MicroPython Documentation

Release 0.0.0

Damien P. George, Paul Sokolovsky, and contributors

Aug 16, 2017

Contents

Welcome! This is the documentation for Adafruit CircuitPython. It is an open source derivative of MicroPython for use on educational development boards designed and sold by Adafruit. Adafruit CircuitPython features unified Python core APIs and a growing list of drivers that work with it.

The Adafruit Express line of boards are specifically designed for use with CircuitPython. They are the CircuitPlayground Express, Feather M0 Express, and Metro M0 Express. Other supported boards include the Arduino Zero, Adafruit Feather M0 Basic, Adafruit Feather HUZZAH and Adafruit Feather M0 Bluefruit LE.

Adafruit has many excellent tutorials available through the Adafruit Learning System. These docs are low-level API docs and may link out to separate getting started guides.

CHAPTER 1

Core Modules

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

Support Matrix

Port	`analogio`	`audiobusio`	`audioio`	`bitbangio`	`board`	`busio`	`digitalio`	`microcontroller`	`multiterminal`	`neopixel_write`	`os`	`pulseio`	`rainbowio`	`random`	`storage`	`supervisor`	`time`	`touchio`	`usb_hid`
SAMD21	**Yes**	No	No	**Yes**	**Yes**	**Yes**	**Yes**	No	**Yes**	**Yes**	No	**Yes**	**Yes**	**Yes**	**Yes**	Debug	**Yes**	**Yes**	**Yes**
SAMD21 Express	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	No	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	Debug	**Yes**	**Yes**	**Yes**
ESP8266	**Yes**	No	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	No	**Yes**	**Yes**	**Yes**	No	Debug	No		

Modules

`analogio` — Analog hardware support

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

Libraries

`AnalogIn` – read analog voltage

Usage:

```
import analogio
from board import *

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

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

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

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

deinit()

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

__enter__()

No-op used by Context Managers.

__exit__()

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

value

Read the value on the analog pin and return it. The returned value will be between 0 and 65535 inclusive (16-bit). Even if the underlying analog to digital converter (ADC) is lower resolution, the result will be scaled to be 16-bit.

Returns the data read

Return type *int*

reference_voltage

The maximum voltage measurable. Also known as the reference voltage.

Returns the reference voltage

Return type *float*

AnalogOut – output analog voltage

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

Example usage:

```
import analogio
from microcontroller import pin

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

class analogio.**AnalogOut**(*pin*)

Use the AnalogOut on the given pin.

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

deinit()

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

__enter__()

No-op used by Context Managers.

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

value
The value on the analog pin. The value must be between 0 and 65535 inclusive (16-bit). Even if the underlying digital to analog converter is lower resolution, the input must be scaled to be 16-bit.

Returns the last value written

Return type *int*

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

For example:

```
import analogio
from board import *

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

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

audiobusio — Support for audio input and output over digital bus

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

Libraries

PDMIn – Record an input PDM audio stream

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

class audiobusio.**PDMIn**(*clock_pin*, *data_pin*, ***, *frequency=8000*, *bit_depth=8*, *mono=True*, *oversample=64*)
Create a PDMIn object associated with the given pins. This allows you to record audio signals from the given pins. Individual ports may put further restrictions on the recording parameters.

Parameters

- **clock_pin** (Pin) – The pin to output the clock to
- **data_pin** (Pin) – The pin to read the data from
- **frequency** (int) – Target frequency of the resulting samples. Check *frequency* for real value.
- **bit_depth** (int) – Final number of bits per sample. Must be divisible by 8
- **mono** (bool) – True when capturing a single channel of audio, captures two channels otherwise

- **oversample** (int) – Number of single bit samples to decimate into a final sample. Must be divisible by 8

Simple record to buffer:

```
import audiobusio
import board

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

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

__enter__()
No-op used by Context Managers.

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

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

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

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

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

audioio — Support for audio input and output

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

Libraries

AudioOut – Output an analog audio signal

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

class audioio.**AudioOut**(*pin*, *sample_source*)
Create a AudioOut object associated with the given pin. This allows you to play audio signals out on the given pin. Sample_source must be a bytes-like object.

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

Parameters

- **pin** (Pin) – The pin to output to
- **sample_source** (*bytes-like*) – The source of the sample

Simple 8ksps 440 Hz sin wave:

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

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

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

Playing a wave file from flash:

```
import board
import audioio
import digitalio

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

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

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

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

__enter__()
No-op used by Context Managers.

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

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

stop()
Stops playback of this sample. If another sample is playing instead, it won't be stopped.

playing
True when the audio sample is being output.

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

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

`bitbangio` — Digital protocols implemented by the CPU

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

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

Libraries

`I2C` — Two wire serial protocol

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

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

Parameters

- **`scl`** (`Pin`) – The clock pin
- **`sda`** (`Pin`) – The data pin
- **`frequency`** (`int`) – The clock frequency of the bus

`deinit`()

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

`__enter__`()

No-op used in Context Managers.

`__exit__`()

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

`scan`()

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

`try_lock`()

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

`unlock`()

Releases the I2C lock.

`readfrom_into`(*address*, *buffer*, *, *start=0*, *end=len(buffer)*)

Read into `buffer` from the slave specified by `address`. The number of bytes read will be the length of `buffer`.

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

Parameters

- **`address`** (`int`) – 7-bit device address
- **`buffer`** (`bytearray`) – buffer to write into

- **start** (int) – Index to start writing at
- **end** (int) – Index to write up to but not include

writeto(*address*, *buffer*, ***, *start=0*, *end=len(buffer)*, *stop=True*)

Write the bytes from `buffer` to the slave specified by `address`. Transmits a stop bit if `stop` is set.

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

Parameters

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

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

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

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

class bitbangio.**OneWire**(*pin*)

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

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

Read a short series of pulses:

```
import bitbangio
import board

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

deinit()

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

__enter__()

No-op used by Context Managers.

__exit__()

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

reset()

Reset the OneWire bus

read_bit()

Read in a bit

Returns bit state read

Return type *bool*

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

SPI – a 3-4 wire serial protocol

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

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

Construct an SPI object on the given pins.

Parameters

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

deinit()
Turn off the SPI bus.

__enter__()
No-op used by Context Managers.

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

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

Parameters

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

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

Returns True when lock has been grabbed

Return type *bool*

unlock()
Releases the SPI lock.

write(*buf*)
Write the data contained in `buf`. Requires the SPI being locked.

readinto(*buf*)
Read into the buffer specified by `buf` while writing zeroes. Requires the SPI being locked.

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

For example:

```
import bitbangio
from board import *

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

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

board — Board specific pin names

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

busio — Hardware accelerated behavior

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

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

Libraries

I2C — Two wire serial protocol

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

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

See also:

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

See also:

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

Parameters

- **scl** (Pin) – The clock pin
- **sda** (Pin) – The data pin
- **frequency** (int) – The clock frequency in Hertz

deinit()

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

__enter__()
No-op used in Context Managers.

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

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

Returns List of device ids on the I2C bus

Return type *list*

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

Returns True when lock has been grabbed

Return type *bool*

unlock()
Releases the I2C lock.

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

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

Parameters

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

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

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

Parameters

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

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

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

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

class busio.**OneWire**(*pin*)

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

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

Read a short series of pulses:

```
import busio
import board

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

deinit()

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

__enter__()

No-op used by Context Managers.

__exit__()

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

reset()

Reset the OneWire bus and read presence

Returns False when at least one device is present

Return type *bool*

read_bit()

Read in a bit

Returns bit state read

Return type *bool*

write_bit(*value*)

Write out a bit based on value.

SPI – a 3-4 wire serial protocol

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

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

Construct an SPI object on the given pins.

See also:

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

See also:

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

Parameters

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

deinit()
Turn off the SPI bus.

__enter__()
No-op used by Context Managers.

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

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

Parameters

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

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

Returns True when lock has been grabbed

Return type *bool*

unlock()
Releases the SPI lock.

write(*buffer*, *, *start=0*, *end=len(buffer)*)
Write the data contained in buf. Requires the SPI being locked.

Parameters

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

readinto(*buffer*, *, *start=0*, *end=len(buffer)*, *write_value=0*)
Read into the buffer specified by buf while writing zeroes. Requires the SPI being locked.

Parameters

- **buffer** (bytearray) – buffer to write into
- **start** (int) – Index to start writing at
- **end** (int) – Index to write up to but not include
- **write_value** (int) – Value to write reading. (Usually ignored.)

UART – a bidirectional serial protocol

class busio.**UART**(*tx*, *rx*, ***, *baudrate=9600*, *bits=8*, *parity=None*, *stop=1*, *timeout=1000*, *receiver_buffer_size=64*)

A common bidirectional serial protocol that uses an an agreed upon speed rather than a shared clock line.

Parameters

- **tx** (Pin) – the pin to transmit with
- **rx** (Pin) – the pin to receive on
- **baudrate** (int) – the transmit and receive speed

deinit()

Deinitialises the UART and releases any hardware resources for reuse.

__enter__()

No-op used by Context Managers.

__exit__()

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

read(*nbytes=None*)

Read characters. If nbytes is specified then read at most that many bytes. Otherwise, read everything that has been buffered.

Returns Data read

Return type *bytes* or None

readinto(*buf*, *nbytes=None*)

Read bytes into the buf. If nbytes is specified then read at most that many bytes. Otherwise, read at most len(buf) bytes.

Returns number of bytes read and stored into buf

Return type *bytes* or None

readline()

Read a line, ending in a newline character.

Returns the line read

Return type *int* or None

write(*buf*)

Write the buffer of bytes to the bus.

Returns the number of bytes written

Return type *int* or None

class busio.UART.**Parity**

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

ODD

Total number of ones should be odd.

EVEN

Total number of ones should be even.

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

For example:

```
import busio
from board import *

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

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

digitalio — Basic digital pin support

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

Libraries

DigitalInOut – digital input and output

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

class digitalio.**DigitalInOut**(*pin*)

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

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

deinit()

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

__enter__()

No-op used by Context Managers.

__exit__()

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

switch_to_output(*value=False*, *drive_mode=digitalio.DriveMode.PUSH_PULL*)

Set the drive mode and value and then switch to writing out digital values.

Parameters

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

switch_to_input(*pull=None*)

Set the pull and then switch to read in digital values.

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

Example usage:

```
import digitalio
import board

switch = digitalio.DigitalInOut(board.SLIDE_SWITCH)
switch.switch_to_input(pull=digitalio.Pull.UP)
```

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

direction
The direction of the pin.

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

value
The digital logic level of the pin.

drive_mode
Get or set the pin drive mode.

pull
Get or set the pin pull.

Raises **AttributeError** – if the direction is ~'digitalio.Direction.OUTPUT'.

Direction – defines the direction of a digital pin

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

INPUT
Read digital data in

OUTPUT
Write digital data out

DriveMode – defines the drive mode of a digital pin

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

PUSH_PULL
Output both high and low digital values

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

Pull – defines the pull of a digital input pin

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

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

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

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

For example:

```
import digitalio
from board import *

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

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

Here is blinky:

```
import digitalio
from board import *
import time

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

`microcontroller` — Pin references and core functionality

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

Libraries

`Pin` — Pin reference

Identifies an IO pin on the microcontroller.

class `microcontroller.`**`Pin`**

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

`microcontroller.`**`delay_us`**(*delay*)

Dedicated delay method used for very short delays. DO NOT do long delays because it will stall any concurrent code.

`microcontroller.`**`disable_interrupts`**()

Disable all interrupts. Be very careful, this can stall everything.

`microcontroller.`**`enable_interrupts`**()

Enable the interrupts that were enabled at the last disable.

`microcontroller.pin` — Microcontroller pin names

References to pins as named by the microcontroller

multiterminal — Manage additional terminal sources

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

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

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

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

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

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

neopixel_write — Low-level neopixel implementation

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

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

```
import board
import neopixel_write
import digitalio

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

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

Parameters

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

os — functions that an OS normally provides

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`pulseio` — Support for pulse based protocols

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

Libraries

`PulseIn` – Read a series of pulse durations

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

class `pulseio.`**`PulseIn`**(*pin*, *maxlen=2*, ***, *idle_state=False*)

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

Parameters

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

Read a short series of pulses:

```
import pulseio
import board

pulses = pulseio.PulseIn(board.D7)

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

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

# Clear the rest
pulses.clear()

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

`deinit`()

Deinitialises the PulseIn and releases any hardware resources for reuse.

`__enter__`()

No-op used by Context Managers.

`__exit__`()

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

`pause`()

Pause pulse capture

`resume`(*trigger_duration=0*)

Resumes pulse capture after an optional trigger pulse.

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

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

clear()
Clears all captured pulses

popleft()
Removes and returns the oldest read pulse.

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

__len__()
Returns the current pulse length

This allows you to:

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

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

This allows you to:

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

PulseOut – Output a pulse train

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

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

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

Send a short series of pulses:

```
import array
import pulseio
import board

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

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

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

__enter__()
No-op used by Context Managers.

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

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

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

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

`PWMOut` – Output a Pulse Width Modulated signal

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

class `pulseio.`**PWMOut**(*pin*, *, *duty_cycle=0*, *frequency=500*, *variable_frequency=False*)
Create a PWM object associated with the given pin. This allows you to write PWM signals out on the given pin. Frequency is fixed after init unless `variable_frequency` is True.

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

Parameters

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

Simple LED fade:

```
import pulseio
import board

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

PWM at specific frequency (servos and motors):

```
import pulseio
import board

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

Variable frequency (usually tones):

```
import pulseio
import board
import time

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

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

__enter__()
No-op used by Context Managers.

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

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

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

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

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

For example:

```
import pulseio
import time
from board import *

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

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

random — psuedo-random numbers and choices

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

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

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

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

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

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

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

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

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

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

storage — storage management

The *storage* provides storage management functionality such as mounting and unmounting which is typically handled by the operating system hosting Python. CircuitPython does not have an OS, so this module provides this functionality directly. Its based on MicroPython's *uos* module but deliberately separates CPython compatible functionality (in *os*) from incompatible functionality (in *storage*).

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

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

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

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

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

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

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

time — time and timing related functions

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

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

Returns the current monotonic time

Return type *float*

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

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

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

Parameters

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

touchio — Touch related IO

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

Libraries

TouchIn – Read the state of a capacitive touch sensor

Usage:

```
import touchio
from board import *

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

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

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

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

__enter__()
No-op used by Context Managers.

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

value
Whether the touch pad is being touched or not.

Returns True when touched, False otherwise.

Return type *bool*

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

For example:

```
import touchio
from board import *

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

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

uheap — Heap size analysis

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

usb_hid — USB Human Interface Device

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

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

Libraries

Device – HID Device

Usage:

```
import usb_hid

mouse = usb_hid.devices[0]

mouse.send_report()
```

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

send_report(*buf*)
Send a HID report.

usage_page
The usage page of the device. Can be thought of a category.

Returns the device's usage page

Return type *int*

usage
The functionality of the device. For example Keyboard is 0x06 within the generic desktop usage page 0x01. Mouse is 0x02 within the same usage page.

Returns the usage within the usage page

Return type *int*

ustack — Stack information and analysis

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

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

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

CHAPTER 2

Additional Libraries 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 thats 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.

CHAPTER 3

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

CHAPTER 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 *microcontroller.Pin* from *board*. Taking in the ~microcontroller.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	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.

CHAPTER 5

Supported Ports

Adafruit's CircuitPython derivative currently has limited support with a focus on the Atmel SAMD21 port and ESP8266 port.

SAMD21x18

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

Pinout

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

microcontroller.pin	*board*			
Datasheet	arduino_zero	circuitplayground_express	feather_m0_adalogger	feather_m0_basi
PA00		ACCELEROMETER_SDA		
PA01		ACCELEROMETER_SCL		
PA02	A0	A0 / SPEAKER	A0	A0
PA03				
PB08	A1	A7 / TX	A1	A1
PB09	A2	A6 / RX	A2	A2
PA04	A3	IR_PROXIMITY	A3	A3
PA05	A4	A1	A4	A4
PA06	D8	A2		
PA07	D9	A3	D9	D9
PA08	D4	MICROPHONE_DO	SD_CS	

Table 5.1 – continued from previous page

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

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

microcontroller.pin	*analogio*		*audioio*	*bitbangio*			*busio*		
Datasheet	AnalogIn	AnalogOut	AudioOut	I2C	OneWire	SPI	I2C - SDA	I2C - SCL	OneW
PA00				**Yes**	**Yes**	**Yes**	**Yes**		**Yes**
PA01				**Yes**	**Yes**	**Yes**		**Yes**	**Yes**
PA02	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**			**Yes**
PA03	**Yes**			**Yes**	**Yes**	**Yes**			**Yes**
PB08	**Yes**			**Yes**	**Yes**	**Yes**	**Yes**		**Yes**
PB09	**Yes**			**Yes**	**Yes**	**Yes**		**Yes**	**Yes**
PA04	**Yes**			**Yes**	**Yes**	**Yes**			**Yes**
PA05	**Yes**			**Yes**	**Yes**	**Yes**			**Yes**
PA06	**Yes**			**Yes**	**Yes**	**Yes**			**Yes**
PA07	**Yes**			**Yes**	**Yes**	**Yes**			**Yes**
PA08	**Yes**			**Yes**	**Yes**	**Yes**	**Yes**		**Yes**
PA09	**Yes**			**Yes**	**Yes**	**Yes**		**Yes**	**Yes**

Table 5.2 – continue

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

Building

To build for the Arduino Zero:

```
make
```

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

```
make BOARD=feather_m0_basic
```

Board names are the directory names in the boards folder.

Deploying

Arduino Bootloader

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

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

No Bootloader via GDB

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

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

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

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

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

Connecting

Serial

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

screen /dev/tty.usbmodem142422 115200

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

Mass storage

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

Port Specific modules

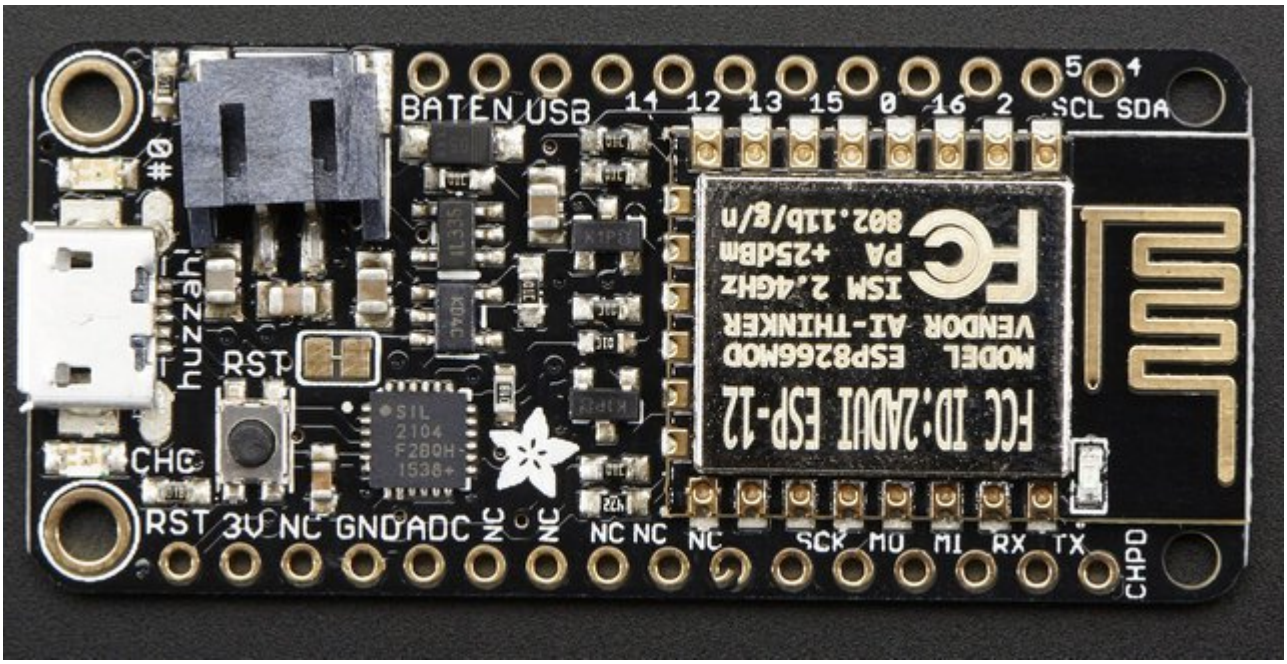
`samd` — SAMD implementation settings

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

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

ESP8266

Quick reference for the ESP8266



The Adafruit Feather HUZZAH board (image attribution: Adafruit).

Installing MicroPython

See the corresponding section of tutorial: *Getting started with MicroPython on the ESP8266*. It also includes a troubleshooting subsection.

General board control

The MicroPython REPL is on UART0 (GPIO1=TX, GPIO3=RX) at baudrate 115200. Tab-completion is useful to find out what methods an object has. Paste mode (ctrl-E) is useful to paste a large slab of Python code into the REPL.

The *machine* module:

```
import machine

machine.freq()          # get the current frequency of the CPU
machine.freq(160000000) # set the CPU frequency to 160 MHz
```

The *esp* module:

```
import esp

esp.osdebug(None)       # turn off vendor O/S debugging messages
esp.osdebug(0)          # redirect vendor O/S debugging messages to UART(0)
```

Networking

The *network* module:

```
import network

wlan = network.WLAN(network.STA_IF) # create station interface
wlan.active(True)       # activate the interface
wlan.scan()             # scan for access points
wlan.isconnected()      # check if the station is connected to an AP
wlan.connect('essid', 'password') # connect to an AP
wlan.config('mac')      # get the interface's MAC adddress
wlan.ifconfig()         # get the interface's IP/netmask/gw/DNS addresses

ap = network.WLAN(network.AP_IF) # create access-point interface
ap.active(True)         # activate the interface
ap.config(essid='ESP-AP') # set the ESSID of the access point
```

A useful function for connecting to your local WiFi network is:

```
def do_connect():
    import network
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    if not wlan.isconnected():
        print('connecting to network...')
        wlan.connect('essid', 'password')
        while not wlan.isconnected():
            pass
    print('network config:', wlan.ifconfig())
```

Once the network is established the *socket* module can be used to create and use TCP/UDP sockets as usual.

Delay and timing

Use the *time* module:

```
import time

time.sleep(1)           # sleep for 1 second
time.sleep_ms(500)      # sleep for 500 milliseconds
time.sleep_us(10)       # sleep for 10 microseconds
start = time.ticks_ms() # get millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start) # compute time difference
```

Timers

Virtual (RTOS-based) timers are supported. Use the *machine.Timer* class with timer ID of -1:

```
from machine import Timer

tim = Timer(-1)
tim.init(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(1))
tim.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(2))
```

The period is in milliseconds.

Pins and GPIO

Use the *machine.Pin* class:

```
from machine import Pin

p0 = Pin(0, Pin.OUT)    # create output pin on GPIO0
p0.on()                 # set pin to "on" (high) level
p0.off()                # set pin to "off" (low) level
p0.value(1)             # set pin to on/high

p2 = Pin(2, Pin.IN)     # create input pin on GPIO2
print(p2.value())       # get value, 0 or 1

p4 = Pin(4, Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor
p5 = Pin(5, Pin.OUT, value=1) # set pin high on creation
```

Available pins are: 0, 1, 2, 3, 4, 5, 12, 13, 14, 15, 16, which correspond to the actual GPIO pin numbers of ESP8266 chip. Note that many end-user boards use their own adhoc pin numbering (marked e.g. D0, D1, ...). As MicroPython supports different boards and modules, physical pin numbering was chosen as the lowest common denominator. For mapping between board logical pins and physical chip pins, consult your board documentation.

Note that Pin(1) and Pin(3) are REPL UART TX and RX respectively. Also note that Pin(16) is a special pin (used for wakeup from deepsleep mode) and may be not available for use with higher-level classes like `Neopixel`.

PWM (pulse width modulation)

PWM can be enabled on all pins except Pin(16). There is a single frequency for all channels, with range between 1 and 1000 (measured in Hz). The duty cycle is between 0 and 1023 inclusive.

Use the `machine.PWM` class:

```
from machine import Pin, PWM

pwm0 = PWM(Pin(0))      # create PWM object from a pin
pwm0.freq()             # get current frequency
pwm0.freq(1000)         # set frequency
pwm0.duty()             # get current duty cycle
pwm0.duty(200)          # set duty cycle
pwm0.deinit()           # turn off PWM on the pin

pwm2 = PWM(Pin(2), freq=500, duty=512) # create and configure in one go
```

ADC (analog to digital conversion)

ADC is available on a dedicated pin. Note that input voltages on the ADC pin must be between 0v and 1.0v.

Use the *machine.ADC* class:

```
from machine import ADC

adc = ADC(0)            # create ADC object on ADC pin
adc.read()              # read value, 0-1024
```

Software SPI bus

There are two SPI drivers. One is implemented in software (bit-banging) and works on all pins, and is accessed via the *machine.SPI* class:

```
from machine import Pin, SPI

# construct an SPI bus on the given pins
# polarity is the idle state of SCK
# phase=0 means sample on the first edge of SCK, phase=1 means the second
spi = SPI(-1, baudrate=100000, polarity=1, phase=0, sck=Pin(0), mosi=Pin(2),␣
↪miso=Pin(4))

spi.init(baudrate=200000) # set the baudrate

spi.read(10)            # read 10 bytes on MISO
spi.read(10, 0xff)      # read 10 bytes while outputing 0xff on MOSI

buf = bytearray(50)     # create a buffer
spi.readinto(buf)       # read into the given buffer (reads 50 bytes in this case)
spi.readinto(buf, 0xff) # read into the given buffer and output 0xff on MOSI

spi.write(b'12345')     # write 5 bytes on MOSI

buf = bytearray(4)      # create a buffer
spi.write_readinto(b'1234', buf) # write to MOSI and read from MISO into the buffer
spi.write_readinto(buf, buf) # write buf to MOSI and read MISO back into buf
```

Hardware SPI bus

The hardware SPI is faster (up to 80Mhz), but only works on following pins: `MISO` is GPIO12, `MOSI` is GPIO13, and `SCK` is GPIO14. It has the same methods as the bitbanging SPI class above, except for the pin parameters for the constructor and init (as those are fixed):

```
from machine import Pin, SPI

hspi = SPI(1, baudrate=80000000, polarity=0, phase=0)
```

(`SPI(0)` is used for FlashROM and not available to users.)

I2C bus

The I2C driver is implemented in software and works on all pins, and is accessed via the *machine.I2C* class:

```
from machine import Pin, I2C

# construct an I2C bus
i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)

i2c.readfrom(0x3a, 4)   # read 4 bytes from slave device with address 0x3a
i2c.writeto(0x3a, '12') # write '12' to slave device with address 0x3a

buf = bytearray(10)     # create a buffer with 10 bytes
i2c.writeto(0x3a, buf)  # write the given buffer to the slave
```

Deep-sleep mode

Connect GPIO16 to the reset pin (RST on HUZZAH). Then the following code can be used to sleep, wake and check the reset cause:

```
import machine

# configure RTC.ALARM0 to be able to wake the device
rtc = machine.RTC()
rtc.irq(trigger=rtc.ALARM0, wake=machine.DEEPSLEEP)

# check if the device woke from a deep sleep
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
    print('woke from a deep sleep')

# set RTC.ALARM0 to fire after 10 seconds (waking the device)
rtc.alarm(rtc.ALARM0, 10000)

# put the device to sleep
machine.deepsleep()
```

OneWire driver

The OneWire driver is implemented in software and works on all pins:

```
from machine import Pin
import onewire

ow = onewire.OneWire(Pin(12)) # create a OneWire bus on GPIO12
ow.scan()               # return a list of devices on the bus
ow.reset()              # reset the bus
ow.readbyte()           # read a byte
ow.writebyte(0x12)      # write a byte on the bus
ow.write('123')         # write bytes on the bus
ow.select_rom(b'12345678') # select a specific device by its ROM code
```

There is a specific driver for DS18S20 and DS18B20 devices:

```
import time, ds18x20
ds = ds18x20.DS18X20(ow)
roms = ds.scan()
ds.convert_temp()
time.sleep_ms(750)
for rom in roms:
    print(ds.read_temp(rom))
```

Be sure to put a 4.7k pull-up resistor on the data line. Note that the `convert_temp()` method must be called each time you want to sample the temperature.

NeoPixel driver

Use the `neopixel` module:

```
from machine import Pin
from neopixel import NeoPixel
```

```
pin = Pin(0, Pin.OUT)    # set GPIO0 to output to drive NeoPixels
np = NeoPixel(pin, 8)    # create NeoPixel driver on GPIO0 for 8 pixels
np[0] = (255, 255, 255) # set the first pixel to white
np.write()               # write data to all pixels
r, g, b = np[0]          # get first pixel colour
```

For low-level driving of a NeoPixel:

```
import esp
esp.neopixel_write(pin, grb_buf, is800khz)
```

APA102 driver

Use the `apa102` module:

```
from machine import Pin
from apa102 import APA102

clock = Pin(14, Pin.OUT)      # set GPIO14 to output to drive the clock
data = Pin(13, Pin.OUT)       # set GPIO13 to output to drive the data
apa = APA102(clock, data, 8) # create APA102 driver on the clock and the data pin for␣
↪8 pixels
apa[0] = (255, 255, 255, 31) # set the first pixel to white with a maximum brightness␣
↪of 31
apa.write()                   # write data to all pixels
r, g, b, brightness = apa[0] # get first pixel colour
```

For low-level driving of an APA102:

```
import esp
esp.apa102_write(clock_pin, data_pin, rgbi_buf)
```

DHT driver

The DHT driver is implemented in software and works on all pins:

```
import dht
import machine

d = dht.DHT11(machine.Pin(4))
d.measure()
d.temperature() # eg. 23 (°C)
d.humidity()    # eg. 41 (% RH)

d = dht.DHT22(machine.Pin(4))
d.measure()
d.temperature() # eg. 23.6 (°C)
d.humidity()    # eg. 41.3 (% RH)
```

WebREPL (web browser interactive prompt)

WebREPL (REPL over WebSockets, accessible via a web browser) is an experimental feature available in ESP8266 port. Download web client from https://github.com/micropython/webrepl (hosted version available at

http://micropython.org/webrepl), and configure it by executing:

```
import webrepl_setup
```

and following on-screen instructions. After reboot, it will be available for connection. If you disabled automatic start-up on boot, you may run configured daemon on demand using:

```
import webrepl
webrepl.start()
```

The supported way to use WebREPL is by connecting to ESP8266 access point, but the daemon is also started on STA interface if it is active, so if your router is set up and works correctly, you may also use WebREPL while connected to your normal Internet access point (use the ESP8266 AP connection method if you face any issues).

Besides terminal/command prompt access, WebREPL also has provision for file transfer (both upload and download). Web client has buttons for the corresponding functions, or you can use command-line client `webrepl_cli.py` from the repository above.

See the MicroPython forum for other community-supported alternatives to transfer files to ESP8266.

General information about the ESP8266 port

ESP8266 is a popular WiFi-enabled System-on-Chip (SoC) by Espressif Systems.

Multitude of boards

There are a multitude of modules and boards from different sources which carry the ESP8266 chip. MicroPython tries to provide a generic port which would run on as many boards/modules as possible, but there may be limitations. Adafruit Feather HUZZAH board is taken as a reference board for the port (for example, testing is performed on it). If you have another board, please make sure you have datasheet, schematics and other reference materials for your board handy to look up various aspects of your board functioning.

To make a generic ESP8266 port and support as many boards as possible, following design and implementation decision were made:

- GPIO pin numbering is based on ESP8266 chip numbering, not some "logical" numbering of a particular board. Please have the manual/pin diagram of your board at hand to find correspondence between your board pins and actual ESP8266 pins. We also encourage users of various boards to share this mapping via MicroPython forum, with the idea to collect community-maintained reference materials eventually.
- All pins which make sense to support, are supported by MicroPython (for example, pins which are used to connect SPI flash are not exposed, as they're unlikely useful for anything else, and operating on them will lead to board lock-up). However, any particular board may expose only subset of pins. Consult your board reference manual.
- Some boards may lack external pins/internal connectivity to support ESP8266 deepsleep mode.

Technical specifications and SoC datasheets

The datasheets and other reference material for ESP8266 chip are available from the vendor site: http://bbs.espressif.com/viewtopic.php?f=67&t=225 . They are the primary reference for the chip technical specifications, capabilities, operating modes, internal functioning, etc.

For your convenience, some of technical specifications are provided below:

- Architecture: Xtensa lx106

- CPU frequency: 80MHz overclockable to 160MHz
- Total RAM available: 96KB (part of it reserved for system)
- BootROM: 64KB
- Internal FlashROM: None
- External FlashROM: code and data, via SPI Flash. Normal sizes 512KB-4MB.
- GPIO: 16 + 1 (GPIOs are multiplexed with other functions, including external FlashROM, UART, deep sleep wake-up, etc.)
- UART: One RX/TX UART (no hardware handshaking), one TX-only UART.
- SPI: 2 SPI interfaces (one used for FlashROM).
- I2C: No native external I2C (bitbang implementation available on any pins).
- I2S: 1.
- Programming: using BootROM bootloader from UART. Due to external FlashROM and always-available BootROM bootloader, ESP8266 is not brickable.

Scarcity of runtime resources

ESP8266 has very modest resources (first of all, RAM memory). So, please avoid allocating too big container objects (lists, dictionaries) and buffers. There is also no full-fledged OS to keep track of resources and automatically clean them up, so that's the task of a user/user application: please be sure to close open files, sockets, etc. as soon as possible after use.

Boot process

On boot, MicroPython EPS8266 port executes `_boot.py` script from internal frozen modules. It mounts filesystem in FlashROM, or if it's not available, performs first-time setup of the module and creates the filesystem. This part of the boot process is considered fixed, and not available for customization for end users (even if you build from source, please refrain from changes to it; customization of early boot process is available only to advanced users and developers, who can diagnose themselves any issues arising from modifying the standard process).

Once the filesystem is mounted, `boot.py` is executed from it. The standard version of this file is created during first-time module set up and has commands to start a WebREPL daemon (disabled by default, configurable with `webrepl_setup` module), etc. This file is customizable by end users (for example, you may want to set some parameters or add other services which should be run on a module start-up). But keep in mind that incorrect modifications to boot.py may still lead to boot loops or lock ups, requiring to reflash a module from scratch. (In particular, it's recommended that you use either `webrepl_setup` module or manual editing to configure WebREPL, but not both).

As a final step of boot procedure, `main.py` is executed from filesystem, if exists. This file is a hook to start up a user application each time on boot (instead of going to REPL). For small test applications, you may name them directly as `main.py`, and upload to module, but instead it's recommended to keep your application(s) in separate files, and have just the following in `main.py`:

```
import my_app
my_app.main()
```

This will allow to keep the structure of your application clear, as well as allow to install multiple applications on a board, and switch among them.

Known Issues

Real-time clock

RTC in ESP8266 has very bad accuracy, drift may be seconds per minute. As a workaround, to measure short enough intervals you can use `utime.time()`, etc. functions, and for wall clock time, synchronize from the net using included `ntpdate.py` module.

Due to limitations of the ESP8266 chip the internal real-time clock (RTC) will overflow every 7:45h. If a long-term working RTC time is required then `time()` or `localtime()` must be called at least once within 7 hours. MicroPython will then handle the overflow.

MicroPython tutorial for ESP8266

This tutorial is intended to get you started using MicroPython on the ESP8266 system-on-a-chip. If it is your first time it is recommended to follow the tutorial through in the order below. Otherwise the sections are mostly self contained, so feel free to skip to those that interest you.

The tutorial does not assume that you know Python, but it also does not attempt to explain any of the details of the Python language. Instead it provides you with commands that are ready to run, and hopes that you will gain a bit of Python knowledge along the way. To learn more about Python itself please refer to https://www.python.org.

Getting started with MicroPython on the ESP8266

Using MicroPython is a great way to get the most of your ESP8266 board. And vice versa, the ESP8266 chip is a great platform for using MicroPython. This tutorial will guide you through setting up MicroPython, getting a prompt, using WebREPL, connecting to the network and communicating with the Internet, using the hardware peripherals, and controlling some external components.

Let's get started!

Requirements

The first thing you need is a board with an ESP8266 chip. The MicroPython software supports the ESP8266 chip itself and any board should work. The main characteristic of a board is how much flash it has, how the GPIO pins are connected to the outside world, and whether it includes a built-in USB-serial convertor to make the UART available to your PC.

The minimum requirement for flash size is 1Mbyte. There is also a special build for boards with 512KB, but it is highly limited comparing to the normal build: there is no support for filesystem, and thus features which depend on it won't work (WebREPL, upip, etc.). As such, 512KB build will be more interesting for users who build from source and fine-tune parameters for their particular application.

Names of pins will be given in this tutorial using the chip names (eg GPIO0) and it should be straightforward to find which pin this corresponds to on your particular board.

Powering the board

If your board has a USB connector on it then most likely it is powered through this when connected to your PC. Otherwise you will need to power it directly. Please refer to the documentation for your board for further details.

Getting the firmware

The first thing you need to do is download the most recent MicroPython firmware .bin file to load onto your ESP8266 device. You can download it from the MicroPython downloads page. From here, you have 3 main choices

- Stable firmware builds for 1024kb modules and above.
- Daily firmware builds for 1024kb modules and above.
- Daily firmware builds for 512kb modules.

The best bet is nearly always to go for the Stable firmware builds. An exception to this though is if you have an ESP8266 module with only 512kb of onboard storage. You can easily tell by trying to load a Stable firmware build and if you get the error below, then you may have to use the Daily firmware builds for 512kb modules.

> WARNING: Unlikely to work as data goes beyond end of flash.

Deploying the firmware

Once you have the MicroPython firmware (compiled code), you need to load it onto your ESP8266 device. There are two main steps to do this: first you need to put your device in boot-loader mode, and second you need to copy across the firmware. The exact procedure for these steps is highly dependent on the particular board and you will need to refer to its documentation for details.

If you have a board that has a USB connector, a USB-serial convertor, and has the DTR and RTS pins wired in a special way then deploying the firmware should be easy as all steps can be done automatically. Boards that have such features include the Adafruit Feather HUZZAH and NodeMCU boards.

For best results it is recommended to first erase the entire flash of your device before putting on new MicroPython firmware.

Currently we only support esptool.py to copy across the firmware. You can find this tool here: https://github.com/espressif/esptool/, or install it using pip:

```
pip install esptool
```

Versions starting with 1.3 support both Python 2.7 and Python 3.4 (or newer). An older version (at least 1.2.1 is needed) works fine but will require Python 2.7.

Any other flashing program should work, so feel free to try them out or refer to the documentation for your board to see its recommendations.

Using esptool.py you can erase the flash with the command:

```
esptool.py --port /dev/ttyUSB0 erase_flash
```

And then deploy the new firmware using:

```
esptool.py --port /dev/ttyUSB0 --baud 460800 write_flash --flash_size=detect 0␣
↪esp8266-20170108-v1.8.7.bin
```

You might need to change the "port" setting to something else relevant for your PC. You may also need to reduce the baudrate if you get errors when flashing (eg down to 115200). The filename of the firmware should also match the file that you have.

For some boards with a particular FlashROM configuration (e.g. some variants of a NodeMCU board) you may need to use the following command to deploy the firmware (note the `-fm dio` option):

```
esptool.py --port /dev/ttyUSB0 --baud 460800 write_flash --flash_size=detect -fm dio␣
↪0 esp8266-20170108-v1.8.7.bin
```

If the above commands run without error then MicroPython should be installed on your board!

Serial prompt

Once you have the firmware on the device you can access the REPL (Python prompt) over UART0 (GPIO1=TX, GPIO3=RX), which might be connected to a USB-serial convertor, depending on your board. The baudrate is 115200. The next part of the tutorial will discuss the prompt in more detail.

WiFi

After a fresh install and boot the device configures itself as a WiFi access point (AP) that you can connect to. The ESSID is of the form MicroPython-xxxxxx where the x's are replaced with part of the MAC address of your device (so will be the same everytime, and most likely different for all ESP8266 chips). The password for the WiFi is micropythoN (note the upper-case N). Its IP address will be 192.168.4.1 once you connect to its network. WiFi configuration will be discussed in more detail later in the tutorial.

Troubleshooting installation problems

If you experience problems during flashing or with running firmware immediately after it, here are troubleshooting recommendations:

- Be aware of and try to exclude hardware problems. There are 2 common problems: bad power source quality and worn-out/defective FlashROM. Speaking of power source, not just raw amperage is important, but also low ripple and noise/EMI in general. If you experience issues with self-made or wall-wart style power supply, try USB power from a computer. Unearthed power supplies are also known to cause problems as they source of increased EMI (electromagnetic interference) - at the very least, and may lead to electrical devices breakdown. So, you are advised to avoid using unearthed power connections when working with ESP8266 and other boards. In regard to FlashROM hardware problems, there are independent (not related to MicroPython in any way) reports (e.g.) that on some ESP8266 modules, FlashROM can be programmed as little as 20 times before programming errors occur. This is *much* less than 100,000 programming cycles cited for FlashROM chips of a type used with ESP8266 by reputable vendors, which points to either production rejects, or second-hand worn-out flash chips to be used on some (apparently cheap) modules/boards. You may want to use your best judgement about source, price, documentation, warranty, post-sales support for the modules/boards you purchase.
- The flashing instructions above use flashing speed of 460800 baud, which is good compromise between speed and stability. However, depending on your module/board, USB-UART convertor, cables, host OS, etc., the above baud rate may be too high and lead to errors. Try a more common 115200 baud rate instead in such cases.
- If lower baud rate didn't help, you may want to try older version of esptool.py, which had a different programming algorithm:

  ```
  pip install esptool==1.0.1
  ```

  This version doesn't support `--flash_size=detect` option, so you will need to specify FlashROM size explicitly (in megabits). It also requires Python 2.7, so you may need to use `pip2` instead of `pip` in the command above.
- The `--flash_size` option in the commands above is mandatory. Omitting it will lead to a corrupted firmware.

- To catch incorrect flash content (e.g. from a defective sector on a chip), add `--verify` switch to the commands above.
- Additionally, you can check the firmware integrity from a MicroPython REPL prompt (assuming you were able to flash it and `--verify` option doesn't report errors):

```
import esp
esp.check_fw()
```

  If the last output value is True, the firmware is OK. Otherwise, it's corrupted and need to be reflashed correctly.
- If you experience any issues with another flashing application (not esptool.py), try esptool.py, it is a generally accepted flashing application in the ESP8266 community.
- If you still experience problems with even flashing the firmware, please refer to esptool.py project page, https://github.com/espressif/esptool for additional documentation and bug tracker where you can report problems.
- If you are able to flash firmware, but `--verify` option or `esp.check_fw()` return errors even after multiple retries, you may have a defective FlashROM chip, as explained above.

Getting a MicroPython REPL prompt

REPL stands for Read Evaluate Print Loop, and is the name given to the interactive MicroPython prompt that you can access on the ESP8266. Using the REPL is by far the easiest way to test out your code and run commands.

There are two ways to access the REPL: either via a wired connection through the UART serial port, or via WiFi.

REPL over the serial port

The REPL is always available on the UART0 serial peripheral, which is connected to the pins GPIO1 for TX and GPIO3 for RX. The baudrate of the REPL is 115200. If your board has a USB-serial convertor on it then you should be able to access the REPL directly from your PC. Otherwise you will need to have a way of communicating with the UART.

To access the prompt over USB-serial you need to use a terminal emulator program. On Windows TeraTerm is a good choice, on Mac you can use the built-in screen program, and Linux has picocom and minicom. Of course, there are many other terminal programs that will work, so pick your favourite!

For example, on Linux you can try running:

```
picocom /dev/ttyUSB0 -b115200
```

Once you have made the connection over the serial port you can test if it is working by hitting enter a few times. You should see the Python REPL prompt, indicated by >>>.

WebREPL - a prompt over WiFi

WebREPL allows you to use the Python prompt over WiFi, connecting through a browser. The latest versions of Firefox and Chrome are supported.

For your convenience, WebREPL client is hosted at http://micropython.org/webrepl . Alternatively, you can install it locally from the the GitHub repository https://github.com/micropython/webrepl .

Before connecting to WebREPL, you should set a password and enable it via a normal serial connection. Initial versions of MicroPython for ESP8266 came with WebREPL automatically enabled on the boot and with the ability to

set a password via WiFi on the first connection, but as WebREPL was becoming more widely known and popular, the initial setup has switched to a wired connection for improved security:

```
import webrepl_setup
```

Follow the on-screen instructions and prompts. To make any changes active, you will need to reboot your device.

To use WebREPL connect your computer to the ESP8266's access point (MicroPython-xxxxxx, see the previous section about this). If you have already reconfigured your ESP8266 to connect to a router then you can skip this part.

Once you are on the same network as the ESP8266 you click the "Connect" button (if you are connecting via a router then you may need to change the IP address, by default the IP address is correct when connected to the ESP8266's access point). If the connection succeeds then you should see a password prompt.

Once you type the password configured at the setup step above, press Enter once more and you should get a prompt looking like >>>. You can now start typing Python commands!

Using the REPL

Once you have a prompt you can start experimenting! Anything you type at the prompt will be executed after you press the Enter key. MicroPython will run the code that you enter and print the result (if there is one). If there is an error with the text that you enter then an error message is printed.

Try typing the following at the prompt:

```
>>> print('hello esp8266!')
hello esp8266!
```

Note that you shouldn't type the >>> arrows, they are there to indicate that you should type the text after it at the prompt. And then the line following is what the device should respond with. In the end, once you have entered the text `print("hello esp8266!")` and pressed the Enter key, the output on your screen should look exactly like it does above.

If you already know some python you can now try some basic commands here. For example:

```
>>> 1 + 2
3
>>> 1 / 2
0.5
>>> 12**34
49222352429520267040371132431220080 64
```

If your board has an LED attached to GPIO2 (the ESP-12 modules do) then you can turn it on and off using the following code:

```
>>> import machine
>>> pin = machine.Pin(2, machine.Pin.OUT)
>>> pin.on()
>>> pin.off()
```

Note that `on` method of a Pin might turn the LED off and `off` might turn it on (or vice versa), depending on how the LED is wired on your board. To resolve this, machine.Signal class is provided.

Line editing

You can edit the current line that you are entering using the left and right arrow keys to move the cursor, as well as the delete and backspace keys. Also, pressing Home or ctrl-A moves the cursor to the start of the line, and pressing End

or ctrl-E moves to the end of the line.

Input history

The REPL remembers a certain number of previous lines of text that you entered (up to 8 on the ESP8266). To recall previous lines use the up and down arrow keys.

Tab completion

Pressing the Tab key will do an auto-completion of the current word that you are entering. This can be very useful to find out functions and methods that a module or object has. Try it out by typing "ma" and then pressing Tab. It should complete to "machine" (assuming you imported machine in the above example). Then type "." and press Tab again to see a list of all the functions that the machine module has.

Line continuation and auto-indent

Certain things that you type will need "continuing", that is, will need more lines of text to make a proper Python statement. In this case the prompt will change to `...` and the cursor will auto-indent the correct amount so you can start typing the next line straight away. Try this by defining the following function:

```
>>> def toggle(p):
...     p.value(not p.value())
...
...
...
>>>
```

In the above, you needed to press the Enter key three times in a row to finish the compound statement (that's the three lines with just dots on them). The other way to finish a compound statement is to press backspace to get to the start of the line, then press the Enter key. (If you did something wrong and want to escape the continuation mode then press ctrl-C; all lines will be ignored.)

The function you just defined allows you to toggle a pin. The pin object you created earlier should still exist (recreate it if it doesn't) and you can toggle the LED using:

```
>>> toggle(pin)
```

Let's now toggle the LED in a loop (if you don't have an LED then you can just print some text instead of calling toggle, to see the effect):

```
>>> import time
>>> while True:
...     toggle(pin)
...     time.sleep_ms(500)
...
...
...
>>>
```

This will toggle the LED at 1Hz (half a second on, half a second off). To stop the toggling press ctrl-C, which will raise a KeyboardInterrupt exception and break out of the loop.

The time module provides some useful functions for making delays and doing timing. Use tab completion to find out what they are and play around with them!

Paste mode

Pressing ctrl-E will enter a special paste mode. This allows you to copy and paste a chunk of text into the REPL. If you press ctrl-E you will see the paste-mode prompt:

```
paste mode; Ctrl-C to cancel, Ctrl-D to finish
===
```

You can then paste (or type) your text in. Note that none of the special keys or commands work in paste mode (eg Tab or backspace), they are just accepted as-is. Press ctrl-D to finish entering the text and execute it.

Other control commands

There are four other control commands:

- Ctrl-A on a blank line will enter raw REPL mode. This is like a permanent paste mode, except that characters are not echoed back.
- Ctrl-B on a blank like goes to normal REPL mode.
- Ctrl-C cancels any input, or interrupts the currently running code.
- Ctrl-D on a blank line will do a soft reset.

Note that ctrl-A and ctrl-D do not work with WebREPL.

The internal filesystem

If your devices has 1Mbyte or more of storage then it will be set up (upon first boot) to contain a filesystem. This filesystem uses the FAT format and is stored in the flash after the MicroPython firmware.

Creating and reading files

MicroPython on the ESP8266 supports the standard way of accessing files in Python, using the built-in `open()` function.

To create a file try:

```
>>> f = open('data.txt', 'w')
>>> f.write('some data')
9
>>> f.close()
```

The "9" is the number of bytes that were written with the `write()` method. Then you can read back the contents of this new file using:

```
>>> f = open('data.txt')
>>> f.read()
'some data'
>>> f.close()
```

Note that the default mode when opening a file is to open it in read-only mode, and as a text file. Specify `'wb'` as the second argument to `open()` to open for writing in binary mode, and `'rb'` to open for reading in binary mode.

Listing file and more

The os module can be used for further control over the filesystem. First import the module:

```
>>> import os
```

Then try listing the contents of the filesystem:

```
>>> os.listdir()
['boot.py', 'port_config.py', 'data.txt']
```

You can make directories:

```
>>> os.mkdir('dir')
```

And remove entries:

```
>>> os.remove('data.txt')
```

Start up scripts

There are two files that are treated specially by the ESP8266 when it starts up: boot.py and main.py. The boot.py script is executed first (if it exists) and then once it completes the main.py script is executed. You can create these files yourself and populate them with the code that you want to run when the device starts up.

Accessing the filesystem via WebREPL

You can access the filesystem over WebREPL using the web client in a browser or via the command-line tool. Please refer to Quick Reference and Tutorial sections for more information about WebREPL.

Network basics

The network module is used to configure the WiFi connection. There are two WiFi interfaces, one for the station (when the ESP8266 connects to a router) and one for the access point (for other devices to connect to the ESP8266). Create instances of these objects using:

```
>>> import network
>>> sta_if = network.WLAN(network.STA_IF)
>>> ap_if = network.WLAN(network.AP_IF)
```

You can check if the interfaces are active by:

```
>>> sta_if.active()
False
>>> ap_if.active()
True
```

You can also check the network settings of the interface by:

```
>>> ap_if.ifconfig()
('192.168.4.1', '255.255.255.0', '192.168.4.1', '8.8.8.8')
```

The returned values are: IP address, netmask, gateway, DNS.

Configuration of the WiFi

Upon a fresh install the ESP8266 is configured in access point mode, so the AP_IF interface is active and the STA_IF interface is inactive. You can configure the module to connect to your own network using the STA_IF interface.

First activate the station interface:

```
>>> sta_if.active(True)
```

Then connect to your WiFi network:

```
>>> sta_if.connect('<your ESSID>', '<your password>')
```

To check if the connection is established use:

```
>>> sta_if.isconnected()
```

Once established you can check the IP address:

```
>>> sta_if.ifconfig()
('192.168.0.2', '255.255.255.0', '192.168.0.1', '8.8.8.8')
```

You can then disable the access-point interface if you no longer need it:

```
>>> ap_if.active(False)
```

Here is a function you can run (or put in your boot.py file) to automatically connect to your WiFi network:

```
def do_connect():
    import network
    sta_if = network.WLAN(network.STA_IF)
    if not sta_if.isconnected():
        print('connecting to network...')
        sta_if.active(True)
        sta_if.connect('<essid>', '<password>')
        while not sta_if.isconnected():
            pass
    print('network config:', sta_if.ifconfig())
```

Sockets

Once the WiFi is set up the way to access the network is by using sockets. A socket represents an endpoint on a network device, and when two sockets are connected together communication can proceed. Internet protocols are built on top of sockets, such as email (SMTP), the web (HTTP), telnet, ssh, among many others. Each of these protocols is assigned a specific port, which is just an integer. Given an IP address and a port number you can connect to a remote device and start talking with it.

The next part of the tutorial discusses how to use sockets to do some common and useful network tasks.

Network - TCP sockets

The building block of most of the internet is the TCP socket. These sockets provide a reliable stream of bytes between the connected network devices. This part of the tutorial will show how to use TCP sockets in a few different cases.

Star Wars Asciimation

The simplest thing to do is to download data from the internet. In this case we will use the Star Wars Asciimation service provided by the blinkenlights.nl website. It uses the telnet protocol on port 23 to stream data to anyone that connects. It's very simple to use because it doesn't require you to authenticate (give a username or password), you can just start downloading data straight away.

The first thing to do is make sure we have the socket module available:

```
>>> import socket
```

Then get the IP address of the server:

```
>>> addr_info = socket.getaddrinfo("towel.blinkenlights.nl", 23)
```

The `getaddrinfo` function actually returns a list of addresses, and each address has more information than we need. We want to get just the first valid address, and then just the IP address and port of the server. To do this use:

```
>>> addr = addr_info[0][-1]
```

If you type `addr_info` and `addr` at the prompt you will see exactly what information they hold.

Using the IP address we can make a socket and connect to the server:

```
>>> s = socket.socket()
>>> s.connect(addr)
```

Now that we are connected we can download and display the data:

```
>>> while True:
...     data = s.recv(500)
...     print(str(data, 'utf8'), end='')
...
```

When this loop executes it should start showing the animation (use ctrl-C to interrupt it).

You should also be able to run this same code on your PC using normal Python if you want to try it out there.

HTTP GET request

The next example shows how to download a webpage. HTTP uses port 80 and you first need to send a "GET" request before you can download anything. As part of the request you need to specify the page to retrieve.

Let's define a function that can download and print a URL:

```
def http_get(url):
    _, _, host, path = url.split('/', 3)
    addr = socket.getaddrinfo(host, 80)[0][-1]
    s = socket.socket()
    s.connect(addr)
    s.send(bytes('GET /%s HTTP/1.0\r\nHost: %s\r\n\r\n' % (path, host), 'utf8'))
    while True:
        data = s.recv(100)
        if data:
            print(str(data, 'utf8'), end='')
        else:
            break
    s.close()
```

Make sure that you import the socket module before running this function. Then you can try:

```
>>> http_get('http://micropython.org/ks/test.html')
```

This should retrieve the webpage and print the HTML to the console.

Simple HTTP server

The following code creates an simple HTTP server which serves a single webpage that contains a table with the state of all the GPIO pins:

```
import machine
pins = [machine.Pin(i, machine.Pin.IN) for i in (0, 2, 4, 5, 12, 13, 14, 15)]

html = """<!DOCTYPE html>
<html>
    <head> <title>ESP8266 Pins</title> </head>
    <body> <h1>ESP8266 Pins</h1>
        <table border="1"> <tr><th>Pin</th><th>Value</th></tr> %s </table>
    </body>
</html>
"""

import socket
addr = socket.getaddrinfo('0.0.0.0', 80)[0][-1]

s = socket.socket()
s.bind(addr)
s.listen(1)

print('listening on', addr)

while True:
    cl, addr = s.accept()
    print('client connected from', addr)
    cl_file = cl.makefile('rwb', 0)
    while True:
        line = cl_file.readline()
        if not line or line == b'\r\n':
            break
    rows = ['<tr><td>%s</td><td>%d</td></tr>' % (str(p), p.value()) for p in pins]
    response = html % '\n'.join(rows)
    cl.send(response)
    cl.close()
```

GPIO Pins

The way to connect your board to the external world, and control other components, is through the GPIO pins. Not all pins are available to use, in most cases only pins 0, 2, 4, 5, 12, 13, 14, 15, and 16 can be used.

The pins are available in the machine module, so make sure you import that first. Then you can create a pin using:

```
>>> pin = machine.Pin(0)
```

Here, the "0" is the pin that you want to access. Usually you want to configure the pin to be input or output, and you do this when constructing it. To make an input pin use:

```
>>> pin = machine.Pin(0, machine.Pin.IN, machine.Pin.PULL_UP)
```

You can either use PULL_UP or None for the input pull-mode. If it's not specified then it defaults to None, which is no pull resistor. You can read the value on the pin using:

```
>>> pin.value()
0
```

The pin on your board may return 0 or 1 here, depending on what it's connected to. To make an output pin use:

```
>>> pin = machine.Pin(0, machine.Pin.OUT)
```

Then set its value using:

```
>>> pin.value(0)
>>> pin.value(1)
```

Or:

```
>>> pin.off()
>>> pin.on()
```

External interrupts

All pins except number 16 can be configured to trigger a hard interrupt if their input changes. You can set code (a callback function) to be executed on the trigger.

Let's first define a callback function, which must take a single argument, being the pin that triggered the function. We will make the function just print the pin:

```
>>> def callback(p):
...     print('pin change', p)
```

Next we will create two pins and configure them as inputs:

```
>>> from machine import Pin
>>> p0 = Pin(0, Pin.IN)
>>> p2 = Pin(2, Pin.IN)
```

An finally we need to tell the pins when to trigger, and the function to call when they detect an event:

```
>>> p0.irq(trigger=Pin.IRQ_FALLING, handler=callback)
>>> p2.irq(trigger=Pin.IRQ_RISING | Pin.IRQ_FALLING, handler=callback)
```

We set pin 0 to trigger only on a falling edge of the input (when it goes from high to low), and set pin 2 to trigger on both a rising and falling edge. After entering this code you can apply high and low voltages to pins 0 and 2 to see the interrupt being executed.

A hard interrupt will trigger as soon as the event occurs and will interrupt any running code, including Python code. As such your callback functions are limited in what they can do (they cannot allocate memory, for example) and should be as short and simple as possible.

Pulse Width Modulation

Pulse width modulation (PWM) is a way to get an artificial analog output on a digital pin. It achieves this by rapidly toggling the pin from low to high. There are two parameters associated with this: the frequency of the toggling, and the duty cycle. The duty cycle is defined to be how long the pin is high compared with the length of a single period (low plus high time). Maximum duty cycle is when the pin is high all of the time, and minimum is when it is low all of the time.

On the ESP8266 the pins 0, 2, 4, 5, 12, 13, 14 and 15 all support PWM. The limitation is that they must all be at the same frequency, and the frequency must be between 1Hz and 1kHz.

To use PWM on a pin you must first create the pin object, for example:

```
>>> import machine
>>> p12 = machine.Pin(12)
```

Then create the PWM object using:

```
>>> pwm12 = machine.PWM(p12)
```

You can set the frequency and duty cycle using:

```
>>> pwm12.freq(500)
>>> pwm12.duty(512)
```

Note that the duty cycle is between 0 (all off) and 1023 (all on), with 512 being a 50% duty. If you print the PWM object then it will tell you its current configuration:

```
>>> pwm12
PWM(12, freq=500, duty=512)
```

You can also call the `freq()` and `duty()` methods with no arguments to get their current values.

The pin will continue to be in PWM mode until you deinitialise it using:

```
>>> pwm12.deinit()
```

Fading an LED

Let's use the PWM feature to fade an LED. Assuming your board has an LED connected to pin 2 (ESP-12 modules do) we can create an LED-PWM object using:

```
>>> led = machine.PWM(machine.Pin(2), freq=1000)
```

Notice that we can set the frequency in the PWM constructor.

For the next part we will use timing and some math, so import these modules:

```
>>> import time, math
```

Then create a function to pulse the LED:

```
>>> def pulse(l, t):
...     for i in range(20):
...         l.duty(int(math.sin(i / 10 * math.pi) * 500 + 500))
...         time.sleep_ms(t)
```

You can try this function out using:

```
>>> pulse(led, 50)
```

For a nice effect you can pulse many times in a row:

```
>>> for i in range(10):
...     pulse(led, 20)
```

Remember you can use ctrl-C to interrupt the code.

Control a hobby servo

Hobby servo motors can be controlled using PWM. They require a frequency of 50Hz and then a duty between about 40 and 115, with 77 being the centre value. If you connect a servo to the power and ground pins, and then the signal line to pin 12 (other pins will work just as well), you can control the motor using:

```
>>> servo = machine.PWM(machine.Pin(12), freq=50)
>>> servo.duty(40)
>>> servo.duty(115)
>>> servo.duty(77)
```

Analog to Digital Conversion

The ESP8266 has a single pin (separate to the GPIO pins) which can be used to read analog voltages and convert them to a digital value. You can construct such an ADC pin object using:

```
>>> import machine
>>> adc = machine.ADC(0)
```

Then read its value with:

```
>>> adc.read()
58
```

The values returned from the `read()` function are between 0 (for 0.0 volts) and 1024 (for 1.0 volts). Please note that this input can only tolerate a maximum of 1.0 volts and you must use a voltage divider circuit to measure larger voltages.

Power control

The ESP8266 provides the ability to change the CPU frequency on the fly, and enter a deep-sleep state. Both can be used to manage power consumption.

Changing the CPU frequency

The machine module has a function to get and set the CPU frequency. To get the current frequency use:

```
>>> import machine
>>> machine.freq()
80000000
```

By default the CPU runs at 80MHz. It can be change to 160MHz if you need more processing power, at the expense of current consumption:

```
>>> machine.freq(160000000)
>>> machine.freq()
160000000
```

You can change to the higher frequency just while your code does the heavy processing and then change back when its finished.

Deep-sleep mode

The deep-sleep mode will shut down the ESP8266 and all its peripherals, including the WiFi (but not including the real-time-clock, which is used to wake the chip). This drastically reduces current consumption and is a good way to make devices that can run for a while on a battery.

To be able to use the deep-sleep feature you must connect GPIO16 to the reset pin (RST on the Adafruit Feather HUZZAH board). Then the following code can be used to sleep and wake the device:

```
import machine

# configure RTC.ALARM0 to be able to wake the device
rtc = machine.RTC()
rtc.irq(trigger=rtc.ALARM0, wake=machine.DEEPSLEEP)

# set RTC.ALARM0 to fire after 10 seconds (waking the device)
rtc.alarm(rtc.ALARM0, 10000)

# put the device to sleep
machine.deepsleep()
```

Note that when the chip wakes from a deep-sleep it is completely reset, including all of the memory. The boot scripts will run as usual and you can put code in them to check the reset cause to perhaps do something different if the device just woke from a deep-sleep. For example, to print the reset cause you can use:

```
if machine.reset_cause() == machine.DEEPSLEEP_RESET:
    print('woke from a deep sleep')
else:
    print('power on or hard reset')
```

Controlling 1-wire devices

The 1-wire bus is a serial bus that uses just a single wire for communication (in addition to wires for ground and power). The DS18B20 temperature sensor is a very popular 1-wire device, and here we show how to use the onewire module to read from such a device.

For the following code to work you need to have at least one DS18S20 or DS18B20 temperature sensor with its data line connected to GPIO12. You must also power the sensors and connect a 4.7k Ohm resistor between the data pin and the power pin.

```
import time
import machine
import onewire, ds18x20

# the device is on GPIO12
```

```
dat = machine.Pin(12)

# create the onewire object
ds = ds18x20.DS18X20(onewire.OneWire(dat))

# scan for devices on the bus
roms = ds.scan()
print('found devices:', roms)

# loop 10 times and print all temperatures
for i in range(10):
    print('temperatures:', end=' ')
    ds.convert_temp()
    time.sleep_ms(750)
    for rom in roms:
        print(ds.read_temp(rom), end=' ')
    print()
```

Note that you must execute the `convert_temp()` function to initiate a temperature reading, then wait at least 750ms before reading the value.

Controlling NeoPixels

NeoPixels, also known as WS2812 LEDs, are full-colour LEDs that are connected in serial, are individually addressable, and can have their red, green and blue components set between 0 and 255. They require precise timing to control them and there is a special neopixel module to do just this.

To create a NeoPixel object do the following:

```
>>> import machine, neopixel
>>> np = neopixel.NeoPixel(machine.Pin(4), 8)
```

This configures a NeoPixel strip on GPIO4 with 8 pixels. You can adjust the "4" (pin number) and the "8" (number of pixel) to suit your set up.

To set the colour of pixels use:

```
>>> np[0] = (255, 0, 0) # set to red, full brightness
>>> np[1] = (0, 128, 0) # set to green, half brightness
>>> np[2] = (0, 0, 64)  # set to blue, quarter brightness
```

Then use the `write()` method to output the colours to the LEDs:

```
>>> np.write()
```

The following demo function makes a fancy show on the LEDs:

```
import time

def demo(np):
    n = np.n

    # cycle
    for i in range(4 * n):
        for j in range(n):
            np[j] = (0, 0, 0)
        np[i % n] = (255, 255, 255)
```

```
        np.write()
        time.sleep_ms(25)

    # bounce
    for i in range(4 * n):
        for j in range(n):
            np[j] = (0, 0, 128)
        if (i // n) % 2 == 0:
            np[i % n] = (0, 0, 0)
        else:
            np[n - 1 - (i % n)] = (0, 0, 0)
        np.write()
        time.sleep_ms(60)

    # fade in/out
    for i in range(0, 4 * 256, 8):
        for j in range(n):
            if (i // 256) % 2 == 0:
                val = i & 0xff
            else:
                val = 255 - (i & 0xff)
            np[j] = (val, 0, 0)
        np.write()

    # clear
    for i in range(n):
        np[i] = (0, 0, 0)
    np.write()
```

Execute it using:

```
>>> demo(np)
```

Temperature and Humidity

DHT (Digital Humidity & Temperature) sensors are low cost digital sensors with capacitive humidity sensors and thermistors to measure the surrounding air. They feature a chip that handles analog to digital conversion and provide a 1-wire interface. Newer sensors additionally provide an I2C interface.

The DHT11 (blue) and DHT22 (white) sensors provide the same 1-wire interface, however, the DHT22 requires a separate object as it has more complex calculation. DHT22 have 1 decimal place resolution for both humidity and temperature readings. DHT11 have whole number for both.

A custom 1-wire protocol, which is different to Dallas 1-wire, is used to get the measurements from the sensor. The payload consists of a humidity value, a temperature value and a checksum.

To use the 1-wire interface, construct the objects referring to their data pin:

```
>>> import dht
>>> import machine
>>> d = dht.DHT11(machine.Pin(4))

>>> import dht
>>> import machine
>>> d = dht.DHT22(machine.Pin(4))
```

Then measure and read their values with:

```
>>> d.measure()
>>> d.temperature()
>>> d.humidity()
```

Values returned from `temperature()` are in degrees Celsius and values returned from `humidity()` are a percentage of relative humidity.

The DHT11 can be called no more than once per second and the DHT22 once every two seconds for most accurate results. Sensor accuracy will degrade over time. Each sensor supports a different operating range. Refer to the product datasheets for specifics.

In 1-wire mode, only three of the four pins are used and in I2C mode, all four pins are used. Older sensors may still have 4 pins even though they do not support I2C. The 3rd pin is simply not connected.

Pin configurations:

Sensor without I2C in 1-wire mode (eg. DHT11, DHT22, AM2301, AM2302):

> 1=VDD, 2=Data, 3=NC, 4=GND

Sensor with I2C in 1-wire mode (eg. DHT12, AM2320, AM2321, AM2322):

> 1=VDD, 2=Data, 3=GND, 4=GND

Sensor with I2C in I2C mode (eg. DHT12, AM2320, AM2321, AM2322):

> 1=VDD, 2=SDA, 3=GND, 4=SCL

You should use pull-up resistors for the Data, SDA and SCL pins.

To make newer I2C sensors work in backwards compatible 1-wire mode, you must connect both pins 3 and 4 to GND. This disables the I2C interface.

DHT22 sensors are now sold under the name AM2302 and are otherwise identical.

Next steps

That brings us to the end of the tutorial! Hopefully by now you have a good feel for the capabilities of MicroPython on the ESP8266 and understand how to control both the WiFi and IO aspects of the chip.

There are many features that were not covered in this tutorial. The best way to learn about them is to read the full documentation of the modules, and to experiment!

Good luck creating your Internet of Things devices!

CHAPTER 6

MicroPython libraries

Warning: Important summary of this section

- MicroPython implements a subset of Python functionality for each module.
- To ease extensibility, MicroPython versions of standard Python modules usually have `u` (micro) prefix.
- Any particular MicroPython variant or port may miss any feature/function described in this general documentation, due to resource constraints.

This chapter describes modules (function and class libraries) which are built into MicroPython and CircuitPython. There are a few categories of modules:

- Modules which implement a subset of standard Python functionality and are not intended to be extended by the user.
- Modules which implement a subset of Python functionality, with a provision for extension by the user (via Python code).
- Modules which implement MicroPython extensions to the Python standard libraries.
- Modules specific to a particular port and thus not portable.

Note about the availability of modules and their contents: This documentation in general aspires to describe all modules and functions/classes which are implemented in MicroPython. However, MicroPython is highly configurable, and each port to a particular board/embedded system makes available only a subset of MicroPython libraries. For officially supported ports, there is an effort to either filter out non-applicable items, or mark individual descriptions with "Availability:" clauses describing which ports provide a given feature. With that in mind, please still be warned that some functions/classes in a module (or even the entire module) described in this documentation may be unavailable in a particular build of MicroPython on a particular board. The best place to find general information of the availability/non-availability of a particular feature is the "General Information" section which contains information pertaining to a specific port.

Beyond the built-in libraries described in this documentation, many more modules from the Python standard library, as well as further MicroPython extensions to it, can be found in the micropython-lib repository.

Python standard libraries and micro-libraries

The following standard Python libraries have been "micro-ified" to fit in with the philosophy of MicroPython. They provide the core functionality of that module and are intended to be a drop-in replacement for the standard Python library. Some modules below use a standard Python name, but prefixed with "u", e.g. `ujson` instead of `json`. This is to signify that such a module is micro-library, i.e. implements only a subset of CPython module functionality. By naming them differently, a user has a choice to write a Python-level module to extend functionality for better compatibility with CPython (indeed, this is what done by micropython-lib project mentioned above).

On some embedded platforms, where it may be cumbersome to add Python-level wrapper modules to achieve naming compatibility with CPython, micro-modules are available both by their u-name, and also by their non-u-name. The non-u-name can be overridden by a file of that name in your package path. For example, `import json` will first search for a file `json.py` or directory `json` and load that package if it is found. If nothing is found, it will fallback to loading the built-in `ujson` module.

MicroPython-specific libraries

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

`btree` – simple BTree database

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 section, etc.
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"

# 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 streamw 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

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

`machine` — functions related to the hardware

The `machine` module contains specific functions related to the hardware on a particular board. Most functions in this module allow to achieve direct and unrestricted access to and control of hardware blocks on a system (like CPU, timers, buses, etc.). Used incorrectly, this can lead to malfunction, lockups, crashes of your board, and in extreme cases, hardware damage. A note of callbacks used by functions and class methods of `machine` module: all these callbacks should be considered as executing in an interrupt context. This is true for both physical devices with IDs >= 0 and "virtual" devices with negative IDs like -1 (these "virtual" devices are still thin shims on top of real hardware and real hardware interrupts). See isr_rules.

Reset related functions

`machine.`**`reset`**`()`
Resets the device in a manner similar to pushing the external RESET button.

`machine.`**`reset_cause`**`()`
Get the reset cause. See *constants* for the possible return values.

Interrupt related functions

machine.**disable_irq**()
: Disable interrupt requests. Returns the previous IRQ state which should be considered an opaque value. This return value should be passed to the `enable_irq` function to restore interrupts to their original state, before `disable_irq` was called.

machine.**enable_irq**(*state*)
: Re-enable interrupt requests. The `state` parameter should be the value that was returned from the most recent call to the `disable_irq` function.

Power related functions

machine.**freq**()
: Returns CPU frequency in hertz.

machine.**idle**()
: Gates the clock to the CPU, useful to reduce power consumption at any time during short or long periods. Peripherals continue working and execution resumes as soon as any interrupt is triggered (on many ports this includes system timer interrupt occurring at regular intervals on the order of millisecond).

machine.**sleep**()
: Stops the CPU and disables all peripherals except for WLAN. Execution is resumed from the point where the sleep was requested. For wake up to actually happen, wake sources should be configured first.

machine.**deepsleep**()
: Stops the CPU and all peripherals (including networking interfaces, if any). Execution is resumed from the main script, just as with a reset. The reset cause can be checked to know that we are coming from `machine.DEEPSLEEP`. For wake up to actually happen, wake sources should be configured first, like `Pin` change or `RTC` timeout.

Miscellaneous functions

machine.**unique_id**()
: Returns a byte string with a unique identifier of a board/SoC. It will vary from a board/SoC instance to another, if underlying hardware allows. Length varies by hardware (so use substring of a full value if you expect a short ID). In some MicroPython ports, ID corresponds to the network MAC address.

machine.**time_pulse_us**(*pin*, *pulse_level*, *timeout_us=1000000*)
: Time a pulse on the given *pin*, and return the duration of the pulse in microseconds. The `pulse_level` argument should be 0 to time a low pulse or 1 to time a high pulse.

  If the current input value of the pin is different to `pulse_level`, the function first (*) waits until the pin input becomes equal to `pulse_level`, then (**) times the duration that the pin is equal to `pulse_level`. If the pin is already equal to `pulse_level` then timing starts straight away.

  The function will return -2 if there was timeout waiting for condition marked (*) above, and -1 if there was timeout during the main measurement, marked (**) above. The timeout is the same for both cases and given by `timeout_us` (which is in microseconds).

Constants

machine.**IDLE**
machine.**SLEEP**

machine.**DEEPSLEEP**
IRQ wake values.

machine.**PWRON_RESET**
machine.**HARD_RESET**
machine.**WDT_RESET**
machine.**DEEPSLEEP_RESET**
machine.**SOFT_RESET**
Reset causes.

machine.**WLAN_WAKE**
machine.**PIN_WAKE**
machine.**RTC_WAKE**
Wake-up reasons.

Classes

class Pin – control I/O pins

A pin object is used to control I/O pins (also known as GPIO - general-purpose input/output). Pin objects are commonly associated with a physical pin that can drive an output voltage and read input voltages. The pin class has methods to set the mode of the pin (IN, OUT, etc) and methods to get and set the digital logic level. For analog control of a pin, see the *ADC* class.

A pin object is constructed by using an identifier which unambiguously specifies a certain I/O pin. The allowed forms of the identifier and the physical pin that the identifier maps to are port-specific. Possibilities for the identifier are an integer, a string or a tuple with port and pin number.

Usage Model:

```
from machine import Pin

# create an output pin on pin #0
p0 = Pin(0, Pin.OUT)

# set the value low then high
p0.value(0)
p0.value(1)

# create an input pin on pin #2, with a pull up resistor
p2 = Pin(2, Pin.IN, Pin.PULL_UP)

# read and print the pin value
print(p2.value())

# reconfigure pin #0 in input mode
p0.mode(p0.IN)

# configure an irq callback
p0.irq(lambda p:print(p))
```

Constructors

class machine.**Pin**(*id*, *mode=-1*, *pull=-1*, ***, *value*, *drive*, *alt*)
Access the pin peripheral (GPIO pin) associated with the given `id`. If additional arguments are given in the

constructor then they are used to initialise the pin. Any settings that are not specified will remain in their previous state.

The arguments are:

- `id` is mandatory and can be an arbitrary object. Among possible value types are: int (an internal Pin identifier), str (a Pin name), and tuple (pair of [port, pin]).
- `mode` specifies the pin mode, which can be one of:
  - `Pin.IN` - Pin is configured for input. If viewed as an output the pin is in high-impedance state.
  - `Pin.OUT` - Pin is configured for (normal) output.
  - `Pin.OPEN_DRAIN` - Pin is configured for open-drain output. Open-drain output works in the following way: if the output value is set to 0 the pin is active at a low level; if the output value is 1 the pin is in a high-impedance state. Not all ports implement this mode, or some might only on certain pins.
  - `Pin.ALT` - Pin is configured to perform an alternative function, which is port specific. For a pin configured in such a way any other Pin methods (except *Pin.init()*) are not applicable (calling them will lead to undefined, or a hardware-specific, result). Not all ports implement this mode.
  - `Pin.ALT_OPEN_DRAIN` - The Same as `Pin.ALT`, but the pin is configured as open-drain. Not all ports implement this mode.
- `pull` specifies if the pin has a (weak) pull resistor attached, and can be one of:
  - `None` - No pull up or down resistor.
  - `Pin.PULL_UP` - Pull up resistor enabled.
  - `Pin.PULL_DOWN` - Pull down resistor enabled.
- `value` is valid only for Pin.OUT and Pin.OPEN_DRAIN modes and specifies initial output pin value if given, otherwise the state of the pin peripheral remains unchanged.
- `drive` specifies the output power of the pin and can be one of: `Pin.LOW_POWER`, `Pin.MED_POWER` or `Pin.HIGH_POWER`. The actual current driving capabilities are port dependent. Not all ports implement this argument.
- `alt` specifies an alternate function for the pin and the values it can take are port dependent. This argument is valid only for `Pin.ALT` and `Pin.ALT_OPEN_DRAIN` modes. It may be used when a pin supports more than one alternate function. If only one pin alternate function is supported the this argument is not required. Not all ports implement this argument.

As specified above, the Pin class allows to set an alternate function for a particular pin, but it does not specify any further operations on such a pin. Pins configured in alternate-function mode are usually not used as GPIO but are instead driven by other hardware peripherals. The only operation supported on such a pin is re-initialising, by calling the constructor or *Pin.init()* method. If a pin that is configured in alternate-function mode is re-initialised with `Pin.IN`, `Pin.OUT`, or `Pin.OPEN_DRAIN`, the alternate function will be removed from the pin.

Methods

`Pin.`**`init`**(*mode=-1*, *pull=-1*, ***, *value*, *drive*, *alt*)

Re-initialise the pin using the given parameters. Only those arguments that are specified will be set. The rest of the pin peripheral state will remain unchanged. See the constructor documentation for details of the arguments.

Returns `None`.

Pin.**value**([*x*])
This method allows to set and get the value of the pin, depending on whether the argument `x` is supplied or not.

If the argument is omitted then this method gets the digital logic level of the pin, returning 0 or 1 corresponding to low and high voltage signals respectively. The behaviour of this method depends on the mode of the pin:

- `Pin.IN` - The method returns the actual input value currently present on the pin.
- `Pin.OUT` - The behaviour and return value of the method is undefined.
- `Pin.OPEN_DRAIN` - If the pin is in state '0' then the behaviour and return value of the method is undefined. Otherwise, if the pin is in state '1', the method returns the actual input value currently present on the pin.

If the argument is supplied then this method sets the digital logic level of the pin. The argument `x` can be anything that converts to a boolean. If it converts to `True`, the pin is set to state '1', otherwise it is set to state '0'. The behaviour of this method depends on the mode of the pin:

- `Pin.IN` - The value is stored in the output buffer for the pin. The pin state does not change, it remains in the high-impedance state. The stored value will become active on the pin as soon as it is changed to `Pin.OUT` or `Pin.OPEN_DRAIN` mode.
- `Pin.OUT` - The output buffer is set to the given value immediately.
- `Pin.OPEN_DRAIN` - If the value is '0' the pin is set to a low voltage state. Otherwise the pin is set to high-impedance state.

When setting the value this method returns `None`.

Pin.**__call__**([*x*])
Pin objects are callable. The call method provides a (fast) shortcut to set and get the value of the pin. It is equivalent to Pin.value([x]). See *Pin.value()* for more details.

Pin.**on**()
Set pin to "1" output level.

Pin.**off**()
Set pin to "0" output level.

Pin.**mode**([*mode*])
Get or set the pin mode. See the constructor documentation for details of the `mode` argument.

Pin.**pull**([*pull*])
Get or set the pin pull state. See the constructor documentation for details of the `pull` argument.

Pin.**drive**([*drive*])
Get or set the pin drive strength. See the constructor documentation for details of the `drive` argument.

Not all ports implement this method.

Availability: WiPy.

Pin.**irq**(*handler=None*, *trigger=(Pin.IRQ_FALLING | Pin.IRQ_RISING)*, *, *priority=1*, *wake=None*)
Configure an interrupt handler to be called when the trigger source of the pin is active. If the pin mode is `Pin.IN` then the trigger source is the external value on the pin. If the pin mode is `Pin.OUT` then the trigger source is the output buffer of the pin. Otherwise, if the pin mode is `Pin.OPEN_DRAIN` then the trigger source is the output buffer for state '0' and the external pin value for state '1'.

The arguments are:

- `handler` is an optional function to be called when the interrupt triggers.
- `trigger` configures the event which can generate an interrupt. Possible values are:
  - `Pin.IRQ_FALLING` interrupt on falling edge.

– `Pin.IRQ_RISING` interrupt on rising edge.

– `Pin.IRQ_LOW_LEVEL` interrupt on low level.

– `Pin.IRQ_HIGH_LEVEL` interrupt on high level.

These values can be OR'ed together to trigger on multiple events.

- `priority` sets the priority level of the interrupt. The values it can take are port-specific, but higher values always represent higher priorities.
- `wake` selects the power mode in which this interrupt can wake up the system. It can be `machine.IDLE`, `machine.SLEEP` or `machine.DEEPSLEEP`. These values can also be OR'ed together to make a pin generate interrupts in more than one power mode.

This method returns a callback object.

Constants

The following constants are used to configure the pin objects. Note that not all constants are available on all ports.

`Pin.`**`IN`**
`Pin.`**`OUT`**
`Pin.`**`OPEN_DRAIN`**
`Pin.`**`ALT`**
`Pin.`**`ALT_OPEN_DRAIN`**
Selects the pin mode.

`Pin.`**`PULL_UP`**
`Pin.`**`PULL_DOWN`**
Selects whether there is a pull up/down resistor. Use the value `None` for no pull.

`Pin.`**`LOW_POWER`**
`Pin.`**`MED_POWER`**
`Pin.`**`HIGH_POWER`**
Selects the pin drive strength.

`Pin.`**`IRQ_FALLING`**
`Pin.`**`IRQ_RISING`**
`Pin.`**`IRQ_LOW_LEVEL`**
`Pin.`**`IRQ_HIGH_LEVEL`**
Selects the IRQ trigger type.

class Signal – control and sense external I/O devices

The Signal class is a simple extension of Pin class. Unlike Pin, which can be only in "absolute" 0 and 1 states, a Signal can be in "asserted" (on) or "deasserted" (off) states, while being inverted (active-low) or not. Summing up, it adds logical inversion support to Pin functionality. While this may seem a simple addition, it is exactly what is needed to support wide array of simple digital devices in a way portable across different boards, which is one of the major MicroPython goals. Regardless whether different users have an active-high or active-low LED, a normally open or normally closed relay - you can develop single, nicely looking application which works with each of them, and capture hardware configuration differences in few lines on the config file of your app.

Following is the guide when Signal vs Pin should be used:

- Use Signal: If you want to control a simple on/off (including software PWM!) devices like LEDs, multi-segment indicators, relays, buzzers, or read simple binary sensors, like normally open or normally closed buttons, pulled

high or low, Reed switches, moisture/flame detectors, etc. etc. Summing up, if you have a real physical device/sensor requiring GPIO access, you likely should use a Signal.

- Use Pin: If you implement a higher-level protocol or bus to communicate with more complex devices.

The split between Pin and Signal come from the usecases above and the architecture of MicroPython: Pin offers the lowest overhead, which may be important when bit-banging protocols. But Signal adds additional flexibility on top of Pin, at the cost of minor overhead (much smaller than if you implemented active-high vs active-low device differences in Python manually!). Also, Pin is low-level object which needs to be implemented for each support board, while Signal is a high-level object which comes for free once Pin is implemented.

If in doubt, give the Signal a try! Once again, it is developed to save developers from the need to handle unexciting differences like active-low vs active-high signals, and allow other users to share and enjoy your application, instead of being frustrated by the fact that it doesn't work for them simply because their LEDs or relays are wired in a slightly different way.

Constructors

class `machine.`**`Signal`**(*pin_obj*, *invert=False*)
class `machine.`**`Signal`**(*pin_arguments...*, *, *invert=False*)

Create a Signal object. There're two ways to create it:

- By wrapping existing Pin object - universal method which works for any board.
- By passing required Pin parameters directly to Signal constructor, skipping the need to create intermediate Pin object. Available on many, but not all boards.

The arguments are:

- `pin_obj` is existing Pin object.
- `pin_arguments` are the same arguments as can be passed to Pin constructor.
- `invert` - if True, the signal will be inverted (active low).

Methods

`Signal.`**`value`**([*x*])

This method allows to set and get the value of the signal, depending on whether the argument `x` is supplied or not.

If the argument is omitted then this method gets the signal level, 1 meaning signal is asserted (active) and 0 - signal inactive.

If the argument is supplied then this method sets the signal level. The argument `x` can be anything that converts to a boolean. If it converts to `True`, the signal is active, otherwise it is inactive.

Correspondence between signal being active and actual logic level on the underlying pin depends on whether signal is inverted (active-low) or not. For non-inverted signal, active status corresponds to logical 1, inactive - to logical 0. For inverted/active-low signal, active status corresponds to logical 0, while inactive - to logical 1.

`Signal.`**`on`**()

Activate signal.

`Signal.`**`off`**()

Deactivate signal.

class UART – duplex serial communication bus

UART implements the standard UART/USART duplex serial communications protocol. At the physical level it consists of 2 lines: RX and TX. The unit of communication is a character (not to be confused with a string character) which can be 8 or 9 bits wide.

UART objects can be created and initialised using:

```
from machine import UART

uart = UART(1, 9600)                         # init with given baudrate
uart.init(9600, bits=8, parity=None, stop=1) # init with given parameters
```

Supported parameters differ on a board:

Pyboard: Bits can be 7, 8 or 9. Stop can be 1 or 2. With `parity=None`, only 8 and 9 bits are supported. With parity enabled, only 7 and 8 bits are supported.

WiPy/CC3200: Bits can be 5, 6, 7, 8. Stop can be 1 or 2.

A UART object acts like a stream object and reading and writing is done using the standard stream methods:

```
uart.read(10)       # read 10 characters, returns a bytes object
uart.read()         # read all available characters
uart.readline()     # read a line
uart.readinto(buf)  # read and store into the given buffer
uart.write('abc')   # write the 3 characters
```

Constructors

class `machine.`**`UART`**(*id*, ...)
Construct a UART object of the given id.

Methods

`UART.`**`deinit`**()
Turn off the UART bus.

`UART.`**`any`**()
Returns an integer counting the number of characters that can be read without blocking. It will return 0 if there are no characters available and a positive number if there are characters. The method may return 1 even if there is more than one character available for reading.

For more sophisticated querying of available characters use select.poll:

```
poll = select.poll()
poll.register(uart, select.POLLIN)
poll.poll(timeout)
```

`UART.`**`read`**([*nbytes*])
Read characters. If `nbytes` is specified then read at most that many bytes, otherwise read as much data as possible.

Return value: a bytes object containing the bytes read in. Returns `None` on timeout.

UART.**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.

Return value: number of bytes read and stored into `buf` or `None` on timeout.

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

Return value: the line read or `None` on timeout.

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

Return value: number of bytes written or `None` on timeout.

UART.**sendbreak**()
Send a break condition on the bus. This drives the bus low for a duration longer than required for a normal transmission of a character.

class SPI – a Serial Peripheral Interface bus protocol (master side)

SPI is a synchronous serial protocol that is driven by a master. At the physical level, a bus consists of 3 lines: SCK, MOSI, MISO. Multiple devices can share the same bus. Each device should have a separate, 4th signal, SS (Slave Select), to select a particular device on a bus with which communication takes place. Management of an SS signal should happen in user code (via machine.Pin class).

Constructors

class machine.**SPI**(*id*, ...)
Construct an SPI object on the given bus, `id`. Values of `id` depend on a particular port and its hardware. Values 0, 1, etc. are commonly used to select hardware SPI block #0, #1, etc. Value -1 can be used for bitbanging (software) implementation of SPI (if supported by a port).

With no additional parameters, the SPI object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See `init` for parameters of initialisation.

Methods

SPI.**init**(*baudrate=1000000*, ***, *polarity=0*, *phase=0*, *bits=8*, *firstbit=SPI.MSB*, *sck=None*, *mosi=None*, *miso=None*, *pins=(SCK*, *MOSI*, *MISO)*)
Initialise the SPI bus with the given parameters:

- `baudrate` is the SCK clock rate.
- `polarity` can be 0 or 1, and is the level the idle clock line sits at.
- `phase` can be 0 or 1 to sample data on the first or second clock edge respectively.
- `bits` is the width in bits of each transfer. Only 8 is guaranteed to be supported by all hardware.
- `firstbit` can be `SPI.MSB` or `SPI.LSB`.
- `sck`, `mosi`, `miso` are pins (machine.Pin) objects to use for bus signals. For most hardware SPI blocks (as selected by `id` parameter to the constructor), pins are fixed and cannot be changed. In some cases, hardware blocks allow 2-3 alternative pin sets for a hardware SPI block. Arbitrary pin assignments are possible only for a bitbanging SPI driver (`id` = -1).

- `pins` - WiPy port doesn't `sck`, `mosi`, `miso` arguments, and instead allows to specify them as a tuple of `pins` parameter.

`SPI.`**`deinit`**`()`
Turn off the SPI bus.

`SPI.`**`read`**`(`*nbytes*, *write=0x00*`)`
Read a number of bytes specified by `nbytes` while continuously writing the single byte given by `write`. Returns a `bytes` object with the data that was read.

`SPI.`**`readinto`**`(`*buf*, *write=0x00*`)`
Read into the buffer specified by `buf` while continuously writing the single byte given by `write`. Returns `None`.

Note: on WiPy this function returns the number of bytes read.

`SPI.`**`write`**`(`*buf*`)`
Write the bytes contained in `buf`. Returns `None`.

Note: on WiPy this function returns the number of bytes written.

`SPI.`**`write_readinto`**`(`*write_buf*, *read_buf*`)`
Write the bytes from `write_buf` while reading into `read_buf`. The buffers can be the same or different, but both buffers must have the same length. Returns `None`.

Note: on WiPy this function returns the number of bytes written.

Constants

`SPI.`**`MASTER`**
for initialising the SPI bus to master; this is only used for the WiPy

`SPI.`**`MSB`**
set the first bit to be the most significant bit

`SPI.`**`LSB`**
set the first bit to be the least significant bit

class I2C – a two-wire serial protocol

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

I2C objects are created attached to a specific bus. They can be initialised when created, or initialised later on.

Printing the I2C object gives you information about its configuration.

Example usage:

```
from machine import I2C

i2c = I2C(freq=400000)          # create I2C peripheral at frequency of 400kHz
                                # depending on the port, extra parameters may be␣
↪required
                                # to select the peripheral and/or pins to use

i2c.scan()                      # scan for slaves, returning a list of 7-bit addresses

i2c.writeto(42, b'123')         # write 3 bytes to slave with 7-bit address 42
```

```
i2c.readfrom(42, 4)              # read 4 bytes from slave with 7-bit address 42

i2c.readfrom_mem(42, 8, 3)       # read 3 bytes from memory of slave 42,
                                 #   starting at memory-address 8 in the slave
i2c.writeto_mem(42, 2, b'\x10') # write 1 byte to memory of slave 42
                                 #   starting at address 2 in the slave
```

Constructors

class machine.**I2C**(*id=-1*, ***, *scl*, *sda*, *freq=400000*)
Construct and return a new I2C object using the following parameters:

- *id* identifies the particular I2C peripheral. The default value of -1 selects a software implementation of I2C which can work (in most cases) with arbitrary pins for SCL and SDA. If *id* is -1 then scl and sda must be specified. Other allowed values for *id* depend on the particular port/board, and specifying scl and sda may or may not be required or allowed in this case.
- scl should be a pin object specifying the pin to use for SCL.
- sda should be a pin object specifying the pin to use for SDA.
- *freq* should be an integer which sets the maximum frequency for SCL.

General Methods

I2C.**init**(*scl*, *sda*, ***, *freq=400000*)
Initialise the I2C bus with the given arguments:

- scl is a pin object for the SCL line
- sda is a pin object for the SDA line
- *freq* is the SCL clock rate

I2C.**deinit**()
Turn off the I2C bus.

Availability: WiPy.

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

Primitive I2C operations

The following methods implement the primitive I2C master bus operations and can be combined to make any I2C transaction. They are provided if you need more control over the bus, otherwise the standard methods (see below) can be used.

I2C.**start**()
Generate a START condition on the bus (SDA transitions to low while SCL is high).

Availability: ESP8266.

I2C.**stop**()
Generate a STOP condition on the bus (SDA transitions to high while SCL is high).

Availability: ESP8266.

I2C.**readinto**(*buf*, *nack=True*)
Reads bytes from the bus and stores them into `buf`. The number of bytes read is the length of `buf`. An ACK will be sent on the bus after receiving all but the last byte. After the last byte is received, if `nack` is true then a NACK will be sent, otherwise an ACK will be sent (and in this case the slave assumes more bytes are going to be read in a later call).

Availability: ESP8266.

I2C.**write**(*buf*)
Write the bytes from `buf` to the bus. Checks that an ACK is received after each byte and stops transmitting the remaining bytes if a NACK is received. The function returns the number of ACKs that were received.

Availability: ESP8266.

Standard bus operations

The following methods implement the standard I2C master read and write operations that target a given slave device.

I2C.**readfrom**(*addr*, *nbytes*, *stop=True*)
Read `nbytes` from the slave specified by `addr`. If *`stop`* is true then a STOP condition is generated at the end of the transfer. Returns a *`bytes`* object with the data read.

I2C.**readfrom_into**(*addr*, *buf*, *stop=True*)
Read into `buf` from the slave specified by `addr`. The number of bytes read will be the length of `buf`. If *`stop`* is true then a STOP condition is generated at the end of the transfer.

The method returns `None`.

I2C.**writeto**(*addr*, *buf*, *stop=True*)
Write the bytes from `buf` to the slave specified by `addr`. If a NACK is received following the write of a byte from `buf` then the remaining bytes are not sent. If *`stop`* is true then a STOP condition is generated at the end of the transfer, even if a NACK is received. The function returns the number of ACKs that were received.

Memory operations

Some I2C devices act as a memory device (or set of registers) that can be read from and written to. In this case there are two addresses associated with an I2C transaction: the slave address and the memory address. The following methods are convenience functions to communicate with such devices.

I2C.**readfrom_mem**(*addr*, *memaddr*, *nbytes*, ***, *addrsize=8*)
Read `nbytes` from the slave specified by `addr` starting from the memory address specified by `memaddr`. The argument `addrsize` specifies the address size in bits. Returns a *`bytes`* object with the data read.

I2C.**readfrom_mem_into**(*addr*, *memaddr*, *buf*, ***, *addrsize=8*)
Read into `buf` from the slave specified by `addr` starting from the memory address specified by `memaddr`. The number of bytes read is the length of `buf`. The argument `addrsize` specifies the address size in bits (on ESP8266 this argument is not recognised and the address size is always 8 bits).

The method returns `None`.

I2C.**writeto_mem**(*addr*, *memaddr*, *buf*, ***, *addrsize=8*)
Write `buf` to the slave specified by `addr` starting from the memory address specified by `memaddr`. The argument `addrsize` specifies the address size in bits (on ESP8266 this argument is not recognised and the address size is always 8 bits).

The method returns `None`.

class RTC – real time clock

The RTC is and independent clock that keeps track of the date and time.

Example usage:

```
rtc = machine.RTC()
rtc.init((2014, 5, 1, 4, 13, 0, 0, 0))
print(rtc.now())
```

Constructors

class machine.**RTC**(*id=0*, *...*)
Create an RTC object. See init for parameters of initialization.

Methods

RTC.**init**(*datetime*)
Initialise the RTC. Datetime is a tuple of the form:

```
(year, month, day[, hour[, minute[, second[, microsecond[,
tzinfo]]]]])
```

RTC.**now**()
Get get the current datetime tuple.

RTC.**deinit**()
Resets the RTC to the time of January 1, 2015 and starts running it again.

RTC.**alarm**(*id*, *time*, */**, *repeat=False*)
Set the RTC alarm. Time might be either a millisecond value to program the alarm to current time + time_in_ms in the future, or a datetimetuple. If the time passed is in milliseconds, repeat can be set to `True` to make the alarm periodic.

RTC.**alarm_left**(*alarm_id=0*)
Get the number of milliseconds left before the alarm expires.

RTC.**cancel**(*alarm_id=0*)
Cancel a running alarm.

RTC.**irq**(***, *trigger*, *handler=None*, *wake=machine.IDLE*)
Create an irq object triggered by a real time clock alarm.

- `trigger` must be `RTC.ALARM0`
- `handler` is the function to be called when the callback is triggered.
- `wake` specifies the sleep mode from where this interrupt can wake up the system.

Constants

RTC.**ALARM0**
irq trigger source

class Timer – control hardware timers

Hardware timers deal with timing of periods and events. Timers are perhaps the most flexible and heterogeneous kind of hardware in MCUs and SoCs, differently greatly from a model to a model. MicroPython's Timer class defines a baseline operation of executing a callback with a given period (or once after some delay), and allow specific boards to define more non-standard behavior (which thus won't be portable to other boards).

See discussion of *important constraints* on Timer callbacks.

Note: Memory can't be allocated inside irq handlers (an interrupt) and so exceptions raised within a handler don't give much information. See `micropython.alloc_emergency_exception_buf()` for how to get around this limitation.

Constructors

class `machine.`**`Timer`**(*id*, ...)
Construct a new timer object of the given id. Id of -1 constructs a virtual timer (if supported by a board).

Methods

`Timer.`**`deinit`**()
Deinitialises the timer. Stops the timer, and disables the timer peripheral.

Constants

`Timer.`**`ONE_SHOT`**

`Timer.`**`PERIODIC`**
Timer operating mode.

class WDT – watchdog timer

The WDT is used to restart the system when the application crashes and ends up into a non recoverable state. Once started it cannot be stopped or reconfigured in any way. After enabling, the application must "feed" the watchdog periodically to prevent it from expiring and resetting the system.

Example usage:

```
from machine import WDT
wdt = WDT(timeout=2000)  # enable it with a timeout of 2s
wdt.feed()
```

Availability of this class: pyboard, WiPy.

Constructors

class `machine.`**`WDT`**(*id=0*, *timeout=5000*)
Create a WDT object and start it. The timeout must be given in seconds and the minimum value that is accepted is 1 second. Once it is running the timeout cannot be changed and the WDT cannot be stopped either.

Methods

wdt.**feed**()
Feed the WDT to prevent it from resetting the system. The application should place this call in a sensible place ensuring that the WDT is only fed after verifying that everything is functioning correctly.

micropython – access and control MicroPython internals

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 – access and control MicroPython internals* 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**(*fun*, *arg*)
Schedule the function fun 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

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'))
```

`uctypes` – access binary data in a structured way

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

See also:

Module *`ustruct`* 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).

CHAPTER 7

Adafruit CircuitPython

This is an open source derivative of MicroPython for use on educational development boards designed and sold by Adafruit.

As a MicroPython derivative, this implements Python 3.x on microcontrollers such as the SAMD21 and ESP8266.

Project Status

This project is in beta. Most APIs should be stable going forward.

Supported boards

Designed for CircuitPython

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

Other

- Adafruit Feather HUZZAH
- Adafruit Feather M0 Basic
- Adafruit Feather M0 Bluefruit LE (uses M0 Basic binaries)
- Adafruit Feather M0 Adalogger (MicroSD card not supported yet.)
- Arduino Zero

Download

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

Documentation

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

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.

Differences from MicroPython

- Port for Atmel SAMD21 (Commonly known as M0 in product names.)
- No `machine` API on Atmel SAMD21 port.
- Only supports Atmel SAMD21 and ESP8266 ports.
- The order that files are run and the state thats shared between them. The goal is to clarify the role of each file and make them 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 its done the vm and hardware is reinitialized. **This means you cannot read state from code.py in the REPL anymore.** This was changed to reduce confusion about pins and memory being in use.
  - After code.py the REPL can be entered by pressing any key. It no longer shares state with code.py so its 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.
- Unified hardware APIs: `audioio`, `analogio`, `busio`, `digitalio`, `pulseio`, `touchio`, `microcontroller`, `board`, `bitbangio` (Only available on atmel-samd21 and ESP8266 currently.)
- Tracks MicroPython's releases (not master).
- No module aliasing. (`uos` and `utime` are not available as `os` and `time` respectively.) Instead `os`, `time`, and `random` are CPython compatible.
- New `storage` module which manages file system mounts. (Functionality from `uos` in MicroPython.)

- Modules with a CPython counterpart, such as `time`, `os` and `random`, are strict subsets of their CPython version. Therefore, code from CircuitPython is runnable on CPython but not necessarily the reverse.
- tick count is available as `time.monotonic()`
- atmel-samd21 features
  - RGB status LED
  - Auto-reload after file write over mass storage. (Disable with `samd.disable_autoreload()`)
  - Wait state after boot and main run, before REPL.
  - Main is one of these: `code.txt`, `code.py`, `main.py`, `main.txt`
  - Boot is one of these: `settings.txt`, `settings.py`, `boot.py`, `boot.txt`

Project Structure

Here is an overview of the top-level directories.

Core

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

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

Ports

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

- `atmel-samd` Support for SAMD21 based boards such as Arduino Zero, Adafruit Feather M0 Basic, and Adafruit Feather M0 Bluefruit LE.
- `bare-arm` A bare minimum version of MicroPython for ARM MCUs.
- `cc3200` Support for boards based CC3200 from TI such as the WiPy 1.0.

- `esp8266` Support for boards based on ESP8266 WiFi modules such as the Adafruit Feather HUZZAH.
- `minimal` A minimal MicroPython port. Start with this if you want to port MicroPython to another microcontroller.
- `pic16bit` Support for 16-bit PIC microcontrollers.
- `qemu-arm` Support for ARM emulation through QEMU.
- `stmhal` Support for boards based on STM32 microcontrollers including the MicroPython flagship PyBoard.
- `teensy` Support for the Teensy line of boards such as the Teensy 3.1.
- `unix` Support for UNIX.
- `windows` Support for Windows.
- `zephyr` Support for Zephyr, a real-time operating system by the Linux Foundation.

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

CHAPTER 8

Contributing

Please note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms. Participation covers any forum used to converse about CircuitPython including unofficial and official spaces. Failure to do so will result in corrective actions such as time out or ban from the project.

Developer contact

@tannewt is the main developer of CircuitPython and is sponsored by Adafruit Industries LLC. He is reachable on Discord as tannewt and Gitter as tannewt during US West Coast working hours. He also checks GitHub issues and the Adafruit support forum.

Licensing

By contributing to this repository you are certifying that you have all necessary permissions to license the code under an MIT License. You still retain the copyright but are granting many permissions under the MIT License.

If you have an employment contract with your employer please make sure that they don't automatically own your work product. Make sure to get any necessary approvals before contributing. Another term for this contribution off-hours is moonlighting.

Code guidelines

We aim to keep our code and commit style compatible with MicroPython upstream. Please review their code conventions to do so. Familiarity with their design philosophy is also useful though not always applicable to CircuitPython.

Furthermore, CircuitPython has a design guide that covers a variety of different topics. Please read it as well.

CHAPTER 9

Contributor Covenant Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at support@adafruit.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4

CHAPTER 10

MicroPython & CircuitPython license information

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

W

Index

Symbols

A

B

C

D

E

F

G

H

I

S

T

U

V

W