tannewt CircuitPython Documentation

Release delete_out_of_date

Feb 22, 2018

API and Usage

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

CHAPTER 1

Adafruit CircuitPython

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

1.1 Status

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

1.2 Supported Boards

1.2.1 Designed for CircuitPython

- Adafruit CircuitPlayground Express
- Adafruit Feather M0 Express
- Adafruit Metro M0 Express
- Adafruit Gemma M0

1.2.2 Other

- Adafruit Feather HUZZAH
- Adafruit Feather M0 Basic

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

1.3 Download

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

1.4 Documentation

Guides and videos are available through the Adafruit Learning System under the CircuitPython category and MicroPython category. An API reference is also available on Read the Docs.

1.5 Contributing

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

1.6 Differences from MicroPython

CircuitPython:

- includes a port for Atmel SAMD21 (Commonly known as M0 in Adafruit product names.)
- supports only Atmel SAMD21 and ESP8266 ports.
- tracks MicroPython's releases (not master).

1.6.1 Behavior

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

- Adds a safe mode that does not run user code after a hard crash or brown out. The hope is that this will make it easier to fix code that causes nasty crashes by making it available through mass storage after the crash. A reset (the button) is needed after its fixed to get back into normal mode.

1.6.2 API

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

1.6.3 Modules

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

1.6.4 atmel-samd21 features

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

1.7 Project Structure

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

1.7.1 Core

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

- `docs` High level user documentation in Sphinx reStructuredText format.
- `drivers` External device drivers written in Python.
- `examples` A few example Python scripts.
- `extmod` Shared C code used in multiple ports' modules.
- `lib` Shared core C code including externally developed libraries such as FATFS.
- `logo` The MicroPython logo.

- `mpy-cross` A cross compiler that converts Python files to byte code prior to being run in MicroPython. Useful for reducing library size.
- `py` Core Python implementation, including compiler, runtime, and core library.
- `shared-bindings` Shared definition of Python modules, their docs and backing C APIs. Ports must implement the C API to support the corresponding module.
- `shared-module` Shared implementation of Python modules that may be based on `common-hal`.
- `tests` Test framework and test scripts.
- `tools` Various tools, including the pyboard.py module.

1.7.2 Ports

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

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

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

back to top

1.8 Full Table of Contents

1.8.1 Supported Ports

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

1.8.2 Troubleshooting

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

File system issues

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

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

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

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

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

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

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

ValueError: Incompatible .mpy file.

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

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

1.8.3 Additional Adafruit Libraries and Drivers on GitHub

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

Bundle

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

The bundles are available on GitHub.

To install them:

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

Foundational Libraries

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

Helper Libraries

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

Drivers

Drivers provide easy access to sensors and other chips without requiring a knowledge of the interface details of the chip itself.

1.8.4 Design Guide

MicroPython has created a great foundation to build upon and to make it even better for beginners we've created CircuitPython. This guide covers a number of ways the core and libraries are geared towards beginners.

Start libraries with the cookiecutter

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

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).

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

Both should have the CircuitPython repository topic on GitHub.

Lifetime and ContextManagers

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

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

```
import digitalio
import board

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

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

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

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

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

```
import digitalio
import board

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

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

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

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

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

Verify your device

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

Getters/Setters

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

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

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

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

Here is more info on properties from Python.

Design for compatibility with CPython

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

Example

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

Document inline

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

Use Sphinx flavor rST for markup.

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

Module description

After the license comment:

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

Class description

Documenting what the object does:

```
class DS3231:
    """Interface to the DS3231 RTC."""
```

Renders as:

class DS3231
Interface to the DS3231 RTC.

Data descriptor description

Comment is after even though its weird:

```
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

Property description

Comment comes from the getter:

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

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

Renders as:

datetime
The current date and time

Use BusDevice

[BusDevice](https://github.com/adafruit/Adafruit_CircuitPython_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

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

    def __init__(self, i2c):
        # Always on address 0x40.
        self.i2c_device = i2c_device.I2CDevice(i2c, 0x40)
        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')
            i2c.readinto(self.buf)
        return self.buf[0]
```

Use composition

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

See also:

Wikipedia has more information on "dependency inversion".

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

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

Lots of small modules

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

Speed second

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

Avoid allocations in drivers

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

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

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

Examples

ustruct.pack

Use *ustruct.pack_into* instead of *ustruct.pack*.

Sensor properties and units

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

Property name	Python type	Units
`acceleration`	(float, float, float)	x, y, z meter per second per second
`magnetic`	(float, float, float)	micro-Tesla (uT)
`orientation`	(float, float, float)	x, y, z degrees
`gyro`	(float, float, float)	x, y, z radians per second
`temperature`	float	degrees centigrade
`distance`	float	centimeters
`light`	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

Common APIs

Outside of sensors, having common methods amongst drivers for similar devices such as devices can be really useful. Its early days however. For now, try to adhere to guidelines in this document. Once a design is settled on, add it as a subsection to this one.

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`.

MicroPython compatibility

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

1.8.5 Adding `*io` support to other ports

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

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

File layout

Common HAL related files are found in these locations:

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

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

Adding support

Modifying the build

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

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

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

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

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

Hooking the modules in

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

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

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

Implementing the Common HAL

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

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

Testing

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

- Adafruit repos with CircuitPython topic
- Adafruit driver bundle

1.8.6 MicroPython libraries

Warning: These modules are inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time.

Python standard libraries and micro-libraries

Builtin functions and exceptions

Warning: These builtins are inherited from MicroPython and may not work in CircuitPython as documented or at all! If work differently from CPython, then their behavior may change.

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

Functions and types

abs()

all()

any()

bin()

class bool

class bytearray

class bytes
|see_cpython| *bytes*.

callable()

chr()

classmethod()

compile()

class complex

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

class dict

dir()

divmod()

enumerate()

eval()

exec()

filter()

class float

class frozenset

getattr()

globals()

hasattr()

hash()

hex()

id()

input()

class int

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

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

isinstance()

issubclass()

iter()

len()

class list

locals()

map()

max()

class memoryview

min()

next()

class object

oct()

open()

ord()

pow()

print()

property()

range()

repr()

reversed()

round()

class set

setattr()

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

sorted()

staticmethod()

class str

sum()

super()

class tuple

type()

zip()

Exceptions

exception AssertionError

exception AttributeError

exception Exception

exception ImportError

exception IndexError

exception KeyboardInterrupt

exception KeyError

exception MemoryError

exception NameError

exception NotImplementedError

exception OSError
|see_cpython| *OSError*. MicroPython doesn't implement `errno` attribute, instead use the standard way to access exception arguments: `exc.args[0]`.

exception RuntimeError

exception StopIteration

exception SyntaxError

exception SystemExit
|see_cpython| `python:SystemExit`.

exception TypeError
|see_cpython| `python:TypeError`.

exception ValueError

exception ZeroDivisionError

`array` – arrays of numeric data

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:array`.

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

Classes

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

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

append(*val*)

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

extend(*iterable*)

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

gc – control the garbage collector

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:gc`.

Functions

`gc.`**enable**()

Enable automatic garbage collection.

`gc.`**disable**()

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

`gc.`**collect**()

Run a garbage collection.

`gc.`**mem_alloc**()

Return the number of bytes of heap RAM that are allocated.

Difference to CPython

This function is MicroPython extension.

`gc.`**mem_free**()

Return the number of bytes of available heap RAM, or -1 if this amount is not known.

Difference to CPython

This function is 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.

---

math – mathematical functions

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:math`.

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

Note: On the pyboard, floating-point numbers have 32-bit precision.

Availability: not available on WiPy. Floating point support required for this module.

Functions

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

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

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

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

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

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

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

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

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

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

math.**cosh**(*x*)
Return the hyperbolic cosine of x.

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

math.**erf**(*x*)
Return the error function of x.

math.**erfc**(*x*)
Return the complementary error function of x.

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

math.**expm1**(*x*)
Return `exp(x) - 1`.

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

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

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

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

math.**gamma**(*x*)
Return the gamma function of x.

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

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

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

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

math.**lgamma**(*x*)
Return the natural logarithm of the gamma function of x.

math.**log**(*x*)
Return the natural logarithm of x.

math.**log10**(*x*)
Return the base-10 logarithm of x.

math.**log2**(*x*)
Return the base-2 logarithm of x.

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

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

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

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

math.**sinh**(*x*)
Return the hyperbolic sine of x.

math.**sqrt**(*x*)
Return the square root of x.

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

math.**tanh**(*x*)
Return the hyperbolic tangent of x.

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

Constants

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

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

sys – system specific functions

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| cpython:sys.

Functions

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

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

Difference to CPython

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

Constants

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

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

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

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

This object is the recommended way to distinguish MicroPython 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 MicroPython.

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

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

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

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

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

`sys.`**`platform`**
The platform that MicroPython 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 a board, e.g. `"pyboard"` for the original MicroPython reference board. It thus can be used to distinguish one board from another. If you need to check whether your program runs on MicroPython (vs other Python implementation), use *sys.implementation* instead.

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

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

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

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

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

`ubinascii` – binary/ASCII conversions

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:binascii`.

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

Functions

`ubinascii.`**`hexlify`**(*data*[, *sep*])
Convert binary data to hexadecimal representation. Returns bytes string.

Difference to CPython

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

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

`ubinascii.`**`a2b_base64`**(*data*)
Decode base64-encoded data, ignoring invalid characters in the input. Conforms to RFC 2045 s.6.8. Returns a bytes object.

`ubinascii.`**`b2a_base64`**(*data*)
Encode binary data in base64 format, as in RFC 3548. Returns the encoded data followed by a newline character, as a bytes object.

`ucollections` – collection and container types

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:collections`.

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

Classes

`ucollections.`**`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 ucollections import namedtuple

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

`ucollections.`**`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 ucollections import OrderedDict

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

Output:

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

`uerrno` – system error codes

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:errno`.

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

Constants

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

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

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

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

`uhashlib` – hashing algorithms

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:hashlib`.

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

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

Constructors

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

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

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

Methods

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

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

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

uheapq – heap queue algorithm

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:heapq`.

This module implements the heap queue algorithm.

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

Functions

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

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

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

uio – input/output streams

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:io`.

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

Conceptual hierarchy

Difference to CPython

Conceptual hierarchy of stream base classes is simplified in MicroPython, as described in this section.

(Abstract) base stream classes, which serve as a foundation for behavior of all the concrete classes, adhere to few dichotomies (pair-wise classifications) in CPython. In MicroPython, they are somewhat simplified and made implicit to achieve higher efficiencies and save resources.

An important dichotomy in CPython is unbuffered vs buffered streams. In MicroPython, all streams are currently unbuffered. This is because all modern OSes, and even many RTOSes and filesystem drivers already perform buffering on their side. Adding another layer of buffering is counter- productive (an issue known as "bufferbloat") and takes precious memory. Note that there still cases where buffering may be useful, so we may introduce optional buffering support at a later time.

But in CPython, another important dichotomy is tied with "bufferedness" - it's whether a stream may incur short read/writes or not. A short read is when a user asks e.g. 10 bytes from a stream, but gets less, similarly for writes. In CPython, unbuffered streams are automatically short operation susceptible, while buffered are guarantee against them. The no short read/writes is an important trait, as it allows to develop more concise and efficient programs - something which is highly desirable for MicroPython. So, while MicroPython doesn't support buffered streams, it still provides for no-short-operations streams. Whether there will be short operations or not depends on each particular class' needs, but developers are strongly advised to favor no-short-operations behavior for the reasons stated above. For example, MicroPython sockets are guaranteed to avoid short read/writes. Actually, at this time, there is no example of a short-operations stream class in the core, and one would be a port-specific class, where such a need is governed by hardware peculiarities.

The no-short-operations behavior gets tricky in case of non-blocking streams, blocking vs non-blocking behavior being another CPython dichotomy, fully supported by MicroPython. Non-blocking streams never wait for data either to arrive or be written - they read/write whatever possible, or signal lack of data (or ability to write data). Clearly, this conflicts with "no-short-operations" policy, and indeed, a case of non-blocking buffered (and this no-short-ops) streams is convoluted in CPython - in some places, such combination is prohibited, in some it's undefined or just not documented, in some cases it raises verbose exceptions. The matter is much simpler in MicroPython: non-blocking stream are important for efficient asynchronous operations, so this property prevails on the "no-short-ops" one. So, while blocking streams will avoid short reads/writes whenever possible (the only case to get a short read is if end of file is reached, or in case of error (but errors don't return short data, but raise exceptions)), non-blocking streams may produce short data to avoid blocking the operation.

The final dichotomy is binary vs text streams. MicroPython of course supports these, but while in CPython text streams are inherently buffered, they aren't in MicroPython. (Indeed, that's one of the cases for which we may introduce buffering support.)

Note that for efficiency, MicroPython doesn't provide abstract base classes corresponding to the hierarchy above, and it's not possible to implement, or subclass, a stream class in pure Python.

Functions

`uio.`**`open`**(*name*, *mode='r'*, ***kwargs*)

Open a file. Builtin `open()` function is aliased to this function. All ports (which provide access to file system) are required to support `mode` parameter, but support for other arguments vary by port.

Classes

class uio.**FileIO**(...)
This is type of a file open in binary mode, e.g. using `open(name, "rb")`. You should not instantiate this class directly.

class uio.**TextIOWrapper**(...)
This is type of a file open in text mode, e.g. using `open(name, "rt")`. You should not instantiate this class directly.

class uio.**StringIO**([*string*])

class uio.**BytesIO**([*string*])
In-memory file-like objects for input/output. *StringIO* is used for text-mode I/O (similar to a normal file opened with "t" modifier). *BytesIO* is used for binary-mode I/O (similar to a normal file opened with "b" modifier). Initial contents of file-like objects can be specified with *string* parameter (should be normal string for *StringIO* or bytes object for *BytesIO*). All the usual file methods like `read()`, `write()`, `seek()`, `flush()`, `close()` are available on these objects, and additionally, a following method:

getvalue()
Get the current contents of the underlying buffer which holds data.

ujson – JSON encoding and decoding

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:json`.

This modules allows to convert between Python objects and the JSON data format.

Functions

ujson.**dumps**(*obj*)
Return `obj` represented as a JSON string.

ujson.**loads**(*str*)
Parse the JSON `str` and return an object. Raises ValueError if the string is not correctly formed.

ure – simple regular expressions

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:re`.

This module implements regular expression operations. Regular expression syntax supported is a subset of CPython `re` module (and actually is a subset of POSIX extended regular expressions).

Supported operators are:

`'.'` Match any character.

`'[]'` Match set of characters. Individual characters and ranges are supported.

`'^'`

`'$'`

`'?'`

`'*'`

`'+'`

`'??'`

`'*?'`

`'+?'`

`'()'` Grouping. Each group is capturing (a substring it captures can be accessed with *match.group()* method).

Counted repetitions (`{m,n}`), more advanced assertions, named groups, etc. are not supported.

Functions

`ure.`**`compile`**`(`*regex_str*`)`
Compile regular expression, return *regex* <*regex*> object.

`ure.`**`match`**`(`*regex_str*, *string*`)`
Compile *regex_str* and match against *string*. Match always happens from starting position in a string.

`ure.`**`search`**`(`*regex_str*, *string*`)`
Compile *regex_str* and search it in a *string*. Unlike *match*, this will search string for first position which matches regex (which still may be 0 if regex is anchored).

`ure.`**`DEBUG`**
Flag value, display debug information about compiled expression.

Regex objects

Compiled regular expression. Instances of this class are created using *ure.compile()*.

`regex.`**`match`**`(`*string*`)`
`regex.`**`search`**`(`*string*`)`
Similar to the module-level functions *match()* and *search()*. Using methods is (much) more efficient if the same regex is applied to multiple strings.

`regex.`**`split`**`(`*string*, *max_split=-1*`)`
Split a *string* using regex. If *max_split* is given, it specifies maximum number of splits to perform. Returns list of strings (there may be up to *max_split+1* elements if it's specified).

Match objects

Match objects as returned by *match()* and *search()* methods.

`match.`**`group`**`(`[*index*]`)`
Return matching (sub)string. *index* is 0 for entire match, 1 and above for each capturing group. Only numeric groups are supported.

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

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:select`.

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

Functions

`uselect.`**`poll`**`()`
Create an instance of the Poll class.

`uselect.`**`select`**`(`*rlist*, *wlist*, *xlist*[, *timeout*]`)`
Wait for activity on a set of objects.

This function is provided by some MicroPython ports for compatibility and is not efficient. Usage of `Poll` is recommended instead.

class `Poll`

Methods

`poll.`**`register`**`(`*obj*[, *eventmask*]`)`
Register *obj* for polling. *eventmask* is logical OR of:

- `select.POLLIN` - data available for reading
- `select.POLLOUT` - more data can be written
- `select.POLLERR` - error occurred
- `select.POLLHUP` - end of stream/connection termination detected

eventmask defaults to `select.POLLIN | select.POLLOUT`.

`poll.`**`unregister`**`(`*obj*`)`
Unregister *obj* from polling.

`poll.`**`modify`**`(`*obj*, *eventmask*`)`
Modify the *eventmask* for *obj*.

`poll.`**`poll`**`(`[*timeout*]`)`
Wait for at least one of the registered objects to become ready. Returns list of `(obj, event, ...)` tuples, `event` element specifies which events happened with a stream and is a combination of `select.POLL*` constants described above. There may be other elements in tuple, depending on a platform and version, so don't assume that its size is 2. In case of timeout, an empty list is returned.

Timeout is in milliseconds.

Difference to CPython

Tuples returned may contain more than 2 elements as described above.

`poll.`**`ipoll`**`(`[*timeout*]`)`

Like *poll.poll()*, but instead returns an iterator which yields callee-owned tuples. This function provides efficient, allocation-free way to poll on streams.

Difference to CPython

This function is a MicroPython extension.

`usocket` – socket module

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:socket`.

This module provides access to the BSD socket interface.

Difference to CPython

For efficiency and consistency, socket objects in MicroPython implement a stream (file-like) interface directly. In CPython, you need to convert a socket to a file-like object using *makefile()* method. This method is still supported by MicroPython (but is a no-op), so where compatibility with CPython matters, be sure to use it.

Socket address format(s)

The native socket address format of the `usocket` module is an opaque data type returned by *getaddrinfo* function, which must be used to resolve textual address (including numeric addresses):

```
sockaddr = usocket.getaddrinfo('www.micropython.org', 80)[0][-1]
# You must use getaddrinfo() even for numeric addresses
sockaddr = usocket.getaddrinfo('127.0.0.1', 80)[0][-1]
# Now you can use that address
sock.connect(addr)
```

Using *getaddrinfo* is the most efficient (both in terms of memory and processing power) and portable way to work with addresses.

However, `socket` module (note the difference with native MicroPython `usocket` module described here) provides CPython-compatible way to specify addresses using tuples, as described below.

Summing up:

- Always use *getaddrinfo* when writing portable applications.
- Tuple addresses described below can be used as a shortcut for quick hacks and interactive use, if your port supports them.

Tuple address format for `socket` module:

- IPv4: *(ipv4_address, port)*, where *ipv4_address* is a string with dot-notation numeric IPv4 address, e.g. `"8.8.8.8"`, and *port* is and integer port number in the range 1-65535. Note the domain names are not accepted as *ipv4_address*, they should be resolved first using *usocket.getaddrinfo()*.
- IPv6: *(ipv6_address, port, flowinfo, scopeid)*, where *ipv6_address* is a string with colon-notation numeric IPv6 address, e.g. `"2001:db8::1"`, and *port* is an integer port number in the range 1-65535. *flowinfo* must be 0. *scopeid* is the interface scope identifier for link-local addresses. Note the domain names are not accepted as *ipv6_address*, they should be resolved first using *usocket.getaddrinfo()*.

Functions

usocket.**socket**(*af=AF_INET*, *type=SOCK_STREAM*, *proto=IPPROTO_TCP*)
Create a new socket using the given address family, socket type and protocol number.

usocket.**getaddrinfo**(*host*, *port*)
Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. The list of 5-tuples has following structure:

```
(family, type, proto, canonname, sockaddr)
```

The following example shows how to connect to a given url:

```
s = socket.socket()
s.connect(socket.getaddrinfo('www.micropython.org', 80)[0][-1])
```

Difference to CPython

CPython raises a `socket.gaierror` exception (*OSError* subclass) in case of error in this function. MicroPython doesn't have `socket.gaierror` and raises OSError directly. Note that error numbers of *getaddrinfo()* form a separate namespace and may not match error numbers from *uerrno* module. To distinguish *getaddrinfo()* errors, they are represented by negative numbers, whereas standard system errors are positive numbers (error numbers are accessible using `e.args[0]` property from an exception object). The use of negative values is a provisional detail which may change in the future.

Constants

usocket.**AF_INET**
usocket.**AF_INET6**
Address family types. Availability depends on a particular board.

usocket.**SOCK_STREAM**
usocket.**SOCK_DGRAM**
Socket types.

usocket.**IPPROTO_UDP**
usocket.**IPPROTO_TCP**
IP protocol numbers.

usocket.SOL_*
Socket option levels (an argument to *setsockopt()*). The exact inventory depends on a MicroPython port.

usocket.SO_*
Socket options (an argument to *setsockopt()*). The exact inventory depends on a MicroPython port.

Constants specific to WiPy:

usocket.**IPPROTO_SEC**
Special protocol value to create SSL-compatible socket.

class socket

Methods

socket.**close**()
Mark the socket closed and release all resources. Once that happens, all future operations on the socket object will fail. The remote end will receive EOF indication if supported by protocol.

Sockets are automatically closed when they are garbage-collected, but it is recommended to *close()* them explicitly as soon you finished working with them.

socket.**bind**(*address*)
Bind the socket to *address*. The socket must not already be bound.

socket.**listen**([*backlog*])
Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it's lower, it will be set to 0); and specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

socket.**accept**()
Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

socket.**connect**(*address*)
Connect to a remote socket at *address*.

socket.**send**(*bytes*)
Send data to the socket. The socket must be connected to a remote socket. Returns number of bytes sent, which may be smaller than the length of data ("short write").

socket.**sendall**(*bytes*)
Send all data to the socket. The socket must be connected to a remote socket. Unlike *send()*, this method will try to send all of data, by sending data chunk by chunk consecutively.

The behavior of this method on non-blocking sockets is undefined. Due to this, on MicroPython, it's recommended to use *write()* method instead, which has the same "no short writes" policy for blocking sockets, and will return number of bytes sent on non-blocking sockets.

socket.**recv**(*bufsize*)
Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by bufsize.

socket.**sendto**(*bytes*, *address*)
Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*.

socket.**recvfrom**(*bufsize*)
Receive data from the socket. The return value is a pair *(bytes, address)* where *bytes* is a bytes object representing the data received and *address* is the address of the socket sending the data.

socket.**setsockopt**(*level*, *optname*, *value*)
Set the value of the given socket option. The needed symbolic constants are defined in the socket module (SO_* etc.). The *value* can be an integer or a bytes-like object representing a buffer.

socket.**settimeout**(*value*)

Set a timeout on blocking socket operations. The value argument can be a nonnegative floating point number expressing seconds, or None. If a non-zero value is given, subsequent socket operations will raise an *OSError* exception if the timeout period value has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If None is given, the socket is put in blocking mode.

Difference to CPython

CPython raises a `socket.timeout` exception in case of timeout, which is an *OSError* subclass. MicroPython raises an OSError directly instead. If you use `except OSError:` to catch the exception, your code will work both in MicroPython and CPython.

---

socket.**setblocking**(*flag*)

Set blocking or non-blocking mode of the socket: if flag is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain *settimeout()* calls:

- `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
- `sock.setblocking(False)` is equivalent to `sock.settimeout(0)`

socket.**makefile**(*mode='rb'*, *buffering=0*)

Return a file object associated with the socket. The exact returned type depends on the arguments given to makefile(). The support is limited to binary modes only ('rb', 'wb', and 'rwb'). CPython's arguments: *encoding*, *errors* and *newline* are not supported.

Difference to CPython

As MicroPython doesn't support buffered streams, values of *buffering* parameter is ignored and treated as if it was 0 (unbuffered).

Difference to CPython

Closing the file object returned by makefile() WILL close the original socket as well.

---

socket.**read**([*size*])

Read up to size bytes from the socket. Return a bytes object. If *size* is not given, it reads all data available from the socket until EOF; as such the method will not return until the socket is closed. This function tries to read as much data as requested (no "short reads"). This may be not possible with non-blocking socket though, and then less data will be returned.

socket.**readinto**(*buf*[, *nbytes*])

Read bytes into the *buf*. If *nbytes* is specified then read at most that many bytes. Otherwise, read at most *len(buf)* bytes. Just as *read()*, this method follows "no short reads" policy.

Return value: number of bytes read and stored into *buf*.

socket.**readline**()

Read a line, ending in a newline character.

Return value: the line read.

socket.**write**(*buf*)

Write the buffer of bytes to the socket. This function will try to write all data to a socket (no "short writes").

This may be not possible with a non-blocking socket though, and returned value will be less than the length of *buf*.

Return value: number of bytes written.

exception socket.**error**
MicroPython does NOT have this exception.

Difference to CPython

CPython used to have a `socket.error` exception which is now deprecated, and is an alias of *OSError*. In MicroPython, use *OSError* directly.

---

ussl – SSL/TLS module

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:ssl`.

This module provides access to Transport Layer Security (previously and widely known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side.

Functions

ssl.**wrap_socket**(*sock*, *server_side=False*, *keyfile=None*, *certfile=None*, *cert_reqs=CERT_NONE*, *ca_certs=None*)
Takes a stream *sock* (usually usocket.socket instance of `SOCK_STREAM` type), and returns an instance of ssl.SSLSocket, which wraps the underlying stream in an SSL context. Returned object has the usual stream interface methods like `read()`, `write()`, etc. In MicroPython, the returned object does not expose socket interface and methods like `recv()`, `send()`. In particular, a server-side SSL socket should be created from a normal socket returned from *accept()* on a non-SSL listening server socket.

Depending on the underlying module implementation for a particular board, some or all keyword arguments above may be not supported.

Warning: Some implementations of `ssl` module do NOT validate server certificates, which makes an SSL connection established prone to man-in-the-middle attacks.

Exceptions

ssl.**SSLError**
This exception does NOT exist. Instead its base class, OSError, is used.

Constants

ssl.**CERT_NONE**
ssl.**CERT_OPTIONAL**

`ssl.`**`CERT_REQUIRED`**
Supported values for *cert_reqs* parameter.

`ustruct` – pack and unpack primitive data types

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:struct`.

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

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

Functions

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

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

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

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

`ustruct.`**`unpack_from`**(*fmt*, *data*, *offset=0*)
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.

`uzlib` – zlib decompression

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

|see_cpython_module| `cpython:zlib`.

This module allows to decompress binary data compressed with DEFLATE algorithm (commonly used in zlib library and gzip archiver). Compression is not yet implemented.

Functions

`uzlib.`**`decompress`**(*data*, *wbits=0*, *bufsize=0*)
Return decompressed *data* as bytes. *wbits* is DEFLATE dictionary window size used during compression (8-15, the dictionary size is power of 2 of that value). Additionally, if value is positive, *data* is assumed to be

zlib stream (with zlib header). Otherwise, if it's negative, it's assumed to be raw DEFLATE stream. *bufsize* parameter is for compatibility with CPython and is ignored.

class uzlib.**DecompIO**(*stream*, *wbits=0*)

Create a stream wrapper which allows transparent decompression of compressed data in another *stream*. This allows to process compressed streams with data larger than available heap size. In addition to values described in *decompress()*, *wbits* may take values 24..31 (16 + 8..15), meaning that input stream has gzip header.

Difference to CPython

This class is MicroPython extension. It's included on provisional basis and may be changed considerably or removed in later versions.

MicroPython-specific libraries

Functionality specific to the MicroPython implementation is available in the following libraries.

btree – simple BTree database

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

The btree module implements a simple key-value database using external storage (disk files, or in general case, a random-access stream). Keys are stored sorted in the database, and besides efficient retrieval by a key value, a database also supports efficient ordered range scans (retrieval of values with the keys in a given range). On the application interface side, BTree database work as close a possible to a way standard *dict* type works, one notable difference is that both keys and values must be *bytes* objects (so, if you want to store objects of other types, you need to serialize them to *bytes* first).

The module is based on the well-known BerkelyDB library, version 1.xx.

Example:

```
import btree

# First, we need to open a stream which holds a database
# This is usually a file, but can be in-memory database
# using uio.BytesIO, a raw flash partition, etc.
# Oftentimes, you want to create a database file if it doesn't
# exist and open if it exists. Idiom below takes care of this.
# DO NOT open database with "a+b" access mode.
try:
    f = open("mydb", "r+b")
except OSError:
    f = open("mydb", "w+b")

# Now open a database itself
db = btree.open(f)

# The keys you add will be sorted internally in the database
db[b"3"] = b"three"
db[b"1"] = b"one"
db[b"2"] = b"two"
```

```
# Assume that any changes are cached in memory unless
# explicitly flushed (or database closed). Flush database
# at the end of each "transaction".
db.flush()

# Prints b'two'
print(db[b"2"])

# Iterate over sorted keys in the database, starting from b"2"
# until the end of the database, returning only values.
# Mind that arguments passed to values() method are *key* values.
# Prints:
#   b'two'
#   b'three'
for word in db.values(b"2"):
    print(word)

del db[b"2"]

# No longer true, prints False
print(b"2" in db)

# Prints:
#  b"1"
#  b"3"
for key in db:
    print(key)

db.close()

# Don't forget to close the underlying stream!
f.close()
```

Functions

btree.**open**(*stream*, *, *flags=0*, *cachesize=0*, *pagesize=0*, *minkeypage=0*)

Open a database from a random-access `stream` (like an open file). All other parameters are optional and keyword-only, and allow to tweak advanced parameters of the database operation (most users will not need them):

- *flags* - Currently unused.
- *cachesize* - Suggested maximum memory cache size in bytes. For a board with enough memory using larger values may improve performance. The value is only a recommendation, the module may use more memory if values set too low.
- *pagesize* - Page size used for the nodes in BTree. Acceptable range is 512-65536. If 0, underlying I/O block size will be used (the best compromise between memory usage and performance).
- *minkeypage* - Minimum number of keys to store per page. Default value of 0 equivalent to 2.

Returns a BTree object, which implements a dictionary protocol (set of methods), and some additional methods described below.

Methods

btree.**close**()
Close the database. It's mandatory to close the database at the end of processing, as some unwritten data may be still in the cache. Note that this does not close underlying stream with which the database was opened, it should be closed separately (which is also mandatory to make sure that data flushed from buffer to the underlying storage).

btree.**flush**()
Flush any data in cache to the underlying stream.

btree.**__getitem__**(*key*)
btree.**get**(*key*, *default=None*)
btree.**__setitem__**(*key*, *val*)
btree.**__delitem__**(*key*)
btree.**__contains__**(*key*)
Standard dictionary methods.

btree.**__iter__**()
A BTree object can be iterated over directly (similar to a dictionary) to get access to all keys in order.

btree.**keys**([*start_key*[, *end_key*[, *flags*]]])
btree.**values**([*start_key*[, *end_key*[, *flags*]]])
btree.**items**([*start_key*[, *end_key*[, *flags*]]])
These methods are similar to standard dictionary methods, but also can take optional parameters to iterate over a key sub-range, instead of the entire database. Note that for all 3 methods, *start_key* and *end_key* arguments represent key values. For example, *values()* method will iterate over values corresponding to they key range given. None values for *start_key* means "from the first key", no *end_key* or its value of None means "until the end of database". By default, range is inclusive of *start_key* and exclusive of *end_key*, you can include *end_key* in iteration by passing *flags* of *btree.INCL*. You can iterate in descending key direction by passing *flags* of *btree.DESC*. The flags values can be ORed together.

Constants

btree.**INCL**
A flag for *keys()*, *values()*, *items()* methods to specify that scanning should be inclusive of the end key.

btree.**DESC**
A flag for *keys()*, *values()*, *items()* methods to specify that scanning should be in descending direction of keys.

framebuf — Frame buffer manipulation

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

This module provides a general frame buffer which can be used to create bitmap images, which can then be sent to a display.

class FrameBuffer

The FrameBuffer class provides a pixel buffer which can be drawn upon with pixels, lines, rectangles, text and even other FrameBuffer's. It is useful when generating output for displays.

For example:

```
import framebuf

# FrameBuffer needs 2 bytes for every RGB565 pixel
fbuf = FrameBuffer(bytearray(10 * 100 * 2), 10, 100, framebuf.RGB565)

fbuf.fill(0)
fbuf.text('MicroPython!', 0, 0, 0xffff)
fbuf.hline(0, 10, 96, 0xffff)
```

Constructors

class framebuf.**FrameBuffer**(*buffer*, *width*, *height*, *format*, *stride=width*)

Construct a FrameBuffer object. The parameters are:

- *buffer* is an object with a buffer protocol which must be large enough to contain every pixel defined by the width, height and format of the FrameBuffer.
- *width* is the width of the FrameBuffer in pixels
- *height* is the height of the FrameBuffer in pixels
- *format* specifies the type of pixel used in the FrameBuffer; valid values are `framebuf.MVLSB`, `framebuf.RGB565` and `framebuf.GS4_HMSB`. MVLSB is monochrome 1-bit color, RGB565 is RGB 16-bit color, and GS4_HMSB is grayscale 4-bit color. Where a color value c is passed to a method, c is a small integer with an encoding that is dependent on the format of the FrameBuffer.
- *stride* is the number of pixels between each horizontal line of pixels in the FrameBuffer. This defaults to *width* but may need adjustments when implementing a FrameBuffer within another larger FrameBuffer or screen. The *buffer* size must accommodate an increased step size.

One must specify valid *buffer*, *width*, *height*, *format* and optionally *stride*. Invalid *buffer* size or dimensions may lead to unexpected errors.

Drawing primitive shapes

The following methods draw shapes onto the FrameBuffer.

FrameBuffer.**fill**(*c*)

Fill the entire FrameBuffer with the specified color.

FrameBuffer.**pixel**(*x*, *y*[, *c*])

If *c* is not given, get the color value of the specified pixel. If *c* is given, set the specified pixel to the given color.

FrameBuffer.**hline**(*x*, *y*, *w*, *c*)

FrameBuffer.**vline**(*x*, *y*, *h*, *c*)

FrameBuffer.**line**(*x1*, *y1*, *x2*, *y2*, *c*)

Draw a line from a set of coordinates using the given color and a thickness of 1 pixel. The *line* method draws the line up to a second set of coordinates whereas the *hline* and *vline* methods draw horizontal and vertical lines respectively up to a given length.

FrameBuffer.**rect**(*x*, *y*, *w*, *h*, *c*)

FrameBuffer.**fill_rect**(*x*, *y*, *w*, *h*, *c*)

Draw a rectangle at the given location, size and color. The *rect* method draws only a 1 pixel outline whereas the *fill_rect* method draws both the outline and interior.

Drawing text

FrameBuffer.**text**(*s*, *x*, *y*[, *c*])

Write text to the FrameBuffer using the the coordinates as the upper-left corner of the text. The color of the text can be defined by the optional argument but is otherwise a default value of 1. All characters have dimensions of 8x8 pixels and there is currently no way to change the font.

Other methods

FrameBuffer.**scroll**(*xstep*, *ystep*)

Shift the contents of the FrameBuffer by the given vector. This may leave a footprint of the previous colors in the FrameBuffer.

FrameBuffer.**blit**(*fbuf*, *x*, *y*[, *key*])

Draw another FrameBuffer on top of the current one at the given coordinates. If *key* is specified then it should be a color integer and the corresponding color will be considered transparent: all pixels with that color value will not be drawn.

This method works between FrameBuffer's utilising different formats, but the resulting colors may be unexpected due to the mismatch in color formats.

Constants

framebuf.**MONO_VLSB**

Monochrome (1-bit) color format This defines a mapping where the bits in a byte are vertically mapped with bit 0 being nearest the top of the screen. Consequently each byte occupies 8 vertical pixels. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered at locations starting at the leftmost edge, 8 pixels lower.

framebuf.**MONO_HLSB**

Monochrome (1-bit) color format This defines a mapping where the bits in a byte are horizontally mapped. Each byte occupies 8 horizontal pixels with bit 0 being the leftmost. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered on the next row, one pixel lower.

framebuf.**MONO_HMSB**

Monochrome (1-bit) color format This defines a mapping where the bits in a byte are horizontally mapped. Each byte occupies 8 horizontal pixels with bit 7 being the leftmost. Subsequent bytes appear at successive horizontal locations until the rightmost edge is reached. Further bytes are rendered on the next row, one pixel lower.

framebuf.**RGB565**

Red Green Blue (16-bit, 5+6+5) color format

framebuf.**GS4_HMSB**

Grayscale (4-bit) color format

micropython – access and control MicroPython internals

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

Functions

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

Used to declare that the expression is a constant so that the compile can optimise it. The use of this function should be as follows:

```
from micropython import const

CONST_X = const(123)
CONST_Y = const(2 * CONST_X + 1)
```

Constants declared this way are still accessible as global variables from outside the module they are declared in. On the other hand, if a constant begins with an underscore then it is hidden, it is not available as a global variable, and does not take up any memory during execution.

This *const* function is recognised directly by the MicroPython parser and is provided as part of the *micropython* module mainly so that scripts can be written which run under both CPython and MicroPython, by following the above pattern.

micropython.**opt_level**([*level*])

If *level* is given then this function sets the optimisation level for subsequent compilation of scripts, and returns `None`. Otherwise it returns the current optimisation level.

micropython.**alloc_emergency_exception_buf**(*size*)

Allocate *size* bytes of RAM for the emergency exception buffer (a good size is around 100 bytes). The buffer is used to create exceptions in cases when normal RAM allocation would fail (eg within an interrupt handler) and therefore give useful traceback information in these situations.

A good way to use this function is to put it at the start of your main script (eg `boot.py` or `main.py`) and then the emergency exception buffer will be active for all the code following it.

micropython.**mem_info**([*verbose*])

Print information about currently used memory. If the *verbose'* argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the amount of stack and heap used. In verbose mode it prints out the entire heap indicating which blocks are used and which are free.

micropython.**qstr_info**([*verbose*])

Print information about currently interned strings. If the *verbose* argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the number of interned strings and the amount of RAM they use. In verbose mode it prints out the names of all RAM-interned strings.

micropython.**stack_use**()

Return an integer representing the current amount of stack that is being used. The absolute value of this is not particularly useful, rather it should be used to compute differences in stack usage at different points.

micropython.**heap_lock**()

micropython.**heap_unlock**()
Lock or unlock the heap. When locked no memory allocation can occur and a *MemoryError* will be raised if any heap allocation is attempted.

These functions can be nested, ie *heap_lock()* can be called multiple times in a row and the lock-depth will increase, and then *heap_unlock()* must be called the same number of times to make the heap available again.

micropython.**kbd_intr**(*chr*)
Set the character that will raise a *KeyboardInterrupt* exception. By default this is set to 3 during script execution, corresponding to Ctrl-C. Passing -1 to this function will disable capture of Ctrl-C, and passing 3 will restore it.

This function can be used to prevent the capturing of Ctrl-C on the incoming stream of characters that is usually used for the REPL, in case that stream is used for other purposes.

micropython.**schedule**(*func*, *arg*)
Schedule the function *func* to be executed "very soon". The function is passed the value *arg* as its single argument. "Very soon" means that the MicroPython runtime will do its best to execute the function at the earliest possible time, given that it is also trying to be efficient, and that the following conditions hold:

- A scheduled function will never preempt another scheduled function.
- Scheduled functions are always executed "between opcodes" which means that all fundamental Python operations (such as appending to a list) are guaranteed to be atomic.
- A given port may define "critical regions" within which scheduled functions will never be executed. Functions may be scheduled within a critical region but they will not be executed until that region is exited. An example of a critical region is a preempting interrupt handler (an IRQ).

A use for this function is to schedule a callback from a preempting IRQ. Such an IRQ puts restrictions on the code that runs in the IRQ (for example the heap may be locked) and scheduling a function to call later will lift those restrictions.

There is a finite stack to hold the scheduled functions and *schedule* will raise a *RuntimeError* if the stack is full.

network — network configuration

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

This module provides network drivers and routing configuration. To use this module, a MicroPython variant/build with network capabilities must be installed. Network drivers for specific hardware are available within this module and are used to configure hardware network interface(s). Network services provided by configured interfaces are then available for use via the `socket` module.

For example:

```
# connect/ show IP config a specific network interface
# see below for examples of specific drivers
import network
import utime
nic = network.Driver(...)
if not nic.isconnected():
    nic.connect()
    print("Waiting for connection...")
    while not nic.isconnected():
        utime.sleep(1)
print(nic.ifconfig())
```

```
# now use usocket as usual
import usocket as socket
addr = socket.getaddrinfo('micropython.org', 80)[0][-1]
s = socket.socket()
s.connect(addr)
s.send(b'GET / HTTP/1.1\r\nHost: micropython.org\r\n\r\n')
data = s.recv(1000)
s.close()
```

Common network adapter interface

This section describes an (implied) abstract base class for all network interface classes implemented by different ports of MicroPython for different hardware. This means that MicroPython does not actually provide *AbstractNIC* class, but any actual NIC class, as described in the following sections, implements methods as described here.

class network.**AbstractNIC**(*id=None*, ...)

Instantiate a network interface object. Parameters are network interface dependent. If there are more than one interface of the same type, the first parameter should be *id*.

network.**active**([*is_active*])
: Activate ("up") or deactivate ("down") the network interface, if a boolean argument is passed. Otherwise, query current state if no argument is provided. Most other methods require an active interface (behavior of calling them on inactive interface is undefined).

network.**connect**([*service_id*, *key=None*, *, ...])
: Connect the interface to a network. This method is optional, and available only for interfaces which are not "always connected". If no parameters are given, connect to the default (or the only) service. If a single parameter is given, it is the primary identifier of a service to connect to. It may be accompanied by a key (password) required to access said service. There can be further arbitrary keyword-only parameters, depending on the networking medium type and/or particular device. Parameters can be used to: a) specify alternative service identifer types; b) provide additional connection parameters. For various medium types, there are different sets of predefined/recommended parameters, among them:

  - WiFi: *bssid* keyword to connect by BSSID (MAC address) instead of access point name

network.**disconnect**()
: Disconnect from network.

network.**isconnected**()
: Returns `True` if connected to network, otherwise returns `False`.

network.**scan**(*, ...)
: Scan for the available network services/connections. Returns a list of tuples with discovered service parameters. For various network media, there are different variants of predefined/ recommended tuple formats, among them:

  - WiFi: (ssid, bssid, channel, RSSI, authmode, hidden). There may be further fields, specific to a particular device.

  The function may accept additional keyword arguments to filter scan results (e.g. scan for a particular service, on a particular channel, for services of a particular set, etc.), and to affect scan duration and other parameters. Where possible, parameter names should match those in connect().

network.**status**()
: Return detailed status of the interface, values are dependent on the network medium/technology.

network.**ifconfig**([*(ip, subnet, gateway, dns)*])
Get/set IP-level network interface parameters: IP address, subnet mask, gateway and DNS server. When called with no arguments, this method returns a 4-tuple with the above information. To set the above values, pass a 4-tuple with the required information. For example:

```
nic.ifconfig(('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
```

network.**config**(*'param'*)
network.**config**(*param=value, ...*)
Get or set general network interface parameters. These methods allow to work with additional parameters beyond standard IP configuration (as dealt with by *ifconfig()*). These include network-specific and hardware-specific parameters and status values. For setting parameters, the keyword argument syntax should be used, and multiple parameters can be set at once. For querying, a parameter name should be quoted as a string, and only one parameter can be queried at a time:

```
# Set WiFi access point name (formally known as ESSID) and WiFi channel
ap.config(essid='My AP', channel=11)
# Query params one by one
print(ap.config('essid'))
print(ap.config('channel'))
# Extended status information also available this way
print(sta.config('rssi'))
```

Functions

network.**phy_mode**([*mode*])
Get or set the PHY mode.

If the *mode* parameter is provided, sets the mode to its value. If the function is called without parameters, returns the current mode.

The possible modes are defined as constants:

- MODE_11B – IEEE 802.11b,
- MODE_11G – IEEE 802.11g,
- MODE_11N – IEEE 802.11n.

class WLAN

This class provides a driver for WiFi network processor in the ESP8266. Example usage:

```
import network
# enable station interface and connect to WiFi access point
nic = network.WLAN(network.STA_IF)
nic.active(True)
nic.connect('your-ssid', 'your-password')
# now use sockets as usual
```

Constructors

class network.**WLAN**(*interface_id*)

Create a WLAN network interface object. Supported interfaces are `network.STA_IF` (station aka client, connects to upstream WiFi access points) and `network.AP_IF` (access point, allows other WiFi clients to connect). Availability of the methods below depends on interface type. For example, only STA interface may *connect()* to an access point.

Methods

`wlan.`**`active`**`(`[*is_active*]`)`
Activate ("up") or deactivate ("down") network interface, if boolean argument is passed. Otherwise, query current state if no argument is provided. Most other methods require active interface.

`wlan.`**`connect`**`(`*ssid*, *password*`)`
Connect to the specified wireless network, using the specified password.

`wlan.`**`disconnect`**`()`
Disconnect from the currently connected wireless network.

`wlan.`**`scan`**`()`
Scan for the available wireless networks.

Scanning is only possible on STA interface. Returns list of tuples with the information about WiFi access points:

(ssid, bssid, channel, RSSI, authmode, hidden)

bssid is hardware address of an access point, in binary form, returned as bytes object. You can use *ubinascii.hexlify()* to convert it to ASCII form.

There are five values for authmode:

- 0 – open
- 1 – WEP
- 2 – WPA-PSK
- 3 – WPA2-PSK
- 4 – WPA/WPA2-PSK

and two for hidden:

- 0 – visible
- 1 – hidden

`wlan.`**`status`**`()`
Return the current status of the wireless connection.

The possible statuses are defined as constants:

- `STAT_IDLE` – no connection and no activity,
- `STAT_CONNECTING` – connecting in progress,
- `STAT_WRONG_PASSWORD` – failed due to incorrect password,
- `STAT_NO_AP_FOUND` – failed because no access point replied,
- `STAT_CONNECT_FAIL` – failed due to other problems,
- `STAT_GOT_IP` – connection successful.

`wlan.`**`isconnected`**`()`
In case of STA mode, returns `True` if connected to a WiFi access point and has a valid IP address. In AP mode returns `True` when a station is connected. Returns `False` otherwise.

wlan.**ifconfig**([*(ip, subnet, gateway, dns)*])

Get/set IP-level network interface parameters: IP address, subnet mask, gateway and DNS server. When called with no arguments, this method returns a 4-tuple with the above information. To set the above values, pass a 4-tuple with the required information. For example:

```
nic.ifconfig(('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
```

wlan.**config**(*'param'*)

wlan.**config**(*param=value, ...*)

Get or set general network interface parameters. These methods allow to work with additional parameters beyond standard IP configuration (as dealt with by *wlan.ifconfig()*). These include network-specific and hardware-specific parameters. For setting parameters, keyword argument syntax should be used, multiple parameters can be set at once. For querying, parameters name should be quoted as a string, and only one parameter can be queries at time:

```
# Set WiFi access point name (formally known as ESSID) and WiFi channel
ap.config(essid='My AP', channel=11)
# Query params one by one
print(ap.config('essid'))
print(ap.config('channel'))
```

Following are commonly supported parameters (availability of a specific parameter depends on network technology type, driver, and MicroPython port).

Parameter	Description
mac	MAC address (bytes)
essid	WiFi access point name (string)
channel	WiFi channel (integer)
hidden	Whether ESSID is hidden (boolean)
authmode	Authentication mode supported (enumeration, see module constants)
password	Access password (string)

uctypes – access binary data in a structured way

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

This module implements "foreign data interface" for MicroPython. The idea behind it is similar to CPython's ctypes modules, but the actual API is different, streamlined and optimized for small size. The basic idea of the module is to define data structure layout with about the same power as the C language allows, and the access it using familiar dot-syntax to reference sub-fields.

See also:

Module struct Standard Python way to access binary data structures (doesn't scale well to large and complex structures).

Defining structure layout

Structure layout is defined by a "descriptor" - a Python dictionary which encodes field names as keys and other properties required to access them as associated values. Currently, uctypes requires explicit specification of offsets for

each field. Offset are given in bytes from a structure start.

Following are encoding examples for various field types:

- Scalar types:

```
"field_name": uctypes.UINT32 | 0
```

  in other words, value is scalar type identifier ORed with field offset (in bytes) from the start of the structure.

- Recursive structures:

```
"sub": (2, {
    "b0": uctypes.UINT8 | 0,
    "b1": uctypes.UINT8 | 1,
})
```

  i.e. value is a 2-tuple, first element of which is offset, and second is a structure descriptor dictionary (note: offsets in recursive descriptors are relative to a structure it defines).

- Arrays of primitive types:

```
"arr": (uctypes.ARRAY | 0, uctypes.UINT8 | 2),
```

  i.e. value is a 2-tuple, first element of which is ARRAY flag ORed with offset, and second is scalar element type ORed number of elements in array.

- Arrays of aggregate types:

```
"arr2": (uctypes.ARRAY | 0, 2, {"b": uctypes.UINT8 | 0}),
```

  i.e. value is a 3-tuple, first element of which is ARRAY flag ORed with offset, second is a number of elements in array, and third is descriptor of element type.

- Pointer to a primitive type:

```
"ptr": (uctypes.PTR | 0, uctypes.UINT8),
```

  i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, and second is scalar element type.

- Pointer to an aggregate type:

```
"ptr2": (uctypes.PTR | 0, {"b": uctypes.UINT8 | 0}),
```

  i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, second is descriptor of type pointed to.

- Bitfields:

```
"bitf0": uctypes.BFUINT16 | 0 | 0 << uctypes.BF_POS | 8 << uctypes.BF_LEN,
```

  i.e. value is type of scalar value containing given bitfield (typenames are similar to scalar types, but prefixes with "BF"), ORed with offset for scalar value containing the bitfield, and further ORed with values for bit offset and bit length of the bitfield within scalar value, shifted by BF_POS and BF_LEN positions, respectively. Bitfield position is counted from the least significant bit, and is the number of right-most bit of a field (in other words, it's a number of bits a scalar needs to be shifted right to extra the bitfield).

  In the example above, first UINT16 value will be extracted at offset 0 (this detail may be important when accessing hardware registers, where particular access size and alignment are required), and then bitfield whose rightmost bit is least-significant bit of this UINT16, and length is 8 bits, will be extracted - effectively, this will access least-significant byte of UINT16.

Note that bitfield operations are independent of target byte endianness, in particular, example above will access least-significant byte of UINT16 in both little- and big-endian structures. But it depends on the least significant bit being numbered 0. Some targets may use different numbering in their native ABI, but `uctypes` always uses normalized numbering described above.

Module contents

class `uctypes.`**struct**(*addr*, *descriptor*, *layout_type=NATIVE*)
Instantiate a "foreign data structure" object based on structure address in memory, descriptor (encoded as a dictionary), and layout type (see below).

`uctypes.`**LITTLE_ENDIAN**
Layout type for a little-endian packed structure. (Packed means that every field occupies exactly as many bytes as defined in the descriptor, i.e. the alignment is 1).

`uctypes.`**BIG_ENDIAN**
Layout type for a big-endian packed structure.

`uctypes.`**NATIVE**
Layout type for a native structure - with data endianness and alignment conforming to the ABI of the system on which MicroPython runs.

`uctypes.`**sizeof**(*struct*)
Return size of data structure in bytes. Argument can be either structure class or specific instantiated structure object (or its aggregate field).

`uctypes.`**addressof**(*obj*)
Return address of an object. Argument should be bytes, bytearray or other object supporting buffer protocol (and address of this buffer is what actually returned).

`uctypes.`**bytes_at**(*addr*, *size*)
Capture memory at the given address and size as bytes object. As bytes object is immutable, memory is actually duplicated and copied into bytes object, so if memory contents change later, created object retains original value.

`uctypes.`**bytearray_at**(*addr*, *size*)
Capture memory at the given address and size as bytearray object. Unlike bytes_at() function above, memory is captured by reference, so it can be both written too, and you will access current value at the given memory address.

Structure descriptors and instantiating structure objects

Given a structure descriptor dictionary and its layout type, you can instantiate a specific structure instance at a given memory address using *`uctypes.struct()`* constructor. Memory address usually comes from following sources:

- Predefined address, when accessing hardware registers on a baremetal system. Lookup these addresses in datasheet for a particular MCU/SoC.
- As a return value from a call to some FFI (Foreign Function Interface) function.
- From uctypes.addressof(), when you want to pass arguments to an FFI function, or alternatively, to access some data for I/O (for example, data read from a file or network socket).

Structure objects

Structure objects allow accessing individual fields using standard dot notation: `my_struct.substruct1.field1`. If a field is of scalar type, getting it will produce a primitive value (Python integer or float) corresponding

to the value contained in a field. A scalar field can also be assigned to.

If a field is an array, its individual elements can be accessed with the standard subscript operator `[]` - both read and assigned to.

If a field is a pointer, it can be dereferenced using `[0]` syntax (corresponding to C `*` operator, though `[0]` works in C too). Subscripting a pointer with other integer values but 0 are supported too, with the same semantics as in C.

Summing up, accessing structure fields generally follows C syntax, except for pointer dereference, when you need to use `[0]` operator instead of `*`.

Limitations

Accessing non-scalar fields leads to allocation of intermediate objects to represent them. This means that special care should be taken to layout a structure which needs to be accessed when memory allocation is disabled (e.g. from an interrupt). The recommendations are:

- Avoid nested structures. For example, instead of `mcu_registers.peripheral_a.register1`, define separate layout descriptors for each peripheral, to be accessed as `peripheral_a.register1`.
- Avoid other non-scalar data, like array. For example, instead of `peripheral_a.register[0]` use `peripheral_a.register0`.

Note that these recommendations will lead to decreased readability and conciseness of layouts, so they should be used only if the need to access structure fields without allocation is anticipated (it's even possible to define 2 parallel layouts - one for normal usage, and a restricted one to use when memory allocation is prohibited).

Libraries specific to the ESP8266

The following libraries are specific to the ESP8266.

esp — functions related to the ESP8266

Warning: This module is inherited from MicroPython and may not work in CircuitPython as documented or at all! If they do work, they may change at any time. It is unsupported.

The `esp` module contains specific functions related to the ESP8266 module.

Functions

`esp.`**`sleep_type`**`(`[*sleep_type*]`)`
Get or set the sleep type.

If the *sleep_type* parameter is provided, sets the sleep type to its value. If the function is called without parameters, returns the current sleep type.

The possible sleep types are defined as constants:

- `SLEEP_NONE` – all functions enabled,
- `SLEEP_MODEM` – modem sleep, shuts down the WiFi Modem circuit.
- `SLEEP_LIGHT` – light sleep, shuts down the WiFi Modem circuit and suspends the processor periodically.

The system enters the set sleep mode automatically when possible.

esp.**deepsleep**(*time=0*)
Enter deep sleep.

The whole module powers down, except for the RTC clock circuit, which can be used to restart the module after the specified time if the pin 16 is connected to the reset pin. Otherwise the module will sleep until manually reset.

esp.**flash_id**()
Read the device ID of the flash memory.

esp.**flash_read**(*byte_offset*, *length_or_buffer*)

esp.**flash_write**(*byte_offset*, *bytes*)

esp.**flash_erase**(*sector_no*)

esp.**set_native_code_location**(*start*, *length*)
Set the location that native code will be placed for execution after it is compiled. Native code is emitted when the `@micropython.native`, `@micropython.viper` and `@micropython.asm_xtensa` decorators are applied to a function. The ESP8266 must execute code from either iRAM or the lower 1MByte of flash (which is memory mapped), and this function controls the location.

If *start* and *length* are both `None` then the native code location is set to the unused portion of memory at the end of the iRAM1 region. The size of this unused portion depends on the firmware and is typically quite small (around 500 bytes), and is enough to store a few very small functions. The advantage of using this iRAM1 region is that it does not get worn out by writing to it.

If neither *start* nor *length* are `None` then they should be integers. *start* should specify the byte offset from the beginning of the flash at which native code should be stored. *length* specifies how many bytes of flash from *start* can be used to store native code. *start* and *length* should be multiples of the sector size (being 4096 bytes). The flash will be automatically erased before writing to it so be sure to use a region of flash that is not otherwise used, for example by the firmware or the filesystem.

When using the flash to store native code *start+length* must be less than or equal to 1MByte. Note that the flash can be worn out if repeated erasures (and writes) are made so use this feature sparingly. In particular, native code needs to be recompiled and rewritten to flash on each boot (including wake from deepsleep).

In both cases above, using iRAM1 or flash, if there is no more room left in the specified region then the use of a native decorator on a function will lead to *MemoryError* exception being raised during compilation of that function.

1.8.7 Adafruit's CircuitPython Documentation

The latest documentation can be found at: http://circuitpython.readthedocs.io/en/latest/

The documentation you see there is generated from the files in the whole tree: https://github.com/adafruit/circuitpython/tree/master

Building the documentation locally

If you're making changes to the documentation, you should build the documentation locally so that you can preview your changes.

Install Sphinx, recommonmark, and optionally (for the RTD-styling), sphinx_rtd_theme, preferably in a virtualenv:

```
pip install sphinx
pip install recommonmark
pip install sphinx_rtd_theme
```

In `circuitpython/`, build the docs:

```
sphinx-build -v -b html . _build/html
```

You'll find the index page at `_build/html/index.html`.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z