNS server to be used.

ap_info: *Network* | None

Network object containing BSSID, SSID, authmode, channel, country and RSSI when connected to an access point. None otherwise.

stations_ap: None

In AP mode, returns list of named tuples, each of which contains: mac: bytearray (read-only) rssi: int (read-only, None on Raspberry Pi Pico W) ipv4_address: ipv4_address (read-only, None if station connected but no address assigned yet or self-assigned address)

start_scanning_networks(*, *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.

Note: In the raspberrypi port (RP2040 CYW43), `start_channel` and `stop_channel` are ignored.

---

stop_scanning_networks() → None

Stop scanning for Wifi networks and free any resources used to do it.

start_station() → None

Starts a Station.

stop_station() → None

Stops the Station.

start_ap(*ssid: str | circuitpython_typing.ReadableBuffer*, *password: str | circuitpython_typing.ReadableBuffer = b''*, *, *channel: int = 1*, *authmode: Iterable[*AuthMode*] = ()*, *max_connections: int | None = 4*) → None

Starts running 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 not None, the access point will use the given authentication modes. If a non-empty password is given, `authmode` must not include `OPEN`. If `authmode` is not given or is an empty iterable, `(wifi.AuthMode.OPEN,)` will be used when the password is the empty string, otherwise `authmode` will be `(wifi.AuthMode.WPA, wifi.AuthMode.WPA2, wifi.AuthMode.PSK)`.

Limitations: On Espressif, `authmode` with a non-empty password must include *wifi.AuthMode.PSK*, and one or both of *wifi.AuthMode.WPA* and *wifi.AuthMode.WPA2*. On Pi Pico W, `authmode` is ignored; it is always `(wifi.AuthMode.WPA2, wifi.AuthMode.PSK)` with a non-empty password, or `(wifi.AuthMode.OPEN)`, when no password is given. On Pi Pico W, the AP can be started and stopped only once per reboot.

The length of `password` must be 8-63 characters if it is ASCII, or exactly 64 hexadecimal characters if it is the hex form of the 256-bit key.

If `max_connections` is given, the access point will allow up to that number of stations to connect.

Note: In the raspberrypi port (RP2040 CYW43), `max_connections` is ignored.

stop_ap() → None

Stops the access point.

connect(*ssid:* *str* | *circuitpython_typing.ReadableBuffer*, *password:* *str* | *circuitpython_typing.ReadableBuffer* = *b''*, *, *channel:* *int* = *0*, *bssid:* *str* | *circuitpython_typing.ReadableBuffer* | *None* = *None*, *timeout:* *float* | *None* = *None*) → None

Connects to the given ssid and waits for an ip address. Reconnections are handled automatically once one connection succeeds.

The length of `password` must be 0 if there is no password, 8-63 characters if it is ASCII, or exactly 64 hexadecimal characters if it is the hex form of the 256-bit key.

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 non-zero, 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 and not None, the scan will start at the first channel or the one given and connect to the AP with the given `bssid` and `ssid`.

set_ipv4_address(*, *ipv4:* ipaddress.IPv4Address, *netmask:* ipaddress.IPv4Address, *gateway:* ipaddress.IPv4Address, *ipv4_dns:* ipaddress.IPv4Address | *None*) → None

Sets the IP v4 address of the station. Must include the netmask and gateway. DNS address is optional. Setting the address manually will stop the DHCP client.

Note: In the raspberrypi port (RP2040 CYW43), the access point needs to be started before the IP v4 address can be set.

set_ipv4_address_ap(*, *ipv4:* ipaddress.IPv4Address, *netmask:* ipaddress.IPv4Address, *gateway:* ipaddress.IPv4Address) → None

Sets the IP v4 address of the access point. Must include the netmask and gateway.

start_dhcp() → None

Starts the station DHCP client.

stop_dhcp() → None

Stops the station DHCP client. Needed to assign a static IP address.

start_dhcp_ap() → None

Starts the access point DHCP server.

stop_dhcp_ap() → None

Stops the access point DHCP server. Needed to assign a static IP address.

ping(*ip:* ipaddress.IPv4Address, *, *timeout:* *float* | *None* = *0.5*) → float | None

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__() → Iterator[*Network*]

Returns itself since it is the iterator.

__next__() → *Network*

Returns the next *wifi.Network*. Raises *StopIteration* if scanning is finished and no other results are available.

12.103 zlib – zlib decompression functionality

The *zlib* module allows limited functionality similar to the CPython zlib library. This module allows to decompress binary data compressed with DEFLATE algorithm (commonly used in zlib library and gzip archiver). Compression is not yet implemented.

zlib.**decompress**(*data: bytes*, *wbits: int | None = 0*, *bufsize: int | None = 0*) → bytes

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.

The wbits parameter controls the size of the history buffer (or "window size"), and what header and trailer format is expected.

Common wbits values:

- To decompress deflate format, use wbits = -15
- To decompress zlib format, use wbits = 15
- To decompress gzip format, use wbits = 31

Parameters

- **data** (*bytes*) – data to be decompressed
- **wbits** (*int*) – DEFLATE dictionary window size used during compression. See above.
- **bufsize** (*int*) – ignored for compatibility with CPython only

12.104 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.

12.105 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 reuse 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 Collec-*

tor. 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-lib
MicroPython is (usually) distributed as a single executable/binary file with just few builtin modules. There is no extensive standard library comparable with *CPython*'s. Instead, there is a related, but separate project micropython-lib which provides implementations for many modules from CPython's standard library.

Some of the modules are implemented in pure Python, and are able to be used on all ports. However, the majority of these modules use *FFI* to access operating system functionality, and as such can only be used on the *MicroPython Unix port* (with limited support for Windows).

Unlike the *CPython* stdlib, micropython-lib modules are intended to be installed individually - either using manual copying or using *mip*.

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.

mip
A package installer for MicroPython (mip - "mip installs packages"). It installs MicroPython packages either from *micropython-lib*, GitHub, or arbitrary URLs. mip can be used on-device on network-capable boards, and internally by tools such as *mpremote*.

mpremote
A tool for interacting with a MicroPython device.

.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". A 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.

upip
A now-obsolete package manager for MicroPython, inspired by *CPython*'s pip, but much smaller and with reduced functionality. See its replacement, *mip*.

webrepl
A way of connecting to the REPL (and transferring files) on a device over the internet from a browser. See https://micropython.org/webrepl

12.106 Adafruit Community Code of Conduct

12.106.1 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.

12.106.2 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.

12.106.3 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.

12.106.4 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 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.

12.106.5 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.

12.106.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, 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.

12.107 MicroPython & CircuitPython License

CHAPTER

THIRTEEN

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