MicroPython Documentation

Release 1.17

Damien P. George, Paul Sokolovsky, and contributors

Jan 07, 2022

CONTENTS

CHAPTER

ONE

MICROPYTHON LIBRARIES

Warning: Important summary of this section

- MicroPython provides built-in modules that mirror the functionality of the Python standard library (e.g. *os*, *time*), as well as MicroPython-specific modules (e.g. *bluetooth*, *machine*).
- Most standard library modules implement a subset of the functionality of the equivalent Python module, and in a few cases provide some MicroPython-specific extensions (e.g. *array*, *os*)
- Due to resource constraints or other limitations, some ports or firmware versions may not include all the functionality documented here.
- To allow for extensibility, the built-in modules can be extended from Python code loaded onto the device.

This chapter describes modules (function and class libraries) which are built into MicroPython. This documentation in general aspires to describe all modules and functions/classes which are implemented in the MicroPython project. However, MicroPython is highly configurable, and each port to a particular board/embedded system may include only a subset of the available MicroPython libraries.

With that in mind, please 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 system. 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 *MicroPython port*.

On some ports you are able to discover the available, built-in libraries that can be imported by entering the following at the *REPL*:

```
help('modules')
```

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 *micropython-lib*.

1.1 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.

1.1.1 array – arrays of numeric data

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

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

Classes

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

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

append(*val*)

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

extend(*iterable*)

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

1.1.2 binascii – binary/ASCII conversions

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

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

Functions

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

Convert the bytes in the *data* object to a hexadecimal representation. Returns a bytes object.

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

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

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

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

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

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

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

1.1.3 `builtins` – builtin functions and exceptions

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

Functions and types

abs()

all()

any()

bin()

class bool

class bytearray

class bytes
See CPython documentation: `bytes`.

callable()

chr()

classmethod()

compile()

class complex

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

class dict

dir()

divmod()

enumerate()

eval()

exec()

filter()

class float

class frozenset

getattr()

globals()

hasattr()

hash()

hex()

id()

input()

class int

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

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

isinstance()

issubclass()

iter()

len()

class list

locals()

map()

max()

class memoryview

min()

next()

class object

oct()

open()

ord()

pow()

print()

property()

range()

repr()

reversed()

round()

class set

setattr()

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

sorted()

staticmethod()

class str

sum()

super()

class tuple

type()

zip()

Exceptions

exception AssertionError

exception AttributeError

exception Exception

exception ImportError

exception IndexError

exception KeyboardInterrupt

exception KeyError

exception MemoryError

exception NameError

exception NotImplementedError

exception OSError

exception RuntimeError

exception StopIteration

exception SyntaxError

exception SystemExit
See CPython documentation: `SystemExit`.

exception TypeError
See CPython documentation: `TypeError`.

exception ValueError

exception ZeroDivisionError

1.1.4 `cmath` – mathematical functions for complex numbers

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

The `cmath` module provides some basic mathematical functions for working with complex numbers.

Availability: not available on WiPy and ESP8266. Floating point support required for this module.

Functions

cmath.**cos**(*z*)
Return the cosine of z.

cmath.**exp**(*z*)
Return the exponential of z.

cmath.**log**(*z*)
Return the natural logarithm of z. The branch cut is along the negative real axis.

cmath.**log10**(*z*)
Return the base-10 logarithm of z. The branch cut is along the negative real axis.

cmath.**phase**(*z*)
Returns the phase of the number z, in the range (-pi, +pi].

cmath.**polar**(*z*)
Returns, as a tuple, the polar form of z.

cmath.**rect**(*r*, *phi*)
Returns the complex number with modulus r and phase phi.

cmath.**sin**(*z*)
Return the sine of z.

cmath.**sqrt**(*z*)
Return the square-root of z.

Constants

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

cmath.**pi**
the ratio of a circles circumference to its diameter

1.1.5 collections – collection and container types

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

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

Classes

collections.**deque**(*iterable*, *maxlen*[, *flags*])
Deques (double-ended queues) are a list-like container that support O(1) appends and pops from either side of the deque. New deques are created using the following arguments:

- *iterable* must be the empty tuple, and the new deque is created empty.
- *maxlen* must be specified and the deque will be bounded to this maximum length. Once the deque is full, any new items added will discard items from the opposite end.
- The optional *flags* can be 1 to check for overflow when adding items.

As well as supporting *bool* and *len*, deque objects have the following methods:

deque.**append**(*x*)
Add *x* to the right side of the deque. Raises IndexError if overflow checking is enabled and there is no more room left.

deque.**popleft**()
Remove and return an item from the left side of the deque. Raises IndexError if no items are present.

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

```
from collections import namedtuple

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

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

```
from collections import OrderedDict

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

Output:

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

1.1.6 errno – system error codes

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

This module provides access to symbolic error codes for *`OSError`* exception. A particular inventory of codes depends on *MicroPython port*.

Constants

EEXIST, EAGAIN, etc.
Error codes, based on ANSI C/POSIX standard. All error codes start with E. As mentioned above, inventory of the codes depends on *MicroPython port*. Errors are usually accessible as `exc.errno` where `exc` is an instance of *OSError*. Usage example:

```
try:
    os.mkdir("my_dir")
except OSError as exc:
    if exc.errno == errno.EEXIST:
        print("Directory already exists")
```

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

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

1.1.7 gc – control the garbage collector

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

Functions

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

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

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

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

Difference to CPython

This function is MicroPython extension.

---

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

Difference to CPython

This function is MicroPython extension.

---

gc.**threshold**([*amount*])
Set or query the additional GC allocation threshold. Normally, a collection is triggered only when a new allocation cannot be satisfied, i.e. on an out-of-memory (OOM) condition. If this function is called, in addition to

OOM, a collection will be triggered each time after *amount* bytes have been allocated (in total, since the previous time such an amount of bytes have been allocated). *amount* is usually specified as less than the full heap size, with the intention to trigger a collection earlier than when the heap becomes exhausted, and in the hope that an early collection will prevent excessive memory fragmentation. This is a heuristic measure, the effect of which will vary from application to application, as well as the optimal value of the *amount* parameter.

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

Difference to CPython

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

1.1.8 `hashlib` – hashing algorithms

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

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

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

Constructors

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

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

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

Methods

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

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

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

1.1.9 heapq – heap queue algorithm

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

This module implements the min heap queue algorithm.

A heap queue is essentially a list that has its elements stored in such a way that the first item of the list is always the smallest.

Functions

heapq.**heappush**(*heap*, *item*)
Push the item onto the heap.

heapq.**heappop**(*heap*)
Pop the first item from the heap, and return it. Raise IndexError if heap is empty.

The returned item will be the smallest item in the heap.

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

1.1.10 io – input/output streams

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

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

Conceptual hierarchy

Difference to CPython

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

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

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

But in CPython, another important dichotomy is tied with bufferedness - its whether a stream may incur short read/writes or not. A short read is when a user asks e.g. 10 bytes from a stream, but gets less, similarly for writes. In CPython, unbuffered streams are automatically short operation susceptible, while buffered are guarantee against them. The no short read/writes is an important trait, as it allows to develop more concise and efficient programs - something which is highly desirable for MicroPython. So, while MicroPython doesnt support buffered streams, it still provides for no-short-operations streams. Whether there will be short operations or not depends on each particular class needs, but developers are strongly advised to favour no-short-operations behaviour for the reasons stated above. For example,

MicroPython sockets are guaranteed to avoid short read/writes. Actually, at this time, there is no example of a short-operations stream class in the core, and one would be a port-specific class, where such a need is governed by hardware peculiarities.

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

The final dichotomy is binary vs text streams. MicroPython of course supports these, but while in CPython text streams are inherently buffered, they arent in MicroPython. (Indeed, thats one of the cases for which we may introduce buffering support.)

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

Functions

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

Classes

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

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

class `io.StringIO`([*string*])

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

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

class `io.StringIO`(*alloc_size*)

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

Difference to CPython

These constructors are a MicroPython extension.

1.1.11 `json` – JSON encoding and decoding

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

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

Functions

`json.`**`dump`**(*obj*, *stream*, *separators=None*)
Serialise *obj* to a JSON string, writing it to the given *stream*.

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

`json.`**`dumps`**(*obj*, *separators=None*)
Return *obj* represented as a JSON string.

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

`json.`**`load`**(*stream*)
Parse the given *stream*, interpreting it as a JSON string and deserialising the data to a Python object. The resulting object is returned.

Parsing continues until end-of-file is encountered. A *ValueError* is raised if the data in *stream* is not correctly formed.

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

1.1.12 `math` – mathematical functions

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

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

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

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

Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Constants

`math.e`
base of the natural logarithm

`math.pi`
the ratio of a circles circumference to its diameter

1.1.13 os – basic operating system services

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

The `os` module contains functions for filesystem access and mounting, terminal redirection and duplication, and the `uname` and `urandom` functions.

General functions

`os.uname()`
Return a tuple (possibly a named tuple) containing information about the underlying machine and/or its operating system. The tuple has five fields in the following order, each of them being a string:

- `sysname` – the name of the underlying system
- `nodename` – the network name (can be the same as `sysname`)
- `release` – the version of the underlying system
- `version` – the MicroPython version and build date
- `machine` – an identifier for the underlying hardware (eg board, CPU)

`os.urandom(`*n*`)`
Return a bytes object with *n* random bytes. Whenever possible, it is generated by the hardware random number generator.

Filesystem access

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

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

`os.ilistdir([`*dir*`])`
This function returns an iterator which then yields tuples corresponding to the entries in the directory that it is listing. With no argument it lists the current directory, otherwise it lists the directory given by *dir*.

The tuples have the form *(name, type, inode[, size])*:

- *name* is a string (or bytes if *dir* is a bytes object) and is the name of the entry;
- *type* is an integer that specifies the type of the entry, with 0x4000 for directories and 0x8000 for regular files;
- *inode* is an integer corresponding to the inode of the file, and may be 0 for filesystems that dont have such a notion.

- Some platforms may return a 4-tuple that includes the entrys *size*. For file entries, *size* is an integer representing the size of the file or -1 if unknown. Its meaning is currently undefined for directory entries.

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 unprivileged users
- `f_files` – number of inodes
- `f_ffree` – number of free inodes
- `f_favail` – number of free inodes for unprivileged users
- `f_flag` – mount flags
- `f_namemax` – maximum filename length

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

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

Terminal redirection and duplication

os.**dupterm**(*stream_object*, *index=0*, */*)
Duplicate or switch the MicroPython terminal (the REPL) on the given `stream`-like object. The *stream_object* argument must be a native stream object, or derive from `io.IOBase` and implement the `readinto()` and `write()` methods. The stream should be in non-blocking mode and `readinto()` should return `None` if there is no data available for reading.

After calling this function all terminal output is repeated on this stream, and any input that is available on the stream is passed on to the terminal input.

The *index* parameter should be a non-negative integer and specifies which duplication slot is set. A given port may implement more than one slot (slot 0 will always be available) and in that case terminal input and output is duplicated on all the slots that are set.

If `None` is passed as the *stream_object* then duplication is cancelled on the slot given by *index*.

The function returns the previous stream-like object in the given slot.

Filesystem mounting

Some ports provide a Virtual Filesystem (VFS) and the ability to mount multiple real filesystems within this VFS. Filesystem objects can be mounted at either the root of the VFS, or at a subdirectory that lives in the root. This allows dynamic and flexible configuration of the filesystem that is seen by Python programs. Ports that have this functionality provide the *mount()* and *umount()* functions, and possibly various filesystem implementations represented by VFS classes.

os.**mount**(*fsobj*, *mount_point*, *, *readonly*)

Mount the filesystem object *fsobj* at the location in the VFS given by the *mount_point* string. *fsobj* can be a a VFS object that has a `mount()` method, or a block device. If its a block device then the filesystem type is automatically detected (an exception is raised if no filesystem was recognised). *mount_point* may be `'/'` to mount *fsobj* at the root, or `'/<name>'` to mount it at a subdirectory under the root.

If *readonly* is `True` then the filesystem is mounted read-only.

During the mount process the method `mount()` is called on the filesystem object.

Will raise `OSError(EPERM)` if *mount_point* is already mounted.

os.**umount**(*mount_point*)

Unmount a filesystem. *mount_point* can be a string naming the mount location, or a previously-mounted filesystem object. During the unmount process the method `umount()` is called on the filesystem object.

Will raise `OSError(EINVAL)` if *mount_point* is not found.

class os.**VfsFat**(*block_dev*)

Create a filesystem object that uses the FAT filesystem format. Storage of the FAT filesystem is provided by *block_dev*. Objects created by this constructor can be mounted using *mount()*.

static mkfs(*block_dev*)

Build a FAT filesystem on *block_dev*.

class os.**VfsLfs1**(*block_dev*, *readsize=32*, *progsize=32*, *lookahead=32*)

Create a filesystem object that uses the littlefs v1 filesystem format. Storage of the littlefs filesystem is provided by *block_dev*, which must support the *extended interface*. Objects created by this constructor can be mounted using *mount()*.

See *Working with filesystems* for more information.

static mkfs(*block_dev*, *readsize=32*, *progsize=32*, *lookahead=32*)

Build a Lfs1 filesystem on *block_dev*.

Note: There are reports of littlefs v1 failing in certain situations, for details see littlefs issue 347.

class os.**VfsLfs2**(*block_dev*, *readsize=32*, *progsize=32*, *lookahead=32*, *mtime=True*)

Create a filesystem object that uses the littlefs v2 filesystem format. Storage of the littlefs filesystem is provided by *block_dev*, which must support the *extended interface*. Objects created by this constructor can be mounted using *mount()*.

The *mtime* argument enables modification timestamps for files, stored using littlefs attributes. This option can be disabled or enabled differently each mount time and timestamps will only be added or updated if *mtime* is enabled, otherwise the timestamps will remain untouched. Littlefs v2 filesystems without timestamps will work without reformatting and timestamps will be added transparently to existing files once they are opened for writing. When *mtime* is enabled `os.stat` on files without timestamps will return 0 for the timestamp.

See *Working with filesystems* for more information.

static mkfs(*block_dev*, *readsize=32*, *progsize=32*, *lookahead=32*)
Build a Lfs2 filesystem on *block_dev*.

Note: There are reports of littlefs v2 failing in certain situations, for details see littlefs issue 295.

Block devices

A block device is an object which implements the block protocol. This enables a device to support MicroPython filesystems. The physical hardware is represented by a user defined class. The `AbstractBlockDev` class is a template for the design of such a class: MicroPython does not actually provide that class, but an actual block device class must implement the methods described below.

A concrete implementation of this class will usually allow access to the memory-like functionality of a piece of hardware (like flash memory). A block device can be formatted to any supported filesystem and mounted using `os` methods.

See *Working with filesystems* for example implementations of block devices using the two variants of the block protocol described below.

Simple and extended interface

There are two compatible signatures for the `readblocks` and `writeblocks` methods (see below), in order to support a variety of use cases. A given block device may implement one form or the other, or both at the same time. The second form (with the offset parameter) is referred to as the extended interface.

Some filesystems (such as littlefs) that require more control over write operations, for example writing to sub-block regions without erasing, may require that the block device supports the extended interface.

class os.**AbstractBlockDev**(...)
Construct a block device object. The parameters to the constructor are dependent on the specific block device.

readblocks(*block_num*, *buf*)
readblocks(*block_num*, *buf*, *offset*)
The first form reads aligned, multiples of blocks. Starting at the block given by the index *block_num*, read blocks from the device into *buf* (an array of bytes). The number of blocks to read is given by the length of *buf*, which will be a multiple of the block size.

The second form allows reading at arbitrary locations within a block, and arbitrary lengths. Starting at block index *block_num*, and byte offset within that block of *offset*, read bytes from the device into *buf* (an array of bytes). The number of bytes to read is given by the length of *buf*.

writeblocks(*block_num*, *buf*)
writeblocks(*block_num*, *buf*, *offset*)
The first form writes aligned, multiples of blocks, and requires that the blocks that are written to be first erased (if necessary) by this method. Starting at the block given by the index *block_num*, write blocks from *buf* (an array of bytes) to the device. The number of blocks to write is given by the length of *buf*, which will be a multiple of the block size.

The second form allows writing at arbitrary locations within a block, and arbitrary lengths. Only the bytes being written should be changed, and the caller of this method must ensure that the relevant blocks are erased via a prior `ioctl` call. Starting at block index *block_num*, and byte offset within that block of *offset*, write bytes from *buf* (an array of bytes) to the device. The number of bytes to write is given by the length of *buf*.

Note that implementations must never implicitly erase blocks if the offset argument is specified, even if it is zero.

ioctl(*op*, *arg*)

Control the block device and query its parameters. The operation to perform is given by *op* which is one of the following integers:

- 1 – initialise the device (*arg* is unused)
- 2 – shutdown the device (*arg* is unused)
- 3 – sync the device (*arg* is unused)
- 4 – get a count of the number of blocks, should return an integer (*arg* is unused)
- 5 – get the number of bytes in a block, should return an integer, or `None` in which case the default value of 512 is used (*arg* is unused)
- 6 – erase a block, *arg* is the block number to erase

As a minimum `ioctl(4, ...)` must be intercepted; for littlefs `ioctl(6, ...)` must also be intercepted. The need for others is hardware dependent.

Prior to any call to `writeblocks(block, ...)` littlefs issues `ioctl(6, block)`. This enables a device driver to erase the block prior to a write if the hardware requires it. Alternatively a driver might intercept `ioctl(6, block)` and return 0 (success). In this case the driver assumes responsibility for detecting the need for erasure.

Unless otherwise stated `ioctl(op, arg)` can return `None`. Consequently an implementation can ignore unused values of `op`. Where `op` is intercepted, the return value for operations 4 and 5 are as detailed above. Other operations should return 0 on success and non-zero for failure, with the value returned being an `OSError` errno code.

1.1.14 `random` – generate random numbers

This module implements a pseudo-random number generator (PRNG).

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

Note: The following notation is used for intervals:

- () are open interval brackets and do not include their endpoints. For example, (0, 1) means greater than 0 and less than 1. In set notation: (0, 1) = {x | 0 < x < 1}.
- [] are closed interval brackets which include all their limit points. For example, [0, 1] means greater than or equal to 0 and less than or equal to 1. In set notation: [0, 1] = {x | 0 <= x <= 1}.

Note: The *randrange()*, *randint()* and *choice()* functions are only available if the `MICROPY_PY_URANDOM_EXTRA_FUNCS` configuration option is enabled.

Functions for integers

random.**getrandbits**(*n*)
Return an integer with *n* random bits (0 <= n <= 32).

random.**randint**(*a*, *b*)
Return a random integer in the range [*a*, *b*].

random.**randrange**(*stop*)
random.**randrange**(*start*, *stop*)
random.**randrange**(*start*, *stop*[, *step*])
The first form returns a random integer from the range [0, *stop*). The second form returns a random integer from the range [*start*, *stop*). The third form returns a random integer from the range [*start*, *stop*) in steps of *step*. For instance, calling `randrange(1, 10, 2)` will return odd numbers between 1 and 9 inclusive.

Functions for floats

random.**random**()
Return a random floating point number in the range [0.0, 1.0).

random.**uniform**(*a*, *b*)
Return a random floating point number N such that *a* <= N <= *b* for *a* <= *b*, and *b* <= N <= *a* for *b* < *a*.

Other Functions

random.**seed**(*n=None*, */*)
Initialise the random number generator module with the seed *n* which should be an integer. When no argument (or `None`) is passed in it will (if supported by the port) initialise the PRNG with a true random number (usually a hardware generated random number).

The `None` case only works if `MICROPY_PY_URANDOM_SEED_INIT_FUNC` is enabled by the port, otherwise it raises `ValueError`.

random.**choice**(*sequence*)
Chooses and returns one item at random from *sequence* (tuple, list or any object that supports the subscript operation).

1.1.15 `re` – simple regular expressions

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

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

Supported operators and special sequences are:

`.` Match any character.

`[...]` Match set of characters. Individual characters and ranges are supported, including negated sets (e.g. `[^a-c]`).

`^` Match the start of the string.

`$` Match the end of the string.

`?` Match zero or one of the previous sub-pattern.

`*` Match zero or more of the previous sub-pattern.

+ Match one or more of the previous sub-pattern.

?? Non-greedy version of ?, match zero or one, with the preference for zero.

***?** Non-greedy version of *, match zero or more, with the preference for the shortest match.

+? Non-greedy version of +, match one or more, with the preference for the shortest match.

| Match either the left-hand side or the right-hand side sub-patterns of this operator.

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

\d Matches digit. Equivalent to `[0-9]`.

\D Matches non-digit. Equivalent to `[^0-9]`.

\s Matches whitespace. Equivalent to `[ \t-\r]`.

\S Matches non-whitespace. Equivalent to `[^ \t-\r]`.

\w Matches word characters (ASCII only). Equivalent to `[A-Za-z0-9_]`.

\W Matches non word characters (ASCII only). Equivalent to `[^A-Za-z0-9_]`.

**** Escape character. Any other character following the backslash, except for those listed above, is taken literally. For example, `\*` is equivalent to literal * (not treated as the * operator). Note that `\r`, `\n`, etc. are not handled specially, and will be equivalent to literal letters `r`, `n`, etc. Due to this, its not recommended to use raw Python strings (`r""`) for regular expressions. For example, `r"\r\n"` when used as the regular expression is equivalent to `"rn"`. To match CR character followed by LF, use `"\r\n"`.

NOT SUPPORTED:

- counted repetitions (`{m,n}`)
- named groups (`(?P<name>...)`)
- non-capturing groups (`(?:...)`)
- more advanced assertions (`\b`, `\B`)
- special character escapes like `\r`, `\n` - use Pythons own escaping instead
- etc.

Example:

```
import re

# As re doesn't support escapes itself, use of r"" strings is not
# recommended.
regex = re.compile("[\r\n]")

regex.split("line1\rline2\nline3\r\n")

# Result:
# ['line1', 'line2', 'line3', '', '']
```

Functions

re.**compile**(*regex_str*[, *flags*])
: Compile regular expression, return *regex* object.

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

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

re.**sub**(*regex_str*, *replace*, *string*, *count=0*, *flags=0*, /)
: Compile *regex_str* and search for it in *string*, replacing all matches with *replace*, and returning the new string.

  replace can be a string or a function. If it is a string then escape sequences of the form \<number> and \g<number> can be used to expand to the corresponding group (or an empty string for unmatched groups). If *replace* is a function then it must take a single argument (the match) and should return a replacement string.

  If *count* is specified and non-zero then substitution will stop after this many substitutions are made. The *flags* argument is ignored.

  Note: availability of this function depends on *MicroPython port*.

re.**DEBUG**
: Flag value, display debug information about compiled expression. (Availability depends on *MicroPython port*.)

Regex objects

Compiled regular expression. Instances of this class are created using `re.compile()`.

regex.**match**(*string*)
regex.**search**(*string*)
regex.**sub**(*replace*, *string*, *count=0*, *flags=0*, /)
: Similar to the module-level functions `match()`, `search()` and `sub()`. Using methods is (much) more efficient if the same regex is applied to multiple strings.

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

Match objects

Match objects as returned by `match()` and `search()` methods, and passed to the replacement function in `sub()`.

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

match.**groups**()
: Return a tuple containing all the substrings of the groups of the match.

  Note: availability of this method depends on *MicroPython port*.

match.**start**([*index*])
match.**end**([*index*])
: Return the index in the original string of the start or end of the substring group that was matched. *index* defaults to the entire group, otherwise it will select a group.

  Note: availability of these methods depends on *MicroPython port*.

match.**span**([*index*])
Returns the 2-tuple `(match.start(index), match.end(index))`.

Note: availability of this method depends on *MicroPython port*.

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

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

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

Functions

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

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

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

class `Poll`

Methods

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

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

Note that flags like `select.POLLHUP` and `select.POLLERR` are *not* valid as input eventmask (these are unsolicited events which will be returned from *poll()* regardless of whether they are asked for). This semantics is per POSIX.

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

It is OK to call this function multiple times for the same *obj*. Successive calls will update *obj*s eventmask to the value of *eventmask* (i.e. will behave as *modify()*).

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

poll.**modify**(*obj*, *eventmask*)
Modify the *eventmask* for *obj*. If *obj* is not registered, *OSError* is raised with error of ENOENT.

poll.**poll**(*timeout=- 1*, */*)
Wait for at least one of the registered objects to become ready or have an exceptional condition, with optional timeout in milliseconds (if *timeout* arg is not specified or -1, there is no timeout).

Returns list of (`obj`, `event`,) tuples. There may be other elements in tuple, depending on a platform and version, so dont assume that its size is 2. The `event` element specifies which events happened with a stream and is a combination of `select.POLL*` constants described above. Note that flags `select.POLLHUP` and `select.POLLERR` can be returned at any time (even if were not asked for), and must be acted on accordingly (the corresponding

stream unregistered from poll and likely closed), because otherwise all further invocations of *poll()* may return immediately with these flags set for this stream again.

In case of timeout, an empty list is returned.

Difference to CPython

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

poll.**ipoll**(*timeout=- 1*, *flags=0*, */*)
Like *poll.poll()*, but instead returns an iterator which yields a *callee-owned tuple*. This function provides an efficient, allocation-free way to poll on streams.

If *flags* is 1, one-shot behaviour for events is employed: streams for which events happened will have their event masks automatically reset (equivalent to `poll.modify(obj, 0)`), so new events for such a stream wont be processed until new mask is set with *poll.modify()*. This behaviour is useful for asynchronous I/O schedulers.

Difference to CPython

This function is a MicroPython extension.

1.1.17 socket – socket module

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

This module provides access to the BSD socket interface.

Difference to CPython

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

Socket address format(s)

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

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

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

However, `socket` module (note the difference with native MicroPython `socket` module described here) provides CPython-compatible way to specify addresses using tuples, as described below. Note that depending on a *MicroPython port*, `socket` module can be builtin or need to be installed from *micropython-lib* (as in the case of *MicroPython*

Unix port), and some ports still accept only numeric addresses in the tuple format, and require to use `getaddrinfo` function to resolve domain names.

Summing up:

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

Tuple address format for `socket` module:

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

Functions

`socket.`**`socket`**(*af=AF_INET*, *type=SOCK_STREAM*, *proto=IPPROTO_TCP*, */*)

Create a new socket using the given address family, socket type and protocol number. Note that specifying *proto* in most cases is not required (and not recommended, as some MicroPython ports may omit `IPPROTO_*` constants). Instead, *type* argument will select needed protocol automatically:

```
# Create STREAM TCP socket
socket(AF_INET, SOCK_STREAM)
# Create DGRAM UDP socket
socket(AF_INET, SOCK_DGRAM)
```

`socket.`**`getaddrinfo`**(*host*, *port*, *af=0*, *type=0*, *proto=0*, *flags=0*, */*)

Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. Arguments *af*, *type*, and *proto* (which have the same meaning as for the `socket()` function) can be used to filter which kind of addresses are returned. If a parameter is not specified or zero, all combinations of addresses can be returned (requiring filtering on the user side).

The resulting list of 5-tuples has the following structure:

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

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

```
s = socket.socket()
# This assumes that if "type" is not specified, an address for
# SOCK_STREAM will be returned, which may be not true
s.connect(socket.getaddrinfo('www.micropython.org', 80)[0][-1])
```

Recommended use of filtering params:

```
s = socket.socket()
# Guaranteed to return an address which can be connect'ed to for
# stream operation.
s.connect(socket.getaddrinfo('www.micropython.org', 80, 0, SOCK_STREAM)[0][-1])
```

Difference to CPython

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

`socket.`**`inet_ntop`**(*af*, *bin_addr*)
Convert a binary network address *bin_addr* of the given address family *af* to a textual representation:

```
>>> socket.inet_ntop(socket.AF_INET, b"\x7f\0\0\1")
'127.0.0.1'
```

`socket.`**`inet_pton`**(*af*, *txt_addr*)
Convert a textual network address *txt_addr* of the given address family *af* to a binary representation:

```
>>> socket.inet_pton(socket.AF_INET, "1.2.3.4")
b'\x01\x02\x03\x04'
```

Constants

`socket.`**`AF_INET`**
`socket.`**`AF_INET6`**
Address family types. Availability depends on a particular *MicroPython port*.

`socket.`**`SOCK_STREAM`**
`socket.`**`SOCK_DGRAM`**
Socket types.

`socket.`**`IPPROTO_UDP`**
`socket.`**`IPPROTO_TCP`**
IP protocol numbers. Availability depends on a particular *MicroPython port*. Note that you dont need to specify these in a call to *socket.socket()*, because *SOCK_STREAM* socket type automatically selects *IPPROTO_TCP*, and *SOCK_DGRAM* - *IPPROTO_UDP*. Thus, the only real use of these constants is as an argument to *setsockopt()*.

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

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

Constants specific to WiPy:

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

class socket

Methods

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

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

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

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

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

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

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

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

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

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

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

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

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

socket.**settimeout**(*value*)
Note: Not every port supports this method, see below.

Set a timeout on blocking socket operations. The value argument can be a nonnegative floating point number expressing seconds, or None. If a non-zero value is given, subsequent socket operations will raise an *OSError*

exception if the timeout period value has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If None is given, the socket is put in blocking mode.

Not every *MicroPython port* supports this method. A more portable and generic solution is to use *select.poll* object. This allows to wait on multiple objects at the same time (and not just on sockets, but on generic *stream* objects which support polling). Example:

```
# Instead of:
s.settimeout(1.0)  # time in seconds
s.read(10)  # may timeout

# Use:
poller = select.poll()
poller.register(s, select.POLLIN)
res = poller.poll(1000)  # time in milliseconds
if not res:
    # s is still not ready for input, i.e. operation timed out
```

Difference to CPython

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

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

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

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

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

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

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

Difference to CPython

As MicroPython doesnt support buffered streams, values of *buffering* parameter is ignored and treated as if it was 0 (unbuffered).

Difference to CPython

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

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

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

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

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

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

Return value: the line read.

socket.**write**(*buf*)
Write the buffer of bytes to the socket. This function will try to write all data to a socket (no short writes). This may be not possible with a non-blocking socket though, and returned value will be less than the length of *buf*.

Return value: number of bytes written.

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

Difference to CPython

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

1.1.18 ssl – SSL/TLS module

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

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

Functions

ssl.**wrap_socket**(*sock*, *server_side=False*, *keyfile=None*, *certfile=None*, *cert_reqs=CERT_NONE*, *ca_certs=None*, *do_handshake=True*)
Takes a *stream* *sock* (usually socket.socket instance of `SOCK_STREAM` type), and returns an instance of ssl.SSLSocket, which wraps the underlying stream in an SSL context. Returned object has the usual *stream* interface methods like `read()`, `write()`, etc. A server-side SSL socket should be created from a normal socket returned from *accept()* on a non-SSL listening server socket.

- *do_handshake* determines whether the handshake is done as part of the `wrap_socket` or whether it is deferred to be done as part of the initial reads or writes (there is no `do_handshake` method as in CPython). For blocking sockets doing the handshake immediately is standard. For non-blocking sockets (i.e. when the *sock* passed into `wrap_socket` is in non-blocking mode) the handshake should generally be deferred because otherwise `wrap_socket` blocks until it completes. Note that in AXTLS the handshake can be deferred until the first read or write but it then blocks until completion.

Depending on the underlying module implementation in a particular *MicroPython port*, some or all keyword arguments above may be not supported.

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

CPythons `wrap_socket` returns an `SSLSocket` object which has methods typical for sockets, such as `send`, `recv`, etc. MicroPythons `wrap_socket` returns an object more similar to CPythons `SSLObject` which does not have these socket methods.

Exceptions

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

Constants

`ssl.CERT_NONE`
`ssl.CERT_OPTIONAL`
`ssl.CERT_REQUIRED`
: Supported values for *cert_reqs* parameter.

1.1.19 `struct` – pack and unpack primitive data types

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

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

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

Difference to CPython

Whitespace is not supported in format strings.

Functions

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

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

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

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

`struct.unpack_from(`*fmt*, *data*, *offset=0*, */*`)`
: Unpack from the *data* starting at *offset* according to the format string *fmt*. *offset* may be negative to count from the end of *buffer*. The return value is a tuple of the unpacked values.

1.1.20 sys – system specific functions

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

Functions

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

`sys.atexit`(*func*)
Register *func* to be called upon termination. *func* must be a callable that takes no arguments, or `None` to disable the call. The `atexit` function will return the previous value set by this function, which is initially `None`.

Difference to CPython

This function is a MicroPython extension intended to provide similar functionality to the `atexit` module in CPython.

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

Difference to CPython

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

`sys.settrace`(*tracefunc*)
Enable tracing of bytecode execution. For details see the CPython documentaion.

This function requires a custom MicroPython build as it is typically not present in pre-built firmware (due to it affecting performance). The relevant configuration option is *MICROPY_PY_SYS_SETTRACE*.

Constants

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

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

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

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

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

Difference to CPython

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

---

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

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

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

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

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

Difference to CPython

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

---

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

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

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

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

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

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

Difference to CPython

Only the first three version numbers (major, minor, micro) are supported and they can be referenced only by index, not by name.

1.1.21 `time` – time related functions

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

The `time` module provides functions for getting the current time and date, measuring time intervals, and for delays.

Time Epoch: Unix port uses standard for POSIX systems epoch of 1970-01-01 00:00:00 UTC. However, embedded ports use epoch of 2000-01-01 00:00:00 UTC.

Maintaining actual calendar date/time: This requires a Real Time Clock (RTC). On systems with underlying OS (including some RTOS), an RTC may be implicit. Setting and maintaining actual calendar time is responsibility of OS/RTOS and is done outside of MicroPython, it just uses OS API to query date/time. On baremetal ports however system time depends on `machine.RTC()` object. The current calendar time may be set using `machine.RTC().datetime(tuple)` function, and maintained by following means:

- By a backup battery (which may be an additional, optional component for a particular board).
- Using networked time protocol (requires setup by a port/user).
- Set manually by a user on each power-up (many boards then maintain RTC time across hard resets, though some may require setting it again in such case).

If actual calendar time is not maintained with a system/MicroPython RTC, functions below which require reference to current absolute time may behave not as expected.

Functions

`time.gmtime([secs])`
`time.localtime([secs])`

Convert the time *secs* expressed in seconds since the Epoch (see above) into an 8-tuple which contains: `(year, month, mday, hour, minute, second, weekday, yearday)` If *secs* is not provided or None, then the current time from the RTC is used.

The *gmtime()* function returns a date-time tuple in UTC, and *localtime()* returns a date-time tuple in local time.

The format of the entries in the 8-tuple are:

- year includes the century (for example 2014).
- month is 1-12
- mday is 1-31
- hour is 0-23
- minute is 0-59
- second is 0-59
- weekday is 0-6 for Mon-Sun
- yearday is 1-366

time.**mktime**()
This is inverse function of localtime. Its argument is a full 8-tuple which expresses a time as per localtime. It returns an integer which is the number of seconds since Jan 1, 2000.

time.**sleep**(*seconds*)
Sleep for the given number of seconds. Some boards may accept *seconds* as a floating-point number to sleep for a fractional number of seconds. Note that other boards may not accept a floating-point argument, for compatibility with them use *sleep_ms()* and *sleep_us()* functions.

time.**sleep_ms**(*ms*)
Delay for given number of milliseconds, should be positive or 0.

This function will delay for at least the given number of milliseconds, but may take longer than that if other processing must take place, for example interrupt handlers or other threads. Passing in 0 for *ms* will still allow this other processing to occur. Use *sleep_us()* for more precise delays.

time.**sleep_us**(*us*)
Delay for given number of microseconds, should be positive or 0.

This function attempts to provide an accurate delay of at least *us* microseconds, but it may take longer if the system has other higher priority processing to perform.

time.**ticks_ms**()
Returns an increasing millisecond counter with an arbitrary reference point, that wraps around after some value.

The wrap-around value is not explicitly exposed, but we will refer to it as *TICKS_MAX* to simplify discussion. Period of the values is *TICKS_PERIOD = TICKS_MAX + 1*. *TICKS_PERIOD* is guaranteed to be a power of two, but otherwise may differ from port to port. The same period value is used for all of *ticks_ms()*, *ticks_us()*, *ticks_cpu()* functions (for simplicity). Thus, these functions will return a value in range [*0* .. *TICKS_MAX*], inclusive, total *TICKS_PERIOD* values. Note that only non-negative values are used. For the most part, you should treat values returned by these functions as opaque. The only operations available for them are *ticks_diff()* and *ticks_add()* functions described below.

Note: Performing standard mathematical operations (+, -) or relational operators (<, <=, >, >=) directly on these value will lead to invalid result. Performing mathematical operations and then passing their results as arguments to *ticks_diff()* or *ticks_add()* will also lead to invalid results from the latter functions.

time.**ticks_us**()
Just like *ticks_ms()* above, but in microseconds.

time.**ticks_cpu**()
Similar to *ticks_ms()* and *ticks_us()*, but with the highest possible resolution in the system. This is usually CPU clocks, and thats why the function is named that way. But it doesnt have to be a CPU clock, some other timing source available in a system (e.g. high-resolution timer) can be used instead. The exact timing unit (resolution) of this function is not specified on `time` module level, but documentation for a specific port may provide more specific information. This function is intended for very fine benchmarking or very tight real-time loops. Avoid using it in portable code.

Availability: Not every port implements this function.

time.**ticks_add**(*ticks*, *delta*)
Offset ticks value by a given number, which can be either positive or negative. Given a *ticks* value, this function allows to calculate ticks value *delta* ticks before or after it, following modular-arithmetic definition of tick values (see *ticks_ms()* above). *ticks* parameter must be a direct result of call to *ticks_ms()*, *ticks_us()*, or *ticks_cpu()* functions (or from previous call to *ticks_add()*). However, *delta* can be an arbitrary integer number or numeric expression. *ticks_add()* is useful for calculating deadlines for events/tasks. (Note: you must use *ticks_diff()* function to work with deadlines.)

Examples:

```
# Find out what ticks value there was 100ms ago
print(ticks_add(time.ticks_ms(), -100))

# Calculate deadline for operation and test for it
deadline = ticks_add(time.ticks_ms(), 200)
while ticks_diff(deadline, time.ticks_ms()) > 0:
    do_a_little_of_something()

# Find out TICKS_MAX used by this port
print(ticks_add(0, -1))
```

time.**ticks_diff**(*ticks1*, *ticks2*)

Measure ticks difference between values returned from *ticks_ms()*, *ticks_us()*, or *ticks_cpu()* functions, as a signed value which may wrap around.

The argument order is the same as for subtraction operator, `ticks_diff(ticks1, ticks2)` has the same meaning as `ticks1 - ticks2`. However, values returned by *ticks_ms()*, etc. functions may wrap around, so directly using subtraction on them will produce incorrect result. That is why *ticks_diff()* is needed, it implements modular (or more specifically, ring) arithmetics to produce correct result even for wrap-around values (as long as they not too distant inbetween, see below). The function returns **signed** value in the range [*-TICKS_PERIOD/2 .. TICKS_PERIOD/2-1*] (thats a typical range definition for twos-complement signed binary integers). If the result is negative, it means that *ticks1* occurred earlier in time than *ticks2*. Otherwise, it means that *ticks1* occurred after *ticks2*. This holds **only** if *ticks1* and *ticks2* are apart from each other for no more than *TICKS_PERIOD/2-1* ticks. If that does not hold, incorrect result will be returned. Specifically, if two tick values are apart for *TICKS_PERIOD/2-1* ticks, that value will be returned by the function. However, if *TICKS_PERIOD/2* of real-time ticks has passed between them, the function will return *-TICKS_PERIOD/2* instead, i.e. result value will wrap around to the negative range of possible values.

Informal rationale of the constraints above: Suppose you are locked in a room with no means to monitor passing of time except a standard 12-notch clock. Then if you look at dial-plate now, and dont look again for another 13 hours (e.g., if you fall for a long sleep), then once you finally look again, it may seem to you that only 1 hour has passed. To avoid this mistake, just look at the clock regularly. Your application should do the same. Too long sleep metaphor also maps directly to application behaviour: dont let your application run any single task for too long. Run tasks in steps, and do time-keeping inbetween.

ticks_diff() is designed to accommodate various usage patterns, among them:

- Polling with timeout. In this case, the order of events is known, and you will deal only with positive results of *ticks_diff()*:

```
# Wait for GPIO pin to be asserted, but at most 500us
start = time.ticks_us()
while pin.value() == 0:
    if time.ticks_diff(time.ticks_us(), start) > 500:
        raise TimeoutError
```

- Scheduling events. In this case, *ticks_diff()* result may be negative if an event is overdue:

```
# This code snippet is not optimized
now = time.ticks_ms()
scheduled_time = task.scheduled_time()
if ticks_diff(scheduled_time, now) > 0:
    print("Too early, let's nap")
    sleep_ms(ticks_diff(scheduled_time, now))
    task.run()
```

(continues on next page)

(continued from previous page)

```
elif ticks_diff(scheduled_time, now) == 0:
    print("Right at time!")
    task.run()
elif ticks_diff(scheduled_time, now) < 0:
    print("Oops, running late, tell task to run faster!")
    task.run(run_faster=true)
```

Note: Do not pass *time()* values to *ticks_diff()*, you should use normal mathematical operations on them. But note that *time()* may (and will) also overflow. This is known as https://en.wikipedia.org/wiki/Year_2038_problem .

time.**time**()
Returns the number of seconds, as an integer, since the Epoch, assuming that underlying RTC is set and maintained as described above. If an RTC is not set, this function returns number of seconds since a port-specific reference point in time (for embedded boards without a battery-backed RTC, usually since power up or reset). If you want to develop portable MicroPython application, you should not rely on this function to provide higher than second precision. If you need higher precision, absolute timestamps, use *time_ns()*. If relative times are acceptable then use the *ticks_ms()* and *ticks_us()* functions. If you need calendar time, *gmtime()* or *localtime()* without an argument is a better choice.

Difference to CPython

In CPython, this function returns number of seconds since Unix epoch, 1970-01-01 00:00 UTC, as a floating-point, usually having microsecond precision. With MicroPython, only Unix port uses the same Epoch, and if floating-point precision allows, returns sub-second precision. Embedded hardware usually doesnt have floating-point precision to represent both long time ranges and subsecond precision, so they use integer value with second precision. Some embedded hardware also lacks battery-powered RTC, so returns number of seconds since last power-up or from other relative, hardware-specific point (e.g. reset).

---

time.**time_ns**()
Similar to *time()* but returns nanoseconds since the Epoch, as an integer (usually a big integer, so will allocate on the heap).

1.1.22 uasyncio asynchronous I/O scheduler

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

Example:

```
import uasyncio

async def blink(led, period_ms):
    while True:
        led.on()
        await uasyncio.sleep_ms(5)
        led.off()
        await uasyncio.sleep_ms(period_ms)

async def main(led1, led2):
    uasyncio.create_task(blink(led1, 700))
    uasyncio.create_task(blink(led2, 400))
```

(continues on next page)

(continued from previous page)

```
    await uasyncio.sleep_ms(10_000)

# Running on a pyboard
from pyb import LED
uasyncio.run(main(LED(1), LED(2)))

# Running on a generic board
from machine import Pin
uasyncio.run(main(Pin(1), Pin(2)))
```

Core functions

uasyncio.**create_task**(*coro*)
Create a new task from the given coroutine and schedule it to run.

Returns the corresponding *Task* object.

uasyncio.**current_task**()
Return the *Task* object associated with the currently running task.

uasyncio.**run**(*coro*)
Create a new task from the given coroutine and run it until it completes.

Returns the value returned by *coro*.

uasyncio.**sleep**(*t*)
Sleep for *t* seconds (can be a float).

This is a coroutine.

uasyncio.**sleep_ms**(*t*)
Sleep for *t* milliseconds.

This is a coroutine, and a MicroPython extension.

Additional functions

uasyncio.**wait_for**(*awaitable*, *timeout*)
Wait for the *awaitable* to complete, but cancel it if it takes longer than *timeout* seconds. If *awaitable* is not a task then a task will be created from it.

If a timeout occurs, it cancels the task and raises `asyncio.TimeoutError`: this should be trapped by the caller. The task receives `asyncio.CancelledError` which may be ignored or trapped using `try...except` or `try...finally` to run cleanup code.

Returns the return value of *awaitable*.

This is a coroutine.

uasyncio.**wait_for_ms**(*awaitable*, *timeout*)
Similar to *wait_for* but *timeout* is an integer in milliseconds.

This is a coroutine, and a MicroPython extension.

uasyncio.**gather**(**awaitables*, *return_exceptions=False*)
Run all *awaitables* concurrently. Any *awaitables* that are not tasks are promoted to tasks.

Returns a list of return values of all *awaitables*.

This is a coroutine.

class Task

class uasyncio.**Task**
This object wraps a coroutine into a running task. Tasks can be waited on using `await task`, which will wait for the task to complete and return the return value of the task.

Tasks should not be created directly, rather use *create_task* to create them.

Task.**cancel**()
Cancel the task by injecting `asyncio.CancelledError` into it. The task may ignore this exception. Cleanup code may be run by trapping it, or via `try ... finally`.

class Event

class uasyncio.**Event**
Create a new event which can be used to synchronise tasks. Events start in the cleared state.

Event.**is_set**()
Returns `True` if the event is set, `False` otherwise.

Event.**set**()
Set the event. Any tasks waiting on the event will be scheduled to run.

Note: This must be called from within a task. It is not safe to call this from an IRQ, scheduler callback, or other thread. See *ThreadSafeFlag*.

Event.**clear**()
Clear the event.

Event.**wait**()
Wait for the event to be set. If the event is already set then it returns immediately.

This is a coroutine.

class ThreadSafeFlag

class uasyncio.**ThreadSafeFlag**
Create a new flag which can be used to synchronise a task with code running outside the asyncio loop, such as other threads, IRQs, or scheduler callbacks. Flags start in the cleared state.

ThreadSafeFlag.**set**()
Set the flag. If there is a task waiting on the event, it will be scheduled to run.

ThreadSafeFlag.**wait**()
Wait for the flag to be set. If the flag is already set then it returns immediately.

A flag may only be waited on by a single task at a time.

This is a coroutine.

class Lock

class uasyncio.**Lock**
Create a new lock which can be used to coordinate tasks. Locks start in the unlocked state.

In addition to the methods below, locks can be used in an `async with` statement.

Lock.**locked**()
Returns `True` if the lock is locked, otherwise `False`.

Lock.**acquire**()
Wait for the lock to be in the unlocked state and then lock it in an atomic way. Only one task can acquire the lock at any one time.

This is a coroutine.

Lock.**release**()
Release the lock. If any tasks are waiting on the lock then the next one in the queue is scheduled to run and the lock remains locked. Otherwise, no tasks are waiting an the lock becomes unlocked.

TCP stream connections

uasyncio.**open_connection**(*host*, *port*)
Open a TCP connection to the given *host* and *port*. The *host* address will be resolved using *socket.getaddrinfo*, which is currently a blocking call.

Returns a pair of streams: a reader and a writer stream. Will raise a socket-specific `OSError` if the host could not be resolved or if the connection could not be made.

This is a coroutine.

uasyncio.**start_server**(*callback*, *host*, *port*, *backlog=5*)
Start a TCP server on the given *host* and *port*. The *callback* will be called with incoming, accepted connections, and be passed 2 arguments: reader and writer streams for the connection.

Returns a *Server* object.

This is a coroutine.

class uasyncio.**Stream**
This represents a TCP stream connection. To minimise code this class implements both a reader and a writer, and both `StreamReader` and `StreamWriter` alias to this class.

Stream.**get_extra_info**(*v*)
Get extra information about the stream, given by *v*. The valid values for *v* are: `peername`.

Stream.**close**()
Close the stream.

Stream.**wait_closed**()
Wait for the stream to close.

This is a coroutine.

Stream.**read**(*n*)
Read up to *n* bytes and return them.

This is a coroutine.

Stream.**readinto**(*buf*)
Read up to n bytes into *buf* with n being equal to the length of *buf*.

Return the number of bytes read into *buf*.

This is a coroutine, and a MicroPython extension.

Stream.**readexactly**(*n*)
Read exactly *n* bytes and return them as a bytes object.

Raises an EOFError exception if the stream ends before reading *n* bytes.

This is a coroutine.

Stream.**readline**()
Read a line and return it.

This is a coroutine.

Stream.**write**(*buf*)
Accumulated *buf* to the output buffer. The data is only flushed when *Stream.drain* is called. It is recommended to call *Stream.drain* immediately after calling this function.

Stream.**drain**()
Drain (write) all buffered output data out to the stream.

This is a coroutine.

class uasyncio.**Server**
This represents the server class returned from *start_server*. It can be used in an `async with` statement to close the server upon exit.

Server.**close**()
Close the server.

Server.**wait_closed**()
Wait for the server to close.

This is a coroutine.

Event Loop

uasyncio.**get_event_loop**()
Return the event loop used to schedule and run tasks. See *Loop*.

uasyncio.**new_event_loop**()
Reset the event loop and return it.

Note: since MicroPython only has a single event loop this function just resets the loops state, it does not create a new one.

class uasyncio.**Loop**
This represents the object which schedules and runs tasks. It cannot be created, use *get_event_loop* instead.

Loop.**create_task**(*coro*)
Create a task from the given *coro* and return the new *Task* object.

Loop.**run_forever**()
Run the event loop until *stop()* is called.

Loop.**run_until_complete**(*awaitable*)
Run the given *awaitable* until it completes. If *awaitable* is not a task then it will be promoted to one.

Loop.**stop**()
Stop the event loop.

Loop.**close**()
Close the event loop.

Loop.**set_exception_handler**(*handler*)
Set the exception handler to call when a Task raises an exception that is not caught. The *handler* should accept two arguments: `(loop, context)`.

Loop.**get_exception_handler**()
Get the current exception handler. Returns the handler, or `None` if no custom handler is set.

Loop.**default_exception_handler**(*context*)
The default exception handler that is called.

Loop.**call_exception_handler**(*context*)
Call the current exception handler. The argument *context* is passed through and is a dictionary containing keys: `'message'`, `'exception'`, `'future'`.

1.1.23 zlib – zlib decompression

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

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

Functions

zlib.**decompress**(*data*, *wbits=0*, *bufsize=0*, */*)
Return decompressed *data* as bytes. *wbits* is DEFLATE dictionary window size used during compression (8-15, the dictionary size is power of 2 of that value). Additionally, if value is positive, *data* is assumed to be zlib stream (with zlib header). Otherwise, if its negative, its assumed to be raw DEFLATE stream. *bufsize* parameter is for compatibility with CPython and is ignored.

class zlib.**DecompIO**(*stream*, *wbits=0*, */*)
Create a *stream* wrapper which allows transparent decompression of compressed data in another *stream*. This allows to process compressed streams with data larger than available heap size. In addition to values described in *decompress()*, *wbits* may take values 24..31 (16 + 8..15), meaning that input stream has gzip header.

Difference to CPython

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

1.1.24 _thread – multithreading support

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

This module implements multithreading support.

This module is highly experimental and its API is not yet fully settled and not yet described in this documentation.

1.2 MicroPython-specific libraries

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

1.2.1 `bluetooth` low-level Bluetooth

This module provides an interface to a Bluetooth controller on a board. Currently this supports Bluetooth Low Energy (BLE) in Central, Peripheral, Broadcaster, and Observer roles, as well as GATT Server and Client and L2CAP connection-oriented-channels. A device may operate in multiple roles concurrently. Pairing (and bonding) is supported on some ports.

This API is intended to match the low-level Bluetooth protocol and provide building-blocks for higher-level abstractions such as specific device types.

Note: This module is still under development and its classes, functions, methods and constants are subject to change.

---

class BLE

Constructor

class bluetooth.**BLE**
Returns the singleton BLE object.

Configuration

BLE.**active**([*active*], /)
Optionally changes the active state of the BLE radio, and returns the current state.

The radio must be made active before using any other methods on this class.

BLE.**config**(*'param'*, /)
BLE.**config**(*, *param*=*value*, ...)
Get or set configuration values of the BLE interface. To get a value the parameter name should be quoted as a string, and just one parameter is queried at a time. To set values use the keyword syntax, and one ore more parameter can be set at a time.

Currently supported values are:

- `'mac'`: The current address in use, depending on the current address mode. This returns a tuple of `(addr_type, addr)`.

  See *gatts_write* for details about address type.

  This may only be queried while the interface is currently active.

- `'addr_mode'`: Sets the address mode. Values can be:
  - 0x00 - PUBLIC - Use the controllers public address.
  - 0x01 - RANDOM - Use a generated static address.
  - 0x02 - RPA - Use resolvable private addresses.
  - 0x03 - NRPA - Use non-resolvable private addresses.

By default the interface mode will use a PUBLIC address if available, otherwise it will use a RANDOM address.

- `'gap_name'`: Get/set the GAP device name used by service 0x1800, characteristic 0x2a00. This can be set at any time and changed multiple times.
- `'rxbuf'`: Get/set the size in bytes of the internal buffer used to store incoming events. This buffer is global to the entire BLE driver and so handles incoming data for all events, including all characteristics. Increasing this allows better handling of bursty incoming data (for example scan results) and the ability to receive larger characteristic values.
- `'mtu'`: Get/set the MTU that will be used during a ATT MTU exchange. The resulting MTU will be the minimum of this and the remote devices MTU. ATT MTU exchange will not happen automatically (unless the remote device initiates it), and must be manually initiated with *gattc_exchange_mtu*. Use the `_IRQ_MTU_EXCHANGED` event to discover the MTU for a given connection.
- `'bond'`: Sets whether bonding will be enabled during pairing. When enabled, pairing requests will set the bond flag and the keys will be stored by both devices.
- `'mitm'`: Sets whether MITM-protection is required for pairing.
- `'io'`: Sets the I/O capabilities of this device.

  Available options are:

```
_IO_CAPABILITY_DISPLAY_ONLY = const(0)
_IO_CAPABILITY_DISPLAY_YESNO = const(1)
_IO_CAPABILITY_KEYBOARD_ONLY = const(2)
_IO_CAPABILITY_NO_INPUT_OUTPUT = const(3)
_IO_CAPABILITY_KEYBOARD_DISPLAY = const(4)
```

- `'le_secure'`: Sets whether LE Secure pairing is required. Default is false (i.e. allow Legacy Pairing).

Event Handling

`BLE.irq(`*handler*`, /)`

Registers a callback for events from the BLE stack. The *handler* takes two arguments, `event` (which will be one of the codes below) and `data` (which is an event-specific tuple of values).

Note: As an optimisation to prevent unnecessary allocations, the `addr`, `adv_data`, `char_data`, `notify_data`, and `uuid` entries in the tuples are read-only memoryview instances pointing to *bluetooth*s internal ringbuffer, and are only valid during the invocation of the IRQ handler function. If your program needs to save one of these values to access after the IRQ handler has returned (e.g. by saving it in a class instance or global variable), then it needs to take a copy of the data, either by using `bytes()` or `bluetooth.UUID()`, like this:

```
connected_addr = bytes(addr)  # equivalently: adv_data, char_data, or notify_data
matched_uuid = bluetooth.UUID(uuid)
```

For example, the IRQ handler for a scan result might inspect the `adv_data` to decide if its the correct device, and only then copy the address data to be used elsewhere in the program. And to print data from within the IRQ handler, `print(bytes(addr))` will be needed.

An event handler showing all possible events:

```
def bt_irq(event, data):
    if event == _IRQ_CENTRAL_CONNECT:
        # A central has connected to this peripheral.
        conn_handle, addr_type, addr = data
```

(continues on next page)

(continued from previous page)

```
    elif event == _IRQ_CENTRAL_DISCONNECT:
        # A central has disconnected from this peripheral.
        conn_handle, addr_type, addr = data
    elif event == _IRQ_GATTS_WRITE:
        # A client has written to this characteristic or descriptor.
        conn_handle, attr_handle = data
    elif event == _IRQ_GATTS_READ_REQUEST:
        # A client has issued a read. Note: this is only supported on STM32.
        # Return a non-zero integer to deny the read (see below), or zero (or None)
        # to accept the read.
        conn_handle, attr_handle = data
    elif event == _IRQ_SCAN_RESULT:
        # A single scan result.
        addr_type, addr, adv_type, rssi, adv_data = data
    elif event == _IRQ_SCAN_DONE:
        # Scan duration finished or manually stopped.
        pass
    elif event == _IRQ_PERIPHERAL_CONNECT:
        # A successful gap_connect().
        conn_handle, addr_type, addr = data
    elif event == _IRQ_PERIPHERAL_DISCONNECT:
        # Connected peripheral has disconnected.
        conn_handle, addr_type, addr = data
    elif event == _IRQ_GATTC_SERVICE_RESULT:
        # Called for each service found by gattc_discover_services().
        conn_handle, start_handle, end_handle, uuid = data
    elif event == _IRQ_GATTC_SERVICE_DONE:
        # Called once service discovery is complete.
        # Note: Status will be zero on success, implementation-specific value␣
↪otherwise.
        conn_handle, status = data
    elif event == _IRQ_GATTC_CHARACTERISTIC_RESULT:
        # Called for each characteristic found by gattc_discover_services().
        conn_handle, def_handle, value_handle, properties, uuid = data
    elif event == _IRQ_GATTC_CHARACTERISTIC_DONE:
        # Called once service discovery is complete.
        # Note: Status will be zero on success, implementation-specific value␣
↪otherwise.
        conn_handle, status = data
    elif event == _IRQ_GATTC_DESCRIPTOR_RESULT:
        # Called for each descriptor found by gattc_discover_descriptors().
        conn_handle, dsc_handle, uuid = data
    elif event == _IRQ_GATTC_DESCRIPTOR_DONE:
        # Called once service discovery is complete.
        # Note: Status will be zero on success, implementation-specific value␣
↪otherwise.
        conn_handle, status = data
    elif event == _IRQ_GATTC_READ_RESULT:
        # A gattc_read() has completed.
        conn_handle, value_handle, char_data = data
    elif event == _IRQ_GATTC_READ_DONE:
        # A gattc_read() has completed.
```

(continues on next page)

(continued from previous page)

```
        # Note: The value_handle will be zero on btstack (but present on NimBLE).
        # Note: Status will be zero on success, implementation-specific value␣
↪otherwise.
        conn_handle, value_handle, status = data
    elif event == _IRQ_GATTC_WRITE_DONE:
        # A gattc_write() has completed.
        # Note: The value_handle will be zero on btstack (but present on NimBLE).
        # Note: Status will be zero on success, implementation-specific value␣
↪otherwise.
        conn_handle, value_handle, status = data
    elif event == _IRQ_GATTC_NOTIFY:
        # A server has sent a notify request.
        conn_handle, value_handle, notify_data = data
    elif event == _IRQ_GATTC_INDICATE:
        # A server has sent an indicate request.
        conn_handle, value_handle, notify_data = data
    elif event == _IRQ_GATTS_INDICATE_DONE:
        # A client has acknowledged the indication.
        # Note: Status will be zero on successful acknowledgment, implementation-
↪specific value otherwise.
        conn_handle, value_handle, status = data
    elif event == _IRQ_MTU_EXCHANGED:
        # ATT MTU exchange complete (either initiated by us or the remote device).
        conn_handle, mtu = data
    elif event == _IRQ_L2CAP_ACCEPT:
        # A new channel has been accepted.
        # Return a non-zero integer to reject the connection, or zero (or None) to␣
↪accept.
        conn_handle, cid, psm, our_mtu, peer_mtu = data
    elif event == _IRQ_L2CAP_CONNECT:
        # A new channel is now connected (either as a result of connecting or␣
↪accepting).
        conn_handle, cid, psm, our_mtu, peer_mtu = data
    elif event == _IRQ_L2CAP_DISCONNECT:
        # Existing channel has disconnected (status is zero), or a connection␣
↪attempt failed (non-zero status).
        conn_handle, cid, psm, status = data
    elif event == _IRQ_L2CAP_RECV:
        # New data is available on the channel. Use l2cap_recvinto to read.
        conn_handle, cid = data
    elif event == _IRQ_L2CAP_SEND_READY:
        # A previous l2cap_send that returned False has now completed and the␣
↪channel is ready to send again.
        # If status is non-zero, then the transmit buffer overflowed and the␣
↪application should re-send the data.
        conn_handle, cid, status = data
    elif event == _IRQ_CONNECTION_UPDATE:
        # The remote device has updated connection parameters.
        conn_handle, conn_interval, conn_latency, supervision_timeout, status = data
    elif event == _IRQ_ENCRYPTION_UPDATE:
        # The encryption state has changed (likely as a result of pairing or␣
↪bonding).
```

(continues on next page)

(continued from previous page)

```
        conn_handle, encrypted, authenticated, bonded, key_size = data
    elif event == _IRQ_GET_SECRET:
        # Return a stored secret.
        # If key is None, return the index'th value of this sec_type.
        # Otherwise return the corresponding value for this sec_type and key.
        sec_type, index, key = data
        return value
    elif event == _IRQ_SET_SECRET:
        # Save a secret to the store for this sec_type and key.
        sec_type, key, value = data
        return True
    elif event == _IRQ_PASSKEY_ACTION:
        # Respond to a passkey request during pairing.
        # See gap_passkey() for details.
        # action will be an action that is compatible with the configured "io"␣
↪config.
        # passkey will be non-zero if action is "numeric comparison".
        conn_handle, action, passkey = data
```

The event codes are:

```
from micropython import const
_IRQ_CENTRAL_CONNECT = const(1)
_IRQ_CENTRAL_DISCONNECT = const(2)
_IRQ_GATTS_WRITE = const(3)
_IRQ_GATTS_READ_REQUEST = const(4)
_IRQ_SCAN_RESULT = const(5)
_IRQ_SCAN_DONE = const(6)
_IRQ_PERIPHERAL_CONNECT = const(7)
_IRQ_PERIPHERAL_DISCONNECT = const(8)
_IRQ_GATTC_SERVICE_RESULT = const(9)
_IRQ_GATTC_SERVICE_DONE = const(10)
_IRQ_GATTC_CHARACTERISTIC_RESULT = const(11)
_IRQ_GATTC_CHARACTERISTIC_DONE = const(12)
_IRQ_GATTC_DESCRIPTOR_RESULT = const(13)
_IRQ_GATTC_DESCRIPTOR_DONE = const(14)
_IRQ_GATTC_READ_RESULT = const(15)
_IRQ_GATTC_READ_DONE = const(16)
_IRQ_GATTC_WRITE_DONE = const(17)
_IRQ_GATTC_NOTIFY = const(18)
_IRQ_GATTC_INDICATE = const(19)
_IRQ_GATTS_INDICATE_DONE = const(20)
_IRQ_MTU_EXCHANGED = const(21)
_IRQ_L2CAP_ACCEPT = const(22)
_IRQ_L2CAP_CONNECT = const(23)
_IRQ_L2CAP_DISCONNECT = const(24)
_IRQ_L2CAP_RECV = const(25)
_IRQ_L2CAP_SEND_READY = const(26)
_IRQ_CONNECTION_UPDATE = const(27)
_IRQ_ENCRYPTION_UPDATE = const(28)
_IRQ_GET_SECRET = const(29)
_IRQ_SET_SECRET = const(30)
```

For the `_IRQ_GATTS_READ_REQUEST` event, the available return codes are:

```
_GATTS_NO_ERROR = const(0x00)
_GATTS_ERROR_READ_NOT_PERMITTED = const(0x02)
_GATTS_ERROR_WRITE_NOT_PERMITTED = const(0x03)
_GATTS_ERROR_INSUFFICIENT_AUTHENTICATION = const(0x05)
_GATTS_ERROR_INSUFFICIENT_AUTHORIZATION = const(0x08)
_GATTS_ERROR_INSUFFICIENT_ENCRYPTION = const(0x0f)
```

For the `_IRQ_PASSKEY_ACTION` event, the available actions are:

```
_PASSKEY_ACTION_NONE = const(0)
_PASSKEY_ACTION_INPUT = const(2)
_PASSKEY_ACTION_DISPLAY = const(3)
_PASSKEY_ACTION_NUMERIC_COMPARISON = const(4)
```

In order to save space in the firmware, these constants are not included on the *bluetooth* module. Add the ones that you need from the list above to your program.

Broadcaster Role (Advertiser)

`BLE.gap_advertise`(*interval_us*, *adv_data=None*, ***, *resp_data=None*, *connectable=True*)

Starts advertising at the specified interval (in **micro**seconds). This interval will be rounded down to the nearest 625us. To stop advertising, set *interval_us* to `None`.

adv_data and *resp_data* can be any type that implements the buffer protocol (e.g. `bytes`, `bytearray`, `str`). *adv_data* is included in all broadcasts, and *resp_data* is send in reply to an active scan.

Note: if *adv_data* (or *resp_data*) is `None`, then the data passed to the previous call to `gap_advertise` will be re-used. This allows a broadcaster to resume advertising with just `gap_advertise(interval_us)`. To clear the advertising payload pass an empty `bytes`, i.e. `b''`.

Observer Role (Scanner)

`BLE.gap_scan`(*duration_ms*, *interval_us=1280000*, *window_us=11250*, *active=False*, */*)

Run a scan operation lasting for the specified duration (in **milli**seconds).

To scan indefinitely, set *duration_ms* to `0`.

To stop scanning, set *duration_ms* to `None`.

Use *interval_us* and *window_us* to optionally configure the duty cycle. The scanner will run for *window_us* **micro**seconds every *interval_us* **micro**seconds for a total of *duration_ms* **milli**seconds. The default interval and window are 1.28 seconds and 11.25 milliseconds respectively (background scanning).

For each scan result the `_IRQ_SCAN_RESULT` event will be raised, with event data `(addr_type, addr, adv_type, rssi, adv_data)`.

`addr_type` values indicate public or random addresses:

- 0x00 - PUBLIC
- 0x01 - RANDOM (either static, RPA, or NRPA, the type is encoded in the address itself)

`adv_type` values correspond to the Bluetooth Specification:

- 0x00 - ADV_IND - connectable and scannable undirected advertising
- 0x01 - ADV_DIRECT_IND - connectable directed advertising

- 0x02 - ADV_SCAN_IND - scannable undirected advertising
- 0x03 - ADV_NONCONN_IND - non-connectable undirected advertising
- 0x04 - SCAN_RSP - scan response

`active` can be set `True` if you want to receive scan responses in the results.

When scanning is stopped (either due to the duration finishing or when explicitly stopped), the `_IRQ_SCAN_DONE` event will be raised.

Central Role

A central device can connect to peripherals that it has discovered using the observer role (see *gap_scan*) or with a known address.

`BLE.gap_connect(`*addr_type*, *addr*, *scan_duration_ms=2000*, *min_conn_interval_us=None*, *max_conn_interval_us=None*, `/)`

Connect to a peripheral.

See *gap_scan* for details about address types.

To cancel an outstanding connection attempt early, call `gap_connect(None)`.

On success, the `_IRQ_PERIPHERAL_CONNECT` event will be raised. If cancelling a connection attempt, the `_IRQ_PERIPHERAL_DISCONNECT` event will be raised.

The device will wait up to *scan_duration_ms* to receive an advertising payload from the device.

The connection interval can be configured in **micro**seconds using either or both of *min_conn_interval_us* and *max_conn_interval_us*. Otherwise a default interval will be chosen, typically between 30000 and 50000 microseconds. A shorter interval will increase throughput, at the expense of power usage.

Peripheral Role

A peripheral device is expected to send connectable advertisements (see *gap_advertise*). It will usually be acting as a GATT server, having first registered services and characteristics using *gatts_register_services*.

When a central connects, the `_IRQ_CENTRAL_CONNECT` event will be raised.

Central & Peripheral Roles

`BLE.gap_disconnect(`*conn_handle*, `/)`

Disconnect the specified connection handle. This can either be a central that has connected to this device (if acting as a peripheral) or a peripheral that was previously connected to by this device (if acting as a central).

On success, the `_IRQ_PERIPHERAL_DISCONNECT` or `_IRQ_CENTRAL_DISCONNECT` event will be raised.

Returns `False` if the connection handle wasnt connected, and `True` otherwise.

GATT Server

A GATT server has a set of registered services. Each service may contain characteristics, which each have a value. Characteristics can also contain descriptors, which themselves have values.

These values are stored locally, and are accessed by their value handle which is generated during service registration. They can also be read from or written to by a remote client device. Additionally, a server can notify a characteristic to a connected client via a connection handle.

A device in either central or peripheral roles may function as a GATT server, however in most cases it will be more common for a peripheral device to act as the server.

Characteristics and descriptors have a default maximum size of 20 bytes. Anything written to them by a client will be truncated to this length. However, any local write will increase the maximum size, so if you want to allow larger writes from a client to a given characteristic, use *gatts_write* after registration. e.g. `gatts_write(char_handle, bytes(100))`.

BLE.**gatts_register_services**(*services_definition*, /)

Configures the server with the specified services, replacing any existing services.

services_definition is a list of **services**, where each **service** is a two-element tuple containing a UUID and a list of **characteristics**.

Each **characteristic** is a two-or-three-element tuple containing a UUID, a **flags** value, and optionally a list of *descriptors*.

Each **descriptor** is a two-element tuple containing a UUID and a **flags** value.

The **flags** are a bitwise-OR combination of the flags defined below. These set both the behaviour of the characteristic (or descriptor) as well as the security and privacy requirements.

The return value is a list (one element per service) of tuples (each element is a value handle). Characteristics and descriptor handles are flattened into the same tuple, in the order that they are defined.

The following example registers two services (Heart Rate, and Nordic UART):

```
HR_UUID = bluetooth.UUID(0x180D)
HR_CHAR = (bluetooth.UUID(0x2A37), bluetooth.FLAG_READ | bluetooth.FLAG_NOTIFY,)
HR_SERVICE = (HR_UUID, (HR_CHAR,),)
UART_UUID = bluetooth.UUID('6E400001-B5A3-F393-E0A9-E50E24DCCA9E')
UART_TX = (bluetooth.UUID('6E400003-B5A3-F393-E0A9-E50E24DCCA9E'), bluetooth.FLAG_
↪READ | bluetooth.FLAG_NOTIFY,)
UART_RX = (bluetooth.UUID('6E400002-B5A3-F393-E0A9-E50E24DCCA9E'), bluetooth.FLAG_
↪WRITE,)
UART_SERVICE = (UART_UUID, (UART_TX, UART_RX,),)
SERVICES = (HR_SERVICE, UART_SERVICE,)
( (hr,), (tx, rx,), ) = bt.gatts_register_services(SERVICES)
```

The three value handles (`hr`, `tx`, `rx`) can be used with *gatts_read*, *gatts_write*, *gatts_notify*, and *gatts_indicate*.

Note: Advertising must be stopped before registering services.

Available flags for characteristics and descriptors are:

```
from micropython import const
_FLAG_BROADCAST = const(0x0001)
_FLAG_READ = const(0x0002)
_FLAG_WRITE_NO_RESPONSE = const(0x0004)
```

(continues on next page)

(continued from previous page)

```
_FLAG_WRITE = const(0x0008)
_FLAG_NOTIFY = const(0x0010)
_FLAG_INDICATE = const(0x0020)
_FLAG_AUTHENTICATED_SIGNED_WRITE = const(0x0040)

_FLAG_AUX_WRITE = const(0x0100)
_FLAG_READ_ENCRYPTED = const(0x0200)
_FLAG_READ_AUTHENTICATED = const(0x0400)
_FLAG_READ_AUTHORIZED = const(0x0800)
_FLAG_WRITE_ENCRYPTED = const(0x1000)
_FLAG_WRITE_AUTHENTICATED = const(0x2000)
_FLAG_WRITE_AUTHORIZED = const(0x4000)
```

As for the IRQs above, any required constants should be added to your Python code.

BLE.**gatts_read**(*value_handle*, /)
Reads the local value for this handle (which has either been written by *gatts_write* or by a remote client).

BLE.**gatts_write**(*value_handle*, *data*, *send_update=False*, /)
Writes the local value for this handle, which can be read by a client.

If *send_update* is `True`, then any subscribed clients will be notified (or indicated, depending on what theyre subscribed to and which operations the characteristic supports) about this write.

BLE.**gatts_notify**(*conn_handle*, *value_handle*, *data=None*, /)
Sends a notification request to a connected client.

If *data* is not `None`, then that value is sent to the client as part of the notification. The local value will not be modified.

Otherwise, if *data* is `None`, then the current local value (as set with *gatts_write*) will be sent.

Note: The notification will be sent regardless of the subscription status of the client to this characteristic.

BLE.**gatts_indicate**(*conn_handle*, *value_handle*, /)
Sends an indication request containing the characteristics current value to a connected client.

On acknowledgment (or failure, e.g. timeout), the `_IRQ_GATTS_INDICATE_DONE` event will be raised.

Note: The indication will be sent regardless of the subscription status of the client to this characteristic.

BLE.**gatts_set_buffer**(*value_handle*, *len*, *append=False*, /)
Sets the internal buffer size for a value in bytes. This will limit the largest possible write that can be received. The default is 20.

Setting *append* to `True` will make all remote writes append to, rather than replace, the current value. At most *len* bytes can be buffered in this way. When you use *gatts_read*, the value will be cleared after reading. This feature is useful when implementing something like the Nordic UART Service.

GATT Client

A GATT client can discover and read/write characteristics on a remote GATT server.

It is more common for a central role device to act as the GATT client, however its also possible for a peripheral to act as a client in order to discover information about the central that has connected to it (e.g. to read the device name from the device information service).

BLE.**gattc_discover_services**(*conn_handle*, *uuid=None*, /)
Query a connected server for its services.

Optionally specify a service *uuid* to query for that service only.

For each service discovered, the `_IRQ_GATTC_SERVICE_RESULT` event will be raised, followed by `_IRQ_GATTC_SERVICE_DONE` on completion.

BLE.**gattc_discover_characteristics**(*conn_handle*, *start_handle*, *end_handle*, *uuid=None*, /)
Query a connected server for characteristics in the specified range.

Optionally specify a characteristic *uuid* to query for that characteristic only.

You can use `start_handle=1`, `end_handle=0xffff` to search for a characteristic in any service.

For each characteristic discovered, the `_IRQ_GATTC_CHARACTERISTIC_RESULT` event will be raised, followed by `_IRQ_GATTC_CHARACTERISTIC_DONE` on completion.

BLE.**gattc_discover_descriptors**(*conn_handle*, *start_handle*, *end_handle*, /)
Query a connected server for descriptors in the specified range.

For each descriptor discovered, the `_IRQ_GATTC_DESCRIPTOR_RESULT` event will be raised, followed by `_IRQ_GATTC_DESCRIPTOR_DONE` on completion.

BLE.**gattc_read**(*conn_handle*, *value_handle*, /)
Issue a remote read to a connected server for the specified characteristic or descriptor handle.

When a value is available, the `_IRQ_GATTC_READ_RESULT` event will be raised. Additionally, the `_IRQ_GATTC_READ_DONE` will be raised.

BLE.**gattc_write**(*conn_handle*, *value_handle*, *data*, *mode=0*, /)
Issue a remote write to a connected server for the specified characteristic or descriptor handle.

The argument *mode* specifies the write behaviour, with the currently supported values being:

- `mode=0` (default) is a write-without-response: the write will be sent to the remote server but no confirmation will be returned, and no event will be raised.
- `mode=1` is a write-with-response: the remote server is requested to send a response/acknowledgement that it received the data.

If a response is received from the remote server the `_IRQ_GATTC_WRITE_DONE` event will be raised.

BLE.**gattc_exchange_mtu**(*conn_handle*, /)
Initiate MTU exchange with a connected server, using the preferred MTU set using `BLE.config(mtu=value)`.

The `_IRQ_MTU_EXCHANGED` event will be raised when MTU exchange completes.

Note: MTU exchange is typically initiated by the central. When using the BlueKitchen stack in the central role, it does not support a remote peripheral initiating the MTU exchange. NimBLE works for both roles.

L2CAP connection-oriented-channels

This feature allows for socket-like data exchange between two BLE devices. Once the devices are connected via GAP, either device can listen for the other to connect on a numeric PSM (Protocol/Service Multiplexer).

Note: This is currently only supported when using the NimBLE stack on STM32 and Unix (not ESP32). Only one L2CAP channel may be active at a given time (i.e. you cannot connect while listening).

Active L2CAP channels are identified by the connection handle that they were established on and a CID (channel ID).

Connection-oriented channels have built-in credit-based flow control. Unlike ATT, where devices negotiate a shared MTU, both the listening and connecting devices each set an independent MTU which limits the maximum amount of outstanding data that the remote device can send before it is fully consumed in *l2cap_recvinto*.

BLE.**l2cap_listen**(*psm*, *mtu*, /)
Start listening for incoming L2CAP channel requests on the specified *psm* with the local MTU set to *mtu*.

When a remote device initiates a connection, the `_IRQ_L2CAP_ACCEPT` event will be raised, which gives the listening server a chance to reject the incoming connection (by returning a non-zero integer).

Once the connection is accepted, the `_IRQ_L2CAP_CONNECT` event will be raised, allowing the server to obtain the channel id (CID) and the local and remote MTU.

Note: It is not currently possible to stop listening.

BLE.**l2cap_connect**(*conn_handle*, *psm*, *mtu*, /)
Connect to a listening peer on the specified *psm* with local MTU set to *mtu*.

On successful connection, the the `_IRQ_L2CAP_CONNECT` event will be raised, allowing the client to obtain the CID and the local and remote (peer) MTU.

An unsuccessful connection will raise the `_IRQ_L2CAP_DISCONNECT` event with a non-zero status.

BLE.**l2cap_disconnect**(*conn_handle*, *cid*, /)
Disconnect an active L2CAP channel with the specified *conn_handle* and *cid*.

BLE.**l2cap_send**(*conn_handle*, *cid*, *buf*, /)
Send the specified *buf* (which must support the buffer protocol) on the L2CAP channel identified by *conn_handle* and *cid*.

The specified buffer cannot be larger than the remote (peer) MTU, and no more than twice the size of the local MTU.

This will return `False` if the channel is now stalled, which means that *l2cap_send* must not be called again until the `_IRQ_L2CAP_SEND_READY` event is received (which will happen when the remote device grants more credits, typically after it has received and processed the data).

BLE.**l2cap_recvinto**(*conn_handle*, *cid*, *buf*, /)
Receive data from the specified *conn_handle* and *cid* into the provided *buf* (which must support the buffer protocol, e.g. bytearray or memoryview).

Returns the number of bytes read from the channel.

If *buf* is None, then returns the number of bytes available.

Note: After receiving the `_IRQ_L2CAP_RECV` event, the application should continue calling *l2cap_recvinto* until no more bytes are available in the receive buffer (typically up to the size of the remote (peer) MTU).

Until the receive buffer is empty, the remote device will not be granted more channel credits and will be unable to send any more data.

Pairing and bonding

Pairing allows a connection to be encrypted and authenticated via exchange of secrets (with optional MITM protection via passkey authentication).

Bonding is the process of storing those secrets into non-volatile storage. When bonded, a device is able to resolve a resolvable private address (RPA) from another device based on the stored identity resolving key (IRK). To support bonding, an application must implement the `_IRQ_GET_SECRET` and `_IRQ_SET_SECRET` events.

Note: This is currently only supported when using the NimBLE stack on STM32 and Unix (not ESP32).

`BLE.gap_pair(`*conn_handle*, `/)`
Initiate pairing with the remote device.

Before calling this, ensure that the `io`, `mitm`, `le_secure`, and `bond` configuration options are set (via *config*).

On successful pairing, the `_IRQ_ENCRYPTION_UPDATE` event will be raised.

`BLE.gap_passkey(`*conn_handle*, *action*, *passkey*, `/)`
Respond to a `_IRQ_PASSKEY_ACTION` event for the specified *conn_handle* and *action*.

The *passkey* is a numeric value and will depend on on the *action* (which will depend on what I/O capability has been set):

- When the *action* is `_PASSKEY_ACTION_INPUT`, then the application should prompt the user to enter the passkey that is shown on the remote device.
- When the *action* is `_PASSKEY_ACTION_DISPLAY`, then the application should generate a random 6-digit passkey and show it to the user.
- When the *action* is `_PASSKEY_ACTION_NUMERIC_COMPARISON`, then the application should show the passkey that was provided in the `_IRQ_PASSKEY_ACTION` event and then respond with either `0` (cancel pairing), or 1 (accept pairing).

class UUID

Constructor

`class bluetooth.UUID(`*value*, `/)`
Creates a UUID instance with the specified **value**.

The **value** can be either:

- A 16-bit integer. e.g. `0x2908`.
- A 128-bit UUID string. e.g. `'6E400001-B5A3-F393-E0A9-E50E24DCCA9E'`.

1.2.2 `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 io.BytesIO, a raw flash partition, etc.
# Oftentimes, you want to create a database file if it doesn't
# exist and open if it exists. Idiom below takes care of this.
# DO NOT open database with "a+b" access mode.
try:
    f = open("mydb", "r+b")
except OSError:
    f = open("mydb", "w+b")

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

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

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

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

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

del db[b"2"]

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

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

db.close()

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

Functions

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

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

- *flags* - Currently unused.
- *pagesize* - Page size used for the nodes in BTree. Acceptable range is 512-65536. If 0, a port-specific default will be used, optimized for ports memory usage and/or performance.
- *cachesize* - Suggested memory cache size in bytes. For a board with enough memory using larger values may improve performance. Cache policy is as follows: entire cache is not allocated at once; instead, accessing a new page in database will allocate a memory buffer for it, until value specified by *cachesize* is reached. Then, these buffers will be managed using LRU (least recently used) policy. More buffers may still be allocated if needed (e.g., if a database contains big keys and/or values). Allocated cache buffers arent reclaimed.
- *minkeypage* - Minimum number of keys to store per page. Default value of 0 equivalent to 2.

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

Methods

btree.**close**()

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

btree.**flush**()

Flush any data in cache to the underlying stream.

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

Standard dictionary methods.

btree.**__iter__**()

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

btree.**keys**([*start_key*[, *end_key*[, *flags*]]])
btree.**values**([*start_key*[, *end_key*[, *flags*]]])
btree.**items**([*start_key*[, *end_key*[, *flags*]]])

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

Constants

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

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

1.2.3 cryptolib – cryptographic ciphers

Classes

class cryptolib.**aes**

classmethod __init__(*key*, *mode*[, *IV*])
Initialize cipher object, suitable for encryption/decryption. Note: after initialization, cipher object can be use only either for encryption or decryption. Running decrypt() operation after encrypt() or vice versa is not supported.

Parameters are:

- *key* is an encryption/decryption key (bytes-like).
- *mode* is:
  - 1 (or `cryptolib.MODE_ECB` if it exists) for Electronic Code Book (ECB).
  - 2 (or `cryptolib.MODE_CBC` if it exists) for Cipher Block Chaining (CBC).
  - 6 (or `cryptolib.MODE_CTR` if it exists) for Counter mode (CTR).
- *IV* is an initialization vector for CBC mode.
- For Counter mode, *IV* is the initial value for the counter.

encrypt(*in_buf*[, *out_buf*])
Encrypt *in_buf*. If no *out_buf* is given result is returned as a newly allocated *bytes* object. Otherwise, result is written into mutable buffer *out_buf*. *in_buf* and *out_buf* can also refer to the same mutable buffer, in which case data is encrypted in-place.

decrypt(*in_buf*[, *out_buf*])
Like *encrypt()*, but for decryption.

1.2.4 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 FrameBuffers. It is useful when generating output for displays.

For example:

```
import framebuf

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

fbuf.fill(0)
fbuf.text('MicroPython!', 0, 0, 0xffff)
fbuf.hline(0, 9, 96, 0xffff)
```

Constructors

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

Construct a FrameBuffer object. The parameters are:

- *buffer* is an object with a buffer protocol which must be large enough to contain every pixel defined by the width, height and format of the FrameBuffer.
- *width* is the width of the FrameBuffer in pixels
- *height* is the height of the FrameBuffer in pixels
- *format* specifies the type of pixel used in the FrameBuffer; permissible values are listed under Constants below. These set the number of bits used to encode a color value and the layout of these bits in *buffer*. Where a color value c is passed to a method, c is a small integer with an encoding that is dependent on the format of the FrameBuffer.
- *stride* is the number of pixels between each horizontal line of pixels in the FrameBuffer. This defaults to *width* but may need adjustments when implementing a FrameBuffer within another larger FrameBuffer or screen. The *buffer* size must accommodate an increased step size.

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

Drawing primitive shapes

The following methods draw shapes onto the FrameBuffer.

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

Fill the entire FrameBuffer with the specified color.

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

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

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

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

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

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

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

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

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

Drawing text

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

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

Other methods

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

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

FrameBuffer.**blit**(*fbuf*, *x*, *y*, *key=- 1*, *palette=None*)

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.

The *palette* argument enables blitting between FrameBuffers with differing formats. Typical usage is to render a monochrome or grayscale glyph/icon to a color display. The *palette* is a FrameBuffer instance whose format is that of the current FrameBuffer. The *palette* height is one pixel and its pixel width is the number of colors in the source FrameBuffer. The *palette* for an N-bit source needs 2**N pixels; the *palette* for a monochrome source would have 2 pixels representing background and foreground colors. The application assigns a color to each pixel in the *palette*. The color of the current pixel will be that of that *palette* pixel whose x position is the color of the corresponding source pixel.

Constants

framebuf.**MONO_VLSB**

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

framebuf.**MONO_HLSB**

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

framebuf.**MONO_HMSB**

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

framebuf.**RGB565**

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

framebuf.**GS2_HMSB**

Grayscale (2-bit) color format

framebuf.**GS4_HMSB**
Grayscale (4-bit) color format

framebuf.**GS8**
Grayscale (8-bit) color format

1.2.5 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 *Writing interrupt handlers*.

Reset related functions

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

machine.**soft_reset**()
Performs a soft reset of the interpreter, deleting all Python objects and resetting the Python heap. It tries to retain the method by which the user is connected to the MicroPython REPL (eg serial, USB, Wifi).

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

machine.**bootloader**([*value*])
Reset the device and enter its bootloader. This is typically used to put the device into a state where it can be programmed with new firmware.

Some ports support passing in an optional *value* argument which can control which bootloader to enter, what to pass to it, or other things.

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**([*hz*])
Returns the CPU frequency in hertz.

On some ports this can also be used to set the CPU frequency by passing in *hz*.

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**()

Note: This function is deprecated, use *lightsleep()* instead with no arguments.

---

machine.**lightsleep**([*time_ms*])
machine.**deepsleep**([*time_ms*])
Stops execution in an attempt to enter a low power state.

If *time_ms* is specified then this will be the maximum time in milliseconds that the sleep will last for. Otherwise the sleep can last indefinitely.

With or without a timeout, execution may resume at any time if there are events that require processing. Such events, or wake sources, should be configured before sleeping, like *Pin* change or *RTC* timeout.

The precise behaviour and power-saving capabilities of lightsleep and deepsleep is highly dependent on the underlying hardware, but the general properties are:

- A lightsleep has full RAM and state retention. Upon wake execution is resumed from the point where the sleep was requested, with all subsystems operational.
- A deepsleep may not retain RAM or any other state of the system (for example peripherals or network interfaces). Upon wake execution is resumed from the main script, similar to a hard or power-on reset. The *reset_cause()* function will return *machine.DEEPSLEEP* and this can be used to distinguish a deepsleep wake from other resets.

machine.**wake_reason**()
Get the wake reason. See *constants* for the possible return values.

Availability: ESP32, WiPy.

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).

machine.**bitstream**(*pin*, *encoding*, *timing*, *data*, /)

Transmits *data* by bit-banging the specified *pin*. The *encoding* argument specifies how the bits are encoded, and *timing* is an encoding-specific timing specification.

The supported encodings are:

- `0` for high low pulse duration modulation. This will transmit 0 and 1 bits as timed pulses, starting with the most significant bit. The *timing* must be a four-tuple of nanoseconds in the format `(high_time_0, low_time_0, high_time_1, low_time_1)`. For example, `(400, 850, 800, 450)` is the timing specification for WS2812 RGB LEDs at 800kHz.

The accuracy of the timing varies between ports. On Cortex M0 at 48MHz, it is at best +/- 120ns, however on faster MCUs (ESP8266, ESP32, STM32, Pyboard), it will be closer to +/-30ns.

Note: For controlling WS2812 / NeoPixel strips, see the *neopixel* module for a higher-level API.

machine.**rng**()

Return a 24-bit software generated random number.

Availability: WiPy.

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 with a pull down resistor
p0.init(p0.IN, p0.PULL_DOWN)

# 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.
  - `Pin.ANALOG` - Pin is configured for analog input, see the *ADC* class.

- `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.irq(*handler=None*, *trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING*, ***, *priority=1*, *wake=None*, *hard=False*)
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. The handler must take exactly one argument which is the `Pin` instance.
- `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 ORed 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 ORed together to make a pin generate interrupts in more than one power mode.
- `hard` if true a hardware interrupt is used. This reduces the delay between the pin change and the handler being called. Hard interrupt handlers may not allocate memory; see *Writing interrupt handlers*. Not all ports support this argument.

This method returns a callback object.

The following methods are not part of the core Pin API and only implemented on certain ports.

Pin.low()
Set pin to 0 output level.

Availability: nrf, rp2, stm32 ports.

Pin.high()
Set pin to 1 output level.

Availability: nrf, rp2, stm32 ports.

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

Availability: cc3200, stm32 ports.

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

Availability: cc3200, stm32 ports.

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

Availability: cc3200 port.

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**
Pin.**ANALOG**
Selects the pin mode.

Pin.**PULL_UP**
Pin.**PULL_DOWN**
Pin.**PULL_HOLD**
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 the *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. In other words, 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 of whether different users have an active-high or active-low LED, a normally open or normally closed relay - you can develop a single, nicely looking application which works with each of them, and capture hardware configuration differences in few lines in the config file of your app.

Example:

```
from machine import Pin, Signal

# Suppose you have an active-high LED on pin 0
led1_pin = Pin(0, Pin.OUT)
# ... and active-low LED on pin 1
led2_pin = Pin(1, Pin.OUT)

# Now to light up both of them using Pin class, you'll need to set
```

(continues on next page)

(continued from previous page)

```
# them to different values
led1_pin.value(1)
led2_pin.value(0)

# Signal class allows to abstract away active-high/active-low
# difference
led1 = Signal(led1_pin, invert=False)
led2 = Signal(led2_pin, invert=True)

# Now lighting up them looks the same
led1.value(1)
led2.value(1)

# Even better:
led1.on()
led2.on()
```

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 use cases 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 a 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 offered 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 doesnt 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. Therere 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 ADC – analog to digital conversion

The ADC class provides an interface to analog-to-digital convertors, and represents a single endpoint that can sample a continuous voltage and convert it to a discretised value.

Example usage:

```
import machine

adc = machine.ADC(pin)    # create an ADC object acting on a pin
val = adc.read_u16()      # read a raw analog value in the range 0-65535
```

Constructors

class machine.**ADC**(*id*)
: Access the ADC associated with a source identified by *id*. This *id* may be an integer (usually specifying a channel number), a *Pin* object, or other value supported by the underlying machine.

Methods

ADC.**read_u16**()
: Take an analog reading and return an integer in the range 0-65535. The return value represents the raw reading taken by the ADC, scaled such that the minimum value is 0 and the maximum value is 65535.

class PWM – pulse width modulation

This class provides pulse width modulation output.

Example usage:

```
from machine import PWM

pwm = PWM(pin)          # create a PWM object on a pin
pwm.duty_u16(32768)     # set duty to 50%

# reinitialise with a period of 200us, duty of 5us
pwm.init(freq=5000, duty_ns=5000)

pwm.duty_ns(3000)       # set pulse width to 3us

pwm.deinit()
```

Constructors

class machine.**PWM**(*dest*, *, *freq*, *duty_u16*, *duty_ns*)

Construct and return a new PWM object using the following parameters:

- *dest* is the entity on which the PWM is output, which is usually a *machine.Pin* object, but a port may allow other values, like integers.
- *freq* should be an integer which sets the frequency in Hz for the PWM cycle.
- *duty_u16* sets the duty cycle as a ratio `duty_u16 / 65535`.
- *duty_ns* sets the pulse width in nanoseconds.

Setting *freq* may affect other PWM objects if the objects share the same underlying PWM generator (this is hardware specific). Only one of *duty_u16* and *duty_ns* should be specified at a time.

Methods

PWM.**init**(*, *freq*, *duty_u16*, *duty_ns*)

Modify settings for the PWM object. See the above constructor for details about the parameters.

PWM.**deinit**()

Disable the PWM output.

PWM.**freq**([*value*])

Get or set the current frequency of the PWM output.

With no arguments the frequency in Hz is returned.

With a single *value* argument the frequency is set to that value in Hz. The method may raise a `ValueError` if the frequency is outside the valid range.

PWM.**duty_u16**([*value*])

Get or set the current duty cycle of the PWM output, as an unsigned 16-bit value in the range 0 to 65535 inclusive.

With no arguments the duty cycle is returned.

With a single *value* argument the duty cycle is set to that value, measured as the ratio `value / 65535`.

PWM.**duty_ns**([*value*])
Get or set the current pulse width of the PWM output, as a value in nanoseconds.

With no arguments the pulse width in nanoseconds is returned.

With a single *value* argument the pulse width is set to that value.

Limitations of PWM

- Not all frequencies can be generated with absolute accuracy due to the discrete nature of the computing hardware. Typically the PWM frequency is obtained by dividing some integer base frequency by an integer divider. For example, if the base frequency is 80MHz and the required PWM frequency is 300kHz the divider must be a non-integer number 80000000 / 300000 = 266.67. After rounding the divider is set to 267 and the PWM frequency will be 80000000 / 267 = 299625.5 Hz, not 300kHz. If the divider is set to 266 then the PWM frequency will be 80000000 / 266 = 300751.9 Hz, but again not 300kHz.
- The duty cycle has the same discrete nature and its absolute accuracy is not achievable. On most hardware platforms the duty will be applied at the next frequency period. Therefore, you should wait more than 1/frequency before measuring the duty.
- The frequency and the duty cycle resolution are usually interdependent. The higher the PWM frequency the lower the duty resolution which is available, and vice versa. For example, a 300kHz PWM frequency can have a duty cycle resolution of 8 bit, not 16-bit as may be expected. In this case, the lowest 8 bits of *duty_u16* are insignificant. So:

```
pwm=PWM(Pin(13), freq=300_000, duty_u16=2**16//2)
```

and:

```
pwm=PWM(Pin(13), freq=300_000, duty_u16=2**16//2 + 255)
```

will generate PWM with the same 50% duty cycle.

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.**init**(*baudrate=9600*, *bits=8*, *parity=None*, *stop=1*, ***, ...)
Initialise the UART bus with the given parameters:

- *baudrate* is the clock rate.
- *bits* is the number of bits per character, 7, 8 or 9.
- *parity* is the parity, `None`, 0 (even) or 1 (odd).
- *stop* is the number of stop bits, 1 or 2.

Additional keyword-only parameters that may be supported by a port are:

- *tx* specifies the TX pin to use.
- *rx* specifies the RX pin to use.
- *rts* specifies the RTS (output) pin to use for hardware receive flow control.
- *cts* specifies the CTS (input) pin to use for hardware transmit flow control.
- *txbuf* specifies the length in characters of the TX buffer.
- *rxbuf* specifies the length in characters of the RX buffer.
- *timeout* specifies the time to wait for the first character (in ms).
- *timeout_char* specifies the time to wait between characters (in ms).
- *invert* specifies which lines to invert.
- *flow* specifies which hardware flow control signals to use. The value is a bitmask.
  - `0` will ignore hardware flow control signals.
  - `UART.RTS` will enable receive flow control by using the RTS output pin to signal if the receive FIFO has sufficient space to accept more data.
  - `UART.CTS` will enable transmit flow control by pausing transmission when the CTS input pin signals that the receiver is running low on buffer space.
  - `UART.RTS | UART.CTS` will enable both, for full hardware flow control.

On the WiPy only the following keyword-only parameter is supported:

- *pins* is a 4 or 2 item list indicating the TX, RX, RTS and CTS pins (in that order). Any of the pins can be None if one wants the UART to operate with limited functionality. If the RTS pin is given the the RX pin must be given as well. The same applies to CTS. When no pins are given, then the default set of TX and

RX pins is taken, and hardware flow control will be disabled. If *pins* is `None`, no pin assignment will be made.

`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. It may return sooner if a timeout is reached. The timeout is configurable in the constructor.

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. It may return sooner if a timeout is reached. The timeout is configurable in the constructor.

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

`UART.readline()`
Read a line, ending in a newline character. It may return sooner if a timeout is reached. The timeout is configurable in the constructor.

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.

`UART.irq(trigger, priority=1, handler=None, wake=machine.IDLE)`
Create a callback to be triggered when data is received on the UART.

- *trigger* can only be `UART.RX_ANY`
- *priority* level of the interrupt. Can take values in the range 1-7. Higher values represent higher priorities.
- *handler* an optional function to be called when new characters arrive.
- *wake* can only be `machine.IDLE`.

Note: The handler will be called whenever any of the following two conditions are met:

- 8 new characters have been received.
- At least 1 new character is waiting in the Rx buffer and the Rx line has been silent for the duration of 1 complete frame.

This means that when the handler function is called there will be between 1 to 8 characters waiting.

Returns an irq object.

Availability: WiPy.

Constants

UART.**RX_ANY**
IRQ trigger sources

Availability: WiPy.

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

SPI is a synchronous serial protocol that is driven by a controller. 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, CS (Chip Select), to select a particular device on a bus with which communication takes place. Management of a CS signal should happen in user code (via machine.Pin class).

Both hardware and software SPI implementations exist via the *machine.SPI* and *machine.SoftSPI* classes. Hardware SPI uses underlying hardware support of the system to perform the reads/writes and is usually efficient and fast but may have restrictions on which pins can be used. Software SPI is implemented by bit-banging and can be used on any pin but is not as efficient. These classes have the same methods available and differ primarily in the way they are constructed.

Example usage:

```
from machine import SPI, Pin

spi = SPI(0, baudrate=400000)           # Create SPI peripheral 0 at frequency of 400kHz.
                                        # Depending on the use case, extra parameters␣
↪may be required
                                        # to select the bus characteristics and/or pins␣
↪to use.
cs = Pin(4, mode=Pin.OUT, value=1)      # Create chip-select on pin 4.

try:
    cs(0)                               # Select peripheral.
    spi.write(b"12345678")              # Write 8 bytes, and don't care about received␣
↪data.
finally:
    cs(1)                               # Deselect peripheral.

try:
    cs(0)                               # Select peripheral.
    rxdata = spi.read(8, 0x42)          # Read 8 bytes while writing 0x42 for each byte.
finally:
    cs(1)                               # Deselect peripheral.

rxdata = bytearray(8)
try:
    cs(0)                               # Select peripheral.
```

(continues on next page)

(continued from previous page)

```
    spi.readinto(rxdata, 0x42)          # Read 8 bytes inplace while writing 0x42 for␣
↪each byte.
finally:
    cs(1)                               # Deselect peripheral.

txdata = b"12345678"
rxdata = bytearray(len(txdata))
try:
    cs(0)                               # Select peripheral.
    spi.write_readinto(txdata, rxdata)  # Simultaneously write and read bytes.
finally:
    cs(1)                               # Deselect peripheral.
```

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.

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.

class machine.**SoftSPI**(*baudrate=500000*, ***, *polarity=0*, *phase=0*, *bits=8*, *firstbit=MSB*, *sck=None*, *mosi=None*, *miso=None*)

Construct a new software SPI object. Additional parameters must be given, usually at least *sck*, *mosi* and *miso*, and these are used to initialise the bus. See *SPI.init* for a description of the parameters.

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 doesnt `sck`, `mosi`, `miso` arguments, and instead allows to specify them as a tuple of `pins` parameter.

In the case of hardware SPI the actual clock frequency may be lower than the requested baudrate. This is dependant on the platform hardware. The actual rate may be determined by printing the SPI object.

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.**CONTROLLER**
for initialising the SPI bus to controller; 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.

Both hardware and software I2C implementations exist via the *machine.I2C* and *machine.SoftI2C* classes. Hardware I2C uses underlying hardware support of the system to perform the reads/writes and is usually efficient and fast but may have restrictions on which pins can be used. Software I2C is implemented by bit-banging and can be used on any pin but is not as efficient. These classes have the same methods available and differ primarily in the way they are constructed.

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 peripherals, returning a list of 7-bit␣
↪addresses
```

(continues on next page)

(continued from previous page)

```
i2c.writeto(42, b'123')         # write 3 bytes to peripheral with 7-bit address 42
i2c.readfrom(42, 4)             # read 4 bytes from peripheral with 7-bit address 42

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

Constructors

class machine.**I2C**(*id*, ***, *scl*, *sda*, *freq=400000*)

Construct and return a new I2C object using the following parameters:

- *id* identifies a particular I2C peripheral. Allowed values for depend on the particular port/board
- *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.

Note that some ports/boards will have default values of *scl* and *sda* that can be changed in this constructor. Others will have fixed values of *scl* and *sda* that cannot be changed.

class machine.**SoftI2C**(*scl*, *sda*, ***, *freq=400000*, *timeout=255*)

Construct a new software I2C object. The parameters are:

- *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.
- *timeout* is the maximum time in microseconds to wait for clock stretching (SCL held low by another device on the bus), after which an `OSError(ETIMEDOUT)` exception is raised.

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 controller 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.

These methods are only available on the *machine.SoftI2C* class.

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

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

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 peripheral assumes more bytes are going to be read in a later call).

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.

Standard bus operations

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

I2C.**readfrom**(*addr*, *nbytes*, *stop=True*, /)
: Read *nbytes* from the peripheral 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 peripheral 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 peripheral 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.

I2C.**writevto**(*addr*, *vector*, *stop=True*, /)
: Write the bytes contained in *vector* to the peripheral specified by *addr*. *vector* should be a tuple or list of objects with the buffer protocol. The *addr* is sent once and then the bytes from each object in *vector* are written out sequentially. The objects in *vector* may be zero bytes in length in which case they dont contribute to the output.

  If a NACK is received following the write of a byte from one of the objects in *vector* then the remaining bytes, and any remaining objects, 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 peripheral 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 peripheral 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 peripheral 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 peripheral 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 I2S – Inter-IC Sound bus protocol

I2S is a synchronous serial protocol used to connect digital audio devices. At the physical level, a bus consists of 3 lines: SCK, WS, SD. The I2S class supports controller operation. Peripheral operation is not supported.

The I2S class is currently available as a Technical Preview. During the preview period, feedback from users is encouraged. Based on this feedback, the I2S class API and implementation may be changed.

I2S objects can be created and initialized using:

```
from machine import I2S
from machine import Pin

# ESP32
sck_pin = Pin(14)   # Serial clock output
ws_pin = Pin(13)    # Word clock output
sd_pin = Pin(12)    # Serial data output

or

# PyBoards
sck_pin = Pin("Y6")   # Serial clock output
ws_pin = Pin("Y5")    # Word clock output
sd_pin = Pin("Y8")    # Serial data output

audio_out = I2S(2,
                sck=sck_pin, ws=ws_pin, sd=sd_pin,
                mode=I2S.TX,
                bits=16,
                format=I2S.MONO,
                rate=44100,
```

(continues on next page)

(continued from previous page)

```
                ibuf=20000)

audio_in = I2S(2,
               sck=sck_pin, ws=ws_pin, sd=sd_pin,
               mode=I2S.RX,
               bits=32,
               format=I2S.STEREO,
               rate=22050,
               ibuf=20000)
```

3 modes of operation are supported:

- blocking
- non-blocking
- uasyncio

blocking:

```
num_written = audio_out.write(buf) # blocks until buf emptied

num_read = audio_in.readinto(buf) # blocks until buf filled
```

non-blocking:

```
audio_out.irq(i2s_callback)          # i2s_callback is called when buf is emptied
num_written = audio_out.write(buf)  # returns immediately

audio_in.irq(i2s_callback)           # i2s_callback is called when buf is filled
num_read = audio_in.readinto(buf)   # returns immediately
```

uasyncio:

```
swriter = uasyncio.StreamWriter(audio_out)
swriter.write(buf)
await swriter.drain()

sreader = uasyncio.StreamReader(audio_in)
num_read = await sreader.readinto(buf)
```

Constructor

class machine.**I2S**(*id*, ***, *sck*, *ws*, *sd*, *mode*, *bits*, *format*, *rate*, *ibuf*)

Construct an I2S object of the given id:

- `id` identifies a particular I2S bus.

`id` is board and port specific:

- PYBv1.0/v1.1: has one I2S bus with id=2.
- PYBD-SFxW: has two I2S buses with id=1 and id=2.
- ESP32: has two I2S buses with id=0 and id=1.

Keyword-only parameters that are supported on all ports:

- `sck` is a pin object for the serial clock line
- `ws` is a pin object for the word select line
- `sd` is a pin object for the serial data line
- `mode` specifies receive or transmit
- `bits` specifies sample size (bits), 16 or 32
- `format` specifies channel format, STEREO or MONO
- `rate` specifies audio sampling rate (samples/s)
- `ibuf` specifies internal buffer length (bytes)

For all ports, DMA runs continuously in the background and allows user applications to perform other operations while sample data is transfered between the internal buffer and the I2S peripheral unit. Increasing the size of the internal buffer has the potential to increase the time that user applications can perform non-I2S operations before underflow (e.g. `write` method) or overflow (e.g. `readinto` method).

Methods

I2S.**init**(*sck*, ...)
see Constructor for argument descriptions

I2S.**deinit**()
Deinitialize the I2S bus

I2S.**readinto**(*buf*)
Read audio samples into the buffer specified by `buf`. `buf` must support the buffer protocol, such as bytearray or array. buf byte ordering is little-endian. For Stereo format, left channel sample precedes right channel sample. For Mono format, the left channel sample data is used. Returns number of bytes read

I2S.**write**(*buf*)
Write audio samples contained in `buf`. `buf` must support the buffer protocol, such as bytearray or array. buf byte ordering is little-endian. For Stereo format, left channel sample precedes right channel sample. For Mono format, the sample data is written to both the right and left channels. Returns number of bytes written

I2S.**irq**(*handler*)
Set a callback. `handler` is called when `buf` is emptied (`write` method) or becomes full (`readinto` method). Setting a callback changes the `write` and `readinto` methods to non-blocking operation. `handler` is called in the context of the MicroPython scheduler.

static I2S.**shift**(*, *buf*, *bits*, *shift*)
bitwise shift of all samples contained in `buf`. `bits` specifies sample size in bits. `shift` specifies the number of bits to shift each sample. Positive for left shift, negative for right shift. Typically used for volume control. Each bit shift changes sample volume by 6dB.

Constants

I2S.**RX**
for initialising the I2S bus `mode` to receive

I2S.**TX**
for initialising the I2S bus `mode` to transmit

I2S.**STEREO**
for initialising the I2S bus `format` to stereo

I2S.**MONO**
for initialising the I2S bus `format` to mono

class RTC – real time clock

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

Example usage:

```
rtc = machine.RTC()
rtc.datetime((2020, 1, 21, 2, 10, 32, 36, 0))
print(rtc.datetime())
```

Constructors

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

Methods

RTC.**datetime**([*datetimetuple*])
Get or set the date and time of the RTC.

With no arguments, this method returns an 8-tuple with the current date and time. With 1 argument (being an 8-tuple) it sets the date and time.

The 8-tuple has the following format:

(year, month, day, weekday, hours, minutes, seconds, subseconds)

The meaning of the `subseconds` field is hardware dependent.

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. MicroPythons 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 behaviour (which thus wont be portable to other boards).

See discussion of *important constraints* on Timer callbacks.

Note: Memory cant be allocated inside irq handlers (an interrupt) and so exceptions raised within a handler dont give much information. See *micropython.alloc_emergency_exception_buf()* for how to get around this limitation.

If you are using a WiPy board please refer to *machine.TimerWiPy* instead of this class.

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). `id` shall not be passed as a keyword argument.

See `init` for parameters of initialisation.

Methods

Timer.**init**(*, *mode=Timer.PERIODIC*, *period=- 1*, *callback=None*)
Initialise the timer. Example:

```
def mycallback(t):
    pass

# periodic with 100ms period
tim.init(period=100, callback=mycallback)

# one shot firing after 1000ms
tim.init(mode=Timer.ONE_SHOT, period=1000, callback=mycallback)
```

Keyword arguments:

- `mode` can be one of:
  - `Timer.ONE_SHOT` - The timer runs once until the configured period of the channel expires.
  - `Timer.PERIODIC` - The timer runs periodically at the configured frequency of the channel.
- `period` - The timer period, in milliseconds.
- `callback` - The callable to call upon expiration of the timer period. The callback must take one argument, which is passed the Timer object. The `callback` argument shall be specified. Otherwise an exception will occurr upon timer expiration: `TypeError: 'NoneType' object isn't callable`

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, esp8266, esp32.

Constructors

class machine.**WDT**(*id=0*, *timeout=5000*)

Create a WDT object and start it. The timeout must be given in milliseconds. Once it is running the timeout cannot be changed and the WDT cannot be stopped either.

Notes: On the esp32 the minimum timeout is 1 second. On the esp8266 a timeout cannot be specified, it is determined by the underlying system.

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.

class SD – secure digital memory card (cc3200 port only)

Warning: This is a non-standard class and is only available on the cc3200 port.

The SD card class allows to configure and enable the memory card module of the WiPy and automatically mount it as `/sd` as part of the file system. There are several pin combinations that can be used to wire the SD card socket to the WiPy and the pins used can be specified in the constructor. Please check the pinout and alternate functions table. for more info regarding the pins which can be remapped to be used with a SD card.

Example usage:

```
from machine import SD
import os
# clk cmd and dat0 pins must be passed along with
# their respective alternate functions
sd = machine.SD(pins=('GP10', 'GP11', 'GP15'))
os.mount(sd, '/sd')
# do normal file operations
```

Constructors

class machine.**SD**(*id*, ...)

Create a SD card object. See `init()` for parameters if initialization.

Methods

SD.**init**(*id=0*, *pins=('GP10', 'GP11', 'GP15')*)

Enable the SD card. In order to initialize the card, give it a 3-tuple: `(clk_pin, cmd_pin, dat0_pin)`.

SD.**deinit**()

Disable the SD card.

class SDCard – secure digital memory card

SD cards are one of the most common small form factor removable storage media. SD cards come in a variety of sizes and physical form factors. MMC cards are similar removable storage devices while eMMC devices are electrically similar storage devices designed to be embedded into other systems. All three form share a common protocol for communication with their host system and high-level support looks the same for them all. As such in MicroPython they are implemented in a single class called *machine.SDCard* .

Both SD and MMC interfaces support being accessed with a variety of bus widths. When being accessed with a 1-bit wide interface they can be accessed using the SPI protocol. Different MicroPython hardware platforms support different widths and pin configurations but for most platforms there is a standard configuration for any given hardware. In general constructing an `SDCard` object with without passing any parameters will initialise the interface to the default card slot for the current hardware. The arguments listed below represent the common arguments that might need to be set in order to use either a non-standard slot or a non-standard pin assignment. The exact subset of arguments supported will vary from platform to platform.

class machine.**SDCard**(*slot=1*, *width=1*, *cd=None*, *wp=None*, *sck=None*, *miso=None*, *mosi=None*, *cs=None*, *freq=20000000*)

This class provides access to SD or MMC storage cards using either a dedicated SD/MMC interface hardware or through an SPI channel. The class implements the block protocol defined by *os.AbstractBlockDev*. This allows the mounting of an SD card to be as simple as:

```
os.mount(machine.SDCard(), "/sd")
```

The constructor takes the following parameters:

- *slot* selects which of the available interfaces to use. Leaving this unset will select the default interface.
- *width* selects the bus width for the SD/MMC interface.
- *cd* can be used to specify a card-detect pin.
- *wp* can be used to specify a write-protect pin.
- *sck* can be used to specify an SPI clock pin.
- *miso* can be used to specify an SPI miso pin.
- *mosi* can be used to specify an SPI mosi pin.
- *cs* can be used to specify an SPI chip select pin.
- *freq* selects the SD/MMC interface frequency in Hz (only supported on the ESP32).

Implementation-specific details

Different implementations of the `SDCard` class on different hardware support varying subsets of the options above.

PyBoard

The standard PyBoard has just one slot. No arguments are necessary or supported.

ESP32

The ESP32 provides two channels of SD/MMC hardware and also supports access to SD Cards through either of the two SPI ports that are generally available to the user. As a result the *slot* argument can take a value between 0 and 3, inclusive. Slots 0 and 1 use the built-in SD/MMC hardware while slots 2 and 3 use the SPI ports. Slot 0 supports 1, 4 or 8-bit wide access while slot 1 supports 1 or 4-bit access; the SPI slots only support 1-bit access.

Note: Slot 0 is used to communicate with on-board flash memory on most ESP32 modules and so will be unavailable to the user.

Note: Most ESP32 modules that provide an SD card slot using the dedicated hardware only wire up 1 data pin, so the default value for *width* is 1.

The pins used by the dedicated SD/MMC hardware are fixed. The pins used by the SPI hardware can be reassigned.

Note: If any of the SPI signals are remapped then all of the SPI signals will pass through a GPIO multiplexer unit which can limit the performance of high frequency signals. Since the normal operating speed for SD cards is 40MHz this can cause problems on some cards.

The default (and preferred) pin assignment are as follows:

Slot	0	1	2	3
Signal	Pin	Pin	Pin	Pin
sck	6	14	18	14
cmd	11	15		
cs			5	15
miso			19	12
mosi			23	13
D0	7	2		
D1	8	4		
D2	9	12		
D3	10	13		
D4	16			
D5	17			
D6	5			
D7	18			

cc3200

You can set the pins used for SPI access by passing a tuple as the *pins* argument.

Note: The current cc3200 SD card implementation names the this class *machine.SD* rather than *machine.SDCard* .

1.2.6 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* module mainly so that scripts can be written which run under both CPython and MicroPython, by following the above pattern.

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

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

The optimisation level controls the following compilation features:

- Assertions: at level 0 assertion statements are enabled and compiled into the bytecode; at levels 1 and higher assertions are not compiled.
- Built-in `__debug__` variable: at level 0 this variable expands to `True`; at levels 1 and higher it expands to `False`.
- Source-code line numbers: at levels 0, 1 and 2 source-code line number are stored along with the bytecode so that exceptions can report the line number they occurred at; at levels 3 and higher line numbers are not stored.

The default optimisation level is usually level 0.

micropython.**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**()

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

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

Both *heap_unlock()* and *heap_locked()* return the current lock depth (after unlocking for the former) as a non-negative integer, with 0 meaning the heap is not locked.

If the REPL becomes active with the heap locked then it will be forcefully unlocked.

Note: *heap_locked()* is not enabled on most ports by default, requires `MICROPY_PY_MICROPYTHON_HEAP_LOCKED`.

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

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

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

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

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

Note: If *schedule()* is called from a preempting IRQ, when memory allocation is not allowed and the callback to be passed to *schedule()* is a bound method, passing this directly will fail. This is because creating a reference to a bound method causes memory allocation. A solution is to create a reference to the method in the class constructor and to pass that reference to *schedule()*. This is discussed in detail here *reference documentation* under Creation of Python objects.

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

1.2.7 neopixel control of WS2812 / NeoPixel LEDs

This module provides a driver for WS2818 / NeoPixel LEDs.

Note: This module is only included by default on the ESP8266 and ESP32 ports. On STM32 / Pyboard, you can download the module and copy it to the filesystem.

class NeoPixel

This class stores pixel data for a WS2812 LED strip connected to a pin. The application should set pixel data and then call *NeoPixel.write()* when it is ready to update the strip.

For example:

```
import neopixel

# 32 LED strip connected to X8.
p = machine.Pin.board.X8
n = neopixel.NeoPixel(p, 32)

# Draw a red gradient.
for i in range(32):
    n[i] = (i * 8, 0, 0)

# Update the strip.
n.write()
```

Constructors

class neopixel.**NeoPixel**(*pin*, *n*, *, *bpp=3*, *timing=1*)

Construct an NeoPixel object. The parameters are:

- *pin* is a machine.Pin instance.
- *n* is the number of LEDs in the strip.
- *bpp* is 3 for RGB LEDs, and 4 for RGBW LEDs.
- *timing* is 0 for 400KHz, and 1 for 800kHz LEDs (most are 800kHz).

Pixel access methods

NeoPixel.**fill**(*pixel*)
Sets the value of all pixels to the specified *pixel* value (i.e. an RGB/RGBW tuple).

NeoPixel.**__len__**()
Returns the number of LEDs in the strip.

NeoPixel.**__setitem__**(*index*, *val*)
Set the pixel at *index* to the value, which is an RGB/RGBW tuple.

NeoPixel.**__getitem__**(*index*)
Returns the pixel at *index* as an RGB/RGBW tuple.

Output methods

NeoPixel.**write**()
Writes the current pixel data to the strip.

1.2.8 `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 time
nic = network.Driver(...)
if not nic.isconnected():
    nic.connect()
    print("Waiting for connection...")
    while not nic.isconnected():
        time.sleep(1)
print(nic.ifconfig())

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

Common network adapter interface

This section describes an (implied) abstract base class for all network interface classes implemented by *MicroPython ports* 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*.

AbstractNIC.**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 (behaviour of calling them on inactive interface is undefined).

AbstractNIC.**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 identifier types; b) provide additional connection parameters. For various medium types, there are different sets of predefined/recommended parameters, among them:

  - WiFi: *bssid* keyword to connect to a specific BSSID (MAC address)

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

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

AbstractNIC.**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().

AbstractNIC.**status**([*param*])
: Query dynamic status information of the interface. When called with no argument the return value describes the network link status. Otherwise *param* should be a string naming the particular status parameter to retrieve.

  The return types and values are dependent on the network medium/technology. Some of the parameters that may be supported are:

  - WiFi STA: use `'rssi'` to retrieve the RSSI of the AP signal
  - WiFi AP: use `'stations'` to retrieve a list of all the STAs connected to the AP. The list contains tuples of the form (MAC, RSSI).

AbstractNIC.**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'))
```

AbstractNIC.**config**(*'param'*)
AbstractNIC.**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. 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'))
```

Specific network class implementations

The following concrete classes implement the AbstractNIC interface and provide a way to control networking interfaces of various kinds.

class WLAN – control built-in WiFi interfaces

This class provides a driver for WiFi network processors. Example usage:

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

Constructors

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

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

Methods

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

WLAN.**connect**(*ssid=None*, *password=None*, *, *bssid=None*)
Connect to the specified wireless network, using the specified password. If *bssid* is given then the connection will be restricted to the access-point with that MAC address (the *ssid* must also be specified in this case).

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

WLAN.**scan**()
Scan for the available wireless networks. Hidden networks – where the SSID is not broadcast – will also be scanned if the WLAN interface allows it.

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

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

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

There are five values for authmode:

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

and two for hidden:

- 0 – visible
- 1 – hidden

WLAN.**status**([*param*])
Return the current status of the wireless connection.

When called with no argument the return value describes the network link status. The possible statuses are defined as constants:

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

When called with one argument *param* should be a string naming the status parameter to retrieve. Supported parameters in WiFI STA mode are: `'rssi'`.

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

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

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

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

WLAN.**config**(*'param'*)

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

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

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

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

Parameter	Description
mac	MAC address (bytes)
essid	WiFi access point name (string)
channel	WiFi channel (integer)
hidden	Whether ESSID is hidden (boolean)
authmode	Authentication mode supported (enumeration, see module constants)
password	Access password (string)
dhcp_hostname	The DHCP hostname to use
reconnects	Number of reconnect attempts to make (integer, 0=none, -1=unlimited)

class WLANWiPy – WiPy specific WiFi control

Note: This class is a non-standard WLAN implementation for the WiPy. It is available simply as `network.WLAN` on the WiPy but is named in the documentation below as `network.WLANWiPy` to distinguish it from the more general *network.WLAN* class.

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

```
import network
import time
# setup as a station
wlan = network.WLAN(mode=WLAN.STA)
wlan.connect('your-ssid', auth=(WLAN.WPA2, 'your-key'))
while not wlan.isconnected():
    time.sleep_ms(50)
print(wlan.ifconfig())
```

(continues on next page)

(continued from previous page)

```
# now use socket as usual
...
```

Constructors

class network.**WLANWiPy**(*id=0*, ...)
Create a WLAN object, and optionally configure it. See *init()* for params of configuration.

Note: The WLAN constructor is special in the sense that if no arguments besides the id are given, it will return the already existing WLAN instance without re-configuring it. This is because WLAN is a system feature of the WiPy. If the already existing instance is not initialized it will do the same as the other constructors an will initialize it with default values.

Methods

WLANWiPy.**init**(*mode*, *, *ssid*, *auth*, *channel*, *antenna*)
Set or get the WiFi network processor configuration.

Arguments are:

- *mode* can be either WLAN.STA or WLAN.AP.
- *ssid* is a string with the ssid name. Only needed when mode is WLAN.AP.
- *auth* is a tuple with (sec, key). Security can be None, WLAN.WEP, WLAN.WPA or WLAN.WPA2. The key is a string with the network password. If sec is WLAN.WEP the key must be a string representing hexadecimal values (e.g. ABC1DE45BF). Only needed when mode is WLAN.AP.
- *channel* a number in the range 1-11. Only needed when mode is WLAN.AP.
- *antenna* selects between the internal and the external antenna. Can be either WLAN.INT_ANT or WLAN.EXT_ANT.

For example, you can do:

```
# create and configure as an access point
wlan.init(mode=WLAN.AP, ssid='wipy-wlan', auth=(WLAN.WPA2,'www.wipy.io'), channel=7,
↪ antenna=WLAN.INT_ANT)
```

or:

```
# configure as an station
wlan.init(mode=WLAN.STA)
```

WLANWiPy.**connect**(*ssid*, *, *auth=None*, *bssid=None*, *timeout=None*)
Connect to a WiFi access point using the given SSID, and other security parameters.

- *auth* is a tuple with (sec, key). Security can be None, WLAN.WEP, WLAN.WPA or WLAN.WPA2. The key is a string with the network password. If sec is WLAN.WEP the key must be a string representing hexadecimal values (e.g. ABC1DE45BF).
- *bssid* is the MAC address of the AP to connect to. Useful when there are several APs with the same ssid.

- *timeout* is the maximum time in milliseconds to wait for the connection to succeed.

WLANWiPy.**scan**()
Performs a network scan and returns a list of named tuples with (ssid, bssid, sec, channel, rssi). Note that channel is always `None` since this info is not provided by the WiPy.

WLANWiPy.**disconnect**()
Disconnect from the WiFi access point.

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

WLANWiPy.**ifconfig**(*if_id=0*, *config=['dhcp' or configtuple]*)
With no parameters given returns a 4-tuple of *(ip, subnet_mask, gateway, DNS_server)*.

if `'dhcp'` is passed as a parameter then the DHCP client is enabled and the IP params are negotiated with the AP.

If the 4-tuple config is given then a static IP is configured. For instance:

```
wlan.ifconfig(config=('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
```

WLANWiPy.**mode**([*mode*])
Get or set the WLAN mode.

WLANWiPy.**ssid**([*ssid*])
Get or set the SSID when in AP mode.

WLANWiPy.**auth**([*auth*])
Get or set the authentication type when in AP mode.

WLANWiPy.**channel**([*channel*])
Get or set the channel (only applicable in AP mode).

WLANWiPy.**antenna**([*antenna*])
Get or set the antenna type (external or internal).

WLANWiPy.**mac**([*mac_addr*])
Get or set a 6-byte long bytes object with the MAC address.

WLANWiPy.**irq**(*, *handler*, *wake*)
Create a callback to be triggered when a WLAN event occurs during `machine.SLEEP` mode. Events are triggered by socket activity or by WLAN connection/disconnection.

- *handler* is the function that gets called when the IRQ is triggered.
- *wake* must be `machine.SLEEP`.

Returns an IRQ object.

Constants

WLANWiPy.**STA**

WLANWiPy.**AP**
selects the WLAN mode

WLANWiPy.**WEP**

WLANWiPy.**WPA**

WLANWiPy.**WPA2**
selects the network security

WLANWiPy.**INT_ANT**

WLANWiPy.**EXT_ANT**
selects the antenna type

class CC3K – control CC3000 WiFi modules

This class provides a driver for CC3000 WiFi modules. Example usage:

```
import network
nic = network.CC3K(pyb.SPI(2), pyb.Pin.board.Y5, pyb.Pin.board.Y4, pyb.Pin.board.Y3)
nic.connect('your-ssid', 'your-password')
while not nic.isconnected():
    pyb.delay(50)
print(nic.ifconfig())

# now use socket as usual
...
```

For this example to work the CC3000 module must have the following connections:

- MOSI connected to Y8
- MISO connected to Y7
- CLK connected to Y6
- CS connected to Y5
- VBEN connected to Y4
- IRQ connected to Y3

It is possible to use other SPI buses and other pins for CS, VBEN and IRQ.

Constructors

class network.**CC3K**(*spi*, *pin_cs*, *pin_en*, *pin_irq*)
Create a CC3K driver object, initialise the CC3000 module using the given SPI bus and pins, and return the CC3K object.

Arguments are:

- *spi* is an *SPI object* which is the SPI bus that the CC3000 is connected to (the MOSI, MISO and CLK pins).
- *pin_cs* is a *Pin object* which is connected to the CC3000 CS pin.
- *pin_en* is a *Pin object* which is connected to the CC3000 VBEN pin.
- *pin_irq* is a *Pin object* which is connected to the CC3000 IRQ pin.

All of these objects will be initialised by the driver, so there is no need to initialise them yourself. For example, you can use:

```
nic = network.CC3K(pyb.SPI(2), pyb.Pin.board.Y5, pyb.Pin.board.Y4, pyb.Pin.board.Y3)
```

Methods

CC3K.**connect**(*ssid*, *key=None*, *, *security=WPA2*, *bssid=None*)
Connect to a WiFi access point using the given SSID, and other security parameters.

CC3K.**disconnect**()
Disconnect from the WiFi access point.

CC3K.**isconnected**()
Returns True if connected to a WiFi access point and has a valid IP address, False otherwise.

CC3K.**ifconfig**()
Returns a 7-tuple with (ip, subnet mask, gateway, DNS server, DHCP server, MAC address, SSID).

CC3K.**patch_version**()
Return the version of the patch program (firmware) on the CC3000.

CC3K.**patch_program**(*'pgm'*)
Upload the current firmware to the CC3000. You must pass pgm as the first argument in order for the upload to proceed.

Constants

CC3K.**WEP**

CC3K.**WPA**

CC3K.**WPA2**
security type to use

class WIZNET5K – control WIZnet5x00 Ethernet modules

This class allows you to control WIZnet5x00 Ethernet adaptors based on the W5200 and W5500 chipsets. The particular chipset that is supported by the firmware is selected at compile-time via the MICROPY_PY_WIZNET5K option.

Example usage:

```
import network
nic = network.WIZNET5K(pyb.SPI(1), pyb.Pin.board.X5, pyb.Pin.board.X4)
print(nic.ifconfig())

# now use socket as usual
...
```

For this example to work the WIZnet5x00 module must have the following connections:

- MOSI connected to X8
- MISO connected to X7
- SCLK connected to X6
- nSS connected to X5
- nRESET connected to X4

It is possible to use other SPI buses and other pins for nSS and nRESET.

Constructors

class network.**WIZNET5K**(*spi*, *pin_cs*, *pin_rst*)

Create a WIZNET5K driver object, initialise the WIZnet5x00 module using the given SPI bus and pins, and return the WIZNET5K object.

Arguments are:

- *spi* is an *SPI object* which is the SPI bus that the WIZnet5x00 is connected to (the MOSI, MISO and SCLK pins).
- *pin_cs* is a *Pin object* which is connected to the WIZnet5x00 nSS pin.
- *pin_rst* is a *Pin object* which is connected to the WIZnet5x00 nRESET pin.

All of these objects will be initialised by the driver, so there is no need to initialise them yourself. For example, you can use:

```
nic = network.WIZNET5K(pyb.SPI(1), pyb.Pin.board.X5, pyb.Pin.board.X4)
```

Methods

WIZNET5K.**isconnected**()

Returns `True` if the physical Ethernet link is connected and up. Returns `False` otherwise.

WIZNET5K.**ifconfig**([*(ip*, *subnet*, *gateway*, *dns)*])

Get/set IP address, subnet mask, gateway and DNS.

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'))
```

WIZNET5K.**regs**()

Dump the WIZnet5x00 registers. Useful for debugging.

Network functions

The following are functions available in the network module.

network.**phy_mode**([*mode*])

Get or set the PHY mode.

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

The possible modes are defined as constants:

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

Availability: ESP8266.

1.2.9 uctypes – access binary data in a structured way

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

Warning: `uctypes` module allows access to arbitrary memory addresses of the machine (including I/O and control registers). Uncareful usage of it may lead to crashes, data loss, and even hardware malfunction.

See also:

Module *`struct`* Standard Python way to access binary data structures (doesnt scale well to large and complex structures).

Usage examples:

```
import uctypes

# Example 1: Subset of ELF file header
# https://wikipedia.org/wiki/Executable_and_Linkable_Format#File_header
ELF_HEADER = {
    "EI_MAG": (0x0 | uctypes.ARRAY, 4 | uctypes.UINT8),
    "EI_DATA": 0x5 | uctypes.UINT8,
    "e_machine": 0x12 | uctypes.UINT16,
}

# "f" is an ELF file opened in binary mode
buf = f.read(uctypes.sizeof(ELF_HEADER, uctypes.LITTLE_ENDIAN))
header = uctypes.struct(uctypes.addressof(buf), ELF_HEADER, uctypes.LITTLE_ENDIAN)
assert header.EI_MAG == b"\x7fELF"
assert header.EI_DATA == 1, "Oops, wrong endianness. Could retry with uctypes.BIG_ENDIAN.
↪"
print("machine:", hex(header.e_machine))


# Example 2: In-memory data structure, with pointers
COORD = {
    "x": 0 | uctypes.FLOAT32,
    "y": 4 | uctypes.FLOAT32,
}

STRUCT1 = {
    "data1": 0 | uctypes.UINT8,
    "data2": 4 | uctypes.UINT32,
    "ptr": (8 | uctypes.PTR, COORD),
}

# Suppose you have address of a structure of type STRUCT1 in "addr"
# uctypes.NATIVE is optional (used by default)
struct1 = uctypes.struct(addr, STRUCT1, uctypes.NATIVE)
print("x:", struct1.ptr[0].x)
```

(continues on next page)

(continued from previous page)

```
# Example 3: Access to CPU registers. Subset of STM32F4xx WWDG block
WWDG_LAYOUT = {
    "WWDG_CR": (0, {
        # BFUINT32 here means size of the WWDG_CR register
        "WDGA": 7 << uctypes.BF_POS | 1 << uctypes.BF_LEN | uctypes.BFUINT32,
        "T": 0 << uctypes.BF_POS | 7 << uctypes.BF_LEN | uctypes.BFUINT32,
    }),
    "WWDG_CFR": (4, {
        "EWI": 9 << uctypes.BF_POS | 1 << uctypes.BF_LEN | uctypes.BFUINT32,
        "WDGTB": 7 << uctypes.BF_POS | 2 << uctypes.BF_LEN | uctypes.BFUINT32,
        "W": 0 << uctypes.BF_POS | 7 << uctypes.BF_LEN | uctypes.BFUINT32,
    }),
}

WWDG = uctypes.struct(0x40002c00, WWDG_LAYOUT)

WWDG.WWDG_CFR.WDGTB = 0b10
WWDG.WWDG_CR.WDGA = 1
print("Current counter:", WWDG.WWDG_CR.T)
```

Defining structure layout

Structure layout is defined by a descriptor - a Python dictionary which encodes field names as keys and other properties required to access them as associated values:

```
{
    "field1": <properties>,
    "field2": <properties>,
    ...
}
```

Currently, `uctypes` requires explicit specification of offsets for each field. Offset are given in bytes from the structure start.

Following are encoding examples for various field types:

- Scalar types:

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

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

- Recursive structures:

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

  i.e. value is a 2-tuple, first element of which is an offset, and second is a structure descriptor dictionary (note: offsets in recursive descriptors are relative to the structure it defines). Of course, recursive structures can be specified not just by a literal dictionary, but by referring to a structure descriptor dictionary (defined earlier) by name.

- Arrays of primitive types:

```
"arr": (offset | uctypes.ARRAY, size | uctypes.UINT8),
```

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

- Arrays of aggregate types:

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

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

- Pointer to a primitive type:

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

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

- Pointer to an aggregate type:

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

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

- Bitfields:

```
"bitf0": offset | uctypes.BFUINT16 | lsbit << uctypes.BF_POS | bitsize << uctypes.
↪BF_LEN,
```

  i.e. value is a type of scalar value containing given bitfield (typenames are similar to scalar types, but prefixes with `BF`), ORed with offset for scalar value containing the bitfield, and further ORed with values for bit position and bit length of the bitfield within the scalar value, shifted by BF_POS and BF_LEN bits, respectively. A bitfield position is counted from the least significant bit of the scalar (having position of 0), and is the number of right-most bit of a field (in other words, its a number of bits a scalar needs to be shifted right to extract the bitfield).

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

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

Module contents

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

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

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

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

uctypes.**sizeof**(*struct*, *layout_type=NATIVE*, */*)
Return size of data structure in bytes. The *struct* argument can be either a structure class or a specific instantiated structure object (or its aggregate field).

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

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

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

uctypes.**UINT8**
uctypes.**INT8**
uctypes.**UINT16**
uctypes.**INT16**
uctypes.**UINT32**
uctypes.**INT32**
uctypes.**UINT64**
uctypes.**INT64**
Integer types for structure descriptors. Constants for 8, 16, 32, and 64 bit types are provided, both signed and unsigned.

uctypes.**FLOAT32**
uctypes.**FLOAT64**
Floating-point types for structure descriptors.

uctypes.**VOID**
`VOID` is an alias for `UINT8`, and is provided to conveniently define Cs void pointers: `(uctypes.PTR, uctypes.VOID)`.

uctypes.**PTR**
uctypes.**ARRAY**
Type constants for pointers and arrays. Note that there is no explicit constant for structures, its implicit: an aggregate type without `PTR` or `ARRAY` flags is a structure.

Structure descriptors and instantiating structure objects

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

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

Structure objects

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

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

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

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

Limitations

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

- Avoid accessing nested structures. For example, instead of `mcu_registers.peripheral_a.register1`, define separate layout descriptors for each peripheral, to be accessed as `peripheral_a.register1`. Or just cache a particular peripheral: `peripheral_a = mcu_registers.peripheral_a`. If a register consists of multiple bitfields, you would need to cache references to a particular register: `reg_a = mcu_registers.peripheral_a.reg_a`.
- Avoid other non-scalar data, like arrays. For example, instead of `peripheral_a.register[0]` use `peripheral_a.register0`. Again, an alternative is to cache intermediate values, e.g. `register0 = peripheral_a.register[0]`.

2. Range of offsets supported by the `uctypes` module is limited. The exact range supported is considered an implementation detail, and the general suggestion is to split structure definitions to cover from a few kilobytes to a few dozen of kilobytes maximum. In most cases, this is a natural situation anyway, e.g. it doesnt make sense to define all registers of an MCU (spread over 32-bit address space) in one structure, but rather a peripheral block by peripheral block. In some extreme cases, you may need to split a structure in several parts artificially (e.g. if accessing native data structure with multi-megabyte array in the middle, though that would be a very synthetic case).

1.3 Port-specific libraries

In some cases the following port/board-specific libraries have functions or classes similar to those in the *machine* library. Where this occurs, the entry in the port specific library exposes hardware functionality unique to that platform.

To write portable code use functions and classes from the *machine* module. To access platform-specific hardware use the appropriate library, e.g. *pyb* in the case of the Pyboard.

1.3.1 Libraries specific to the pyboard

The following libraries are specific to the pyboard.

pyb functions related to the board

The pyb module contains specific functions related to the board.

Time related functions

pyb.**delay**(*ms*)
Delay for the given number of milliseconds.

pyb.**udelay**(*us*)
Delay for the given number of microseconds.

pyb.**millis**()
Returns the number of milliseconds since the board was last reset.

The result is always a MicroPython smallint (31-bit signed number), so after 2^30 milliseconds (about 12.4 days) this will start to return negative numbers.

Note that if *pyb.stop()* is issued the hardware counter supporting this function will pause for the duration of the sleeping state. This will affect the outcome of *pyb.elapsed_millis()*.

pyb.**micros**()
Returns the number of microseconds since the board was last reset.

The result is always a MicroPython smallint (31-bit signed number), so after 2^30 microseconds (about 17.8 minutes) this will start to return negative numbers.

Note that if *pyb.stop()* is issued the hardware counter supporting this function will pause for the duration of the sleeping state. This will affect the outcome of *pyb.elapsed_micros()*.

pyb.**elapsed_millis**(*start*)
Returns the number of milliseconds which have elapsed since `start`.

This function takes care of counter wrap, and always returns a positive number. This means it can be used to measure periods up to about 12.4 days.

Example:

```
start = pyb.millis()
while pyb.elapsed_millis(start) < 1000:
    # Perform some operation
```

pyb.**elapsed_micros**(*start*)
Returns the number of microseconds which have elapsed since `start`.

This function takes care of counter wrap, and always returns a positive number. This means it can be used to measure periods up to about 17.8 minutes.

Example:

```
start = pyb.micros()
while pyb.elapsed_micros(start) < 1000:
    # Perform some operation
    pass
```

Reset related functions

pyb.**hard_reset**()
Resets the pyboard in a manner similar to pushing the external RESET button.

pyb.**bootloader**()
Activate the bootloader without BOOT* pins.

pyb.**fault_debug**(*value*)
Enable or disable hard-fault debugging. A hard-fault is when there is a fatal error in the underlying system, like an invalid memory access.

If the *value* argument is `False` then the board will automatically reset if there is a hard fault.

If *value* is `True` then, when the board has a hard fault, it will print the registers and the stack trace, and then cycle the LEDs indefinitely.

The default value is disabled, i.e. to automatically reset.

Interrupt related functions

pyb.**disable_irq**()
Disable interrupt requests. Returns the previous IRQ state: `False`/`True` for disabled/enabled IRQs respectively. This return value can be passed to enable_irq to restore the IRQ to its original state.

pyb.**enable_irq**(*state=True*)
Enable interrupt requests. If `state` is `True` (the default value) then IRQs are enabled. If `state` is `False` then IRQs are disabled. The most common use of this function is to pass it the value returned by `disable_irq` to exit a critical section.

Power related functions

pyb.**freq**([*sysclk*[, *hclk*[, *pclk1*[, *pclk2*]]]])
If given no arguments, returns a tuple of clock frequencies: (sysclk, hclk, pclk1, pclk2). These correspond to:

- sysclk: frequency of the CPU
- hclk: frequency of the AHB bus, core memory and DMA
- pclk1: frequency of the APB1 bus
- pclk2: frequency of the APB2 bus

If given any arguments then the function sets the frequency of the CPU, and the buses if additional arguments are given. Frequencies are given in Hz. Eg freq(120000000) sets sysclk (the CPU frequency) to 120MHz. Note that not all values are supported and the largest supported frequency not greater than the given value will be selected.

Supported sysclk frequencies are (in MHz): 8, 16, 24, 30, 32, 36, 40, 42, 48, 54, 56, 60, 64, 72, 84, 96, 108, 120, 144, 168.

The maximum frequency of hclk is 168MHz, of pclk1 is 42MHz, and of pclk2 is 84MHz. Be sure not to set frequencies above these values.

The hclk, pclk1 and pclk2 frequencies are derived from the sysclk frequency using a prescaler (divider). Supported prescalers for hclk are: 1, 2, 4, 8, 16, 64, 128, 256, 512. Supported prescalers for pclk1 and pclk2 are: 1, 2, 4, 8. A prescaler will be chosen to best match the requested frequency.

A sysclk frequency of 8MHz uses the HSE (external crystal) directly and 16MHz uses the HSI (internal oscillator) directly. The higher frequencies use the HSE to drive the PLL (phase locked loop), and then use the output of the PLL.

Note that if you change the frequency while the USB is enabled then the USB may become unreliable. It is best to change the frequency in boot.py, before the USB peripheral is started. Also note that sysclk frequencies below 36MHz do not allow the USB to function correctly.

pyb.**wfi**()
Wait for an internal or external interrupt.

This executes a `wfi` instruction which reduces power consumption of the MCU until any interrupt occurs (be it internal or external), at which point execution continues. Note that the system-tick interrupt occurs once every millisecond (1000Hz) so this function will block for at most 1ms.

pyb.**stop**()
Put the pyboard in a sleeping state.

This reduces power consumption to less than 500 uA. To wake from this sleep state requires an external interrupt or a real-time-clock event. Upon waking execution continues where it left off.

See `rtc.wakeup()` to configure a real-time-clock wakeup event.

pyb.**standby**()
Put the pyboard into a deep sleep state.

This reduces power consumption to less than 50 uA. To wake from this sleep state requires a real-time-clock event, or an external interrupt on X1 (PA0=WKUP) or X18 (PC13=TAMP1). Upon waking the system undergoes a hard reset.

See `rtc.wakeup()` to configure a real-time-clock wakeup event.

Miscellaneous functions

pyb.**have_cdc**()
Return True if USB is connected as a serial device, False otherwise.

Note: This function is deprecated. Use pyb.USB_VCP().isconnected() instead.

pyb.**hid**(*(buttons, x, y, z)*)
Takes a 4-tuple (or list) and sends it to the USB host (the PC) to signal a HID mouse-motion event.

Note: This function is deprecated. Use *pyb.USB_HID.send()* instead.

pyb.**info**([*dump_alloc_table*])
Print out lots of information about the board.

pyb.**main**(*filename*)
Set the filename of the main script to run after boot.py is finished. If this function is not called then the default file main.py will be executed.

It only makes sense to call this function from within boot.py.

pyb.**mount**(*device*, *mountpoint*, *, *readonly=False*, *mkfs=False*)

Note: This function is deprecated. Mounting and unmounting devices should be performed by *os.mount()* and *os.umount()* instead.

Mount a block device and make it available as part of the filesystem. `device` must be an object that provides the block protocol. (The following is also deprecated. See *os.AbstractBlockDev* for the correct way to create a block device.)

- `readblocks(self, blocknum, buf)`
- `writeblocks(self, blocknum, buf)` (optional)
- `count(self)`
- `sync(self)` (optional)

`readblocks` and `writeblocks` should copy data between `buf` and the block device, starting from block number `blocknum` on the device. `buf` will be a bytearray with length a multiple of 512. If `writeblocks` is not defined then the device is mounted read-only. The return value of these two functions is ignored.

`count` should return the number of blocks available on the device. `sync`, if implemented, should sync the data on the device.

The parameter `mountpoint` is the location in the root of the filesystem to mount the device. It must begin with a forward-slash.

If `readonly` is `True`, then the device is mounted read-only, otherwise it is mounted read-write.

If `mkfs` is `True`, then a new filesystem is created if one does not already exist.

pyb.**repl_uart**(*uart*)
Get or set the UART object where the REPL is repeated on.

pyb.**rng**()
Return a 30-bit hardware generated random number.

pyb.**sync**()
Sync all file systems.

pyb.**unique_id**()
Returns a string of 12 bytes (96 bits), which is the unique ID of the MCU.

pyb.**usb_mode**([*modestr*], *port=-1*, *vid=0xf055*, *pid=-1*, *msc=()*, *hid=pyb.hid_mouse*, *high_speed=False*)
If called with no arguments, return the current USB mode as a string.

If called with *modestr* provided, attempts to configure the USB mode. The following values of *modestr* are understood:

- `None`: disables USB
- `'VCP'`: enable with VCP (Virtual COM Port) interface

- `'MSC'`: enable with MSC (mass storage device class) interface
- `'VCP+MSC'`: enable with VCP and MSC
- `'VCP+HID'`: enable with VCP and HID (human interface device)
- `'VCP+MSC+HID'`: enabled with VCP, MSC and HID (only available on PYBD boards)

For backwards compatibility, `'CDC'` is understood to mean `'VCP'` (and similarly for `'CDC+MSC'` and `'CDC+HID'`).

The *port* parameter should be an integer (0, 1,) and selects which USB port to use if the board supports multiple ports. A value of -1 uses the default or automatically selected port.

The *vid* and *pid* parameters allow you to specify the VID (vendor id) and PID (product id). A *pid* value of -1 will select a PID based on the value of *modestr*.

If enabling MSC mode, the *msc* parameter can be used to specify a list of SCSI LUNs to expose on the mass storage interface. For example `msc=(pyb.Flash(), pyb.SDCard())`.

If enabling HID mode, you may also specify the HID details by passing the *hid* keyword parameter. It takes a tuple of (subclass, protocol, max packet length, polling interval, report descriptor). By default it will set appropriate values for a USB mouse. There is also a `pyb.hid_keyboard` constant, which is an appropriate tuple for a USB keyboard.

The *high_speed* parameter, when set to `True`, enables USB HS mode if it is supported by the hardware.

Classes

class Accel – accelerometer control

Accel is an object that controls the accelerometer. Example usage:

```
accel = pyb.Accel()
for i in range(10):
    print(accel.x(), accel.y(), accel.z())
```

Raw values are between -32 and 31.

Constructors

class pyb.**Accel**
Create and return an accelerometer object.

Methods

Accel.**filtered_xyz**()
Get a 3-tuple of filtered x, y and z values.

Implementation note: this method is currently implemented as taking the sum of 4 samples, sampled from the 3 previous calls to this function along with the sample from the current call. Returned values are therefore 4 times the size of what they would be from the raw x(), y() and z() calls.

Accel.**tilt**()
Get the tilt register.

Accel.**x**()
Get the x-axis value.

Accel.**y**()
Get the y-axis value.

Accel.**z**()
Get the z-axis value.

Hardware Note

The accelerometer uses I2C bus 1 to communicate with the processor. Consequently when readings are being taken pins X9 and X10 should be unused (other than for I2C). Other devices using those pins, and which therefore cannot be used concurrently, are UART 1 and Timer 4 channels 1 and 2.

class ADC – analog to digital conversion

Usage:

```
import pyb

adc = pyb.ADC(pin)                  # create an analog object from a pin
val = adc.read()                    # read an analog value

adc = pyb.ADCAll(resolution)        # create an ADCAll object
adc = pyb.ADCAll(resolution, mask)  # create an ADCAll object for selected analog␣
↪channels
val = adc.read_channel(channel)     # read the given channel
val = adc.read_core_temp()          # read MCU temperature
val = adc.read_core_vbat()          # read MCU VBAT
val = adc.read_core_vref()          # read MCU VREF
val = adc.read_vref()               # read MCU supply voltage
```

Constructors

class pyb.**ADC**(*pin*)
Create an ADC object associated with the given pin. This allows you to then read analog values on that pin.

Methods

ADC.**read**()
Read the value on the analog pin and return it. The returned value will be between 0 and 4095.

ADC.**read_timed**(*buf*, *timer*)
Read analog values into `buf` at a rate set by the `timer` object.

`buf` can be bytearray or array.array for example. The ADC values have 12-bit resolution and are stored directly into `buf` if its element size is 16 bits or greater. If `buf` has only 8-bit elements (eg a bytearray) then the sample resolution will be reduced to 8 bits.

`timer` should be a Timer object, and a sample is read each time the timer triggers. The timer must already be initialised and running at the desired sampling frequency.

To support previous behaviour of this function, `timer` can also be an integer which specifies the frequency (in Hz) to sample at. In this case Timer(6) will be automatically configured to run at the given frequency.

Example using a Timer object (preferred way):

```
adc = pyb.ADC(pyb.Pin.board.X19)    # create an ADC on pin X19
tim = pyb.Timer(6, freq=10)         # create a timer running at 10Hz
buf = bytearray(100)                # creat a buffer to store the samples
adc.read_timed(buf, tim)            # sample 100 values, taking 10s
```

Example using an integer for the frequency:

```
adc = pyb.ADC(pyb.Pin.board.X19)    # create an ADC on pin X19
buf = bytearray(100)                # create a buffer of 100 bytes
adc.read_timed(buf, 10)             # read analog values into buf at 10Hz
                                    #   this will take 10 seconds to finish
for val in buf:                     # loop over all values
    print(val)                      # print the value out
```

This function does not allocate any heap memory. It has blocking behaviour: it does not return to the calling program until the buffer is full.

ADC.**read_timed_multi**(*(adcx, adcy, ...)*, *(bufx, bufy, ...)*, *timer*)

This is a static method. It can be used to extract relative timing or phase data from multiple ADCs.

It reads analog values from multiple ADCs into buffers at a rate set by the *timer* object. Each time the timer triggers a sample is rapidly read from each ADC in turn.

ADC and buffer instances are passed in tuples with each ADC having an associated buffer. All buffers must be of the same type and length and the number of buffers must equal the number of ADCs.

Buffers can be `bytearray` or `array.array` for example. The ADC values have 12-bit resolution and are stored directly into the buffer if its element size is 16 bits or greater. If buffers have only 8-bit elements (eg a `bytearray`) then the sample resolution will be reduced to 8 bits.

timer must be a Timer object. The timer must already be initialised and running at the desired sampling frequency.

Example reading 3 ADCs:

```
adc0 = pyb.ADC(pyb.Pin.board.X1)    # Create ADC's
adc1 = pyb.ADC(pyb.Pin.board.X2)
adc2 = pyb.ADC(pyb.Pin.board.X3)
tim = pyb.Timer(8, freq=100)        # Create timer
rx0 = array.array('H', (0 for i in range(100))) # ADC buffers of
rx1 = array.array('H', (0 for i in range(100))) # 100 16-bit words
rx2 = array.array('H', (0 for i in range(100)))
# read analog values into buffers at 100Hz (takes one second)
pyb.ADC.read_timed_multi((adc0, adc1, adc2), (rx0, rx1, rx2), tim)
for n in range(len(rx0)):
    print(rx0[n], rx1[n], rx2[n])
```

This function does not allocate any heap memory. It has blocking behaviour: it does not return to the calling program until the buffers are full.

The function returns `True` if all samples were acquired with correct timing. At high sample rates the time taken to acquire a set of samples can exceed the timer period. In this case the function returns `False`, indicating a loss of precision in the sample interval. In extreme cases samples may be missed.

The maximum rate depends on factors including the data width and the number of ADCs being read. In testing two ADCs were sampled at a timer rate of 210kHz without overrun. Samples were missed at 215kHz. For three ADCs the limit is around 140kHz, and for four it is around 110kHz. At high sample rates disabling interrupts for the duration can reduce the risk of sporadic data loss.

The ADCAll Object

Instantiating this changes all masked ADC pins to analog inputs. The preprocessed MCU temperature, VREF and VBAT data can be accessed on ADC channels 16, 17 and 18 respectively. Appropriate scaling is handled according to reference voltage used (usually 3.3V). The temperature sensor on the chip is factory calibrated and allows to read the die temperature to +/- 1 degree centigrade. Although this sounds pretty accurate, dont forget that the MCUs internal temperature is measured. Depending on processing loads and I/O subsystems active the die temperature may easily be tens of degrees above ambient temperature. On the other hand a pyboard woken up after a long standby period will show correct ambient temperature within limits mentioned above.

The `ADCAll` `read_core_vbat()`, `read_vref()` and `read_core_vref()` methods read the backup battery voltage, reference voltage and the (1.21V nominal) reference voltage using the actual supply as a reference. All results are floating point numbers giving direct voltage values.

`read_core_vbat()` returns the voltage of the backup battery. This voltage is also adjusted according to the actual supply voltage. To avoid analog input overload the battery voltage is measured via a voltage divider and scaled according to the divider value. To prevent excessive loads to the backup battery, the voltage divider is only active during ADC conversion.

`read_vref()` is evaluated by measuring the internal voltage reference and backscale it using factory calibration value of the internal voltage reference. In most cases the reading would be close to 3.3V. If the pyboard is operated from a battery, the supply voltage may drop to values below 3.3V. The pyboard will still operate fine as long as the operating conditions are met. With proper settings of MCU clock, flash access speed and programming mode it is possible to run the pyboard down to 2 V and still get useful ADC conversion.

It is very important to make sure analog input voltages never exceed actual supply voltage.

Other analog input channels (0..15) will return unscaled integer values according to the selected precision.

To avoid unwanted activation of analog inputs (channel 0..15) a second parameter can be specified. This parameter is a binary pattern where each requested analog input has the corresponding bit set. The default value is 0xffffffff which means all analog inputs are active. If just the internal channels (16..18) are required, the mask value should be 0x70000.

Example:

```
adcall = pyb.ADCAll(12, 0x70000) # 12 bit resolution, internal channels
temp = adcall.read_core_temp()
```

class CAN – controller area network communication bus

CAN implements the standard CAN communications protocol. At the physical level it consists of 2 lines: RX and TX. Note that to connect the pyboard to a CAN bus you must use a CAN transceiver to convert the CAN logic signals from the pyboard to the correct voltage levels on the bus.

Example usage (works without anything connected):

```
from pyb import CAN
can = CAN(1, CAN.LOOPBACK)
can.setfilter(0, CAN.LIST16, 0, (123, 124, 125, 126))  # set a filter to receive␣
↪messages with id=123, 124, 125 and 126
```

(continues on next page)

(continued from previous page)

```
can.send('message!', 123)   # send a message with id 123
can.recv(0)                 # receive message on FIFO 0
```

Constructors

class pyb.**CAN**(*bus*, ...)

Construct a CAN object on the given bus. *bus* can be 1-2, or `'YA'` or `'YB'`. With no additional parameters, the CAN 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 *CAN.init()* for parameters of initialisation.

The physical pins of the CAN buses are:

- `CAN(1)` is on `YA`: `(RX, TX) = (Y3, Y4) = (PB8, PB9)`
- `CAN(2)` is on `YB`: `(RX, TX) = (Y5, Y6) = (PB12, PB13)`

Class Methods

classmethod CAN.**initfilterbanks**(*nr*)

Reset and disable all filter banks and assign how many banks should be available for CAN(1).

STM32F405 has 28 filter banks that are shared between the two available CAN bus controllers. This function configures how many filter banks should be assigned to each. *nr* is the number of banks that will be assigned to CAN(1), the rest of the 28 are assigned to CAN(2). At boot, 14 banks are assigned to each controller.

Methods

CAN.**init**(*mode*, *extframe=False*, *prescaler=100*, ***, *sjw=1*, *bs1=6*, *bs2=8*, *auto_restart=False*, *baudrate=0*, *sample_point=75*)

Initialise the CAN bus with the given parameters:

- *mode* is one of: NORMAL, LOOPBACK, SILENT, SILENT_LOOPBACK
- if *extframe* is True then the bus uses extended identifiers in the frames (29 bits); otherwise it uses standard 11 bit identifiers
- *prescaler* is used to set the duration of 1 time quanta; the time quanta will be the input clock (PCLK1, see *pyb.freq()*) divided by the prescaler
- *sjw* is the resynchronisation jump width in units of the time quanta; it can be 1, 2, 3, 4
- *bs1* defines the location of the sample point in units of the time quanta; it can be between 1 and 1024 inclusive
- *bs2* defines the location of the transmit point in units of the time quanta; it can be between 1 and 16 inclusive
- *auto_restart* sets whether the controller will automatically try and restart communications after entering the bus-off state; if this is disabled then *restart()* can be used to leave the bus-off state
- *baudrate* if a baudrate other than 0 is provided, this function will try to automatically calculate a CAN bit-timing (overriding *prescaler*, *bs1* and *bs2*) that satisfies both the baudrate and the desired *sample_point*.
- *sample_point* given in a percentage of the bit time, the *sample_point* specifies the position of the last bit sample with respect to the whole bit time. The default *sample_point* is 75%.

The time quanta tq is the basic unit of time for the CAN bus. tq is the CAN prescaler value divided by PCLK1 (the frequency of internal peripheral bus 1); see *pyb.freq()* to determine PCLK1.

A single bit is made up of the synchronisation segment, which is always 1 tq. Then follows bit segment 1, then bit segment 2. The sample point is after bit segment 1 finishes. The transmit point is after bit segment 2 finishes. The baud rate will be 1/bittime, where the bittime is 1 + BS1 + BS2 multiplied by the time quanta tq.

For example, with PCLK1=42MHz, prescaler=100, sjw=1, bs1=6, bs2=8, the value of tq is 2.38 microseconds. The bittime is 35.7 microseconds, and the baudrate is 28kHz.

See page 680 of the STM32F405 datasheet for more details.

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

CAN.**restart**()
Force a software restart of the CAN controller without resetting its configuration.

If the controller enters the bus-off state then it will no longer participate in bus activity. If the controller is not configured to automatically restart (see *init()*) then this method can be used to trigger a restart, and the controller will follow the CAN protocol to leave the bus-off state and go into the error active state.

CAN.**state**()
Return the state of the controller. The return value can be one of:

- CAN.STOPPED – the controller is completely off and reset;
- CAN.ERROR_ACTIVE – the controller is on and in the Error Active state (both TEC and REC are less than 96);
- CAN.ERROR_WARNING – the controller is on and in the Error Warning state (at least one of TEC or REC is 96 or greater);
- CAN.ERROR_PASSIVE – the controller is on and in the Error Passive state (at least one of TEC or REC is 128 or greater);
- CAN.BUS_OFF – the controller is on but not participating in bus activity (TEC overflowed beyond 255).

CAN.**info**([*list*])
Get information about the controllers error states and TX and RX buffers. If *list* is provided then it should be a list object with at least 8 entries, which will be filled in with the information. Otherwise a new list will be created and filled in. In both cases the return value of the method is the populated list.

The values in the list are:

- TEC value
- REC value
- number of times the controller enterted the Error Warning state (wrapped around to 0 after 65535)
- number of times the controller enterted the Error Passive state (wrapped around to 0 after 65535)
- number of times the controller enterted the Bus Off state (wrapped around to 0 after 65535)
- number of pending TX messages
- number of pending RX messages on fifo 0
- number of pending RX messages on fifo 1

CAN.**setfilter**(*bank*, *mode*, *fifo*, *params*, *, *rtr*)
Configure a filter bank:

- *bank* is the filter bank that is to be configured.

- *mode* is the mode the filter should operate in.
- *fifo* is which fifo (0 or 1) a message should be stored in, if it is accepted by this filter.
- *params* is an array of values the defines the filter. The contents of the array depends on the *mode* argument.

mode	contents of *params* array
CAN.LIST16	Four 16 bit ids that will be accepted
CAN.LIST32	Two 32 bit ids that will be accepted
CAN.MASK16	**Two 16 bit id/mask pairs. E.g. (1, 3, 4, 4)** The first pair, 1 and 3 will accept all ids that have bit 0 = 1 and bit 1 = 0. The second pair, 4 and 4, will accept all ids that have bit 2 = 1.
CAN.MASK32	As with CAN.MASK16 but with only one 32 bit id/mask pair.

- *rtr* is an array of booleans that states if a filter should accept a remote transmission request message. If this argument is not given then it defaults to `False` for all entries. The length of the array depends on the *mode* argument.

mode	length of *rtr* array
CAN.LIST16	4
CAN.LIST32	2
CAN.MASK16	2
CAN.MASK32	1

`CAN.clearfilter`(*bank*)

Clear and disables a filter bank:

- *bank* is the filter bank that is to be cleared.

`CAN.any`(*fifo*)

Return `True` if any message waiting on the FIFO, else `False`.

`CAN.recv`(*fifo*, *list=None*, *, *timeout=5000*)

Receive data on the bus:

- *fifo* is an integer, which is the FIFO to receive on
- *list* is an optional list object to be used as the return value
- *timeout* is the timeout in milliseconds to wait for the receive.

Return value: A tuple containing four values.

- The id of the message.
- A boolean that indicates if the message is an RTR message.
- The FMI (Filter Match Index) value.
- An array containing the data.

If *list* is `None` then a new tuple will be allocated, as well as a new bytes object to contain the data (as the fourth element in the tuple).

If *list* is not `None` then it should be a list object with a least four elements. The fourth element should be a memoryview object which is created from either a bytearray or an array of type B or b, and this array must have enough room for at least 8 bytes. The list object will then be populated with the first three return values above, and the memoryview object will be resized inplace to the size of the data and filled in with that data. The same list and memoryview objects can be reused in subsequent calls to this method, providing a way of receiving data without using the heap. For example:

```
buf = bytearray(8)
lst = [0, 0, 0, memoryview(buf)]
# No heap memory is allocated in the following call
can.recv(0, lst)
```

`CAN.`**`send`**(*data*, *id*, *, *timeout=0*, *rtr=False*)

Send a message on the bus:

- *data* is the data to send (an integer to send, or a buffer object).
- *id* is the id of the message to be sent.
- *timeout* is the timeout in milliseconds to wait for the send.
- *rtr* is a boolean that specifies if the message shall be sent as a remote transmission request. If *rtr* is True then only the length of *data* is used to fill in the DLC slot of the frame; the actual bytes in *data* are unused.

If timeout is 0 the message is placed in a buffer in one of three hardware buffers and the method returns immediately. If all three buffers are in use an exception is thrown. If timeout is not 0, the method waits until the message is transmitted. If the message cant be transmitted within the specified time an exception is thrown.

Return value: `None`.

`CAN.`**`rxcallback`**(*fifo*, *fun*)

Register a function to be called when a message is accepted into a empty fifo:

- *fifo* is the receiving fifo.
- *fun* is the function to be called when the fifo becomes non empty.

The callback function takes two arguments the first is the can object it self the second is a integer that indicates the reason for the callback.

Reason	
0	A message has been accepted into a empty FIFO.
1	The FIFO is full
2	A message has been lost due to a full FIFO

Example use of rxcallback:

```
def cb0(bus, reason):
  print('cb0')
  if reason == 0:
      print('pending')
  if reason == 1:
      print('full')
  if reason == 2:
      print('overflow')
```

(continues on next page)

(continued from previous page)

```
can = CAN(1, CAN.LOOPBACK)
can.rxcallback(0, cb0)
```

Constants

CAN.**NORMAL**
CAN.**LOOPBACK**
CAN.**SILENT**
CAN.**SILENT_LOOPBACK**
The mode of the CAN bus used in *init()*.

CAN.**STOPPED**
CAN.**ERROR_ACTIVE**
CAN.**ERROR_WARNING**
CAN.**ERROR_PASSIVE**
CAN.**BUS_OFF**
Possible states of the CAN controller returned from *state()*.

CAN.**LIST16**
CAN.**MASK16**
CAN.**LIST32**
CAN.**MASK32**
The operation mode of a filter used in *setfilter()*.

class DAC – digital to analog conversion

The DAC is used to output analog values (a specific voltage) on pin X5 or pin X6. The voltage will be between 0 and 3.3V.

This module will undergo changes to the API.

Example usage:

```
from pyb import DAC

dac = DAC(1)            # create DAC 1 on pin X5
dac.write(128)          # write a value to the DAC (makes X5 1.65V)

dac = DAC(1, bits=12)   # use 12 bit resolution
dac.write(4095)         # output maximum value, 3.3V
```

To output a continuous sine-wave:

```
import math
from pyb import DAC

# create a buffer containing a sine-wave
buf = bytearray(100)
for i in range(len(buf)):
    buf[i] = 128 + int(127 * math.sin(2 * math.pi * i / len(buf)))

# output the sine-wave at 400Hz
```

(continues on next page)

(continued from previous page)

```
dac = DAC(1)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

To output a continuous sine-wave at 12-bit resolution:

```
import math
from array import array
from pyb import DAC

# create a buffer containing a sine-wave, using half-word samples
buf = array('H', 2048 + int(2047 * math.sin(2 * math.pi * i / 128)) for i in range(128))

# output the sine-wave at 400Hz
dac = DAC(1, bits=12)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

Constructors

class pyb.**DAC**(*port*, *bits=8*, ***, *buffering=None*)

Construct a new DAC object.

`port` can be a pin object, or an integer (1 or 2). DAC(1) is on pin X5 and DAC(2) is on pin X6.

`bits` is an integer specifying the resolution, and can be 8 or 12. The maximum value for the write and write_timed methods will be 2**``bits``-1.

The *buffering* parameter selects the behaviour of the DAC op-amp output buffer, whose purpose is to reduce the output impedance. It can be `None` to select the default (buffering enabled for *DAC.noise()*, *DAC.triangle()* and *DAC.write_timed()*, and disabled for *DAC.write()*), `False` to disable buffering completely, or `True` to enable output buffering.

When buffering is enabled the DAC pin can drive loads down to 5K. Otherwise it has an output impedance of 15K maximum: consequently to achieve a 1% accuracy without buffering requires the applied load to be less than 1.5M. Using the buffer incurs a penalty in accuracy, especially near the extremes of range.

Methods

DAC.**init**(*bits=8*, ***, *buffering=None*)

Reinitialise the DAC. *bits* can be 8 or 12. *buffering* can be `None`, `False` or `True`; see above constructor for the meaning of this parameter.

DAC.**deinit**()

De-initialise the DAC making its pin available for other uses.

DAC.**noise**(*freq*)

Generate a pseudo-random noise signal. A new random sample is written to the DAC output at the given frequency.

DAC.**triangle**(*freq*)

Generate a triangle wave. The value on the DAC output changes at the given frequency and ramps through the full 12-bit range (up and down). Therefore the frequency of the repeating triangle wave itself is 8192 times smaller.

DAC.**write**(*value*)

Direct access to the DAC output. The minimum value is 0. The maximum value is 2**``bits``-1, where `bits` is set when creating the DAC object or by using the `init` method.

DAC.**write_timed**(*data*, *freq*, *, *mode=DAC.NORMAL*)

Initiates a burst of RAM to DAC using a DMA transfer. The input data is treated as an array of bytes in 8-bit mode, and an array of unsigned half-words (array typecode H) in 12-bit mode.

`freq` can be an integer specifying the frequency to write the DAC samples at, using Timer(6). Or it can be an already-initialised Timer object which is used to trigger the DAC sample. Valid timers are 2, 4, 5, 6, 7 and 8.

`mode` can be `DAC.NORMAL` or `DAC.CIRCULAR`.

Example using both DACs at the same time:

```
dac1 = DAC(1)
dac2 = DAC(2)
dac1.write_timed(buf1, pyb.Timer(6, freq=100), mode=DAC.CIRCULAR)
dac2.write_timed(buf2, pyb.Timer(7, freq=200), mode=DAC.CIRCULAR)
```

class ExtInt – configure I/O pins to interrupt on external events

There are a total of 22 interrupt lines. 16 of these can come from GPIO pins and the remaining 6 are from internal sources.

For lines 0 through 15, a given line can map to the corresponding line from an arbitrary port. So line 0 can map to Px0 where x is A, B, C, and line 1 can map to Px1 where x is A, B, C,

```
def callback(line):
    print("line =", line)
```

Note: ExtInt will automatically configure the gpio line as an input.

```
extint = pyb.ExtInt(pin, pyb.ExtInt.IRQ_FALLING, pyb.Pin.PULL_UP, callback)
```

Now every time a falling edge is seen on the X1 pin, the callback will be called. Caution: mechanical pushbuttons have bounce and pushing or releasing a switch will often generate multiple edges. See: http://www.eng.utah.edu/~cs5780/debouncing.pdf for a detailed explanation, along with various techniques for debouncing.

Trying to register 2 callbacks onto the same pin will throw an exception.

If pin is passed as an integer, then it is assumed to map to one of the internal interrupt sources, and must be in the range 16 through 22.

All other pin objects go through the pin mapper to come up with one of the gpio pins.

```
extint = pyb.ExtInt(pin, mode, pull, callback)
```

Valid modes are pyb.ExtInt.IRQ_RISING, pyb.ExtInt.IRQ_FALLING, pyb.ExtInt.IRQ_RISING_FALLING, pyb.ExtInt.EVT_RISING, pyb.ExtInt.EVT_FALLING, and pyb.ExtInt.EVT_RISING_FALLING.

Only the IRQ_xxx modes have been tested. The EVT_xxx modes have something to do with sleep mode and the WFE instruction.

Valid pull values are pyb.Pin.PULL_UP, pyb.Pin.PULL_DOWN, pyb.Pin.PULL_NONE.

There is also a C API, so that drivers which require EXTI interrupt lines can also use this code. See extint.h for the available functions and usrsw.h for an example of using this.

Constructors

class pyb.**ExtInt**(*pin*, *mode*, *pull*, *callback*)
Create an ExtInt object:

- `pin` is the pin on which to enable the interrupt (can be a pin object or any valid pin name).
- `mode` can be one of: - `ExtInt.IRQ_RISING` - trigger on a rising edge; - `ExtInt.IRQ_FALLING` - trigger on a falling edge; - `ExtInt.IRQ_RISING_FALLING` - trigger on a rising or falling edge.
- `pull` can be one of: - `pyb.Pin.PULL_NONE` - no pull up or down resistors; - `pyb.Pin.PULL_UP` - enable the pull-up resistor; - `pyb.Pin.PULL_DOWN` - enable the pull-down resistor.
- `callback` is the function to call when the interrupt triggers. The callback function must accept exactly 1 argument, which is the line that triggered the interrupt.

Class methods

classmethod ExtInt.**regs**()
Dump the values of the EXTI registers.

Methods

ExtInt.**disable**()
Disable the interrupt associated with the ExtInt object. This could be useful for debouncing.

ExtInt.**enable**()
Enable a disabled interrupt.

ExtInt.**line**()
Return the line number that the pin is mapped to.

ExtInt.**swint**()
Trigger the callback from software.

Constants

ExtInt.**IRQ_FALLING**
interrupt on a falling edge

ExtInt.**IRQ_RISING**
interrupt on a rising edge

ExtInt.**IRQ_RISING_FALLING**
interrupt on a rising or falling edge

class Flash – access to built-in flash storage

The Flash class allows direct access to the primary flash device on the pyboard.

In most cases, to store persistent data on the device, youll want to use a higher-level abstraction, for example the filesystem via Pythons standard file API, but this interface is useful to *customise the filesystem configuration* or implement a low-level storage system for your application.

Constructors

class pyb.**Flash**
Create and return a block device that represents the flash device presented to the USB mass storage interface.

It includes a virtual partition table at the start, and the actual flash starts at block `0x100`.

This constructor is deprecated and will be removed in a future version of MicroPython.

class pyb.**Flash**(*, *start=- 1*, *len=- 1*)
Create and return a block device that accesses the flash at the specified offset. The length defaults to the remaining size of the device.

The *start* and *len* offsets are in bytes, and must be a multiple of the block size (typically 512 for internal flash).

Methods

Flash.**readblocks**(*block_num*, *buf*)
Flash.**readblocks**(*block_num*, *buf*, *offset*)

Flash.**writeblocks**(*block_num*, *buf*)
Flash.**writeblocks**(*block_num*, *buf*, *offset*)

Flash.**ioctl**(*cmd*, *arg*)
These methods implement the simple and *extended* block protocol defined by `os.AbstractBlockDev`.

Hardware Note

On boards with external spiflash (e.g. Pyboard D), the MicroPython firmware will be configured to use that as the primary flash storage. On all other boards, the internal flash inside the *MCU* will be used.

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.

Example:

```
from pyb import I2C

i2c = I2C(1)                           # create on bus 1
i2c = I2C(1, I2C.CONTROLLER)           # create and init as a controller
i2c.init(I2C.CONTROLLER, baudrate=20000) # init as a controller
```

(continues on next page)

(continued from previous page)

```
i2c.init(I2C.PERIPHERAL, addr=0x42)      # init as a peripheral with given address
i2c.deinit()                             # turn off the I2C unit
```

Printing the i2c object gives you information about its configuration.

The basic methods are send and recv:

```
i2c.send('abc')      # send 3 bytes
i2c.send(0x42)       # send a single byte, given by the number
data = i2c.recv(3)   # receive 3 bytes
```

To receive inplace, first create a bytearray:

```
data = bytearray(3)  # create a buffer
i2c.recv(data)       # receive 3 bytes, writing them into data
```

You can specify a timeout (in ms):

```
i2c.send(b'123', timeout=2000)   # timeout after 2 seconds
```

A controller must specify the recipients address:

```
i2c.init(I2C.CONTROLLER)
i2c.send('123', 0x42)        # send 3 bytes to peripheral with address 0x42
i2c.send(b'456', addr=0x42)  # keyword for address
```

Master also has other methods:

```
i2c.is_ready(0x42)           # check if peripheral 0x42 is ready
i2c.scan()                   # scan for peripherals on the bus, returning
                             #   a list of valid addresses
i2c.mem_read(3, 0x42, 2)     # read 3 bytes from memory of peripheral 0x42,
                             #   starting at address 2 in the peripheral
i2c.mem_write('abc', 0x42, 2, timeout=1000) # write 'abc' (3 bytes) to memory of␣
↪peripheral 0x42
                                            # starting at address 2 in the peripheral,␣
↪timeout after 1 second
```

Constructors

class pyb.**I2C**(*bus*, ...)

Construct an I2C object on the given bus. `bus` can be 1 or 2, X or Y. With no additional parameters, the I2C 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.

The physical pins of the I2C buses on Pyboards V1.0 and V1.1 are:

- `I2C(1)` is on the X position: `(SCL, SDA) = (X9, X10) = (PB6, PB7)`
- `I2C(2)` is on the Y position: `(SCL, SDA) = (Y9, Y10) = (PB10, PB11)`

On the Pyboard Lite:

- `I2C(1)` is on the X position: `(SCL, SDA) = (X9, X10) = (PB6, PB7)`
- `I2C(3)` is on the Y position: `(SCL, SDA) = (Y9, Y10) = (PA8, PB8)`

Calling the constructor with X or Y enables portability between Pyboard types.

Methods

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

I2C.**init**(*mode*, ***, *addr=0x12*, *baudrate=400000*, *gencall=False*, *dma=False*)
Initialise the I2C bus with the given parameters:

- `mode` must be either `I2C.CONTROLLER` or `I2C.PERIPHERAL`
- `addr` is the 7-bit address (only sensible for a peripheral)
- `baudrate` is the SCL clock rate (only sensible for a controller)
- `gencall` is whether to support general call mode
- `dma` is whether to allow the use of DMA for the I2C transfers (note that DMA transfers have more precise timing but currently do not handle bus errors properly)

I2C.**is_ready**(*addr*)
Check if an I2C device responds to the given address. Only valid when in controller mode.

I2C.**mem_read**(*data*, *addr*, *memaddr*, ***, *timeout=5000*, *addr_size=8*)
Read from the memory of an I2C device:

- `data` can be an integer (number of bytes to read) or a buffer to read into
- `addr` is the I2C device address
- `memaddr` is the memory location within the I2C device
- `timeout` is the timeout in milliseconds to wait for the read
- `addr_size` selects width of memaddr: 8 or 16 bits

Returns the read data. This is only valid in controller mode.

I2C.**mem_write**(*data*, *addr*, *memaddr*, ***, *timeout=5000*, *addr_size=8*)
Write to the memory of an I2C device:

- `data` can be an integer or a buffer to write from
- `addr` is the I2C device address
- `memaddr` is the memory location within the I2C device
- `timeout` is the timeout in milliseconds to wait for the write
- `addr_size` selects width of memaddr: 8 or 16 bits

Returns `None`. This is only valid in controller mode.

I2C.**recv**(*recv*, *addr=0x00*, ***, *timeout=5000*)
Receive data on the bus:

- `recv` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes
- `addr` is the address to receive from (only required in controller mode)
- `timeout` is the timeout in milliseconds to wait for the receive

Return value: if `recv` is an integer then a new buffer of the bytes received, otherwise the same buffer that was passed in to `recv`.

I2C.**send**(*send*, *addr=0x00*, *, *timeout=5000*)
Send data on the bus:

- `send` is the data to send (an integer to send, or a buffer object)
- `addr` is the address to send to (only required in controller mode)
- `timeout` is the timeout in milliseconds to wait for the send

Return value: `None`.

I2C.**scan**()
Scan all I2C addresses from 0x01 to 0x7f and return a list of those that respond. Only valid when in controller mode.

Constants

I2C.**CONTROLLER**
for initialising the bus to controller mode

I2C.**PERIPHERAL**
for initialising the bus to peripheral mode

class LCD – LCD control for the LCD touch-sensor pyskin

The LCD class is used to control the LCD on the LCD touch-sensor pyskin, LCD32MKv1.0. The LCD is a 128x32 pixel monochrome screen, part NHD-C12832A1Z.

The pyskin must be connected in either the X or Y positions, and then an LCD object is made using:

```
lcd = pyb.LCD('X')      # if pyskin is in the X position
lcd = pyb.LCD('Y')      # if pyskin is in the Y position
```

Then you can use:

```
lcd.light(True)                 # turn the backlight on
lcd.write('Hello world!\n')     # print text to the screen
```

This driver implements a double buffer for setting/getting pixels. For example, to make a bouncing dot, try:

```
x = y = 0
dx = dy = 1
while True:
    # update the dot's position
    x += dx
    y += dy

    # make the dot bounce of the edges of the screen
    if x <= 0 or x >= 127: dx = -dx
    if y <= 0 or y >= 31: dy = -dy

    lcd.fill(0)                 # clear the buffer
    lcd.pixel(x, y, 1)          # draw the dot
    lcd.show()                  # show the buffer
    pyb.delay(50)               # pause for 50ms
```

Constructors

class pyb.LCD(*skin_position*)
Construct an LCD object in the given skin position. `skin_position` can be X or Y, and should match the position where the LCD pyskin is plugged in.

Methods

LCD.command(*instr_data*, *buf*)
Send an arbitrary command to the LCD. Pass 0 for `instr_data` to send an instruction, otherwise pass 1 to send data. `buf` is a buffer with the instructions/data to send.

LCD.contrast(*value*)
Set the contrast of the LCD. Valid values are between 0 and 47.

LCD.fill(*colour*)
Fill the screen with the given colour (0 or 1 for white or black).

This method writes to the hidden buffer. Use `show()` to show the buffer.

LCD.get(*x*, *y*)
Get the pixel at the position `(x, y)`. Returns 0 or 1.

This method reads from the visible buffer.

LCD.light(*value*)
Turn the backlight on/off. True or 1 turns it on, False or 0 turns it off.

LCD.pixel(*x*, *y*, *colour*)
Set the pixel at `(x, y)` to the given colour (0 or 1).

This method writes to the hidden buffer. Use `show()` to show the buffer.

LCD.show()
Show the hidden buffer on the screen.

LCD.text(*str*, *x*, *y*, *colour*)
Draw the given text to the position `(x, y)` using the given colour (0 or 1).

This method writes to the hidden buffer. Use `show()` to show the buffer.

LCD.write(*str*)
Write the string `str` to the screen. It will appear immediately.

class LED – LED object

The LED object controls an individual LED (Light Emitting Diode).

Constructors

class pyb.**LED**(*id*)
Create an LED object associated with the given LED:

- `id` is the LED number, 1-4.

Methods

LED.**intensity**([*value*])
Get or set the LED intensity. Intensity ranges between 0 (off) and 255 (full on). If no argument is given, return the LED intensity. If an argument is given, set the LED intensity and return `None`.

Note: Only LED(3) and LED(4) can have a smoothly varying intensity, and they use timer PWM to implement it. LED(3) uses Timer(2) and LED(4) uses Timer(3). These timers are only configured for PWM if the intensity of the relevant LED is set to a value between 1 and 254. Otherwise the timers are free for general purpose use.

LED.**off**()
Turn the LED off.

LED.**on**()
Turn the LED on, to maximum intensity.

LED.**toggle**()
Toggle the LED between on (maximum intensity) and off. If the LED is at non-zero intensity then it is considered on and toggle will turn it off.

class Pin – control I/O pins

A pin is the basic object to control I/O pins. It has methods to set the mode of the pin (input, output, etc) and methods to get and set the digital logic level. For analog control of a pin, see the ADC class.

Usage Model:

All Board Pins are predefined as pyb.Pin.board.Name:

```
x1_pin = pyb.Pin.board.X1

g = pyb.Pin(pyb.Pin.board.X1, pyb.Pin.IN)
```

CPU pins which correspond to the board pins are available as `pyb.Pin.cpu.Name`. For the CPU pins, the names are the port letter followed by the pin number. On the PYBv1.0, `pyb.Pin.board.X1` and `pyb.Pin.cpu.A0` are the same pin.

You can also use strings:

```
g = pyb.Pin('X1', pyb.Pin.OUT_PP)
```

Users can add their own names:

```
MyMapperDict = { 'LeftMotorDir' : pyb.Pin.cpu.C12 }
pyb.Pin.dict(MyMapperDict)
g = pyb.Pin("LeftMotorDir", pyb.Pin.OUT_OD)
```

and can query mappings:

```
pin = pyb.Pin("LeftMotorDir")
```

Users can also add their own mapping function:

```
def MyMapper(pin_name):
   if pin_name == "LeftMotorDir":
       return pyb.Pin.cpu.A0

pyb.Pin.mapper(MyMapper)
```

So, if you were to call: `pyb.Pin("LeftMotorDir", pyb.Pin.OUT_PP)` then `"LeftMotorDir"` is passed directly to the mapper function.

To summarise, the following order determines how things get mapped into an ordinal pin number:

1. Directly specify a pin object
2. User supplied mapping function
3. User supplied mapping (object must be usable as a dictionary key)
4. Supply a string which matches a board pin
5. Supply a string which matches a CPU port/pin

You can set `pyb.Pin.debug(True)` to get some debug information about how a particular object gets mapped to a pin.

When a pin has the `Pin.PULL_UP` or `Pin.PULL_DOWN` pull-mode enabled, that pin has an effective 40k Ohm resistor pulling it to 3V3 or GND respectively (except pin Y5 which has 11k Ohm resistors).

Now every time a falling edge is seen on the gpio pin, the callback will be executed. Caution: mechanical push buttons have bounce and pushing or releasing a switch will often generate multiple edges. See: http://www.eng.utah.edu/~cs5780/debouncing.pdf for a detailed explanation, along with various techniques for debouncing.

All pin objects go through the pin mapper to come up with one of the gpio pins.

Constructors

class pyb.**Pin**(*id*, ...)
Create a new Pin object associated with the id. If additional arguments are given, they are used to initialise the pin. See `pin.init()`.

Class methods

classmethod Pin.**debug**([*state*])
Get or set the debugging state (`True` or `False` for on or off).

classmethod Pin.**dict**([*dict*])
Get or set the pin mapper dictionary.

classmethod Pin.**mapper**([*fun*])
Get or set the pin mapper function.

Methods

Pin.**init**(*mode*, *pull=Pin.PULL_NONE*, *, *value=None*, *alt=-1*)

Initialise the pin:

- *mode* can be one of:
  - `Pin.IN` - configure the pin for input;
  - `Pin.OUT_PP` - configure the pin for output, with push-pull control;
  - `Pin.OUT_OD` - configure the pin for output, with open-drain control;
  - `Pin.AF_PP` - configure the pin for alternate function, pull-pull;
  - `Pin.AF_OD` - configure the pin for alternate function, open-drain;
  - `Pin.ANALOG` - configure the pin for analog.
- *pull* can be one of:
  - `Pin.PULL_NONE` - no pull up or down resistors;
  - `Pin.PULL_UP` - enable the pull-up resistor;
  - `Pin.PULL_DOWN` - enable the pull-down resistor.
- *value* if not None will set the port output value before enabling the pin.
- *alt* can be used when mode is `Pin.AF_PP` or `Pin.AF_OD` to set the index or name of one of the alternate functions associated with a pin. This arg was previously called *af* which can still be used if needed.

Returns: `None`.

Pin.**value**([*value*])

Get or set the digital logic level of the pin:

- With no argument, return 0 or 1 depending on the logic level of the pin.
- With `value` given, set the logic level of the pin. `value` can be anything that converts to a boolean. If it converts to `True`, the pin is set high, otherwise it is set low.

Pin.**__str__**()

Return a string describing the pin object.

Pin.**af**()

Returns the currently configured alternate-function of the pin. The integer returned will match one of the allowed constants for the af argument to the init function.

Pin.**af_list**()

Returns an array of alternate functions available for this pin.

Pin.**gpio**()

Returns the base address of the GPIO block associated with this pin.

Pin.**mode**()

Returns the currently configured mode of the pin. The integer returned will match one of the allowed constants for the mode argument to the init function.

Pin.**name**()

Get the pin name.

Pin.**names**()

Returns the cpu and board names for this pin.

Pin.**pin**()
Get the pin number.

Pin.**port**()
Get the pin port.

Pin.**pull**()
Returns the currently configured pull of the pin. The integer returned will match one of the allowed constants for the pull argument to the init function.

Constants

Pin.**AF_OD**
initialise the pin to alternate-function mode with an open-drain drive

Pin.**AF_PP**
initialise the pin to alternate-function mode with a push-pull drive

Pin.**ANALOG**
initialise the pin to analog mode

Pin.**IN**
initialise the pin to input mode

Pin.**OUT_OD**
initialise the pin to output mode with an open-drain drive

Pin.**OUT_PP**
initialise the pin to output mode with a push-pull drive

Pin.**PULL_DOWN**
enable the pull-down resistor on the pin

Pin.**PULL_NONE**
dont enable any pull up or down resistors on the pin

Pin.**PULL_UP**
enable the pull-up resistor on the pin

class PinAF – Pin Alternate Functions

A Pin represents a physical pin on the microprocessor. Each pin can have a variety of functions (GPIO, I2C SDA, etc). Each PinAF object represents a particular function for a pin.

Usage Model:

```
x3 = pyb.Pin.board.X3
x3_af = x3.af_list()
```

x3_af will now contain an array of PinAF objects which are available on pin X3.

For the pyboard, x3_af would contain: [Pin.AF1_TIM2, Pin.AF2_TIM5, Pin.AF3_TIM9, Pin.AF7_USART2]

Normally, each peripheral would configure the af automatically, but sometimes the same function is available on multiple pins, and having more control is desired.

To configure X3 to expose TIM2_CH3, you could use:

```
pin = pyb.Pin(pyb.Pin.board.X3, mode=pyb.Pin.AF_PP, af=pyb.Pin.AF1_TIM2)
```

or:

```
pin = pyb.Pin(pyb.Pin.board.X3, mode=pyb.Pin.AF_PP, af=1)
```

Methods

pinaf.**__str__**()
Return a string describing the alternate function.

pinaf.**index**()
Return the alternate function index.

pinaf.**name**()
Return the name of the alternate function.

pinaf.**reg**()
Return the base register associated with the peripheral assigned to this alternate function. For example, if the alternate function were TIM2_CH3 this would return stm.TIM2

class RTC – real time clock

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

Example usage:

```
rtc = pyb.RTC()
rtc.datetime((2014, 5, 1, 4, 13, 0, 0, 0))
print(rtc.datetime())
```

Constructors

class pyb.**RTC**
Create an RTC object.

Methods

RTC.**datetime**([*datetimetuple*])
Get or set the date and time of the RTC.

With no arguments, this method returns an 8-tuple with the current date and time. With 1 argument (being an 8-tuple) it sets the date and time (and `subseconds` is reset to 255).

The 8-tuple has the following format:

(year, month, day, weekday, hours, minutes, seconds, subseconds)

`weekday` is 1-7 for Monday through Sunday.

`subseconds` counts down from 255 to 0

RTC.**wakeup**(*timeout*, *callback=None*)

Set the RTC wakeup timer to trigger repeatedly at every `timeout` milliseconds. This trigger can wake the pyboard from both the sleep states: *pyb.stop()* and *pyb.standby()*.

If `timeout` is `None` then the wakeup timer is disabled.

If `callback` is given then it is executed at every trigger of the wakeup timer. `callback` must take exactly one argument.

RTC.**info**()

Get information about the startup time and reset source.

- The lower 0xffff are the number of milliseconds the RTC took to start up.
- Bit 0x10000 is set if a power-on reset occurred.
- Bit 0x20000 is set if an external reset occurred

RTC.**calibration**(*cal*)

Get or set RTC calibration.

With no arguments, `calibration()` returns the current calibration value, which is an integer in the range [-511 : 512]. With one argument it sets the RTC calibration.

The RTC Smooth Calibration mechanism adjusts the RTC clock rate by adding or subtracting the given number of ticks from the 32768 Hz clock over a 32 second period (corresponding to 2^20 clock ticks.) Each tick added will speed up the clock by 1 part in 2^20, or 0.954 ppm; likewise the RTC clock it slowed by negative values. The usable calibration range is: (-511 * 0.954) ~= -487.5 ppm up to (512 * 0.954) ~= 488.5 ppm

class Servo – 3-wire hobby servo driver

Servo objects control standard hobby servo motors with 3-wires (ground, power, signal). There are 4 positions on the pyboard where these motors can be plugged in: pins X1 through X4 are the signal pins, and next to them are 4 sets of power and ground pins.

Example usage:

```
import pyb

s1 = pyb.Servo(1)   # create a servo object on position X1
s2 = pyb.Servo(2)   # create a servo object on position X2

s1.angle(45)        # move servo 1 to 45 degrees
s2.angle(0)         # move servo 2 to 0 degrees

# move servo1 and servo2 synchronously, taking 1500ms
s1.angle(-60, 1500)
s2.angle(30, 1500)
```

Note: The Servo objects use Timer(5) to produce the PWM output. You can use Timer(5) for Servo control, or your own purposes, but not both at the same time.

Constructors

class pyb.**Servo**(*id*)
Create a servo object. `id` is 1-4, and corresponds to pins X1 through X4.

Methods

Servo.**angle**([*angle*, *time=0*])
If no arguments are given, this function returns the current angle.

If arguments are given, this function sets the angle of the servo:

- `angle` is the angle to move to in degrees.
- `time` is the number of milliseconds to take to get to the specified angle. If omitted, then the servo moves as quickly as possible to its new position.

Servo.**speed**([*speed*, *time=0*])
If no arguments are given, this function returns the current speed.

If arguments are given, this function sets the speed of the servo:

- `speed` is the speed to change to, between -100 and 100.
- `time` is the number of milliseconds to take to get to the specified speed. If omitted, then the servo accelerates as quickly as possible.

Servo.**pulse_width**([*value*])
If no arguments are given, this function returns the current raw pulse-width value.

If an argument is given, this function sets the raw pulse-width value.

Servo.**calibration**([*pulse_min*, *pulse_max*, *pulse_centre*[, *pulse_angle_90*, *pulse_speed_100*]])
If no arguments are given, this function returns the current calibration data, as a 5-tuple.

If arguments are given, this function sets the timing calibration:

- `pulse_min` is the minimum allowed pulse width.
- `pulse_max` is the maximum allowed pulse width.
- `pulse_centre` is the pulse width corresponding to the centre/zero position.
- `pulse_angle_90` is the pulse width corresponding to 90 degrees.
- `pulse_speed_100` is the pulse width corresponding to a speed of 100.

class SPI – a controller-driven serial protocol

SPI is a serial protocol that is driven by a controller. At the physical level there are 3 lines: SCK, MOSI, MISO.

See usage model of I2C; SPI is very similar. Main difference is parameters to init the SPI bus:

```
from pyb import SPI
spi = SPI(1, SPI.CONTROLLER, baudrate=600000, polarity=1, phase=0, crc=0x7)
```

Only required parameter is mode, SPI.CONTROLLER or SPI.PERIPHERAL. 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. Crc can be None for no CRC, or a polynomial specifier.

Additional methods for SPI:

```
data = spi.send_recv(b'1234')        # send 4 bytes and receive 4 bytes
buf = bytearray(4)
spi.send_recv(b'1234', buf)          # send 4 bytes and receive 4 into buf
spi.send_recv(buf, buf)              # send/recv 4 bytes from/to buf
```

Constructors

class pyb.**SPI**(*bus*, ...)

Construct an SPI object on the given bus. `bus` can be 1 or 2, or X or Y. 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.

The physical pins of the SPI buses are:

- `SPI(1)` is on the X position: `(NSS, SCK, MISO, MOSI) = (X5, X6, X7, X8) = (PA4, PA5, PA6, PA7)`
- `SPI(2)` is on the Y position: `(NSS, SCK, MISO, MOSI) = (Y5, Y6, Y7, Y8) = (PB12, PB13, PB14, PB15)`

At the moment, the NSS pin is not used by the SPI driver and is free for other use.

Methods

SPI.**deinit**()

Turn off the SPI bus.

SPI.**init**(*mode*, *baudrate=328125*, ***, *prescaler*, *polarity=1*, *phase=0*, *bits=8*, *firstbit=SPI.MSB*, *ti=False*, *crc=None*)

Initialise the SPI bus with the given parameters:

- `mode` must be either `SPI.CONTROLLER` or `SPI.PERIPHERAL`.
- `baudrate` is the SCK clock rate (only sensible for a controller).
- `prescaler` is the prescaler to use to derive SCK from the APB bus frequency; use of `prescaler` overrides `baudrate`.
- `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` can be 8 or 16, and is the number of bits in each transferred word.
- `firstbit` can be `SPI.MSB` or `SPI.LSB`.
- `ti` True indicates Texas Instruments, as opposed to Motorola, signal conventions.
- `crc` can be None for no CRC, or a polynomial specifier.

Note that the SPI clock frequency will not always be the requested baudrate. The hardware only supports baudrates that are the APB bus frequency (see *pyb.freq()*) divided by a prescaler, which can be 2, 4, 8, 16, 32, 64, 128 or 256. SPI(1) is on AHB2, and SPI(2) is on AHB1. For precise control over the SPI clock frequency, specify `prescaler` instead of `baudrate`.

Printing the SPI object will show you the computed baudrate and the chosen prescaler.

SPI.**recv**(*recv*, ***, *timeout=5000*)

Receive data on the bus:

- `recv` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: if `recv` is an integer then a new buffer of the bytes received, otherwise the same buffer that was passed in to `recv`.

`SPI.send`(*send*, *, *timeout=5000*)
Send data on the bus:

- `send` is the data to send (an integer to send, or a buffer object).
- `timeout` is the timeout in milliseconds to wait for the send.

Return value: `None`.

`SPI.send_recv`(*send*, *recv=None*, *, *timeout=5000*)
Send and receive data on the bus at the same time:

- `send` is the data to send (an integer to send, or a buffer object).
- `recv` is a mutable buffer which will be filled with received bytes. It can be the same as `send`, or omitted. If omitted, a new buffer will be created.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: the buffer with the received bytes.

Constants

`SPI.CONTROLLER`

`SPI.PERIPHERAL`
for initialising the SPI bus to controller or peripheral mode

`SPI.LSB`

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

class Switch – switch object

A Switch object is used to control a push-button switch.

Usage:

```
sw = pyb.Switch()       # create a switch object
sw.value()              # get state (True if pressed, False otherwise)
sw()                    # shorthand notation to get the switch state
sw.callback(f)          # register a callback to be called when the
                        #   switch is pressed down
sw.callback(None)       # remove the callback
```

Example:

```
pyb.Switch().callback(lambda: pyb.LED(1).toggle())
```

Constructors

class pyb.**Switch**
Create and return a switch object.

Methods

Switch.**__call__**()
Call switch object directly to get its state: `True` if pressed down, `False` otherwise.

Switch.**value**()
Get the switch state. Returns `True` if pressed down, otherwise `False`.

Switch.**callback**(*fun*)
Register the given function to be called when the switch is pressed down. If `fun` is `None`, then it disables the callback.

class Timer – control internal timers

Timers can be used for a great variety of tasks. At the moment, only the simplest case is implemented: that of calling a function periodically.

Each timer consists of a counter that counts up at a certain rate. The rate at which it counts is the peripheral clock frequency (in Hz) divided by the timer prescaler. When the counter reaches the timer period it triggers an event, and the counter resets back to zero. By using the callback method, the timer event can call a Python function.

Example usage to toggle an LED at a fixed frequency:

```
tim = pyb.Timer(4)              # create a timer object using timer 4
tim.init(freq=2)                # trigger at 2Hz
tim.callback(lambda t:pyb.LED(1).toggle())
```

Example using named function for the callback:

```
def tick(timer):                # we will receive the timer object when being called
    print(timer.counter())      # show current timer's counter value
tim = pyb.Timer(4, freq=1)      # create a timer object using timer 4 - trigger at 1Hz
tim.callback(tick)              # set the callback to our tick function
```

Further examples:

```
tim = pyb.Timer(4, freq=100)    # freq in Hz
tim = pyb.Timer(4, prescaler=0, period=99)
tim.counter()                   # get counter (can also set)
tim.prescaler(2)                # set prescaler (can also get)
tim.period(199)                 # set period (can also get)
tim.callback(lambda t: ...)     # set callback for update interrupt (t=tim instance)
tim.callback(None)              # clear callback
```

Note: Timer(2) and Timer(3) are used for PWM to set the intensity of LED(3) and LED(4) respectively. But these timers are only configured for PWM if the intensity of the relevant LED is set to a value between 1 and 254. If the intensity feature of the LEDs is not used then these timers are free for general purpose use. Similarly, Timer(5) controls the servo driver, and Timer(6) is used for timed ADC/DAC reading/writing. It is recommended to use the other timers in your programs.

Note: Memory cant be allocated during a callback (an interrupt) and so exceptions raised within a callback dont give much information. See *micropython.alloc_emergency_exception_buf()* for how to get around this limitation.

Constructors

class pyb.**Timer**(*id*, ...)

Construct a new timer object of the given id. If additional arguments are given, then the timer is initialised by `init(...)`. `id` can be 1 to 14.

Methods

Timer.**init**(*, *freq*, *prescaler*, *period*, *mode=Timer.UP*, *div=1*, *callback=None*, *deadtime=0*)

Initialise the timer. Initialisation must be either by frequency (in Hz) or by prescaler and period:

```
tim.init(freq=100)                  # set the timer to trigger at 100Hz
tim.init(prescaler=83, period=999)  # set the prescaler and period directly
```

Keyword arguments:

- `freq` specifies the periodic frequency of the timer. You might also view this as the frequency with which the timer goes through one complete cycle.
- `prescaler` [0-0xffff] - specifies the value to be loaded into the timers Prescaler Register (PSC). The timer clock source is divided by (`prescaler + 1`) to arrive at the timer clock. Timers 2-7 and 12-14 have a clock source of 84 MHz (pyb.freq()[2] * 2), and Timers 1, and 8-11 have a clock source of 168 MHz (pyb.freq()[3] * 2).
- `period` [0-0xffff] for timers 1, 3, 4, and 6-15. [0-0x3fffffff] for timers 2 & 5. Specifies the value to be loaded into the timers AutoReload Register (ARR). This determines the period of the timer (i.e. when the counter cycles). The timer counter will roll-over after `period + 1` timer clock cycles.
- `mode` can be one of:
  - `Timer.UP` - configures the timer to count from 0 to ARR (default)
  - `Timer.DOWN` - configures the timer to count from ARR down to 0.
  - `Timer.CENTER` - configures the timer to count from 0 to ARR and then back down to 0.
- `div` can be one of 1, 2, or 4. Divides the timer clock to determine the sampling clock used by the digital filters.
- `callback` - as per Timer.callback()
- `deadtime` - specifies the amount of dead or inactive time between transitions on complimentary channels (both channels will be inactive) for this time). `deadtime` may be an integer between 0 and 1008, with the following restrictions: 0-128 in steps of 1. 128-256 in steps of 2, 256-512 in steps of 8, and 512-1008 in steps of 16. `deadtime` measures ticks of `source_freq` divided by `div` clock ticks. `deadtime` is only available on timers 1 and 8.

You must either specify freq or both of period and prescaler.

Timer.**deinit**()

Deinitialises the timer.

Disables the callback (and the associated irq).

Disables any channel callbacks (and the associated irq). Stops the timer, and disables the timer peripheral.

Timer.**callback**(*fun*)
Set the function to be called when the timer triggers. `fun` is passed 1 argument, the timer object. If `fun` is `None` then the callback will be disabled.

Timer.**channel**(*channel*, *mode*, ...)
If only a channel number is passed, then a previously initialized channel object is returned (or `None` if there is no previous channel).

Otherwise, a TimerChannel object is initialized and returned.

Each channel can be configured to perform pwm, output compare, or input capture. All channels share the same underlying timer, which means that they share the same timer clock.

Keyword arguments:

- `mode` can be one of:
  - `Timer.PWM` configure the timer in PWM mode (active high).
  - `Timer.PWM_INVERTED` configure the timer in PWM mode (active low).
  - `Timer.OC_TIMING` indicates that no pin is driven.
  - `Timer.OC_ACTIVE` the pin will be made active when a compare match occurs (active is determined by polarity)
  - `Timer.OC_INACTIVE` the pin will be made inactive when a compare match occurs.
  - `Timer.OC_TOGGLE` the pin will be toggled when an compare match occurs.
  - `Timer.OC_FORCED_ACTIVE` the pin is forced active (compare match is ignored).
  - `Timer.OC_FORCED_INACTIVE` the pin is forced inactive (compare match is ignored).
  - `Timer.IC` configure the timer in Input Capture mode.
  - `Timer.ENC_A` configure the timer in Encoder mode. The counter only changes when CH1 changes.
  - `Timer.ENC_B` configure the timer in Encoder mode. The counter only changes when CH2 changes.
  - `Timer.ENC_AB` configure the timer in Encoder mode. The counter changes when CH1 or CH2 changes.
- `callback` - as per TimerChannel.callback()
- `pin` None (the default) or a Pin object. If specified (and not None) this will cause the alternate function of the the indicated pin to be configured for this timer channel. An error will be raised if the pin doesnt support any alternate functions for this timer channel.

Keyword arguments for Timer.PWM modes:

- `pulse_width` - determines the initial pulse width value to use.
- `pulse_width_percent` - determines the initial pulse width percentage to use.

Keyword arguments for Timer.OC modes:

- `compare` - determines the initial value of the compare register.
- `polarity` can be one of:
  - `Timer.HIGH` - output is active high
  - `Timer.LOW` - output is active low

Optional keyword arguments for Timer.IC modes:

- `polarity` can be one of:

- `Timer.RISING` - captures on rising edge.
- `Timer.FALLING` - captures on falling edge.
- `Timer.BOTH` - captures on both edges.

Note that capture only works on the primary channel, and not on the complimentary channels.

Notes for Timer.ENC modes:

- Requires 2 pins, so one or both pins will need to be configured to use the appropriate timer AF using the Pin API.
- Read the encoder value using the timer.counter() method.
- Only works on CH1 and CH2 (and not on CH1N or CH2N)
- The channel number is ignored when setting the encoder mode.

PWM Example:

```
timer = pyb.Timer(2, freq=1000)
ch2 = timer.channel(2, pyb.Timer.PWM, pin=pyb.Pin.board.X2, pulse_width=8000)
ch3 = timer.channel(3, pyb.Timer.PWM, pin=pyb.Pin.board.X3, pulse_width=16000)
```

`Timer.`**`counter`**`([`*value*`])`
Get or set the timer counter.

`Timer.`**`freq`**`([`*value*`])`
Get or set the frequency for the timer (changes prescaler and period if set).

`Timer.`**`period`**`([`*value*`])`
Get or set the period of the timer.

`Timer.`**`prescaler`**`([`*value*`])`
Get or set the prescaler for the timer.

`Timer.`**`source_freq`**`()`
Get the frequency of the source of the timer.

class TimerChannel setup a channel for a timer

Timer channels are used to generate/capture a signal using a timer.

TimerChannel objects are created using the Timer.channel() method.

Methods

`timerchannel.`**`callback`**`(`*fun*`)`
Set the function to be called when the timer channel triggers. `fun` is passed 1 argument, the timer object. If `fun` is `None` then the callback will be disabled.

`timerchannel.`**`capture`**`([`*value*`])`
Get or set the capture value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. capture is the logical name to use when the channel is in input capture mode.

`timerchannel.`**`compare`**`([`*value*`])`
Get or set the compare value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. compare is the logical name to use when the channel is in output compare mode.

timerchannel.**pulse_width**([*value*])
: Get or set the pulse width value associated with a channel. capture, compare, and pulse_width are all aliases for the same function. pulse_width is the logical name to use when the channel is in PWM mode.

  In edge aligned mode, a pulse_width of `period + 1` corresponds to a duty cycle of 100% In center aligned mode, a pulse width of `period` corresponds to a duty cycle of 100%

timerchannel.**pulse_width_percent**([*value*])
: Get or set the pulse width percentage associated with a channel. The value is a number between 0 and 100 and sets the percentage of the timer period for which the pulse is active. The value can be an integer or floating-point number for more accuracy. For example, a value of 25 gives a duty cycle of 25%.

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 pyb import UART

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

Bits can be 7, 8 or 9. Parity can be None, 0 (even) or 1 (odd). Stop can be 1 or 2.

Note: with parity=None, only 8 and 9 bits are supported. With parity enabled, only 7 and 8 bits are supported.

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
```

Individual characters can be read/written using:

```
uart.readchar()     # read 1 character and returns it as an integer
uart.writechar(42)  # write 1 character
```

To check if there is anything to be read, use:

```
uart.any()          # returns the number of characters waiting
```

Note: The stream functions `read`, `write`, etc. are new in MicroPython v1.3.4. Earlier versions use `uart.send` and `uart.recv`.

Constructors

class pyb.**UART**(*bus*, ...)

Construct a UART object on the given bus. For Pyboard `bus` can be 1-4, 6, XA, XB, YA, or YB. For Pyboard Lite `bus` can be 1, 2, 6, XB, or YA. For Pyboard D `bus` can be 1-4, XA, YA or YB. With no additional parameters, the UART 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.

The physical pins of the UART buses on Pyboard are:

- `UART(4)` is on `XA`: `(TX, RX) = (X1, X2) = (PA0, PA1)`
- `UART(1)` is on `XB`: `(TX, RX) = (X9, X10) = (PB6, PB7)`
- `UART(6)` is on `YA`: `(TX, RX) = (Y1, Y2) = (PC6, PC7)`
- `UART(3)` is on `YB`: `(TX, RX) = (Y9, Y10) = (PB10, PB11)`
- `UART(2)` is on: `(TX, RX) = (X3, X4) = (PA2, PA3)`

The Pyboard Lite supports UART(1), UART(2) and UART(6) only, pins are:

- `UART(1)` is on `XB`: `(TX, RX) = (X9, X10) = (PB6, PB7)`
- `UART(6)` is on `YA`: `(TX, RX) = (Y1, Y2) = (PC6, PC7)`
- `UART(2)` is on: `(TX, RX) = (X1, X2) = (PA2, PA3)`

The Pyboard D supports UART(1), UART(2), UART(3) and UART(4) only, pins are:

- `UART(4)` is on `XA`: `(TX, RX) = (X1, X2) = (PA0, PA1)`
- `UART(1)` is on `YA`: `(TX, RX) = (Y1, Y2) = (PA9, PA10)`
- `UART(3)` is on `YB`: `(TX, RX) = (Y9, Y10) = (PB10, PB11)`
- `UART(2)` is on: `(TX, RX) = (X3, X4) = (PA2, PA3)`

Note: Pyboard D has `UART(1)` on `YA`, unlike Pyboard and Pyboard Lite that both have `UART(1)` on `XB` and `UART(6)` on `YA`.

Methods

UART.**init**(*baudrate*, *bits=8*, *parity=None*, *stop=1*, ***, *timeout=0*, *flow=0*, *timeout_char=0*, *read_buf_len=64*)

Initialise the UART bus with the given parameters:

- `baudrate` is the clock rate.
- `bits` is the number of bits per character, 7, 8 or 9.
- `parity` is the parity, `None`, 0 (even) or 1 (odd).
- `stop` is the number of stop bits, 1 or 2.
- `flow` sets the flow control type. Can be 0, `UART.RTS`, `UART.CTS` or `UART.RTS | UART.CTS`.
- `timeout` is the timeout in milliseconds to wait for writing/reading the first character.
- `timeout_char` is the timeout in milliseconds to wait between characters while writing or reading.
- `read_buf_len` is the character length of the read buffer (0 to disable).

This method will raise an exception if the baudrate could not be set within 5% of the desired value. The minimum baudrate is dictated by the frequency of the bus that the UART is on; UART(1) and UART(6) are APB2, the rest

are on APB1. The default bus frequencies give a minimum baudrate of 1300 for UART(1) and UART(6) and 650 for the others. Use *pyb.freq* to reduce the bus frequencies to get lower baudrates.

Note: with parity=None, only 8 and 9 bits are supported. With parity enabled, only 7 and 8 bits are supported.

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

UART.**any**()
Returns the number of bytes waiting (may be 0).

UART.**read**([*nbytes*])
Read characters. If `nbytes` is specified then read at most that many bytes. If `nbytes` are available in the buffer, returns immediately, otherwise returns when sufficient characters arrive or the timeout elapses.

If `nbytes` is not given then the method reads as much data as possible. It returns after the timeout has elapsed.

Note: for 9 bit characters each character takes two bytes, `nbytes` must be even, and the number of characters is `nbytes/2`.

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

UART.**readchar**()
Receive a single character on the bus.

Return value: The character read, as an integer. Returns -1 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. If such a line exists, return is immediate. If the timeout elapses, all available data is returned regardless of whether a newline exists.

Return value: the line read or `None` on timeout if no data is available.

UART.**write**(*buf*)
Write the buffer of bytes to the bus. If characters are 7 or 8 bits wide then each byte is one character. If characters are 9 bits wide then two bytes are used for each character (little endian), and `buf` must contain an even number of bytes.

Return value: number of bytes written. If a timeout occurs and no bytes were written returns `None`.

UART.**writechar**(*char*)
Write a single character on the bus. `char` is an integer to write. Return value: `None`. See note below if CTS flow control is used.

UART.**sendbreak**()
Send a break condition on the bus. This drives the bus low for a duration of 13 bits. Return value: `None`.

Constants

UART.**RTS**
UART.**CTS**
to select the flow control type.

Flow Control

On Pyboards V1 and V1.1 `UART(2)` and `UART(3)` support RTS/CTS hardware flow control using the following pins:

- `UART(2)` is on: `(TX, RX, nRTS, nCTS) = (X3, X4, X2, X1) = (PA2, PA3, PA1, PA0)`
- `UART(3)` is on :`(TX, RX, nRTS, nCTS) = (Y9, Y10, Y7, Y6) = (PB10, PB11, PB14, PB13)`

On the Pyboard Lite only `UART(2)` supports flow control on these pins:

`(TX, RX, nRTS, nCTS) = (X1, X2, X4, X3) = (PA2, PA3, PA1, PA0)`

In the following paragraphs the term target refers to the device connected to the UART.

When the UARTs `init()` method is called with `flow` set to one or both of `UART.RTS` and `UART.CTS` the relevant flow control pins are configured. `nRTS` is an active low output, `nCTS` is an active low input with pullup enabled. To achieve flow control the Pyboards `nCTS` signal should be connected to the targets `nRTS` and the Pyboards `nRTS` to the targets `nCTS`.

CTS: target controls Pyboard transmitter

If CTS flow control is enabled the write behaviour is as follows:

If the Pyboards `UART.write(buf)` method is called, transmission will stall for any periods when `nCTS` is `False`. This will result in a timeout if the entire buffer was not transmitted in the timeout period. The method returns the number of bytes written, enabling the user to write the remainder of the data if required. In the event of a timeout, a character will remain in the UART pending `nCTS`. The number of bytes composing this character will be included in the return value.

If `UART.writechar()` is called when `nCTS` is `False` the method will time out unless the target asserts `nCTS` in time. If it times out `OSError 116` will be raised. The character will be transmitted as soon as the target asserts `nCTS`.

RTS: Pyboard controls targets transmitter

If RTS flow control is enabled, behaviour is as follows:

If buffered input is used (`read_buf_len` > 0), incoming characters are buffered. If the buffer becomes full, the next character to arrive will cause `nRTS` to go `False`: the target should cease transmission. `nRTS` will go `True` when characters are read from the buffer.

Note that the `any()` method returns the number of bytes in the buffer. Assume a buffer length of `N` bytes. If the buffer becomes full, and another character arrives, `nRTS` will be set False, and `any()` will return the count `N`. When characters are read the additional character will be placed in the buffer and will be included in the result of a subsequent `any()` call.

If buffered input is not used (`read_buf_len` == 0) the arrival of a character will cause `nRTS` to go `False` until the character is read.

class USB_HID – USB Human Interface Device (HID)

The USB_HID class allows creation of an object representing the USB Human Interface Device (HID) interface. It can be used to emulate a peripheral such as a mouse or keyboard.

Before you can use this class, you need to use *pyb.usb_mode()* to set the USB mode to include the HID interface.

Constructors

class pyb.**USB_HID**
Create a new USB_HID object.

Methods

USB_HID.**recv**(*data*, *, *timeout=5000*)
Receive data on the bus:

- `data` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: if `data` is an integer then a new buffer of the bytes received, otherwise the number of bytes read into `data` is returned.

USB_HID.**send**(*data*)
Send data over the USB HID interface:

- `data` is the data to send (a tuple/list of integers, or a bytearray).

class USB_VCP – USB virtual comm port

The USB_VCP class allows creation of a *stream*-like object representing the USB virtual comm port. It can be used to read and write data over USB to the connected host.

Constructors

class pyb.**USB_VCP**(*id=0*)
Create a new USB_VCP object. The *id* argument specifies which USB VCP port to use.

Methods

USB_VCP.**init**(*, *flow=- 1*)
Configure the USB VCP port. If the *flow* argument is not -1 then the value sets the flow control, which can be a bitwise-or of `USB_VCP.RTS` and `USB_VCP.CTS`. RTS is used to control read behaviour and CTS, to control write behaviour.

USB_VCP.**setinterrupt**(*chr*)
Set the character which interrupts running Python code. This is set to 3 (CTRL-C) by default, and when a CTRL-C character is received over the USB VCP port, a KeyboardInterrupt exception is raised.

Set to -1 to disable this interrupt feature. This is useful when you want to send raw bytes over the USB VCP port.

USB_VCP.**isconnected**()
Return `True` if USB is connected as a serial device, else `False`.

USB_VCP.**any**()
Return `True` if any characters waiting, else `False`.

USB_VCP.**close**()
This method does nothing. It exists so the USB_VCP object can act as a file.

USB_VCP.**read**([*nbytes*])
Read at most `nbytes` from the serial device and return them as a bytes object. If `nbytes` is not specified then the method reads all available bytes from the serial device. USB_VCP *stream* implicitly works in non-blocking mode, so if no pending data available, this method will return immediately with `None` value.

USB_VCP.**readinto**(*buf*[, *maxlen*])
Read bytes from the serial device and store them into `buf`, which should be a buffer-like object. At most `len(buf)` bytes are read. If `maxlen` is given and then at most `min(maxlen, len(buf))` bytes are read.

Returns the number of bytes read and stored into `buf` or `None` if no pending data available.

USB_VCP.**readline**()
Read a whole line from the serial device.

Returns a bytes object containing the data, including the trailing newline character or `None` if no pending data available.

USB_VCP.**readlines**()
Read as much data as possible from the serial device, breaking it into lines.

Returns a list of bytes objects, each object being one of the lines. Each line will include the newline character.

USB_VCP.**write**(*buf*)
Write the bytes from `buf` to the serial device.

Returns the number of bytes written.

USB_VCP.**recv**(*data*, *, *timeout=5000*)
Receive data on the bus:

- `data` can be an integer, which is the number of bytes to receive, or a mutable buffer, which will be filled with received bytes.
- `timeout` is the timeout in milliseconds to wait for the receive.

Return value: if `data` is an integer then a new buffer of the bytes received, otherwise the number of bytes read into `data` is returned.

USB_VCP.**send**(*data*, *, *timeout=5000*)
Send data over the USB VCP:

- `data` is the data to send (an integer to send, or a buffer object).
- `timeout` is the timeout in milliseconds to wait for the send.

Return value: number of bytes sent.

USB_VCP.**irq**(*handler=None*, *trigger=IRQ_RX*, *hard=False*)
Register *handler* to be called whenever an event specified by *trigger* occurs. The *handler* function must take exactly one argument, which will be the USB VCP object. Pass in `None` to disable the callback.

Valid values for *trigger* are:

- `USB_VCP.IRQ_RX`: new data is available for reading from the USB VCP object.

Constants

USB_VCP.**RTS**
USB_VCP.**CTS**
to select the flow control type.

USB_VCP.**IRQ_RX**
IRQ trigger values for *USB_VCP.irq()*.

stm functionality specific to STM32 MCUs

This module provides functionality specific to STM32 microcontrollers, including direct access to peripheral registers.

Memory access

The module exposes three objects used for raw memory access.

stm.**mem8**
Read/write 8 bits of memory.

stm.**mem16**
Read/write 16 bits of memory.

stm.**mem32**
Read/write 32 bits of memory.

Use subscript notation `[...]` to index these objects with the address of interest.

These memory objects can be used in combination with the peripheral register constants to read and write registers of the MCU hardware peripherals, as well as all other areas of address space.

Peripheral register constants

The module defines constants for registers which are generated from CMSIS header files, and the constants available depend on the microcontroller series that is being compiled for. Examples of some constants include:

stm.**GPIOA**
Base address of the GPIOA peripheral.

stm.**GPIOB**
Base address of the GPIOB peripheral.

stm.**GPIO_BSRR**
Offset of the GPIO bit set/reset register.

stm.**GPIO_IDR**
Offset of the GPIO input data register.

stm.**GPIO_ODR**
Offset of the GPIO output data register.

Constants that are named after a peripheral, like `GPIOA`, are the absolute address of that peripheral. Constants that have a prefix which is the name of a peripheral, like `GPIO_BSRR`, are relative offsets of the register. Accessing peripheral registers requires adding the absolute base address of the peripheral and the relative register offset. For example `GPIOA + GPIO_BSRR` is the full, absolute address of the `GPIOA->BSRR` register.

Example use:

```
# set PA2 high
stm.mem32[stm.GPIOA + stm.GPIO_BSRR] = 1 << 2

# read PA3
value = (stm.mem32[stm.GPIOA + stm.GPIO_IDR] >> 3) & 1
```

Functions specific to STM32WBxx MCUs

These functions are available on STM32WBxx microcontrollers, and interact with the second CPU, the RF core.

stm.**rfcore_status**()
: Returns the status of the second CPU as an integer (the first word of device info table).

stm.**rfcore_fw_version**(*id*)
: Get the version of the firmware running on the second CPU. Pass in 0 for *id* to get the FUS version, and 1 to get the WS version.

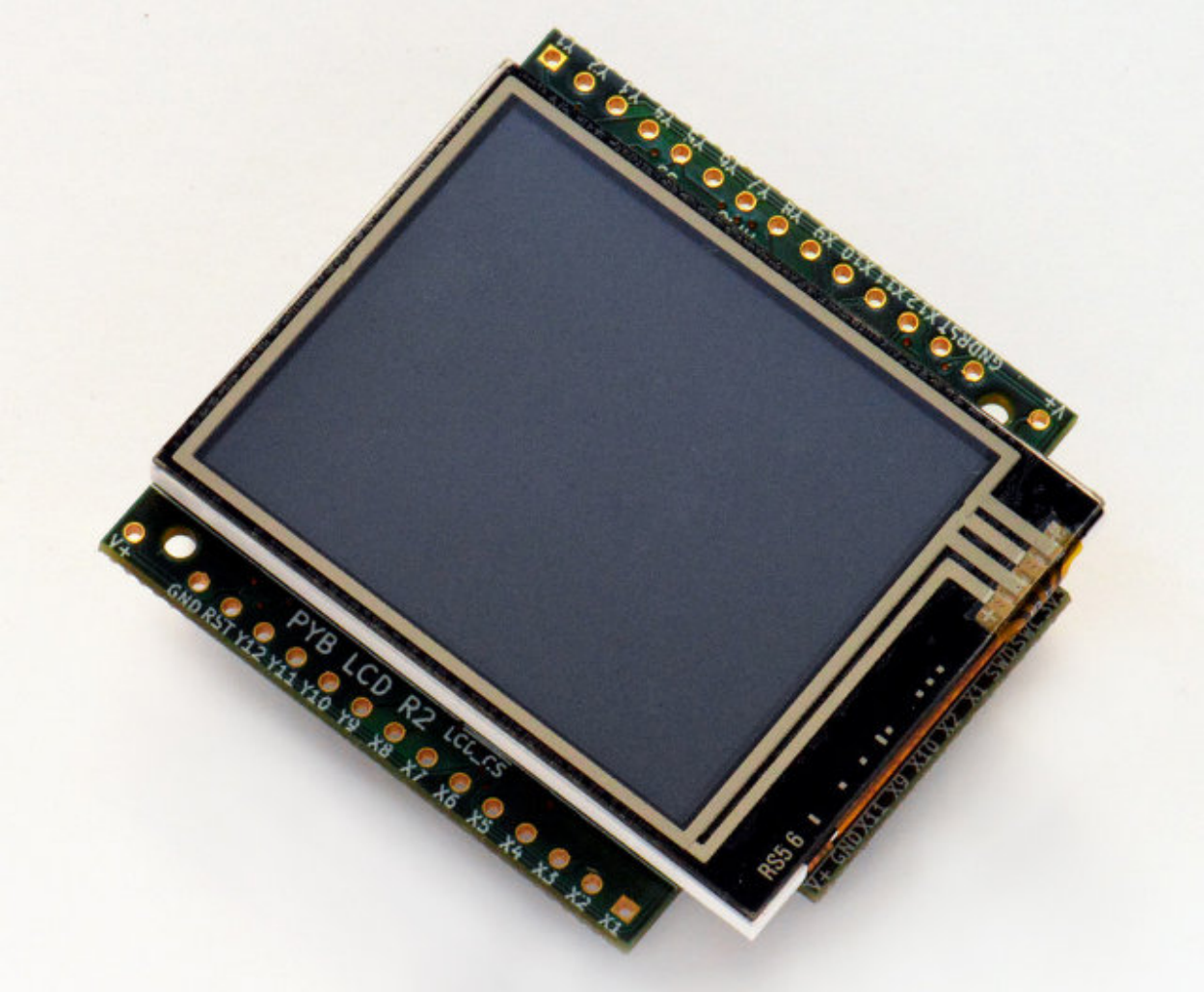
  Returns a 5-tuple with the full version number.

stm.**rfcore_sys_hci**(*ogf*, *ocf*, *data*, *timeout_ms=0*)
: Execute a HCI command on the SYS channel. The execution is synchronous.

  Returns a bytes object with the result of the SYS command.

lcd160cr control of LCD160CR display

This module provides control of the MicroPython LCD160CR display.



Further resources are available via the following links:

- LCD160CRv1.0 reference manual (100KiB PDF)
- LCD160CRv1.0 schematics (1.6MiB PDF)

class LCD160CR

The LCD160CR class provides an interface to the display. Create an instance of this class and use its methods to draw to the LCD and get the status of the touch panel.

For example:

```
import lcd160cr

lcd = lcd160cr.LCD160CR('X')
lcd.set_orient(lcd160cr.PORTRAIT)
lcd.set_pos(0, 0)
lcd.set_text_color(lcd.rgb(255, 0, 0), lcd.rgb(0, 0, 0))
lcd.set_font(1)
```

(continues on next page)

(continued from previous page)

```
lcd.write('Hello MicroPython!')
print('touch:', lcd.get_touch())
```

Constructors

class lcd160cr.**LCD160CR**(*connect=None*, ***, *pwr=None*, *i2c=None*, *spi=None*, *i2c_addr=98*)

Construct an LCD160CR object. The parameters are:

- *connect* is a string specifying the physical connection of the LCD display to the board; valid values are X, Y, XY, YX. Use X when the display is connected to a pyboard in the X-skin position, and Y when connected in the Y-skin position. XY and YX are used when the display is connected to the right or left side of the pyboard, respectively.
- *pwr* is a Pin object connected to the LCDs power/enabled pin.
- *i2c* is an I2C object connected to the LCDs I2C interface.
- *spi* is an SPI object connected to the LCDs SPI interface.
- *i2c_addr* is the I2C address of the display.

One must specify either a valid *connect* or all of *pwr*, *i2c* and *spi*. If a valid *connect* is given then any of *pwr*, *i2c* or *spi* which are not passed as parameters (i.e. they are `None`) will be created based on the value of *connect*. This allows to override the default interface to the display if needed.

The default values are:

- X is for the X-skin and uses: `pwr=Pin("X4")`, `i2c=I2C("X")`, `spi=SPI("X")`
- Y is for the Y-skin and uses: `pwr=Pin("Y4")`, `i2c=I2C("Y")`, `spi=SPI("Y")`
- XY is for the right-side and uses: `pwr=Pin("X4")`, `i2c=I2C("Y")`, `spi=SPI("X")`
- YX is for the left-side and uses: `pwr=Pin("Y4")`, `i2c=I2C("X")`, `spi=SPI("Y")`

See this image for how the display can be connected to the pyboard.

Static methods

static LCD160CR.**rgb**(*r*, *g*, *b*)

Return a 16-bit integer representing the given rgb color values. The 16-bit value can be used to set the font color (see *LCD160CR.set_text_color()*) pen color (see *LCD160CR.set_pen()*) and draw individual pixels.

LCD160CR.clip_line(data, w, h):

Clip the given line data. This is for internal use.

Instance members

The following instance members are publicly accessible.

LCD160CR.**w**

LCD160CR.**h**

The width and height of the display, respectively, in pixels. These members are updated when calling *LCD160CR.set_orient()* and should be considered read-only.

Setup commands

LCD160CR.**set_power**(*on*)
Turn the display on or off, depending on the given value of *on*: 0 or `False` will turn the display off, and 1 or `True` will turn it on.

LCD160CR.**set_orient**(*orient*)
Set the orientation of the display. The *orient* parameter can be one of *PORTRAIT*, *LANDSCAPE*, *PORTRAIT_UPSIDEDOWN*, *LANDSCAPE_UPSIDEDOWN*.

LCD160CR.**set_brightness**(*value*)
Set the brightness of the display, between 0 and 31.

LCD160CR.**set_i2c_addr**(*addr*)
Set the I2C address of the display. The *addr* value must have the lower 2 bits cleared.

LCD160CR.**set_uart_baudrate**(*baudrate*)
Set the baudrate of the UART interface.

LCD160CR.**set_startup_deco**(*value*)
Set the start-up decoration of the display. The *value* parameter can be a logical or of *STARTUP_DECO_NONE*, *STARTUP_DECO_MLOGO*, *STARTUP_DECO_INFO*.

LCD160CR.**save_to_flash**()
Save the following parameters to flash so they persist on restart and power up: initial decoration, orientation, brightness, UART baud rate, I2C address.

Pixel access methods

The following methods manipulate individual pixels on the display.

LCD160CR.**set_pixel**(*x*, *y*, *c*)
Set the specified pixel to the given color. The color should be a 16-bit integer and can be created by *LCD160CR.rgb()*.

LCD160CR.**get_pixel**(*x*, *y*)
Get the 16-bit value of the specified pixel.

LCD160CR.**get_line**(*x*, *y*, *buf*)
Low-level method to get a line of pixels into the given buffer. To read *n* pixels *buf* should be *2*n+1* bytes in length. The first byte is a dummy byte and should be ignored, and subsequent bytes represent the pixels in the line starting at coordinate *(x, y)*.

LCD160CR.**screen_dump**(*buf*, *x=0*, *y=0*, *w=None*, *h=None*)
Dump the contents of the screen to the given buffer. The parameters *x* and *y* specify the starting coordinate, and *w* and *h* the size of the region. If *w* or *h* are `None` then they will take on their maximum values, set by the size of the screen minus the given *x* and *y* values. *buf* should be large enough to hold `2*w*h` bytes. If its smaller then only the initial horizontal lines will be stored.

LCD160CR.**screen_load**(*buf*)
Load the entire screen from the given buffer.

Drawing text

To draw text one sets the position, color and font, and then uses *LCD160CR.write* to draw the text.

LCD160CR.**set_pos**(*x*, *y*)
Set the position for text output using *LCD160CR.write()*. The position is the upper-left corner of the text.

LCD160CR.**set_text_color**(*fg*, *bg*)
Set the foreground and background color of the text.

LCD160CR.**set_font**(*font*, *scale=0*, *bold=0*, *trans=0*, *scroll=0*)
Set the font for the text. Subsequent calls to *write* will use the newly configured font. The parameters are:

- *font* is the font family to use, valid values are 0, 1, 2, 3.
- *scale* is a scaling value for each character pixel, where the pixels are drawn as a square with side length equal to *scale* + *1*. The value can be between 0 and 63.
- *bold* controls the number of pixels to overdraw each character pixel, making a bold effect. The lower 2 bits of *bold* are the number of pixels to overdraw in the horizontal direction, and the next 2 bits are for the vertical direction. For example, a *bold* value of 5 will overdraw 1 pixel in both the horizontal and vertical directions.
- *trans* can be either 0 or 1 and if set to 1 the characters will be drawn with a transparent background.
- *scroll* can be either 0 or 1 and if set to 1 the display will do a soft scroll if the text moves to the next line.

LCD160CR.**write**(*s*)
Write text to the display, using the current position, color and font. As text is written the position is automatically incremented. The display supports basic VT100 control codes such as newline and backspace.

Drawing primitive shapes

Primitive drawing commands use a foreground and background color set by the *set_pen* method.

LCD160CR.**set_pen**(*line*, *fill*)
Set the line and fill color for primitive shapes.

LCD160CR.**erase**()
Erase the entire display to the pen fill color.

LCD160CR.**dot**(*x*, *y*)
Draw a single pixel at the given location using the pen line color.

LCD160CR.**rect**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_outline**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_interior**(*x*, *y*, *w*, *h*)
Draw a rectangle at the given location and size using the pen line color for the outline, and the pen fill color for the interior. The *rect* method draws the outline and interior, while the other methods just draw one or the other.

LCD160CR.**line**(*x1*, *y1*, *x2*, *y2*)
Draw a line between the given coordinates using the pen line color.

LCD160CR.**dot_no_clip**(*x*, *y*)

LCD160CR.**rect_no_clip**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_outline_no_clip**(*x*, *y*, *w*, *h*)

LCD160CR.**rect_interior_no_clip**(*x*, *y*, *w*, *h*)

LCD160CR.**line_no_clip**(*x1*, *y1*, *x2*, *y2*)
: These methods are as above but dont do any clipping on the input coordinates. They are faster than the clipping versions and can be used when you know that the coordinates are within the display.

LCD160CR.**poly_dot**(*data*)
: Draw a sequence of dots using the pen line color. The *data* should be a buffer of bytes, with each successive pair of bytes corresponding to coordinate pairs (x, y).

LCD160CR.**poly_line**(*data*)
: Similar to *LCD160CR.poly_dot()* but draws lines between the dots.

Touch screen methods

LCD160CR.**touch_config**(*calib=False*, *save=False*, *irq=None*)
: Configure the touch panel:

  - If *calib* is `True` then the call will trigger a touch calibration of the resistive touch sensor. This requires the user to touch various parts of the screen.
  - If *save* is `True` then the touch parameters will be saved to NVRAM to persist across reset/power up.
  - If *irq* is `True` then the display will be configured to pull the IRQ line low when a touch force is detected. If *irq* is `False` then this feature is disabled. If *irq* is `None` (the default value) then no change is made to this setting.

LCD160CR.**is_touched**()
: Returns a boolean: `True` if there is currently a touch force on the screen, `False` otherwise.

LCD160CR.**get_touch**()
: Returns a 3-tuple of: *(active, x, y)*. If there is currently a touch force on the screen then *active* is 1, otherwise it is 0. The *x* and *y* values indicate the position of the current or most recent touch.

Advanced commands

LCD160CR.**set_spi_win**(*x*, *y*, *w*, *h*)
: Set the window that SPI data is written to.

LCD160CR.**fast_spi**(*flush=True*)
: Ready the display to accept RGB pixel data on the SPI bus, resetting the location of the first byte to go to the top-left corner of the window set by *LCD160CR.set_spi_win()*. The method returns an SPI object which can be used to write the pixel data.

  Pixels should be sent as 16-bit RGB values in the 5-6-5 format. The destination counter will increase as data is sent, and data can be sent in arbitrary sized chunks. Once the destination counter reaches the end of the window specified by *LCD160CR.set_spi_win()* it will wrap around to the top-left corner of that window.

LCD160CR.**show_framebuf**(*buf*)
: Show the given buffer on the display. *buf* should be an array of bytes containing the 16-bit RGB values for the pixels, and they will be written to the area specified by *LCD160CR.set_spi_win()*, starting from the top-left corner.

  The framebuf module can be used to construct frame buffers and provides drawing primitives. Using a frame buffer will improve performance of animations when compared to drawing directly to the screen.

LCD160CR.**set_scroll**(*on*)
: Turn scrolling on or off. This controls globally whether any window regions will scroll.

LCD160CR.**set_scroll_win**(*win*, *x=- 1*, *y=0*, *w=0*, *h=0*, *vec=0*, *pat=0*, *fill=0x07e0*, *color=0*)
Configure a window region for scrolling:

- *win* is the window id to configure. There are 0..7 standard windows for general purpose use. Window 8 is the text scroll window (the ticker).
- *x*, *y*, *w*, *h* specify the location of the window in the display.
- *vec* specifies the direction and speed of scroll: it is a 16-bit value of the form `0bF.ddSSSSSSSSSSSS`. *dd* is 0, 1, 2, 3 for +x, +y, -x, -y scrolling. *F* sets the speed format, with 0 meaning that the window is shifted *S % 256* pixel every frame, and 1 meaning that the window is shifted 1 pixel every *S* frames.
- *pat* is a 16-bit pattern mask for the background.
- *fill* is the fill color.
- *color* is the extra color, either of the text or pattern foreground.

LCD160CR.**set_scroll_win_param**(*win*, *param*, *value*)
Set a single parameter of a scrolling window region:

- *win* is the window id, 0..8.
- *param* is the parameter number to configure, 0..7, and corresponds to the parameters in the *set_scroll_win* method.
- *value* is the value to set.

LCD160CR.**set_scroll_buf**(*s*)
Set the string for scrolling in window 8. The parameter *s* must be a string with length 32 or less.

LCD160CR.**jpeg**(*buf*)
Display a JPEG. *buf* should contain the entire JPEG data. JPEG data should not include EXIF information. The following encodings are supported: Baseline DCT, Huffman coding, 8 bits per sample, 3 color components, YCbCr4:2:2. The origin of the JPEG is set by *LCD160CR.set_pos()*.

LCD160CR.**jpeg_start**(*total_len*)

LCD160CR.**jpeg_data**(*buf*)
Display a JPEG with the data split across multiple buffers. There must be a single call to *jpeg_start* to begin with, specifying the total number of bytes in the JPEG. Then this number of bytes must be transferred to the display using one or more calls to the *jpeg_data* command.

LCD160CR.**feed_wdt**()
The first call to this method will start the displays internal watchdog timer. Subsequent calls will feed the watchdog. The timeout is roughly 30 seconds.

LCD160CR.**reset**()
Reset the display.

Constants

lcd160cr.**PORTRAIT**
lcd160cr.**LANDSCAPE**
lcd160cr.**PORTRAIT_UPSIDEDOWN**
lcd160cr.**LANDSCAPE_UPSIDEDOWN**
Orientations of the display, used by *LCD160CR.set_orient()*.

lcd160cr.**STARTUP_DECO_NONE**
lcd160cr.**STARTUP_DECO_MLOGO**

lcd160cr.**STARTUP_DECO_INFO**
Types of start-up decoration, can be ORed together, used by *LCD160CR.set_startup_deco()*.

1.3.2 Libraries specific to the WiPy

The following libraries and classes are specific to the WiPy.

wipy – WiPy specific features

The wipy module contains functions to control specific features of the WiPy, such as the heartbeat LED.

Functions

wipy.**heartbeat**([*enable*])
Get or set the state (enabled or disabled) of the heartbeat LED. Accepts and returns boolean values (True or False).

class ADCWiPy – analog to digital conversion

Note: This class is a non-standard ADC implementation for the WiPy. It is available simply as machine.ADC on the WiPy but is named in the documentation below as machine.ADCWiPy to distinguish it from the more general *machine.ADC* class.

Usage:

```
import machine

adc = machine.ADC()             # create an ADC object
apin = adc.channel(pin='GP3')   # create an analog pin on GP3
val = apin()                    # read an analog value
```

Constructors

class machine.**ADCWiPy**(*id=0*, ***, *bits=12*)
Create an ADC object associated with the given pin. This allows you to then read analog values on that pin. For more info check the pinout and alternate functions table.

Warning: ADC pin input range is 0-1.4V (being 1.8V the absolute maximum that it can withstand). When GP2, GP3, GP4 or GP5 are remapped to the ADC block, 1.8 V is the maximum. If these pins are used in digital mode, then the maximum allowed input is 3.6V.

Methods

ADCWiPy.**channel**(*id*, *, *pin*)
Create an analog pin. If only channel ID is given, the correct pin will be selected. Alternatively, only the pin can be passed and the correct channel will be selected. Examples:

```
# all of these are equivalent and enable ADC channel 1 on GP3
apin = adc.channel(1)
apin = adc.channel(pin='GP3')
apin = adc.channel(id=1, pin='GP3')
```

ADCWiPy.**init**()
Enable the ADC block.

ADCWiPy.**deinit**()
Disable the ADC block.

class ADCChannel read analog values from internal or external sources

ADC channels can be connected to internal points of the MCU or to GPIO pins. ADC channels are created using the ADC.channel method.

machine.**adcchannel**()
Fast method to read the channel value.

adcchannel.**value**()
Read the channel value.

adcchannel.**init**()
Re-init (and effectively enable) the ADC channel.

adcchannel.**deinit**()
Disable the ADC channel.

class TimerWiPy – control hardware timers

Note: This class is a non-standard Timer implementation for the WiPy. It is available simply as `machine.Timer` on the WiPy but is named in the documentation below as `machine.TimerWiPy` to distinguish it from the more general *machine.Timer* class.

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. MicroPythons 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 behaviour (which thus wont be portable to other boards).

See discussion of *important constraints* on Timer callbacks.

Note: Memory cant be allocated inside irq handlers (an interrupt) and so exceptions raised within a handler dont give much information. See *micropython.alloc_emergency_exception_buf()* for how to get around this limitation.

Constructors

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

Methods

TimerWiPy.**init**(*mode*, *, *width=16*)
Initialise the timer. Example:

```
tim.init(Timer.PERIODIC)              # periodic 16-bit timer
tim.init(Timer.ONE_SHOT, width=32)    # one shot 32-bit timer
```

Keyword arguments:

- `mode` can be one of:
  - `TimerWiPy.ONE_SHOT` - The timer runs once until the configured period of the channel expires.
  - `TimerWiPy.PERIODIC` - The timer runs periodically at the configured frequency of the channel.
  - `TimerWiPy.PWM` - Output a PWM signal on a pin.
- `width` must be either 16 or 32 (bits). For really low frequencies < 5Hz (or large periods), 32-bit timers should be used. 32-bit mode is only available for `ONE_SHOT` AND `PERIODIC` modes.

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

TimerWiPy.**channel**(*channel*, **, *freq*, *period*, *polarity=TimerWiPy.POSITIVE*, *duty_cycle=0*)
If only a channel identifier passed, then a previously initialized channel object is returned (or `None` if there is no previous channel).

Otherwise, a TimerChannel object is initialized and returned.

The operating mode is is the one configured to the Timer object that was used to create the channel.

- `channel` if the width of the timer is 16-bit, then must be either `TIMER.A`, `TIMER.B`. If the width is 32-bit then it **must be** `TIMER.A | TIMER.B`.

Keyword only arguments:

- `freq` sets the frequency in Hz.
- `period` sets the period in microseconds.

Note: Either `freq` or `period` must be given, never both.

- `polarity` this is applicable for `PWM`, and defines the polarity of the duty cycle
- `duty_cycle` only applicable to `PWM`. Its a percentage (0.00-100.00). Since the WiPy doesnt support floating point numbers the duty cycle must be specified in the range 0-10000, where 10000 would represent 100.00, 5050 represents 50.50, and so on.

Note: When the channel is in PWM mode, the corresponding pin is assigned automatically, therefore theres no need to assign the alternate function of the pin via the `Pin` class. The pins which support PWM functionality are the following:

- GP24 on Timer 0 channel A.
- GP25 on Timer 1 channel A.
- GP9 on Timer 2 channel B.
- GP10 on Timer 3 channel A.
- GP11 on Timer 3 channel B.

class TimerChannel setup a channel for a timer

Timer channels are used to generate/capture a signal using a timer.

TimerChannel objects are created using the Timer.channel() method.

Methods

timerchannel.**irq**(*, *trigger*, *priority=1*, *handler=None*)

The behaviour of this callback is heavily dependent on the operating mode of the timer channel:

- If mode is TimerWiPy.PERIODIC the callback is executed periodically with the configured frequency or period.
- If mode is TimerWiPy.ONE_SHOT the callback is executed once when the configured timer expires.
- If mode is TimerWiPy.PWM the callback is executed when reaching the duty cycle value.

The accepted params are:

- priority level of the interrupt. Can take values in the range 1-7. Higher values represent higher priorities.
- handler is an optional function to be called when the interrupt is triggered.
- trigger must be TimerWiPy.TIMEOUT when the operating mode is either TimerWiPy.PERIODIC or TimerWiPy.ONE_SHOT. In the case that mode is TimerWiPy.PWM then trigger must be equal to TimerWiPy.MATCH.

Returns a callback object.

timerchannel.**freq**([*value*])

Get or set the timer channel frequency (in Hz).

timerchannel.**period**([*value*])

Get or set the timer channel period (in microseconds).

timerchannel.**duty_cycle**([*value*])

Get or set the duty cycle of the PWM signal. Its a percentage (0.00-100.00). Since the WiPy doesnt support floating point numbers the duty cycle must be specified in the range 0-10000, where 10000 would represent 100.00, 5050 represents 50.50, and so on.

Constants

`TimerWiPy.`**ONE_SHOT**

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

1.3.3 Libraries specific to the ESP8266 and ESP32

The following libraries are specific to the ESP8266 and ESP32.

esp functions related to the ESP8266 and ESP32

The `esp` module contains specific functions related to both the ESP8266 and ESP32 modules. Some functions are only available on one or the other of these ports.

Functions

`esp.`**sleep_type**([*sleep_type*])
Note: ESP8266 only

Get or set the sleep type.

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

The possible sleep types are defined as constants:

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

The system enters the set sleep mode automatically when possible.

`esp.`**deepsleep**(*time_us=0*, */*)
Note: ESP8266 only - use *machine.deepsleep()* on ESP32

Enter deep sleep.

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

`esp.`**flash_id**()
Note: ESP8266 only

Read the device ID of the flash memory.

`esp.`**flash_size**()
Read the total size of the flash memory.

`esp.`**flash_user_start**()
Read the memory offset at which the user flash space begins.

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

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

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

esp.**set_native_code_location**(*start*, *length*)

Note: ESP8266 only

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

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

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

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

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

esp32 functionality specific to the ESP32

The `esp32` module contains functions and classes specifically aimed at controlling ESP32 modules.

Functions

esp32.**wake_on_touch**(*wake*)

Configure whether or not a touch will wake the device from sleep. *wake* should be a boolean value.

esp32.**wake_on_ext0**(*pin*, *level*)

Configure how EXT0 wakes the device from sleep. *pin* can be `None` or a valid Pin object. *level* should be `esp32.WAKEUP_ALL_LOW` or `esp32.WAKEUP_ANY_HIGH`.

esp32.**wake_on_ext1**(*pins*, *level*)

Configure how EXT1 wakes the device from sleep. *pins* can be `None` or a tuple/list of valid Pin objects. *level* should be `esp32.WAKEUP_ALL_LOW` or `esp32.WAKEUP_ANY_HIGH`.

esp32.**raw_temperature**()

Read the raw value of the internal temperature sensor, returning an integer.

esp32.**hall_sensor**()

Read the raw value of the internal Hall sensor, returning an integer.

esp32.**idf_heap_info**(*capabilities*)

Returns information about the ESP-IDF heap memory regions. One of them contains the MicroPython heap and the others are used by ESP-IDF, e.g., for network buffers and other data. This data is useful to get a sense of how much memory is available to ESP-IDF and the networking stack in particular. It may shed some light on situations where ESP-IDF operations fail due to allocation failures. The information returned is *not* useful to troubleshoot Python allocation failures, use *micropython.mem_info()* instead.

The capabilities parameter corresponds to ESP-IDFs `MALLOC_CAP_XXX` values but the two most useful ones are predefined as *esp32.HEAP_DATA* for data heap regions and *esp32.HEAP_EXEC* for executable regions as used by the native code emitter.

The return value is a list of 4-tuples, where each 4-tuple corresponds to one heap and contains: the total bytes, the free bytes, the largest free block, and the minimum free seen over time.

Example after booting:

```
>>> import esp32; esp32.idf_heap_info(esp32.HEAP_DATA)
[(240, 0, 0, 0), (7288, 0, 0, 0), (16648, 4, 4, 4), (79912, 35712, 35512, 35108),
 (15072, 15036, 15036, 15036), (113840, 0, 0, 0)]
```

Flash partitions

This class gives access to the partitions in the devices flash memory and includes methods to enable over-the-air (OTA) updates.

class esp32.**Partition**(*id*)
Create an object representing a partition. *id* can be a string which is the label of the partition to retrieve, or one of the constants: `BOOT` or `RUNNING`.

classmethod Partition.**find**(*type=TYPE_APP*, *subtype=0xff*, *label=None*)
Find a partition specified by *type*, *subtype* and *label*. Returns a (possibly empty) list of Partition objects. Note: `subtype=0xff` matches any subtype and `label=None` matches any label.

Partition.**info**()
Returns a 6-tuple `(type, subtype, addr, size, label, encrypted)`.

Partition.**readblocks**(*block_num*, *buf*)
Partition.**readblocks**(*block_num*, *buf*, *offset*)

Partition.**writeblocks**(*block_num*, *buf*)
Partition.**writeblocks**(*block_num*, *buf*, *offset*)

Partition.**ioctl**(*cmd*, *arg*)
These methods implement the simple and *extended* block protocol defined by *os.AbstractBlockDev*.

Partition.**set_boot**()
Sets the partition as the boot partition.

Partition.**get_next_update**()
Gets the next update partition after this one, and returns a new Partition object. Typical usage is `Partition(Partition.RUNNING).get_next_update()` which returns the next partition to update given the current running one.

classmethod Partition.**mark_app_valid_cancel_rollback**()
Signals that the current boot is considered successful. Calling `mark_app_valid_cancel_rollback` is required on the first boot of a new partition to avoid an automatic rollback at the next boot. This uses the ESP-IDF app rollback feature with CONFIG_BOOTLOADER_APP_ROLLBACK_ENABLE and an `OSError(-261)` is raised if called on firmware that doesnt have the feature enabled. It is OK to call `mark_app_valid_cancel_rollback` on every boot and it is not necessary when booting firmare that was loaded using esptool.

Constants

Partition.**BOOT**
Partition.**RUNNING**
: Used in the *Partition* constructor to fetch various partitions: `BOOT` is the partition that will be booted at the next reset and `RUNNING` is the currently running partition.

Partition.**TYPE_APP**
Partition.**TYPE_DATA**
: Used in *Partition.find* to specify the partition type: `APP` is for bootable firmware partitions (typically labelled `factory`, `ota_0`, `ota_1`), and `DATA` is for other partitions, e.g. `nvs`, `otadata`, `phy_init`, `vfs`.

esp32.**HEAP_DATA**
esp32.**HEAP_EXEC**
: Used in *idf_heap_info*.

RMT

The RMT (Remote Control) module, specific to the ESP32, was originally designed to send and receive infrared remote control signals. However, due to a flexible design and very accurate (as low as 12.5ns) pulse generation, it can also be used to transmit or receive many other types of digital signals:

```
import esp32
from machine import Pin

r = esp32.RMT(0, pin=Pin(18), clock_div=8)
r  # RMT(channel=0, pin=18, source_freq=80000000, clock_div=8, idle_level=0)

# To apply a carrier frequency to the high output
r = esp32.RMT(0, pin=Pin(18), clock_div=8, tx_carrier=(38000, 50, 1))

# The channel resolution is 100ns (1/(source_freq/clock_div)).
r.write_pulses((1, 20, 2, 40), 0)  # Send 0 for 100ns, 1 for 2000ns, 0 for 200ns, 1 for␣
↪4000ns
```

The input to the RMT module is an 80MHz clock (in the future it may be able to configure the input clock but, for now, its fixed). `clock_div` *divides* the clock input which determines the resolution of the RMT channel. The numbers specified in `write_pulses` are multiplied by the resolution to define the pulses.

`clock_div` is an 8-bit divider (0-255) and each pulse can be defined by multiplying the resolution by a 15-bit (0-32,768) number. There are eight channels (0-7) and each can have a different clock divider.

So, in the example above, the 80MHz clock is divided by 8. Thus the resolution is (1/(80Mhz/8)) 100ns. Since the `start` level is 0 and toggles with each number, the bitstream is `0101` with durations of [100ns, 2000ns, 100ns, 4000ns].

For more details see Espressifs ESP-IDF RMT documentation..

> **Warning:** The current MicroPython RMT implementation lacks some features, most notably receiving pulses. RMT should be considered a *beta feature* and the interface may change in the future.

class esp32.**RMT**(*channel*, ***, *pin=None*, *clock_div=8*, *idle_level=False*, *tx_carrier=None*)
: This class provides access to one of the eight RMT channels. *channel* is required and identifies which RMT channel (0-7) will be configured. *pin*, also required, configures which Pin is bound to the RMT channel. *clock_div* is an 8-bit clock divider that divides the source clock (80MHz) to the RMT channel allowing the resolution to be

specified. *idle_level* specifies what level the output will be when no transmission is in progress and can be any value that converts to a boolean, with `True` representing high voltage and `False` representing low.

To enable the transmission carrier feature, *tx_carrier* should be a tuple of three positive integers: carrier frequency, duty percent (`0` to `100`) and the output level to apply the carrier to (a boolean as per *idle_level*).

`RMT.source_freq()`
Returns the source clock frequency. Currently the source clock is not configurable so this will always return 80MHz.

`RMT.clock_div()`
Return the clock divider. Note that the channel resolution is `1 / (source_freq / clock_div)`.

`RMT.wait_done(*, timeout=0)`
Returns `True` if the channel is idle or `False` if a sequence of pulses started with *RMT.write_pulses* is being transmitted. If the *timeout* keyword argument is given then block for up to this many milliseconds for transmission to complete.

`RMT.loop(enable_loop)`
Configure looping on the channel. *enable_loop* is bool, set to `True` to enable looping on the *next* call to *RMT.write_pulses*. If called with `False` while a looping sequence is currently being transmitted then the current loop iteration will be completed and then transmission will stop.

`RMT.write_pulses(duration, data=True)`
Begin transmitting a sequence. There are three ways to specify this:

Mode 1: *duration* is a list or tuple of durations. The optional *data* argument specifies the initial output level. The output level will toggle after each duration.

Mode 2: *duration* is a positive integer and *data* is a list or tuple of output levels. *duration* specifies a fixed duration for each.

Mode 3: *duration* and *data* are lists or tuples of equal length, specifying individual durations and the output level for each.

Durations are in integer units of the channel resolution (as described above), between 1 and 32767 units. Output levels are any value that can be converted to a boolean, with `True` representing high voltage and `False` representing low.

If transmission of an earlier sequence is in progress then this method will block until that transmission is complete before beginning the new sequence.

If looping has been enabled with *RMT.loop*, the sequence will be repeated indefinitely. Further calls to this method will block until the end of the current loop iteration before immediately beginning to loop the new sequence of pulses. Looping sequences longer than 126 pulses is not supported by the hardware.

Ultra-Low-Power co-processor

`class esp32.ULP`
This class provides access to the Ultra-Low-Power co-processor.

`ULP.set_wakeup_period(period_index, period_us)`
Set the wake-up period.

`ULP.load_binary(load_addr, program_binary)`
Load a *program_binary* into the ULP at the given *load_addr*.

`ULP.run(entry_point)`
Start the ULP running at the given *entry_point*.

Constants

esp32.**WAKEUP_ALL_LOW**
esp32.**WAKEUP_ANY_HIGH**
Selects the wake level for pins.

Non-Volatile Storage

This class gives access to the Non-Volatile storage managed by ESP-IDF. The NVS is partitioned into namespaces and each namespace contains typed key-value pairs. The keys are strings and the values may be various integer types, strings, and binary blobs. The driver currently only supports 32-bit signed integers and blobs.

Warning: Changes to NVS need to be committed to flash by calling the commit method. Failure to call commit results in changes being lost at the next reset.

class esp32.**NVS**(*namespace*)
Create an object providing access to a namespace (which is automatically created if not present).

NVS.**set_i32**(*key*, *value*)
Sets a 32-bit signed integer value for the specified key. Remember to call *commit*!

NVS.**get_i32**(*key*)
Returns the signed integer value for the specified key. Raises an OSError if the key does not exist or has a different type.

NVS.**set_blob**(*key*, *value*)
Sets a binary blob value for the specified key. The value passed in must support the buffer protocol, e.g. bytes, bytearray, str. (Note that esp-idf distinguishes blobs and strings, this method always writes a blob even if a string is passed in as value.) Remember to call *commit*!

NVS.**get_blob**(*key*, *buffer*)
Reads the value of the blob for the specified key into the buffer, which must be a bytearray. Returns the actual length read. Raises an OSError if the key does not exist, has a different type, or if the buffer is too small.

NVS.**erase_key**(*key*)
Erases a key-value pair.

NVS.**commit**()
Commits changes made by *set_xxx* methods to flash.

1.3.4 Libraries specific to the RP2040

The following libraries are specific to the RP2040, as used in the Raspberry Pi Pico.

rp2 functionality specific to the RP2040

The rp2 module contains functions and classes specific to the RP2040, as used in the Raspberry Pi Pico.

See the RP2040 Python datasheet for more information, and pico-micropython-examples for example code.

PIO related functions

The rp2 module includes functions for assembling PIO programs.

For running PIO programs, see *rp2.StateMachine*.

rp2.**asm_pio**(*, *out_init=None*, *set_init=None*, *sideset_init=None*, *in_shiftdir=0*, *out_shiftdir=0*, *autopush=False*, *autopull=False*, *push_thresh=32*, *pull_thresh=32*, *fifo_join=PIO.JOIN_NONE*)

Assemble a PIO program.

The following parameters control the initial state of the GPIO pins, as one of *PIO.IN_LOW*, *PIO.IN_HIGH*, *PIO.OUT_LOW* or *PIO.OUT_HIGH*. If the program uses more than one pin, provide a tuple, e.g. `out_init=(PIO.OUT_LOW, PIO.OUT_LOW)`.

- *out_init* configures the pins used for `out()` instructions.
- *set_init* configures the pins used for `set()` instructions. There can be at most 5.
- *sideset_init* configures the pins used side-setting. There can be at most 5.

The following parameters are used by default, but can be overridden in *StateMachine.init()*:

- *in_shiftdir* is the default direction the ISR will shift, either *PIO.SHIFT_LEFT* or *PIO.SHIFT_RIGHT*.
- *out_shiftdir* is the default direction the OSR will shift, either *PIO.SHIFT_LEFT* or *PIO.SHIFT_RIGHT*.
- *push_thresh* is the threshold in bits before auto-push or conditional re-pushing is triggered.
- *pull_thresh* is the threshold in bits before auto-push or conditional re-pushing is triggered.

The remaining parameters are:

- *autopush* configures whether auto-push is enabled.
- *autopull* configures whether auto-pull is enabled.
- *fifo_join* configures whether the 4-word TX and RX FIFOs should be combined into a single 8-word FIFO for one direction only. The options are *PIO.JOIN_NONE*, *PIO.JOIN_RX* and *PIO.JOIN_TX*.

rp2.**asm_pio_encode**(*instr*, *sideset_count*, *sideset_opt=False*)

Assemble a single PIO instruction. You usually want to use *asm_pio()* instead.

```
>>> rp2.asm_pio_encode("set(0, 1)", 0)
57345
```

class rp2.**PIOASMError**

This exception is raised from *asm_pio()* or *asm_pio_encode()* if there is an error assembling a PIO program.

PIO assembly language instructions

PIO state machines are programmed in a custom assembly language with nine core PIO-machine instructions. In MicroPython, PIO assembly routines are written as a Python function with the decorator `@rp2.asm_pio()`, and they use Python syntax. Such routines support standard Python variables and arithmetic, as well as the following custom functions that encode PIO instructions and direct the assembler. See sec 3.4 of the RP2040 datasheet for further details.

wrap_target() Specify the location where execution continues after program wrapping. By default this is the start of the PIO routine.

wrap() Specify the location where the program finishes and wraps around. If this directive is not used then it is added automatically at the end of the PIO routine. Wrapping does not cost any execution cycles.

label(label) Define a label called *label* at the current location. *label* can be a string or integer.

word(instr, label=None) Insert an arbitrary 16-bit word in the assembled output.

- *instr*: the 16-bit value
- *label*: if given, look up the label and logical-or the labels value with *instr*

jmp() This instruction takes two forms:

jmp(label)

- *label*: label to jump to unconditionally

jmp(cond, label)

- *cond*: the condition to check, one of:
  - `not_x`, `not_y`: true if register is zero
  - `x_dec`, `y_dec`: true if register is non-zero, and do post decrement
  - `x_not_y`: true if X is not equal to Y
  - `pin`: true if the input pin is set
  - `not_osre`: true if OSR is not empty (hasnt reached its threshold)
- *label*: label to jump to if condition is true

wait(polarity, src, index) Block, waiting for high/low on a pin or IRQ line.

- *polarity*: 0 or 1, whether to wait for a low or high value
- *src*: one of: `gpio` (absolute pin), `pin` (pin relative to StateMachines `in_base` argument), `irq`
- *index*: 0-31, the index for *src*

in_(src, bit_count) Shift data in from *src* to ISR.

- *src*: one of: `pins`, `x`, `y`, `null`, `isr`, `osr`
- *bit_count*: number of bits to shift in (1-32)

out(dest, bit_count) Shift data out from OSR to *dest*.

- *dest*: one of: `pins`, `x`, `y`, `pindirs`, `pc`, `isr`, `exec`
- *bit_count*: number of bits to shift out (1-32)

push() Push ISR to the RX FIFO, then clear ISR to zero. This instruction takes the following forms:

- push()
- push(block)

- push(noblock)
- push(iffull)
- push(iffull, block)
- push(iffull, noblock)

If `block` is used then the instruction stalls if the RX FIFO is full. The default is to block. If `iffull` is used then it only pushes if the input shift count has reached its threshold.

pull() Pull from the TX FIFO into OSR. This instruction takes the following forms:

- pull()
- pull(block)
- pull(noblock)
- pull(ifempty)
- pull(ifempty, block)
- pull(ifempty, noblock)

If `block` is used then the instruction stalls if the TX FIFO is empty. The default is to block. If `ifempty` is used then it only pulls if the output shift count has reached its threshold.

mov(dest, src) Move into *dest* the value from *src*.

- *dest*: one of: `pins`, `x`, `y`, `exec`, `pc`, `isr`, `osr`
- *src*: one of: `pins`, `x`, `y`, `null`, `status`, `isr`, `osr`; this argument can be optionally modified by wrapping it in `invert()` or `reverse()` (but not both together)

irq() Set or clear an IRQ flag. This instruction takes two forms:

irq(index)

- *index*: 0-7, or `rel(0)` to `rel(7)`

irq(mode, index)

- *mode*: one of: `block`, `clear`
- *index*: 0-7, or `rel(0)` to `rel(7)`

If `block` is used then the instruction stalls until the flag is cleared by another entity. If `clear` is used then the flag is cleared instead of being set. Relative IRQ indices add the state machine ID to the IRQ index with modulo-4 addition. IRQs 0-3 are visible from to the processor, 4-7 are internal to the state machines.

set(dest, data) Set *dest* with the value *data*.

- *dest*: `pins`, `x`, `y`, `pindirs`
- *data*: value (0-31)

nop() This is a pseudoinstruction that assembles to `mov(y, y)` and has no side effect.

.side(value) This is a modifier which can be applied to any instruction, and is used to control side-set pin values.

- *value*: the value (bits) to output on the side-set pins

.delay(value) This is a modifier which can be applied to any instruction, and specifies how many cycles to delay for after the instruction executes.

- *value*: cycles to delay, 0-31 (maximum value reduced if side-set pins are used)

[value] This is a modifier and is equivalent to `.delay(value)`.

Classes

class Flash – access to built-in flash storage

This class gives access to the SPI flash memory.

In most cases, to store persistent data on the device, youll want to use a higher-level abstraction, for example the filesystem via Pythons standard file API, but this interface is useful to *customise the filesystem configuration* or implement a low-level storage system for your application.

Constructors

class rp2.**Flash**
Gets the singleton object for accessing the SPI flash memory.

Methods

Flash.**readblocks**(*block_num*, *buf*)
Flash.**readblocks**(*block_num*, *buf*, *offset*)

Flash.**writeblocks**(*block_num*, *buf*)
Flash.**writeblocks**(*block_num*, *buf*, *offset*)

Flash.**ioctl**(*cmd*, *arg*)
These methods implement the simple and extended *block protocol* defined by `os.AbstractBlockDev`.

class PIO – advanced PIO usage

The `PIO` class gives access to an instance of the RP2040s PIO (programmable I/O) interface.

The preferred way to interact with PIO is using `rp2.StateMachine`, the PIO class is for advanced use.

For assembling PIO programs, see `rp2.asm_pio()`.

Constructors

class rp2.**PIO**(*id*)
Gets the PIO instance numbered *id*. The RP2040 has two PIO instances, numbered 0 and 1.

Raises a `ValueError` if any other argument is provided.

Methods

PIO.**add_program**(*program*)
Add the *program* to the instruction memory of this PIO instance.

The amount of memory available for programs on each PIO instance is limited. If there isnt enough space left in the PIOs program memory this method will raise `OSError(ENOMEM)`.

PIO.**remove_program**([*program*])
Remove *program* from the instruction memory of this PIO instance.

If no program is provided, it removes all programs.

It is not an error to remove a program which has already been removed.

PIO.**state_machine**(*id*[, *program*, ...])
Gets the state machine numbered *id*. On the RP2040, each PIO instance has four state machines, numbered 0 to 3.

Optionally initialize it with a *program*: see *StateMachine.init*.

```
>>> rp2.PIO(1).state_machine(3)
StateMachine(7)
```

PIO.**irq**(*handler=None*, *trigger=IRQ_SM0 | IRQ_SM1 | IRQ_SM2 | IRQ_SM3*, *hard=False*)
Returns the IRQ object for this PIO instance.

MicroPython only uses IRQ 0 on each PIO instance. IRQ 1 is not available.

Optionally configure it.

Constants

PIO.**IN_LOW**
PIO.**IN_HIGH**
PIO.**OUT_LOW**
PIO.**OUT_HIGH**
These constants are used for the *out_init*, *set_init*, and *sideset_init* arguments to *asm_pio*.

PIO.**SHIFT_LEFT**
PIO.**SHIFT_RIGHT**
These constants are used for the *in_shiftdir* and *out_shiftdir* arguments to *asm_pio* or *StateMachine.init*.

PIO.**JOIN_NONE**
PIO.**JOIN_TX**
PIO.**JOIN_RX**
These constants are used for the *fifo_join* argument to *asm_pio*.

PIO.**IRQ_SM0**
PIO.**IRQ_SM1**
PIO.**IRQ_SM2**
PIO.**IRQ_SM3**
These constants are used for the *trigger* argument to *PIO.irq*.

class StateMachine – access to the RP2040s programmable I/O interface

The *StateMachine* class gives access to the RP2040s PIO (programmable I/O) interface.

For assembling PIO programs, see *rp2.asm_pio()*.

Constructors

class rp2.**StateMachine**(*id*[, *program*, ...])

Get the state machine numbered *id*. The RP2040 has two identical PIO instances, each with 4 state machines: so there are 8 state machines in total, numbered 0 to 7.

Optionally initialize it with the given program *program*: see `StateMachine.init`.

Methods

StateMachine.**init**(*program*, *freq=- 1*, ***, *in_base=None*, *out_base=None*, *set_base=None*, *jmp_pin=None*, *sideset_base=None*, *in_shiftdir=None*, *out_shiftdir=None*, *push_thresh=None*, *pull_thresh=None*)

Configure the state machine instance to run the given *program*.

The program is added to the instruction memory of this PIO instance. If the instruction memory already contains this program, then its offset is re-used so as to save on instruction memory.

- *freq* is the frequency in Hz to run the state machine at. Defaults to the system clock frequency.

  The clock divider is computed as `system clock frequency / freq`, so there can be slight rounding errors.

  The minimum possible clock divider is one 65536th of the system clock: so at the default system clock frequency of 125MHz, the minimum value of *freq* is `1908`. To run state machines at slower frequencies, youll need to reduce the system clock speed with `machine.freq()`.

- *in_base* is the first pin to use for `in()` instructions.
- *out_base* is the first pin to use for `out()` instructions.
- *set_base* is the first pin to use for `set()` instructions.
- *jmp_pin* is the first pin to use for `jmp(pin, ...)` instructions.
- *sideset_base* is the first pin to use for side-setting.
- *in_shiftdir* is the direction the ISR will shift, either `PIO.SHIFT_LEFT` or `PIO.SHIFT_RIGHT`.
- *out_shiftdir* is the direction the OSR will shift, either `PIO.SHIFT_LEFT` or `PIO.SHIFT_RIGHT`.
- *push_thresh* is the threshold in bits before auto-push or conditional re-pushing is triggered.
- *pull_thresh* is the threshold in bits before auto-push or conditional re-pushing is triggered.

StateMachine.**active**([*value*])

Gets or sets whether the state machine is currently running.

```
>>> sm.active()
True
>>> sm.active(0)
False
```

StateMachine.**restart**()

Restarts the state machine and jumps to the beginning of the program.

This method clears the state machines internal state using the RP2040s `SM_RESTART` register. This includes:

- input and output shift counters
- the contents of the input shift register

- the delay counter
- the waiting-on-IRQ state
- a stalled instruction run using *StateMachine.exec()*

StateMachine.**exec**(*instr*)
Execute a single PIO instruction. Uses *asm_pio_encode* to encode the instruction from the given string *instr*.

```
>>> sm.exec("set(0, 1)")
```

StateMachine.**get**(*buf=None*, *shift=0*)
Pull a word from the state machines RX FIFO.

If the FIFO is empty, it blocks until data arrives (i.e. the state machine pushes a word).

The value is shifted right by *shift* bits before returning, i.e. the return value is `word >> shift`.

StateMachine.**put**(*value*, *shift=0*)
Push a word onto the state machines TX FIFO.

If the FIFO is full, it blocks until there is space (i.e. the state machine pulls a word).

The value is first shifted left by *shift* bits, i.e. the state machine receives `value << shift`.

StateMachine.**rx_fifo**()
Returns the number of words in the state machines RX FIFO. A value of 0 indicates the FIFO is empty.

Useful for checking if data is waiting to be read, before calling *StateMachine.get()*.

StateMachine.**tx_fifo**()
Returns the number of words in the state machines TX FIFO. A value of 0 indicates the FIFO is empty.

Useful for checking if there is space to push another word using *StateMachine.put()*.

StateMachine.**irq**(*handler=None*, *trigger=0 | 1*, *hard=False*)
Returns the IRQ object for the given StateMachine.

Optionally configure it.

1.3.5 Libraries specific to Zephyr

The following libraries are specific to the Zephyr port.

`zephyr` functionality specific to the Zephyr port

The `zephyr` module contains functions and classes specific to the Zephyr port.

Functions

zephyr.**is_preempt_thread**()
Returns true if the current thread is a preemptible thread.

Zephyr preemptible threads are those with non-negative priority values (low priority levels), which therefore, can be supplanted as soon as a higher or equal priority thread becomes ready.

zephyr.**current_tid**()
Returns the thread id of the current thread, which is used to reference the thread.

zephyr.**thread_analyze**()
Runs the Zephyr debug thread analyzer on the current thread and prints stack size statistics in the format:

thread_name-20s: STACK: unused available_stack_space usage stack_space_used / stack_size (percent_stack_space_used %); CPU: cpu_utilization %

- *CPU utilization is only printed if runtime statistics are configured via the ``CONFIG_THREAD_RUNTIME_STATS`` kconfig*

This function can only be accessed if CONFIG_THREAD_ANALYZER is configured for the port in zephyr/prj.conf. For more infomation, see documentation for Zephyr thread analyzer.

zephyr.**shell_exec**(*cmd_in*)
Executes the given command on an UART backend. This function can only be accessed if CONFIG_SHELL_BACKEND_SERIAL is configured for the port in zephyr/prj.conf.

A list of possible commands can be found in the documentation for Zephyr shell commands.

Classes

class DiskAccess – access to disk storage

Uses Zephyr Disk Access API.

This class allows access to storage devices on the board, such as support for SD card controllers and interfacing with SD cards via SPI. Disk devices are automatically detected and initialized on boot using Zephyr devicetree data.

The Zephyr disk access class enables the transfer of data between a disk device and an accessible memory buffer given a disk name, buffer, starting disk block, and number of sectors to read. MicroPython reads as many blocks as necessary to fill the buffer, so the number of sectors to read is found by dividing the buffer length by block size of the disk.

Constructors

class zephyr.**DiskAccess**(*disk_name*)
Gets an object for accessing disk memory of the specific disk. For accessing an SD card on the mimxrt1050_evk, disk_name would be SDHC. See board documentation and devicetree for usable disk names for your board (ex. RT boards use style USDHC#).

Methods

DiskAccess.**readblocks**(*block_num*, *buf*)
DiskAccess.**readblocks**(*block_num*, *buf*, *offset*)

DiskAccess.**writeblocks**(*block_num*, *buf*)
DiskAccess.**writeblocks**(*block_num*, *buf*, *offset*)

DiskAccess.**ioctl**(*cmd*, *arg*)
These methods implement the simple and extended *block protocol* defined by uos.AbstractBlockDev.

class FlashArea – access to built-in flash storage

Uses Zephyr flash map API.

This class allows access to device flash partition data. Flash area structs consist of a globally unique ID number, the name of the flash device the partition is in, the start offset (expressed in relation to the flash memory beginning address per partition), and the size of the partition that the device represents. For fixed flash partitions, data from the device tree is used; however, fixed flash partitioning is not enforced in MicroPython because MCUBoot is not enabled.

Constructors

class zephyr.**FlashArea**(*id*, *block_size*)
Gets an object for accessing flash memory at partition specified by `id` and with block size of `block_size`.

`id` values are integers correlating to fixed flash partitions defined in the devicetree. A commonly used partition is the designated flash storage area defined as `FlashArea.STORAGE` if `FLASH_AREA_LABEL_EXISTS(storage)` returns true at boot. Zephyr devicetree fixed flash partitions are `boot_partition`, `slot0_partition`, `slot1_partition`, and `scratch_partition`. Because MCUBoot is not enabled by default for MicroPython, these fixed partitions can be accessed by ID integer values 1, 2, 3, and 4, respectively.

Methods

FlashArea.**readblocks**(*block_num*, *buf*)
FlashArea.**readblocks**(*block_num*, *buf*, *offset*)

FlashArea.**writeblocks**(*block_num*, *buf*)
FlashArea.**writeblocks**(*block_num*, *buf*, *offset*)

FlashArea.**ioctl**(*cmd*, *arg*)
These methods implement the simple and extended *block protocol* defined by `uos.AbstractBlockDev`.

Additional Modules

zsensor Zephyr sensor bindings

The `zsensor` module contains a class for using sensors with Zephyr.

class Sensor sensor control for the Zephyr port

Use this class to access data from sensors on your board. See Zephyr documentation for sensor usage here: Sensors.

Sensors are defined in the Zephyr devicetree for each board. The quantities that a given sensor can measure are called a sensor channels. Sensors can have multiple channels to represent different axes of one property or different properties a sensor can measure. See *Channels* below for defined sensor channels.

Constructor

class zsensor.**Sensor**(*device_name*)
Device names are defined in the devicetree for your board. For example, the device name for the accelerometer in the FRDM-k64f board is FXOS8700.

Methods

Sensor.**measure**()
Obtains a measurement sample from the sensor device using Zephyr sensor_sample_fetch and stores it in an internal driver buffer as a useful value, a pair of (integer part of value, fractional part of value in 1-millionths). Returns none if successful or OSError value if failure.

Sensor.**get_float**(*sensor_channel*)
Returns the value of the sensor measurement sample as a float.

Sensor.**get_micros**(*sensor_channel*)
Returns the value of the sensor measurement sample in millionths. (Ex. value of `(1, 500000)` returns as `1500000`)

Sensor.**get_millis**(*sensor_channel*)
Returns the value of sensor measurement sample in thousandths. (Ex. value of `(1, 500000)` returns as `1500`)

Sensor.**get_int**(*sensor_channel*)
Returns only the integer value of the measurement sample. (Ex. value of `(1, 500000)` returns as `1`)

Channels

zsensor.**ACCEL_X**
Acceleration on the X axis, in m/s^2.

zsensor.**ACCEL_Y**
Acceleration on the Y axis, in m/s^2.

zsensor.**ACCEL_Z**
Acceleration on the Z axis, in m/s^2.

zsensor.**GYRO_X**
Angular velocity around the X axis, in radians/s.

zsensor.**GYRO_Y**
Angular velocity around the Y axis, in radians/s.

zsensor.**GYRO_Z**
Angular velocity around the Z axis, in radians/s.

zsensor.**MAGN_X**
Magnetic field on the X axis, in Gauss.

zsensor.**MAGN_Y**
Magnetic field on the Y axis, in Gauss.

zsensor.**MAGN_Z**
Magnetic field on the Z axis, in Gauss.

zsensor.**DIE_TEMP**
Device die temperature in degrees Celsius.

`zsensor.`**`PRESS`**
Pressure in kilopascal.

`zsensor.`**`PROX`**
Proximity. Dimensionless. A value of 1 indicates that an object is close.

`zsensor.`**`HUMIDITY`**
Humidity, in percent.

`zsensor.`**`LIGHT`**
Illuminance in visible spectrum, in lux.

`zsensor.`**`ALTITUDE`**
Altitude, in meters.

1.4 Extending built-in libraries from Python

In most cases, the above modules are actually named `umodule` rather than `module`, but MicroPython will alias any module prefixed with a `u` to the non-`u` version. However a file (or *frozen module*) named `module.py` will take precedence over this alias.

This allows the user to provide an extended implementation of a built-in library (perhaps to provide additional CPython compatibility). The user-provided module (in `module.py`) can still use the built-in functionality by importing `umodule` directly. This is used extensively in *micropython-lib*. See *Distribution packages, package management, and deploying applications* for more information.

This applies to both the Python standard libraries (e.g. `os`, `time`, etc), but also the MicroPython libraries too (e.g. `machine`, `bluetooth`, etc). The main exception is the port-specific libraries (`pyb`, `esp`, etc).

Other than when you specifically want to force the use of the built-in module, we recommend always using ``import module`` rather than ``import umodule``.

CHAPTER

TWO

MICROPYTHON LANGUAGE AND IMPLEMENTATION

MicroPython aims to implement the Python 3.4 standard (with selected features from later versions) with respect to language syntax, and most of the features of MicroPython are identical to those described by the Language Reference documentation at docs.python.org.

The MicroPython standard library is described in the *corresponding chapter*. The *MicroPython differences from CPython* chapter describes differences between MicroPython and CPython (which mostly concern standard library and types, but also some language-level features).

This chapter describes features and peculiarities of MicroPython implementation and the best practices to use them.

2.1 Glossary

baremetal A system without a (full-fledged) operating system, for example an *MCU*-based system. When running on a baremetal system, MicroPython effectively functions like a small operating system, running user programs and providing a command interpreter (*REPL*).

buffer protocol Any Python object that can be automatically converted into bytes, such as `bytes`, `bytearray`, `memoryview` and `str` objects, which all implement the buffer protocol.

board Typically this refers to a printed circuit board (PCB) containing a *microcontroller* and supporting components. MicroPython firmware is typically provided per-board, as the firmware contains both MCU-specific functionality but also board-level functionality such as drivers or pin names.

bytecode A compact representation of a Python program that generated by compiling the Python source code. This is what the VM actually executes. Bytecode is typically generated automatically at runtime and is invisible to the user. Note that while *CPython* and MicroPython both use bytecode, the format is different. You can also pre-compile source code offline using the *cross-compiler*.

callee-owned tuple This is a MicroPython-specific construct where, for efficiency reasons, some built-in functions or methods may re-use the same underlying tuple object to return data. This avoids having to allocate a new tuple for every call, and reduces heap fragmentation. Programs should not hold references to callee-owned tuples and instead only extract data from them (or make a copy).

CircuitPython A variant of MicroPython developed by Adafruit Industries.

CPython CPython is the reference implementation of the Python programming language, and the most well-known one. It is, however, one of many implementations (including Jython, IronPython, PyPy, and MicroPython). While MicroPythons implementation differs substantially from CPython, it aims to maintain as much compatibility as possible.

cross-compiler Also known as `mpy-cross`. This tool runs on your PC and converts a *.py file* containing MicroPython code into a *.mpy file* containing MicroPython bytecode. This means it loads faster (the board doesnt have to compile the code), and uses less space on flash (the bytecode is more space efficient).

driver A MicroPython library that implements support for a particular component, such as a sensor or display.

FFI Acronym for Foreign Function Interface. A mechanism used by the *MicroPython Unix port* to access operating system functionality. This is not available on *baremetal* ports.

filesystem Most MicroPython ports and boards provide a filesystem stored in flash that is available to user code via the standard Python file APIs such as `open()`. Some boards also make this internal filesystem accessible to the host via USB mass-storage.

frozen module A Python module that has been cross compiled and bundled into the firmware image. This reduces RAM requirements as the code is executed directly from flash.

Garbage Collector A background process that runs in Python (and MicroPython) to reclaim unused memory in the *heap*.

GPIO General-purpose input/output. The simplest means to control electrical signals (commonly referred to as pins) on a microcontroller. GPIO typically allows pins to be either input or output, and to set or get their digital value (logical 0 or 1). MicroPython abstracts GPIO access using the `machine.Pin` and `machine.Signal` classes.

GPIO port A group of *GPIO* pins, usually based on hardware properties of these pins (e.g. controllable by the same register).

heap A region of RAM where MicroPython stores dynamic data. It is managed automatically by the *Garbage Collector*. Different MCUs and boards have vastly different amounts of RAM available for the heap, so this will affect how complex your program can be.

interned string An optimisation used by MicroPython to improve the efficiency of working with strings. An interned string is referenced by its (unique) identity rather than its address and can therefore be quickly compared just by its identifier. It also means that identical strings can be de-duplicated in memory. String interning is almost always invisible to the user.

MCU Microcontroller. Microcontrollers usually have much less resources than a desktop, laptop, or phone, but are smaller, cheaper and require much less power. MicroPython is designed to be small and optimized enough to run on an average modern microcontroller.

micropython-lib MicroPython is (usually) distributed as a single executable/binary file with just few builtin modules. There is no extensive standard library comparable with *CPython*s. Instead, there is a related, but separate project micropython-lib which provides implementations for many modules from CPythons standard library.

Some of the modules are are implemented in pure Python, and are able to be used on all ports. However, the majority of these modules use *FFI* to access operating system functionality, and as such can only be used on the *MicroPython Unix port* (with limited support for Windows).

Unlike the *CPython* stdlib, micropython-lib modules are intended to be installed individually - either using manual copying or using *upip*.

MicroPython port MicroPython supports different *boards*, RTOSes, and OSes, and can be relatively easily adapted to new systems. MicroPython with support for a particular system is called a port to that system. Different ports may have widely different functionality. This documentation is intended to be a reference of the generic APIs available across different ports (MicroPython core). Note that some ports may still omit some APIs described here (e.g. due to resource constraints). Any such differences, and port-specific extensions beyond the MicroPython core functionality, would be described in the separate port-specific documentation.

MicroPython Unix port The unix port is one of the major *MicroPython ports*. It is intended to run on POSIX-compatible operating systems, like Linux, MacOS, FreeBSD, Solaris, etc. It also serves as the basis of Windows port. The Unix port is very useful for quick development and testing of the MicroPython language and machine-independent features. It can also function in a similar way to *CPython*s `python` executable.

.mpy file The output of the *cross-compiler*. A compiled form of a *.py file* that contains MicroPython bytecode instead of Python source code.

native Usually refers to native code, i.e. machine code for the target microcontroller (such as ARM Thumb, Xtensa, x86/x64). The `@native` decorator can be applied to a MicroPython function to generate native code instead of bytecode for that function, which will likely be faster but use more RAM.

port Usually short for *MicroPython port*, but could also refer to *GPIO port*.

.py file A file containing Python source code.

REPL An acronym for Read, Eval, Print, Loop. This is the interactive Python prompt, useful for debugging or testing short snippets of code. Most MicroPython boards make a REPL available over a UART, and this is typically accessible on a host PC via USB.

stream Also known as a file-like object. A Python object which provides sequential read-write access to the underlying data. A stream object implements a corresponding interface, which consists of methods like `read()`, `write()`, `readinto()`, `seek()`, `flush()`, `close()`, etc. A stream is an important concept in MicroPython; many I/O objects implement the stream interface, and thus can be used consistently and interchangeably in different contexts. For more information on streams in MicroPython, see the `io` module.

UART Acronym for Universal Asynchronous Receiver/Transmitter. This is a peripheral that sends data over a pair of pins (TX & RX). Many boards include a way to make at least one of the UARTs available to a host PC as a serial port over USB.

upip (Literally, micro pip). A package manager for MicroPython, inspired by *CPython*s pip, but much smaller and with reduced functionality. upip runs both on the *Unix port* and on *baremetal* ports which offer filesystem and networking support.

2.2 The MicroPython Interactive Interpreter Mode (aka REPL)

This section covers some characteristics of the MicroPython Interactive Interpreter Mode. A commonly used term for this is REPL (read-eval-print-loop) which will be used to refer to this interactive prompt.

2.2.1 Auto-indent

When typing python statements which end in a colon (for example if, for, while) then the prompt will change to three dots () and the cursor will be indented by 4 spaces. When you press return, the next line will continue at the same level of indentation for regular statements or an additional level of indentation where appropriate. If you press the backspace key then it will undo one level of indentation.

If your cursor is all the way back at the beginning, pressing RETURN will then execute the code that youve entered. The following shows what youd see after entering a for statement (the underscore shows where the cursor winds up):

```
>>> for i in range(30):
...     _
```

If you then enter an if statement, an additional level of indentation will be provided:

```
>>> for i in range(30):
...     if i > 3:
...         _
```

Now enter `break` followed by RETURN and press BACKSPACE:

```
>>> for i in range(30):
...     if i > 3:
```

(continues on next page)

(continued from previous page)

```
...         break
...     _
```

Finally type `print(i)`, press RETURN, press BACKSPACE and press RETURN again:

```
>>> for i in range(30):
...     if i > 3:
...         break
...     print(i)
...
0
1
2
3
>>>
```

Auto-indent wont be applied if the previous two lines were all spaces. This means that you can finish entering a compound statement by pressing RETURN twice, and then a third press will finish and execute.

2.2.2 Auto-completion

While typing a command at the REPL, if the line typed so far corresponds to the beginning of the name of something, then pressing TAB will show possible things that could be entered. For example, first import the machine module by entering `import machine` and pressing RETURN. Then type `m` and press TAB and it should expand to `machine`. Enter a dot `.` and press TAB again. You should see something like:

```
>>> machine.
__name__        info            unique_id       reset
bootloader      freq            rng             idle
sleep           deepsleep       disable_irq     enable_irq
Pin
```

The word will be expanded as much as possible until multiple possibilities exist. For example, type `machine.Pin.AF3` and press TAB and it will expand to `machine.Pin.AF3_TIM`. Pressing TAB a second time will show the possible expansions:

```
>>> machine.Pin.AF3_TIM
AF3_TIM10       AF3_TIM11       AF3_TIM8        AF3_TIM9
>>> machine.Pin.AF3_TIM
```

2.2.3 Interrupting a running program

You can interrupt a running program by pressing Ctrl-C. This will raise a KeyboardInterrupt which will bring you back to the REPL, providing your program doesnt intercept the KeyboardInterrupt exception.

For example:

```
>>> for i in range(1000000):
...     print(i)
...
0
```

(continues on next page)

(continued from previous page)

```
1
2
3
...
6466
6467
6468
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt:
>>>
```

2.2.4 Paste mode

If you want to paste some code into your terminal window, the auto-indent feature will mess things up. For example, if you had the following python code:

```
def foo():
    print('This is a test to show paste mode')
    print('Here is a second line')
foo()
```

and you try to paste this into the normal REPL, then you will see something like this:

```
>>> def foo():
...         print('This is a test to show paste mode')
...             print('Here is a second line')
...             foo()
...
  File "<stdin>", line 3
IndentationError: unexpected indent
```

If you press Ctrl-E, then you will enter paste mode, which essentially turns off the auto-indent feature, and changes the prompt from >>> to ===. For example:

```
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
=== def foo():
===     print('This is a test to show paste mode')
===     print('Here is a second line')
=== foo()
===
This is a test to show paste mode
Here is a second line
>>>
```

Paste Mode allows blank lines to be pasted. The pasted text is compiled as if it were a file. Pressing Ctrl-D exits paste mode and initiates the compilation.

2.2.5 Soft reset

A soft reset will reset the python interpreter, but tries not to reset the method by which youre connected to the MicroPython board (USB-serial, or Wifi).

You can perform a soft reset from the REPL by pressing Ctrl-D, or from your python code by executing:

```
machine.soft_reset()
```

For example, if you reset your MicroPython board, and you execute a dir() command, youd see something like this:

```
>>> dir()
['__name__', 'pyb']
```

Now create some variables and repeat the dir() command:

```
>>> i = 1
>>> j = 23
>>> x = 'abc'
>>> dir()
['j', 'x', '__name__', 'pyb', 'i']
>>>
```

Now if you enter Ctrl-D, and repeat the dir() command, youll see that your variables no longer exist:

```
MPY: sync filesystems
MPY: soft reboot
MicroPython v1.5-51-g6f70283-dirty on 2015-10-30; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>> dir()
['__name__', 'pyb']
>>>
```

2.2.6 The special variable _ (underscore)

When you use the REPL, you may perform computations and see the results. MicroPython stores the results of the previous statement in the variable _ (underscore). So you can use the underscore to save the result in a variable. For example:

```
>>> 1 + 2 + 3 + 4 + 5
15
>>> x = _
>>> x
15
>>>
```

2.2.7 Raw mode and raw-paste mode

Raw mode (also called raw REPL) is not something that a person would normally use. It is intended for programmatic use and essentially behaves like paste mode with echo turned off, and with optional flow control.

Raw mode is entered using Ctrl-A. You then send your python code, followed by a Ctrl-D. The Ctrl-D will be acknowledged by OK and then the python code will be compiled and executed. Any output (or errors) will be sent back. Entering Ctrl-B will leave raw mode and return the the regular (aka friendly) REPL.

Raw-paste mode is an additional mode within the raw REPL that includes flow control, and which compiles code as it receives it. This makes it more robust for high-speed transfer of code into the device, and it also uses less RAM when receiving because it does not need to store a verbatim copy of the code before compiling (unlike standard raw mode).

Raw-paste mode uses the following protocol:

1. Enter raw REPL as usual via ctrl-A.
2. Write 3 bytes: `b"\x05A\x01"` (ie ctrl-E then A then ctrl-A).
3. Read 2 bytes to determine if the device entered raw-paste mode:
    - If the result is `b"R\x00"` then the device understands the command but doesnt support raw paste.
    - If the result is `b"R\x01"` then the device does support raw paste and has entered this mode.
    - Otherwise the result should be `b"ra"` and the device doesnt support raw paste and the string `b"w REPL; CTRL-B to exit\r\n>"` should be read and discarded.
4. If the device is in raw-paste mode then continue, otherwise fallback to standard raw mode.
5. Read 2 bytes, this is the flow control window-size-increment (in bytes) stored as a 16-bit unsigned little endian integer. The initial value for the remaining-window-size variable should be set to this number.
6. Write out the code to the device:
    - While there are bytes to send, write up to the remaining-window-size worth of bytes, and decrease the remaining-window-size by the number of bytes written.
    - If the remaining-window-size is 0, or there is a byte waiting to read, read 1 byte. If this byte is `b"\x01"` then increase the remaining-window-size by the window-size-increment from step 5. If this byte is `b"\x04"` then the device wants to end the data reception, and `b"\x04"` should be written to the device and no more code sent after that. (Note: if there is a byte waiting to be read from the device then it does not need to be read and acted upon immediately, the device will continue to consume incoming bytes as long as reamining-window-size is greater than 0.)
7. When all code has been written to the device, write `b"\x04"` to indicate end-of-data.
8. Read from the device until `b"\x04"` is received. At this point the device has received and compiled all of the code that was sent and is executing it.
9. The device outputs any characters produced by the executing code. When (if) the code finishes `b"\x04"` will be output, followed by any exception that was uncaught, followed again by `b"\x04"`. It then goes back to the standard raw REPL and outputs `b">"`.

For example, starting at a new line at the normal (friendly) REPL, if you write:

```
b"\x01\x05A\x01print(123)\x04"
```

Then the device will respond with something like:

```
b"\r\nraw REPL; CTRL-B to exit\r\n>R\x01\x80\x00\x01\x04123\r\n\x04\x04>"
```

Broken down over time this looks like:

```
# Step 1: enter raw REPL
write: b"\x01"
read: b"\r\nraw REPL; CTRL-B to exit\r\n>"

# Step 2-5: enter raw-paste mode
write: b"\x05A\x01"
read: b"R\x01\x80\x00\x01"

# Step 6-8: write out code
write: b"print(123)\x04"
read: b"\x04"

# Step 9: code executes and result is read
read: b"123\r\n\x04\x04>"
```

In this case the flow control window-size-increment is 128 and there are two windows worth of data immediately available at the start, one from the initial window-size-increment value and one from the explicit `b"\x01"` value that is sent. So this means up to 256 bytes can be written to begin with before waiting or checking for more incoming flow-control characters.

The `tools/pyboard.py` program uses the raw REPL, including raw-paste mode, to execute Python code on a MicroPython-enabled board.

2.3 MicroPython remote control: mpremote

The `mpremote` command line tool provides an integrated set of utilities to remotely interact with and automate a MicroPython device over a serial connection.

To use mpremote install it via `pip`:

```
$ pip install mpremote
```

The simplest way to use this tool is just by invoking it without any arguments:

```
mpremote
```

This command automatically detects and connects to the first available serial device and provides an interactive REPL. Serial ports are opened in exclusive mode, so running a second (or third, etc) instance of `mpremote` will connect to subsequent serial devices, if any are available.

2.3.1 Commands

For REPL access, running `mpremote` without any arguments is usually all that is needed. `mpremote` also supports a set of commands given at the command line which will perform various actions on remote MicroPython devices.

The full list of supported commands are:

- connect to a specified device via a device-name shortcut:

  ```
  $ mpremote <device-shortcut>
  ```

- connect to specified device via name:

```
$ mpremote connect <device>
```

<device> may be one of:

- list: list available devices
- auto: connect to the first available device
- id:<serial>: connect to the device with USB serial number <serial> (the second entry in the output from the connect list command)
- port:<path>: connect to the device with the given path
- any valid device name/path, to connect to that device

- disconnect current device:

```
$ mpremote disconnect
```

- enter the REPL on the connected device:

```
$ mpremote repl [options]
```

Options are:

- --capture <file>, to capture output of the REPL session to the given file
- --inject-code <string>, to specify characters to inject at the REPL when Ctrl-J is pressed
- --inject-file <file>, to specify a file to inject at the REPL when Ctrl-K is pressed

- evaluate and print the result of a Python expression:

```
$ mpremote eval <string>
```

- execute the given Python code:

```
$ mpremote exec <string>
```

- run a script from the local filesystem:

```
$ mpremote run <file>
```

- execute filesystem commands on the device:

```
$ mpremote fs <command>
```

<command> may be:

- cat <file..> to show the contents of a file or files on the device
- ls to list the current directory
- ls <dirs...> to list the given directories
- cp [-r] <src...> <dest> to copy files; use : as a prefix to specify a file on the device
- rm <src...> to remove files on the device
- mkdir <dirs...> to create directories on the device
- rmdir <dirs...> to remove directories on the device

- mount the local directory on the remote device:

```
$ mpremote mount <local-dir>
```

Multiple commands can be specified and they will be run sequentially. Connection and disconnection will be done automatically at the start and end of the execution of the tool, if such commands are not explicitly given. Automatic connection will search for the first available serial device. If no action is specified then the REPL will be entered.

2.3.2 Shortcuts

Shortcuts can be defined using the macro system. Built-in shortcuts are:

```
- ``devs``: list available devices (shortcut for ``connect list``)
```

- `a0`, `a1`, `a2`, `a3`: connect to /dev/ttyACM?
- `u0`, `u1`, `u2`, `u3`: connect to /dev/ttyUSB?
- `c0`, `c1`, `c2`, `c3`: connect to COM?
- `cat`, `ls`, `cp`, `rm`, `mkdir`, `rmdir`, `df`: filesystem commands
- `reset`: reset the device
- `bootloader`: make the device enter its bootloader

Any user configuration, including user-defined shortcuts, can be placed in the file `.config/mpremote/config.py`. For example:

```
commands = {
    "c33": "connect id:334D335C3138",
    "bl": "bootloader",
    "double x=4": "eval x*2",  # x is an argument, with default 4
    "wl_scan": ["exec", """
import network
wl = network.WLAN()
wl.active(1)
for ap in wl.scan():
    print(ap)
""",],
    "test": ["mount", ".", "exec", "import test"],
}
```

2.3.3 Examples

```
mpremote

mpremote a1

mpremote connect /dev/ttyUSB0 repl

mpremote ls

mpremote a1 ls
```

(continues on next page)

(continued from previous page)

```
mpremote exec "import micropython; micropython.mem_info()"

mpremote eval 1/2 eval 3/4

mpremote mount .

mpremote mount . exec "import local_script"

mpremote ls

mpremote cat boot.py

mpremote cp :main.py .

mpremote cp main.py :

mpremote cp -r dir/ :
```

2.4 MicroPython .mpy files

MicroPython defines the concept of an .mpy file which is a binary container file format that holds precompiled code, and which can be imported like a normal .py module. The file `foo.mpy` can be imported via `import foo`, as long as `foo.mpy` can be found in the usual way by the import machinery. Usually, each directory listed in `sys.path` is searched in order. When searching a particular directory `foo.py` is looked for first and if that is not found then `foo.mpy` is looked for, then the search continues in the next directory if neither is found. As such, `foo.py` will take precedence over `foo.mpy`.

These .mpy files can contain bytecode which is usually generated from Python source files (.py files) via the `mpy-cross` program. For some architectures an .mpy file can also contain native machine code, which can be generated in a variety of ways, most notably from C source code.

2.4.1 Versioning and compatibility of .mpy files

A given .mpy file may or may not be compatible with a given MicroPython system. Compatibility is based on the following:

- Version of the .mpy file: the version of the file must match the version supported by the system loading it.
- Bytecode features used in the .mpy file: there are two bytecode features which must match between the file and the system: unicode support and inline caching of map lookups in the bytecode.
- Small integer bits: the .mpy file will require a minimum number of bits in a small integer and the system loading it must support at least this many bits.
- Qstr compression window size: the .mpy file will require a minimum window size for qstr decompression and the system loading it must have a window greater or equal to this size.
- Native architecture: if the .mpy file contains native machine code then it will specify the architecture of that machine code and the system loading it must support execution of that architectures code.

If a MicroPython system supports importing .mpy files then the `sys.implementation.mpy` field will exist and return an integer which encodes the version (lower 8 bits), features and native architecture.

Trying to import an .mpy file that fails one of the first four tests will raise `ValueError('incompatible .mpy file')`. Trying to import an .mpy file that fails the native architecture test (if it contains native machine code) will raise `ValueError('incompatible .mpy arch')`.

If importing an .mpy file fails then try the following:

- Determine the .mpy version and flags supported by your MicroPython system by executing:

```
import sys
sys_mpy = sys.implementation.mpy
arch = [None, 'x86', 'x64',
    'armv6', 'armv6m', 'armv7m', 'armv7em', 'armv7emsp', 'armv7emdp',
    'xtensa', 'xtensawin'][sys_mpy >> 10]
print('mpy version:', sys_mpy & 0xff)
print('mpy flags:', end='')
if arch:
    print(' -march=' + arch, end='')
if not sys_mpy & 0x200:
    print(' -mno-unicode', end='')
print()
```

- Check the validity of the .mpy file by inspecting the first two bytes of the file. The first byte should be an uppercase M and the second byte will be the version number, which should match the system version from above. If it doesnt match then rebuild the .mpy file.
- Check if the system .mpy version matches the version emitted by `mpy-cross` that was used to build the .mpy file, found by `mpy-cross --version`. If it doesnt match then recompile `mpy-cross` from the Git repository checked out at the tag (or hash) reported by `mpy-cross --version`.
- Make sure you are using the correct `mpy-cross` flags, found by the code above, or by inspecting the `MPY_CROSS_FLAGS` Makefile variable for the port that you are using.

The following table shows the correspondence between MicroPython release and .mpy version.

MicroPython release	.mpy version
v1.12 and up	5
v1.11	4
v1.9.3 - v1.10	3
v1.9 - v1.9.2	2
v1.5.1 - v1.8.7	0

For completeness, the next table shows the Git commit of the main MicroPython repository at which the .mpy version was changed.

.mpy version change	Git commit
4 to 5	5716c5cf65e9b2cb46c2906f40302401bdd27517
3 to 4	9a5f92ea72754c01cc03e5efcdfe94021120531e
2 to 3	ff93fd4f50321c6190e1659b19e64fef3045a484
1 to 2	dd11af209d226b7d18d5148b239662e30ed60bad
0 to 1	6a11048af1d01c78bdacddadd1b72dc7ba7c6478
initial version 0	d8c834c95d506db979ec871417de90b7951edc30

2.4.2 Binary encoding of .mpy files

MicroPython .mpy files are a binary container format with code objects stored internally in a nested hierarchy. To keep files small while still providing a large range of possible values it uses the concept of a variably-encoded-unsigned-integer (vuint) in many places. Similar to utf-8 encoding, this encoding stores 7 bits per byte with the 8th bit (MSB) set if one or more bytes follow. The bits of the unsigned integer are stored in the vuint in LSB form.

The top-level of an .mpy file consists of two parts:

- The header.
- The raw-code for the outer scope of the module. This outer scope is executed when the .mpy file is imported.

The header

The .mpy header is:

size	field
byte	value 0x4d (ASCII M)
byte	.mpy version number
byte	feature flags
byte	number of bits in a small int
vuint	size of qstr window

Raw code elements

A raw-code element contains code, either bytecode or native machine code. Its contents are:

size	field
vuint	type and size
	code (bytecode or machine code)
vuint	number of constant objects
vuint	number of sub-raw-code elements
	constant objects
	sub-raw-code elements

The first vuint in a raw-code element encodes the type of code stored in this element (the two least-significant bits), and the decompressed length of the code (the amount of RAM to allocate for it).

Following the vuint comes the code itself. In the case of bytecode it also contains compressed qstr values.

Following the code comes a vuint counting the number of constant objects, and another vuint counting the number of sub-raw-code elements.

The constant objects are then stored next.

Finally any sub-raw-code elements are stored, recursively.

2.5 Writing interrupt handlers

On suitable hardware MicroPython offers the ability to write interrupt handlers in Python. Interrupt handlers - also known as interrupt service routines (ISRs) - are defined as callback functions. These are executed in response to an event such as a timer trigger or a voltage change on a pin. Such events can occur at any point in the execution of the program code. This carries significant consequences, some specific to the MicroPython language. Others are common to all systems capable of responding to real time events. This document covers the language specific issues first, followed by a brief introduction to real time programming for those new to it.

This introduction uses vague terms like slow or as fast as possible. This is deliberate, as speeds are application dependent. Acceptable durations for an ISR are dependent on the rate at which interrupts occur, the nature of the main program, and the presence of other concurrent events.

2.5.1 Tips and recommended practices

This summarises the points detailed below and lists the principal recommendations for interrupt handler code.

- Keep the code as short and simple as possible.
- Avoid memory allocation: no appending to lists or insertion into dictionaries, no floating point.
- Consider using `micropython.schedule` to work around the above constraint.
- Where an ISR returns multiple bytes use a pre-allocated `bytearray`. If multiple integers are to be shared between an ISR and the main program consider an array (`array.array`).
- Where data is shared between the main program and an ISR, consider disabling interrupts prior to accessing the data in the main program and re-enabling them immediately afterwards (see Critical Sections).
- Allocate an emergency exception buffer (see below).

2.5.2 MicroPython issues

The emergency exception buffer

If an error occurs in an ISR, MicroPython is unable to produce an error report unless a special buffer is created for the purpose. Debugging is simplified if the following code is included in any program using interrupts.

```
import micropython
micropython.alloc_emergency_exception_buf(100)
```

The emergency exception buffer can only hold one exception stack trace. This means that if a second exception is thrown during the handling of an exception while the heap is locked, that second exceptions stack trace will replace the original one - even if the second exception is cleanly handled. This can lead to confusing exception messages if the buffer is later printed.

Simplicity

For a variety of reasons it is important to keep ISR code as short and simple as possible. It should do only what has to be done immediately after the event which caused it: operations which can be deferred should be delegated to the main program loop. Typically an ISR will deal with the hardware device which caused the interrupt, making it ready for the next interrupt to occur. It will communicate with the main loop by updating shared data to indicate that the interrupt has occurred, and it will return. An ISR should return control to the main loop as quickly as possible. This is not a specific MicroPython issue so is covered in more detail *below*.

Communication between an ISR and the main program

Normally an ISR needs to communicate with the main program. The simplest means of doing this is via one or more shared data objects, either declared as global or shared via a class (see below). There are various restrictions and hazards around doing this, which are covered in more detail below. Integers, `bytes` and `bytearray` objects are commonly used for this purpose along with arrays (from the array module) which can store various data types.

The use of object methods as callbacks

MicroPython supports this powerful technique which enables an ISR to share instance variables with the underlying code. It also enables a class implementing a device driver to support multiple device instances. The following example causes two LEDs to flash at different rates.

```
import pyb, micropython
micropython.alloc_emergency_exception_buf(100)
class Foo(object):
    def __init__(self, timer, led):
        self.led = led
        timer.callback(self.cb)
    def cb(self, tim):
        self.led.toggle()

red = Foo(pyb.Timer(4, freq=1), pyb.LED(1))
green = Foo(pyb.Timer(2, freq=0.8), pyb.LED(2))
```

In this example the `red` instance associates timer 4 with LED 1: when a timer 4 interrupt occurs `red.cb()` is called causing LED 1 to change state. The `green` instance operates similarly: a timer 2 interrupt results in the execution of `green.cb()` and toggles LED 2. The use of instance methods confers two benefits. Firstly a single class enables code to be shared between multiple hardware instances. Secondly, as a bound method the callback functions first argument is `self`. This enables the callback to access instance data and to save state between successive calls. For example, if the class above had a variable `self.count` set to zero in the constructor, `cb()` could increment the counter. The `red` and `green` instances would then maintain independent counts of the number of times each LED had changed state.

Creation of Python objects

ISRs cannot create instances of Python objects. This is because MicroPython needs to allocate memory for the object from a store of free memory block called the heap. This is not permitted in an interrupt handler because heap allocation is not re-entrant. In other words the interrupt might occur when the main program is part way through performing an allocation - to maintain the integrity of the heap the interpreter disallows memory allocations in ISR code.

A consequence of this is that ISRs cant use floating point arithmetic; this is because floats are Python objects. Similarly an ISR cant append an item to a list. In practice it can be hard to determine exactly which code constructs will attempt to perform memory allocation and provoke an error message: another reason for keeping ISR code short and simple.

One way to avoid this issue is for the ISR to use pre-allocated buffers. For example a class constructor creates a `bytearray` instance and a boolean flag. The ISR method assigns data to locations in the buffer and sets the flag. The memory allocation occurs in the main program code when the object is instantiated rather than in the ISR.

The MicroPython library I/O methods usually provide an option to use a pre-allocated buffer. For example `pyb.i2c.recv()` can accept a mutable buffer as its first argument: this enables its use in an ISR.

A means of creating an object without employing a class or globals is as follows:

```
def set_volume(t, buf=bytearray(3)):
    buf[0] = 0xa5
    buf[1] = t >> 4
    buf[2] = 0x5a
    return buf
```

The compiler instantiates the default `buf` argument when the function is loaded for the first time (usually when the module its in is imported).

An instance of object creation occurs when a reference to a bound method is created. This means that an ISR cannot pass a bound method to a function. One solution is to create a reference to the bound method in the class constructor and to pass that reference in the ISR. For example:

```
class Foo():
    def __init__(self):
        self.bar_ref = self.bar  # Allocation occurs here
        self.x = 0.1
        tim = pyb.Timer(4)
        tim.init(freq=2)
        tim.callback(self.cb)

    def bar(self, _):
        self.x *= 1.2
        print(self.x)

    def cb(self, t):
        # Passing self.bar would cause allocation.
        micropython.schedule(self.bar_ref, 0)
```

Other techniques are to define and instantiate the method in the constructor or to pass `Foo.bar()` with the argument *self*.

Use of Python objects

A further restriction on objects arises because of the way Python works. When an `import` statement is executed the Python code is compiled to bytecode, with one line of code typically mapping to multiple bytecodes. When the code runs the interpreter reads each bytecode and executes it as a series of machine code instructions. Given that an interrupt can occur at any time between machine code instructions, the original line of Python code may be only partially executed. Consequently a Python object such as a set, list or dictionary modified in the main loop may lack internal consistency at the moment the interrupt occurs.

A typical outcome is as follows. On rare occasions the ISR will run at the precise moment in time when the object is partially updated. When the ISR tries to read the object, a crash results. Because such problems typically occur on rare, random occasions they can be hard to diagnose. There are ways to circumvent this issue, described in *Critical Sections* below.

It is important to be clear about what constitutes the modification of an object. An alteration to a built-in type such as a dictionary is problematic. Altering the contents of an array or bytearray is not. This is because bytes or words are written as a single machine code instruction which is not interruptible: in the parlance of real time programming the write is atomic. A user defined object might instantiate an integer, array or bytearray. It is valid for both the main loop and the ISR to alter the contents of these.

MicroPython supports integers of arbitrary precision. Values between 2**30 -1 and -2**30 will be stored in a single machine word. Larger values are stored as Python objects. Consequently changes to long integers cannot be considered atomic. The use of long integers in ISRs is unsafe because memory allocation may be attempted as the variables value changes.

Overcoming the float limitation

In general it is best to avoid using floats in ISR code: hardware devices normally handle integers and conversion to floats is normally done in the main loop. However there are a few DSP algorithms which require floating point. On platforms with hardware floating point (such as the Pyboard) the inline ARM Thumb assembler can be used to work round this limitation. This is because the processor stores float values in a machine word; values can be shared between the ISR and main program code via an array of floats.

Using micropython.schedule

This function enables an ISR to schedule a callback for execution very soon. The callback is queued for execution which will take place at a time when the heap is not locked. Hence it can create Python objects and use floats. The callback is also guaranteed to run at a time when the main program has completed any update of Python objects, so the callback will not encounter partially updated objects.

Typical usage is to handle sensor hardware. The ISR acquires data from the hardware and enables it to issue a further interrupt. It then schedules a callback to process the data.

Scheduled callbacks should comply with the principles of interrupt handler design outlined below. This is to avoid problems resulting from I/O activity and the modification of shared data which can arise in any code which pre-empts the main program loop.

Execution time needs to be considered in relation to the frequency with which interrupts can occur. If an interrupt occurs while the previous callback is executing, a further instance of the callback will be queued for execution; this will run after the current instance has completed. A sustained high interrupt repetition rate therefore carries a risk of unconstrained queue growth and eventual failure with a `RuntimeError`.

If the callback to be passed to *`schedule()`* is a bound method, consider the note in Creation of Python objects.

2.5.3 Exceptions

If an ISR raises an exception it will not propagate to the main loop. The interrupt will be disabled unless the exception is handled by the ISR code.

2.5.4 General issues

This is merely a brief introduction to the subject of real time programming. Beginners should note that design errors in real time programs can lead to faults which are particularly hard to diagnose. This is because they can occur rarely and at intervals which are essentially random. It is crucial to get the initial design right and to anticipate issues before they arise. Both interrupt handlers and the main program need to be designed with an appreciation of the following issues.

Interrupt handler design

As mentioned above, ISRs should be designed to be as simple as possible. They should always return in a short, predictable period of time. This is important because when the ISR is running, the main loop is not: inevitably the main loop experiences pauses in its execution at random points in the code. Such pauses can be a source of hard to diagnose bugs particularly if their duration is long or variable. In order to understand the implications of ISR run time, a basic grasp of interrupt priorities is required.

Interrupts are organised according to a priority scheme. ISR code may itself be interrupted by a higher priority interrupt. This has implications if the two interrupts share data (see Critical Sections below). If such an interrupt occurs it interposes a delay into the ISR code. If a lower priority interrupt occurs while the ISR is running, it will be delayed until the ISR is complete: if the delay is too long, the lower priority interrupt may fail. A further issue with slow ISRs is the case where a second interrupt of the same type occurs during its execution. The second interrupt will be handled on termination of the first. However if the rate of incoming interrupts consistently exceeds the capacity of the ISR to service them the outcome will not be a happy one.

Consequently looping constructs should be avoided or minimised. I/O to devices other than to the interrupting device should normally be avoided: I/O such as disk access, `print` statements and UART access is relatively slow, and its duration may vary. A further issue here is that filesystem functions are not reentrant: using filesystem I/O in an ISR and the main program would be hazardous. Crucially ISR code should not wait on an event. I/O is acceptable if the code can be guaranteed to return in a predictable period, for example toggling a pin or LED. Accessing the interrupting device via I2C or SPI may be necessary but the time taken for such accesses should be calculated or measured and its impact on the application assessed.

There is usually a need to share data between the ISR and the main loop. This may be done either through global variables or via class or instance variables. Variables are typically integer or boolean types, or integer or byte arrays (a pre-allocated integer array offers faster access than a list). Where multiple values are modified by the ISR it is necessary to consider the case where the interrupt occurs at a time when the main program has accessed some, but not all, of the values. This can lead to inconsistencies.

Consider the following design. An ISR stores incoming data in a bytearray, then adds the number of bytes received to an integer representing total bytes ready for processing. The main program reads the number of bytes, processes the bytes, then clears down the number of bytes ready. This will work until an interrupt occurs just after the main program has read the number of bytes. The ISR puts the added data into the buffer and updates the number received, but the main program has already read the number, so processes the data originally received. The newly arrived bytes are lost.

There are various ways of avoiding this hazard, the simplest being to use a circular buffer. If it is not possible to use a structure with inherent thread safety other ways are described below.

Reentrancy

A potential hazard may occur if a function or method is shared between the main program and one or more ISRs or between multiple ISRs. The issue here is that the function may itself be interrupted and a further instance of that function run. If this is to occur, the function must be designed to be reentrant. How this is done is an advanced topic beyond the scope of this tutorial.

Critical sections

An example of a critical section of code is one which accesses more than one variable which can be affected by an ISR. If the interrupt happens to occur between accesses to the individual variables, their values will be inconsistent. This is an instance of a hazard known as a race condition: the ISR and the main program loop race to alter the variables. To avoid inconsistency a means must be employed to ensure that the ISR does not alter the values for the duration of the critical section. One way to achieve this is to issue `pyb.disable_irq()` before the start of the section, and `pyb.enable_irq()` at the end. Here is an example of this approach:

```
import pyb, micropython, array
micropython.alloc_emergency_exception_buf(100)

class BoundsException(Exception):
    pass

ARRAYSIZE = const(20)
index = 0
data = array.array('i', 0 for x in range(ARRAYSIZE))

def callback1(t):
    global data, index
    for x in range(5):
        data[index] = pyb.rng() # simulate input
        index += 1
        if index >= ARRAYSIZE:
            raise BoundsException('Array bounds exceeded')

tim4 = pyb.Timer(4, freq=100, callback=callback1)

for loop in range(1000):
    if index > 0:
        irq_state = pyb.disable_irq() # Start of critical section
        for x in range(index):
            print(data[x])
        index = 0
        pyb.enable_irq(irq_state) # End of critical section
        print('loop {}'.format(loop))
    pyb.delay(1)

tim4.callback(None)
```

A critical section can comprise a single line of code and a single variable. Consider the following code fragment.

```
count = 0
def cb(): # An interrupt callback
    count +=1
```

(continues on next page)

(continued from previous page)

```
def main():
    # Code to set up the interrupt callback omitted
    while True:
        count += 1
```

This example illustrates a subtle source of bugs. The line `count += 1` in the main loop carries a specific race condition hazard known as a read-modify-write. This is a classic cause of bugs in real time systems. In the main loop MicroPython reads the value of `t.counter`, adds 1 to it, and writes it back. On rare occasions the interrupt occurs after the read and before the write. The interrupt modifies `t.counter` but its change is overwritten by the main loop when the ISR returns. In a real system this could lead to rare, unpredictable failures.

As mentioned above, care should be taken if an instance of a Python built in type is modified in the main code and that instance is accessed in an ISR. The code performing the modification should be regarded as a critical section to ensure that the instance is in a valid state when the ISR runs.

Particular care needs to be taken if a dataset is shared between different ISRs. The hazard here is that the higher priority interrupt may occur when the lower priority one has partially updated the shared data. Dealing with this situation is an advanced topic beyond the scope of this introduction other than to note that mutex objects described below can sometimes be used.

Disabling interrupts for the duration of a critical section is the usual and simplest way to proceed, but it disables all interrupts rather than merely the one with the potential to cause problems. It is generally undesirable to disable an interrupt for long. In the case of timer interrupts it introduces variability to the time when a callback occurs. In the case of device interrupts, it can lead to the device being serviced too late with possible loss of data or overrun errors in the device hardware. Like ISRs, a critical section in the main code should have a short, predictable duration.

An approach to dealing with critical sections which radically reduces the time for which interrupts are disabled is to use an object termed a mutex (name derived from the notion of mutual exclusion). The main program locks the mutex before running the critical section and unlocks it at the end. The ISR tests whether the mutex is locked. If it is, it avoids the critical section and returns. The design challenge is defining what the ISR should do in the event that access to the critical variables is denied. A simple example of a mutex may be found here. Note that the mutex code does disable interrupts, but only for the duration of eight machine instructions: the benefit of this approach is that other interrupts are virtually unaffected.

Interrupts and the REPL

Interrupt handlers, such as those associated with timers, can continue to run after a program terminates. This may produce unexpected results where you might have expected the object raising the callback to have gone out of scope. For example on the Pyboard:

```
def bar():
    foo = pyb.Timer(2, freq=4, callback=lambda t: print('.', end=''))

bar()
```

This continues to run until the timer is explicitly disabled or the board is reset with `ctrl D`.

2.6 Maximising MicroPython speed

Contents

This tutorial describes ways of improving the performance of MicroPython code. Optimisations involving other languages are covered elsewhere, namely the use of modules written in C and the MicroPython inline assembler.

The process of developing high performance code comprises the following stages which should be performed in the order listed.

- Design for speed.
- Code and debug.

Optimisation steps:

- Identify the slowest section of code.
- Improve the efficiency of the Python code.
- Use the native code emitter.
- Use the viper code emitter.
- Use hardware-specific optimisations.

2.6.1 Designing for speed

Performance issues should be considered at the outset. This involves taking a view on the sections of code which are most performance critical and devoting particular attention to their design. The process of optimisation begins when the code has been tested: if the design is correct at the outset optimisation will be straightforward and may actually be unnecessary.

Algorithms

The most important aspect of designing any routine for performance is ensuring that the best algorithm is employed. This is a topic for textbooks rather than for a MicroPython guide but spectacular performance gains can sometimes be achieved by adopting algorithms known for their efficiency.

RAM allocation

To design efficient MicroPython code it is necessary to have an understanding of the way the interpreter allocates RAM. When an object is created or grows in size (for example where an item is appended to a list) the necessary RAM is allocated from a block known as the heap. This takes a significant amount of time; further it will on occasion trigger a process known as garbage collection which can take several milliseconds.

Consequently the performance of a function or method can be improved if an object is created once only and not permitted to grow in size. This implies that the object persists for the duration of its use: typically it will be instantiated in a class constructor and used in various methods.

This is covered in further detail *Controlling garbage collection* below.

Buffers

An example of the above is the common case where a buffer is required, such as one used for communication with a device. A typical driver will create the buffer in the constructor and use it in its I/O methods which will be called repeatedly.

The MicroPython libraries typically provide support for pre-allocated buffers. For example, objects which support stream interface (e.g., file or UART) provide `read()` method which allocates new buffer for read data, but also a `readinto()` method to read data into an existing buffer.

Floating point

Some MicroPython ports allocate floating point numbers on heap. Some other ports may lack dedicated floating-point coprocessor, and perform arithmetic operations on them in software at considerably lower speed than on integers. Where performance is important, use integer operations and restrict the use of floating point to sections of the code where performance is not paramount. For example, capture ADC readings as integers values to an array in one quick go, and only then convert them to floating-point numbers for signal processing.

Arrays

Consider the use of the various types of array classes as an alternative to lists. The *array* module supports various element types with 8-bit elements supported by Pythons built in *bytes* and *bytearray* classes. These data structures all store elements in contiguous memory locations. Once again to avoid memory allocation in critical code these should be pre-allocated and passed as arguments or as bound objects.

When passing slices of objects such as *bytearray* instances, Python creates a copy which involves allocation of the size proportional to the size of slice. This can be alleviated using a *memoryview* object. The *memoryview* itself is allocated on the heap, but is a small, fixed-size object, regardless of the size of slice it points too. Slicing a *memoryview* creates a new *memoryview*, so this cannot be done in an interrupt service routine. Further, the slice syntax `a:b` causes further allocation by instantiating a `slice(a, b)` object.

```
ba = bytearray(10000)  # big array
func(ba[30:2000])      # a copy is passed, ~2K new allocation
mv = memoryview(ba)    # small object is allocated
func(mv[30:2000])      # a pointer to memory is passed
```

A *memoryview* can only be applied to objects supporting the buffer protocol - this includes arrays but not lists. Small caveat is that while memoryview object is live, it also keeps alive the original buffer object. So, a memoryview isnt a universal panacea. For instance, in the example above, if you are done with 10K buffer and just need those bytes 30:2000 from it, it may be better to make a slice, and let the 10K buffer go (be ready for garbage collection), instead of making a long-living memoryview and keeping 10K blocked for GC.

Nonetheless, *memoryview* is indispensable for advanced preallocated buffer management. `readinto()` method discussed above puts data at the beginning of buffer and fills in entire buffer. What if you need to put data in the middle of existing buffer? Just create a memoryview into the needed section of buffer and pass it to `readinto()`.

2.6.2 Identifying the slowest section of code

This is a process known as profiling and is covered in textbooks and (for standard Python) supported by various software tools. For the type of smaller embedded application likely to be running on MicroPython platforms the slowest function or method can usually be established by judicious use of the timing `ticks` group of functions documented in *time*. Code execution time can be measured in ms, us, or CPU cycles.

The following enables any function or method to be timed by adding an `@timed_function` decorator:

```
def timed_function(f, *args, **kwargs):
    myname = str(f).split(' ')[1]
    def new_func(*args, **kwargs):
        t = time.ticks_us()
        result = f(*args, **kwargs)
        delta = time.ticks_diff(time.ticks_us(), t)
        print('Function {} Time = {:6.3f}ms'.format(myname, delta/1000))
        return result
    return new_func
```

2.6.3 MicroPython code improvements

The const() declaration

MicroPython provides a `const()` declaration. This works in a similar way to `#define` in C in that when the code is compiled to bytecode the compiler substitutes the numeric value for the identifier. This avoids a dictionary lookup at runtime. The argument to `const()` may be anything which, at compile time, evaluates to an integer e.g. `0x100` or `1 << 8`.

Caching object references

Where a function or method repeatedly accesses objects performance is improved by caching the object in a local variable:

```
class foo(object):
    def __init__(self):
        self.ba = bytearray(100)
    def bar(self, obj_display):
        ba_ref = self.ba
        fb = obj_display.framebuffer
        # iterative code using these two objects
```

This avoids the need repeatedly to look up `self.ba` and `obj_display.framebuffer` in the body of the method `bar()`.

Controlling garbage collection

When memory allocation is required, MicroPython attempts to locate an adequately sized block on the heap. This may fail, usually because the heap is cluttered with objects which are no longer referenced by code. If a failure occurs, the process known as garbage collection reclaims the memory used by these redundant objects and the allocation is then tried again - a process which can take several milliseconds.

There may be benefits in pre-empting this by periodically issuing *gc.collect()*. Firstly doing a collection before it is actually required is quicker - typically on the order of 1ms if done frequently. Secondly you can determine the point in code where this time is used rather than have a longer delay occur at random points, possibly in a speed critical section. Finally performing collections regularly can reduce fragmentation in the heap. Severe fragmentation can lead to non-recoverable allocation failures.

2.6.4 The Native code emitter

This causes the MicroPython compiler to emit native CPU opcodes rather than bytecode. It covers the bulk of the MicroPython functionality, so most functions will require no adaptation (but see below). It is invoked by means of a function decorator:

```
@micropython.native
def foo(self, arg):
    buf = self.linebuf # Cached object
    # code
```

There are certain limitations in the current implementation of the native code emitter.

- Context managers are not supported (the `with` statement).

- Generators are not supported.
- If `raise` is used an argument must be supplied.

The trade-off for the improved performance (roughly twice as fast as bytecode) is an increase in compiled code size.

2.6.5 The Viper code emitter

The optimisations discussed above involve standards-compliant Python code. The Viper code emitter is not fully compliant. It supports special Viper native data types in pursuit of performance. Integer processing is non-compliant because it uses machine words: arithmetic on 32 bit hardware is performed modulo 2**32.

Like the Native emitter Viper produces machine instructions but further optimisations are performed, substantially increasing performance especially for integer arithmetic and bit manipulations. It is invoked using a decorator:

```
@micropython.viper
def foo(self, arg: int) -> int:
    # code
```

As the above fragment illustrates it is beneficial to use Python type hints to assist the Viper optimiser. Type hints provide information on the data types of arguments and of the return value; these are a standard Python language feature formally defined here PEP0484. Viper supports its own set of types namely `int`, `uint` (unsigned integer), `ptr`, `ptr8`, `ptr16` and `ptr32`. The `ptrX` types are discussed below. Currently the `uint` type serves a single purpose: as a type hint for a function return value. If such a function returns `0xffffffff` Python will interpret the result as 2**32 -1 rather than as -1.

In addition to the restrictions imposed by the native emitter the following constraints apply:

- Functions may have up to four arguments.
- Default argument values are not permitted.
- Floating point may be used but is not optimised.

Viper provides pointer types to assist the optimiser. These comprise

- `ptr` Pointer to an object.
- `ptr8` Points to a byte.
- `ptr16` Points to a 16 bit half-word.
- `ptr32` Points to a 32 bit machine word.

The concept of a pointer may be unfamiliar to Python programmers. It has similarities to a Python *memoryview* object in that it provides direct access to data stored in memory. Items are accessed using subscript notation, but slices are not supported: a pointer can return a single item only. Its purpose is to provide fast random access to data stored in contiguous memory locations - such as data stored in objects which support the buffer protocol, and memory-mapped peripheral registers in a microcontroller. It should be noted that programming using pointers is hazardous: bounds checking is not performed and the compiler does nothing to prevent buffer overrun errors.

Typical usage is to cache variables:

```
@micropython.viper
def foo(self, arg: int) -> int:
    buf = ptr8(self.linebuf) # self.linebuf is a bytearray or bytes object
    for x in range(20, 30):
        bar = buf[x] # Access a data item through the pointer
        # code omitted
```

In this instance the compiler knows that `buf` is the address of an array of bytes; it can emit code to rapidly compute the address of `buf[x]` at runtime. Where casts are used to convert objects to Viper native types these should be performed at the start of the function rather than in critical timing loops as the cast operation can take several microseconds. The rules for casting are as follows:

- Casting operators are currently: `int`, `bool`, `uint`, `ptr`, `ptr8`, `ptr16` and `ptr32`.
- The result of a cast will be a native Viper variable.
- Arguments to a cast can be a Python object or a native Viper variable.
- If argument is a native Viper variable, then cast is a no-op (i.e. costs nothing at runtime) that just changes the type (e.g. from `uint` to `ptr8`) so that you can then store/load using this pointer.
- If the argument is a Python object and the cast is `int` or `uint`, then the Python object must be of integral type and the value of that integral object is returned.
- The argument to a bool cast must be integral type (boolean or integer); when used as a return type the viper function will return True or False objects.
- If the argument is a Python object and the cast is `ptr`, `ptr`, `ptr16` or `ptr32`, then the Python object must either have the buffer protocol (in which case a pointer to the start of the buffer is returned) or it must be of integral type (in which case the value of that integral object is returned).

Writing to a pointer which points to a read-only object will lead to undefined behaviour.

The following example illustrates the use of a `ptr16` cast to toggle pin X1 `n` times:

```
BIT0 = const(1)
@micropython.viper
def toggle_n(n: int):
    odr = ptr16(stm.GPIOA + stm.GPIO_ODR)
    for _ in range(n):
        odr[0] ^= BIT0
```

A detailed technical description of the three code emitters may be found on Kickstarter here Note 1 and here Note 2

2.6.6 Accessing hardware directly

Note: Code examples in this section are given for the Pyboard. The techniques described however may be applied to other MicroPython ports too.

This comes into the category of more advanced programming and involves some knowledge of the target MCU. Consider the example of toggling an output pin on the Pyboard. The standard approach would be to write

```
mypin.value(mypin.value() ^ 1) # mypin was instantiated as an output pin
```

This involves the overhead of two calls to the *Pin* instances *value()* method. This overhead can be eliminated by performing a read/write to the relevant bit of the chips GPIO port output data register (odr). To facilitate this the `stm` module provides a set of constants providing the addresses of the relevant registers. A fast toggle of pin `P4` (CPU pin `A14`) - corresponding to the green LED - can be performed as follows:

```
import machine
import stm

```

(continues on next page)

(continued from previous page)

```
BIT14 = const(1 << 14)
machine.mem16[stm.GPIOA + stm.GPIO_ODR] ^= BIT14
```

2.7 MicroPython on microcontrollers

MicroPython is designed to be capable of running on microcontrollers. These have hardware limitations which may be unfamiliar to programmers more familiar with conventional computers. In particular the amount of RAM and nonvolatile disk (flash memory) storage is limited. This tutorial offers ways to make the most of the limited resources. Because MicroPython runs on controllers based on a variety of architectures, the methods presented are generic: in some cases it will be necessary to obtain detailed information from platform specific documentation.

2.7.1 Flash memory

On the Pyboard the simple way to address the limited capacity is to fit a micro SD card. In some cases this is impractical, either because the device does not have an SD card slot or for reasons of cost or power consumption; hence the on-chip flash must be used. The firmware including the MicroPython subsystem is stored in the onboard flash. The remaining capacity is available for use. For reasons connected with the physical architecture of the flash memory part of this capacity may be inaccessible as a filesystem. In such cases this space may be employed by incorporating user modules into a firmware build which is then flashed to the device.

There are two ways to achieve this: frozen modules and frozen bytecode. Frozen modules store the Python source with the firmware. Frozen bytecode uses the cross compiler to convert the source to bytecode which is then stored with the firmware. In either case the module may be accessed with an import statement:

```
import mymodule
```

The procedure for producing frozen modules and bytecode is platform dependent; instructions for building the firmware can be found in the README files in the relevant part of the source tree.

In general terms the steps are as follows:

- Clone the MicroPython repository.
- Acquire the (platform specific) toolchain to build the firmware.
- Build the cross compiler.
- Place the modules to be frozen in a specified directory (dependent on whether the module is to be frozen as source or as bytecode).
- Build the firmware. A specific command may be required to build frozen code of either type - see the platform documentation.
- Flash the firmware to the device.

2.7.2 RAM

When reducing RAM usage there are two phases to consider: compilation and execution. In addition to memory consumption, there is also an issue known as heap fragmentation. In general terms it is best to minimise the repeated creation and destruction of objects. The reason for this is covered in the section covering the *heap*.

Compilation phase

When a module is imported, MicroPython compiles the code to bytecode which is then executed by the MicroPython virtual machine (VM). The bytecode is stored in RAM. The compiler itself requires RAM, but this becomes available for use when the compilation has completed.

If a number of modules have already been imported the situation can arise where there is insufficient RAM to run the compiler. In this case the import statement will produce a memory exception.

If a module instantiates global objects on import it will consume RAM at the time of import, which is then unavailable for the compiler to use on subsequent imports. In general it is best to avoid code which runs on import; a better approach is to have initialisation code which is run by the application after all modules have been imported. This maximises the RAM available to the compiler.

If RAM is still insufficient to compile all modules one solution is to precompile modules. MicroPython has a cross compiler capable of compiling Python modules to bytecode (see the README in the mpy-cross directory). The resulting bytecode file has a .mpy extension; it may be copied to the filesystem and imported in the usual way. Alternatively some or all modules may be implemented as frozen bytecode: on most platforms this saves even more RAM as the bytecode is run directly from flash rather than being stored in RAM.

Execution phase

There are a number of coding techniques for reducing RAM usage.

Constants

MicroPython provides a `const` keyword which may be used as follows:

```
from micropython import const
ROWS = const(33)
_COLS = const(0x10)
a = ROWS
b = _COLS
```

In both instances where the constant is assigned to a variable the compiler will avoid coding a lookup to the name of the constant by substituting its literal value. This saves bytecode and hence RAM. However the `ROWS` value will occupy at least two machine words, one each for the key and value in the globals dictionary. The presence in the dictionary is necessary because another module might import or use it. This RAM can be saved by prepending the name with an underscore as in `_COLS`: this symbol is not visible outside the module so will not occupy RAM.

The argument to `const()` may be anything which, at compile time, evaluates to an integer e.g. `0x100` or `1 << 8`. It can even include other const symbols that have already been defined, e.g. `1 << BIT`.

Constant data structures

Where there is a substantial volume of constant data and the platform supports execution from Flash, RAM may be saved as follows. The data should be located in Python modules and frozen as bytecode. The data must be defined as *`bytes`* objects. The compiler knows that *`bytes`* objects are immutable and ensures that the objects remain in flash memory rather than being copied to RAM. The *`struct`* module can assist in converting between *`bytes`* types and other Python built-in types.

When considering the implications of frozen bytecode, note that in Python strings, floats, bytes, integers and complex numbers are immutable. Accordingly these will be frozen into flash. Thus, in the line

```
mystring = "The quick brown fox"
```

the actual string The quick brown fox will reside in flash. At runtime a reference to the string is assigned to the *variable* `mystring`. The reference occupies a single machine word. In principle a long integer could be used to store constant data:

```
bar = 0xDEADBEEF0000DEADBEEF
```

As in the string example, at runtime a reference to the arbitrarily large integer is assigned to the variable `bar`. That reference occupies a single machine word.

It might be expected that tuples of integers could be employed for the purpose of storing constant data with minimal RAM use. With the current compiler this is ineffective (the code works, but RAM is not saved).

```
foo = (1, 2, 3, 4, 5, 6, 100000)
```

At runtime the tuple will be located in RAM. This may be subject to future improvement.

Needless object creation

There are a number of situations where objects may unwittingly be created and destroyed. This can reduce the usability of RAM through fragmentation. The following sections discuss instances of this.

String concatenation

Consider the following code fragments which aim to produce constant strings:

```
var = "foo" + "bar"
var1 = "foo" "bar"
var2 = """\
foo\
bar"""
```

Each produces the same outcome, however the first needlessly creates two string objects at runtime, allocates more RAM for concatenation before producing the third. The others perform the concatenation at compile time which is more efficient, reducing fragmentation.

Where strings must be dynamically created before being fed to a stream such as a file it will save RAM if this is done in a piecemeal fashion. Rather than creating a large string object, create a substring and feed it to the stream before dealing with the next.

The best way to create dynamic strings is by means of the string `format()` method:

```
var = "Temperature {:5.2f} Pressure {:06d}\n".format(temp, press)
```

Buffers

When accessing devices such as instances of UART, I2C and SPI interfaces, using pre-allocated buffers avoids the creation of needless objects. Consider these two loops:

```
while True:
    var = spi.read(100)
    # process data

buf = bytearray(100)
```

(continues on next page)

(continued from previous page)

```
while True:
    spi.readinto(buf)
    # process data in buf
```

The first creates a buffer on each pass whereas the second re-uses a pre-allocated buffer; this is both faster and more efficient in terms of memory fragmentation.

Bytes are smaller than ints

On most platforms an integer consumes four bytes. Consider the two calls to the function `foo()`:

```
def foo(bar):
    for x in bar:
        print(x)
foo((1, 2, 0xff))
foo(b'\1\2\xff')
```

In the first call a tuple of integers is created in RAM. The second efficiently creates a *bytes* object consuming the minimum amount of RAM. If the module were frozen as bytecode, the *bytes* object would reside in flash.

Strings Versus Bytes

Python3 introduced Unicode support. This introduced a distinction between a string and an array of bytes. MicroPython ensures that Unicode strings take no additional space so long as all characters in the string are ASCII (i.e. have a value < 126). If values in the full 8-bit range are required *bytes* and *bytearray* objects can be used to ensure that no additional space will be required. Note that most string methods (e.g. `str.strip()`) apply also to *bytes* instances so the process of eliminating Unicode can be painless.

```
s = 'the quick brown fox'   # A string instance
b = b'the quick brown fox'  # A bytes instance
```

Where it is necessary to convert between strings and bytes the `str.encode()` and the `bytes.decode()` methods can be used. Note that both strings and bytes are immutable. Any operation which takes as input such an object and produces another implies at least one RAM allocation to produce the result. In the second line below a new bytes object is allocated. This would also occur if `foo` were a string.

```
foo = b'   empty whitespace'
foo = foo.lstrip()
```

Runtime compiler execution

The Python funcitons *eval* and *exec* invoke the compiler at runtime, which requires significant amounts of RAM. Note that the `pickle` library from *micropython-lib* employs *exec*. It may be more RAM efficient to use the *json* library for object serialisation.

Storing strings in flash

Python strings are immutable hence have the potential to be stored in read only memory. The compiler can place in flash strings defined in Python code. As with frozen modules it is necessary to have a copy of the source tree on the PC and the toolchain to build the firmware. The procedure will work even if the modules have not been fully debugged, so long as they can be imported and run.

After importing the modules, execute:

```
micropython.qstr_info(1)
```

Then copy and paste all the Q(xxx) lines into a text editor. Check for and remove lines which are obviously invalid. Open the file qstrdefsport.h which will be found in ports/stm32 (or the equivalent directory for the architecture in use).

Copy and paste the corrected lines at the end of the file. Save the file, rebuild and flash the firmware. The outcome can be checked by importing the modules and again issuing:

```
micropython.qstr_info(1)
```

The Q(xxx) lines should be gone.

2.7.3 The heap

When a running program instantiates an object the necessary RAM is allocated from a fixed size pool known as the heap. When the object goes out of scope (in other words becomes inaccessible to code) the redundant object is known as garbage. A process known as garbage collection (GC) reclaims that memory, returning it to the free heap. This process runs automatically, however it can be invoked directly by issuing *gc.collect()*.

The discourse on this is somewhat involved. For a quick fix issue the following periodically:

```
gc.collect()
gc.threshold(gc.mem_free() // 4 + gc.mem_alloc())
```

Fragmentation

Say a program creates an object `foo`, then an object `bar`. Subsequently `foo` goes out of scope but `bar` remains. The RAM used by `foo` will be reclaimed by GC. However if `bar` was allocated to a higher address, the RAM reclaimed from `foo` will only be of use for objects no bigger than `foo`. In a complex or long running program the heap can become fragmented: despite there being a substantial amount of RAM available, there is insufficient contiguous space to allocate a particular object, and the program fails with a memory error.

The techniques outlined above aim to minimise this. Where large permanent buffers or other objects are required it is best to instantiate these early in the process of program execution before fragmentation can occur. Further improvements may be made by monitoring the state of the heap and by controlling GC; these are outlined below.

Reporting

A number of library functions are available to report on memory allocation and to control GC. These are to be found in the *gc* and *micropython* modules. The following example may be pasted at the REPL (`ctrl e` to enter paste mode, `ctrl d` to run it).

```
import gc
import micropython
gc.collect()
micropython.mem_info()
print('------------------------------')
print('Initial free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
def func():
    a = bytearray(10000)
gc.collect()
print('Func definition: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
func()
print('Func run free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
gc.collect()
print('Garbage collect free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
print('------------------------------')
micropython.mem_info(1)
```

Methods employed above:

- *gc.collect()* Force a garbage collection. See footnote.
- *micropython.mem_info()* Print a summary of RAM utilisation.
- *gc.mem_free()* Return the free heap size in bytes.
- *gc.mem_alloc()* Return the number of bytes currently allocated.
- `micropython.mem_info(1)` Print a table of heap utilisation (detailed below).

The numbers produced are dependent on the platform, but it can be seen that declaring the function uses a small amount of RAM in the form of bytecode emitted by the compiler (the RAM used by the compiler has been reclaimed). Running the function uses over 10KiB, but on return `a` is garbage because it is out of scope and cannot be referenced. The final *gc.collect()* recovers that memory.

The final output produced by `micropython.mem_info(1)` will vary in detail but may be interpreted as follows:

Symbol	Meaning
.	free block
h	head block
=	tail block
m	marked head block
T	tuple
L	list
D	dict
F	float
B	byte code
M	module

Each letter represents a single block of memory, a block being 16 bytes. So each line of the heap dump represents 0x400 bytes or 1KiB of RAM.

Control of garbage collection

A GC can be demanded at any time by issuing *gc.collect()*. It is advantageous to do this at intervals, firstly to pre-empt fragmentation and secondly for performance. A GC can take several milliseconds but is quicker when there is little work to do (about 1ms on the Pyboard). An explicit call can minimise that delay while ensuring it occurs at points in the program when it is acceptable.

Automatic GC is provoked under the following circumstances. When an attempt at allocation fails, a GC is performed and the allocation re-tried. Only if this fails is an exception raised. Secondly an automatic GC will be triggered if the amount of free RAM falls below a threshold. This threshold can be adapted as execution progresses:

```
gc.collect()
gc.threshold(gc.mem_free() // 4 + gc.mem_alloc())
```

This will provoke a GC when more than 25% of the currently free heap becomes occupied.

In general modules should instantiate data objects at runtime using constructors or other initialisation functions. The reason is that if this occurs on initialisation the compiler may be starved of RAM when subsequent modules are imported. If modules do instantiate data on import then *gc.collect()* issued after the import will ameliorate the problem.

2.7.4 String operations

MicroPython handles strings in an efficient manner and understanding this can help in designing applications to run on microcontrollers. When a module is compiled, strings which occur multiple times are stored once only, a process known as string interning. In MicroPython an interned string is known as a `qstr`. In a module imported normally that single instance will be located in RAM, but as described above, in modules frozen as bytecode it will be located in flash.

String comparisons are also performed efficiently using hashing rather than character by character. The penalty for using strings rather than integers may hence be small both in terms of performance and RAM usage - a fact which may come as a surprise to C programmers.

2.7.5 Postscript

MicroPython passes, returns and (by default) copies objects by reference. A reference occupies a single machine word so these processes are efficient in RAM usage and speed.

Where variables are required whose size is neither a byte nor a machine word there are standard libraries which can assist in storing these efficiently and in performing conversions. See the *array*, *struct* and *uctypes* modules.

Footnote: gc.collect() return value

On Unix and Windows platforms the *gc.collect()* method returns an integer which signifies the number of distinct memory regions that were reclaimed in the collection (more precisely, the number of heads that were turned into frees). For efficiency reasons bare metal ports do not return this value.

2.8 MicroPython manifest files

When building firmware for a device the following components are included in the compilation process:

- the core MicroPython virtual machine and runtime
- port-specific system code and drivers to interface with the microcontroller/device that the firmware is targeting
- standard built-in modules, like `sys`
- extended built-in modules, like `json` and `machine`
- extra modules written in C/C++
- extra modules written in Python

All the modules included in the firmware are available via `import` from Python code. The extra modules written in Python that are included in a build (the last point above) are called *frozen modules*, and are specified by a `manifest.py` file. Changing this manifest requires rebuilding the firmware.

Its also possible to add additional modules to the filesystem of the device once it is up and running. Adding and removing modules to/from the filesystem does not require rebuilding the firmware so is a simpler process than rebuilding firmware. The benefit of using a manifest is that frozen modules are more efficient: they are faster to import and take up less RAM once imported.

MicroPython manifest files are Python files and can contain arbitrary Python code. There are also a set of commands (predefined functions) which are used to specify the Python source files to include. These commands are described below.

2.8.1 Freezing source code

freeze(*path*, *script=None*, *opt=0*)
Freeze the input specified by *path*, automatically determining its type. A `.py` script will be compiled to a `.mpy` first then frozen, and a `.mpy` file will be frozen directly.

path must be a directory, which is the base directory to begin searching for files. When importing the resulting frozen modules, the name of the module will start after *path*, i.e. *path* is excluded from the module name.

If *path* is relative, it is resolved to the current `manifest.py`. Use `$(MPY_DIR)`, `$(MPY_LIB_DIR)`, `$(PORT_DIR)`, `$(BOARD_DIR)` if you need to access specific paths.

If *script* is None, all files in *path* will be frozen.

If *script* is an iterable then `freeze()` is called on all items of the iterable (with the same *path* and *opt* passed through).

If *script* is a string then it specifies the file or directory to freeze, and can include extra directories before the file or last directory. The file or directory will be searched for in *path*. If *script* is a directory then all files in that directory will be frozen.

opt is the optimisation level to pass to mpy-cross when compiling `.py` to `.mpy`.

freeze_as_str(*path*)
Freeze the given *path* and all `.py` scripts within it as a string, which will be compiled upon import.

freeze_as_mpy(*path*, *script=None*, *opt=0*)
Freeze the input by first compiling the `.py` scripts to `.mpy` files, then freezing the resulting `.mpy` files. See `freeze()` for further details on the arguments.

freeze_mpy(*path*, *script=None*, *opt=0*)
Freeze the input, which must be `.mpy` files that are frozen directly. See `freeze()` for further details on the arguments.

2.8.2 Including other manifest files

include(*manifest*, ***kwargs*)
Include another manifest.

The *manifest* argument can be a string (filename) or an iterable of strings.

Relative paths are resolved with respect to the current manifest file.

Optional *kwargs* can be provided which will be available to the included script via the *options* variable.

For example:

```
include("path.py", extra_features=True)
```

then in path.py:

```
options.defaults(standard_features=True)
# freeze minimal modules.
if options.standard_features:
    # freeze standard modules.
if options.extra_features:
    # freeze extra modules.
```

2.8.3 Examples

To freeze a single file which is available as `import mydriver`, use:

```
freeze(".", "mydriver.py")
```

To freeze a set of files which are available as `import test1` and `import test2`, and which are compiled with optimisation level 3, use:

```
freeze("/path/to/tests", ("test1.py", "test2.py"), opt=3)
```

To freeze a module which can be imported as `import mymodule`, use:

```
freeze(
    "../relative/path",
    (
        "mymodule/__init__.py",
        "mymodule/core.py",
        "mymodule/extra.py",
    ),
)
```

To include a manifest from the MicroPython repository, use:

```
include("$(MPY_DIR)/extmod/uasyncio/manifest.py")
```

2.9 Distribution packages, package management, and deploying applications

Just as the big Python, MicroPython supports creation of third party packages, distributing them, and easily installing them in each users environment. This chapter discusses how these actions are achieved. Some familiarity with Python packaging is recommended.

2.9.1 Overview

Steps below represent a high-level workflow when creating and consuming packages:

1. Python modules and packages are turned into distribution package archives, and published at the Python Package Index (PyPI).
2. *upip* package manager can be used to install a distribution package on a *MicroPython port* with networking capabilities (for example, on the Unix port).
3. For ports without networking capabilities, an installation image can be prepared on the Unix port, and transferred to a device by suitable means.
4. For low-memory ports, the installation image can be frozen as the bytecode into MicroPython executable, thus minimizing the memory storage overheads.

The sections below describe this process in details.

2.9.2 Distribution packages

Python modules and packages can be packaged into archives suitable for transfer between systems, storing at the well-known location (PyPI), and downloading on demand for deployment. These archives are known as *distribution packages* (to differentiate them from Python packages (means to organize Python source code)).

The MicroPython distribution package format is a well-known tar.gz format, with some adaptations however. The Gzip compressor, used as an external wrapper for TAR archives, by default uses 32KB dictionary size, which means that to uncompress a compressed stream, 32KB of contiguous memory needs to be allocated. This requirement may be not satisfiable on low-memory devices, which may have total memory available less than that amount, and even if not, a contiguous block like that may be hard to allocate due to memory fragmentation. To accommodate these constraints, MicroPython distribution packages use Gzip compression with the dictionary size of 4K, which should be a suitable compromise with still achieving some compression while being able to uncompressed even by the smallest devices.

Besides the small compression dictionary size, MicroPython distribution packages also have other optimizations, like removing any files from the archive which arent used by the installation process. In particular, *upip* package manager doesnt execute `setup.py` during installation (see below), and thus that file is not included in the archive.

At the same time, these optimizations make MicroPython distribution packages not compatible with *CPython*s package manager, `pip`. This isnt considered a big problem, because:

1. Packages can be installed with *upip*, and then can be used with CPython (if they are compatible with it).
2. In the other direction, majority of CPython packages would be incompatible with MicroPython by various reasons, first of all, the reliance on features not implemented by MicroPython.

Summing up, the MicroPython distribution package archives are highly optimized for MicroPythons target environments, which are highly resource constrained devices.

2.9.3 `upip` package manager

MicroPython distribution packages are intended to be installed using the *upip* package manager. *upip* is a Python application which is usually distributed (as frozen bytecode) with network-enabled *MicroPython ports*. At the very least, *upip* is available in the *MicroPython Unix port*.

On any *MicroPython port* providing *upip*, it can be accessed as following:

```
import upip
upip.help()
upip.install(package_or_package_list, [path])
```

Where *package_or_package_list* is the name of a distribution package to install, or a list of such names to install multiple packages. Optional *path* parameter specifies filesystem location to install under and defaults to the standard library location (see below).

An example of installing a specific package and then using it:

```
>>> import upip
>>> upip.install("micropython-pystone_lowmem")
[...]
>>> import pystone_lowmem
>>> pystone_lowmem.main()
```

Note that the name of Python package and the name of distribution package for it in general dont have to match, and oftentimes they dont. This is because PyPI provides a central package repository for all different Python implementations and versions, and thus distribution package names may need to be namespaced for a particular implementation.

For example, all packages from *micropython-lib* follow this naming convention: for a Python module or package named `foo`, the distribution package name is `micropython-foo`.

For the ports which run MicroPython executable from the OS command prompts (like the Unix port), *upip* can be (and indeed, usually is) run from the command line instead of MicroPythons own REPL. The commands which corresponds to the example above are:

```
micropython -m upip -h
micropython -m upip install [-p <path>] <packages>...
micropython -m upip install micropython-pystone_lowmem
```

[TODO: Describe installation path.]

2.9.4 Cross-installing packages

For *MicroPython ports* without native networking capabilities, the recommend process is cross-installing them into a directory image using the *MicroPython Unix port*, and then transferring this image to a device by suitable means.

Installing to a directory image involves using `-p` switch to *upip*:

```
micropython -m upip install -p install_dir micropython-pystone_lowmem
```

After this command, the package content (and contents of every dependency packages) will be available in the `install_dir/` subdirectory. You would need to transfer contents of this directory (without the `install_dir/` prefix) to the device, at the suitable location, where it can be found by the Python `import` statement (see discussion of the *upip* installation path above).

2.9.5 Cross-installing packages with freezing

For the low-memory *MicroPython ports*, the process described in the previous section does not provide the most efficient resource usage,because the packages are installed in the source form, so need to be compiled to the bytecome on each import. This compilation requires RAM, and the resulting bytecode is also stored in RAM, reducing its amount available for storing application data. Moreover, the process above requires presence of the filesystem on a device, and the most resource-constrained devices may not even have it.

The bytecode freezing is a process which resolves all the issues mentioned above:

- The source code is pre-compiled into bytecode and store as such.
- The bytecode is stored in ROM, not RAM.
- Filesystem is not required for frozen packages.

Using frozen bytecode requires building the executable (firmware) for a given *MicroPython port* from the C source code. Consequently, the process is:

1. Follow the instructions for a particular port on setting up a toolchain and building the port. For example, for ESP8266 port, study instructions in `ports/esp8266/README.md` and follow them. Make sure you can build the port and deploy the resulting executable/firmware successfully before proceeding to the next steps.
2. Build *MicroPython Unix port* and make sure it is in your PATH and you can execute `micropython`.
3. Change to ports directory (e.g. `ports/esp8266/` for ESP8266).
4. Run `make clean-frozen`. This step cleans up any previous modules which were installed for freezing (consequently, you need to skip this step to add additional modules, instead of starting from scratch).
5. Run `micropython -m upip install -p modules <packages>...` to install packages you want to freeze.

6. Run `make clean`.
7. Run `make`.

After this, you should have the executable/firmware with modules as the bytecode inside, which you can deploy the usual way.

Few notes:

1. Step 5 in the sequence above assumes that the distribution package is available from PyPI. If that is not the case, you would need to copy Python source files manually to `modules/` subdirectory of the port directory. (Note that upip does not support installing from e.g. version control repositories).
2. The firmware for baremetal devices usually has size restrictions, so adding too many frozen modules may overflow it. Usually, you would get a linking error if this happens. However, in some cases, an image may be produced, which is not runnable on a device. Such cases are in general bugs, and should be reported and further investigated. If you face such a situation, as an initial step, you may want to decrease the amount of frozen modules included.

2.9.6 Creating distribution packages

Distribution packages for MicroPython are created in the same manner as for CPython or any other Python implementation, see references at the end of chapter. Setuptools (instead of distutils) should be used, because distutils do not support dependencies and other features. Source distribution (`sdist`) format is used for packaging. The post-processing discussed above, (and pre-processing discussed in the following section) is achieved by using custom `sdist` command for setuptools. Thus, packaging steps remain the same as for the standard setuptools, the user just needs to override `sdist` command implementation by passing the appropriate argument to `setup()` call:

```
from setuptools import setup
import sdist_upip

setup(
    ...,
    cmdclass={'sdist': sdist_upip.sdist}
)
```

The sdist_upip.py module as referenced above can be found in *micropython-lib*: https://github.com/micropython/micropython-lib/blob/master/sdist_upip.py

2.9.7 Application resources

A complete application, besides the source code, oftentimes also consists of data files, e.g. web page templates, game images, etc. Its clear how to deal with those when application is installed manually - you just put those data files in the filesystem at some location and use the normal file access functions.

The situation is different when deploying applications from packages - this is more advanced, streamlined and flexible way, but also requires more advanced approach to accessing data files. This approach is treating the data files as resources, and abstracting away access to them.

Python supports resource access using its setuptools library, using `pkg_resources` module. MicroPython, following its usual approach, implements subset of the functionality of that module, specifically `pkg_resources.resource_stream(package, resource)` function. The idea is that an application calls this function, passing a resource identifier, which is a relative path to data file within the specified package (usually top-level application package). It returns a stream object which can be used to access resource contents. Thus, the `resource_stream()` emulates interface of the standard *open()* function.

Implementation-wise, `resource_stream()` uses file operations underlyingly, if distribution package is install in the filesystem. However, it also supports functioning without the underlying filesystem, e.g. if the package is frozen as the bytecode. This however requires an extra intermediate step when packaging application - creation of Python resource module.

The idea of this module is to convert binary data to a Python bytes object, and put it into the dictionary, indexed by the resource name. This conversion is done automatically using overridden `sdist` command described in the previous section.

Lets trace the complete process using the following example. Suppose your application has the following structure:

```
my_app/
    __main__.py
    utils.py
    data/
        page.html
        image.png
```

`__main__.py` and `utils.py` should access resources using the following calls:

```
import pkg_resources

pkg_resources.resource_stream(__name__, "data/page.html")
pkg_resources.resource_stream(__name__, "data/image.png")
```

You can develop and debug using the *MicroPython Unix port* as usual. When time comes to make a distribution package out of it, just use overridden sdist command from sdist_upip.py module as described in the previous section.

This will create a Python resource module named `R.py`, based on the files declared in `MANIFEST` or `MANIFEST.in` files (any non-`.py` file will be considered a resource and added to `R.py`) - before proceeding with the normal packaging steps.

Prepared like this, your application will work both when deployed to filesystem and as frozen bytecode.

If you would like to debug `R.py` creation, you can run:

```
python3 setup.py sdist --manifest-only
```

Alternatively, you can use tools/mpy_bin2res.py script from the MicroPython distribution, in which can you will need to pass paths to all resource files:

```
mpy_bin2res.py data/page.html data/image.png
```

2.9.8 References

- Python Packaging User Guide: https://packaging.python.org/
- Setuptools documentation: https://setuptools.readthedocs.io/
- Distutils documentation: https://docs.python.org/3/library/distutils.html

2.10 Inline assembler for Thumb2 architectures

This document assumes some familiarity with assembly language programming and should be read after studying the *tutorial*. For a detailed description of the instruction set consult the Architecture Reference Manual detailed below. The inline assembler supports a subset of the ARM Thumb-2 instruction set described here. The syntax tries to be as close as possible to that defined in the above ARM manual, converted to Python function calls.

Instructions operate on 32 bit signed integer data except where stated otherwise. Most supported instructions operate on registers `R0-R7` only: where `R8-R15` are supported this is stated. Registers `R8-R12` must be restored to their initial value before return from a function. Registers `R13-R15` constitute the Link Register, Stack Pointer and Program Counter respectively.

2.10.1 Document conventions

Where possible the behaviour of each instruction is described in Python, for example

- add(Rd, Rn, Rm) `Rd = Rn + Rm`

This enables the effect of instructions to be demonstrated in Python. In certain case this is impossible because Python doesnt support concepts such as indirection. The pseudocode employed in such cases is described on the relevant page.

2.10.2 Instruction categories

The following sections details the subset of the ARM Thumb-2 instruction set supported by MicroPython.

Register move instructions

Document conventions

Notation: `Rd, Rn` denote ARM registers R0-R15. `immN` denotes an immediate value having a width of N bits. These instructions affect the condition flags.

Register moves

Where immediate values are used, these are zero-extended to 32 bits. Thus `mov(R0, 0xff)` will set R0 to 255.

- mov(Rd, imm8) `Rd = imm8`
- mov(Rd, Rn) `Rd = Rn`
- movw(Rd, imm16) `Rd = imm16`
- movt(Rd, imm16) `Rd = (Rd & 0xffff) | (imm16 << 16)`

movt writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

- movwt(Rd, imm32) `Rd = imm32`

movwt is a pseudo-instruction: the MicroPython assembler emits a `movw` followed by a `movt` to move a 32-bit value into Rd.

Load register from memory

Document conventions

Notation: `Rt, Rn` denote ARM registers R0-R7 except where stated. `immN` represents an immediate value having a width of N bits hence `imm5` is constrained to the range 0-31. `[Rn + immN]` is the contents of the memory address obtained by adding Rn and the offset `immN`. Offsets are measured in bytes. These instructions affect the condition flags.

Register Load

- ldr(Rt, [Rn, imm7]) `Rt = [Rn + imm7]` Load a 32 bit word
- ldrb(Rt, [Rn, imm5]) `Rt = [Rn + imm5]` Load a byte
- ldrh(Rt, [Rn, imm6]) `Rt = [Rn + imm6]` Load a 16 bit half word

Where a byte or half word is loaded, it is zero-extended to 32 bits.

The specified immediate offsets are measured in bytes. Hence in the case of `ldr` the 7 bit value enables 32 bit word aligned values to be accessed with a maximum offset of 31 words. In the case of `ldrh` the 6 bit value enables 16 bit half-word aligned values to be accessed with a maximum offset of 31 half-words.

Store register to memory

Document conventions

Notation: `Rt, Rn` denote ARM registers R0-R7 except where stated. `immN` represents an immediate value having a width of N bits hence `imm5` is constrained to the range 0-31. `[Rn + imm5]` is the contents of the memory address obtained by adding Rn and the offset `imm5`. Offsets are measured in bytes. These instructions do not affect the condition flags.

Register Store

- str(Rt, [Rn, imm7]) `[Rn + imm7] = Rt` Store a 32 bit word
- strb(Rt, [Rn, imm5]) `[Rn + imm5] = Rt` Store a byte (b0-b7)
- strh(Rt, [Rn, imm6]) `[Rn + imm6] = Rt` Store a 16 bit half word (b0-b15)

The specified immediate offsets are measured in bytes. Hence in the case of `str` the 7 bit value enables 32 bit word aligned values to be accessed with a maximum offset of 31 words. In the case of `strh` the 6 bit value enables 16 bit half-word aligned values to be accessed with a maximum offset of 31 half-words.

Logical & bitwise instructions

Document conventions

Notation: `Rd, Rn` denote ARM registers R0-R7 except in the case of the special instructions where R0-R15 may be used. `Rn<a-b>` denotes an ARM register whose contents must lie in range `a <= contents <= b`. In the case of instructions with two register arguments, it is permissible for them to be identical. For example the following will zero R0 (Python `R0 ^= R0`) regardless of its initial contents.

- eor(r0, r0)

These instructions affect the condition flags except where stated.

Logical instructions

- and_(Rd, Rn) `Rd &= Rn`
- orr(Rd, Rn) `Rd |= Rn`
- eor(Rd, Rn) `Rd ^= Rn`
- mvn(Rd, Rn) `Rd = Rn ^ 0xffffffff` i.e. Rd = 1s complement of Rn
- bic(Rd, Rn) `Rd &= ~Rn` bit clear Rd using mask in Rn

Note the use of and_ instead of and, because and is a reserved keyword in Python.

Shift and rotation instructions

- lsl(Rd, Rn<0-31>) `Rd <<= Rn`
- lsr(Rd, Rn<1-32>) `Rd = (Rd & 0xffffffff) >> Rn` Logical shift right
- asr(Rd, Rn<1-32>) `Rd >>= Rn` arithmetic shift right
- ror(Rd, Rn<1-31>) `Rd = rotate_right(Rd, Rn)` Rd is rotated right Rn bits.

A rotation by (for example) three bits works as follows. If Rd initially contains bits `b31 b30..b0` after rotation it will contain `b2 b1 b0 b31 b30..b3`

Special instructions

Condition codes are unaffected by these instructions.

- clz(Rd, Rn) `Rd = count_leading_zeros(Rn)`

count_leading_zeros(Rn) returns the number of binary zero bits before the first binary one bit in Rn.

- rbit(Rd, Rn) `Rd = bit_reverse(Rn)`

bit_reverse(Rn) returns the bit-reversed contents of Rn. If Rn contains bits `b31 b30..b0` Rd will be set to `b0 b1 b2..b31`

Trailing zeros may be counted by performing a bit reverse prior to executing clz.

Arithmetic instructions

Document conventions

Notation: `Rd, Rm, Rn` denote ARM registers R0-R7. `immN` denotes an immediate value having a width of N bits e.g. `imm8`, `imm3`. `carry` denotes the carry condition flag, `not(carry)` denotes its complement. In the case of instructions with more than one register argument, it is permissible for some to be identical. For example the following will add the contents of R0 to itself, placing the result in R0:

- add(r0, r0, r0)

Arithmetic instructions affect the condition flags except where stated.

Addition

- add(Rdn, imm8) `Rdn = Rdn + imm8`
- add(Rd, Rn, imm3) `Rd = Rn + imm3`
- add(Rd, Rn, Rm) `Rd = Rn +Rm`
- adc(Rd, Rn) `Rd = Rd + Rn + carry`

Subtraction

- sub(Rdn, imm8) `Rdn = Rdn - imm8`
- sub(Rd, Rn, imm3) `Rd = Rn - imm3`
- sub(Rd, Rn, Rm) `Rd = Rn - Rm`
- sbc(Rd, Rn) `Rd = Rd - Rn - not(carry)`

Negation

- neg(Rd, Rn) `Rd = -Rn`

Multiplication and division

- mul(Rd, Rn) `Rd = Rd * Rn`

This produces a 32 bit result with overflow lost. The result may be treated as signed or unsigned according to the definition of the operands.

- sdiv(Rd, Rn, Rm) `Rd = Rn / Rm`
- udiv(Rd, Rn, Rm) `Rd = Rn / Rm`

These functions perform signed and unsigned division respectively. Condition flags are not affected.

Comparison instructions

These perform an arithmetic or logical instruction on two arguments, discarding the result but setting the condition flags. Typically these are used to test data values without changing them prior to executing a conditional branch.

Document conventions

Notation: `Rd, Rm, Rn` denote ARM registers R0-R7. `imm8` denotes an immediate value having a width of 8 bits.

The Application Program Status Register (APSR)

This contains four bits which are tested by the conditional branch instructions. Typically a conditional branch will test multiple bits, for example `bge(LABEL)`. The meaning of condition codes can depend on whether the operands of an arithmetic instruction are viewed as signed or unsigned integers. Thus `bhi(LABEL)` assumes unsigned numbers were processed while `bgt(LABEL)` assumes signed operands.

APSR Bits

- Z (zero)

This is set if the result of an operation is zero or the operands of a comparison are equal.

- N (negative)

Set if the result is negative.

- C (carry)

An addition sets the carry flag when the result overflows out of the MSB, for example adding 0x80000000 and 0x80000000. By the nature of twos complement arithmetic this behaviour is reversed on subtraction, with a borrow indicated by the carry bit being clear. Thus 0x10 - 0x01 is executed as 0x10 + 0xffffffff which will set the carry bit.

- V (overflow)

The overflow flag is set if the result, viewed as a twos compliment number, has the wrong sign in relation to the operands. For example adding 1 to 0x7fffffff will set the overflow bit because the result (0x8000000), viewed as a twos complement integer, is negative. Note that in this instance the carry bit is not set.

Comparison instructions

These set the APSR (Application Program Status Register) N (negative), Z (zero), C (carry) and V (overflow) flags.

- cmp(Rn, imm8) `Rn - imm8`
- cmp(Rn, Rm) `Rn - Rm`
- cmn(Rn, Rm) `Rn + Rm`
- tst(Rn, Rm) `Rn & Rm`

Conditional execution

The `it` and `ite` instructions provide a means of conditionally executing from one to four subsequent instructions without the need for a label.

- it(<condition>) If then

Execute the next instruction if <condition> is true:

```
cmp(r0, r1)
it(eq)
mov(r0, 100) # runs if r0 == r1
# execution continues here
```

- ite(<condition>) If then else

If <condtion> is true, execute the next instruction, otherwise execute the subsequent one. Thus:

```
cmp(r0, r1)
ite(eq)
mov(r0, 100) # runs if r0 == r1
mov(r0, 200) # runs if r0 != r1
# execution continues here
```

This may be extended to control the execution of upto four subsequent instructions: it[x[y[z]]] where x,y,z=t/e; e.g. itt, itee, itete, ittte, itttt, iteee, etc.

Branch instructions

These cause execution to jump to a target location usually specified by a label (see the `label` assembler directive). Conditional branches and the `it` and `ite` instructions test the Application Program Status Register (APSR) N (negative), Z (zero), C (carry) and V (overflow) flags to determine whether the branch should be executed.

Most of the exposed assembler instructions (including move operations) set the flags but there are explicit comparison instructions to enable values to be tested.

Further detail on the meaning of the condition flags is provided in the section describing comparison functions.

Document conventions

Notation: `Rm` denotes ARM registers R0-R15. `LABEL` denotes a label defined with the `label()` assembler directive. `<condition>` indicates one of the following condition specifiers:

- eq Equal to (result was zero)
- ne Not equal
- cs Carry set
- cc Carry clear
- mi Minus (negative)
- pl Plus (positive)
- vs Overflow set
- vc Overflow clear
- hi > (unsigned comparison)
- ls <= (unsigned comparison)
- ge >= (signed comparison)
- lt < (signed comparison)
- gt > (signed comparison)
- le <= (signed comparison)

Branch to label

- b(LABEL) Unconditional branch
- beq(LABEL) branch if equal
- bne(LABEL) branch if not equal
- bge(LABEL) branch if greater than or equal
- bgt(LABEL) branch if greater than
- blt(LABEL) branch if less than (<) (signed)
- ble(LABEL) branch if less than or equal to (<=) (signed)
- bcs(LABEL) branch if carry flag is set
- bcc(LABEL) branch if carry flag is clear
- bmi(LABEL) branch if negative
- bpl(LABEL) branch if positive
- bvs(LABEL) branch if overflow flag set
- bvc(LABEL) branch if overflow flag is clear
- bhi(LABEL) branch if higher (unsigned)
- bls(LABEL) branch if lower or equal (unsigned)

Long branches

The code produced by the branch instructions listed above uses a fixed bit width to specify the branch destination, which is PC relative. Consequently in long programs where the branch instruction is remote from its destination the assembler will produce a branch not in range error. This can be overcome with the wide variants such as

- beq_w(LABEL) long branch if equal

Wide branches use 4 bytes to encode the instruction (compared with 2 bytes for standard branch instructions).

Subroutines (functions)

When entering a subroutine the processor stores the return address in register r14, also known as the link register (lr). Return to the instruction after the subroutine call is performed by updating the program counter (r15 or pc) from the link register, This process is handled by the following instructions.

- bl(LABEL)

Transfer execution to the instruction after `LABEL` storing the return address in the link register (r14).

- bx(Rm) Branch to address specified by Rm.

Typically `bx(lr)` is issued to return from a subroutine. For nested subroutines the link register of outer scopes must be saved (usually on the stack) before performing inner subroutine calls.

Stack push and pop

Document conventions

The `push()` and `pop()` instructions accept as their argument a register set containing a subset, or possibly all, of the general-purpose registers R0-R12 and the link register (lr or R14). As with any Python set the order in which the registers are specified is immaterial. Thus the in the following example the pop() instruction would restore R1, R7 and R8 to their contents prior to the push():

- push({r1, r8, r7}) Save three registers on the stack.
- pop({r7, r1, r8}) Restore them

Stack operations

- push({regset}) Push a set of registers onto the stack
- pop({regset}) Restore a set of registers from the stack

Miscellaneous instructions

- nop() `pass` no operation.
- wfi() Suspend execution in a low power state until an interrupt occurs.
- cpsid(flags) set the Priority Mask Register - disable interrupts.
- cpsie(flags) clear the Priority Mask Register - enable interrupts.
- mrs(Rd, special_reg) `Rd = special_reg` copy a special register to a general register. The special register may be IPSR (Interrupt Status Register) or BASEPRI (Base Priority Register). The IPSR provides a means of determining the exception number of an interrupt being processed. It contains zero if no interrupt is being processed.

Currently the `cpsie()` and `cpsid()` functions are partially implemented. They require but ignore the flags argument and serve as a means of enabling and disabling interrupts.

Floating point instructions

These instructions support the use of the ARM floating point coprocessor (on platforms such as the Pyboard which are equipped with one). The FPU has 32 registers known as `s0-s31` each of which can hold a single precision float. Data can be passed between the FPU registers and the ARM core registers with the `vmov` instruction.

Note that MicroPython doesnt support passing floats to assembler functions, nor can you put a float into `r0` and expect a reasonable result. There are two ways to overcome this. The first is to use arrays, and the second is to pass and/or return integers and convert to and from floats in code.

Document conventions

Notation: `Sd, Sm, Sn` denote FPU registers, `Rd, Rm, Rn` denote ARM core registers. The latter can be any ARM core register although registers `R13-R15` are unlikely to be appropriate in this context.

Arithmetic

- vadd(Sd, Sn, Sm) `Sd = Sn + Sm`
- vsub(Sd, Sn, Sm) `Sd = Sn - Sm`
- vneg(Sd, Sm) `Sd = -Sm`
- vmul(Sd, Sn, Sm) `Sd = Sn * Sm`
- vdiv(Sd, Sn, Sm) `Sd = Sn / Sm`
- vsqrt(Sd, Sm) `Sd = sqrt(Sm)`

Registers may be identical: `vmul(S0, S0, S0)` will execute `S0 = S0*S0`

Move between ARM core and FPU registers

- vmov(Sd, Rm) `Sd = Rm`
- vmov(Rd, Sm) `Rd = Sm`

The FPU has a register known as FPSCR, similar to the ARM cores APSR, which stores condition codes plus other data. The following instructions provide access to this.

- vmrs(APSR_nzcv, FPSCR)

Move the floating-point N, Z, C, and V flags to the APSR N, Z, C, and V flags.

This is done after an instruction such as an FPU comparison to enable the condition codes to be tested by the assembler code. The following is a more general form of the instruction.

- vmrs(Rd, FPSCR) `Rd = FPSCR`

Move between FPU register and memory

- vldr(Sd, [Rn, offset]) `Sd = [Rn + offset]`
- vstr(Sd, [Rn, offset]) `[Rn + offset] = Sd`

Where `[Rn + offset]` denotes the memory address obtained by adding Rn to the offset. This is specified in bytes. Since each float value occupies a 32 bit word, when accessing arrays of floats the offset must always be a multiple of four bytes.

Data comparison

- vcmp(Sd, Sm)

Compare the values in Sd and Sm and set the FPU N, Z, C, and V flags. This would normally be followed by `vmrs(APSR_nzcv, FPSCR)` to enable the results to be tested.

Convert between integer and float

- vcvt_f32_s32(Sd, Sm) `Sd = float(Sm)`
- vcvt_s32_f32(Sd, Sm) `Sd = int(Sm)`

Assembler directives

Labels

- label(INNER1)

This defines a label for use in a branch instruction. Thus elsewhere in the code a `b(INNER1)` will cause execution to continue with the instruction after the label directive.

Defining inline data

The following assembler directives facilitate embedding data in an assembler code block.

- data(size, d0, d1 .. dn)

The data directive creates n array of data values in memory. The first argument specifies the size in bytes of the subsequent arguments. Hence the first statement below will cause the assembler to put three bytes (with values 2, 3 and 4) into consecutive memory locations while the second will cause it to emit two four byte words.

```
data(1, 2, 3, 4)
data(4, 2, 100000)
```

Data values longer than a single byte are stored in memory in little-endian format.

- align(nBytes)

Align the following instruction to an nBytes value. ARM Thumb-2 instructions must be two byte aligned, hence its advisable to issue `align(2)` after `data` directives and prior to any subsequent code. This ensures that the code will run irrespective of the size of the data array.

2.10.3 Usage examples

These sections provide further code examples and hints on the use of the assembler.

Hints and tips

The following are some examples of the use of the inline assembler and some information on how to work around its limitations. In this document the term assembler function refers to a function declared in Python with the `@micropython.asm_thumb` decorator, whereas subroutine refers to assembler code called from within an assembler function.

Code branches and subroutines

It is important to appreciate that labels are local to an assembler function. There is currently no way for a subroutine defined in one function to be called from another.

To call a subroutine the instruction `bl(LABEL)` is issued. This transfers control to the instruction following the `label(LABEL)` directive and stores the return address in the link register (`lr` or `r14`). To return the instruction `bx(lr)` is issued which causes execution to continue with the instruction following the subroutine call. This mechanism implies that, if a subroutine is to call another, it must save the link register prior to the call and restore it before terminating.

The following rather contrived example illustrates a function call. Note that its necessary at the start to branch around all subroutine calls: subroutines end execution with `bx(lr)` while the outer function simply drops off the end in the style of Python functions.

```
@micropython.asm_thumb
def quad(r0):
    b(START)
    label(DOUBLE)
    add(r0, r0, r0)
    bx(lr)
    label(START)
    bl(DOUBLE)
    bl(DOUBLE)

print(quad(10))
```

The following code example demonstrates a nested (recursive) call: the classic Fibonacci sequence. Here, prior to a recursive call, the link register is saved along with other registers which the program logic requires to be preserved.

```
@micropython.asm_thumb
def fib(r0):
    b(START)
    label(DOFIB)
    push({r1, r2, lr})
    cmp(r0, 1)
    ble(FIBDONE)
    sub(r0, 1)
    mov(r2, r0) # r2 = n -1
    bl(DOFIB)
    mov(r1, r0) # r1 = fib(n -1)
    sub(r0, r2, 1)
    bl(DOFIB)   # r0 = fib(n -2)
    add(r0, r0, r1)
    label(FIBDONE)
    pop({r1, r2, lr})
    bx(lr)
    label(START)
    bl(DOFIB)
```

(continues on next page)

(continued from previous page)

```
for n in range(10):
    print(fib(n))
```

Argument passing and return

The tutorial details the fact that assembler functions can support from zero to three arguments, which must (if used) be named `r0`, `r1` and `r2`. When the code executes the registers will be initialised to those values.

The data types which can be passed in this way are integers and memory addresses. With current firmware all possible 32 bit values may be passed and returned. If the return value may have the most significant bit set a Python type hint should be employed to enable MicroPython to determine whether the value should be interpreted as a signed or unsigned integer: types are `int` or `uint`.

```
@micropython.asm_thumb
def uadd(r0, r1) -> uint:
    add(r0, r0, r1)
```

`hex(uadd(0x40000000,0x40000000))` will return 0x80000000, demonstrating the passing and return of integers where bits 30 and 31 differ.

The limitations on the number of arguments and return values can be overcome by means of the `array` module which enables any number of values of any type to be accessed.

Multiple arguments

If a Python array of integers is passed as an argument to an assembler function, the function will receive the address of a contiguous set of integers. Thus multiple arguments can be passed as elements of a single array. Similarly a function can return multiple values by assigning them to array elements. Assembler functions have no means of determining the length of an array: this will need to be passed to the function.

This use of arrays can be extended to enable more than three arrays to be used. This is done using indirection: the `uctypes` module supports `addressof()` which will return the address of an array passed as its argument. Thus you can populate an integer array with the addresses of other arrays:

```
from uctypes import addressof
@micropython.asm_thumb
def getindirect(r0):
    ldr(r0, [r0, 0]) # Address of array loaded from passed array
    ldr(r0, [r0, 4]) # Return element 1 of indirect array (24)

def testindirect():
    a = array.array('i',[23, 24])
    b = array.array('i',[0,0])
    b[0] = addressof(a)
    print(getindirect(b))
```

Non-integer data types

These may be handled by means of arrays of the appropriate data type. For example, single precision floating point data may be processed as follows. This code example takes an array of floats and replaces its contents with their squares.

```
from array import array

@micropython.asm_thumb
def square(r0, r1):
    label(LOOP)
    vldr(s0, [r0, 0])
    vmul(s0, s0, s0)
    vstr(s0, [r0, 0])
    add(r0, 4)
    sub(r1, 1)
    bgt(LOOP)

a = array('f', (x for x in range(10)))
square(a, len(a))
print(a)
```

The uctypes module supports the use of data structures beyond simple arrays. It enables a Python data structure to be mapped onto a bytearray instance which may then be passed to the assembler function.

Named constants

Assembler code may be made more readable and maintainable by using named constants rather than littering code with numbers. This may be achieved thus:

```
MYDATA = const(33)

@micropython.asm_thumb
def foo():
    mov(r0, MYDATA)
```

The const() construct causes MicroPython to replace the variable name with its value at compile time. If constants are declared in an outer Python scope they can be shared between multiple assembler functions and with Python code.

Assembler code as class methods

MicroPython passes the address of the object instance as the first argument to class methods. This is normally of little use to an assembler function. It can be avoided by declaring the function as a static method thus:

```
class foo:
  @staticmethod
  @micropython.asm_thumb
  def bar(r0):
    add(r0, r0, r0)
```

Use of unsupported instructions

These can be coded using the data statement as shown below. While `push()` and `pop()` are supported the example below illustrates the principle. The necessary machine code may be found in the ARM v7-M Architecture Reference Manual. Note that the first argument of data calls such as

```
data(2, 0xe92d, 0x0f00) # push r8,r9,r10,r11
```

indicates that each subsequent argument is a two byte quantity.

Overcoming MicroPythons integer restriction

The Pyboard chip includes a CRC generator. Its use presents a problem in MicroPython because the returned values cover the full gamut of 32 bit quantities whereas small integers in MicroPython cannot have differing values in bits 30 and 31. This limitation is overcome with the following code, which uses assembler to put the result into an array and Python code to coerce the result into an arbitrary precision unsigned integer.

```
from array import array
import stm

def enable_crc():
    stm.mem32[stm.RCC + stm.RCC_AHB1ENR] |= 0x1000

def reset_crc():
    stm.mem32[stm.CRC+stm.CRC_CR] = 1

@micropython.asm_thumb
def getval(r0, r1):
    movwt(r3, stm.CRC + stm.CRC_DR)
    str(r1, [r3, 0])
    ldr(r2, [r3, 0])
    str(r2, [r0, 0])

def getcrc(value):
    a = array('i', [0])
    getval(a, value)
    return a[0] & 0xffffffff # coerce to arbitrary precision

enable_crc()
reset_crc()
for x in range(20):
    print(hex(getcrc(0)))
```

2.10.4 References

- *Assembler Tutorial*
- Wiki hints and tips
- uPy Inline Assembler source-code, emitinlinethumb.c
- ARM Thumb2 Instruction Set Quick Reference Card
- RM0090 Reference Manual
- ARM v7-M Architecture Reference Manual (Available on the ARM site after a simple registration procedure. Also available on academic sites but beware of out of date versions.)

2.11 Working with filesystems

Contents

This tutorial describes how MicroPython provides an on-device filesystem, allowing standard Python file I/O methods to be used with persistent storage.

MicroPython automatically creates a default configuration and auto-detects the primary filesystem, so this tutorial will be mostly useful if you want to modify the partitioning, filesystem type, or use custom block devices.

The filesystem is typically backed by internal flash memory on the device, but can also use external flash, RAM, or a custom block device.

On some ports (e.g. STM32), the filesystem may also be available over USB MSC to a host PC. *The pyboard.py tool* also provides a way for the host PC to access to the filesystem on all ports.

Note: This is mainly for use on bare-metal ports like STM32 and ESP32. On ports with an operating system (e.g. the Unix port) the filesystem is provided by the host OS.

2.11.1 VFS

MicroPython implements a Unix-like Virtual File System (VFS) layer. All mounted filesystems are combined into a single virtual filesystem, starting at the root `/`. Filesystems are mounted into directories in this structure, and at startup the working directory is changed to where the primary filesystem is mounted.

On STM32 / Pyboard, the internal flash is mounted at `/flash`, and optionally the SDCard at `/sd`. On ESP8266/ESP32, the primary filesystem is mounted at `/`.

2.11.2 Block devices

A block device is an instance of a class that implements the *os.AbstractBlockDev* protocol.

Built-in block devices

Ports provide built-in block devices to access their primary flash.

On power-on, MicroPython will attempt to detect the filesystem on the default flash and configure and mount it automatically. If no filesystem is found, MicroPython will attempt to create a FAT filesystem spanning the entire flash. Ports can also provide a mechanism to factory reset the primary flash, usually by some combination of button presses at power on.

STM32 / Pyboard

The *pyb.Flash* class provides access to the internal flash. On some boards which have larger external flash (e.g. Pyboard D), it will use that instead. The `start` kwarg should always be specified, i.e. `pyb.Flash(start=0)`.

Note: For backwards compatibility, when constructed with no arguments (i.e. `pyb.Flash()`), it only implements the simple block interface and reflects the virtual device presented to USB MSC (i.e. it includes a virtual partition table at the start).

ESP8266

The internal flash is exposed as a block device object which is created in the `flashbdev` module on start up. This object is by default added as a global variable so it can usually be accessed simply as `bdev`. This implements the extended interface.

ESP32

The *esp32.Partition* class implements a block device for partitions defined for the board. Like ESP8266, there is a global variable `bdev` which points to the default partition. This implements the extended interface.

Custom block devices

The following class implements a simple block device that stores its data in RAM using a `bytearray`:

```
class RAMBlockDev:
    def __init__(self, block_size, num_blocks):
        self.block_size = block_size
        self.data = bytearray(block_size * num_blocks)

    def readblocks(self, block_num, buf):
        for i in range(len(buf)):
            buf[i] = self.data[block_num * self.block_size + i]

    def writeblocks(self, block_num, buf):
        for i in range(len(buf)):
            self.data[block_num * self.block_size + i] = buf[i]

    def ioctl(self, op, arg):
        if op == 4: # get number of blocks
            return len(self.data) // self.block_size
        if op == 5: # get block size
            return self.block_size
```

It can be used as follows:

```
import os

bdev = RAMBlockDev(512, 50)
os.VfsFat.mkfs(bdev)
os.mount(bdev, '/ramdisk')
```

An example of a block device that supports both the simple and extended interface (i.e. both signatures and behaviours of the *os.AbstractBlockDev.readblocks()* and *os.AbstractBlockDev.writeblocks()* methods) is:

```
class RAMBlockDev:
    def __init__(self, block_size, num_blocks):
        self.block_size = block_size
        self.data = bytearray(block_size * num_blocks)

    def readblocks(self, block_num, buf, offset=0):
        addr = block_num * self.block_size + offset
        for i in range(len(buf)):
            buf[i] = self.data[addr + i]

    def writeblocks(self, block_num, buf, offset=None):
        if offset is None:
            # do erase, then write
            for i in range(len(buf) // self.block_size):
                self.ioctl(6, block_num + i)
            offset = 0
        addr = block_num * self.block_size + offset
        for i in range(len(buf)):
            self.data[addr + i] = buf[i]
```

(continues on next page)

(continued from previous page)

```
    def ioctl(self, op, arg):
        if op == 4: # block count
            return len(self.data) // self.block_size
        if op == 5: # block size
            return self.block_size
        if op == 6: # block erase
            return 0
```

As it supports the extended interface, it can be used with *littlefs*:

```
import os

bdev = RAMBlockDev(512, 50)
os.VfsLfs2.mkfs(bdev)
os.mount(bdev, '/ramdisk')
```

Once mounted, the filesystem (regardless of its type) can be used as it normally would be used from Python code, for example:

```
with open('/ramdisk/hello.txt', 'w') as f:
    f.write('Hello world')
print(open('/ramdisk/hello.txt').read())
```

2.11.3 Filesystems

MicroPython ports can provide implementations of *FAT*, *littlefs v1* and *littlefs v2*.

The following table shows which filesystems are included in the firmware by default for given port/board combinations, however they can be optionally enabled in a custom firmware build.

Board	FAT	littlefs v1	littlefs v2
pyboard 1.0, 1.1, D	Yes	No	Yes
Other STM32	Yes	No	No
ESP8266 (1M)	No	No	Yes
ESP8266 (2M+)	Yes	No	Yes
ESP32	Yes	No	Yes

FAT

The main advantage of the FAT filesystem is that it can be accessed over USB MSC on supported boards (e.g. STM32) without any additional drivers required on the host PC.

However, FAT is not tolerant to power failure during writes and this can lead to filesystem corruption. For applications that do not require USB MSC, it is recommended to use littlefs instead.

To format the entire flash using FAT:

```
# ESP8266 and ESP32
import os
os.umount('/')
os.VfsFat.mkfs(bdev)
```

(continues on next page)

(continued from previous page)

```
os.mount(bdev, '/')

# STM32
import os, pyb
os.umount('/flash')
os.VfsFat.mkfs(pyb.Flash(start=0))
os.mount(pyb.Flash(start=0), '/flash')
os.chdir('/flash')
```

Littlefs

Littlefs is a filesystem designed for flash-based devices, and is much more resistant to filesystem corruption.

Note: There are reports of littlefs v1 and v2 failing in certain situations, for details see littlefs issue 347 and littlefs issue 295.

To format the entire flash using littlefs v2:

```
# ESP8266 and ESP32
import os
os.umount('/')
os.VfsLfs2.mkfs(bdev)
os.mount(bdev, '/')

# STM32
import os, pyb
os.umount('/flash')
os.VfsLfs2.mkfs(pyb.Flash(start=0))
os.mount(pyb.Flash(start=0), '/flash')
os.chdir('/flash')
```

A littlefs filesystem can be still be accessed on a PC over USB MSC using the littlefs FUSE driver. Note that you must specify both the `--block_size` and `--block_count` options to override the defaults. For example (after building the littlefs-fuse executable):

```
$ ./lfs --block_size=4096 --block_count=512 -o allow_other /dev/sdb1 mnt
```

This will allow the boards littlefs filesystem to be accessed at the `mnt` directory. To get the correct values of `block_size` and `block_count` use:

```
import pyb
f = pyb.Flash(start=0)
f.ioctl(1, 1)  # initialise flash in littlefs raw-block mode
block_count = f.ioctl(4, 0)
block_size = f.ioctl(5, 0)
```

Hybrid (STM32)

By using the `start` and `len` kwargs to `pyb.Flash`, you can create block devices spanning a subset of the flash device.

For example, to configure the first 256kiB as FAT (and available over USB MSC), and the remainder as littlefs:

```
import os, pyb
os.umount('/flash')
p1 = pyb.Flash(start=0, len=256*1024)
p2 = pyb.Flash(start=256*1024)
os.VfsFat.mkfs(p1)
os.VfsLfs2.mkfs(p2)
os.mount(p1, '/flash')
os.mount(p2, '/data')
os.chdir('/flash')
```

This might be useful to make your Python files, configuration and other rarely-modified content available over USB MSC, but allowing for frequently changing application data to reside on littlefs with better resilience to power failure, etc.

The partition at offset `0` will be mounted automatically (and the filesystem type automatically detected), but you can add:

```
import os, pyb
p2 = pyb.Flash(start=256*1024)
os.mount(p2, '/data')
```

to `boot.py` to mount the data partition.

Hybrid (ESP32)

On ESP32, if you build custom firmware, you can modify `partitions.csv` to define an arbitrary partition layout.

At boot, the partition named vfs will be mounted at `/` by default, but any additional partitions can be mounted in your `boot.py` using:

```
import esp32, os
p = esp32.Partition.find(esp32.Partition.TYPE_DATA, label='foo')
os.mount(p, '/foo')
```

2.12 The pyboard.py tool

This is a standalone Python tool that runs on your PC that provides a way to:

- Quickly run a Python script or command on a MicroPython device. This is useful while developing MicroPython programs to quickly test code without needing to copy files to/from the device.
- Access the filesystem on a device. This allows you to deploy your code to the device (even if the board doesnt support USB MSC).

Despite the name, `pyboard.py` works on all MicroPython ports that support the raw REPL (including STM32, ESP32, ESP8266, NRF).

You can download the latest version from GitHub. The only dependency is the `pyserial` library which can be installed from PiPy or your system package manager.

Running `pyboard.py --help` gives the following output:

```
usage: pyboard [-h] [-d DEVICE] [-b BAUDRATE] [-u USER] [-p PASSWORD]
               [-c COMMAND] [-w WAIT] [--follow | --no-follow] [-f]
               [files [files ...]]

Run scripts on the pyboard.

positional arguments:
  files                 input files

optional arguments:
  -h, --help            show this help message and exit
  -d DEVICE, --device DEVICE
                        the serial device or the IP address of the pyboard
  -b BAUDRATE, --baudrate BAUDRATE
                        the baud rate of the serial device
  -u USER, --user USER  the telnet login username
  -p PASSWORD, --password PASSWORD
                        the telnet login password
  -c COMMAND, --command COMMAND
                        program passed in as string
  -w WAIT, --wait WAIT  seconds to wait for USB connected board to become
                        available
  --follow              follow the output after running the scripts
                        [default if no scripts given]
  -f, --filesystem      perform a filesystem action: cp local :device | cp
                        :device local | cat path | ls [path] | rm path | mkdir
                        path | rmdir path
```

2.12.1 Running a command on the device

This is useful for testing short snippets of code, or to script an interaction with the device.:

```
$ pyboard.py --device /dev/ttyACM0 -c 'print(1+1)'
2
```

If you are often interacting with the same device, you can set the environment variable `PYBOARD_DEVICE` as an alternative to using the `--device` command line option. For example, the following is equivalent to the previous example:

```
$ export PYBOARD_DEVICE=/dev/ttyACM0
$ pyboard.py -c 'print(1+1)'
```

Similarly, the `PYBOARD_BAUDRATE` environment variable can be used to set the default for the `--baudrate` option.

2.12.2 Running a script on the device

If you have a script, `app.py` that you want to run on a device, then use:

```
$ pyboard.py --device /dev/ttyACM0 app.py
```

Note that this doesnt actually copy app.py to the devices filesystem, it just loads the code into RAM and executes it. Any output generated by the program will be displayed.

If the program app.py does not finish then youll need to stop `pyboard.py`, eg with Ctrl-C. The program `app.py` will still continue to run on the MicroPython device.

2.12.3 Filesystem access

Using the `-f` flag, the following filesystem operations are supported:

- `cp src [src...] dest` Copy files to/from the device.
- `cat path` Print the contents of a file on the device.
- `ls [path]` List contents of a directory (defaults to current working directory).
- `rm path` Remove a file.
- `mkdir path` Create a directory.
- `rmdir path` Remove a directory.

The `cp` command uses a `ssh`-like convention for referring to local and remote files. Any path starting with a `:` will be interpreted as on the device, otherwise it will be local. So:

```
$ pyboard.py --device /dev/ttyACM0 -f cp main.py :main.py
```

will copy main.py from the current directory on the PC to a file named main.py on the device. The filename can be omitted, e.g.:

```
$ pyboard.py --device /dev/ttyACM0 -f cp main.py :
```

is equivalent to the above.

Some more examples:

```
# Copy main.py from the device to the local PC.
$ pyboard.py --device /dev/ttyACM0 -f cp :main.py main.py
# Same, but using . instead.
$ pyboard.py --device /dev/ttyACM0 -f cp :main.py .

# Copy three files to the device, keeping their names
# and paths (note: `lib` must exist on the device)
$ pyboard.py --device /dev/ttyACM0 -f cp main.py app.py lib/foo.py :

# Remove a file from the device.
$ pyboard.py --device /dev/ttyACM0 -f rm util.py

# Print the contents of a file on the device.
$ pyboard.py --device /dev/ttyACM0 -f cat boot.py
...contents of boot.py...
```

2.12.4 Using the pyboard library

You can also use `pyboard.py` as a library for scripting interactions with a MicroPython board.

```
import pyboard
pyb = pyboard.Pyboard('/dev/ttyACM0', 115200)
pyb.enter_raw_repl()
ret = pyb.exec('print(1+1)')
print(ret)
pyb.exit_raw_repl()
```

CHAPTER

THREE

MICROPYTHON DIFFERENCES FROM CPYTHON

MicroPython implements Python 3.4 and some select features of Python 3.5 and above. The sections below describe the current status of these features.

3.1 Python 3.5

Below is a list of finalised/accepted PEPs for Python 3.5 grouped into their impact to MicroPython.

Extensions to the syntax:		**Status**
PEP 448	additional unpacking generalizations	
PEP 465	a new matrix multiplication operator	Completed
PEP 492	coroutines with async and await syntax	Completed
Extensions and changes to runtime:		
PEP 461	% formatting for binary strings	Completed
PEP 475	retrying system calls that fail with EINTR	Completed
PEP 479	change StopIteration handling inside generators	Completed
Standard library changes:		
PEP 471	os.scandir()	
PEP 485	math.isclose(), a function for testing approximate equality	Completed
Miscellaneous changes:		
PEP 441	improved Python zip application support	
PEP 486	make the Python Laucher aware of virtual environments	
PEP 484	type hints (advisory only)	In Progress
PEP 488	elimination of PYO files	Not relevant
PEP 489	redesigning extension module loading	

Other Language Changes:

Added the *namereplace* error handlers. The *backslashreplace* error handlers now work with decoding and translating.	
Property docstrings are now writable. This is especially useful for collections.namedtuple() docstrings	
Circular imports involving relative imports are now supported.	

New Modules:

- typing
- zipzap

Changes to built-in modules:

The *OrderedDict* class is now implemented in C, which makes it 4 to 100 times faster.
OrderedDict.items() , *OrderedDict.keys()* , *OrderedDict.values()* views now support reversed() iteration.
The deque class now defines *index()*, *insert()*, and *copy()*, and supports the + and * operators.
Docstrings produced by namedtuple() can now be updated.
The UserString class now implements the *__getnewargs__()*, *__rmod__()*, *casefold()*, *format_map()*, *isprintable()*, and *maketrans()* me
Element comparison in *merge()* can now be customized by passing a key function in a new optional key keyword argument, and a new
A new *BufferedIOBase.readinto1()* method, that uses at most one call to the underlying raw streams *RawIOBase.read()* or *RawIOBase*
json
JSON decoder now raises JSONDecodeError instead of ValueError to provide better context information about the error.
Two new constants have been added to the math module: *inf* and *nan*.
A new function *isclose()* provides a way to test for approximate equality.
A new *gcd()* function has been added. The *fractions.gcd()* function is now deprecated.
The new *scandir()* function returning an iterator of DirEntry objects has been added.
The *urandom()* function now uses the *getrandom()* syscall on Linux 3.17 or newer, and *getentropy()* on OpenBSD 5.6 and newer, remo
New *get_blocking()* and *set_blocking()* functions allow getting and setting a file descriptors blocking mode (O_NONBLOCK.)
There is a new *os.path.commonpath()* function returning the longest common sub-path of each passed pathname
re
References and conditional references to groups with fixed length are now allowed in lookbehind assertions
The number of capturing groups in regular expressions is no longer limited to 100.
The *sub()* and *subn()* functions now replace unmatched groups with empty strings instead of raising an exception.
The *re.error* exceptions have new attributes, msg, pattern, pos, lineno, and colno, that provide better context information about the erro
Functions with timeouts now use a monotonic clock, instead of a system clock.
A new *socket.sendfile()* method allows sending a file over a socket by using the high-performance *os.sendfile()* function on UNIX, resu
The *socket.sendall()* method no longer resets the socket timeout every time bytes are received or sent. The socket timeout is now the m
The backlog argument of the *socket.listen()* method is now optional. By default it is set to SOMAXCONN or to 128, whichever is less
Memory BIO Support
Application-Layer Protocol Negotiation Support
There is a new *SSLSocket.version()* method to query the actual protocol version in use.
The SSLSocket class now implements a *SSLSocket.sendfile()* method.
The *SSLSocket.send()* method now raises either the *ssl.SSLWantReadError* or *ssl.SSLWantWriteError* exception on a non-blocking soc
The *cert_time_to_seconds()* function now interprets the input time as UTC and not as local time, per RFC 5280. Additionally, the retu
New *SSLObject.shared_ciphers()* and *SSLSocket.shared_ciphers()* methods return the list of ciphers sent by the client during the hands
The *SSLSocket.do_handshake()*, *SSLSocket.read()*, *SSLSocket.shutdown()*, and *SSLSocket.write()* methods of the SSLSocket class no l
The *match_hostname()* function now supports matching of IP addresses.
A new *set_coroutine_wrapper()* function allows setting a global hook that will be called whenever a coroutine object is created by an a
A new *is_finalizing()* function can be used to check if the Python interpreter is shutting down.
The *monotonic()* function is now always available

3.2 Python 3.6

Python 3.6 beta 1 was released on 12 Sep 2016, and a summary of the new features can be found here:

New Syntax Features:		**Status**
PEP 498	Literal String Formatting	
PEP 515	Underscores in Numeric Literals	
PEP 525	Asynchronous Generators	
PEP 526	Syntax for Variable Annotations (provisional)	
PEP 530	Asynchronous Comprehensions	
New Built-in Features:		
PEP 468	Preserving the order of *kwargs* in a function	
PEP 487	Simpler customization of class creation	
PEP 520	Preserving Class Attribute Definition Order	
Standard Library Changes:		
PEP 495	Local Time Disambiguation	
PEP 506	Adding A Secrets Module To The Standard Library	
PEP 519	Adding a file system path protocol	
CPython internals:		
PEP 509	Add a private version to dict	
PEP 523	Adding a frame evaluation API to CPython	
Linux/Window Changes		
PEP 524	Make os.urandom() blocking on Linux (during system startup)	
PEP 528	Change Windows console encoding to UTF-8	
PEP 529	Change Windows filesystem encoding to UTF-8	

Other Language Changes:

A *global* or *nonlocal* statement must now textually appear before the first use of the affected name in the same scope. Previously this was a SyntaxWarning.	
It is now possible to set a special method to None to indicate that the corresponding operation is not available. For example, if a class sets *__iter__()* to *None* , the class is not iterable.	
Long sequences of repeated traceback lines are now abbreviated as *[Previous line repeated {count} more times]*	
Import now raises the new exception *ModuleNotFoundError* when it cannot find a module. Code that currently checks for ImportError (in try-except) will still work.	
Class methods relying on zero-argument *super()* will now work correctly when called from metaclass methods during class creation.	

Changes to built-in modules:

array
Exhausted iterators of *array.array* will now stay exhausted even if the iterated array is extended.
binascii
The b2a_base64() function now accepts an optional newline keyword argument to control whether the newline character is appended t
cmath
The new cmath.tau () constant has been added
New constants: *cmath.inf* and *cmath.nan* to match *math.inf* and *math.nan* , and also *cmath.infj* and *cmath.nanj* to match the format us

Table 2 – continued from previous page

The new Collection abstract base class has been added to represent sized iterable container classes
The new *Reversible* abstract base class represents iterable classes that also provide the *__reversed__()* method.
The new *AsyncGenerator* abstract base class represents asynchronous generators.
The *namedtuple()* function now accepts an optional keyword argument module, which, when specified, is used for the *__module__* att
The verbose and rename arguments for *namedtuple()* are now keyword-only.
Recursive *collections.deque* instances can now be pickled.
BLAKE2 hash functions were added to the module. *blake2b()* and *blake2s()* are always available and support the full feature set of BL
The SHA-3 hash functions *sha3_224()*, *sha3_256()*, *sha3_384()*, *sha3_512()*, and *SHAKE* hash functions *shake_128()* and *shake_256*
The password-based key derivation function *scrypt()* is now available with OpenSSL 1.1.0 and newer.
json.load() and *json.loads()* now support binary input. Encoded JSON should be represented using either UTF-8, UTF-16, or UTF-32
The new math.tau () constant has been added
A new *close()* method allows explicitly closing a *scandir()* iterator. The *scandir()* iterator now supports the context manager protocol.
On Linux, *os.urandom()* now blocks until the system urandom entropy pool is initialized to increase the security.
The Linux *getrandom()* syscall (get random bytes) is now exposed as the new *os.getrandom()* function.
Added support of modifier spans in regular expressions. Examples: *(?i:p)ython* matches python and Python, but not PYTHON; *(?i)g(*
Match object groups can be accessed by *__getitem__*, which is equivalent to *group()*. So *mo[name]* is now equivalent to *mo.group(na*
Match objects now support index-like objects as group indices.
The *ioctl()* function now supports the *SIO_LOOPBACK_FAST_PATH* control code.
The *getsockopt()* constants *SO_DOMAIN* , *SO_PROTOCOL*, *SO_PEERSEC* , and *SO_PASSSEC* are now supported.
The *setsockopt()* now supports the *setsockopt(level, optname, None, optlen: int)* form.
The socket module now supports the address family *AF_ALG* to interface with Linux Kernel crypto API. *ALG_*, *SOL_ALG* and *sendm*
New Linux constants *TCP_USER_TIMEOUT* and *TCP_CONGESTION* were added.
ssl supports OpenSSL 1.1.0. The minimum recommend version is 1.0.2.
3DES has been removed from the default cipher suites and ChaCha20 Poly1305 cipher suites have been added.
SSLContext has better default configuration for options and ciphers.
SSL session can be copied from one client-side connection to another with the new *SSLSession* class. TLS session resumption can spe
The new *get_ciphers()* method can be used to get a list of enabled ciphers in order of cipher priority.
All constants and flags have been converted to *IntEnum* and *IntFlags*.
Server and client-side specific TLS protocols for *SSLContext* were added.
Added *SSLContext.post_handshake_auth* to enable and *ssl.SSLSocket.verify_client_post_handshake()* to initiate TLS 1.3 post-handsha
struct
now supports IEEE 754 half-precision floats via the e format specifier.
sys
The new *getfilesystemencodeerrors()* function returns the name of the error mode used to convert between Unicode filenames and byte
zlib
The *compress()* and *decompress()* functions now accept keyword arguments

3.3 Python 3.7

New Features:

Features:		Status
PEP 538	Coercing the legacy C locale to a UTF-8 based locale	
PEP 539	A New C-API for Thread-Local Storage in CPython	
PEP 540	UTF-8 mode	
PEP 552	Deterministic pyc	
PEP 553	Built-in breakpoint()	
PEP 557	Data Classes	
PEP 560	Core support for typing module and generic types	
PEP 562	Module __getattr__ and __dir__	Partially done
PEP 563	Postponed Evaluation of Annotations	
PEP 564	Time functions with nanosecond resolution	
PEP 565	Show DeprecationWarning in __main__	
PEP 567	Context Variables	

Other Language Changes:

async and await are now reserved keywords	Completed
dict objects must preserve insertion-order	
More than 255 arguments can now be passed to a function; a function can now have more than 255 parameters	
bytes.fromhex() and bytearray.fromhex() now ignore all ASCII whitespace, not only spaces	
str, bytes, and bytearray gained support for the new isascii() method, which can be used to test if a string or bytes contain only the ASCII characters	
ImportError now displays module name and module __file__ path whenfrom import fails	
Circular imports involving absolute imports with binding a submodule to a name are now supported	
object.__format__(x,) is now equivalent to str(x) rather than format(str(self),)	
In order to better support dynamic creation of stack traces, types.TracebackType can now be instantiated from Python code, and the tb_next attribute on tracebacks is now writable	
When using the -m switch, sys.path[0] is now eagerly expanded to the full starting directory path, rather than being left as the empty directory (which allows imports from the current working directory at the time when an import occurs)	
The new -X importtime option or the PYTHONPROFILEIMPORTTIME environment variable can be used to show the timing of each module import	

Changes to built-in modules:

asyncio	
asyncio (many, may need a separate ticket)	
gc	
New features include *gc.freeze()*, *gc.unfreeze()*, *gc-get_freeze_count*	
math	
math.remainder() added to implement IEEE 754-style remainder	
re	
A number of tidy up features including better support for splitting on empty strings and copy support for compiled expressions and match objects	
sys	
sys.breakpointhook() added. sys.get(/set)_coroutine_origin_tracking_depth() added	
time	
Mostly updates to support nanosecond resolution in PEP564, see above	

3.4 Python 3.8

Python 3.8.0 (final) was released on the 14 October 2019. The Features for 3.8 are defined in PEP 569 and a detailed description of the changes can be found in Whats New in Python 3.8.

Features:		Status
PEP 570	Positional-only arguments	
PEP 572	Assignment Expressions	
PEP 574	Pickle protocol 5 with out-of-band data	
PEP 578	Runtime audit hooks	
PEP 587	Python Initialization Configuration	
PEP 590	Vectorcall: a fast calling protocol for CPython	
Miscellaneous		
f-strings support = for self-documenting expressions and debugging		Completed

Other Language Changes:

A *continue* statement was illegal in the *finally* clause due to a problem with the implementation. In Python 3.8 this restriction was lifted	Completed
The *bool*, *int* , and *fractions.Fraction* types now have an *as_integer_ratio()* method like that found in *float* and *decimal.Decimal*	
Constructors of *int*, *float* and *complex* will now use the *__index__()* special method, if available and the corresponding method *__int__()*, *__float__()* or *__complex__()* is not available	
Added support of *N{name}* escapes in regular expressions	
Dict and dictviews are now iterable in reversed insertion order using *reversed()*	
The syntax allowed for keyword names in function calls was further restricted. In particular, f((keyword)=arg) is no longer allowed	
Generalized iterable unpacking in yield and return statements no longer requires enclosing parentheses	
When a comma is missed in code such as [(10, 20) (30, 40)], the compiler displays a SyntaxWarning with a helpful suggestion	
Arithmetic operations between subclasses of *datetime.date* or *datetime.datetime* and *datetime.timedelta* objects now return an instance of the subclass, rather than the base class	
When the Python interpreter is interrupted by *Ctrl-C (SIGINT)* and the resulting *KeyboardInterrupt* exception is not caught, the Python process now exits via a SIGINT signal or with the correct exit code such that the calling process can detect that it died due to a *Ctrl-C*	
Some advanced styles of programming require updating the *types.CodeType* object for an existing function	
For integers, the three-argument form of the pow() function now permits the exponent to be negative in the case where the base is relatively prime to the modulus	
Dict comprehensions have been synced-up with dict literals so that the key is computed first and the value second	
The *object.__reduce__()* method can now return a tuple from two to six elements long	

Changes to built-in modules:

asyncio	
asyncio.run() has graduated from the provisional to stable API	Completed
Running *python -m asyncio* launches a natively async REPL	
The exception *asyncio.CancelledError* now inherits from *BaseException* rather than *Exception* and no longer inherits from *concurrent.futures.CancelledError*	Completed
Added *asyncio.Task.get_coro()* for getting the wrapped coroutine within an *asyncio.Task*	
Asyncio tasks can now be named, either by passing the name keyword argument to *asyncio.create_task()* or the *create_task()* event loop method, or by calling the *set_name()* method on the task object	
Added support for Happy Eyeballs to *asyncio.loop.create_connection()*. To specify the behavior, two new parameters have been added: *happy_eyeballs_delay* and interleave.	
gc	
get_objects() can now receive an optional generation parameter indicating a generation to get objects from. (Note, though, that while *gc* is a built-in, *get_objects()* is not implemented for MicroPython)	
math	
Added new function *math.dist()* for computing Euclidean distance between two points	
Expanded the *math.hypot()* function to handle multiple dimensions	
Added new function, *math.prod()*, as analogous function to *sum()* that returns the product of a start value (default: 1) times an iterable of numbers	
Added two new combinatoric functions *math.perm()* and *math.comb()*	
Added a new function *math.isqrt()* for computing accurate integer square roots without conversion to floating point	
The function *math.factorial()* no longer accepts arguments that are not int-like	Completed
sys	
Add new *sys.unraisablehook()* function which can be overridden to control how unraisable exceptions are handled	

3.5 Python 3.9

Python 3.9.0 (final) was released on the 5th October 2020. The Features for 3.9 are defined in PEP 596 and a detailed description of the changes can be found in Whats New in Python 3.9

Features:		**Status**
PEP 573	fast access to module state from methods of C extension types	
PEP 584	union operators added to dict	
PEP 585	type hinting generics in standard collections	
PEP 593	flexible function and variable annotations	
PEP 602	CPython adopts an annual release cycle. Instead of annual, aiming for two month release cycle	
PEP 614	relaxed grammar restrictions on decorators	
PEP 615	the IANA Time Zone Database is now present in the standard library in the zoneinfo module	
PEP 616	string methods to remove prefixes and suffixes	
PEP 617	CPython now uses a new parser based on PEG	

Other Language Changes:

__import__() now raises *ImportError* instead of *ValueError*	Completed
Python now gets the absolute path of the script filename specified on the command line (ex: *python3 script.py*): the *__file__* attribute of the *__main__* module became an absolute path, rather than a relative path	
By default, for best performance, the errors argument is only checked at the first encoding/decoding error and the encoding argument is sometimes ignored for empty strings	
.replace(, s, n) now returns *s* instead of an empty string for all non-zero n. It is now consistent with *.replace(, s)*	
Any valid expression can now be used as a decorator. Previously, the grammar was much more restrictive	
Parallel running of *aclose() / asend() / athrow()* is now prohibited, and *ag_running* now reflects the actual running status of the async generator	
Unexpected errors in calling the *__iter__* method are no longer masked by TypeError in the in operator and functions contains(), indexOf() and countOf() of the operator module	
Unparenthesized lambda expressions can no longer be the expression part in an if clause in comprehensions and generator expressions	

Changes to built-in modules:

asyncio	
Due to significant security concerns, the reuse_address parameter of *asyncio.loop.create_datagram_endpoint()* is no longer supported	
Added a new coroutine *shutdown_default_executor()* that schedules a shutdown for the default executor that waits on the *ThreadPoolExecutor* to finish closing. Also, *asyncio.run()* has been updated to use the new coroutine.	
Added *asyncio.PidfdChildWatcher*, a Linux-specific child watcher implementation that polls process file descriptors	
added a new *coroutine asyncio.to_thread()*	
When cancelling the task due to a timeout, *asyncio.wait_for()* will now wait until the cancellation is complete also in the case when timeout is <= 0, like it does with positive timeouts	
asyncio now raises *TyperError* when calling incompatible methods with an *ssl.SSLSocket* socket	
gc	
Garbage collection does not block on resurrected objects	
Added a new function *gc.is_finalized()* to check if an object has been finalized by the garbage collector	
math	
Expanded the *math.gcd()* function to handle multiple arguments. Formerly, it only supported two arguments	
Added *math.lcm()*: return the least common multiple of specified arguments	
Added *math.nextafter()*: return the next floating-point value after x towards y	
Added *math.ulp()*: return the value of the least significant bit of a float	
os	
Exposed the Linux-specific *os.pidfd_open()* and *os.P_PIDFD*	
The *os.unsetenv()* function is now also available on Windows	Completed
The *os.putenv()* and *os.unsetenv()* functions are now always available	Completed
Added *os.waitstatus_to_exitcode()* function: convert a wait status to an exit code	
random	
Added a new *random.Random.randbytes* method: generate random bytes	
sys	
Added a new *sys.platlibdir* attribute: name of the platform-specific library directory	
Previously, *sys.stderr* was block-buffered when non-interactive. Now stderr defaults to always being line-buffered	

For the features of Python that are implemented by MicroPython, there are sometimes differences in their behaviour compared to standard Python. The operations listed in the sections below produce conflicting results in MicroPython when compared to standard Python.

3.6 Syntax

Generated Fri 07 Jan 2022 12:14:15 UTC

3.6.1 Operators

MicroPython allows using := to assign to the variable of a comprehension, CPython raises a SyntaxError.

Cause: MicroPython is optimised for code size and doesnt check this case.

Workaround: Do not rely on this behaviour if writing CPython compatible code.

Sample code:

```
print([i := -1 for i in range(4)])
```

CPy output:	uPy output:
File "<stdin>", line 7 SyntaxError: assignment expression cannot rebind comprehension iteration variable 'i'	[-1, -1, -1, -1]

3.6.2 Spaces

uPy requires spaces between literal numbers and keywords, CPy doesnt

Sample code:

```
try:
    print(eval("1and 0"))
except SyntaxError:
    print("Should have worked")
try:
    print(eval("1or 0"))
except SyntaxError:
    print("Should have worked")
try:
    print(eval("1if 1else 0"))
except SyntaxError:
    print("Should have worked")
```

CPy output:	uPy output:
0 1 1	Should have worked Should have worked Should have worked

3.6.3 Unicode

Unicode name escapes are not implemented

Sample code:

```
print("\N{LATIN SMALL LETTER A}")
```

CPy output:	uPy output:
a	NotImplementedError: unicode name escapes

3.7 Core language

Generated Fri 07 Jan 2022 12:14:15 UTC

3.7.1 f-strings dont support concatenation with adjacent literals if the adjacent literals contain braces or are f-strings

Cause: MicroPython is optimised for code space.

Workaround: Use the + operator between literal strings when either or both are f-strings

Sample code:

```
x, y = 1, 2
print("aa" f"{x}")  # works
print(f"{x}" "ab")  # works
print("a{}a" f"{x}")  # fails
print(f"{x}" "a{}b")  # fails
print(f"{x}" f"{y}")  # fails
```

CPy output:	uPy output:
aa1 1ab a{}a1 1a{}b 12	Traceback (most recent call last): File "<stdin>", line 13 SyntaxError: invalid syntax

3.7.2 f-strings cannot support expressions that require parsing to resolve unbalanced nested braces and brackets

Cause: MicroPython is optimised for code space.

Workaround: Always use balanced braces and brackets in expressions inside f-strings

Sample code:

```
print(f'{"hello { world"}')
print(f'{"hello ] world"}')
```

CPy output:	uPy output:
hello { world hello] world	Traceback (most recent call last): File "<stdin>", line 9 SyntaxError: invalid syntax

3.7.3 Raw f-strings are not supported

Cause: MicroPython is optimised for code space.

Sample code:

```
rf"hello"
```

CPy output:	uPy output:
	Traceback (most recent call last): File "<stdin>", line 8 SyntaxError: raw f-strings are not␣ ↪supported

3.7.4 f-strings dont support the !r, !s, and !a conversions

Cause: MicroPython is optimised for code space.

Workaround: Use repr(), str(), and ascii() explictly.

Sample code:

```
class X:
    def __repr__(self):
        return "repr"

    def __str__(self):
        return "str"


print(f"{X()!r}")
print(f"{X()!s}")
```

CPy output:	uPy output:
repr str	Traceback (most recent call last): File "<stdin>", line 17 SyntaxError: invalid syntax

Classes

3.7.5 Special method __del__ not implemented for user-defined classes

Sample code:

```
import gc


class Foo:
    def __del__(self):
        print("__del__")


f = Foo()
del f

gc.collect()
```

CPy output:	uPy output:
__del__	

3.7.6 Method Resolution Order (MRO) is not compliant with CPython

Cause: Depth first non-exhaustive method resolution order

Workaround: Avoid complex class hierarchies with multiple inheritance and complex method overrides. Keep in mind that many languages dont support multiple inheritance at all.

Sample code:

```
class Foo:
    def __str__(self):
        return "Foo"


class C(tuple, Foo):
    pass


t = C((1, 2, 3))
print(t)
```

CPy output:	uPy output:
Foo	(1, 2, 3)

3.7.7 When inheriting from multiple classes super() only calls one class

Cause: See *Method Resolution Order (MRO) is not compliant with CPython*

Workaround: See *Method Resolution Order (MRO) is not compliant with CPython*

Sample code:

```
class A:
    def __init__(self):
        print("A.__init__")


class B(A):
    def __init__(self):
        print("B.__init__")
        super().__init__()


class C(A):
    def __init__(self):
        print("C.__init__")
        super().__init__()


class D(B, C):
    def __init__(self):
        print("D.__init__")
        super().__init__()


D()
```

CPy output:	uPy output:
D.__init__ B.__init__ C.__init__ A.__init__	D.__init__ B.__init__ A.__init__

3.7.8 Calling super() getter property in subclass will return a property object, not the value

Sample code:

```
class A:
    @property
    def p(self):
        return {"a": 10}


class AA(A):
    @property
    def p(self):
        return super().p


a = AA()
print(a.p)
```

CPy output:	uPy output:
`{'a': 10}`	`<property>`

Functions

3.7.9 Error messages for methods may display unexpected argument counts

Cause: MicroPython counts self as an argument.

Workaround: Interpret error messages with the information above in mind.

Sample code:

```
try:
    [].append()
except Exception as e:
    print(e)
```

CPy output:	uPy output:
`list.append() takes exactly one argument␣ ↪(0 given)`	`function takes 2 positional arguments but␣ ↪1 were given`

3.7.10 Function objects do not have the `__module__` attribute

Cause: MicroPython is optimized for reduced code size and RAM usage.

Workaround: Use `sys.modules[function.__globals__['__name__']]` for non-builtin modules.

Sample code:

```
def f():
    pass


print(f.__module__)
```

CPy output:	uPy output:

`__main__`	`Traceback (most recent call last):`
	`  File "<stdin>", line 13, in <module>`
	`AttributeError: 'function' object has no attribute '__module__'`

3.7.11 User-defined attributes for functions are not supported

Cause: MicroPython is highly optimized for memory usage.

Workaround: Use external dictionary, e.g. `FUNC_X[f] = 0`.

Sample code:

```
def f():
    pass


f.x = 0
print(f.x)
```

CPy output:	uPy output:

`0`	`Traceback (most recent call last):`
	`  File "<stdin>", line 13, in <module>`
	`AttributeError: 'function' object has no attribute 'x'`

Generator

3.7.12 Context manager __exit__() not called in a generator which does not run to completion

Sample code:

```
class foo(object):
    def __enter__(self):
        print("Enter")

    def __exit__(self, *args):
        print("Exit")


def bar(x):
    with foo():
        while True:
            x += 1
            yield x


def func():
    g = bar(0)
    for _ in range(3):
        print(next(g))


func()
```

CPy output:	uPy output:
Enter 1 2 3 Exit	Enter 1 2 3

Runtime

3.7.13 Local variables arent included in locals() result

Cause: MicroPython doesnt maintain symbolic local environment, it is optimized to an array of slots. Thus, local variables cant be accessed by a name.

Sample code:

```
def test():
    val = 2
    print(locals())
```

(continues on next page)

(continued from previous page)

```
test()
```

CPy output:	uPy output:
`{'val': 2}`	`{'test': <function test at 0x7f9bfee94100> ↪, '__name__': '__main__', '__file__': ' ↪<stdin>'}`

3.7.14 Code running in eval() function doesnt have access to local variables

Cause: MicroPython doesnt maintain symbolic local environment, it is optimized to an array of slots. Thus, local variables cant be accessed by a name. Effectively, `eval(expr)` in MicroPython is equivalent to `eval(expr, globals(), globals())`.

Sample code:

```
val = 1


def test():
    val = 2
    print(val)
    eval("print(val)")


test()
```

CPy output:	uPy output:
`2` `2`	`2` `1`

import

3.7.15 __all__ is unsupported in __init__.py in MicroPython.

Cause: Not implemented.

Workaround: Manually import the sub-modules directly in __init__.py using `from . import foo, bar`.

Sample code:

```
from modules3 import *

foo.hello()
```

CPy output:	uPy output:

`hello`	`Traceback (most recent call last):`
	`  File "<stdin>", line 9, in <module>`
	`NameError: name 'foo' isn't defined`

3.7.16 __path__ attribute of a package has a different type (single string instead of list of strings) in MicroPython

Cause: MicroPython doest support namespace packages split across filesystem. Beyond that, MicroPythons import system is highly optimized for minimal memory usage.

Workaround: Details of import handling is inherently implementation dependent. Dont rely on such details in portable applications.

Sample code:

```
import modules

print(modules.__path__)
```

CPy output:	uPy output:
`['/home/micropython/micropython-autodocs/` `↪tests/cpydiff/modules']`	`../tests/cpydiff//modules`

3.7.17 Failed to load modules are still registered as loaded

Cause: To make module handling more efficient, its not wrapped with exception handling.

Workaround: Test modules before production use; during development, use `del sys.modules["name"]`, or just soft or hard reset the board.

Sample code:

```
import sys

try:
    from modules import foo
except NameError as e:
    print(e)
try:
    from modules import foo

    print("Should not get here")
except NameError as e:
    print(e)
```

CPy output:	uPy output:
foo name 'xxx' is not defined foo name 'xxx' is not defined	foo name 'xxx' isn't defined Should not get here

3.7.18 MicroPython doest support namespace packages split across filesystem.

Cause: MicroPythons import system is highly optimized for simplicity, minimal memory usage, and minimal filesystem search overhead.

Workaround: Dont install modules belonging to the same namespace package in different directories. For MicroPython, its recommended to have at most 3-component module search paths: for your current application, per-user (writable), system-wide (non-writable).

Sample code:

```
import sys

sys.path.append(sys.path[1] + "/modules")
sys.path.append(sys.path[1] + "/modules2")

import subpkg.foo
import subpkg.bar

print("Two modules of a split namespace package imported")
```

CPy output:	uPy output:
Two modules of a split namespace package ↪imported	Traceback (most recent call last): File "<stdin>", line 13, in <module> ImportError: no module named 'subpkg.bar'

3.8 Builtin types

Generated Fri 07 Jan 2022 12:14:15 UTC

3.8.1 Exception

All exceptions have readable `value` and `errno` attributes, not just `StopIteration` and `OSError`.

Cause: MicroPython is optimised to reduce code size.

Workaround: Only use `value` on `StopIteration` exceptions, and `errno` on `OSError` exceptions. Do not use or rely on these attributes on other exceptions.

Sample code:

```
e = Exception(1)
print(e.value)
print(e.errno)
```

CPy output:	uPy output:

`Traceback (most recent call last):`
`  File "<stdin>", line 8, in <module>`
`AttributeError: 'Exception' object has no attribute 'value'`
	`1`
	`1`

Exception chaining not implemented

Sample code:

```
try:
    raise TypeError
except TypeError:
    raise ValueError
```

CPy output:	uPy output:

`Traceback (most recent call last):`
`  File "<stdin>", line 8, in <module>`
`TypeError`
 	
`During handling of the above exception, another exception occurred:`
 	
`Traceback (most recent call last):`
`  File "<stdin>", line 10, in <module>`
`ValueError`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 10, in <module>`
	`ValueError:`

User-defined attributes for builtin exceptions are not supported

Cause: MicroPython is highly optimized for memory usage.

Workaround: Use user-defined exception subclasses.

Sample code:

```
e = Exception()
e.x = 0
print(e.x)
```

CPy output:	uPy output:
0	Traceback (most recent call last): File "<stdin>", line 8, in <module> AttributeError: 'Exception' object has no attribute 'x'

Exception in while loop condition may have unexpected line number

Cause: Condition checks are optimized to happen at the end of loop body, and that line number is reported.

Sample code:

```
l = ["-foo", "-bar"]

i = 0
while l[i][0] == "-":
    print("iter")
    i += 1
```

CPy output:	uPy output:

iter
iter
Traceback (most recent call last):
  File "<stdin>", line 10, in <module>
IndexError: list index out of range
	iter
	iter
	Traceback (most recent call last):
	  File "<stdin>", line 12, in <module>
	IndexError: list index out of range

Exception.__init__ method does not exist.

Cause: Subclassing native classes is not fully supported in MicroPython.

Workaround: Call using `super()` instead:

```
class A(Exception):
    def __init__(self):
        super().__init__()
```

Sample code:

```
class A(Exception):
    def __init__(self):
        Exception.__init__(self)


a = A()
```

CPy output:	uPy output:

	Traceback (most recent call last):
	  File "<stdin>", line 18, in <module>
	  File "<stdin>", line 15, in __init__
	AttributeError: type object 'Exception' has no attribute '__init__'

3.8.2 bytearray

Array slice assignment with unsupported RHS

Sample code:

```
b = bytearray(4)
b[0:1] = [1, 2]
print(b)
```

CPy output:	uPy output:
bytearray(b'\x01\x02\x00\x00\x00')	Traceback (most recent call last): File "<stdin>", line 8, in <module> NotImplementedError: array/bytes required on right side

3.8.3 bytes

bytes objects support .format() method

Cause: MicroPython strives to be a more regular implementation, so if both *str* and *bytes* support `__mod__()` (the % operator), it makes sense to support `format()` for both too. Support for `__mod__` can also be compiled out, which leaves only `format()` for bytes formatting.

Workaround: If you are interested in CPython compatibility, dont use `.format()` on bytes objects.

Sample code:

```
print(b"{}".format(1))
```

CPy output:	uPy output:
Traceback (most recent call last): File "<stdin>", line 7, in <module> AttributeError: 'bytes' object has no attribute 'format'	b'1'

bytes() with keywords not implemented

Workaround: Pass the encoding as a positional parameter, e.g. `print(bytes('abc', 'utf-8'))`

Sample code:

```
print(bytes("abc", encoding="utf8"))
```

CPy output:	uPy output:

`b'abc'`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: keyword argument(s) not yet implemented - use normal args instead`

Bytes subscription with step != 1 not implemented

Cause: MicroPython is highly optimized for memory usage.

Workaround: Use explicit loop for this very rare operation.

Sample code:

```
print(b"123"[0:3:2])
```

CPy output:	uPy output:

`b'13'`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: only slices with step=1 (aka None) are supported`

3.8.4 dict

Dictionary keys view does not behave as a set.

Cause: Not implemented.

Workaround: Explicitly convert keys to a set before using set operations.

Sample code:

```
print({1: 2, 3: 4}.keys() & {1})
```

CPy output:	uPy output:
{1}	Traceback (most recent call last): File "<stdin>", line 7, in <module> TypeError: unsupported types for __and__: ↪'dict_view', 'set'

3.8.5 float

uPy and CPython outputs formats may differ

Sample code:

```
print("%.1g" % -9.9)
```

CPy output:	uPy output:
-1e+01	-10

3.8.6 int

bit_length method doesnt exist.

Cause: bit_length method is not implemented.

Workaround: Avoid using this method on MicroPython.

Sample code:

```
x = 255
print("{} is {} bits long.".format(x, x.bit_length()))
```

CPy output:	uPy output:
255 is 8 bits long.	Traceback (most recent call last): File "<stdin>", line 9, in <module> AttributeError: 'int' object has no␣ ↪attribute 'bit_length'

No int conversion for int-derived types available

Workaround: Avoid subclassing builtin types unless really needed. Prefer https://en.wikipedia.org/wiki/Composition_over_inheritance .

Sample code:

```
class A(int):
    __add__ = lambda self, other: A(int(self) + other)


a = A(42)
print(a + a)
```

CPy output:	uPy output:

84	Traceback (most recent call last):
	  File "<stdin>", line 14, in <module>
	  File "<stdin>", line 10, in <lambda>
	TypeError: unsupported types for __radd__
	↪: 'int', 'int'

3.8.7 list

List delete with step != 1 not implemented

Workaround: Use explicit loop for this rare operation.

Sample code:

```
l = [1, 2, 3, 4]
del l[0:4:2]
print(l)
```

CPy output:	uPy output:
[2, 4]	Traceback (most recent call last): File "<stdin>", line 8, in <module> NotImplementedError:

List slice-store with non-iterable on RHS is not implemented

Cause: RHS is restricted to be a tuple or list

Workaround: Use `list(<iter>)` on RHS to convert the iterable to a list

Sample code:

```
l = [10, 20]
l[0:1] = range(4)
print(l)
```

CPy output:	uPy output:

`[0, 1, 2, 3, 20]`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`TypeError: object 'range' isn't a tuple or list`

List store with step != 1 not implemented

Workaround: Use explicit loop for this rare operation.

Sample code:

```
l = [1, 2, 3, 4]
l[0:4:2] = [5, 6]
print(l)
```

CPy output:	uPy output:

`[5, 2, 6, 4]`	`Traceback (most recent call last):`
	`  File "<stdin>", line 8, in <module>`
	`NotImplementedError:`

3.8.8 str

Start/end indices such as str.endswith(s, start) not implemented

Sample code:

```
print("abc".endswith("c", 1))
```

CPy output:	uPy output:

`True`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: start/end indices`

Attributes/subscr not implemented

Sample code:

```
print("{a[0]}".format(a=[1, 2]))
```

CPy output:	uPy output:
1	Traceback (most recent call last): File "<stdin>", line 7, in <module> NotImplementedError: attributes not supported yet

str() with keywords not implemented

Workaround: Input the encoding format directly. eg `print(bytes('abc', 'utf-8'))`

Sample code:

```
print(str(b"abc", encoding="utf8"))
```

CPy output:	uPy output:
abc	Traceback (most recent call last): File "<stdin>", line 7, in <module> NotImplementedError: keyword argument(s) not yet implemented - use normal args instead

str.ljust() and str.rjust() not implemented

Cause: MicroPython is highly optimized for memory usage. Easy workarounds available.

Workaround: Instead of `s.ljust(10)` use `"%-10s" % s`, instead of `s.rjust(10)` use `"% 10s" % s`. Alternatively, `"{:<10}".format(s)` or `"{:>10}".format(s)`.

Sample code:

```
print("abc".ljust(10))
```

CPy output:	uPy output:
abc	Traceback (most recent call last): File "<stdin>", line 7, in <module> AttributeError: 'str' object has no attribute 'ljust'

None as first argument for rsplit such as str.rsplit(None, n) not implemented

Sample code:

```
print("a a a".rsplit(None, 1))
```

CPy output:	uPy output:

`['a a', 'a']`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: rsplit(None,n)`

Subscript with step != 1 is not yet implemented

Sample code:

```
print("abcdefghi"[0:9:2])
```

CPy output:	uPy output:

`acegi`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: only slices with step=1 (aka None) are supported`

3.8.9 tuple

Tuple load with step != 1 not implemented

Sample code:

```
print((1, 2, 3, 4)[0:4:2])
```

CPy output:	uPy output:

`(1, 3)`	`Traceback (most recent call last):`
	`  File "<stdin>", line 7, in <module>`
	`NotImplementedError: only slices with step=1 (aka None) are supported`

3.9 Modules

Generated Fri 07 Jan 2022 12:14:15 UTC

3.9.1 array

Comparison between different typecodes not supported

Cause: Code size

Workaround: Compare individual elements

Sample code:

```
import array

array.array("b", [1, 2]) == array.array("i", [1, 2])
```

CPy output:	uPy output:
	Traceback (most recent call last): File "<stdin>", line 9, in <module> NotImplementedError:

Overflow checking is not implemented

Cause: MicroPython implements implicit truncation in order to reduce code size and execution time

Workaround: If CPython compatibility is needed then mask the value explicitly

Sample code:

```
import array

a = array.array("b", [257])
print(a)
```

CPy output:	uPy output:
Traceback (most recent call last): File "<stdin>", line 9, in <module> OverflowError: signed char is greater␣ ↪than maximum	array('b', [1])

Looking for integer not implemented

Sample code:

```
import array

print(1 in array.array("B", b"12"))
```

CPy output:	uPy output:

`False`	`Traceback (most recent call last):`
	`  File "<stdin>", line 9, in <module>`
	`NotImplementedError:`

Array deletion not implemented

Sample code:

```
import array

a = array.array("b", (1, 2, 3))
del a[1]
print(a)
```

CPy output:	uPy output:

`array('b', [1, 3])`	`Traceback (most recent call last):`
	`  File "<stdin>", line 10, in <module>`
	`TypeError: 'array' object doesn't support item deletion`

Subscript with step != 1 is not yet implemented

Sample code:

```
import array

a = array.array("b", (1, 2, 3))
print(a[3:2:2])
```

CPy output:	uPy output:

`array('b')`	`Traceback (most recent call last):`
	`  File "<stdin>", line 10, in <module>`
	`NotImplementedError: only slices with step=1 (aka None) are supported`

3.9.2 builtins

Second argument to next() is not implemented

Cause: MicroPython is optimised for code space.

Workaround: Instead of `val = next(it, deflt)` use:

```
try:
    val = next(it)
except StopIteration:
    val = deflt
```

Sample code:

```
print(next(iter(range(0)), 42))
```

CPy output:	uPy output:
42	Traceback (most recent call last): File "<stdin>", line 12, in <module> TypeError: function takes 1 positional␣ ↪arguments but 2 were given

3.9.3 deque

Deque not implemented

Workaround: Use regular lists. micropython-lib has implementation of collections.deque.

Sample code:

```
import collections

D = collections.deque()
print(D)
```

CPy output:	uPy output:
deque([])	Traceback (most recent call last): File "<stdin>", line 9, in <module> TypeError: function missing 2 required␣ ↪positional arguments

3.9.4 json

JSON module does not throw exception when object is not serialisable

Sample code:

```
import json

a = bytes(x for x in range(256))
try:
    z = json.dumps(a)
    x = json.loads(z)
    print("Should not get here")
except TypeError:
    print("TypeError")
```

CPy output:	uPy output:
`TypeError`	`Should not get here`

3.9.5 os

`environ` attribute is not implemented

Workaround: Use `getenv`, `putenv` and `unsetenv`

Sample code:

```
import os

try:
    print(os.environ.get("NEW_VARIABLE"))
    os.environ["NEW_VARIABLE"] = "VALUE"
    print(os.environ["NEW_VARIABLE"])
except AttributeError:
    print("should not get here")
    print(os.getenv("NEW_VARIABLE"))
    os.putenv("NEW_VARIABLE", "VALUE")
    print(os.getenv("NEW_VARIABLE"))
```

CPy output:	uPy output:

`None`
`VALUE`
	`should not get here`
	`None`
	`VALUE`

`getenv` returns actual value instead of cached value

Cause: The `environ` attribute is not implemented

Sample code:

```
import os

print(os.getenv("NEW_VARIABLE"))
os.putenv("NEW_VARIABLE", "VALUE")
print(os.getenv("NEW_VARIABLE"))
```

CPy output:	uPy output:
`None` `None`	`None` `VALUE`

`getenv` only allows one argument

Workaround: Test that the return value is `None`

Sample code:

```
import os

try:
    print(os.getenv("NEW_VARIABLE", "DEFAULT"))
except TypeError:
    print("should not get here")
    # this assumes NEW_VARIABLE is never an empty variable
    print(os.getenv("NEW_VARIABLE") or "DEFAULT")
```

CPy output:	uPy output:
`DEFAULT`	`should not get here` `DEFAULT`

3.9.6 random

`getrandbits` method can only return a maximum of 32 bits at a time.

Cause: PRNGs internal state is only 32bits so it can only return a maximum of 32 bits of data at a time.

Workaround: If you need a number that has more than 32 bits then utilize the random module from micropython-lib.

Sample code:

```
import random
```

(continues on next page)

(continued from previous page)

```
x = random.getrandbits(64)
print("{}".format(x))
```

CPy output:	uPy output:
9534879574767561583	Traceback (most recent call last): File "<stdin>", line 11, in <module> ValueError: bits must be 32 or less

`randint` method can only return an integer that is at most the native word size.

Cause: PRNG is only able to generate 32 bits of state at a time. The result is then cast into a native sized int instead of a full int object.

Workaround: If you need integers larger than native wordsize use the random module from micropython-lib.

Sample code:

```
import random


x = random.randint(2 ** 128 - 1, 2 ** 128)
print("x={}".format(x))
```

CPy output:	uPy output:
x=340282366920938463463374607431768211455	Traceback (most recent call last): File "<stdin>", line 11, in <module> AttributeError: 'module' object has no␣ ↪attribute 'randint'

3.9.7 struct

Struct pack with too few args, not checked by uPy

Sample code:

```
import struct

try:
    print(struct.pack("bb", 1))
    print("Should not get here")
except:
    print("struct.error")
```

CPy output:	uPy output:
`struct.error`	`b'\x01\x00'` `Should not get here`

Struct pack with too many args, not checked by uPy

Sample code:

```
import struct

try:
    print(struct.pack("bb", 1, 2, 3))
    print("Should not get here")
except:
    print("struct.error")
```

CPy output:	uPy output:
`struct.error`	`b'\x01\x02'` `Should not get here`

Struct pack with whitespace in format, whitespace ignored by CPython, error on uPy

Cause: MicroPython is optimised for code size.

Workaround: Dont use spaces in format strings.

Sample code:

```
import struct

try:
    print(struct.pack("b b", 1, 2))
    print("Should have worked")
except:
    print("struct.error")
```

CPy output:	uPy output:
`b'\x01\x02'` `Should have worked`	`struct.error`

3.9.8 sys

Overriding sys.stdin, sys.stdout and sys.stderr not possible

Cause: They are stored in read-only memory.

Sample code:

```
import sys

sys.stdin = None
print(sys.stdin)
```

CPy output:	uPy output:

`None`	`Traceback (most recent call last):`
	`  File "<stdin>", line 9, in <module>`
	`AttributeError: 'module' object has no attribute 'stdin'`

CHAPTER

FOUR

MICROPYTHON INTERNALS

This chapter covers a tour of MicroPython from the perspective of a developer, contributing to MicroPython. It acts as a comprehensive resource on the implementation details of MicroPython for both novice and expert contributors.

Development around MicroPython usually involves modifying the core runtime, porting or maintaining a new library. This guide describes at great depth, the implementation details of MicroPython including a getting started guide, compiler internals, porting MicroPython to a new platform and implementing a core MicroPython library.

4.1 Getting Started

This guide covers a step-by-step process on setting up version control, obtaining and building a copy of the source code for a port, building the documentation, running tests, and a description of the directory structure of the MicroPython code base.

4.1.1 Source control with git

MicroPython is hosted on GitHub and uses Git for source control. The workflow is such that code is pulled and pushed to and from the main repository. Install the respective version of Git for your operating system to follow through the rest of the steps.

Note: For a reference on the installation instructions, please refer to the Git installation instructions. Learn about the basic git commands in this Git Handbook or any other sources on the internet.

Note: A .git-blame-ignore-revs file is included which avoids the output of git blame getting cluttered by commits which are only for formatting code but have no functional changes. See git blame documentation on how to use this.

4.1.2 Get the code

It is recommended that you maintain a fork of the MicroPython repository for your development purposes. The process of obtaining the source code includes the following:

1. Fork the repository https://github.com/micropython/micropython
2. You will now have a fork at <https://github.com/<your-user-name>/micropython>.
3. Clone the forked repository using the following command:

```
$ git clone https://github.com/<your-user-name>/micropython
```

Then, configure the remote repositories to be able to collaborate on the MicroPython project.

Configure remote upstream:

```
$ cd micropython
$ git remote add upstream https://github.com/micropython/micropython
```

It is common to configure `upstream` and `origin` on a forked repository to assist with sharing code changes. You can maintain your own mapping but it is recommended that `origin` maps to your fork and `upstream` to the main MicroPython repository.

After the above configuration, your setup should be similar to this:

```
$ git remote -v
origin      https://github.com/<your-user-name>/micropython (fetch)
origin      https://github.com/<your-user-name>/micropython (push)
upstream    https://github.com/micropython/micropython (fetch)
upstream    https://github.com/micropython/micropython (push)
```

You should now have a copy of the source code. By default, you are pointing to the master branch. To prepare for further development, it is recommended to work on a development branch.

```
$ git checkout -b dev-branch
```

You can give it any name. You will have to compile MicroPython whenever you change to a different branch.

4.1.3 Compile and build the code

When compiling MicroPython, you compile a specific *port*, usually targeting a specific board. Start by installing the required dependencies. Then build the MicroPython cross-compiler before you can successfully compile and build. This applies specifically when using Linux to compile. The Windows instructions are provided in a later section.

Required dependencies

Install the required dependencies for Linux:

```
$ sudo apt-get install build-essential libffi-dev git pkg-config
```

For the stm32 port, the ARM cross-compiler is required:

```
$ sudo apt-get install arm-none-eabi-gcc arm-none-eabi-binutils arm-none-eabi-newlib
```

See the ARM GCC toolchain for the latest details.

Python is also required. Python 2 is supported for now, but we recommend using Python 3. Check that you have Python available on your system:

```
$ python3
Python 3.5.0 (default, Jul 17 2020, 14:04:10)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

All supported ports have different dependency requirements, see their respective readme files.

Building the MicroPython cross-compiler

Almost all ports require building `mpy-cross` first to perform pre-compilation of Python code that will be included in the port firmware:

```
$ cd mpy-cross
$ make
```

Note: Note that, `mpy-cross` must be built for the host architecture and not the target architecture.

If it built successfully, you should see a message similar to this:

```
LINK mpy-cross
   text          data     bss     dec     hex filename
 279328          776     880  280984   44998 mpy-cross
```

Note: Use `make -C mpy-cross` to build the cross-compiler in one statement without moving to the `mpy-cross` directory otherwise, you will need to do `cd ..` for the next steps.

Building the Unix port of MicroPython

The Unix port is a version of MicroPython that runs on Linux, macOS, and other Unix-like operating systems. Its extremely useful for developing MicroPython as it avoids having to deploy your code to a device to test it. In many ways, it works a lot like CPythons python binary.

To build for the Unix port, make sure all Linux related dependencies are installed as detailed in the required dependencies section. See the *Required dependencies* to make sure that all dependencies are installed for this port. Also, make sure you have a working environment for `gcc` and `GNU make`. Ubuntu 20.04 has been used for the example below but other unixes ought to work with little modification:

```
$ gcc --version
gcc (Ubuntu 9.3.0-10ubuntu2) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.then build:
```

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

If MicroPython built correctly, you should see the following:

```
LINK micropython
   text          data     bss     dec     hex filename
 412033          5680    2496  420209   66971 micropython
```

Now run it:

```
$ ./micropython
MicroPython v1.13-38-gc67012d-dirty on 2020-09-13; linux version
Use Ctrl-D to exit, Ctrl-E for paste mode
>>> print("hello world")
hello world
>>>
```

Building the Windows port

The Windows port includes a Visual Studio project file micropython.vcxproj that you can use to build micropython.exe. It can be opened in Visual Studio or built from the command line using msbuild. Alternatively, it can be built using mingw, either in Windows with Cygwin, or on Linux. See windows port documentation for more information.

Building the STM32 port

Like the Unix port, you need to install some required dependencies as detailed in the *Required dependencies* section, then build:

```
$ cd ports/stm32
$ make submodules
$ make
```

Please refer to the stm32 documentation for more details on flashing the firmware.

Note: See the *Required dependencies* to make sure that all dependencies are installed for this port. The cross-compiler is needed. `arm-none-eabi-gcc` should also be in the $PATH or specified manually via CROSS_COMPILE, either by setting the environment variable or in the `make` command line arguments.

You can also specify which board to use:

```
$ cd ports/stm32
$ make submodules
$ make BOARD=<board>
```

See ports/stm32/boards for the available boards. e.g. PYBV11 or NUCLEO_WB55.

4.1.4 Building the documentation

MicroPython documentation is created using `Sphinx`. If you have already installed Python, then install `Sphinx` using `pip`. It is recommended that you use a virtual environment:

```
$ python3 -m venv env
$ source env/bin/activate
$ pip install sphinx
```

Navigate to the `docs` directory:

```
$ cd docs
```

Build the docs:

```
$ make html
```

Open `docs/build/html/index.html` in your browser to view the docs locally. Refer to the documentation on importing your documentation to use Read the Docs.

4.1.5 Running the tests

To run all tests in the test suite on the Unix port use:

```
$ cd ports/unix
$ make test
```

To run a selection of tests on a board/device connected over USB use:

```
$ cd tests
$ ./run-tests.py --target minimal --device /dev/ttyACM0
```

See also *Writing tests*.

4.1.6 Folder structure

There are a couple of directories to take note of in terms of where certain implementation details are. The following is a break down of the top-level folders in the source code.

py

: Contains the compiler, runtime, and core library implementation.

mpy-cross

: Has the MicroPython cross-compiler which pre-compiles the Python scripts to bytecode.

ports

: Code for all the versions of MicroPython for the supported ports.

lib

: Low-level C libraries used by any port which are mostly 3rd-party libraries.

drivers

: Has drivers for specific hardware and intended to work across multiple ports.

extmod

: Contains a C implementation of more non-core modules.

docs

: Has the standard documentation found at https://docs.micropython.org/.

tests

: An implementation of the test suite.

tools

: Contains helper tools including the `upip` and the `pyboard.py` module.

examples

Example code for building MicroPython as a library as well as native modules.

4.2 Writing tests

Tests in MicroPython are located at the path `tests/`. The following is a listing of key directories and the run-tests.py runner script:

```
.
  basics
  extmod
  float
  micropython
  run-tests.py
 ...
```

There are subfolders maintained to categorize the tests. Add a test by creating a new file in one of the existing folders or in a new folder. Its also possible to make custom tests outside this tests folder, which would be recommended for a custom port.

For example, add the following code in a file `print.py` in the `tests/unix/` subdirectory:

```
def print_one():
    print(1)

print_one()
```

If you run your tests, this test should appear in the test output:

```
$ cd ports/unix
$ make tests
skip  unix/extra_coverage.py
pass  unix/ffi_callback.py
pass  unix/ffi_float.py
pass  unix/ffi_float2.py
pass  unix/print.py
pass  unix/time.py
pass  unix/time2.py
```

Tests are run by comparing the output from the test target against the output from CPython. So any test should use print statements to indicate test results.

For tests that cant be compared to CPython (i.e. micropython-specific functionality), you can provide a `.py.exp` file which will be used as the truth for comparison.

The other way to run tests, which is useful when running on targets other than the Unix port, is:

```
$ cd tests
$ ./run-tests.py
```

Then to run on a board:

```
$ ./run-tests.py --target minimal --device /dev/ttyACM0
```

And to run only a certain set of tests (eg a directory):

```
$ ./run-tests.py -d basics
$ ./run-tests.py float/builtin*.py
```

4.3 The Compiler

The compilation process in MicroPython involves the following steps:

- The lexer converts the stream of text that makes up a MicroPython program into tokens.
- The parser then converts the tokens into an abstract syntax (parse tree).
- Then bytecode or native code is emitted based on the parse tree.

For purposes of this discussion we are going to add a simple language feature `add1` that can be use in Python as:

```
>>> add1 3
4
>>>
```

The `add1` statement takes an integer as argument and adds 1 to it.

4.3.1 Adding a grammar rule

MicroPythons grammar is based on the CPython grammar and is defined in py/grammar.h. This grammar is what is used to parse MicroPython source files.

There are two macros you need to know to define a grammar rule: `DEF_RULE` and `DEF_RULE_NC`. `DEF_RULE` allows you to define a rule with an associated compile function, while `DEF_RULE_NC` has no compile (NC) function for it.

A simple grammar definition with a compile function for our new `add1` statement looks like the following:

```
DEF_RULE(add1_stmt, c(add1_stmt), and(2), tok(KW_ADD1), rule(testlist))
```

The second argument `c(add1_stmt)` is the corresponding compile function that should be implemented in `py/compile.c` to turn this rule into executable code.

The third required argument can be `or` or `and`. This specifies the number of nodes associated with a statement. For example, in this case, our `add1` statement is similar to ADD1 in assembly language. It takes one numeric argument. Therefore, the `add1_stmt` has two nodes associated with it. One node is for the statement itself, i.e the literal `add1` corresponding to `KW_ADD1`, and the other for its argument, a `testlist` rule which is the top-level expression rule.

Note: The `add1` rule here is just an example and not part of the standard MicroPython grammar.

The fourth argument in this example is the token associated with the rule, `KW_ADD1`. This token should be defined in the lexer by editing `py/lexer.h`.

Defining the same rule without a compile function is achieved by using the `DEF_RULE_NC` macro and omitting the compile function argument:

```
DEF_RULE_NC(add1_stmt, and(2), tok(KW_ADD1), rule(testlist))
```

The remaining arguments take on the same meaning. A rule without a compile function must be handled explicitly by all rules that may have this rule as a node. Such NC-rules are usually used to express sub-parts of a complicated grammar structure that cannot be expressed in a single rule.

Note: The macros `DEF_RULE` and `DEF_RULE_NC` take other arguments. For an in-depth understanding of supported parameters, see py/grammar.h.

4.3.2 Adding a lexical token

Every rule defined in the grammar should have a token associated with it that is defined in `py/lexer.h`. Add this token by editing the `_mp_token_kind_t` enum:

```
typedef enum _mp_token_kind_t {
    ...
    MP_TOKEN_KW_OR,
    MP_TOKEN_KW_PASS,
    MP_TOKEN_KW_RAISE,
    MP_TOKEN_KW_RETURN,
    MP_TOKEN_KW_TRY,
    MP_TOKEN_KW_WHILE,
    MP_TOKEN_KW_WITH,
    MP_TOKEN_KW_YIELD,
    MP_TOKEN_KW_ADD1,
    ...
} mp_token_kind_t;
```

Then also edit `py/lexer.c` to add the new keyword literal text:

```
STATIC const char *const tok_kw[] = {
    ...
    "or",
    "pass",
    "raise",
    "return",
    "try",
    "while",
    "with",
    "yield",
    "add1",
    ...
};
```

Notice the keyword is named depending on what you want it to be. For consistency, maintain the naming standard accordingly.

Note: The order of these keywords in `py/lexer.c` must match the order of tokens in the enum defined in `py/lexer.h`.

4.3.3 Parsing

In the parsing stage the parser takes the tokens produced by the lexer and converts them to an abstract syntax tree (AST) or *parse tree*. The implementation for the parser is defined in py/parse.c.

The parser also maintains a table of constants for use in different aspects of parsing, similar to what a symbol table does.

Several optimizations like constant folding on integers for most operations e.g. logical, binary, unary, etc, and optimizing enhancements on parenthesis around expressions are performed during this phase, along with some optimizations on strings.

Its worth noting that *docstrings* are discarded and not accessible to the compiler. Even optimizations like string interning are not applied to *docstrings*.

4.3.4 Compiler passes

Like many compilers, MicroPython compiles all code to MicroPython bytecode or native code. The functionality that achieves this is implemented in py/compile.c. The most relevant method you should know about is this:

```
mp_obj_t mp_compile(mp_parse_tree_t *parse_tree, qstr source_file, bool is_repl) {
    // Compile the input parse_tree to a raw-code structure.
    mp_raw_code_t *rc = mp_compile_to_raw_code(parse_tree, source_file, is_repl);
    // Create and return a function object that executes the outer module.
    return mp_make_function_from_raw_code(rc, MP_OBJ_NULL, MP_OBJ_NULL);
}
```

The compiler compiles the code in four passes: scope, stack size, code size and emit. Each pass runs the same C code over the same AST data structure, with different things being computed each time based on the results of the previous pass.

First pass

In the first pass, the compiler learns about the known identifiers (variables) and their scope, being global, local, closed over, etc. In the same pass the emitter (bytecode or native code) also computes the number of labels needed for the emitted code.

```
// Compile pass 1.
comp->emit = emit_bc;
comp->emit_method_table = &emit_bc_method_table;

uint max_num_labels = 0;
for (scope_t *s = comp->scope_head; s != NULL && comp->compile_error == MP_OBJ_NULL; s =␣
↪s->next) {
    if (s->emit_options == MP_EMIT_OPT_ASM) {
        compile_scope_inline_asm(comp, s, MP_PASS_SCOPE);
    } else {
        compile_scope(comp, s, MP_PASS_SCOPE);

        // Check if any implicitly declared variables should be closed over.
        for (size_t i = 0; i < s->id_info_len; ++i) {
            id_info_t *id = &s->id_info[i];
            if (id->kind == ID_INFO_KIND_GLOBAL_IMPLICIT) {
```

(continues on next page)

(continued from previous page)

```
                scope_check_to_close_over(s, id);
            }
        }
    }
    ...
}
```

Second and third passes

The second and third passes involve computing the Python stack size and code size for the bytecode or native code. After the third pass the code size cannot change, otherwise jump labels will be incorrect.

```
for (scope_t *s = comp->scope_head; s != NULL && comp->compile_error == MP_OBJ_NULL; s =␣
↪s->next) {
    ...

    // Pass 2: Compute the Python stack size.
    compile_scope(comp, s, MP_PASS_STACK_SIZE);

    // Pass 3: Compute the code size.
    if (comp->compile_error == MP_OBJ_NULL) {
        compile_scope(comp, s, MP_PASS_CODE_SIZE);
    }

    ...
}
```

Just before pass two there is a selection for the type of code to be emitted, which can either be native or bytecode.

```
// Choose the emitter type.
switch (s->emit_options) {
    case MP_EMIT_OPT_NATIVE_PYTHON:
    case MP_EMIT_OPT_VIPER:
        if (emit_native == NULL) {
            emit_native = NATIVE_EMITTER(new)(&comp->compile_error, &comp->next_label,␣
↪max_num_labels);
        }
        comp->emit_method_table = NATIVE_EMITTER_TABLE;
        comp->emit = emit_native;
        break;

    default:
        comp->emit = emit_bc;
        comp->emit_method_table = &emit_bc_method_table;
        break;
}
```

The bytecode option is the default but something unique to note for the native code option is that there is another option via `VIPER`. See the *Emitting native code* section for more details on viper annotations.

There is also support for *inline assembly code*, where assembly instructions are written as Python function calls but are emitted directly as the corresponding machine code. This assembler has only three passes (scope, code size, emit) and uses a different implementation, not the `compile_scope` function. See the inline assembler tutorial for more details.

Fourth pass

The fourth pass emits the final code that can be executed, either bytecode in the virtual machine, or native code directly by the CPU.

```
for (scope_t *s = comp->scope_head; s != NULL && comp->compile_error == MP_OBJ_NULL; s =␣
↪s->next) {
    ...

    // Pass 4: Emit the compiled bytecode or native code.
    if (comp->compile_error == MP_OBJ_NULL) {
        compile_scope(comp, s, MP_PASS_EMIT);
    }
}
```

4.3.5 Emitting bytecode

Statements in Python code usually correspond to emitted bytecode, for example `a + b` generates push a then push b then binary op add. Some statements do not emit anything but instead affect other things like the scope of variables, for example `global a`.

The implementation of a function that emits bytecode looks similar to this:

```
void mp_emit_bc_unary_op(emit_t *emit, mp_unary_op_t op) {
    emit_write_bytecode_byte(emit, 0, MP_BC_UNARY_OP_MULTI + op);
}
```

We use the unary operator expressions for an example here but the implementation details are similar for other statements/expressions. The method `emit_write_bytecode_byte()` is a wrapper around the main function `emit_get_cur_to_write_bytecode()` that all functions must call to emit bytecode.

4.3.6 Emitting native code

Similar to how bytecode is generated, there should be a corresponding function in `py/emitnative.c` for each code statement:

```
STATIC void emit_native_unary_op(emit_t *emit, mp_unary_op_t op) {
    vtype_kind_t vtype;
    emit_pre_pop_reg(emit, &vtype, REG_ARG_2);
    if (vtype == VTYPE_PYOBJ) {
        emit_call_with_imm_arg(emit, MP_F_UNARY_OP, op, REG_ARG_1);
        emit_post_push_reg(emit, VTYPE_PYOBJ, REG_RET);
    } else {
        adjust_stack(emit, 1);
        EMIT_NATIVE_VIPER_TYPE_ERROR(emit,
            MP_ERROR_TEXT("unary op %q not implemented"), mp_unary_op_method_name[op]);
    }
}
```

The difference here is that we have to handle *viper typing*. Viper annotations allow us to handle more than one type of variable. By default all variables are Python objects, but with viper a variable can also be declared as a machine-typed variable like a native integer or pointer. Viper can be thought of as a superset of Python, where normal Python objects are handled as usual, while native machine variables are handled in an optimised way by using direct machine

instructions for the operations. Viper typing may break Python equivalence because, for example, integers become native integers and can overflow (unlike Python integers which extend automatically to arbitrary precision).

4.4 Memory Management

Unlike programming languages such as C/C++, MicroPython hides memory management details from the developer by supporting automatic memory management. Automatic memory management is a technique used by operating systems or applications to automatically manage the allocation and deallocation of memory. This eliminates challenges such as forgetting to free the memory allocated to an object. Automatic memory management also avoids the critical issue of using memory that is already released. Automatic memory management takes many forms, one of them being garbage collection (GC).

The garbage collector usually has two responsibilities;

1. Allocate new objects in available memory.
2. Free unused memory.

There are many GC algorithms but MicroPython uses the Mark and Sweep policy for managing memory. This algorithm has a mark phase that traverses the heap marking all live objects while the sweep phase goes through the heap reclaiming all unmarked objects.

Garbage collection functionality in MicroPython is available through the `gc` built-in module:

```
>>> x = 5
>>> x
5
>>> import gc
>>> gc.enable()
>>> gc.mem_alloc()
1312
>>> gc.mem_free()
2071392
>>> gc.collect()
19
>>> gc.disable()
>>>
```

Even when `gc.disable()` is invoked, collection can be triggered with `gc.collect()`.

4.4.1 The object model

All MicroPython objects are referred to by the `mp_obj_t` data type. This is usually word-sized (i.e. the same size as a pointer on the target architecture), and can be typically 32-bit (STM32, nRF, ESP32, Unix x86) or 64-bit (Unix x64). It can also be greater than a word-size for certain object representations, for example `OBJ_REPR_D` has a 64-bit sized `mp_obj_t` on a 32-bit architecture.

An `mp_obj_t` represents a MicroPython object, for example an integer, float, type, dict or class instance. Some objects, like booleans and small integers, have their value stored directly in the `mp_obj_t` value and do not require additional memory. Other objects have their value store elsewhere in memory (for example on the garbage-collected heap) and their `mp_obj_t` contains a pointer to that memory. A portion of `mp_obj_t` is the tag which tells what type of object it is.

See `py/mpconfig.h` for the specific details of the available representations.

Pointer tagging

Because pointers are word-aligned, when they are stored in an `mp_obj_t` the lower bits of this object handle will be zero. For example on a 32-bit architecture the lower 2 bits will be zero:

********|********|********|******00

These bits are reserved for purposes of storing a tag. The tag stores extra information as opposed to introducing a new field to store that information in the object, which may be inefficient. In MicroPython the tag tells if we are dealing with a small integer, interned (small) string or a concrete object, and different semantics apply to each of these.

For small integers the mapping is this:

********|********|********|*******1

Where the asterisks hold the actual integer value. For an interned string or an immediate object (e.g. `True`) the layout of the `mp_obj_t` value is, respectively:

********|********|********|*****010

********|********|********|*****110

While a concrete object that is none of the above takes the form:

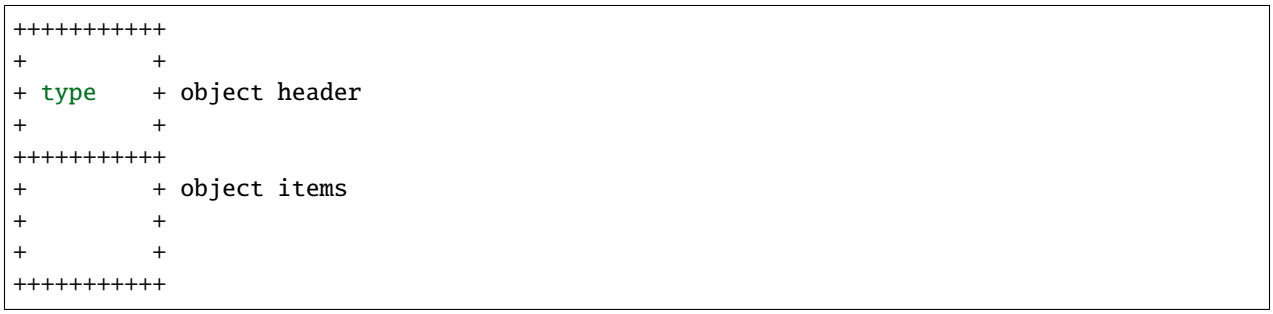
********|********|********|******00

The stars here correspond to the address of the concrete object in memory.

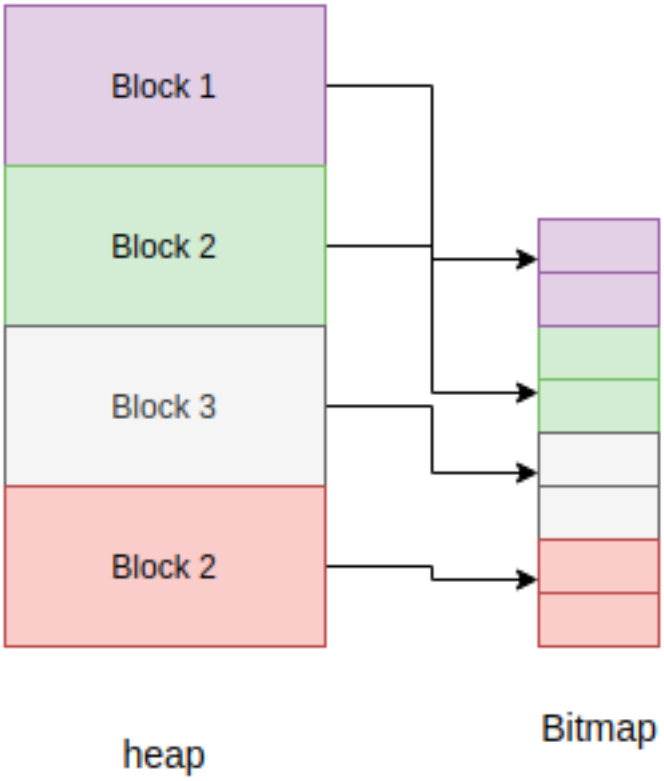
4.4.2 Allocation of objects

The value of a small integer is stored directly in the `mp_obj_t` and will be allocated in-place, not on the heap or elsewhere. As such, creation of small integers does not affect the heap. Similarly for interned strings that already have their textual data stored elsewhere, and immediate values like `None`, `False` and `True`.

Everything else which is a concrete object is allocated on the heap and its object structure is such that a field is reserved in the object header to store the type of the object.



The heaps smallest unit of allocation is a block, which is four machine words in size (16 bytes on a 32-bit machine, 32 bytes on a 64-bit machine). Another structure also allocated on the heap tracks the allocation of objects in each block. This structure is called a *bitmap*.



The bitmap tracks whether a block is free or in use and use two bits to track this state for each block.

The mark-sweep garbage collector manages the objects allocated on the heap, and also utilises the bitmap to mark objects that are still in use. See py/gc.c for the full implementation of these details.

Allocation: heap layout

The heap is arranged such that it consists of blocks in pools. A block can have different properties:

- *ATB(allocation table byte):* If set, then the block is a normal block
- *FREE:* Free block
- *HEAD:* Head of a chain of blocks
- *TAIL:* In the tail of a chain of blocks
- *MARK :* Marked head block
- *FTB(finaliser table byte):* If set, then the block has a finaliser

4.5 Implementing a Module

This chapter details how to implement a core module in MicroPython. MicroPython modules can be one of the following:

- Built-in module: A general module that is be part of the MicroPython repository.
- User module: A module that is useful for your specific project that you maintain in your own repository or private codebase.
- Dynamic module: A module that can be deployed and imported at runtime to your device.

A module in MicroPython can be implemented in one of the following locations:

- py/: A core library that mirrors core CPython functionality.
- extmod/: A CPython or MicroPython-specific module that is shared across multiple ports.
- ports/<port>/: A port-specific module.

Note: This chapter describes modules implemented in `py/` or core modules. See *Extending MicroPython in C* for details on implementing an external module. For details on port-specific modules, see *Porting MicroPython*.

4.5.1 Implementing a core module

Like CPython, MicroPython has core builtin modules that can be accessed through import statements. An example is the `gc` module discussed in *Memory Management*.

```
>>> import gc
>>> gc.enable()
>>>
```

MicroPython has several other builtin standard/core modules like `io`, `array` etc. Adding a new core module involves several modifications.

First, create the `C` file in the `py/` directory. In this example we are adding a hypothetical new module `subsystem` in the file `modsubsystem.c`:

```
#include "py/builtin.h"
#include "py/runtime.h"

#if MICROPY_PY_SUBSYSTEM

// info()
STATIC mp_obj_t py_subsystem_info(void) {
    return MP_OBJ_NEW_SMALL_INT(42);
}
MP_DEFINE_CONST_FUN_OBJ_0(subsystem_info_obj, py_subsystem_info);

STATIC const mp_rom_map_elem_t mp_module_subsystem_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_subsystem) },
    { MP_ROM_QSTR(MP_QSTR_info), MP_ROM_PTR(&subsystem_info_obj) },
};
STATIC MP_DEFINE_CONST_DICT(mp_module_subsystem_globals, mp_module_subsystem_globals_
↪table);

const mp_obj_module_t mp_module_subsystem = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&mp_module_subsystem_globals,
};

MP_REGISTER_MODULE(MP_QSTR_subsystem, mp_module_subsystem, MICROPY_PY_SUBSYSTEM);

#endif
```

The implementation includes a definition of all functions related to the module and adds the functions to the modules global table in `mp_module_subsystem_globals_table`. It also creates the module object with `mp_module_subsystem`. The module is then registered with the wider system via the `MP_REGISTER_MODULE` macro.

After building and running the modified MicroPython, the module should now be importable:

```
>>> import subsystem
>>> subsystem.info()
42
>>>
```

Our `info()` function currently returns just a single number but can be extended to do anything. Similarly, more functions can be added to this new module.

4.6 Optimizations

MicroPython uses several optimizations to save RAM but also ensure the efficient execution of programs. This chapter discusses some of these optimizations.

Note: *MicroPython string interning* and *Maps and Dictionaries* details other optimizations on strings and dictionaries.

4.6.1 Frozen bytecode

When MicroPython loads Python code from the filesystem, it first has to parse the file into a temporary in-memory representation, and then generate bytecode for execution, both of which are stored in the heap (in RAM). This can lead to significant amounts of memory being used. The MicroPython cross compiler can be used to generate a `.mpy` file, containing the pre-compiled bytecode for a Python module. This will still be loaded into RAM, but it avoids the additional overhead of the parsing stage.

As a further optimisation, the pre-compiled bytecode from a `.mpy` file can be frozen into the firmware image as part of the main firmware compilation process, which means that the bytecode will be executed from ROM. This can lead to a significant memory saving, and reduce heap fragmentation.

4.6.2 Variables

MicroPython processes local and global variables differently. Global variables are stored and looked up from a global dictionary that is allocated on the heap (note that each module has its own separate dict, so separate namespace). Local variables on the other hand are are stored on the Python value stack, which may live on the C stack or on the heap. They are accessed directly by their offset within the Python stack, which is more efficient than a global lookup in a dict.

The length of global variable names also affects how much RAM is used as identifiers are stored in RAM. The shorter the identifier, the less memory is used.

The other aspect is that `const` variables that start with an underscore are treated as proper constants and are not allocated or added in a dictionary, hence saving some memory. These variables use `const()` from the MicroPython library. Therefore:

```
from micropython import const

X = const(1)
```

(continues on next page)

(continued from previous page)

```
_Y = const(2)
foo(X, _Y)
```

Compiles to:

```
X = 1
foo(1, 2)
```

4.6.3 Allocation of memory

Most of the common MicroPython constructs are not allocated on the heap. However the following are:

- Dynamic data structures like lists, mappings, etc;
- Functions, classes and object instances;
- imports; and
- First-time assignment of global variables (to create the slot in the global dict).

For a detailed discussion on a more user-centric perspective on optimization, see Maximising MicroPython speed

4.7 MicroPython string interning

MicroPython uses string interning to save both RAM and ROM. This avoids having to store duplicate copies of the same string. Primarily, this applies to identifiers in your code, as something like a function or variable name is very likely to appear in multiple places in the code. In MicroPython an interned string is called a QSTR (uniQue STRing).

A QSTR value (with type `qstr`) is a index into a linked list of QSTR pools. QSTRs store their length and a hash of their contents for fast comparison during the de-duplication process. All bytecode operations that work with strings use a QSTR argument.

4.7.1 Compile-time QSTR generation

In the MicroPython C code, any strings that should be interned in the final firmware are written as `MP_QSTR_Foo`. At compile time this will evaluate to a `qstr` value that points to the index of `"Foo"` in the QSTR pool.

A multi-step process in the `Makefile` makes this work. In summary this process has three parts:

1. Find all `MP_QSTR_Foo` tokens in the code.
2. Generate a static QSTR pool containing all the string data (including lengths and hashes).
3. Replace all `MP_QSTR_Foo` (via the preprocessor) with their corresponding index.

`MP_QSTR_Foo` tokens are searched for in two sources:

1. All files referenced in `$(SRC_QSTR)`. This is all C code (i.e. `py`, `extmod`, `ports/stm32`) but not including third-party code such as `lib`.
2. Additional `$(QSTR_GLOBAL_DEPENDENCIES)` (which includes `mpconfig*.h`).

Note: `frozen_mpy.c` (generated by mpy-tool.py) has its own QSTR generation and pool.

Some additional strings that cant be expressed using the `MP_QSTR_Foo` syntax (e.g. they contain non-alphanumeric characters) are explicitly provided in `qstrdefs.h` and `qstrdefsport.h` via the `$(QSTR_DEFS)` variable.

Processing happens in the following stages:

1. `qstr.i.last` is the concatenation of putting every single input file through the C pre-processor. This means that any conditionally disabled code will be removed, and macros expanded. This means we dont add strings to the pool that wont be used in the final firmware. Because at this stage (thanks to the `NO_QSTR` macro added by `QSTR_GEN_CFLAGS`) there is no definition for `MP_QSTR_Foo` it passes through this stage unaffected. This file also includes comments from the preprocessor that include line number information. Note that this step only uses files that have changed, which means that `qstr.i.last` will only contain data from files that have changed since the last compile.
2. `qstr.split` is an empty file created after running `makeqstrdefs.py split` on qstr.i.last. Its just used as a dependency to indicate that the step ran. This script outputs one file per input C file, `genhdr/qstr/...file.c.qstr`, which contains only the matched QSTRs. Each QSTR is printed as `Q(Foo)`. This step is necessary to combine the existing files with the new data generated from the incremental update in `qstr.i.last`.
3. `qstrdefs.collected.h` is the output of concatenating `genhdr/qstr/*` using `makeqstrdefs.py cat`. This is now the full set of `MP_QSTR_Foo`s found in the code, now formatted as `Q(Foo)`, one-per-line, with duplicates. This file is only updated if the set of qstrs has changed. A hash of the QSTR data is written to another file (`qstrdefs.collected.h.hash`) which allows it to track changes across builds.
4. Generate an enumeration, each entry of which maps a `MP_QSTR_Foo` to its corresponding index. It concatenates `qstrdefs.collected.h` with `qstrdefs*.h`, then it transforms each line from `Q(Foo)` to `"Q(Foo)"` so they pass through the preprocessor unchanged. Then the preprocessor is used to deal with any conditional compilation in `qstrdefs*.h`. Then the transformation is undone back to `Q(Foo)`, and saved as `qstrdefs.preprocessed.h`.
5. `qstrdefs.generated.h` is the output of `makeqstrdata.py`. For each `Q(Foo)` in qstrdefs.preprocessed.h (plus some extra hard-coded ones), it outputs `QDEF(MP_QSTR_Foo, (const byte*)"hash" "Foo")`.

Then in the main compile, two things happen with `qstrdefs.generated.h`:

1. In qstr.h, each QDEF becomes an entry in an enum, which makes `MP_QSTR_Foo` available to code and equal to the index of that string in the QSTR table.
2. In qstr.c, the actual QSTR data table is generated as elements of the `mp_qstr_const_pool->qstrs`.

4.7.2 Run-time QSTR generation

Additional QSTR pools can be created at runtime so that strings can be added to them. For example, the code:

```
foo[x] = 3
```

Will need to create a QSTR for the value of `x` so it can be used by the load attr bytecode.

Also, when compiling Python code, identifiers and literals need to have QSTRs created. Note: only literals shorter than 10 characters become QSTRs. This is because a regular string on the heap always takes up a minimum of 16 bytes (one GC block), whereas QSTRs allow them to be packed more efficiently into the pool.

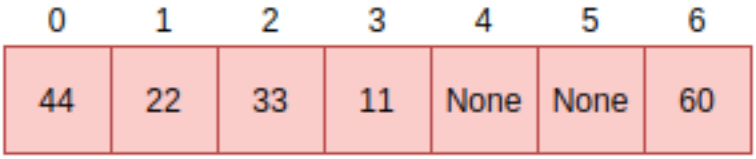
QSTR pools (and the underlying chunks that store the string data) are allocated on-demand on the heap with a minimum size.

4.8 Maps and Dictionaries

MicroPython dictionaries and maps use techniques called open addressing and linear probing. This chapter details both of these methods.

4.8.1 Open addressing

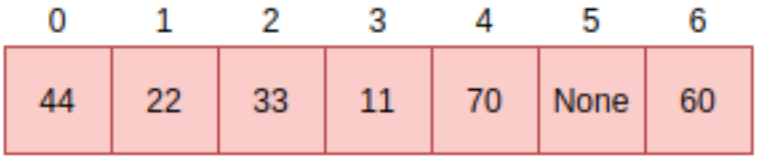
Open addressing is used to resolve collisions. Collisions are very common occurrences and happen when two items happen to hash to the same slot or location. For example, given a hash setup as this:



If there is a request to fill slot `0` with `70`, since the slot `0` is not empty, open addressing finds the next available slot in the dictionary to service this request. This sequential search for an alternate location is called *probing*. There are several sequence probing algorithms but MicroPython uses linear probing that is described in the next section.

4.8.2 Linear probing

Linear probing is one of the methods for finding an available address or slot in a dictionary. In MicroPython, it is used with open addressing. To service the request described above, unlike other probing algorithms, linear probing assumes a fixed interval of `1` between probes. The request will therefore be serviced by placing the item in the next free slot which is slot `4` in our example:



The same methods i.e open addressing and linear probing are used to search for an item in a dictionary. Assume we want to search for the data item `33`. The computed hash value will be 2. Looking at slot 2 reveals `33`, at this point, we return `True`. Searching for `70` is quite different as there was a collision at the time of insertion. Therefore computing the hash value is `0` which is currently holding `44`. Instead of simply returning `False`, we perform a sequential search starting at point `1` until the item `70` is found or we encounter a free slot. This is the general way of performing look-ups in hashes:

```
// not yet found, keep searching in this table
pos = (pos + 1) % set->alloc;

if (pos == start_pos) {
    // search got back to starting position, so index is not in table
    if (lookup_kind & MP_MAP_LOOKUP_ADD_IF_NOT_FOUND) {
```

(continues on next page)

(continued from previous page)

```
        if (avail_slot != NULL) {
            // there was an available slot, so use that
            set->used++;
            *avail_slot = index;
            return index;
        } else {
            // not enough room in table, rehash it
            mp_set_rehash(set);
            // restart the search for the new element
            start_pos = pos = hash % set->alloc;
        }
    }
} else {
     return MP_OBJ_NULL;
}
```

4.9 The public C API

The public C-API comprises functions defined in all C header files in the `py/` directory. Most of the important core runtime C APIs are exposed in `runtime.h` and `obj.h`.

The following is an example of public API functions from `obj.h`:

```
mp_obj_t mp_obj_new_list(size_t n, mp_obj_t *items);
mp_obj_t mp_obj_list_append(mp_obj_t self_in, mp_obj_t arg);
mp_obj_t mp_obj_list_remove(mp_obj_t self_in, mp_obj_t value);
void mp_obj_list_get(mp_obj_t self_in, size_t *len, mp_obj_t **items);
```

At its core, any functions and macros in header files make up the public API and can be used to access very low-level details of MicroPython. Static inline functions in header files are fine too, such functions will be inlined in the code when used.

Header files in the `ports` directory are only exposed to the functionality specific to a given port.

4.10 Extending MicroPython in C

This chapter describes options for implementing additional functionality in C, but from code written outside of the main MicroPython repository. The first approach is useful for building your own custom firmware with some project-specific additional modules or functions that can be accessed from Python. The second approach is for building modules that can be loaded at runtime.

Please see the *library section* for more information on building core modules that live in the main MicroPython repository.

4.10.1 MicroPython external C modules

When developing modules for use with MicroPython you may find you run into limitations with the Python environment, often due to an inability to access certain hardware resources or Python speed limitations.

If your limitations cant be resolved with suggestions in *Maximising MicroPython speed*, writing some or all of your module in C (and/or C++ if implemented for your port) is a viable option.

If your module is designed to access or work with commonly available hardware or libraries please consider implementing it inside the MicroPython source tree alongside similar modules and submitting it as a pull request. If however youre targeting obscure or proprietary systems it may make more sense to keep this external to the main MicroPython repository.

This chapter describes how to compile such external modules into the MicroPython executable or firmware image. Both Make and CMake build tools are supported, and when writing an external module its a good idea to add the build files for both of these tools so the module can be used on all ports. But when compiling a particular port you will only need to use one method of building, either Make or CMake.

An alternative approach is to use *Native machine code in .mpy files* which allows writing custom C code that is placed in a .mpy file, which can be imported dynamically in to a running MicroPython system without the need to recompile the main firmware.

Structure of an external C module

A MicroPython user C module is a directory with the following files:

- `*.c` / `*.cpp` / `*.h` source code files for your module.

  These will typically include the low level functionality being implemented and the MicroPython binding functions to expose the functions and module(s).

  Currently the best reference for writing these functions/modules is to find similar modules within the MicroPython tree and use them as examples.

- `micropython.mk` contains the Makefile fragment for this module.

  `$(USERMOD_DIR)` is available in `micropython.mk` as the path to your module directory. As its redefined for each c module, is should be expanded in your `micropython.mk` to a local make variable, eg `EXAMPLE_MOD_DIR := $(USERMOD_DIR)`

  Your `micropython.mk` must add your modules source files relative to your expanded copy of `$(USERMOD_DIR)` to `SRC_USERMOD`, eg `SRC_USERMOD += $(EXAMPLE_MOD_DIR)/example.c`

  If you have custom compiler options (like `-I` to add directories to search for header files), these should be added to `CFLAGS_USERMOD` for C code and to `CXXFLAGS_USERMOD` for C++ code.

- `micropython.cmake` contains the CMake configuration for this module.

  In `micropython.cmake`, you may use `${CMAKE_CURRENT_LIST_DIR}` as the path to the current module.

  Your `micropython.cmake` should define an `INTERFACE` library and associate your source files, compile definitions and include directories with it. The library should then be linked to the `usermod` target.

```
add_library(usermod_cexample INTERFACE)

target_sources(usermod_cexample INTERFACE
    ${CMAKE_CURRENT_LIST_DIR}/examplemodule.c
)

target_include_directories(usermod_cexample INTERFACE
```

(continues on next page)

(continued from previous page)

```
    ${CMAKE_CURRENT_LIST_DIR}
)

target_link_libraries(usermod INTERFACE usermod_cexample)
```

See below for full usage example.

Basic example

This simple module named `cexample` provides a single function `cexample.add_ints(a, b)` which adds the two integer args together and returns the result. It can be found in the MicroPython source tree in the examples directory and has a source file and a Makefile fragment with content as described above:

```
micropython/
examples/
   usercmodule/
      cexample/
          examplemodule.c
          micropython.mk
          micropython.cmake
```

Refer to the comments in these files for additional explanation. Next to the `cexample` module theres also `cppexample` which works in the same way but shows one way of mixing C and C++ code in MicroPython.

Compiling the cmodule into MicroPython

To build such a module, compile MicroPython (see getting started), applying 2 modifications:

1. Set the build-time flag `USER_C_MODULES` to point to the modules you want to include. For ports that use Make this variable should be a directory which is searched automatically for modules. For ports that use CMake this variable should be a file which includes the modules to build. See below for details.
2. Enable the modules by setting the corresponding C preprocessor macro to 1. This is only needed if the modules you are building are not automatically enabled.

For building the example modules which come with MicroPython, set `USER_C_MODULES` to the `examples/usercmodule` directory for Make, or to `examples/usercmodule/micropython.cmake` for CMake.

For example, heres how the to build the unix port with the example modules:

```
cd micropython/ports/unix
make USER_C_MODULES=../../examples/usercmodule
```

You may need to run `make clean` once at the start when including new user modules in the build. The build output will show the modules found:

```
...
Including User C Module from ../../examples/usercmodule/cexample
Including User C Module from ../../examples/usercmodule/cppexample
...
```

For a CMake-based port such as rp2, this will look a little different (note that CMake is actually invoked by `make`):

```
cd micropython/ports/rp2
make USER_C_MODULES=../../examples/usercmodule/micropython.cmake
```

Again, you may need to run `make clean` first for CMake to pick up the user modules. The CMake build output lists the modules by name:

```
...
Including User C Module(s) from ../../examples/usercmodule/micropython.cmake
Found User C Module(s): usermod_cexample, usermod_cppexample
...
```

The contents of the top-level `micropython.cmake` can be used to control which modules are enabled.

For your own projects its more convenient to keep custom code out of the main MicroPython source tree, so a typical project directory structure will look like this:

```
my_project/
 modules/
    example1/
       example1.c
       micropython.mk
       micropython.cmake
    example2/
       example2.c
       micropython.mk
       micropython.cmake
    micropython.cmake
 micropython/
    ports/
   ... stm32/
      ...
```

When building with Make set `USER_C_MODULES` to the `my_project/modules` directory. For example, building the stm32 port:

```
cd my_project/micropython/ports/stm32
make USER_C_MODULES=../../../modules
```

When building with CMake the top level `micropython.cmake` – found directly in the `my_project/modules` directory – should `include` all of the modules you want to have available:

```
include(${CMAKE_CURRENT_LIST_DIR}/example1/micropython.cmake)
include(${CMAKE_CURRENT_LIST_DIR}/example2/micropython.cmake)
```

Then build with:

```
cd my_project/micropython/ports/esp32
make USER_C_MODULES=../../../../modules/micropython.cmake
```

Note that the esp32 port needs the extra `..` for relative paths due to the location of its main `CMakeLists.txt` file. You can also specify absolute paths to `USER_C_MODULES`.

All modules specified by the `USER_C_MODULES` variable (either found in this directory when using Make, or added via `include` when using CMake) will be compiled, but only those which are enabled will be available for importing. User modules are usually enabled by default (this is decided by the developer of the module), in which case there is nothing more to do than set `USER_C_MODULES` as described above.

If a module is not enabled by default then the corresponding C preprocessor macro must be enabled. This macro name can be found by searching for the `MP_REGISTER_MODULE` line in the modules source code (it usually appears at the end of the main source file). The third argument to `MP_REGISTER_MODULE` is the macro name, and this must be set to 1 using `CFLAGS_EXTRA` to make the module available. If the third argument is just the number 1 then the module is enabled by default.

For example, the `examples/usercmodule/cexample` module is enabled by default so has the following line in its source code:

```
MP_REGISTER_MODULE(MP_QSTR_cexample, example_user_cmodule, 1);
```

Alternatively, to make this module disabled by default but selectable through a preprocessor configuration option, it would be:

```
MP_REGISTER_MODULE(MP_QSTR_cexample, example_user_cmodule, MODULE_CEXAMPLE_
↪ENABLED);
```

In this case the module is enabled by adding `CFLAGS_EXTRA=-DMODULE_CEXAMPLE_ENABLED=1` to the `make` command, or editing `mpconfigport.h` or `mpconfigboard.h` to add

```
#define MODULE_CEXAMPLE_ENABLED (1)
```

Note that the exact method depends on the port as they have different structures. If not done correctly it will compile but importing will fail to find the module.

Module usage in MicroPython

Once built into your copy of MicroPython, the module can now be accessed in Python just like any other builtin module, e.g.

```
import cexample
print(cexample.add_ints(1, 3))
# should display 4
```

4.10.2 Native machine code in .mpy files

This section describes how to build and work with .mpy files that contain native machine code from a language other than Python. This allows you to write code in a language like C, compile and link it into a .mpy file, and then import this file like a normal Python module. This can be used for implementing functionality which is performance critical, or for including an existing library written in another language.

One of the main advantages of using native .mpy files is that native machine code can be imported by a script dynamically, without the need to rebuild the main MicroPython firmware. This is in contrast to *MicroPython external C modules* which also allows defining custom modules in C but they must be compiled into the main firmware image.

The focus here is on using C to build native modules, but in principle any language which can be compiled to stand-alone machine code can be put into a .mpy file.

A native .mpy module is built using the `mpy_ld.py` tool, which is found in the `tools/` directory of the project. This tool takes a set of object files (.o files) and links them together to create a native .mpy files. It requires CPython 3 and the library pyelftools v0.25 or greater.

Supported features and limitations

A .mpy file can contain MicroPython bytecode and/or native machine code. If it contains native machine code then the .mpy file has a specific architecture associated with it. Current supported architectures are (these are the valid options for the `ARCH` variable, see below):

- `x86` (32 bit)
- `x64` (64 bit x86)
- `armv7m` (ARM Thumb 2, eg Cortex-M3)
- `armv7emsp` (ARM Thumb 2, single precision float, eg Cortex-M4F, Cortex-M7)
- `armv7emdp` (ARM Thumb 2, double precision float, eg Cortex-M7)
- `xtensa` (non-windowed, eg ESP8266)
- `xtensawin` (windowed with window size 8, eg ESP32)

When compiling and linking the native .mpy file the architecture must be chosen and the corresponding file can only be imported on that architecture. For more details about .mpy files see *MicroPython .mpy files*.

Native code must be compiled as position independent code (PIC) and use a global offset table (GOT), although the details of this varies from architecture to architecture. When importing .mpy files with native code the import machinery is able to do some basic relocation of the native code. This includes relocating text, rodata and BSS sections.

Supported features of the linker and dynamic loader are:

- executable code (text)
- read-only data (rodata), including strings and constant data (arrays, structs, etc)
- zeroed data (BSS)
- pointers in text to text, rodata and BSS
- pointers in rodata to text, rodata and BSS

The known limitations are:

- data sections are not supported; workaround: use BSS data and initialise the data values explicitly
- static BSS variables are not supported; workaround: use global BSS variables

So, if your C code has writable data, make sure the data is defined globally, without an initialiser, and only written to within functions.

Linker limitation: the native module is not linked against the symbol table of the full MicroPython firmware. Rather, it is linked against an explicit table of exported symbols found in `mp_fun_table` (in `py/nativeglue.h`), that is fixed at firmware build time. It is thus not possible to simply call some arbitrary HAL/OS/RTOS/system function, for example.

New symbols can be added to the end of the table and the firmware rebuilt. The symbols also need to be added to `tools/mpy_ld.py`s `fun_table` dict in the same location. This allows `mpy_ld.py` to be able to pick the new symbols up and provide relocations for them when the mpy is imported. Finally, if the symbol is a function, a macro or stub should be added to `py/dynruntime.h` to make it easy to call the function.

Defining a native module

A native .mpy module is defined by a set of files that are used to build the .mpy. The filesystem layout consists of two main parts, the source files and the Makefile:

- In the simplest case only a single C source file is required, which contains all the code that will be compiled into the .mpy module. This C source code must include the `py/dynruntime.h` file to access the MicroPython dynamic API, and must at least define a function called `mpy_init`. This function will be the entry point of the module, called when the module is imported.

  The module can be split into multiple C source files if desired. Parts of the module can also be implemented in Python. All source files should be listed in the Makefile, by adding them to the `SRC` variable (see below). This includes both C source files as well as any Python files which will be included in the resulting .mpy file.

- The `Makefile` contains the build configuration for the module and list the source files used to build the .mpy module. It should define `MPY_DIR` as the location of the MicroPython repository (to find header files, the relevant Makefile fragment, and the `mpy_ld.py` tool), `MOD` as the name of the module, `SRC` as the list of source files, optionally specify the machine architecture via `ARCH`, and then include `py/dynruntime.mk`.

Minimal example

This section provides a fully working example of a simple module named `factorial`. This module provides a single function `factorial.factorial(x)` which computes the factorial of the input and returns the result.

Directory layout:

```
factorial/
 factorial.c
 Makefile
```

The file `factorial.c` contains:

```
// Include the header file to get access to the MicroPython API
#include "py/dynruntime.h"

// Helper function to compute factorial
STATIC mp_int_t factorial_helper(mp_int_t x) {
    if (x == 0) {
        return 1;
    }
    return x * factorial_helper(x - 1);
}

// This is the function which will be called from Python, as factorial(x)
STATIC mp_obj_t factorial(mp_obj_t x_obj) {
    // Extract the integer from the MicroPython input object
    mp_int_t x = mp_obj_get_int(x_obj);
    // Calculate the factorial
    mp_int_t result = factorial_helper(x);
    // Convert the result to a MicroPython integer object and return it
    return mp_obj_new_int(result);
}
// Define a Python reference to the function above
STATIC MP_DEFINE_CONST_FUN_OBJ_1(factorial_obj, factorial);
```

(continues on next page)

(continued from previous page)

```
// This is the entry point and is called when the module is imported
mp_obj_t mpy_init(mp_obj_fun_bc_t *self, size_t n_args, size_t n_kw, mp_obj_t *args) {
    // This must be first, it sets up the globals dict and other things
    MP_DYNRUNTIME_INIT_ENTRY

    // Make the function available in the module's namespace
    mp_store_global(MP_QSTR_factorial, MP_OBJ_FROM_PTR(&factorial_obj));

    // This must be last, it restores the globals dict
    MP_DYNRUNTIME_INIT_EXIT
}
```

The file `Makefile` contains:

```
# Location of top-level MicroPython directory
MPY_DIR = ../../..

# Name of module
MOD = factorial

# Source files (.c or .py)
SRC = factorial.c

# Architecture to build for (x86, x64, armv7m, xtensa, xtensawin)
ARCH = x64

# Include to get the rules for compiling and linking the module
include $(MPY_DIR)/py/dynruntime.mk
```

Compiling the module

The prerequisite tools needed to build a native .mpy file are:

- The MicroPython repository (at least the `py/` and `tools/` directories).
- CPython 3, and the library pyelftools (eg `pip install 'pyelftools>=0.25'`).
- GNU make.
- A C compiler for the target architecture (if C source is used).
- Optionally `mpy-cross`, built from the MicroPython repository (if .py source is used).

Be sure to select the correct `ARCH` for the target you are going to run on. Then build with:

```
$ make
```

Without modifying the Makefile you can specify the target architecture via:

```
$ make ARCH=armv7m
```

Module usage in MicroPython

Once the module is built there should be a file called `factorial.mpy`. Copy this so it is accessible on the filesystem of your MicroPython system and can be found in the import path. The module can now be accessed in Python just like any other module, for example:

```
import factorial
print(factorial.factorial(10))
# should display 3628800
```

Further examples

See `examples/natmod/` for further examples which show many of the available features of native .mpy modules. Such features include:

- using multiple C source files
- including Python code alongside C code
- rodata and BSS data
- memory allocation
- use of floating point
- exception handling
- including external C libraries

4.11 Porting MicroPython

The MicroPython project contains several ports to different microcontroller families and architectures. The project repository has a ports directory containing a subdirectory for each supported port.

A port will typically contain definitions for multiple boards, each of which is a specific piece of hardware that that port can run on, e.g. a development kit or device.

The minimal port is available as a simplified reference implementation of a MicroPython port. It can run on both the host system and an STM32F4xx MCU.

In general, starting a port requires:

- Setting up the toolchain (configuring Makefiles, etc).
- Implementing boot configuration and CPU initialization.
- Initialising basic drivers required for development and debugging (e.g. GPIO, UART).
- Performing the board-specific configurations.
- Implementing the port-specific modules.

4.11.1 Minimal MicroPython firmware

The best way to start porting MicroPython to a new board is by integrating a minimal MicroPython interpreter. For this walkthrough, create a subdirectory for the new port in the `ports` directory:

```
$ cd ports
$ mkdir example_port
```

The basic MicroPython firmware is implemented in the main port file, e.g `main.c`:

```
#include "py/compile.h"
#include "py/gc.h"
#include "py/mperrno.h"
#include "py/stackctrl.h"
#include "shared/runtime/gchelper.h"
#include "shared/runtime/pyexec.h"

// Allocate memory for the MicroPython GC heap.
static char heap[4096];

int main(int argc, char **argv) {
    // Initialise the MicroPython runtime.
    mp_stack_ctrl_init();
    gc_init(heap, heap + sizeof(heap));
    mp_init();

    // Start a normal REPL; will exit when ctrl-D is entered on a blank line.
    pyexec_friendly_repl();

    // Deinitialise the runtime.
    gc_sweep_all();
    mp_deinit();
    return 0;
}

// Handle uncaught exceptions (should never be reached in a correct C implementation).
void nlr_jump_fail(void *val) {
    for (;;) {
    }
}

// Do a garbage collection cycle.
void gc_collect(void) {
    gc_collect_start();
    gc_helper_collect_regs_and_stack();
    gc_collect_end();
}

// There is no filesystem so stat'ing returns nothing.
mp_import_stat_t mp_import_stat(const char *path) {
    return MP_IMPORT_STAT_NO_EXIST;
}

// There is no filesystem so opening a file raises an exception.
```

(continues on next page)

(continued from previous page)

```
mp_lexer_t *mp_lexer_new_from_file(const char *filename) {
    mp_raise_OSError(MP_ENOENT);
}
```

We also need a Makefile at this point for the port:

```
# Include the core environment definitions; this will set $(TOP).
include ../../py/mkenv.mk

# Include py core make definitions.
include $(TOP)/py/py.mk

# Set CFLAGS and libraries.
CFLAGS = -I. -I$(BUILD) -I$(TOP)
LIBS = -lm

# Define the required source files.
SRC_C = \
    main.c \
    mphalport.c \
    shared/readline/readline.c \
    shared/runtime/gchelper_generic.c \
    shared/runtime/pyexec.c \
    shared/runtime/stdout_helpers.c \

# Define the required object files.
OBJ = $(PY_CORE_O) $(addprefix $(BUILD)/, $(SRC_C:.c=.o))

# Define the top-level target, the main firmware.
all: $(BUILD)/firmware.elf

# Define how to build the firmware.
$(BUILD)/firmware.elf: $(OBJ)
    $(ECHO) "LINK $@"
    $(Q)$(CC) $(LDFLAGS) -o $@ $^ $(LIBS)
    $(Q)$(SIZE) $@

# Include remaining core make rules.
include $(TOP)/py/mkrules.mk
```

Remember to use proper tabs to indent the Makefile.

4.11.2 MicroPython Configurations

After integrating the minimal code above, the next step is to create the MicroPython configuration files for the port. The compile-time configurations are specified in `mpconfigport.h` and additional hardware-abstraction functions, such as time keeping, in `mphalport.h`.

The following is an example of an `mpconfigport.h` file:

```
#include <stdint.h>
```

(continues on next page)

(continued from previous page)

```
// Python internal features.
#define MICROPY_ENABLE_GC                       (1)
#define MICROPY_HELPER_REPL                     (1)
#define MICROPY_ERROR_REPORTING                 (MICROPY_ERROR_REPORTING_TERSE)
#define MICROPY_FLOAT_IMPL                      (MICROPY_FLOAT_IMPL_FLOAT)

// Fine control over Python builtins, classes, modules, etc.
#define MICROPY_PY_ASYNC_AWAIT                  (0)
#define MICROPY_PY_BUILTINS_SET                 (0)
#define MICROPY_PY_ATTRTUPLE                    (0)
#define MICROPY_PY_COLLECTIONS                  (0)
#define MICROPY_PY_MATH                         (0)
#define MICROPY_PY_IO                           (0)
#define MICROPY_PY_STRUCT                       (0)

// Type definitions for the specific machine.

typedef intptr_t mp_int_t; // must be pointer size
typedef uintptr_t mp_uint_t; // must be pointer size
typedef long mp_off_t;

// We need to provide a declaration/definition of alloca().
#include <alloca.h>

// Define the port's name and hardware.
#define MICROPY_HW_BOARD_NAME "example-board"
#define MICROPY_HW_MCU_NAME   "unknown-cpu"

#define MP_STATE_PORT MP_STATE_VM

#define MICROPY_PORT_ROOT_POINTERS \
    const char *readline_hist[8];
```

This configuration file contains machine-specific configurations including aspects like if different MicroPython features should be enabled e.g. `#define MICROPY_ENABLE_GC (1)`. Making this Setting `(0)` disables the feature.

Other configurations include type definitions, root pointers, board name, microcontroller name etc.

Similarly, an minimal example `mphalport.h` file looks like this:

```
static inline void mp_hal_set_interrupt_char(char c) {}
```

4.11.3 Support for standard input/output

MicroPython requires at least a way to output characters, and to have a REPL it also requires a way to input characters. Functions for this can be implemented in the file `mphalport.c`, for example:

```
#include <unistd.h>
#include "py/mpconfig.h"

// Receive single character, blocking until one is available.
int mp_hal_stdin_rx_chr(void) {
```

(continues on next page)

(continued from previous page)

```
    unsigned char c = 0;
    int r = read(STDIN_FILENO, &c, 1);
    (void)r;
    return c;
}

// Send the string of given length.
void mp_hal_stdout_tx_strn(const char *str, mp_uint_t len) {
    int r = write(STDOUT_FILENO, str, len);
    (void)r;
}
```

These input and output functions have to be modified depending on the specific board API. This example uses the standard input/output stream.

4.11.4 Building and running

At this stage the directory of the new port should contain:

```
ports/example_port/
 main.c
 Makefile
 mpconfigport.h
 mphalport.c
 mphalport.h
```

The port can now be built by running `make` (or otherwise, depending on your system).

If you are using the default compiler settings in the Makefile given above then this will create an executable called `build/firmware.elf` which can be executed directly. To get a functional REPL you may need to first configure the terminal to raw mode:

```
$ stty raw opost -echo
$ ./build/firmware.elf
```

That should give a MicroPython REPL. You can then run commands like:

```
MicroPython v1.13 on 2021-01-01; example-board with unknown-cpu
>>> import sys
>>> sys.implementation
('micropython', (1, 13, 0))
>>>
```

Use Ctrl-D to exit, and then run `reset` to reset the terminal.

4.11.5 Adding a module to the port

To add a custom module like `myport`, first add the module definition in a file `modmyport.c`:

```
#include "py/runtime.h"

STATIC mp_obj_t myport_info(void) {
    mp_printf(&mp_plat_print, "info about my port\n");
    return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_0(myport_info_obj, myport_info);

STATIC const mp_rom_map_elem_t myport_module_globals_table[] = {
    { MP_OBJ_NEW_QSTR(MP_QSTR___name__), MP_OBJ_NEW_QSTR(MP_QSTR_myport) },
    { MP_ROM_QSTR(MP_QSTR_info), MP_ROM_PTR(&myport_info_obj) },
};
STATIC MP_DEFINE_CONST_DICT(myport_module_globals, myport_module_globals_table);

const mp_obj_module_t myport_module = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&myport_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_myport, myport_module, 1);
```

Note: the 1 as the third argument in `MP_REGISTER_MODULE` enables this new module unconditionally. To allow it to be conditionally enabled, replace the 1 by `MICROPY_PY_MYPORT` and then add `#define MICROPY_PY_MYPORT (1)` in `mpconfigport.h` accordingly.

You will also need to edit the Makefile to add `modmyport.c` to the `SRC_C` list, and a new line adding the same file to `SRC_QSTR` (so qstrs are searched for in this new file), like this:

```
SRC_C = \
    main.c \
    modmyport.c \
    mphalport.c \
    ...

SRC_QSTR += modport.c
```

If all went correctly then, after rebuilding, you should be able to import the new module:

```
>>> import myport
>>> myport.info()
info about my port
>>>
```

CHAPTER

FIVE

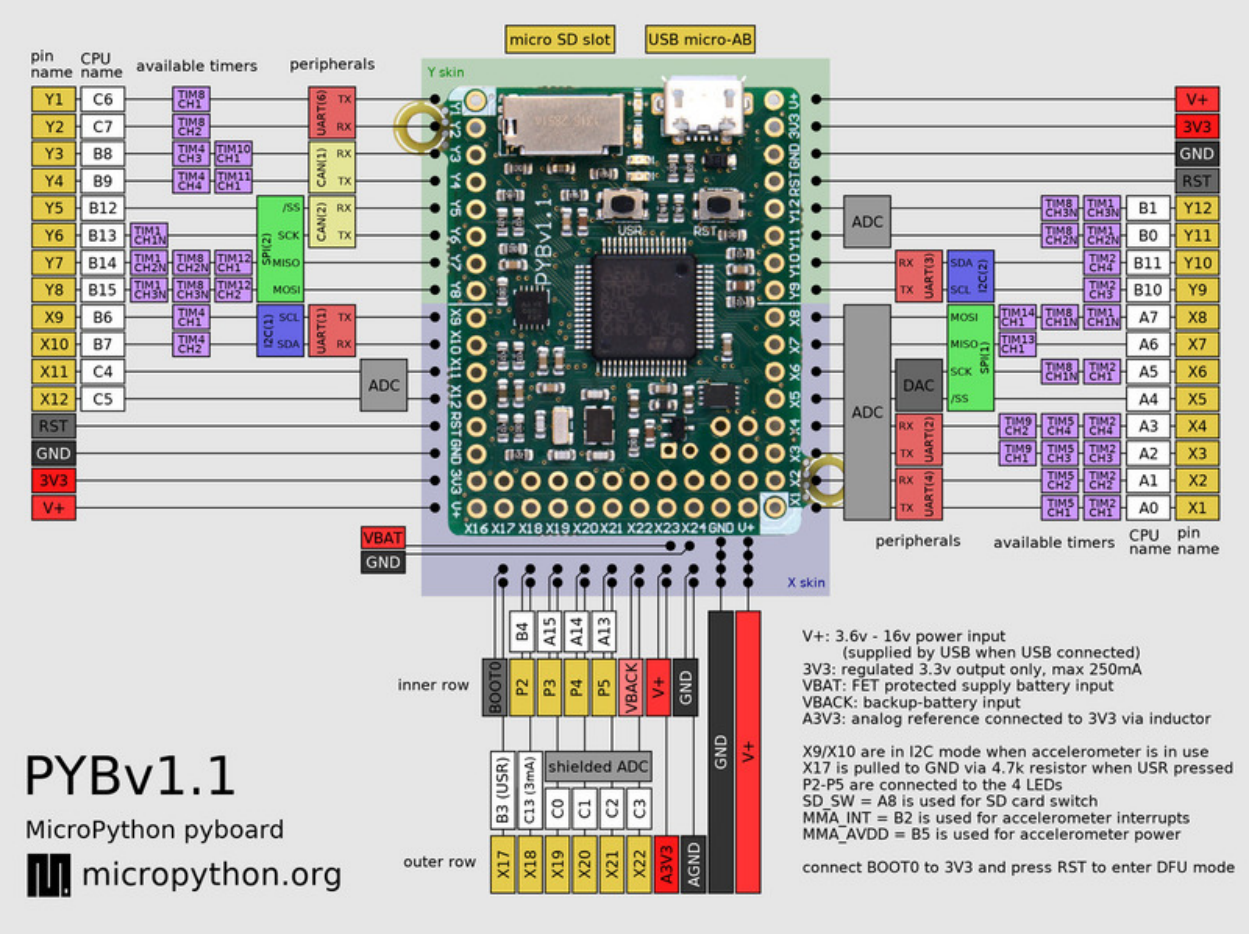
MICROPYTHON LICENSE INFORMATION

CHAPTER

SIX

QUICK REFERENCE FOR THE PYBOARD

The below pinout is for PYBv1.1. You can also view pinouts for other versions of the pyboard: PYBv1.0 or PYBLITEv1.0-AC or PYBLITEv1.0.



Below is a quick reference for the pyboard. If it is your first time working with this board please consider reading the following sections first:

6.1 General information about the pyboard

Contents

6.1.1 Local filesystem and SD card

There is a small internal filesystem (a drive) on the pyboard, called `/flash`, which is stored within the microcontrollers flash memory. If a micro SD card is inserted into the slot, it is available as `/sd`.

When the pyboard boots up, it needs to choose a filesystem to boot from. If there is no SD card, then it uses the internal filesystem `/flash` as the boot filesystem, otherwise, it uses the SD card `/sd`. After the boot, the current directory is set to one of the directories above.

If needed, you can prevent the use of the SD card by creating an empty file called `/flash/SKIPSD`. If this file exists when the pyboard boots up then the SD card will be skipped and the pyboard will always boot from the internal filesystem (in this case the SD card wont be mounted but you can still mount and use it later in your program using `os.mount`).

(Note that on older versions of the board, `/flash` is called `0:/` and `/sd` is called `1:/`).

The boot filesystem is used for 2 things: it is the filesystem from which the `boot.py` and `main.py` files are searched for, and it is the filesystem which is made available on your PC over the USB cable.

The filesystem will be available as a USB flash drive on your PC. You can save files to the drive, and edit `boot.py` and `main.py`.

Remember to eject (on Linux, unmount) the USB drive before you reset your pyboard.

6.1.2 Boot modes

If you power up normally, or press the reset button, the pyboard will boot into standard mode: the `boot.py` file will be executed first, then the USB will be configured, then `main.py` will run.

You can override this boot sequence by holding down the user switch as the board is booting up. Hold down user switch and press reset, and then as you continue to hold the user switch, the LEDs will count in binary. When the LEDs have reached the mode you want, let go of the user switch, the LEDs for the selected mode will flash quickly, and the board will boot.

The modes are:

1. Green LED only, *standard boot*: run `boot.py` then `main.py`.

2. Orange LED only, *safe boot*: dont run any scripts on boot-up.
3. Green and orange LED together, *filesystem reset*: resets the flash filesystem to its factory state, then boots in safe mode.

If your filesystem becomes corrupt, boot into mode 3 to fix it. If resetting the filesystem while plugged into your compute doesnt work, you can try doing the same procedure while the board is plugged into a USB charger, or other USB power supply without data connection.

6.1.3 Errors: flashing LEDs

There are currently 2 kinds of errors that you might see:

1. **If the red and green LEDs flash alternatively, then a Python script** (eg `main.py`) has an error. Use the REPL to debug it.
2. If all 4 LEDs cycle on and off slowly, then there was a hard fault. This cannot be recovered from and you need to do a hard reset.

6.1.4 Guide for using the pyboard with Windows

The following PDF guide gives information about using the pyboard with Windows, including setting up the serial prompt and downloading new firmware using DFU programming: PDF guide.

6.1.5 The pyboard hardware

For the pyboard:

- **v1.1**
  - PYBv1.1 schematics and layout (2.9MiB PDF)
- **v1.0**
  - PYBv1.0 schematics and layout (2.4MiB PDF)
  - PYBv1.0 metric dimensions (360KiB PDF)
  - PYBv1.0 imperial dimensions (360KiB PDF)

For the official skin modules:

- LCD32MKv1.0 schematics (194KiB PDF)
- AMPv1.0 schematics (209KiB PDF)
- LCD160CRv1.0: see *lcd160cr*

6.1.6 Datasheets for the components on the pyboard

- The microcontroller: STM32F405RGT6 (link to manufacturers site)
- The accelerometer: Freescale MMA7660 (800kiB PDF)
- The LDO voltage regulator: Microchip MCP1802 (400kiB PDF)

6.1.7 Datasheets for other components

- The LCD display on the LCD touch-sensor skin: Newhaven Display NHD-C12832A1Z-FSW-FBW-3V3 (460KiB PDF)
- The touch sensor chip on the LCD touch-sensor skin: Freescale MPR121 (280KiB PDF)
- The digital potentiometer on the audio skin: Microchip MCP4541 (2.7MiB PDF)

6.2 MicroPython tutorial for the pyboard

This tutorial is intended to get you started with your pyboard. All you need is a pyboard and a micro-USB cable to connect it to your PC. If it is your first time, it is recommended to follow the tutorial through in the order below.

6.2.1 Introduction to the pyboard

To get the most out of your pyboard, there are a few basic things to understand about how it works.

Caring for your pyboard

Because the pyboard does not have a housing it needs a bit of care:

- Be gentle when plugging/unplugging the USB cable. Whilst the USB connector is soldered through the board and is relatively strong, if it breaks off it can be very difficult to fix.
- Static electricity can shock the components on the pyboard and destroy them. If you experience a lot of static electricity in your area (eg dry and cold climates), take extra care not to shock the pyboard. If your pyboard came in a black plastic box, then this box is the best way to store and carry the pyboard as it is an anti-static box (it is made of a conductive plastic, with conductive foam inside).

As long as you take care of the hardware, you should be okay. Its almost impossible to break the software on the pyboard, so feel free to play around with writing code as much as you like. If the filesystem gets corrupt, see below on how to reset it. In the worst case you might need to reflash the MicroPython software, but that can be done over USB.

Layout of the pyboard

The micro USB connector is on the top right, the micro SD card slot on the top left of the board. There are 4 LEDs between the SD slot and USB connector. The colours are: red on the bottom, then green, orange, and blue on the top. There are 2 switches: the right one is the reset switch, the left is the user switch.

Plugging in and powering on

The pyboard can be powered via USB. Connect it to your PC via a micro USB cable. There is only one way that the cable will fit. Once connected, the green LED on the board should flash quickly.

Powering by an external power source

The pyboard can be powered by a battery or other external power source.

Be sure to connect the positive lead of the power supply to VIN, and ground to GND. There is no polarity protection on the pyboard so you must be careful when connecting anything to VIN.

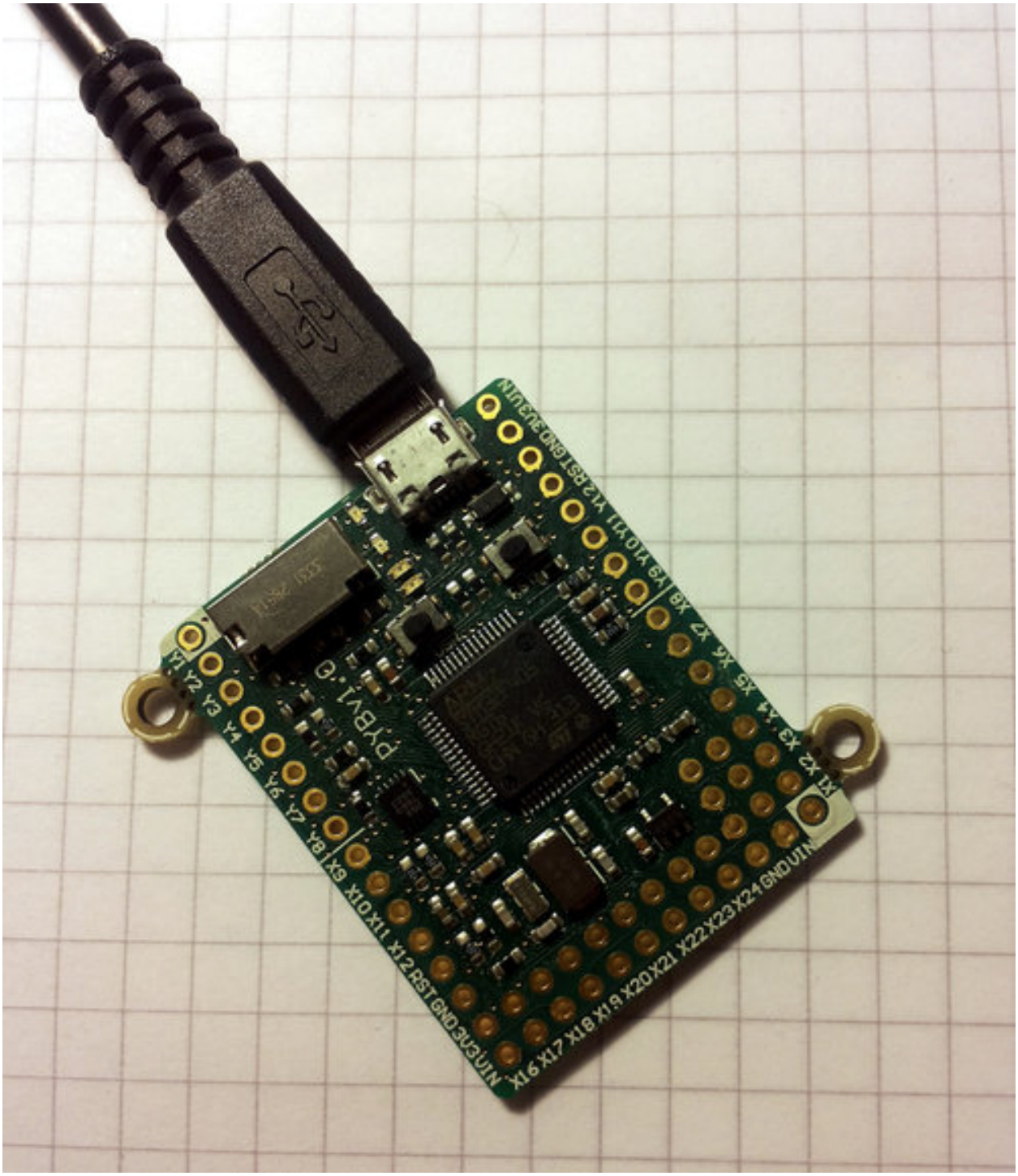
The input voltage must be between 3.6V and 10V.

6.2.2 Running your first script

Lets jump right in and get a Python script running on the pyboard. After all, thats what its all about!

Connecting your pyboard

Connect your pyboard to your PC (Windows, Mac or Linux) with a micro USB cable. There is only one way that the cable will connect, so you cant get it wrong.



When the pyboard is connected to your PC it will power on and enter the start up process (the boot process). The green LED should light up for half a second or less, and when it turns off it means the boot process has completed.

Opening the pyboard USB drive

Your PC should now recognise the pyboard. It depends on the type of PC you have as to what happens next:

- **Windows**: Your pyboard will appear as a removable USB flash drive. Windows may automatically pop-up a window, or you may need to go there using Explorer.

  Windows will also see that the pyboard has a serial device, and it will try to automatically configure this device. If it does, cancel the process. We will get the serial device working in the next tutorial.

- **Mac**: Your pyboard will appear on the desktop as a removable disc. It will probably be called `PYBFLASH`. Click on it to open the pyboard folder.

- **Linux**: Your pyboard will appear as a removable medium. On Ubuntu it will mount automatically and pop-up a window with the pyboard folder. On other Linux distributions, the pyboard may be mounted automatically, or you may need to do it manually. At a terminal command line, type `lsblk` to see a list of connected drives, and then `mount /dev/sdb1` (replace `sdb1` with the appropriate device). You may need to be root to do this.

Okay, so you should now have the pyboard connected as a USB flash drive, and a window (or command line) should be showing the files on the pyboard drive.

The drive you are looking at is known as `/flash` by the pyboard, and should contain the following 4 files:

- **boot.py – the various configuration options for the pyboard.** It is executed when the pyboard boots up.
- **main.py – the Python program to be run.** It is executed after `boot.py`.
- **README.txt – basic information about getting started with the pyboard.** This provides pointers for new users and can be safely deleted.
- **pybcdc.inf – the Windows driver file to configure the serial USB device.** More about this in the next tutorial.

Editing `main.py`

Now we are going to write our Python program, so open the `main.py` file in a text editor. On Windows you can use notepad, or any other editor. On Mac and Linux, use your favourite text editor. With the file open you will see it contains 1 line:

```
# main.py -- put your code here!
```

This line starts with a # character, which means that it is a *comment*. Such lines will not do anything, and are there for you to write notes about your program.

Lets add 2 lines to this `main.py` file, to make it look like this:

```
# main.py -- put your code here!
import pyb
pyb.LED(4).on()
```

The first line we wrote says that we want to use the `pyb` module. This module contains all the functions and classes to control the features of the pyboard.

The second line that we wrote turns the blue LED on: it first gets the `LED` class from the `pyb` module, creates LED number 4 (the blue LED), and then turns it on.

Resetting the pyboard

To run this little script, you need to first save and close the `main.py` file, and then eject (or unmount) the pyboard USB drive. Do this like you would a normal USB flash drive.

When the drive is safely ejected/unmounted you can get to the fun part: press the RST switch on the pyboard to reset and run your script. The RST switch is the small black button just below the USB connector on the board, on the right edge.

When you press RST the green LED will flash quickly, and then the blue LED should turn on and stay on.

Congratulations! You have written and run your very first MicroPython program!

6.2.3 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 pyboard. Using the REPL is by far the easiest way to test out your code and run commands. You can use the REPL in addition to writing scripts in `main.py`.

To use the REPL, you must connect to the serial USB device on the pyboard. How you do this depends on your operating system.

Windows

You need to install the pyboard driver to use the serial USB device. The driver is on the pyboards USB flash drive, and is called `pybcdc.inf`.

To install this driver you need to go to Device Manager for your computer, find the pyboard in the list of devices (it should have a warning sign next to it because its not working yet), right click on the pyboard device, select Properties, then Install Driver. You need to then select the option to find the driver manually (dont use Windows auto update), navigate to the pyboards USB drive, and select that. It should then install. After installing, go back to the Device Manager to find the installed pyboard, and see which COM port it is (eg COM4). More comprehensive instructions can be found in the Guide for pyboard on Windows (PDF). Please consult this guide if you are having problems installing the driver.

You now need to run your terminal program. You can use HyperTerminal if you have it installed, or download the free program PuTTY: putty.exe. Using your serial program you must connect to the COM port that you found in the previous step. With PuTTY, click on Session in the left-hand panel, then click the Serial radio button on the right, then enter you COM port (eg COM4) in the Serial Line box. Finally, click the Open button.

Mac OS X

Open a terminal and run:

```
screen /dev/tty.usbmodem*
```

When you are finished and want to exit screen, type CTRL-A CTRL-\.

Linux

Open a terminal and run:

```
screen /dev/ttyACM0
```

You can also try `picocom` or `minicom` instead of screen. You may have to use `/dev/ttyACM1` or a higher number for `ttyACM`. And, you may need to give yourself the correct permissions to access this devices (eg group `uucp` or `dialout`, or use sudo).

Using the REPL prompt

Now lets try running some MicroPython code directly on the pyboard.

With your serial program open (PuTTY, screen, picocom, etc) you may see a blank screen with a flashing cursor. Press Enter and you should be presented with a MicroPython prompt, i.e. >>>. Lets make sure it is working with the obligatory test:

```
>>> print("hello pyboard!")
hello pyboard!
```

In the above, you should not type in the >>> characters. They are there to indicate that you should type the text after it at the prompt. In the end, once you have entered the text `print("hello pyboard!")` and pressed Enter, the output on your screen should look like it does above.

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

If any of this is not working you can try either a hard reset or a soft reset; see below.

Go ahead and try typing in some other commands. For example:

```
>>> pyb.LED(1).on()
>>> pyb.LED(2).on()
>>> 1 + 2
3
>>> 1 / 2
0.5
>>> 20 * 'py'
'pypypypypypypypypypypypypypypypypypypypy'
```

Resetting the board

If something goes wrong, you can reset the board in two ways. The first is to press CTRL-D at the MicroPython prompt, which performs a soft reset. You will see a message something like

```
>>>
MPY: sync filesystems
MPY: soft reboot
Micro Python v1.0 on 2014-05-03; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>>
```

If that isnt working you can perform a hard reset (turn-it-off-and-on-again) by pressing the RST switch (the small black button closest to the micro-USB socket on the board). This will end your session, disconnecting whatever program (PuTTY, screen, etc) that you used to connect to the pyboard.

If you are going to do a hard-reset, its recommended to first close your serial program and eject/unmount the pyboard drive.

6.2.4 Turning on LEDs and basic Python concepts

The easiest thing to do on the pyboard is to turn on the LEDs attached to the board. Connect the board, and log in as described in tutorial 1. We will start by turning and LED on in the interpreter, type the following

```
>>> myled = pyb.LED(1)
>>> myled.on()
>>> myled.off()
```

These commands turn the LED on and off.

This is all very well but we would like this process to be automated. Open the file MAIN.PY on the pyboard in your favourite text editor. Write or paste the following lines into the file. If you are new to python, then make sure you get the indentation correct since this matters!

```
led = pyb.LED(2)
while True:
    led.toggle()
    pyb.delay(1000)
```

When you save, the red light on the pyboard should turn on for about a second. To run the script, do a soft reset (CTRL-D). The pyboard will then restart and you should see a green light continuously flashing on and off. Success, the first step on your path to building an army of evil robots! When you are bored of the annoying flashing light then press CTRL-C at your terminal to stop it running.

So what does this code do? First we need some terminology. Python is an object-oriented language, almost everything in python is a *class* and when you create an instance of a class you get an *object*. Classes have *methods* associated to them. A method (also called a member function) is used to interact with or control the object.

The first line of code creates an LED object which we have then called led. When we create the object, it takes a single parameter which must be between 1 and 4, corresponding to the 4 LEDs on the board. The pyb.LED class has three important member functions that we will use: on(), off() and toggle(). The other function that we use is pyb.delay() this simply waits for a given time in milliseconds. Once we have created the LED object, the statement while True: creates an infinite loop which toggles the led between on and off and waits for 1 second.

Exercise: Try changing the time between toggling the led and turning on a different LED.

Exercise: Connect to the pyboard directly, create a pyb.LED object and turn it on using the on() method.

A Disco on your pyboard

So far we have only used a single LED but the pyboard has 4 available. Lets start by creating an object for each LED so we can control each of them. We do that by creating a list of LEDS with a list comprehension.

```
leds = [pyb.LED(i) for i in range(1,5)]
```

If you call pyb.LED() with a number that isnt 1,2,3,4 you will get an error message. Next we will set up an infinite loop that cycles through each of the LEDs turning them on and off.

```
n = 0
while True:
  n = (n + 1) % 4
```

(continues on next page)

(continued from previous page)

```
  leds[n].toggle()
  pyb.delay(50)
```

Here, n keeps track of the current LED and every time the loop is executed we cycle to the next n (the % sign is a modulus operator that keeps n between 0 and 3.) Then we access the nth LED and toggle it. If you run this you should see each of the LEDs turning on then all turning off again in sequence.

One problem you might find is that if you stop the script and then start it again that the LEDs are stuck on from the previous run, ruining our carefully choreographed disco. We can fix this by turning all the LEDs off when we initialise the script and then using a try/finally block. When you press CTRL-C, MicroPython generates a VCPInterrupt exception. Exceptions normally mean something has gone wrong and you can use a try: command to catch an exception. In this case it is just the user interrupting the script, so we dont need to catch the error but just tell MicroPython what to do when we exit. The finally block does this, and we use it to make sure all the LEDs are off. The full code is:

```
leds = [pyb.LED(i) for i in range(1,5)]
for l in leds:
    l.off()

n = 0
try:
   while True:
      n = (n + 1) % 4
      leds[n].toggle()
      pyb.delay(50)
finally:
    for l in leds:
        l.off()
```

The Special LEDs

The yellow and blue LEDs are special. As well as turning them on and off, you can control their intensity using the intensity() method. This takes a number between 0 and 255 that determines how bright it is. The following script makes the blue LED gradually brighter then turns it off again.

```
led = pyb.LED(4)
intensity = 0
while True:
    intensity = (intensity + 1) % 255
    led.intensity(intensity)
    pyb.delay(20)
```

You can call intensity() on LEDs 1 and 2 but they can only be off or on. 0 sets them off and any other number up to 255 turns them on.

6.2.5 Switches, callbacks and interrupts

The pyboard has 2 small switches, labelled USR and RST. The RST switch is a hard-reset switch, and if you press it then it restarts the pyboard from scratch, equivalent to turning the power off then back on.

The USR switch is for general use, and is controlled via a Switch object. To make a switch object do:

```
>>> sw = pyb.Switch()
```

Remember that you may need to type `import pyb` if you get an error that the name `pyb` does not exist.

With the switch object you can get its status:

```
>>> sw.value()
False
```

This will print `False` if the switch is not held, or `True` if it is held. Try holding the USR switch down while running the above command.

There is also a shorthand notation to get the switch status, by calling the switch object:

```
>>> sw()
False
```

Switch callbacks

The switch is a very simple object, but it does have one advanced feature: the `sw.callback()` function. The callback function sets up something to run when the switch is pressed, and uses an interrupt. Its probably best to start with an example before understanding how interrupts work. Try running the following at the prompt:

```
>>> sw.callback(lambda:print('press!'))
```

This tells the switch to print `press!` each time the switch is pressed down. Go ahead and try it: press the USR switch and watch the output on your PC. Note that this print will interrupt anything you are typing, and is an example of an interrupt routine running asynchronously.

As another example try:

```
>>> sw.callback(lambda:pyb.LED(1).toggle())
```

This will toggle the red LED each time the switch is pressed. And it will even work while other code is running.

To disable the switch callback, pass `None` to the callback function:

```
>>> sw.callback(None)
```

You can pass any function (that takes zero arguments) to the switch callback. Above we used the `lambda` feature of Python to create an anonymous function on the fly. But we could equally do:

```
>>> def f():
...   pyb.LED(1).toggle()
...
>>> sw.callback(f)
```

This creates a function called `f` and assigns it to the switch callback. You can do things this way when your function is more complicated than a `lambda` will allow.

Note that your callback functions must not allocate any memory (for example they cannot create a tuple or list). Callback functions should be relatively simple. If you need to make a list, make it beforehand and store it in a global variable (or make it local and close over it). If you need to do a long, complicated calculation, then use the callback to set a flag which some other code then responds to.

Technical details of interrupts

Lets step through the details of what is happening with the switch callback. When you register a function with `sw.callback()`, the switch sets up an external interrupt trigger (falling edge) on the pin that the switch is connected to. This means that the microcontroller will listen on the pin for any changes, and the following will occur:

1. When the switch is pressed a change occurs on the pin (the pin goes from low to high), and the microcontroller registers this change.
2. The microcontroller finishes executing the current machine instruction, stops execution, and saves its current state (pushes the registers on the stack). This has the effect of pausing any code, for example your running Python script.
3. The microcontroller starts executing the special interrupt handler associated with the switchs external trigger. This interrupt handler gets the function that you registered with `sw.callback()` and executes it.
4. Your callback function is executed until it finishes, returning control to the switch interrupt handler.
5. The switch interrupt handler returns, and the microcontroller is notified that the interrupt has been dealt with.
6. The microcontroller restores the state that it saved in step 2.
7. Execution continues of the code that was running at the beginning. Apart from the pause, this code does not notice that it was interrupted.

The above sequence of events gets a bit more complicated when multiple interrupts occur at the same time. In that case, the interrupt with the highest priority goes first, then the others in order of their priority. The switch interrupt is set at the lowest priority.

Further reading

For further information about using hardware interrupts see *writing interrupt handlers*.

6.2.6 The accelerometer

Here you will learn how to read the accelerometer and signal using LEDs states like tilt left and tilt right.

Using the accelerometer

The pyboard has an accelerometer (a tiny mass on a tiny spring) that can be used to detect the angle of the board and motion. There is a different sensor for each of the x, y, z directions. To get the value of the accelerometer, create a pyb.Accel() object and then call the x() method.

```
>>> accel = pyb.Accel()
>>> accel.x()
7
```

This returns a signed integer with a value between around -30 and 30. Note that the measurement is very noisy, this means that even if you keep the board perfectly still there will be some variation in the number that you measure. Because of this, you shouldnt use the exact value of the x() method but see if it is in a certain range.

We will start by using the accelerometer to turn on a light if it is not flat.

```
accel = pyb.Accel()
light = pyb.LED(3)
SENSITIVITY = 3

while True:
    x = accel.x()
    if abs(x) > SENSITIVITY:
        light.on()
    else:
        light.off()

    pyb.delay(100)
```

We create Accel and LED objects, then get the value of the x direction of the accelerometer. If the magnitude of x is bigger than a certain value `SENSITIVITY`, then the LED turns on, otherwise it turns off. The loop has a small `pyb.delay()` otherwise the LED flashes annoyingly when the value of x is close to `SENSITIVITY`. Try running this on the pyboard and tilt the board left and right to make the LED turn on and off.

Exercise: Change the above script so that the blue LED gets brighter the more you tilt the pyboard. HINT: You will need to rescale the values, intensity goes from 0-255.

Making a spirit level

The example above is only sensitive to the angle in the x direction but if we use the `y()` value and more LEDs we can turn the pyboard into a spirit level.

```
xlights = (pyb.LED(2), pyb.LED(3))
ylights = (pyb.LED(1), pyb.LED(4))

accel = pyb.Accel()
SENSITIVITY = 3

while True:
    x = accel.x()
    if x > SENSITIVITY:
        xlights[0].on()
        xlights[1].off()
    elif x < -SENSITIVITY:
        xlights[1].on()
        xlights[0].off()
    else:
        xlights[0].off()
        xlights[1].off()

    y = accel.y()
    if y > SENSITIVITY:
        ylights[0].on()
        ylights[1].off()
    elif y < -SENSITIVITY:
        ylights[1].on()
        ylights[0].off()
```

(continues on next page)

(continued from previous page)

```
    else:
        ylights[0].off()
        ylights[1].off()

    pyb.delay(100)
```

We start by creating a tuple of LED objects for the x and y directions. Tuples are immutable objects in python which means they cant be modified once they are created. We then proceed as before but turn on a different LED for positive and negative x values. We then do the same for the y direction. This isnt particularly sophisticated but it does the job. Run this on your pyboard and you should see different LEDs turning on depending on how you tilt the board.

6.2.7 Safe mode and factory reset

If something goes wrong with your pyboard, dont panic! It is almost impossible for you to break the pyboard by programming the wrong thing.

The first thing to try is to enter safe mode: this temporarily skips execution of `boot.py` and `main.py` and gives default USB settings.

If you have problems with the filesystem you can do a factory reset, which restores the filesystem to its original state.

Safe mode

To enter safe mode, do the following steps:

1. Connect the pyboard to USB so it powers up.
2. Hold down the USR switch.
3. While still holding down USR, press and release the RST switch.
4. The LEDs will then cycle green to orange to green+orange and back again.
5. Keep holding down USR until *only the orange LED is lit*, and then let go of the USR switch.
6. The orange LED should flash quickly 4 times, and then turn off.
7. You are now in safe mode.

In safe mode, the `boot.py` and `main.py` files are not executed, and so the pyboard boots up with default settings. This means you now have access to the filesystem (the USB drive should appear), and you can edit `boot.py` and `main.py` to fix any problems.

Entering safe mode is temporary, and does not make any changes to the files on the pyboard.

Factory reset the filesystem

If you pyboards filesystem gets corrupted (for example, you forgot to eject/unmount it), or you have some code in `boot.py` or `main.py` which you cant escape from, then you can reset the filesystem.

Resetting the filesystem deletes all files on the internal pyboard storage (not the SD card), and restores the files `boot.py`, `main.py`, `README.txt` and `pybcdc.inf` back to their original state.

To do a factory reset of the filesystem you follow a similar procedure as you did to enter safe mode, but release USR on green+orange:

1. Connect the pyboard to USB so it powers up.

2. Hold down the USR switch.
3. While still holding down USR, press and release the RST switch.
4. The LEDs will then cycle green to orange to green+orange and back again.
5. Keep holding down USR until *both the green and orange LEDs are lit*, and then let go of the USR switch.
6. The green and orange LEDs should flash quickly 4 times.
7. The red LED will turn on (so red, green and orange are now on).
8. The pyboard is now resetting the filesystem (this takes a few seconds).
9. The LEDs all turn off.
10. You now have a reset filesystem, and are in safe mode.
11. Press and release the RST switch to boot normally.

6.2.8 Making the pyboard act as a USB mouse

The pyboard is a USB device, and can configured to act as a mouse instead of the default USB flash drive.

To do this we must first edit the `boot.py` file to change the USB configuration. If you have not yet touched your `boot.py` file then it will look something like this:

```
# boot.py -- run on boot to configure USB and filesystem
# Put app code in main.py

import pyb
#pyb.main('main.py') # main script to run after this one
#pyb.usb_mode('VCP+MSC') # act as a serial and a storage device
#pyb.usb_mode('VCP+HID') # act as a serial device and a mouse
```

To enable the mouse mode, uncomment the last line of the file, to make it look like:

```
pyb.usb_mode('VCP+HID') # act as a serial device and a mouse
```

If you already changed your `boot.py` file, then the minimum code it needs to work is:

```
import pyb
pyb.usb_mode('VCP+HID')
```

This tells the pyboard to configure itself as a VCP (Virtual COM Port, ie serial port) and HID (human interface device, in our case a mouse) USB device when it boots up.

Eject/unmount the pyboard drive and reset it using the RST switch. Your PC should now detect the pyboard as a mouse!

Sending mouse events by hand

To get the py-mouse to do anything we need to send mouse events to the PC. We will first do this manually using the REPL prompt. Connect to your pyboard using your serial program and type the following (no need to type the # and text following it):

```
>>> hid = pyb.USB_HID()
>>> hid.send((0, 100, 0, 0)) # (button status, x-direction, y-direction, scroll)
```

Your mouse should move 100 pixels to the right! In the command above you are sending 4 pieces of information: **button status**, **x-direction**, **y-direction**, and **scroll**. The number 100 is telling the PC that the mouse moved 100 pixels in the x direction.

Lets make the mouse oscillate left and right:

```
>>> import math
>>> def osc(n, d):
...   for i in range(n):
...     hid.send((0, int(20 * math.sin(i / 10)), 0, 0))
...     pyb.delay(d)
...
>>> osc(100, 50)
```

The first argument to the function `osc` is the number of mouse events to send, and the second argument is the delay (in milliseconds) between events. Try playing around with different numbers.

Exercise: make the mouse go around in a circle.

Making a mouse with the accelerometer

Now lets make the mouse move based on the angle of the pyboard, using the accelerometer. The following code can be typed directly at the REPL prompt, or put in the `main.py` file. Here, well put in in `main.py` because to do that we will learn how to go into safe mode.

At the moment the pyboard is acting as a serial USB device and an HID (a mouse). So you cannot access the filesystem to edit your `main.py` file.

You also cant edit your `boot.py` to get out of HID-mode and back to normal mode with a USB drive

To get around this we need to go into *safe mode*. This was described in the [safe mode tutorial](tut-reset), but we repeat the instructions here:

1. Hold down the USR switch.
2. While still holding down USR, press and release the RST switch.
3. The LEDs will then cycle green to orange to green+orange and back again.
4. Keep holding down USR until *only the orange LED is lit*, and then let go of the USR switch.
5. The orange LED should flash quickly 4 times, and then turn off.
6. You are now in safe mode.

In safe mode, the `boot.py` and `main.py` files are not executed, and so the pyboard boots up with default settings. This means you now have access to the filesystem (the USB drive should appear), and you can edit `main.py`. (Leave `boot.py` as-is, because we still want to go back to HID-mode after we finish editing `main.py`.)

In `main.py` put the following code:

```
import pyb

switch = pyb.Switch()
accel = pyb.Accel()
hid = pyb.USB_HID()

while not switch():
    hid.send((0, accel.x(), accel.y(), 0))
    pyb.delay(20)
```

Save your file, eject/unmount your pyboard drive, and reset it using the RST switch. It should now act as a mouse, and the angle of the board will move the mouse around. Try it out, and see if you can make the mouse stand still!

Press the USR switch to stop the mouse motion.

Youll note that the y-axis is inverted. Thats easy to fix: just put a minus sign in front of the y-coordinate in the `hid.send()` line above.

Restoring your pyboard to normal

If you leave your pyboard as-is, itll behave as a mouse everytime you plug it in. You probably want to change it back to normal. To do this you need to first enter safe mode (see above), and then edit the `boot.py` file. In the `boot.py` file, comment out (put a # in front of) the line with the `VCP+HID` setting, so it looks like:

```
#pyb.usb_mode('VCP+HID') # act as a serial device and a mouse
```

Save your file, eject/unmount the drive, and reset the pyboard. It is now back to normal operating mode.

6.2.9 The Timers

The pyboard has 14 timers which each consist of an independent counter running at a user-defined frequency. They can be set up to run a function at specific intervals. The 14 timers are numbered 1 through 14, but 3 is reserved for internal use, and 5 and 6 are used for servo and ADC/DAC control. Avoid using these timers if possible.

Lets create a timer object:

```
>>> tim = pyb.Timer(4)
```

Now lets see what we just created:

```
>>> tim
Timer(4)
```

The pyboard is telling us that `tim` is attached to timer number 4, but its not yet initialised. So lets initialise it to trigger at 10 Hz (thats 10 times per second):

```
>>> tim.init(freq=10)
```

Now that its initialised, we can see some information about the timer:

```
>>> tim
Timer(4, prescaler=624, period=13439, mode=UP, div=1)
```

The information means that this timer is set to run at the peripheral clock speed divided by 624+1, and it will count from 0 up to 13439, at which point it triggers an interrupt, and then starts counting again from 0. These numbers are set to make the timer trigger at 10 Hz: the source frequency of the timer is 84MHz (found by running `tim.source_freq()`) so we get 84MHz / 625 / 13440 = 10Hz.

Timer counter

So what can we do with our timer? The most basic thing is to get the current value of its counter:

```
>>> tim.counter()
21504
```

This counter will continuously change, and counts up.

Timer callbacks

The next thing we can do is register a callback function for the timer to execute when it triggers (see the *switch tutorial* for an introduction to callback functions):

```
>>> tim.callback(lambda t:pyb.LED(1).toggle())
```

This should start the red LED flashing right away. It will be flashing at 5 Hz (2 toggles are needed for 1 flash, so toggling at 10 Hz makes it flash at 5 Hz). You can change the frequency by re-initialising the timer:

```
>>> tim.init(freq=20)
```

You can disable the callback by passing it the value `None`:

```
>>> tim.callback(None)
```

The function that you pass to callback must take 1 argument, which is the timer object that triggered. This allows you to control the timer from within the callback function.

We can create 2 timers and run them independently:

```
>>> tim4 = pyb.Timer(4, freq=10)
>>> tim7 = pyb.Timer(7, freq=20)
>>> tim4.callback(lambda t: pyb.LED(1).toggle())
>>> tim7.callback(lambda t: pyb.LED(2).toggle())
```

Because the callbacks are proper hardware interrupts, we can continue to use the pyboard for other things while these timers are running.

Making a microsecond counter

You can use a timer to create a microsecond counter, which might be useful when you are doing something which requires accurate timing. We will use timer 2 for this, since timer 2 has a 32-bit counter (so does timer 5, but if you use timer 5 then you cant use the Servo driver at the same time).

We set up timer 2 as follows:

```
>>> micros = pyb.Timer(2, prescaler=83, period=0x3fffffff)
```

The prescaler is set at 83, which makes this timer count at 1 MHz. This is because the CPU clock, running at 168 MHz, is divided by 2 and then by prescaler+1, giving a frequency of 168 MHz/2/(83+1)=1 MHz for timer 2. The period is set to a large number so that the timer can count up to a large number before wrapping back around to zero. In this case it will take about 17 minutes before it cycles back to zero.

To use this timer, its best to first reset it to 0:

```
>>> micros.counter(0)
```

and then perform your timing:

```
>>> start_micros = micros.counter()

... do some stuff ...

>>> end_micros = micros.counter()
```

6.2.10 Inline assembler

Here you will learn how to write inline assembler in MicroPython.

Note: this is an advanced tutorial, intended for those who already know a bit about microcontrollers and assembly language.

MicroPython includes an inline assembler. It allows you to write assembly routines as a Python function, and you can call them as you would a normal Python function.

Returning a value

Inline assembler functions are denoted by a special function decorator. Lets start with the simplest example:

```
@micropython.asm_thumb
def fun():
    movw(r0, 42)
```

You can enter this in a script or at the REPL. This function takes no arguments and returns the number 42. `r0` is a register, and the value in this register when the function returns is the value that is returned. MicroPython always interprets the `r0` as an integer, and converts it to an integer object for the caller.

If you run `print(fun())` you will see it print out 42.

Accessing peripherals

For something a bit more complicated, lets turn on an LED:

```
@micropython.asm_thumb
def led_on():
    movwt(r0, stm.GPIOA)
    movw(r1, 1 << 13)
    strh(r1, [r0, stm.GPIO_BSRRL])
```

This code uses a few new concepts:

- `stm` is a module which provides a set of constants for easy access to the registers of the pyboards microcontroller. Try running `import stm` and then `help(stm)` at the REPL. It will give you a list of all the available constants.

- `stm.GPIOA` is the address in memory of the GPIOA peripheral. On the pyboard, the red LED is on port A, pin PA13.
- `movwt` moves a 32-bit number into a register. It is a convenience function that turns into 2 thumb instructions: `movw` followed by `movt`. The `movt` also shifts the immediate value right by 16 bits.
- `strh` stores a half-word (16 bits). The instruction above stores the lower 16-bits of `r1` into the memory location `r0 + stm.GPIO_BSRRL`. This has the effect of setting high all those pins on port A for which the corresponding bit in `r0` is set. In our example above, the 13th bit in `r0` is set, so PA13 is pulled high. This turns on the red LED.

Accepting arguments

Inline assembler functions can accept up to 4 arguments. If they are used, they must be named `r0`, `r1`, `r2` and `r3` to reflect the registers and the calling conventions.

Here is a function that adds its arguments:

```
@micropython.asm_thumb
def asm_add(r0, r1):
    add(r0, r0, r1)
```

This performs the computation `r0 = r0 + r1`. Since the result is put in `r0`, that is what is returned. Try `asm_add(1, 2)`, it should return 3.

Loops

We can assign labels with `label(my_label)`, and branch to them using `b(my_label)`, or a conditional branch like `bgt(my_label)`.

The following example flashes the green LED. It flashes it `r0` times.

```
@micropython.asm_thumb
def flash_led(r0):
    # get the GPIOA address in r1
    movwt(r1, stm.GPIOA)

    # get the bit mask for PA14 (the pin LED #2 is on)
    movw(r2, 1 << 14)

    b(loop_entry)

    label(loop1)

    # turn LED on
    strh(r2, [r1, stm.GPIO_BSRRL])

    # delay for a bit
    movwt(r4, 5599900)
    label(delay_on)
    sub(r4, r4, 1)
    cmp(r4, 0)
    bgt(delay_on)

    # turn LED off
```

(continues on next page)

(continued from previous page)

```
    strh(r2, [r1, stm.GPIO_BSRRH])

    # delay for a bit
    movwt(r4, 5599900)
    label(delay_off)
    sub(r4, r4, 1)
    cmp(r4, 0)
    bgt(delay_off)

    # loop r0 times
    sub(r0, r0, 1)
    label(loop_entry)
    cmp(r0, 0)
    bgt(loop1)
```

Further reading

For further information about supported instructions of the inline assembler, see the *reference documentation*.

6.2.11 Power control

pyb.wfi() is used to reduce power consumption while waiting for an event such as an interrupt. You would use it in the following situation:

```
while True:
    do_some_processing()
    pyb.wfi()
```

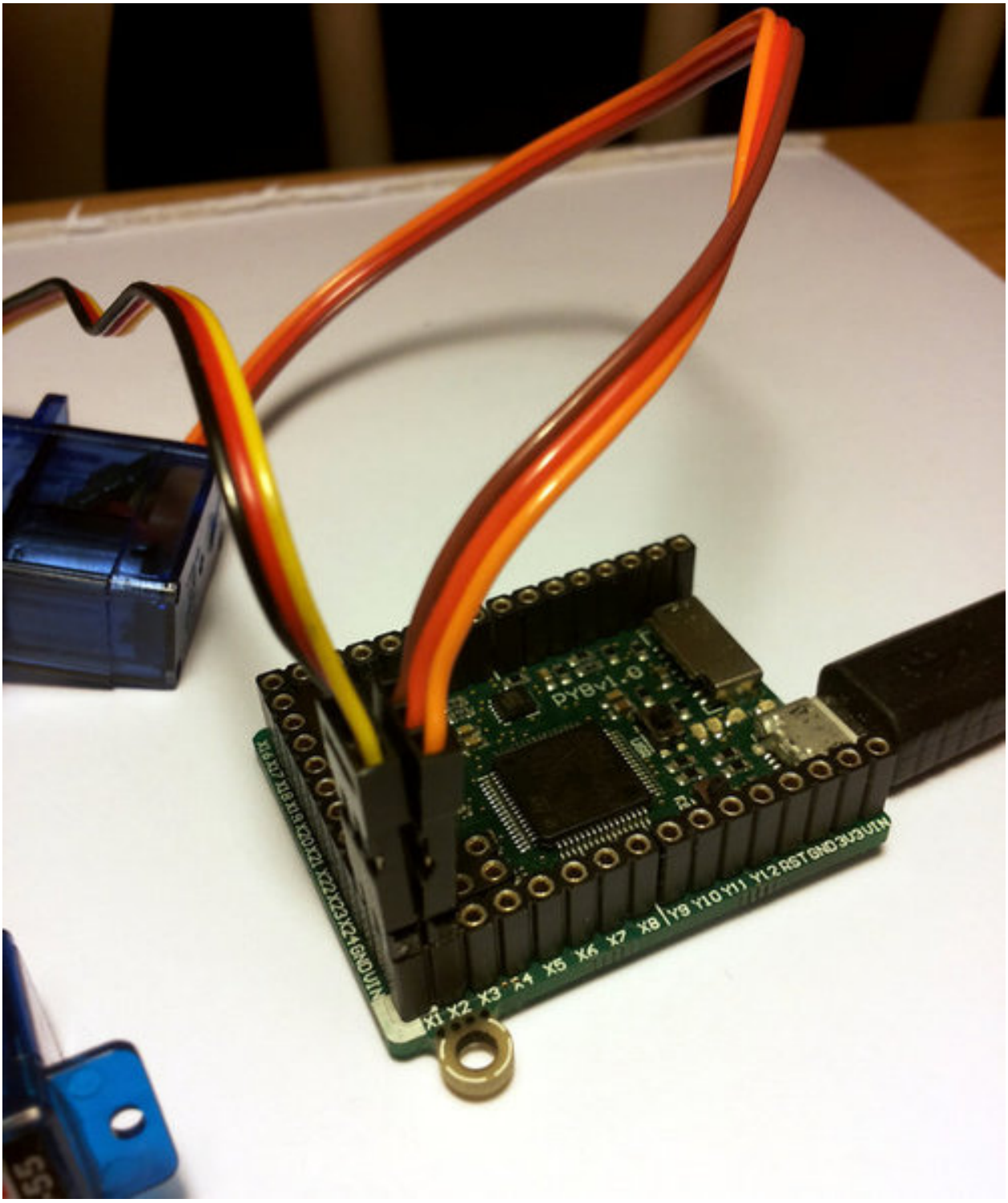
Control the frequency using *pyb.freq()*:

```
pyb.freq(30000000) # set CPU frequency to 30MHz
```

6.2.12 Tutorials requiring extra components

Controlling hobby servo motors

There are 4 dedicated connection points on the pyboard for connecting up hobby servo motors (see eg Wikipedia). These motors have 3 wires: ground, power and signal. On the pyboard you can connect them in the bottom right corner, with the signal pin on the far right. Pins X1, X2, X3 and X4 are the 4 dedicated servo signal pins.



In this picture there are male-male double adaptors to connect the servos to the header pins on the pyboard.

The ground wire on a servo is usually the darkest coloured one, either black or dark brown. The power wire will most likely be red.

The power pin for the servos (labelled VIN) is connected directly to the input power source of the pyboard. When powered via USB, VIN is powered through a diode by the 5V USB power line. Connect to USB, the pyboard can power at least 4 small to medium sized servo motors.

If using a battery to power the pyboard and run servo motors, make sure it is not greater than 6V, since this is the maximum voltage most servo motors can take. (Some motors take only up to 4.8V, so check what type you are using.)

Creating a Servo object

Plug in a servo to position 1 (the one with pin X1) and create a servo object using:

```
>>> servo1 = pyb.Servo(1)
```

To change the angle of the servo use the `angle` method:

```
>>> servo1.angle(45)
>>> servo1.angle(-60)
```

The angle here is measured in degrees, and ranges from about -90 to +90, depending on the motor. Calling `angle` without parameters will return the current angle:

```
>>> servo1.angle()
-60
```

Note that for some angles, the returned angle is not exactly the same as the angle you set, due to rounding errors in setting the pulse width.

You can pass a second parameter to the `angle` method, which specifies how long to take (in milliseconds) to reach the desired angle. For example, to take 1 second (1000 milliseconds) to go from the current position to 50 degrees, use

```
>>> servo1.angle(50, 1000)
```

This command will return straight away and the servo will continue to move to the desired angle, and stop when it gets there. You can use this feature as a speed control, or to synchronise 2 or more servo motors. If we have another servo motor (`servo2 = pyb.Servo(2)`) then we can do

```
>>> servo1.angle(-45, 2000); servo2.angle(60, 2000)
```

This will move the servos together, making them both take 2 seconds to reach their final angles.

Note: the semicolon between the 2 expressions above is used so that they are executed one after the other when you press enter at the REPL prompt. In a script you dont need to do this, you can just write them one line after the other.

Continuous rotation servos

So far we have been using standard servos that move to a specific angle and stay at that angle. These servo motors are useful to create joints of a robot, or things like pan-tilt mechanisms. Internally, the motor has a variable resistor (potentiometer) which measures the current angle and applies power to the motor proportional to how far it is from the desired angle. The desired angle is set by the width of a high-pulse on the servo signal wire. A pulse width of 1500 microsecond corresponds to the centre position (0 degrees). The pulses are sent at 50 Hz, ie 50 pulses per second.

You can also get **continuous rotation** servo motors which turn continuously clockwise or counterclockwise. The direction and speed of rotation is set by the pulse width on the signal wire. A pulse width of 1500 microseconds corresponds to a stopped motor. A pulse width smaller or larger than this means rotate one way or the other, at a given speed.

On the pyboard, the servo object for a continuous rotation motor is the same as before. In fact, using `angle` you can set the speed. But to make it easier to understand what is intended, there is another method called `speed` which sets the speed:

```
>>> servo1.speed(30)
```

`speed` has the same functionality as `angle`: you can get the speed, set it, and set it with a time to reach the final speed.

```
>>> servo1.speed()
30
>>> servo1.speed(-20)
>>> servo1.speed(0, 2000)
```

The final command above will set the motor to stop, but take 2 seconds to do it. This is essentially a control over the acceleration of the continuous servo.

A servo speed of 100 (or -100) is considered maximum speed, but actually you can go a bit faster than that, depending on the particular motor.

The only difference between the `angle` and `speed` methods (apart from the name) is the way the input numbers (angle or speed) are converted to a pulse width.

Calibration

The conversion from angle or speed to pulse width is done by the servo object using its calibration values. To get the current calibration, use

```
>>> servo1.calibration()
(640, 2420, 1500, 2470, 2200)
```

There are 5 numbers here, which have meaning:

1. Minimum pulse width; the smallest pulse width that the servo accepts.
2. Maximum pulse width; the largest pulse width that the servo accepts.
3. Centre pulse width; the pulse width that puts the servo at 0 degrees or 0 speed.
4. The pulse width corresponding to 90 degrees. This sets the conversion in the method `angle` of angle to pulse width.
5. The pulse width corresponding to a speed of 100. This sets the conversion in the method `speed` of speed to pulse width.

You can recalibrate the servo (change its default values) by using:

```
>>> servo1.calibration(700, 2400, 1510, 2500, 2000)
```

Of course, you would change the above values to suit your particular servo motor.

Fading LEDs

In addition to turning LEDs on and off, it is also possible to control the brightness of an LED using Pulse-Width Modulation (PWM), a common technique for obtaining variable output from a digital pin. This allows us to fade an LED:

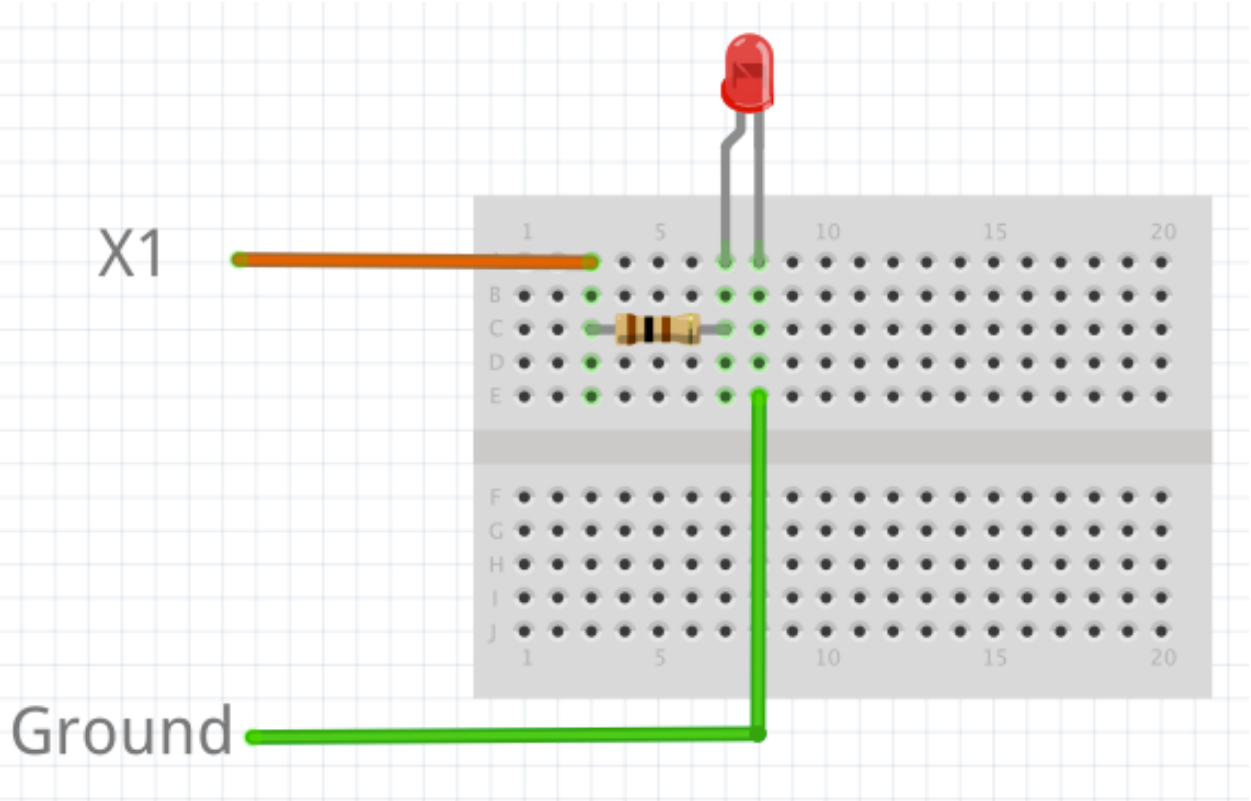
Components

You will need:

- Standard 5 or 3 mm LED
- 100 Ohm resistor
- Wires
- Breadboard (optional, but makes things easier)

Connecting Things Up

For this tutorial, we will use the `X1` pin. Connect one end of the resistor to `X1`, and the other end to the **anode** of the LED, which is the longer leg. Connect the **cathode** of the LED to ground.



Code

By examining the *Quick reference for the pyboard*, we see that `X1` is connected to channel 1 of timer 5 (`TIM5 CH1`). Therefore we will first create a `Timer` object for timer 5, then create a `TimerChannel` object for channel 1:

```
from pyb import Timer
from time import sleep

# timer 5 will be created with a frequency of 100 Hz
tim = pyb.Timer(5, freq=100)
tchannel = tim.channel(1, Timer.PWM, pin=pyb.Pin.board.X1, pulse_width=0)
```

Brightness of the LED in PWM is controlled by controlling the pulse-width, that is the amount of time the LED is on every cycle. With a timer frequency of 100 Hz, each cycle takes 0.01 second, or 10 ms.

To achieve the fading effect shown at the beginning of this tutorial, we want to set the pulse-width to a small value, then slowly increase the pulse-width to brighten the LED, and start over when we reach some maximum brightness:

```
# maximum and minimum pulse-width, which corresponds to maximum
# and minimum brightness
max_width = 200000
min_width = 20000

# how much to change the pulse-width by each step
wstep = 1500
cur_width = min_width

while True:
  tchannel.pulse_width(cur_width)

  # this determines how often we change the pulse-width. It is
  # analogous to frames-per-second
  sleep(0.01)

  cur_width += wstep

  if cur_width > max_width:
    cur_width = min_width
```

Breathing Effect

If we want to have a breathing effect, where the LED fades from dim to bright then bright to dim, then we simply need to reverse the sign of `wstep` when we reach maximum brightness, and reverse it again at minimum brightness. To do this we modify the `while` loop to be:

```
while True:
  tchannel.pulse_width(cur_width)

  sleep(0.01)

  cur_width += wstep

  if cur_width > max_width:
    cur_width = max_width
    wstep *= -1
  elif cur_width < min_width:
    cur_width = min_width
    wstep *= -1
```

Advanced Exercise

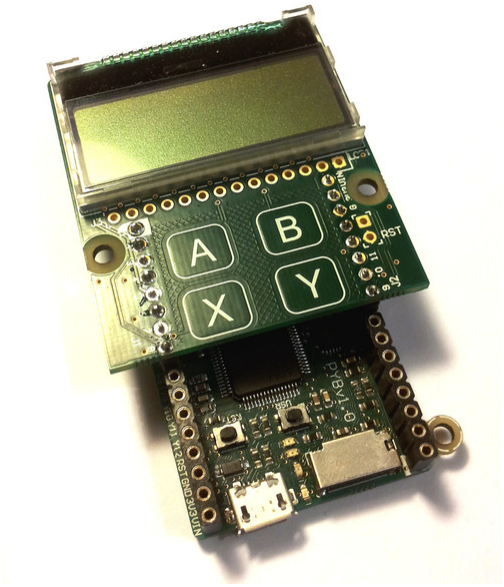
You may have noticed that the LED brightness seems to fade slowly, but increases quickly. This is because our eyes interprets brightness logarithmically (Webers Law), while the LEDs brightness changes linearly, that is by the same amount each time. How do you solve this problem? (Hint: what is the opposite of the logarithmic function?)

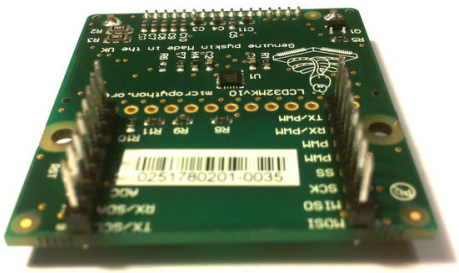
Addendum

We could have also used the digital-to-analog converter (DAC) to achieve the same effect. The PWM method has the advantage that it drives the LED with the same current each time, but for different lengths of time. This allows better control over the brightness, because LEDs do not necessarily exhibit a linear relationship between the driving current and brightness.

The LCD and touch-sensor skin

Soldering and using the LCD and touch-sensor skin.





The following video shows how to solder the headers onto the LCD skin. At the end of the video, it shows you how to correctly connect the LCD skin to the pyboard.

For circuit schematics and datasheets for the components on the skin see *The pyboard hardware*.

Using the LCD

To get started using the LCD, try the following at the MicroPython prompt. Make sure the LCD skin is attached to the pyboard as pictured at the top of this page.

```
>>> import pyb
>>> lcd = pyb.LCD('X')
>>> lcd.light(True)
>>> lcd.write('Hello uPy!\n')
```

You can make a simple animation using the code:

```
import pyb
lcd = pyb.LCD('X')
lcd.light(True)
for x in range(-80, 128):
    lcd.fill(0)
    lcd.text('Hello uPy!', x, 10, 1)
    lcd.show()
    pyb.delay(25)
```

Using the touch sensor

To read the touch-sensor data you need to use the I2C bus. The MPR121 capacitive touch sensor has address 90.

To get started, try:

```
>>> import pyb
>>> i2c = pyb.I2C(1, pyb.I2C.CONTROLLER)
>>> i2c.mem_write(4, 90, 0x5e)
>>> touch = i2c.mem_read(1, 90, 0)[0]
```

The first line above makes an I2C object, and the second line enables the 4 touch sensors. The third line reads the touch status and the `touch` variable holds the state of the 4 touch buttons (A, B, X, Y).

There is a simple driver here which allows you to set the threshold and debounce parameters, and easily read the touch status and electrode voltage levels. Copy this script to your pyboard (either flash or SD card, in the top directory or `lib/` directory) and then try:

```
>>> import pyb
>>> import mpr121
>>> m = mpr121.MPR121(pyb.I2C(1, pyb.I2C.CONTROLLER))
>>> for i in range(100):
...   print(m.touch_status())
...   pyb.delay(100)
...
```

This will continuously print out the touch status of all electrodes. Try touching each one in turn.

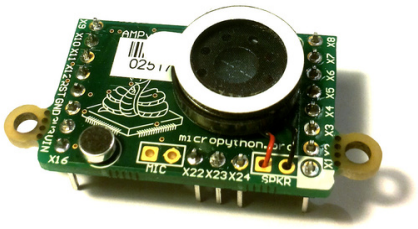
Note that if you put the LCD skin in the Y-position, then you need to initialise the I2C bus using:

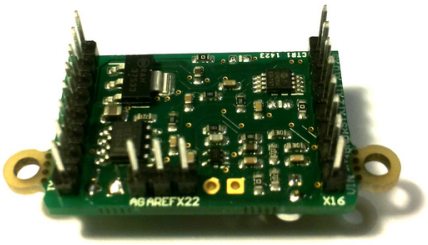
```
>>> m = mpr121.MPR121(pyb.I2C(2, pyb.I2C.CONTROLLER))
```

There is also a demo which uses the LCD and the touch sensors together, and can be found here.

The AMP audio skin

Soldering and using the AMP audio skin.





The following video shows how to solder the headers, microphone and speaker onto the AMP skin.

For circuit schematics and datasheets for the components on the skin see *The pyboard hardware*.

Example code

The AMP skin has a speaker which is connected to `DAC(1)` via a small power amplifier. The volume of the amplifier is controlled by a digital potentiometer, which is an I2C device with address 46 on the `IC2(1)` bus.

To set the volume, define the following function:

```
import pyb
def volume(val):
    pyb.I2C(1, pyb.I2C.CONTROLLER).mem_write(val, 46, 0)
```

Then you can do:

```
>>> volume(0)   # minimum volume
>>> volume(127) # maximum volume
```

To play a sound, use the `write_timed` method of the `DAC` object. For example:

```
import math
from pyb import DAC

# create a buffer containing a sine-wave
buf = bytearray(100)
for i in range(len(buf)):
    buf[i] = 128 + int(127 * math.sin(2 * math.pi * i / len(buf)))
```

(continues on next page)

(continued from previous page)

```
# output the sine-wave at 400Hz
dac = DAC(1)
dac.write_timed(buf, 400 * len(buf), mode=DAC.CIRCULAR)
```

You can also play WAV files using the Python `wave` module. You can get the wave module here and you will also need the chunk module available here. Put these on your pyboard (either on the flash or the SD card in the top-level directory). You will need an 8-bit WAV file to play, such as this one, or to convert any file you have with the command:

```
avconv -i original.wav -ar 22050 -codec pcm_u8 test.wav
```

Then you can do:

```
>>> import wave
>>> from pyb import DAC
>>> dac = DAC(1)
>>> f = wave.open('test.wav')
>>> dac.write_timed(f.readframes(f.getnframes()), f.getframerate())
```

This should play the WAV file. Note that this will read the whole file into RAM so it has to be small enough to fit in it.

To play larger wave files you will have to use the micro-SD card to store it. Also the file must be read and sent to the DAC in small chunks that will fit the RAM limit of the microcontroller. Here is an example function that can play 8-bit wave files with up to 16kHz sampling:

```
import wave
from pyb import DAC
from pyb import delay
dac = DAC(1)

def play(filename):
    f = wave.open(filename, 'r')
    total_frames = f.getnframes()
    framerate = f.getframerate()

    for position in range(0, total_frames, framerate):
        f.setpos(position)
        dac.write_timed(f.readframes(framerate), framerate)
        delay(1000)
```

This function reads one second worth of data and sends it to DAC. It then waits one second and moves the file cursor to the new position to read the next second of data in the next iteration of the for-loop. It plays one second of audio at a time every one second.

The LCD160CR skin

This tutorial shows how to get started using the LCD160CR skin.

For detailed documentation of the driver for the display see the *lcd160cr* module.

Plugging in the display

The display can be plugged directly into a pyboard (all pyboard versions are supported). You plug the display onto the top of the pyboard either in the X or Y positions. The display should cover half of the pyboard. See the picture above for how to achieve this; the left half of the picture shows the X position, and the right half shows the Y position.

Getting the driver

You can control the display directly using a power/enable pin and an I2C bus, but it is much more convenient to use the driver provided by the *lcd160cr* module. This driver is included in recent version of the pyboard firmware (see here). You can also find the driver in the GitHub repository here, and to use this version you will need to copy the file to your board, into a directory that is searched by import (usually the lib/ directory).

Once you have the driver installed you need to import it to use it:

```
import lcd160cr
```

Testing the display

There is a test program which you can use to test the features of the display, and which also serves as a basis to start creating your own code that uses the LCD. This test program is included in recent versions of the pyboard firmware and is also available on GitHub here.

To run the test from the MicroPython prompt do:

```
>>> import lcd160cr_test
```

It will then print some brief instructions. You will need to know which position your display is connected to (X or Y) and then you can run (assuming you have the display on position X):

```
>>> test_all('X')
```

Drawing some graphics

You must first create an LCD160CR object which will control the display. Do this using:

```
>>> import lcd160cr
>>> lcd = lcd160cr.LCD160CR('X')
```

This assumes your display is connected in the X position. If its in the Y position then use `lcd = lcd160cr.LCD160CR('Y')` instead.

To erase the screen and draw a line, try:

```
>>> lcd.set_pen(lcd.rgb(255, 0, 0), lcd.rgb(64, 64, 128))
>>> lcd.erase()
>>> lcd.line(10, 10, 50, 80)
```

The next example draws random rectangles on the screen. You can copy-and-paste it into the MicroPython prompt by first pressing Ctrl-E at the prompt, then Ctrl-D once you have pasted the text.

```
from random import randint
for i in range(1000):
    fg = lcd.rgb(randint(128, 255), randint(128, 255), randint(128, 255))
    bg = lcd.rgb(randint(0, 128), randint(0, 128), randint(0, 128))
    lcd.set_pen(fg, bg)
    lcd.rect(randint(0, lcd.w), randint(0, lcd.h), randint(10, 40), randint(10, 40))
```

Using the touch sensor

The display includes a resistive touch sensor that can report the position (in pixels) of a single force-based touch on the screen. To see if there is a touch on the screen use:

```
>>> lcd.is_touched()
```

This will return either `False` or `True`. Run the above command while touching the screen to see the result.

To get the location of the touch you can use the method:

```
>>> lcd.get_touch()
```

This will return a 3-tuple, with the first entry being 0 or 1 depending on whether there is currently anything touching the screen (1 if there is), and the second and third entries in the tuple being the x and y coordinates of the current (or most recent) touch.

Directing the MicroPython output to the display

The display supports input from a UART and implements basic VT100 commands, which means it can be used as a simple, general purpose terminal. Lets set up the pyboard to redirect its output to the display.

First you need to create a UART object:

```
>>> import pyb
>>> uart = pyb.UART('XA', 115200)
```

This assumes your display is connected to position X. If its on position Y then use `uart = pyb.UART('YA', 115200)` instead.

Now, connect the REPL output to this UART:

```
>>> pyb.repl_uart(uart)
```

From now on anything you type at the MicroPython prompt, and any output you receive, will appear on the display.

No set-up commands are required for this mode to work and you can use the display to monitor the output of any UART, not just from the pyboard. All that is needed is for the display to have power, ground and the power/enable pin driven high. Then any characters on the displays UART input will be printed to the screen. You can adjust the UART baudrate from the default of 115200 using the *set_uart_baudrate* method.

6.2.13 Tips, tricks and useful things to know

Debouncing a pin input

A pin used as input from a switch or other mechanical device can have a lot of noise on it, rapidly changing from low to high when the switch is first pressed or released. This noise can be eliminated using a capacitor (a debouncing circuit). It can also be eliminated using a simple function that makes sure the value on the pin is stable.

The following function does just this. It gets the current value of the given pin, and then waits for the value to change. The new pin value must be stable for a continuous 20ms for it to register the change. You can adjust this time (to say 50ms) if you still have noise.

```
import pyb

def wait_pin_change(pin):
    # wait for pin to change value
    # it needs to be stable for a continuous 20ms
    cur_value = pin.value()
    active = 0
    while active < 20:
        if pin.value() != cur_value:
            active += 1
        else:
```

(continues on next page)

(continued from previous page)

```
            active = 0
        pyb.delay(1)
```

Use it something like this:

```
import pyb

pin_x1 = pyb.Pin('X1', pyb.Pin.IN, pyb.Pin.PULL_DOWN)
while True:
    wait_pin_change(pin_x1)
    pyb.LED(4).toggle()
```

Making a UART - USB pass through

Its as simple as:

```
import pyb
import select

def pass_through(usb, uart):
    usb.setinterrupt(-1)
    while True:
        select.select([usb, uart], [], [])
        if usb.any():
            uart.write(usb.read(256))
        if uart.any():
            usb.write(uart.read(256))

pass_through(pyb.USB_VCP(), pyb.UART(1, 9600, timeout=0))
```

6.3 General board control

See *pyb*.

```
import pyb

pyb.repl_uart(pyb.UART(1, 9600)) # duplicate REPL on UART(1)
pyb.wfi() # pause CPU, waiting for interrupt
pyb.freq() # get CPU and bus frequencies
pyb.freq(60000000) # set CPU freq to 60MHz
pyb.stop() # stop CPU, waiting for external interrupt
```

6.4 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 value of millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start) # compute time difference
```

6.5 Internal LEDs

See *pyb.LED*.

```
from pyb import LED

led = LED(1) # 1=red, 2=green, 3=yellow, 4=blue
led.toggle()
led.on()
led.off()

# LEDs 3 and 4 support PWM intensity (0-255)
LED(4).intensity()    # get intensity
LED(4).intensity(128) # set intensity to half
```

6.6 Internal switch

See *pyb.Switch*.

```
from pyb import Switch

sw = Switch()
sw.value() # returns True or False
sw.callback(lambda: pyb.LED(1).toggle())
```

6.7 Pins and GPIO

See *pyb.Pin*.

```
from pyb import Pin

p_out = Pin('X1', Pin.OUT_PP)
p_out.high()
p_out.low()
```

(continues on next page)

(continued from previous page)

```
p_in = Pin('X2', Pin.IN, Pin.PULL_UP)
p_in.value() # get value, 0 or 1
```

6.8 Servo control

See *pyb.Servo*.

```
from pyb import Servo

s1 = Servo(1) # servo on position 1 (X1, VIN, GND)
s1.angle(45) # move to 45 degrees
s1.angle(-60, 1500) # move to -60 degrees in 1500ms
s1.speed(50) # for continuous rotation servos
```

6.9 External interrupts

See *pyb.ExtInt*.

```
from pyb import Pin, ExtInt

callback = lambda e: print("intr")
ext = ExtInt(Pin('Y1'), ExtInt.IRQ_RISING, Pin.PULL_NONE, callback)
```

6.10 Timers

See *pyb.Timer*.

```
from pyb import Timer

tim = Timer(1, freq=1000)
tim.counter() # get counter value
tim.freq(0.5) # 0.5 Hz
tim.callback(lambda t: pyb.LED(1).toggle())
```

6.11 RTC (real time clock)

See *pyb.RTC*

```
from pyb import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 1, 12, 48, 0, 0)) # set a specific date and time
rtc.datetime() # get date and time
```

6.12 PWM (pulse width modulation)

See *pyb.Pin* and *pyb.Timer*.

```
from pyb import Pin, Timer

p = Pin('X1') # X1 has TIM2, CH1
tim = Timer(2, freq=1000)
ch = tim.channel(1, Timer.PWM, pin=p)
ch.pulse_width_percent(50)
```

6.13 ADC (analog to digital conversion)

See *pyb.Pin* and *pyb.ADC*.

```
from pyb import Pin, ADC

adc = ADC(Pin('X19'))
adc.read() # read value, 0-4095
```

6.14 DAC (digital to analog conversion)

See *pyb.Pin* and *pyb.DAC*.

```
from pyb import Pin, DAC

dac = DAC(Pin('X5'))
dac.write(120) # output between 0 and 255
```

6.15 UART (serial bus)

See *pyb.UART*.

```
from pyb import UART

uart = UART(1, 9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

6.16 SPI bus

See *pyb.SPI*.

```
from pyb import SPI

spi = SPI(1, SPI.CONTROLLER, baudrate=200000, polarity=1, phase=0)
spi.send('hello')
spi.recv(5) # receive 5 bytes on the bus
spi.send_recv('hello') # send and receive 5 bytes
```

6.17 I2C bus

Hardware I2C is available on the X and Y halves of the pyboard via `I2C('X')` and `I2C('Y')`. Alternatively pass in the integer identifier of the peripheral, eg `I2C(1)`. Software I2C is also available by explicitly specifying the `scl` and `sda` pins instead of the bus name. For more details see *machine.I2C*.

```
from machine import I2C

i2c = I2C('X', freq=400000)                 # create hardware I2c object
i2c = I2C(scl='X1', sda='X2', freq=100000)  # create software I2C object

i2c.scan()                            # returns list of peripheral addresses
i2c.writeto(0x42, 'hello')            # write 5 bytes to peripheral with address 0x42
i2c.readfrom(0x42, 5)                 # read 5 bytes from peripheral

i2c.readfrom_mem(0x42, 0x10, 2)       # read 2 bytes from peripheral 0x42, peripheral␣
↪memory 0x10
i2c.writeto_mem(0x42, 0x10, 'xy')     # write 2 bytes to peripheral 0x42, peripheral␣
↪memory 0x10
```

Note: for legacy I2C support see *pyb.I2C*.

6.18 I2S bus

See *machine.I2S*.

```
from machine import I2S, Pin

i2s = I2S(2, sck=Pin('Y6'), ws=Pin('Y5'), sd=Pin('Y8'), mode=I2S.TX, bits=16, format=I2S.
↪STEREO, rate=44100, ibuf=40000) # create I2S object
i2s.write(buf)             # write buffer of audio samples to I2S device

i2s = I2S(1, sck=Pin('X5'), ws=Pin('X6'), sd=Pin('Y4'), mode=I2S.RX, bits=16, format=I2S.
↪MONO, rate=22050, ibuf=40000) # create I2S object
i2s.readinto(buf)          # fill buffer with audio samples from I2S device
```

The I2S class is currently available as a Technical Preview. During the preview period, feedback from users is encouraged. Based on this feedback, the I2S class API and implementation may be changed.

PYBv1.0/v1.1 has one I2S bus with id=2. PYBD-SFxW has two I2S buses with id=1 and id=2. I2S is shared with SPI.

6.19 CAN bus (controller area network)

See *pyb.CAN*.

```
from pyb import CAN

can = CAN(1, CAN.LOOPBACK)
can.setfilter(0, CAN.LIST16, 0, (123, 124, 125, 126))
can.send('message!', 123)   # send a message with id 123
can.recv(0)                 # receive message on FIFO 0
```

6.20 Internal accelerometer

See *pyb.Accel*.

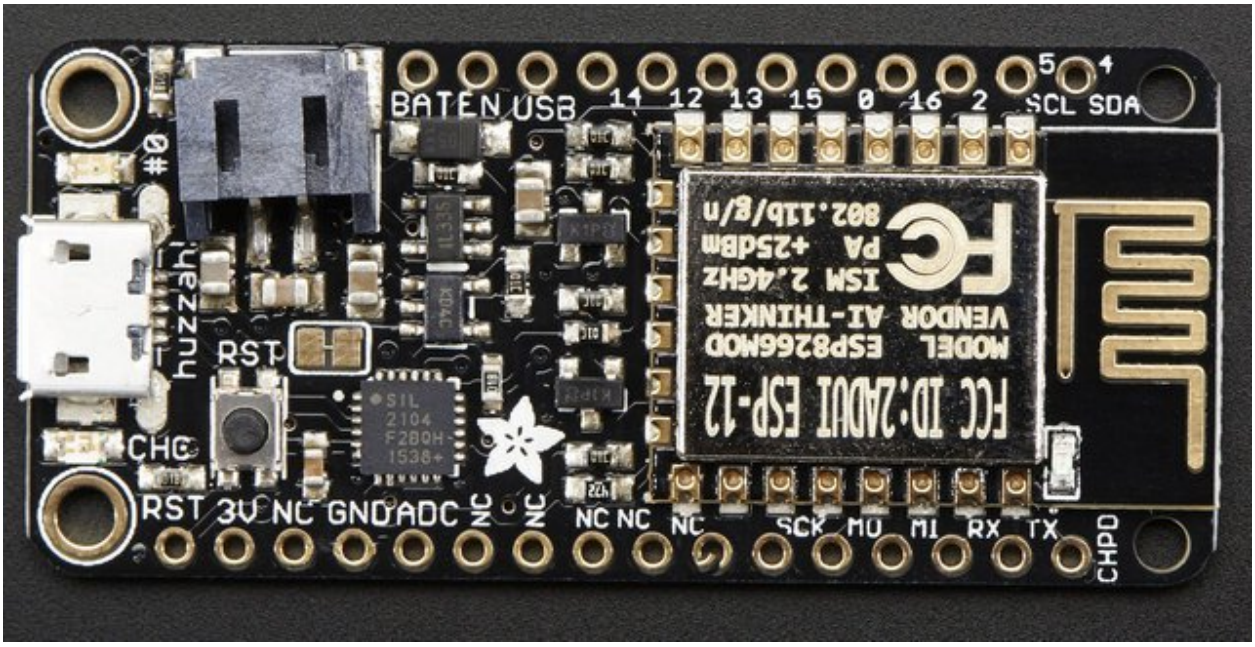
```
from pyb import Accel

accel = Accel()
print(accel.x(), accel.y(), accel.z(), accel.tilt())
```

CHAPTER

SEVEN

QUICK REFERENCE FOR THE ESP8266



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

Below is a quick reference for ESP8266-based boards. If it is your first time working with this board please consider reading the following sections first:

7.1 General information about the ESP8266 port

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

7.1.1 Multitude of boards

There is 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 a 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, the 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 theyre 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.

7.1.2 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.

7.1.3 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 thats the task of a user/user application: please be sure to close open files, sockets, etc. as soon as possible after use.

7.1.4 Boot process

On boot, MicroPython EPS8266 port executes `_boot.py` script from internal frozen modules. It mounts filesystem in FlashROM, or if its 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, its 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 its 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.

7.1.5 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 `time.time()`, etc. functions, and for wall clock time, synchronize from the net using included `ntptime.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.

Simultaneous operation of STA_IF and AP_IF

Simultaneous operation of STA_IF and AP_IF interfaces is supported.

However, due to restrictions of the hardware, there may be performance issues in the AP_IF, if the STA_IF is not connected and searching. An application should manage these interfaces and for example deactivate the STA_IF in environments where only the AP_IF is used.

Sockets and WiFi buffers overflow

Socket instances remain active until they are explicitly closed. This has two consequences. Firstly they occupy RAM, so an application which opens sockets without closing them may eventually run out of memory. Secondly not properly closed socket can cause the low-level part of the vendor WiFi stack to emit `Lmac` errors. This occurs if data comes in for a socket and is not processed in a timely manner. This can overflow the WiFi stack input queue and lead to a deadlock. The only recovery is by a hard reset.

The above may also happen after an application terminates and quits to the REPL for any reason including an exception. Subsequent arrival of data provokes the failure with the above error message repeatedly issued. So, sockets should be closed in any case, regardless whether an application terminates successfully or by an exception, for example using try/finally:

```
sock = socket(...)
try:
    # Use sock
finally:
    sock.close()
```

SSL/TLS limitations

ESP8266 uses axTLS library, which is one of the smallest TLS libraries with compatible licensing. However, it also has some known issues/limitations:

1. No support for Diffie-Hellman (DH) key exchange and Elliptic-curve cryptography (ECC). This means it cant work with sites which require the use of these features (it works ok with the typical sites that use RSA certificates).
2. Half-duplex communication nature. axTLS uses a single buffer for both sending and receiving, which leads to considerable memory saving and works well with protocols like HTTP. But there may be problems with protocols which dont follow classic request-response model.

Besides axTLSs own limitations, the configuration used for MicroPython is highly optimized for code size, which leads to additional limitations (these may be lifted in the future):

3. Optimized RSA algorithms are not enabled, which may lead to slow SSL handshakes.
4. Session Reuse is not enabled, which means every connection must undergo the full, expensive SSL handshake.

Besides axTLS specific limitations described above, theres another generic limitation with usage of TLS on the low-memory devices:

5. The TLS standard specifies the maximum length of the TLS record (unit of TLS communication, the entire record must be buffered before it can be processed) as 16KB. Thats almost half of the available ESP8266 memory, and inside a more or less advanced application would be hard to allocate due to memory fragmentation issues. As a compromise, a smaller buffer is used, with the idea that the most interesting usage for SSL would be accessing various REST APIs, which usually require much smaller messages. The buffers size is on the order of 5KB, and is adjusted from time to time, taking as a reference being able to access https://google.com . The smaller buffer however means that some sites cant be accessed using it, and its not possible to stream large amounts of data.

axTLS does have support for TLSs Max Fragment Size extension, but no HTTPS website does, so use of the extension is really only effective for local communication with other devices.

There are also some not implemented features specifically in MicroPythons `ssl` module based on axTLS:

6. Certificates are not validated (this makes connections susceptible to man-in-the-middle attacks).
7. There is no support for client certificates (scheduled to be fixed in 1.9.4 release).

7.2 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.

7.2.1 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.

Lets 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 wont 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.

If you are just starting with MicroPython, the best bet is to go for the Stable firmware builds. If you are an advanced, experienced MicroPython ESP8266 user who would like to follow development closely and help with testing new features, there are daily builds (note: you actually may need some development experience, e.g. being ready to follow git history to know what new changes and features were introduced).

Support for 512kb modules is provided on a feature preview basis. For end users, its recommended to use modules with flash of 1024kb or more. As such, only daily builds for 512kb modules are provided.

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.

If you do not have such a board, you need keep GPIO0 pulled to ground and reset the device by pulling the reset pin to ground and releasing it again to enter programming mode.

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 manually set a compatible SPI Flash Mode. Youd usually pick the fastest option that is compatible with your device, but the `-fm dout` option (the slowest option) should have the best compatibility:

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

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

If you pulled GPIO0 manually to ground to enter programming mode, release it now and reset the device by again pulling the reset pin to ground for a short duration.

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 xs 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 didnt 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 doesnt 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 doesnt report errors):

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

If the last output value is True, the firmware is OK. Otherwise, its 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.

7.2.2 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 ESP8266s 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 ESP8266s 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 shouldnt 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
49222352429520267040371132431220080064
```

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 (thats 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 doesnt) and you can toggle the LED using:

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

Lets now toggle the LED in a loop (if you dont 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.

7.2.3 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.

7.2.4 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.

7.2.5 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. Its very simple to use because it doesnt 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.

Lets define a function that can download and print a URL:

```
def http_get(url):
    import socket
    _, _, 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()
```

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>
```

(continues on next page)

(continued from previous page)

```
    <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('HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n')
    cl.send(response)
    cl.close()
```

7.2.6 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 its not specified then it defaults to None, which is no pull resistor. GPIO16 has no pull-up mode. 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 its 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.

Lets 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.

7.2.7 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. Values beyond this min/max will be clipped. 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

Lets 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)
```

7.2.8 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.

7.2.9 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 changed 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')
```

7.2.10 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)
```

(continues on next page)

(continued from previous page)

```
# 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.

7.2.11 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
```

For LEDs with more than 3 colours, such as RGBW pixels or RGBY pixels, the NeoPixel class takes a `bpp` parameter. To setup a NeoPixel object for an RGBW Pixel, do the following:

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

In a 4-bpp mode, remember to use 4-tuples instead of 3-tuples to set the colour. For example to set the first three pixels use:

```
>>> np[0] = (255, 0, 0, 128) # Orange in an RGBY Setup
>>> np[1] = (0, 255, 0, 128) # Yellow-green in an RGBY Setup
>>> np[2] = (0, 0, 255, 128) # Green-blue in an RGBY Setup
```

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)
```

7.2.12 Controlling APA102 LEDs

APA102 LEDs, also known as DotStar LEDs, are individually addressable full-colour RGB LEDs, generally in a string formation. They differ from NeoPixels in that they require two pins to control - both a Clock and Data pin. They can operate at a much higher data and PWM frequencies than NeoPixels and are more suitable for persistence-of-vision effects.

To create an APA102 object do the following:

```
>>> import machine, apa102
>>> strip = apa102.APA102(machine.Pin(5), machine.Pin(4), 60)
```

This configures an 60 pixel APA102 strip with clock on GPIO5 and data on GPIO4. You can adjust the pin numbers and the number of pixels to suit your needs.

The RGB colour data, as well as a brightness level, is sent to the APA102 in a certain order. Usually this is `(Red, Green, Blue, Brightness)`. If you are using one of the newer APA102C LEDs the green and blue are swapped, so the order is `(Red, Blue, Green, Brightness)`. The APA102 has more of a square lens while the APA102C has more of a round one. If you are using a APA102C strip and would prefer to provide colours in RGB order instead of RBG, you can customise the tuple colour order like so:

```
>>> strip.ORDER = (0, 2, 1, 3)
```

To set the colour of pixels use:

```
>>> strip[0] = (255, 255, 255, 31) # set to white, full brightness
>>> strip[1] = (255, 0, 0, 31) # set to red, full brightness
>>> strip[2] = (0, 255, 0, 15) # set to green, half brightness
>>> strip[3] = (0, 0, 255, 7)  # set to blue, quarter brightness
```

Use the `write()` method to output the colours to the LEDs:

```
>>> strip.write()
```

Demonstration:

```
import time
import machine, apa102

# 1M strip with 60 LEDs
strip = apa102.APA102(machine.Pin(5), machine.Pin(4), 60)

brightness = 1  # 0 is off, 1 is dim, 31 is max

# Helper for converting 0-255 offset to a colour tuple
def wheel(offset, brightness):
    # The colours are a transition r - g - b - back to r
    offset = 255 - offset
    if offset < 85:
        return (255 - offset * 3, 0, offset * 3, brightness)
    if offset < 170:
        offset -= 85
        return (0, offset * 3, 255 - offset * 3, brightness)
    offset -= 170
    return (offset * 3, 255 - offset * 3, 0, brightness)
```

(continues on next page)

(continued from previous page)

```
# Demo 1: RGB RGB RGB
red = 0xff0000
green = red >> 8
blue = red >> 16
for i in range(strip.n):
    colour = red >> (i % 3) * 8
    strip[i] = ((colour & red) >> 16, (colour & green) >> 8, (colour & blue), brightness)
strip.write()

# Demo 2: Show all colours of the rainbow
for i in range(strip.n):
    strip[i] = wheel((i * 256 // strip.n) % 255, brightness)
strip.write()

# Demo 3: Fade all pixels together through rainbow colours, offset each pixel
for r in range(5):
    for n in range(256):
        for i in range(strip.n):
            strip[i] = wheel(((i * 256 // strip.n) + n) & 255, brightness)
        strip.write()
    time.sleep_ms(25)

# Demo 4: Same colour, different brightness levels
for b in range(31,-1,-1):
    strip[0] = (255, 153, 0, b)
    strip.write()
    time.sleep_ms(250)

# End: Turn off all the LEDs
strip.fill((0, 0, 0, 0))
strip.write()
```

7.2.13 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))
```

(continues on next page)

(continued from previous page)

```
>>> 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.

7.2.14 Using a SSD1306 OLED display

The SSD1306 OLED display uses either a SPI or I2C interface and comes in a variety of sizes (128x64, 128x32, 72x40, 64x48) and colours (white, yellow, blue, yellow + blue).

Hardware SPI interface:

```
from machine import Pin, SPI
import ssd1306

hspi = SPI(1)  # sck=14 (scl), mosi=13 (sda), miso=12 (unused)

dc = Pin(4)    # data/command
rst = Pin(5)   # reset
cs = Pin(15)   # chip select, some modules do not have a pin for this

display = ssd1306.SSD1306_SPI(128, 64, hspi, dc, rst, cs)
```

Software SPI interface:

```
from machine import Pin, SoftSPI
import ssd1306

spi = SoftSPI(baudrate=500000, polarity=1, phase=0, sck=Pin(14), mosi=Pin(13),␣
↪miso=Pin(12))

dc = Pin(4)   # data/command
rst = Pin(5)  # reset
cs = Pin(15)  # chip select, some modules do not have a pin for this

display = ssd1306.SSD1306_SPI(128, 64, spi, dc, rst, cs)
```

I2C interface:

```
from machine import Pin, I2C
import ssd1306

# using default address 0x3C
i2c = I2C(sda=Pin(4), scl=Pin(5))
display = ssd1306.SSD1306_I2C(128, 64, i2c)
```

Print Hello World on the first line:

```
display.text('Hello, World!', 0, 0, 1)
display.show()
```

Basic functions:

```
display.poweroff()      # power off the display, pixels persist in memory
display.poweron()       # power on the display, pixels redrawn
display.contrast(0)     # dim
display.contrast(255)   # bright
display.invert(1)       # display inverted
display.invert(0)       # display normal
display.rotate(True)    # rotate 180 degrees
display.rotate(False)   # rotate 0 degrees
display.show()          # write the contents of the FrameBuffer to display memory
```

Subclassing FrameBuffer provides support for graphics primitives:

```
display.fill(0)                         # fill entire screen with colour=0
display.pixel(0, 10)                    # get pixel at x=0, y=10
display.pixel(0, 10, 1)                 # set pixel at x=0, y=10 to colour=1
display.hline(0, 8, 4, 1)               # draw horizontal line x=0, y=8, width=4,␣
↪colour=1
display.vline(0, 8, 4, 1)               # draw vertical line x=0, y=8, height=4, colour=1
display.line(0, 0, 127, 63, 1)          # draw a line from 0,0 to 127,63
display.rect(10, 10, 107, 43, 1)        # draw a rectangle outline 10,10 to 117,53,␣
↪colour=1
display.fill_rect(10, 10, 107, 43, 1)   # draw a solid rectangle 10,10 to 117,53,␣
↪colour=1
display.text('Hello World', 0, 0, 1)    # draw some text at x=0, y=0, colour=1
```

(continues on next page)

(continued from previous page)

```
display.scroll(20, 0)                    # scroll 20 pixels to the right

# draw another FrameBuffer on top of the current one at the given coordinates
import framebuf
fbuf = framebuf.FrameBuffer(bytearray(8 * 8 * 1), 8, 8, framebuf.MONO_VLSB)
fbuf.line(0, 0, 7, 7, 1)
display.blit(fbuf, 10, 10, 0)            # draw on top at x=10, y=10, key=0
display.show()
```

Draw the MicroPython logo and print some text:

```
display.fill(0)
display.fill_rect(0, 0, 32, 32, 1)
display.fill_rect(2, 2, 28, 28, 0)
display.vline(9, 8, 22, 1)
display.vline(16, 2, 22, 1)
display.vline(23, 8, 22, 1)
display.fill_rect(26, 24, 2, 4, 1)
display.text('MicroPython', 40, 0, 1)
display.text('SSD1306', 40, 12, 1)
display.text('OLED 128x64', 40, 24, 1)
display.show()
```

7.2.15 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!

7.3 Installing MicroPython

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

7.4 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)
```

7.5 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 address
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.

7.6 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
```

7.7 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.

7.8 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`.

Theres a higher-level abstraction *machine.Signal* which can be used to invert a pin. Useful for illuminating active-low LEDs using `on()` or `value(1)`.

7.9 UART (serial bus)

See *machine.UART*.

```
from machine import UART
uart = UART(0, baudrate=9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

Two UARTs are available. UART0 is on Pins 1 (TX) and 3 (RX). UART0 is bidirectional, and by default is used for the REPL. UART1 is on Pins 2 (TX) and 8 (RX) however Pin 8 is used to connect the flash chip, so UART1 is TX only.

When UART0 is attached to the REPL, all incoming chars on UART(0) go straight to stdin so uart.read() will always return None. Use sys.stdin.read() if its needed to read characters from the UART(0) while its also used for the REPL (or detach, read, then reattach). When detached the UART(0) can be used for other purposes.

If there are no objects in any of the dupterm slots when the REPL is started (on hard or soft reset) then UART(0) is automatically attached. Without this, the only way to recover a board without a REPL would be to completely erase and reflash (which would install the default boot.py which attaches the REPL).

To detach the REPL from UART0, use:

```
import os
os.dupterm(None, 1)
```

The REPL is attached by default. If you have detached it, to reattach it use:

```
import os, machine
uart = machine.UART(0, 115200)
os.dupterm(uart, 1)
```

7.10 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
```

7.11 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
```

7.12 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.SoftSPI* class:

```
from machine import Pin, SoftSPI

# 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 = SoftSPI(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 outputting 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
```

7.13 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.)

7.14 I2C bus

The I2C driver is implemented in software and works on all pins, and is accessed via the *machine.I2C* class (which is an alias of *machine.SoftI2C*):

```
from machine import Pin, I2C

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

(continues on next page)

(continued from previous page)

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

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

7.15 Real time clock (RTC)

See *machine.RTC*

```
from machine import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 1, 12, 48, 0, 0)) # set a specific date and time
rtc.datetime() # get date and time

# synchronize with ntp
# need to be connected to wifi
import ntptime
ntptime.settime() # set the rtc datetime from the remote server
rtc.datetime()    # get the date and time in UTC
```

Note: Not all methods are implemented: *RTC.now()*, *RTC.irq(handler=*)* (using a custom handler), *RTC.init()* and *RTC.deinit()* are currently not supported.

7.16 WDT (Watchdog timer)

See *machine.WDT*.

```
from machine import WDT

# enable the WDT
wdt = WDT()
wdt.feed()
```

7.17 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()
```

(continues on next page)

(continued from previous page)

```
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()
```

7.18 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.

7.19 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
```

(continues on next page)

(continued from previous page)

```
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)
```

Warning: By default `NeoPixel` is configured to control the more popular *800kHz* units. It is possible to use alternative timing to control other (typically 400kHz) devices by passing `timing=0` when constructing the `NeoPixel` object.

7.20 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)
```

7.21 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 (řC)
d.humidity()    # eg. 41 (% RH)

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

(continues on next page)

(continued from previous page)

```
d.measure()
d.temperature() # eg. 23.6 (řC)
d.humidity()    # eg. 41.3 (% RH)
```

7.22 SSD1306 driver

Driver for SSD1306 monochrome OLED displays. See tutorial *Using a SSD1306 OLED display*.

```
from machine import Pin, I2C
import ssd1306

i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)
display = ssd1306.SSD1306_I2C(128, 64, i2c)

display.text('Hello World', 0, 0, 1)
display.show()
```

7.23 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).

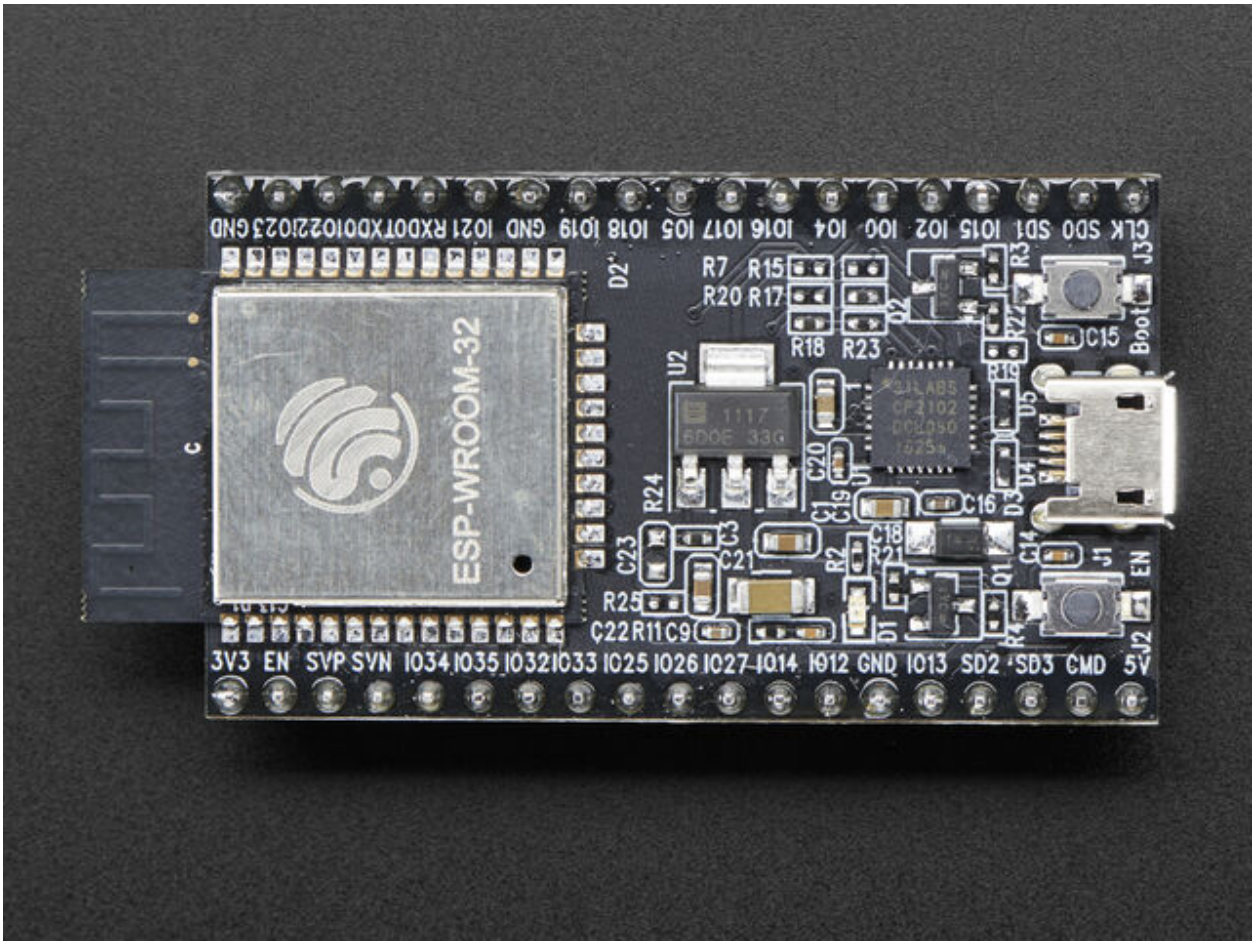
Besides terminal/command prompt access, WebREPL also has provision for file transfer (both upload and download). Web client has buttons for the corresponding functions, or you can use command-line client `webrepl_cli.py` from the repository above.

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

CHAPTER

EIGHT

QUICK REFERENCE FOR THE ESP32



The Espressif ESP32 Development Board (image attribution: Adafruit).

Below is a quick reference for ESP32-based boards. If it is your first time working with this board it may be useful to get an overview of the microcontroller:

8.1 General information about the ESP32 port

The ESP32 is a popular WiFi and Bluetooth enabled System-on-Chip (SoC) by Espressif Systems.

8.1.1 Multitude of boards

There is a multitude of modules and boards from different sources which carry the ESP32 chip. MicroPython tries to provide a generic port which would run on as many boards/modules as possible, but there may be limitations. Espressif development boards are taken as reference for the port (for example, testing is performed on them). For any board you are using please make sure you have a datasheet, schematics and other reference materials so you can look up any board-specific functions.

To make a generic ESP32 port and support as many boards as possible the following design and implementation decision were made:

- GPIO pin numbering is based on ESP32 chip numbering. Please have the manual/pin diagram of your board at hand to find correspondence between your board pins and actual ESP32 pins.
- All pins are supported by MicroPython but not all are usable on any given board. For example pins that are connected to external SPI flash should not be used, and a board may only expose a certain selection of pins.

8.1.2 Technical specifications and SoC datasheets

The datasheets and other reference material for ESP32 chip are available from the vendor site: https://www.espressif.com/en/support/download/documents?keys=esp32 . 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 Dual-Core 32-bit LX6
- CPU frequency: up to 240MHz
- Total RAM available: 528KB (part of it reserved for system)
- BootROM: 448KB
- Internal FlashROM: none
- External FlashROM: code and data, via SPI Flash; usual size 4MB
- GPIO: 34 (GPIOs are multiplexed with other functions, including external FlashROM, UART, etc.)
- UART: 3 RX/TX UART (no hardware handshaking), one TX-only UART
- SPI: 4 SPI interfaces (one used for FlashROM)
- I2C: 2 I2C (bitbang implementation available on any pins)
- I2S: 2
- ADC: 12-bit SAR ADC up to 18 channels
- DAC: 2 8-bit DACs
- RMT: 8 channels allowing accurate pulse transmit/receive
- Programming: using BootROM bootloader from UART - due to external FlashROM and always-available BootROM bootloader, the ESP32 is not brickable

For more information see the ESP32 datasheet: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

MicroPython is implemented on top of the ESP-IDF, Espressifs development framework for the ESP32. This is a FreeRTOS based system. See the ESP-IDF Programming Guide for details.

8.2 MicroPython tutorial for ESP32

This tutorial is intended to get you started using MicroPython on the ESP32 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.

8.2.1 Getting started with MicroPython on the ESP32

Using MicroPython is a great way to get the most of your ESP32 board. And vice versa, the ESP32 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.

Lets get started!

Requirements

The first thing you need is a board with an ESP32 chip. The MicroPython software supports the ESP32 chip itself and any board should work. The main characteristic of a board is 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.

Names of pins will be given in this tutorial using the chip names (eg GPIO2) 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 ESP32 device. You can download it from the MicroPython downloads page. From here, you have 3 main choices:

- Stable firmware builds
- Daily firmware builds
- Daily firmware builds with SPIRAM support

If you are just starting with MicroPython, the best bet is to go for the Stable firmware builds. If you are an advanced, experienced MicroPython ESP32 user who would like to follow development closely and help with testing new features, there are daily builds. If your board has SPIRAM support you can use either the standard firmware or the firmware with SPIRAM support, and in the latter case you will have access to more RAM for Python objects.

Deploying the firmware

Once you have the MicroPython firmware you need to load it onto your ESP32 device. There are two main steps to do this: first you need to put your device in bootloader 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.

Fortunately, most boards have a USB connector, a USB-serial convertor, and 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 HUZZAH32, M5Stack, Wemos LOLIN32, and TinyPICO boards, along with the Espressif DevKitC, PICO-KIT, WROVER-KIT dev-kits.

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.

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 --chip esp32 --port /dev/ttyUSB0 write_flash -z 0x1000 esp32-20180511-v1.9.4.
↪bin
```

Notes:

- You might need to change the port setting to something else relevant for your PC
- You may need to reduce the baudrate if you get errors when flashing (eg down to 115200 by adding `--baud 115200` into the command)
- For some boards with a particular FlashROM configuration you may need to change the flash mode (eg by adding `-fm dio` into the command)
- The filename of the firmware should match the file that you have

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.

From here you can now follow the ESP8266 tutorial, because these two Espressif chips are very similar when it comes to using MicroPython on them. The ESP8266 tutorial is found at *MicroPython tutorial for ESP8266* (but skip the Introduction section).

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. The most reliable and convenient power source is a USB port.
- 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.
- To catch incorrect flash content (e.g. from a defective sector on a chip), add `--verify` switch to the commands above.
- If you still experience problems with flashing the firmware please refer to esptool.py project page, https://github.com/espressif/esptool for additional documentation and a bug tracker where you can report problems.
- If you are able to flash the firmware but the `--verify` option returns errors even after multiple retries the you may have a defective FlashROM chip.

8.2.2 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.

- More comprehensive example with all 16 PWM channels and 8 timers:

```
from machine import Pin, PWM
try:
    f = 100  # Hz
    d = 1024 // 16  # 6.25%
    pins = (15, 2, 4, 16, 18, 19, 22, 23, 25, 26, 27, 14 , 12, 13, 32, 33)
    pwms = []
    for i, pin in enumerate(pins):
        pwms.append(PWM(Pin(pin), freq=f * (i // 2 + 1), duty= 1023 if i==15 else d␣
↪* (i + 1)))
        print(pwms[i])
finally:
    for pwm in pwms:
        try:
            pwm.deinit()
        except:
            pass
```

Output is:

```
PWM(Pin(15), freq=100, duty=64, resolution=10, mode=0, channel=0, timer=0)
PWM(Pin(2), freq=100, duty=128, resolution=10, mode=0, channel=1, timer=0)
PWM(Pin(4), freq=200, duty=192, resolution=10, mode=0, channel=2, timer=1)
PWM(Pin(16), freq=200, duty=256, resolution=10, mode=0, channel=3, timer=1)
PWM(Pin(18), freq=300, duty=320, resolution=10, mode=0, channel=4, timer=2)
```

(continues on next page)

(continued from previous page)

```
PWM(Pin(19), freq=300, duty=384, resolution=10, mode=0, channel=5, timer=2)
PWM(Pin(22), freq=400, duty=448, resolution=10, mode=0, channel=6, timer=3)
PWM(Pin(23), freq=400, duty=512, resolution=10, mode=0, channel=7, timer=3)
PWM(Pin(25), freq=500, duty=576, resolution=10, mode=1, channel=0, timer=0)
PWM(Pin(26), freq=500, duty=640, resolution=10, mode=1, channel=1, timer=0)
PWM(Pin(27), freq=600, duty=704, resolution=10, mode=1, channel=2, timer=1)
PWM(Pin(14), freq=600, duty=768, resolution=10, mode=1, channel=3, timer=1)
PWM(Pin(12), freq=700, duty=832, resolution=10, mode=1, channel=4, timer=2)
PWM(Pin(13), freq=700, duty=896, resolution=10, mode=1, channel=5, timer=2)
PWM(Pin(32), freq=800, duty=960, resolution=10, mode=1, channel=6, timer=3)
PWM(Pin(33), freq=800, duty=1023, resolution=10, mode=1, channel=7, timer=3)
```

- Example of a smooth frequency change:

```
from utime import sleep
from machine import Pin, PWM

F_MIN = 500
F_MAX = 1000

f = F_MIN
delta_f = 1

p = PWM(Pin(5), f)
print(p)

while True:
    p.freq(f)

    sleep(10 / F_MIN)

    f += delta_f
    if f >= F_MAX or f <= F_MIN:
        delta_f = -delta_f
```

See PWM wave at Pin(5) with an oscilloscope.

- Example of a smooth duty change:

```
from utime import sleep
from machine import Pin, PWM

DUTY_MAX = 2**16 - 1

duty_u16 = 0
delta_d = 16

p = PWM(Pin(5), 1000, duty_u16=duty_u16)
print(p)

while True:
    p.duty_u16(duty_u16)
```

(continues on next page)

(continued from previous page)

```
    sleep(1 / 1000)

    duty_u16 += delta_d
    if duty_u16 >= DUTY_MAX:
        duty_u16 = DUTY_MAX
        delta_d = -delta_d
    elif duty_u16 <= 0:
        duty_u16 = 0
        delta_d = -delta_d
```

See PWM wave at Pin(5) with an oscilloscope.

Note: the Pin.OUT mode does not need to be specified. The channel is initialized to PWM mode internally once for each Pin that is passed to the PWM constructor.

The following code is wrong:

```
pwm = PWM(Pin(5, Pin.OUT), freq=1000, duty=512)  # Pin(5) in PWM mode here
pwm = PWM(Pin(5, Pin.OUT), freq=500, duty=256)  # Pin(5) in OUT mode here, PWM is off
```

Use this code instead:

```
pwm = PWM(Pin(5), freq=1000, duty=512)
pwm.init(freq=500, duty=256)
```

8.2.3 Accessing peripherals directly via registers

The ESP32s peripherals can be controlled via direct register reads and writes. This requires reading the datasheet to know what registers to use and what values to write to them. The following example shows how to turn on and change the prescaler of the MCPWM0 peripheral.

```
from micropython import const
from machine import mem32

# Define the register addresses that will be used.
DR_REG_DPORT_BASE = const(0x3FF00000)
DPORT_PERIP_CLK_EN_REG = const(DR_REG_DPORT_BASE + 0x0C0)
DPORT_PERIP_RST_EN_REG = const(DR_REG_DPORT_BASE + 0x0C4)
DPORT_PWM0_CLK_EN = const(1 << 17)
MCPWM0 = const(0x3FF5E000)
MCPWM1 = const(0x3FF6C000)

# Enable CLK and disable RST.
print(hex(mem32[DPORT_PERIP_CLK_EN_REG] & 0xffffffff))
print(hex(mem32[DPORT_PERIP_RST_EN_REG] & 0xffffffff))
mem32[DPORT_PERIP_CLK_EN_REG] |= DPORT_PWM0_CLK_EN
mem32[DPORT_PERIP_RST_EN_REG] &= ~DPORT_PWM0_CLK_EN
print(hex(mem32[DPORT_PERIP_CLK_EN_REG] & 0xffffffff))
print(hex(mem32[DPORT_PERIP_RST_EN_REG] & 0xffffffff))

# Change the MCPWM0 prescaler.
print(hex(mem32[MCPWM0])) # read PWM_CLK_CFG_REG (reset value = 0)
```

(continues on next page)

(continued from previous page)

```
mem32[MCPWM0] = 0x55       # change PWM_CLK_PRESCALE
print(hex(mem32[MCPWM0])) # read PWM_CLK_CFG_REG
```

Note that before a peripheral can be used its clock must be enabled and it must be taken out of reset. In the above example the following registers are used for this:

- DPORT_PERI_CLK_EN_REG: used to enable a peripheral clock
- DPORT_PERI_RST_EN_REG: used to reset (or take out of reset) a peripheral

The MCPWM0 peripheral is in bit position 17 of the above two registers, hence the value of DPORT_PWM0_CLK_EN.

8.3 Installing MicroPython

See the corresponding section of tutorial: *Getting started with MicroPython on the ESP32*. It also includes a troubleshooting subsection.

8.4 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(240000000) # set the CPU frequency to 240 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)

# low level methods to interact with flash storage
esp.flash_size()
esp.flash_user_start()
esp.flash_erase(sector_no)
esp.flash_write(byte_offset, buffer)
esp.flash_read(byte_offset, buffer)
```

The *esp32* module:

```
import esp32

esp32.hall_sensor()     # read the internal hall sensor
esp32.raw_temperature() # read the internal temperature of the MCU, in Fahrenheit
esp32.ULP()             # access to the Ultra-Low-Power Co-processor
```

Note that the temperature sensor in the ESP32 will typically read higher than ambient due to the IC getting warm while it runs. This effect can be minimised by reading the temperature sensor immediately after waking up from sleep.

8.5 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 address
wlan.ifconfig()         # get the interface's IP/netmask/gw/DNS addresses

ap = network.WLAN(network.AP_IF) # create access-point interface
ap.config(essid='ESP-AP') # set the ESSID of the access point
ap.config(max_clients=10) # set how many clients can connect to the network
ap.active(True)         # activate the interface
```

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, and the `urequests` module for convenient HTTP requests.

After a call to `wlan.connect()`, the device will by default retry to connect **forever**, even when the authentication failed or no AP is in range. `wlan.status()` will return `network.STAT_CONNECTING` in this state until a connection succeeds or the interface gets disabled. This can be changed by calling `wlan.config(reconnects=n)`, where n are the number of desired reconnect attempts (0 means it wont retry, -1 will restore the default behaviour of trying to reconnect forever).

8.6 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
```

8.7 Timers

The ESP32 port has four hardware timers. Use the *machine.Timer* class with a timer ID from 0 to 3 (inclusive):

```
from machine import Timer

tim0 = Timer(0)
tim0.init(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(0))

tim1 = Timer(1)
tim1.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(1))
```

The period is in milliseconds.

Virtual timers are not currently supported on this port.

8.8 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 from the following ranges (inclusive): 0-19, 21-23, 25-27, 32-39. These correspond to the actual GPIO pin numbers of ESP32 chip. Note that many end-user boards use their own adhoc pin numbering (marked e.g. D0, D1,). For mapping between board logical pins and physical chip pins consult your board documentation.

Notes:

- Pins 1 and 3 are REPL UART TX and RX respectively

- Pins 6, 7, 8, 11, 16, and 17 are used for connecting the embedded flash, and are not recommended for other uses
- Pins 34-39 are input only, and also do not have internal pull-up resistors
- The pull value of some pins can be set to `Pin.PULL_HOLD` to reduce power consumption during deepsleep.

Theres a higher-level abstraction *machine.Signal* which can be used to invert a pin. Useful for illuminating active-low LEDs using `on()` or `value(1)`.

8.9 UART (serial bus)

See *machine.UART*.

```
from machine import UART

uart1 = UART(1, baudrate=9600, tx=33, rx=32)
uart1.write('hello')  # write 5 bytes
uart1.read(5)         # read up to 5 bytes
```

The ESP32 has three hardware UARTs: UART0, UART1 and UART2. They each have default GPIO assigned to them, however depending on your ESP32 variant and board, these pins may conflict with embedded flash, onboard PSRAM or peripherals.

Any GPIO can be used for hardware UARTs using the GPIO matrix, so to avoid conflicts simply provide `tx` and `rx` pins when constructing. The default pins listed below.

	UART0	UART1	UART2
tx	1	10	17
rx	3	9	16

8.10 PWM (pulse width modulation)

PWM can be enabled on all output-enabled pins. The base frequency can range from 1Hz to 40MHz but there is a tradeoff; as the base frequency *increases* the duty resolution *decreases*. See LED Control for more details.

Use the *machine.PWM* class:

```
from machine import Pin, PWM

pwm0 = PWM(Pin(0))          # create PWM object from a pin
freq = pwm0.freq()          # get current frequency (default 5kHz)
pwm0.freq(1000)             # set PWM frequency from 1Hz to 40MHz

duty = pwm0.duty()          # get current duty cycle, range 0-1023 (default 512, 50%)
pwm0.duty(256)              # set duty cycle from 0 to 1023 as a ratio duty/1023, (now 25
↪%)

duty_u16 = pwm0.duty_u16() # get current duty cycle, range 0-65535
pwm0.duty_u16(2**16*3//4)  # set duty cycle from 0 to 65535 as a ratio duty_u16/65535,␣
↪(now 75%)
```

(continues on next page)

(continued from previous page)

```
duty_ns = pwm0.duty_ns()    # get current pulse width in ns
pwm0.duty_ns(250_000)       # set pulse width in nanoseconds from 0 to 1_000_000_000/freq,
↪ (now 25%)

pwm0.deinit()               # turn off PWM on the pin

pwm2 = PWM(Pin(2), freq=20000, duty=512)  # create and configure in one go
print(pwm2)                               # view PWM settings
```

ESP chips have different hardware peripherals:

Hardware specification	ESP32	ESP32-S2	ESP32-C3
Number of groups (speed modes)	2	1	1
Number of timers per group	4	4	4
Number of channels per group	8	8	6
Different PWM frequencies (groups * timers)	8	4	4
Total PWM channels (Pins, duties) (groups * channels)	16	8	6

A maximum number of PWM channels (Pins) are available on the ESP32 - 16 channels, but only 8 different PWM frequencies are available, the remaining 8 channels must have the same frequency. On the other hand, 16 independent PWM duty cycles are possible at the same frequency.

See more examples in the *Pulse Width Modulation* tutorial.

8.11 ADC (analog to digital conversion)

On the ESP32 ADC functionality is available on Pins 32-39. Note that, when using the default configuration, input voltages on the ADC pin must be between 0.0v and 1.0v (anything above 1.0v will just read as 4095). Attenuation must be applied in order to increase this usable voltage range.

Use the *machine.ADC* class:

```
from machine import ADC

adc = ADC(Pin(32))          # create ADC object on ADC pin
adc.read()                  # read value, 0-4095 across voltage range 0.0v - 1.0v

adc.atten(ADC.ATTN_11DB)    # set 11dB input attenuation (voltage range roughly 0.0v - 3.
↪6v)
adc.width(ADC.WIDTH_9BIT)   # set 9 bit return values (returned range 0-511)
adc.read()                  # read value using the newly configured attenuation and width
```

ESP32 specific ADC class method reference:

`ADC.`**`atten`**(*attenuation*)

This method allows for the setting of the amount of attenuation on the input of the ADC. This allows for a wider possible input voltage range, at the cost of accuracy (the same number of bits now represents a wider range). The possible attenuation options are:

- `ADC.ATTN_0DB`: 0dB attenuation, gives a maximum input voltage of 1.00v - this is the default configuration
- `ADC.ATTN_2_5DB`: 2.5dB attenuation, gives a maximum input voltage of approximately 1.34v
- `ADC.ATTN_6DB`: 6dB attenuation, gives a maximum input voltage of approximately 2.00v

- `ADC.ATTN_11DB`: 11dB attenuation, gives a maximum input voltage of approximately 3.6v

Warning: Despite 11dB attenuation allowing for up to a 3.6v range, note that the absolute maximum voltage rating for the input pins is 3.6v, and so going near this boundary may be damaging to the IC!

`ADC.`**`width`**(*width*)

This method allows for the setting of the number of bits to be utilised and returned during ADC reads. Possible width options are:

- `ADC.WIDTH_9BIT`: 9 bit data
- `ADC.WIDTH_10BIT`: 10 bit data
- `ADC.WIDTH_11BIT`: 11 bit data
- `ADC.WIDTH_12BIT`: 12 bit data - this is the default configuration

8.12 Software SPI bus

Software SPI (using bit-banging) works on all pins, and is accessed via the *machine.SoftSPI* class:

```
from machine import Pin, SoftSPI

# construct a SoftSPI 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 = SoftSPI(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 outputting 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
```

Warning: Currently *all* of `sck`, `mosi` and `miso` *must* be specified when initialising Software SPI.

8.13 Hardware SPI bus

There are two hardware SPI channels that allow faster transmission rates (up to 80Mhz). These may be used on any IO pins that support the required direction and are otherwise unused (see *Pins and GPIO*) but if they are not configured to their default pins then they need to pass through an extra layer of GPIO multiplexing, which can impact their reliability at high speeds. Hardware SPI channels are limited to 40MHz when used on pins other than the default ones listed below.

	HSPI (id=1)	VSPI (id=2)
sck	14	18
mosi	13	23
miso	12	19

Hardware SPI is accessed via the *machine.SPI* class and has the same methods as software SPI above:

```
from machine import Pin, SPI

hspi = SPI(1, 10000000)
hspi = SPI(1, 10000000, sck=Pin(14), mosi=Pin(13), miso=Pin(12))
vspi = SPI(2, baudrate=80000000, polarity=0, phase=0, bits=8, firstbit=0, sck=Pin(18),␣
↪mosi=Pin(23), miso=Pin(19))
```

8.14 Software I2C bus

Software I2C (using bit-banging) works on all output-capable pins, and is accessed via the *machine.SoftI2C* class:

```
from machine import Pin, SoftI2C

i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)

i2c.scan()              # scan for devices

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

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

8.15 Hardware I2C bus

There are two hardware I2C peripherals with identifiers 0 and 1. Any available output-capable pins can be used for SCL and SDA but the defaults are given below.

	I2C(0)	I2C(1)
scl	18	25
sda	19	26

The driver is accessed via the *machine.I2C* class and has the same methods as software I2C above:

```
from machine import Pin, I2C

i2c = I2C(0)
i2c = I2C(1, scl=Pin(5), sda=Pin(4), freq=400000)
```

8.16 I2S bus

See *machine.I2S*.

```
from machine import I2S, Pin

i2s = I2S(0, sck=Pin(13), ws=Pin(14), sd=Pin(34), mode=I2S.TX, bits=16, format=I2S.
↪STEREO, rate=44100, ibuf=40000) # create I2S object
i2s.write(buf)             # write buffer of audio samples to I2S device

i2s = I2S(1, sck=Pin(33), ws=Pin(25), sd=Pin(32), mode=I2S.RX, bits=16, format=I2S.MONO,␣
↪rate=22050, ibuf=40000) # create I2S object
i2s.readinto(buf)          # fill buffer with audio samples from I2S device
```

The I2S class is currently available as a Technical Preview. During the preview period, feedback from users is encouraged. Based on this feedback, the I2S class API and implementation may be changed.

ESP32 has two I2S buses with id=0 and id=1

8.17 Real time clock (RTC)

See *machine.RTC*

```
from machine import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 1, 12, 48, 0, 0)) # set a specific date and time
rtc.datetime() # get date and time
```

8.18 WDT (Watchdog timer)

See *machine.WDT*.

```
from machine import WDT

# enable the WDT with a timeout of 5s (1s is the minimum)
wdt = WDT(timeout=5000)
wdt.feed()
```

8.19 Deep-sleep mode

The following code can be used to sleep, wake and check the reset cause:

```
import machine

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

# put the device to sleep for 10 seconds
machine.deepsleep(10000)
```

Notes:

- Calling `deepsleep()` without an argument will put the device to sleep indefinitely
- A software reset does not change the reset cause
- There may be some leakage current flowing through enabled internal pullups. To further reduce power consumption it is possible to disable the internal pullups:

  ```
  p1 = Pin(4, Pin.IN, Pin.PULL_HOLD)
  ```

  After leaving deepsleep it may be necessary to un-hold the pin explicitly (e.g. if it is an output pin) via:

  ```
  p1 = Pin(4, Pin.OUT, None)
  ```

8.20 SD card

See *machine.SDCard*.

```
import machine, os

# Slot 2 uses pins sck=18, cs=5, miso=19, mosi=23
sd = machine.SDCard(slot=2)
os.mount(sd, "/sd")  # mount

os.listdir('/sd')    # list directory contents

os.umount('/sd')     # eject
```

8.21 RMT

The RMT is ESP32-specific and allows generation of accurate digital pulses with 12.5ns resolution. See *esp32.RMT* for details. Usage is:

```
import esp32
from machine import Pin

r = esp32.RMT(0, pin=Pin(18), clock_div=8)
```

(continues on next page)

(continued from previous page)

```
r   # RMT(channel=0, pin=18, source_freq=80000000, clock_div=8)
# The channel resolution is 100ns (1/(source_freq/clock_div)).
r.write_pulses((1, 20, 2, 40), start=0) # Send 0 for 100ns, 1 for 2000ns, 0 for 200ns, 1␣
↪for 4000ns
```

8.22 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.

8.23 NeoPixel and APA106 driver

Use the `neopixel` and `apa106` modules:

```
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
```

The APA106 driver extends NeoPixel, but internally uses a different colour order:

```
from apa106 import APA106
ap = APA106(pin, 8)
r, g, b = ap[0]
```

For low-level driving of a NeoPixel:

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

Warning: By default `NeoPixel` is configured to control the more popular *800kHz* units. It is possible to use alternative timing to control other (typically 400kHz) devices by passing `timing=0` when constructing the `NeoPixel` object.

APA102 (DotStar) uses a different driver as it has an additional clock pin.

8.24 Capacitive touch

Use the `TouchPad` class in the `machine` module:

```
from machine import TouchPad, Pin

t = TouchPad(Pin(14))
t.read()              # Returns a smaller number when touched
```

`TouchPad.read` returns a value relative to the capacitive variation. Small numbers (typically in the *tens*) are common when a pin is touched, larger numbers (above *one thousand*) when no touch is present. However the values are *relative* and can vary depending on the board and surrounding composition so some calibration may be required.

There are ten capacitive touch-enabled pins that can be used on the ESP32: 0, 2, 4, 12, 13 14, 15, 27, 32, 33. Trying to assign to any other pins will result in a `ValueError`.

Note that TouchPads can be used to wake an ESP32 from sleep:

```
import machine
from machine import TouchPad, Pin
import esp32

t = TouchPad(Pin(14))
t.config(500)               # configure the threshold at which the pin is considered␣
↪touched
esp32.wake_on_touch(True)
machine.lightsleep()        # put the MCU to sleep until a touchpad is touched
```

For more details on touchpads refer to Espressif Touch Sensor.

8.25 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 (řC)
d.humidity()    # eg. 41 (% RH)

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

8.26 WebREPL (web browser interactive prompt)

WebREPL (REPL over WebSockets, accessible via a web browser) is an experimental feature available in ESP32 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()

# or, start with a specific password
webrepl.start(password='mypass')
```

The WebREPL daemon listens on all active interfaces, which can be STA or AP. This allows you to connect to the ESP32 via a router (the STA interface) or directly when connected to its access point.

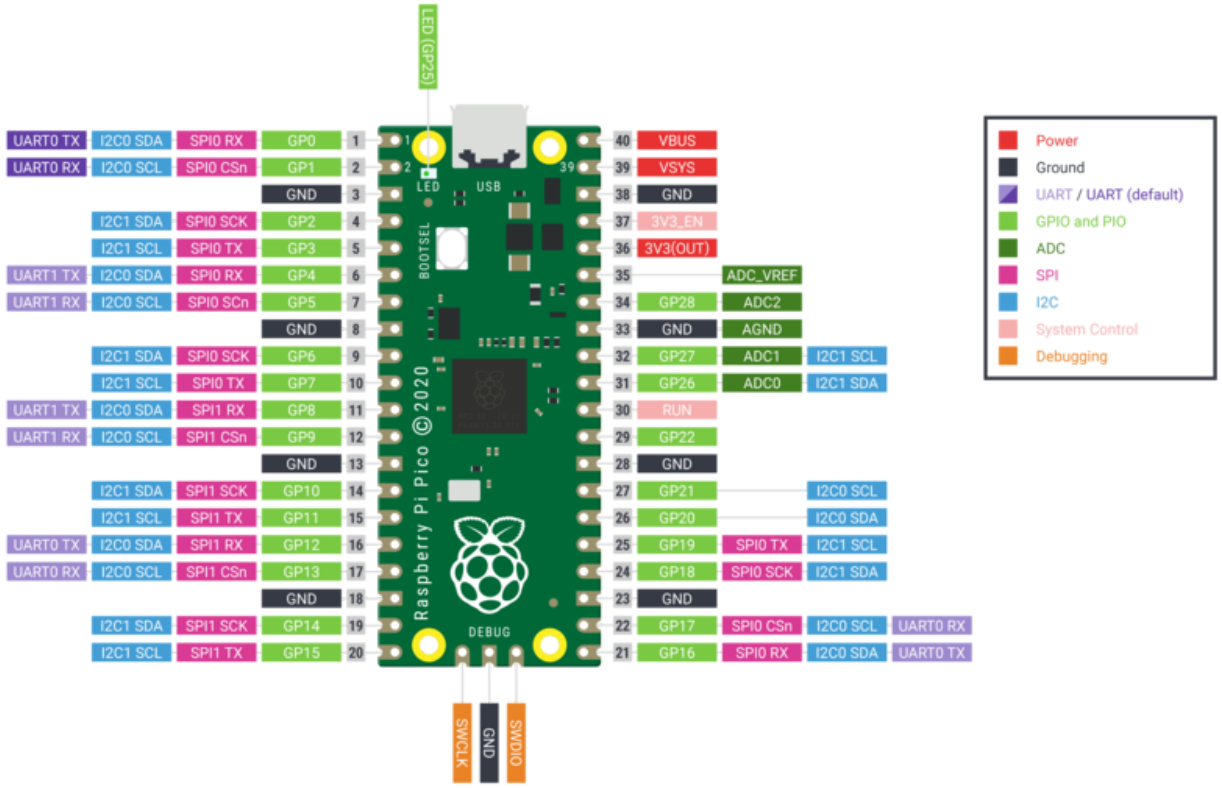
In addition to terminal/command prompt access, WebREPL also has provision for file transfer (both upload and download). The web client has buttons for the corresponding functions, or you can use the command-line client `webrepl_cli.py` from the repository above.

See the MicroPython forum for other community-supported alternatives to transfer files to an ESP32 board.

CHAPTER

NINE

QUICK REFERENCE FOR THE RP2



The Raspberry Pi Pico Development Board (image attribution: Raspberry Pi Foundation).

Below is a quick reference for Raspberry Pi RP2xxx boards. If it is your first time working with this board it may be useful to get an overview of the microcontroller:

9.1 General information about the RP2xxx port

The rp2 port supports boards powered by the Raspberry Pi Foundations RP2xxx family of microcontrollers, most notably the Raspberry Pi Pico that employs the RP2040.

9.1.1 Technical specifications and SoC datasheets

For detailed technical specifications, please refer to the datasheets

The RP2040 microcontroller is manufactured on a 40 nm silicon process in a 7x7mm QFN-56 SMD package. The key features include:

- 133 MHz dual ARM Cortex-M0+ cores (overclockable to over 400 MHz)
- 264KB SRAM in six independent banks
- No internal Flash or EEPROM memory (after reset, the bootloader loads firmware from either the external flash memory or USB bus into internal SRAM)
- QSPI bus controller, which supports up to 16 MB of external Flash memory
- On-chip programmable LDO to generate core voltage
- 2 on-chip PLLs to generate USB and core clocks
- 30 GPIO pins, of which 4 can optionally be used as analog inputs

The peripherals include:

- 2 UARTs
- 2 SPI controllers
- 2 I2C contollers
- 16 PWM channels
- USB 1.1 controller
- 8 PIO state machines

9.2 Getting started with MicroPython on the RP2xxx

Lets get started!

9.2.1 Programmable IO

The RP2040 has hardware support for standard communication protocols like I2C, SPI and UART. For protocols where there is no hardware support, or where there is a requirement of custom I/O behaviour, Programmable Input Output (PIO) comes into play. Also, some MicroPython applications make use of a technique called bit banging in which pins are rapidly turned on and off to transmit data. This can make the entire process slow as the processor concentrates on bit banging rather than executing other logic. However, PIO allows bit banging to happen in the background while the CPU is executing the main work.

Along with the two central Cortex-M0+ processing cores, the RP2040 has two PIO blocks each of which has four independent state machines. These state machines can transfer data to/from other entities using First-In-First-Out (FIFO) buffers, which allow the state machine and main processor to work independently yet also synchronise their data. Each FIFO has four words (each of 32 bits) which can be linked to the DMA to transfer larger amounts of data.

All PIO instructions follow a common pattern:

```
<instruction> .side(<side_set_value>) [<delay_value>]
```

The side-set `.side(...)` and delay `[...]` parts are both optional, and if specified allow the instruction to perform more than one operation. This keeps PIO programs small and efficient.

There are nine instructions which perform the following tasks:

- `jmp()` transfers control to a different part of the code
- `wait()` pauses until a particular action happens
- `in_()` shifts the bits from a source (scratch register or set of pins) to the input shift register
- `out()` shifts the bits from the output shift register to a destination
- `push()` sends data to the RX FIFO
- `pull()` receives data from the TX FIFO
- `mov()` moves data from a source to a destination
- `irq()` sets or clears an IRQ flag
- `set()` writes a literal value to a destination

The instruction modifiers are:

- `.side()` sets the side-set pins at the start of the instruction
- `[]` delays for a certain number of cycles after execution of the instruction

There are also directives:

- `wrap_target()` specifies where the program execution will get continued from
- `wrap()` specifies the instruction where the control flow of the program will get wrapped from
- `label()` sets a label for use with `jmp()` instructions
- `word()` emits a raw 16-bit value which acts as an instruction in the program

An example

Take the `pio_1hz.py` example for a simple understanding of how to use the PIO and state machines. Below is the code for reference.

```
# Example using PIO to blink an LED and raise an IRQ at 1Hz.

import time
from machine import Pin
import rp2


@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink_1hz():
    # Cycles: 1 + 1 + 6 + 32 * (30 + 1) = 1000
    irq(rel(0))
    set(pins, 1)
    set(x, 31)                  [5]
    label("delay_high")
```

(continues on next page)

(continued from previous page)

```
    nop()                       [29]
    jmp(x_dec, "delay_high")

    # Cycles: 1 + 7 + 32 * (30 + 1) = 1000
    set(pins, 0)
    set(x, 31)                  [6]
    label("delay_low")
    nop()                       [29]
    jmp(x_dec, "delay_low")


# Create the StateMachine with the blink_1hz program, outputting on Pin(25).
sm = rp2.StateMachine(0, blink_1hz, freq=2000, set_base=Pin(25))

# Set the IRQ handler to print the millisecond timestamp.
sm.irq(lambda p: print(time.ticks_ms()))

# Start the StateMachine.
sm.active(1)
```

This creates an instance of class *rp2.StateMachine* which runs the `blink_1hz` program at 2000Hz, and connects to pin 25. The `blink_1hz` program uses the PIO to blink an LED connected to this pin at 1Hz, and also raises an IRQ as the LED turns on. This IRQ then calls the `lambda` function which prints out a millisecond timestamp.

The `blink_1hz` program is a PIO assembler routine. It connects to a single pin which is configured as an output and starts out low. The instructions do the following:

- `irq(rel(0))` raises the IRQ associated with the state machine.
- The LED is turned on via the `set(pins, 1)` instruction.
- The value 31 is put into register X, and then there is a delay for 5 more cycles, specified by the `[5]`.
- The `nop() [29]` instruction waits for 30 cycles.
- The `jmp(x_dec, "delay_high")` will keep looping to the `delay_high` label as long as the register X is non-zero, and will also post-decrement X. Since X starts with the value 31 this jump will happen 31 times, so the `nop() [29]` runs 32 times in total (note there is also one instruction cycle taken by the `jmp` for each of these 32 loops).
- `set(pins, 0)` will turn the LED off by setting pin 25 low.
- Another 32 loops of `nop() [29]` and `jmp(...)` will execute.
- Because `wrap_target()` and `wrap()` are not specified, their default will be used and execution of the program will wrap around from the bottom to the top. This wrapping does not cost any execution cycles.

The entire routine takes exactly 2000 cycles of the state machine. Setting the frequency of the state machine to 2000Hz makes the LED blink at 1Hz.

9.3 Installing MicroPython

See the corresponding section of tutorial: *Getting started with MicroPython on the RP2xxx*. It also includes a troubleshooting subsection.

9.4 General board control

The MicroPython REPL is accessed via the USB serial port. 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(240000000) # set the CPU frequency to 240 MHz
```

The *rp2* module:

```
import rp2
```

9.5 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
```

9.6 Timers

RP2040s system timer peripheral provides a global microsecond timebase and generates interrupts for it. The software timer is available currently, and there are unlimited number of them (memory permitting). There is no need to specify the timer id (id=-1 is supported at the moment) as it will default to this.

Use the *machine.Timer* class:

```
from machine import Timer

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

9.7 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
```

9.8 Programmable IO (PIO)

PIO is useful to build low-level IO interfaces from scratch. See the *rp2* module for detailed explaination of the assembly instructions.

Example using PIO to blink an LED at 1Hz:

```
from machine import Pin
import rp2

@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink_1hz():
    # Cycles: 1 + 7 + 32 * (30 + 1) = 1000
    set(pins, 1)
    set(x, 31)                  [6]
    label("delay_high")
    nop()                       [29]
    jmp(x_dec, "delay_high")

    # Cycles: 1 + 7 + 32 * (30 + 1) = 1000
    set(pins, 0)
    set(x, 31)                  [6]
    label("delay_low")
    nop()                       [29]
    jmp(x_dec, "delay_low")

# Create and start a StateMachine with blink_1hz, outputting on Pin(25)
sm = rp2.StateMachine(0, blink_1hz, freq=2000, set_base=Pin(25))
sm.active(1)
```

9.9 UART (serial bus)

There are two UARTs, UART0 and UART1. UART0 can be mapped to GPIO 0/1, 12/13 and 16/17, and UART1 to GPIO 4/5 and 8/9.

See *machine.UART*.

```
from machine import UART, Pin
uart1 = UART(1, baudrate=9600, tx=Pin(4), rx=Pin(5))
uart1.write('hello')  # write 5 bytes
uart1.read(5)         # read up to 5 bytes
```

Note: REPL over UART is disabled by default. You can see the *Getting started with MicroPython on the RP2xxx* for details on how to enable REPL over UART.

9.10 PWM (pulse width modulation)

There are 8 independent channels each of which have 2 outputs making it 16 PWM channels in total which can be clocked from 7Hz to 125Mhz.

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_u16()         # get current duty cycle, range 0-65535
pwm0.duty_u16(200)      # set duty cycle, range 0-65535
pwm0.deinit()           # turn off PWM on the pin
```

9.11 ADC (analog to digital conversion)

RP2040 has five ADC channels in total, four of which are 12-bit SAR based ADCs: GP26, GP27, GP28 and GP29. The input signal for ADC0, ADC1, ADC2 and ADC3 can be connected with GP26, GP27, GP28, GP29 respectively (On Pico board, GP29 is connected to VSYS). The standard ADC range is 0-3.3V. The fifth channel is connected to the in-built temperature sensor and can be used for measuring the temperature.

Use the *machine.ADC* class:

```
from machine import ADC, Pin
adc = ADC(Pin(26))      # create ADC object on ADC pin
adc.read_u16()          # read value, 0-65535 across voltage range 0.0v - 3.3v
```

9.12 Software SPI bus

Software SPI (using bit-banging) works on all pins, and is accessed via the *machine.SoftSPI* class:

```
from machine import Pin, SoftSPI

# construct a SoftSPI 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 = SoftSPI(baudrate=100_000, 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 outputting 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
```

Warning: Currently *all* of `sck`, `mosi` and `miso` *must* be specified when initialising Software SPI.

9.13 Hardware SPI bus

The RP2040 has 2 hardware SPI buses which is accessed via the *machine.SPI* class and has the same methods as software SPI above:

```
from machine import Pin, SPI

spi = SPI(1, 10_000_000)  # Default assignment: sck=Pin(10), mosi=Pin(11), miso=Pin(8)
spi = SPI(1, 10_000_000, sck=Pin(14), mosi=Pin(15), miso=Pin(12))
spi = SPI(0, baudrate=80_000_000, polarity=0, phase=0, bits=8, sck=Pin(6), mosi=Pin(7),␣
↪miso=Pin(4))
```

9.14 Software I2C bus

Software I2C (using bit-banging) works on all output-capable pins, and is accessed via the *machine.SoftI2C* class:

```
from machine import Pin, SoftI2C

i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100_000)

i2c.scan()              # scan for devices

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

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

9.15 Hardware I2C bus

The driver is accessed via the *machine.I2C* class and has the same methods as software I2C above:

```
from machine import Pin, I2C

i2c = I2C(0)   # default assignment: scl=Pin(9), sda=Pin(8)
i2c = I2C(1, scl=Pin(3), sda=Pin(2), freq=400_000)
```

9.16 I2S bus

See *machine.I2S*.

```
from machine import I2S, Pin

i2s = I2S(0, sck=Pin(16), ws=Pin(17), sd=Pin(18), mode=I2S.TX, bits=16, format=I2S.
↪STEREO, rate=44100, ibuf=40000) # create I2S object
i2s.write(buf)             # write buffer of audio samples to I2S device

i2s = I2S(1, sck=Pin(0), ws=Pin(1), sd=Pin(2), mode=I2S.RX, bits=16, format=I2S.MONO,␣
↪rate=22050, ibuf=40000) # create I2S object
i2s.readinto(buf)          # fill buffer with audio samples from I2S device
```

The `ws` pin number must be one greater than the `sck` pin number.

The I2S class is currently available as a Technical Preview. During the preview period, feedback from users is encouraged. Based on this feedback, the I2S class API and implementation may be changed.

Two I2S buses are supported with id=0 and id=1.

9.17 Real time clock (RTC)

See *machine.RTC*

```
from machine import RTC

rtc = RTC()
rtc.datetime((2017, 8, 23, 2, 12, 48, 0, 0)) # set a specific date and
                                             # time, eg. 2017/8/23 1:12:48
rtc.datetime() # get date and time
```

9.18 WDT (Watchdog timer)

The RP2040 has a watchdog which is a countdown timer that can restart parts of the chip if it reaches zero.

See *machine.WDT*.

```
from machine import WDT

# enable the WDT with a timeout of 5s (1s is the minimum)
wdt = WDT(timeout=5000)
wdt.feed()
```

9.19 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.

9.20 NeoPixel and APA106 driver

Use the `neopixel` and `apa106` modules:

```
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
```

The APA106 driver extends NeoPixel, but internally uses a different colour order:

```
from apa106 import APA106
ap = APA106(pin, 8)
r, g, b = ap[0]
```

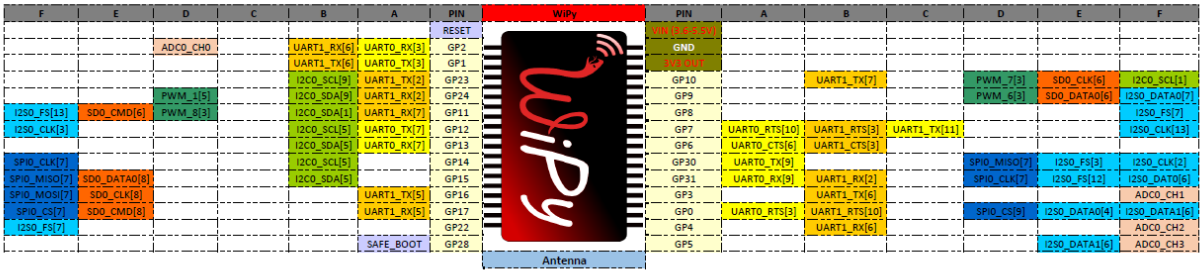
APA102 (DotStar) uses a different driver as it has an additional clock pin.

CHAPTER

TEN

QUICK REFERENCE FOR THE WIPY

WiPy pinout and alternate functions table



Timer	Channel	PWM pin
0	A	PWM_1
	B	
1	A	PWM_3
	B	
2	A	
	B	PWM_6
3	A	PWM_7
	B	PWM_8

Remarks:

- *The number in brackets next to each function is the one to be used when remapping the pin. In order to use the pin in GPIO mode, alternate function 0 must be selected*
- *ADC pin input range is 0-1.4V (being 1.8V the absolute maximum that it can withstand). When GP2, GP3, GP4 or GP5 are remapped to the ADC block, 1.8 V is the maximum. If these pins are used in digital mode, then the maximum allowed input is 3.6V.*
- *The heart beat LED is connected to GP25 and also has PWM_3 functionality with the alternate function 9.*

Below is a quick reference for CC3200/WiPy. If it is your first time working with this board please consider reading the following sections first:

10.1 General information about the WiPy

10.1.1 No floating point support

Due to space reasons, theres no floating point support, and no math module. This means that floating point numbers cannot be used anywhere in the code, and that all divisions must be performed using // instead of /. Example:

```
>>> r = 4 // 2  # this will work
>>> r = 4 / 2   # this WON'T
```

10.1.2 Before applying power

Warning: The GPIO pins of the WiPy are NOT 5V tolerant, connecting them to voltages higher than 3.6V will cause irreparable damage to the board. ADC pins, when configured in analog mode cannot withstand voltages above 1.8V. Keep these considerations in mind when wiring your electronics.

10.1.3 WLAN default behaviour

When the WiPy boots with the default factory configuration starts in Access Point mode with `ssid` that starts with: `wipy-wlan` and `key: www.wipy.io`. Connect to this network and the WiPy will be reachable at `192.168.1.1`. In order to gain access to the interactive prompt, open a telnet session to that IP address on the default port (23). You will be asked for credentials: `login: micro` and `password: python`

10.1.4 Telnet REPL

Linux stock telnet works like a charm (also on OSX), but other tools like putty work quite well too. The default credentials are: **user:** `micro`, **password:** `python`. See *network.Server* for info on how to change the defaults. For instance, on a linux shell (when connected to the WiPy in AP mode):

```
$ telnet 192.168.1.1
```

10.1.5 Local file system and FTP access

There is a small internal file system (a drive) on the WiPy, called `/flash`, which is stored within the external serial flash memory. If a micro SD card is hooked-up and mounted, it will be available as well.

When the WiPy starts up, it always boots from the `boot.py` located in the `/flash` file system. On boot up, the current directory is `/flash`.

The file system is accessible via the native FTP server running in the WiPy. Open your FTP client of choice and connect to:

url: `ftp://192.168.1.1`, **user:** `micro`, **password:** `python`

See *network.Server* for info on how to change the defaults. The recommended clients are: Linux stock FTP (also in OSX), Filezilla and FireFTP. For example, on a linux shell:

```
$ ftp 192.168.1.1
```

The FTP server on the WiPy doesnt support active mode, only passive, therefore, if using the native unix ftp client, just after logging in do:

```
ftp> passive
```

Besides that, the FTP server only supports one data connection at a time. Check out the Filezilla settings section below for more info.

10.1.6 FileZilla settings

Do not use the quick connect button, instead, open the site manager and create a new configuration. In the `General` tab make sure that encryption is set to: `Only use plain FTP (insecure)`. In the Transfer Settings tab limit the max number of connections to one, otherwise FileZilla will try to open a second command connection when retrieving and saving files, and for simplicity and to reduce code size, only one command and one data connections are possible. Other FTP clients might behave in a similar way.

10.1.7 Upgrading the firmware Over The Air

OTA software updates can be performed through the FTP server. Upload the `mcuimg.bin` file to: `/flash/sys/mcuimg.bin` it will take around 6s. You wont see the file being stored inside `/flash/sys/` because its actually saved bypassing the user file system, so it ends up inside the internal **hidden** file system, but rest assured that it was successfully transferred, and it has been signed with a MD5 checksum to verify its integrity. Now, reset the WiPy by pressing the switch on the board, or by typing:

```
>>> import machine
>>> machine.reset()
```

Software updates can be found in: https://github.com/wipy/wipy/releases (**Binaries.zip**). Its always recommended to update to the latest software, but make sure to read the **release notes** before.

Note: The `bootloader.bin` found inside `Binaries.zip` is there only for reference, its not needed for the Over The Air update.

In order to check your software version, do:

```
>>> import os
>>> os.uname().release
```

If the version number is lower than the latest release found in the releases, go ahead and update your WiPy!

10.1.8 Boot modes and safe boot

If you power up normally, or press the reset button, the WiPy will boot into standard mode; the `boot.py` file will be executed first, then `main.py` will run.

You can override this boot sequence by pulling `GP28` **up** (connect it to the 3v3 output pin) during reset. This procedure also allows going back in time to old firmware versions. The WiPy can hold up to 3 different firmware versions, which are: the factory firmware plus 2 user updates.

After reset, if `GP28` is held high, the heartbeat LED will start flashing slowly, if after 3 seconds the pin is still being held high, the LED will start blinking a bit faster and the WiPy will select the previous user update to boot. If the previous user update is the desired firmware image, `GP28` must be released before 3 more seconds elapse. If 3 seconds later the pin is still high, the factory firmware will be selected, the LED will flash quickly for 1.5 seconds and the WiPy will proceed to boot. The firmware selection mechanism is as follows:

Safe Boot Pin `GP28` released during:

1st 3 secs window	2nd 3 secs window	Final 1.5 secs window
Safe boot, *latest* firmware is selected	Safe boot, *previous* user update selected	Safe boot, the *factory* firmware is selected

On all of the above 3 scenarios, safe boot mode is entered, meaning that the execution of both `boot.py` and `main.py` is skipped. This is useful to recover from crash situations caused by the user scripts. The selection made during safe boot is not persistent, therefore after the next normal reset the latest firmware will run again.

10.1.9 The heartbeat LED

By default the heartbeat LED flashes once every 4s to signal that the system is alive. This can be overridden through the *wipy* module:

```
>>> import wipy
>>> wipy.heartbeat(False)
```

There are currently 2 kinds of errors that you might see:

1. If the heartbeat LED flashes quickly, then a Python script (eg `main.py`) has an error. Use the REPL to debug it.
2. If the heartbeat LED stays on, then there was a hard fault, you cannot recover from this, the only way out is to press the reset switch.

10.1.10 Details on sleep modes

- `machine.idle()`: Power consumption: ~12mA (in WLAN STA mode). Wake sources: any hardware interrupt (including systick with period of 1ms), no special configuration required.
- `machine.lightsleep()`: 950uA (in WLAN STA mode). Wake sources are `Pin`, `RTC` and `WLAN`
- `machine.deepsleep()`: ~350uA. Wake sources are `Pin` and `RTC`.

10.1.11 Additional details for machine.Pin

On the WiPy board the pins are identified by their string id:

```
from machine import Pin
g = machine.Pin('GP9', mode=Pin.OUT, pull=None, drive=Pin.MED_POWER, alt=-1)
```

You can also configure the Pin to generate interrupts. For instance:

```
from machine import Pin

def pincb(pin):
    print(pin.id())

pin_int = Pin('GP10', mode=Pin.IN, pull=Pin.PULL_DOWN)
pin_int.irq(trigger=Pin.IRQ_RISING, handler=pincb)
# the callback can be triggered manually
```

(continues on next page)

(continued from previous page)

```
pin_int.irq()()
# to disable the callback
pin_int.irq().disable()
```

Now every time a falling edge is seen on the gpio pin, the callback will be executed. Caution: mechanical push buttons have bounce and pushing or releasing a switch will often generate multiple edges. See: http://www.eng.utah.edu/~cs5780/debouncing.pdf for a detailed explanation, along with various techniques for debouncing.

All pin objects go through the pin mapper to come up with one of the gpio pins.

For the `drive` parameter the strengths are:

- `Pin.LOW_POWER` - 2mA drive capability.
- `Pin.MED_POWER` - 4mA drive capability.
- `Pin.HIGH_POWER` - 6mA drive capability.

For the `alt` parameter please refer to the pinout and alternate functions table at <https://raw.githubusercontent.com/wipy/wipy/master/docs/PinOUT.png>`_ for the specific alternate functions that each pin supports.

For interrupts, the `priority` can take values in the range 1-7. And the `wake` parameter has the following properties:

- If `wake_from=machine.Sleep.ACTIVE` any pin can wake the board.
- If `wake_from=machine.Sleep.SUSPENDED` pins `GP2`, `GP4`, `GP10`, `GP11`, GP17`` or `GP24` can wake the board. Note that only 1 of this pins can be enabled as a wake source at the same time, so, only the last enabled pin as a `machine.Sleep.SUSPENDED` wake source will have effect.
- If `wake_from=machine.Sleep.SUSPENDED` pins `GP2`, `GP4`, `GP10`, `GP11`, `GP17` and `GP24` can wake the board. In this case all of the 6 pins can be enabled as a `machine.Sleep.HIBERNATE` wake source at the same time.

Additional Pin methods:

`machine.Pin.alt_list()`
: Returns a list of the alternate functions supported by the pin. List items are a tuple of the form: `('ALT_FUN_NAME', ALT_FUN_INDEX)`

10.1.12 Additional details for machine.I2C

On the WiPy there is a single hardware I2C peripheral, identified by 0. By default this is the peripheral that is used when constructing an I2C instance. The default pins are GP23 for SCL and GP13 for SDA, and one can create the default I2C peripheral simply by doing:

```
i2c = machine.I2C()
```

The pins and frequency can be specified as:

```
i2c = machine.I2C(freq=400000, scl='GP23', sda='GP13')
```

Only certain pins can be used as SCL/SDA. Please refer to the pinout for further information.

10.1.13 Known issues

Incompatible way to create SSL sockets

SSL sockets need to be created the following way before wrapping them with. `ssl.wrap_socket`:

```
import socket
import ssl
s = socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_SEC)
ss = ssl.wrap_socket(s)
```

Certificates must be used in order to validate the other side of the connection, and also to authenticate ourselves with the other end. Such certificates must be stored as files using the FTP server, and they must be placed in specific paths with specific names.

- The certificate to validate the other side goes in: **/flash/cert/ca.pem**
- The certificate to authenticate ourselves goes in: **/flash/cert/cert.pem**
- The key for our own certificate goes in: **/flash/cert/private.key**

Note: When these files are stored, they are placed inside the internal **hidden** file system (just like firmware updates), and therefore they are never visible.

For instance to connect to the Blynk servers using certificates, take the file `ca.pem` located in the blynk examples folder. and put it in /flash/cert/. Then do:

```
import socket
import ssl
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_SEC)
ss = ssl.wrap_socket(s, cert_reqs=ssl.CERT_REQUIRED, ca_certs='/flash/cert/ca.pem')
ss.connect(socket.getaddrinfo('cloud.blynk.cc', 8441)[0][-1])
```

Incompatibilities in hashlib module

Due to hardware implementation details of the WiPy, data must be buffered before being digested, which would make it impossible to calculate the hash of big blocks of data that do not fit in RAM. In this case, since most likely the total size of the data is known in advance, the size can be passed to the constructor and hence the HASH hardware engine of the WiPy can be properly initialized without needing buffering. If `block_size` is to be given, an initial chunk of `data` must be passed as well. **When using this extension, care must be taken to make sure that the length of all intermediate chunks (including the initial one) is a multiple of 4 bytes.** The last chunk may be of any length.

Example:

```
hash = hashlib.sha1('abcd1234', 1001)    # length of the initial piece is multiple of 4␣
↪bytes
hash.update('1234')                       # also multiple of 4 bytes
...
hash.update('12345')                      # last chunk may be of any length
hash.digest()
```

Unrelated function in machine module

main(*filename*)
Set the filename of the main script to run after boot.py is finished. If this function is not called then the default file main.py will be executed.

It only makes sense to call this function from within boot.py.

Adhoc way to control telnet/FTP server via network module

The `Server` class controls the behaviour and the configuration of the FTP and telnet services running on the WiPy. Any changes performed using this class methods will affect both.

Example:

```
import network
server = network.Server()
server.deinit() # disable the server
# enable the server again with new settings
server.init(login=('user', 'password'), timeout=600)
```

class network.**Server**(*id*, ...)
Create a server instance, see `init` for parameters of initialization.

server.**init**(*, *login=('micro', 'python')*, *timeout=300*)
Init (and effectively start the server). Optionally a new `user`, `password` and `timeout` (in seconds) can be passed.

server.**deinit**()
Stop the server

server.**timeout**([*timeout_in_seconds*])
Get or set the server timeout.

server.**isrunning**()
Returns `True` if the server is running, `False` otherwise.

Adhoc VFS-like support

WiPy doesnt implement full MicroPython VFS support, instead following functions are defined in `os` module:

mount(*block_device*, *mount_point*, *, *readonly=False*)
Mounts a block device (like an `SD` object) in the specified mount point. Example:

```
os.mount(sd, '/sd')
```

unmount(*path*)
Unmounts a previously mounted block device from the given path.

mkfs(*block_device or path*)
Formats the specified path, must be either `/flash` or `/sd`. A block device can also be passed like an `SD` object before being mounted.

10.2 WiPy tutorials and examples

Before starting, make sure that you are running the latest firmware, for instructions see *OTA How-To*.

10.2.1 Introduction to the WiPy

To get the most out of your WiPy, there are a few basic things to understand about how it works.

Caring for your WiPy and expansion board

Because the WiPy/expansion board does not have a housing it needs a bit of care:

- Be gentle when plugging/unplugging the USB cable. Whilst the USB connector is well soldered and is relatively strong, if it breaks off it can be very difficult to fix.
- Static electricity can shock the components on the WiPy and destroy them. If you experience a lot of static electricity in your area (eg dry and cold climates), take extra care not to shock the WiPy. If your WiPy came in a ESD bag, then this bag is the best way to store and carry the WiPy as it will protect it against static discharges.

As long as you take care of the hardware, you should be okay. Its almost impossible to break the software on the WiPy, so feel free to play around with writing code as much as you like. If the filesystem gets corrupt, see below on how to reset it. In the worst case you might need to do a safe boot, which is explained in detail in *Boot modes and safe boot*.

Plugging into the expansion board and powering on

The expansion board can power the WiPy via USB. The WiPy comes with a sticker on top of the RF shield that labels all pins, and this should match the label numbers on the expansion board headers. When plugging it in, the WiPy antenna will end up on top of the SD card connector of the expansion board. A video showing how to do this can be found here on YouTube.

Expansion board hardware guide

The document explaining the hardware details of the expansion board can be found in this PDF.

Powering by an external power source

The WiPy can be powered by a battery or other external power source.

Be sure to connect the positive lead of the power supply to VIN, and ground to GND. There is no polarity protection on the WiPy so you must be careful when connecting anything to VIN.

- When powering via `VIN`:

  The input voltage must be between 3.6V and 5.5V.

- When powering via `3V3`:

  The input voltage must be exactly 3V3, ripple free and from a supply capable of sourcing at least 300mA of current

Performing firmware upgrades

For detailed instructions see *OTA How-To*.

10.2.2 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 WiPy. Using the REPL is by far the easiest way to test out your code and run commands. You can use the REPL in addition to writing scripts in `main.py`.

To use the REPL, you must connect to the WiPy either via *telnet*, or with a USB to serial converter wired to one of the two UARTs on the WiPy. To enable REPL duplication on UART0 (the one accessible via the expansion board) do:

```
>>> from machine import UART
>>> import os
>>> uart = UART(0, 115200)
>>> os.dupterm(uart)
```

Place this piece of code inside your `boot.py` so that its done automatically after reset.

Windows

First you need to install the FTDI drivers for the expansion boards USB to serial converter. Then you need a terminal software. The best option is to download the free program PuTTY: putty.exe.

In order to get to the telnet REPL:

Using putty, select `Telnet` as connection type, leave the default port (23) and enter the IP address of your WiPy (192.168.1.1 when in `WLAN.AP` mode), then click open.

In order to get to the REPL UART:

Using your serial program you must connect to the COM port that you found in the previous step. With PuTTY, click on Session in the left-hand panel, then click the Serial radio button on the right, then enter you COM port (eg COM4) in the Serial Line box. Finally, click the Open button.

Mac OS X

Open a terminal and run:

```
$ telnet 192.168.1.1
```

or:

```
$ screen /dev/tty.usbmodem* 115200
```

When you are finished and want to exit `screen`, type CTRL-A CTRL-\. If your keyboard does not have a \-key (i.e. you need an obscure combination for \ like ALT-SHIFT-7) you can remap the `quit` command:

- create `~/.screenrc`
- add `bind q quit`

This will allow you to quit `screen` by hitting CTRL-A Q.

Linux

Open a terminal and run:

```
$ telnet 192.168.1.1
```

or:

```
$ screen /dev/ttyUSB0 115200
```

You can also try `picocom` or `minicom` instead of screen. You may have to use `/dev/ttyUSB01` or a higher number for `ttyUSB`. And, you may need to give yourself the correct permissions to access this devices (eg group `uucp` or `dialout`, or use sudo).

Using the REPL prompt

Now lets try running some MicroPython code directly on the WiPy.

With your serial program open (PuTTY, screen, picocom, etc) you may see a blank screen with a flashing cursor. Press Enter and you should be presented with a MicroPython prompt, i.e. >>>. Lets make sure it is working with the obligatory test:

```
>>> print("hello WiPy!")
hello WiPy!
```

In the above, you should not type in the >>> characters. They are there to indicate that you should type the text after it at the prompt. In the end, once you have entered the text `print("hello WiPy!")` and pressed Enter, the output on your screen should look like it does above.

If you already know some Python you can now try some basic commands here.

If any of this is not working you can try either a hard reset or a soft reset; see below.

Go ahead and try typing in some other commands. For example:

```
>>> from machine import Pin
>>> import wipy
>>> wipy.heartbeat(False)  # disable the heartbeat
>>> led = Pin('GP25', mode=Pin.OUT)
>>> led(1)
>>> led(0)
>>> led.toggle()
>>> 1 + 2
3
>>> 4 // 2
2
>>> 20 * 'py'
'pypypypypypypypypypypypypypypypypypypypy'
```

Resetting the board

If something goes wrong, you can reset the board in two ways. The first is to press CTRL-D at the MicroPython prompt, which performs a soft reset. You will see a message something like:

```
>>>
MPY: soft reboot
MicroPython v1.4.6-146-g1d8b5e5 on 2015-10-21; WiPy with CC3200
Type "help()" for more information.
>>>
```

If that isnt working you can perform a hard reset (turn-it-off-and-on-again) by pressing the RST switch (the small black button next to the heartbeat LED). During telnet, this will end your session, disconnecting whatever program that you used to connect to the WiPy.

10.2.3 Getting started with Blynk and the WiPy

Blynk provides iOS and Android apps to control any hardware over the Internet or directly using Bluetooth. You can easily build graphic interfaces for all your projects by simply dragging and dropping widgets, right on your smartphone.

Before anything else, make sure that your WiPy is running the latest software, check *OTA How-To* for instructions.

1. Get the Blynk library and put it in `/flash/lib/` via FTP.
2. Get the Blynk example for WiPy, edit the network settings, and afterwards upload it to `/flash/` via FTP as well.
3. Follow the instructions on each example to setup the Blynk dashboard on your smartphone or tablet.
4. Give it a try, for instance:

```
>>> execfile('sync_virtual.py')
```

10.2.4 WLAN step by step

The WLAN is a system feature of the WiPy, therefore it is always enabled (even while in `machine.SLEEP`), except when deepsleep mode is entered.

In order to retrieve the current WLAN instance, do:

```
>>> from network import WLAN
>>> wlan = WLAN() # we call the constructor without params
```

You can check the current mode (which is always `WLAN.AP` after power up):

```
>>> wlan.mode()
```

Warning: When you change the WLAN mode following the instructions below, your WLAN connection to the WiPy will be broken. This means you will not be able to run these commands interactively over the WLAN.

There are two ways around this::

1. put this setup code into your *boot.py file* so that it gets executed automatically after reset.
2. *duplicate the REPL on UART*, so that you can run commands via USB.

Connecting to your home router

The WLAN network card always boots in `WLAN.AP` mode, so we must first configure it as a station:

```
from network import WLAN
wlan = WLAN(mode=WLAN.STA)
```

Now you can proceed to scan for networks:

```
nets = wlan.scan()
for net in nets:
    if net.ssid == 'mywifi':
        print('Network found!')
        wlan.connect(net.ssid, auth=(net.sec, 'mywifikey'), timeout=5000)
        while not wlan.isconnected():
            machine.idle() # save power while waiting
        print('WLAN connection succeeded!')
        break
```

Assigning a static IP address when booting

If you want your WiPy to connect to your home router after boot-up, and with a fixed IP address so that you can access it via telnet or FTP, use the following script as /flash/boot.py:

```
import machine
from network import WLAN
wlan = WLAN() # get current object, without changing the mode

if machine.reset_cause() != machine.SOFT_RESET:
    wlan.init(WLAN.STA)
    # configuration below MUST match your home router settings!!
    wlan.ifconfig(config=('192.168.178.107', '255.255.255.0', '192.168.178.1', '8.8.8.8
↪'))

if not wlan.isconnected():
    # change the line below to match your network ssid, security and password
    wlan.connect('mywifi', auth=(WLAN.WPA2, 'mywifikey'), timeout=5000)
    while not wlan.isconnected():
        machine.idle() # save power while waiting
```

Note: Notice how we check for the reset cause and the connection status, this is crucial in order to be able to soft reset the WiPy during a telnet session without breaking the connection.

10.2.5 Hardware timers

Timers can be used for a great variety of tasks, calling a function periodically, counting events, and generating a PWM signal are among the most common use cases. Each timer consists of two 16-bit channels and this channels can be tied together to form one 32-bit timer. The operating mode needs to be configured per timer, but then the period (or the frequency) can be independently configured on each channel. By using the callback method, the timer event can call a Python function.

Example usage to toggle an LED at a fixed frequency:

```
from machine import Timer
from machine import Pin
led = Pin('GP16', mode=Pin.OUT)                    # enable GP16 as output to drive the LED
tim = Timer(3)                                     # create a timer object using timer 3
tim.init(mode=Timer.PERIODIC)                      # initialize it in periodic mode
tim_ch = tim.channel(Timer.A, freq=5)              # configure channel A at a frequency of␣
↪5Hz
tim_ch.irq(handler=lambda t:led.toggle(), trigger=Timer.TIMEOUT)        # toggle a LED␣
↪on every cycle of the timer
```

Example using named function for the callback:

```
from machine import Timer
from machine import Pin
tim = Timer(1, mode=Timer.PERIODIC, width=32)
tim_a = tim.channel(Timer.A | Timer.B, freq=1)    # 1 Hz frequency requires a 32 bit timer

led = Pin('GP16', mode=Pin.OUT) # enable GP16 as output to drive the LED

def tick(timer):                # we will receive the timer object when being called
    global led
    led.toggle()                # toggle the LED

tim_a.irq(handler=tick, trigger=Timer.TIMEOUT)         # create the interrupt
```

Further examples:

```
from machine import Timer
tim1 = Timer(1, mode=Timer.ONE_SHOT)                                    # initialize it in␣
↪one shot mode
tim2 = Timer(2, mode=Timer.PWM)                                         # initialize it in␣
↪PWM mode
tim1_ch = tim1.channel(Timer.A, freq=10, polarity=Timer.POSITIVE)  # start the event␣
↪counter with a frequency of 10Hz and triggered by positive edges
tim2_ch = tim2.channel(Timer.B, freq=10000, duty_cycle=5000)       # start the PWM on␣
↪channel B with a 50% duty cycle
tim2_ch.freq(20)                                                        # set the frequency␣
↪(can also get)
tim2_ch.duty_cycle(3010)                                                # set the duty cycle␣
↪to 30.1% (can also get)
tim2_ch.duty_cycle(3020, Timer.NEGATIVE)                                # set the duty cycle␣
↪to 30.2% and change the polarity to negative
tim2_ch.period(2000000)                                                 # change the period␣
↪to 2 seconds
```

Additional constants for Timer class

`Timer.`**`PWM`**
PWM timer operating mode.

`Timer.`**`A`**

`Timer.`**`B`**
Selects the timer channel. Must be ORed (`Timer.A | Timer.B`) when using a 32-bit timer.

`Timer.`**`POSITIVE`**

`Timer.`**`NEGATIVE`**
Timer channel polarity selection (only relevant in PWM mode).

`Timer.`**`TIMEOUT`**

`Timer.`**`MATCH`**
Timer channel IRQ triggers.

10.2.6 Reset and boot modes

There are soft resets and hard resets.

- A soft reset simply clears the state of the MicroPython virtual machine, but leaves hardware peripherals unaffected. To do a soft reset, simply press **Ctrl+D** on the REPL, or within a script do:

```
import sys
sys.exit()
```

- A hard reset is the same as performing a power cycle to the board. In order to hard reset the WiPy, press the switch on the board or:

```
import machine
machine.reset()
```

Safe boot

If something goes wrong with your WiPy, dont panic! It is almost impossible for you to break the WiPy by programming the wrong thing.

The first thing to try is to boot in safe mode: this temporarily skips execution of `boot.py` and `main.py` and gives default WLAN settings.

If you have problems with the filesystem you can *format the internal flash drive*.

To boot in safe mode, follow the detailed instructions described *here*.

In safe mode, the `boot.py` and `main.py` files are not executed, and so the WiPy boots up with default settings. This means you now have access to the filesystem, and you can edit `boot.py` and `main.py` to fix any problems.

Entering safe mode is temporary, and does not make any changes to the files on the WiPy.

Factory reset the filesystem

If you WiPys filesystem gets corrupted (very unlikely, but possible), you can format it very easily by doing:

```
>>> import os
>>> os.mkfs('/flash')
```

Resetting the filesystem deletes all files on the internal WiPy storage (not the SD card), and restores the files `boot.py` and `main.py` back to their original state after the next reset.

10.3 General board control (including sleep modes)

See the *machine* module:

```
import machine

help(machine) # display all members from the machine module
machine.freq() # get the CPU frequency
machine.unique_id() # return the 6-byte unique id of the board (the WiPy's MAC address)

machine.idle()        # average current decreases to (~12mA), any interrupts wake it up
machine.lightsleep()  # everything except for WLAN is powered down (~950uA avg. current)
                      # wakes from Pin, RTC or WLAN
machine.deepsleep()   # deepest sleep mode, MCU starts from reset. Wakes from Pin and␣
↪RTC.
```

10.4 Pins and GPIO

See *machine.Pin*.

```
from machine import Pin

# initialize GP2 in gpio mode (alt=0) and make it an output
p_out = Pin('GP2', mode=Pin.OUT)
p_out.value(1)
p_out.value(0)
p_out.toggle()
p_out(True)

# make GP1 an input with the pull-up enabled
p_in = Pin('GP1', mode=Pin.IN, pull=Pin.PULL_UP)
p_in() # get value, 0 or 1
```

10.5 Timers

See *machine.TimerWiPy* and *machine.Pin*. Timer ids take values from 0 to 3.:

```
from machine import Timer
from machine import Pin

tim = Timer(0, mode=Timer.PERIODIC)
tim_a = tim.channel(Timer.A, freq=1000)
tim_a.freq(5) # 5 Hz

p_out = Pin('GP2', mode=Pin.OUT)
tim_a.irq(trigger=Timer.TIMEOUT, handler=lambda t: p_out.toggle())
```

10.6 PWM (pulse width modulation)

See *machine.Pin* and *machine.Timer*.

```
from machine import Timer

# timer 1 in PWM mode and width must be 16 buts
tim = Timer(1, mode=Timer.PWM, width=16)

# enable channel A @1KHz with a 50.55% duty cycle
tim_a = tim.channel(Timer.A, freq=1000, duty_cycle=5055)
```

10.7 ADC (analog to digital conversion)

See *machine.ADCWiPy*.

```
from machine import ADC

adc = ADC()
apin = adc.channel(pin='GP3')
apin() # read value, 0-4095
```

10.8 UART (serial bus)

See *machine.UART*.

```
from machine import UART
uart = UART(0, baudrate=9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

10.9 SPI bus

See *machine.SPI*.

```
from machine import SPI

# configure the SPI controller @ 2MHz
spi = SPI(0, SPI.CONTROLLER, baudrate=2_000_000, polarity=0, phase=0)
spi.write('hello')
spi.read(5) # receive 5 bytes on the bus
rbuf = bytearray(5)
spi.write_readinto('hello', rbuf) # send and receive 5 bytes
```

10.10 I2C bus

See *machine.I2C*.

```
from machine import I2C
# configure the I2C bus
i2c = I2C(baudrate=100000)
i2c.scan() # returns list of peripheral addresses
i2c.writeto(0x42, 'hello') # send 5 bytes to peripheral with address 0x42
i2c.readfrom(0x42, 5) # receive 5 bytes from peripheral
i2c.readfrom_mem(0x42, 0x10, 2) # read 2 bytes from peripheral 0x42, peripheral memory␣
↪0x10
i2c.writeto_mem(0x42, 0x10, 'xy') # write 2 bytes to peripheral 0x42, peripheral memory␣
↪0x10
```

10.11 Watchdog timer (WDT)

See *machine.WDT*.

```
from machine import WDT

# enable the WDT with a timeout of 5s (1s is the minimum)
wdt = WDT(timeout=5000)
wdt.feed()
```

10.12 Real time clock (RTC)

See *machine.RTC*

```
from machine import RTC

rtc = RTC() # init with default time and date
rtc = RTC(datetime=(2015, 8, 29, 9, 0, 0, 0, None)) # init with a specific time and date
print(rtc.now())
```

(continues on next page)

(continued from previous page)

```
def alarm_handler (rtc_o):
    pass
    # do some non blocking operations
    # warning printing on an irq via telnet is not
    # possible, only via UART

# create a RTC alarm that expires after 5 seconds
rtc.alarm(time=5000, repeat=False)

# enable RTC interrupts
rtc_i = rtc.irq(trigger=RTC.ALARM0, handler=alarm_handler, wake=machine.SLEEP)

# go into suspended mode waiting for the RTC alarm to expire and wake us up
machine.lightsleep()
```

10.13 SD card

See *machine.SD*.

```
from machine import SD
import os

# clock pin, cmd pin, data0 pin
sd = SD(pins=('GP10', 'GP11', 'GP15'))
# or use default ones for the expansion board
sd = SD()
os.mount(sd, '/sd')
```

10.14 WLAN (WiFi)

See *network.WLAN* and *machine*.

```
import machine
from network import WLAN

# configure the WLAN subsystem in station mode (the default is AP)
wlan = WLAN(mode=WLAN.STA)
# go for fixed IP settings
wlan.ifconfig(config=('192.168.0.107', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
wlan.scan()     # scan for available networks
wlan.connect(ssid='mynetwork', auth=(WLAN.WPA2, 'mynetworkkey'))
while not wlan.isconnected():
    pass
print(wlan.ifconfig())
# enable wake on WLAN
wlan.irq(trigger=WLAN.ANY_EVENT, wake=machine.SLEEP)
# go to sleep
```

(continues on next page)

(continued from previous page)

```
machine.lightsleep()
# now, connect to the FTP or the Telnet server and the WiPy will wake-up
```

10.15 Telnet and FTP server

See *network.Server*

```
from network import Server

# init with new user, password and seconds timeout
server = Server(login=('user', 'password'), timeout=60)
server.timeout(300) # change the timeout
server.timeout() # get the timeout
server.isrunning() # check whether the server is running or not
```

10.16 Heart beat LED

See *wipy*.

```
import wipy

wipy.heartbeat(False)  # disable the heartbeat LED
wipy.heartbeat(True)   # enable the heartbeat LED
wipy.heartbeat()       # get the heartbeat state
```

CHAPTER

ELEVEN

QUICK REFERENCE FOR THE UNIX AND WINDOWS PORTS

11.1 Command line options

Usage:

```
micropython [ -h ] [ -i ] [ -O<level> ] [ -v ] [ -X <option> ] [ -c <command> | -m
↪<module> | <script> ] [ <args> ]
```

Invocation options:

-c <command>
: Runs the code in <command>. The code can be one or more Python statements.

-m <module>
: Runs the module <module>. The module must be in `sys.path`.

<script>
: Runs the file <script>. The script must be a valid MicroPython source code file.

If none of the 3 options above are given, then MicroPython is run in an interactive REPL mode.

<args>
: Any additional arguments after the module or script will be passed to `sys.argv` (not supported with the *-c* option).

General options:

-h
: Prints a help message containing the command line usage and exits.

-i
: Enables inspection. When this flag is set, MicroPython will enter the interactive REPL mode after the command, module or script has finished. This can be useful for debugging the state after an unhandled exception. Also see the *MICROPYINSPECT* environment variable.

-O | -O<level> | -OO...
: Sets the optimization level. The `O` can be followed by a number or can be repeated multiple times to indicate the level. E.g. `-O3` is the same as `-OOO`.

-v
: Increases the verbosity level. This option can be given multiple times. This option only has an effect if `MICROPY_DEBUG_PRINTERS` was enabled when MicroPython itself was compiled.

-X <option>
: Specifies additional implementation-specific options. Possible options are:

  - `-X compile-only` compiles the command, module or script but does not run it.

- `-X emit={bytecode,native,viper}` sets the default code emitter. Native emitters may not be available depending on the settings when MicroPython itself was compiled.
- `-X heapsize=<n>[w][K|M]` sets the heap size for the garbage collector. The suffix `w` means words instead of bytes. `K` means x1024 and `M` means x1024x1024.

11.2 Environment variables

MICROPYPATH
Overrides the default search path for MicroPython libraries. `MICROPYPATH` should be set to a colon (semicolon for Windows port) separated list of directories. If `MICROPYPATH` is not defined, the search path will be `~/.micropython/lib:/usr/lib/micropython` (`~/.micropython/lib` for Windows port) or the value of the `MICROPY_PY_SYS_PATH_DEFAULT` option if it was set when MicroPython itself was compiled.

MICROPYINSPECT
Enables inspection. If `MICROPYINSPECT` is set to a non-empty string, it has the same effect as setting the `-i` command line option.

CHAPTER

TWELVE

QUICK REFERENCE FOR THE ZEPHYR PORT

Below is a quick reference for the Zephyr port. If it is your first time working with this port please consider reading the following sections first:

12.1 General information about the Zephyr port

The Zephyr Project is a Linux Foundation hosted Collaboration Project. Its an open source collaborative effort uniting developers and users in building a small, scalable, real-time operating system (RTOS) optimized for resource-constrained devices, across multiple architectures.

12.1.1 Multitude of boards

There is a multitude of modules and boards from different sources that are supported by the Zephyr OS. All boards supported by Zephyr (with standard level of features support, like UART console) should work with MicroPython (but not all were tested). The FRDM-K64f board is taken as a reference board for the port for this documentation. If you have another board, please make sure you have a datasheet, schematics and other reference materials for your board handy to look up various aspects of your board functioning.

For a full list of Zephyr supported boards click here (external link)

12.2 MicroPython tutorial for the Zephyr port

This tutorial is intended to get you started with the Zephyr port.

12.2.1 Getting started with MicroPython on the Zephyr port

Lets get started!

Requirements

To use the MicroPython Zephyr port, you will need a Zephyr supported board (for a list of acceptable boards see *General information about the Zephyr port*).

Powering up

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 and deploying the firmware

The first step you will need to do is either clone the MicroPython repository or download it from the MicroPython downloads page. If you are an end user of MicroPython, it is recommended to start with the stable firmware builds. If you would like to work on development, you may follow the daily builds on git.

Next, follow the Zephyr port readme document (`ports/zephyr/README.md`) to build and run the application on your board.

12.2.2 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 your board through Zephyr. It is recommended to use REPL to test out your code and run commands.

REPL over the serial port

The REPL is available on a UART serial peripheral specified for the board by the `zephyr,console` devicetree node. 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.

To access the prompt over USB-serial you will need to use a terminal emulator program. For a Linux or Mac machine, open a terminal and run:

```
screen /dev/ttyACM0 115200
```

You can also try `picocom` or `minicom` instead of screen. You may have to use `/dev/ttyACM1` or a higher number for `ttyACM`. Additional permissions may be necessary to access this device (eg group `uucp` or `dialout`, or use sudo). For Windows, get a terminal software, such as puTTY and connect via a serial session using the proper COM port.

Using the REPL

With your serial program open (PuTTY, screen, picocom, etc) you may see a blank screen with a flashing cursor. Press Enter (or reset the board) and you should be presented with the following text:

```
*** Booting Zephyr OS build zephyr-v2.7.0  ***
MicroPython v1.17-288-gb695f5a70-dirty on 2022-01-03; zephyr-frdm_k64f with mk64f12
Type "help()" for more information.
>>>
```

Now you can try running MicroPython code directly on your board.

Anything you type at the prompt, indicated by >>>, will be executed after you press the Enter key. If there is an error with the text that you enter then an error message is printed.

Start by typing the following at the prompt to make sure it is working:

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

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

```
>>> 1 + 2
3
>>> 1 / 2
0.5
>>> 3 * 'Zephyr'
ZephyrZephyrZephyr
```

If your board has an LED, you can blink it using the following code:

```
>>>import time
>>>from machine import Pin

>>>LED = Pin(("GPIO_1", 21), Pin.OUT)
>>>while True:
...     LED.value(1)
...     time.sleep(0.5)
...     LED.value(0)
...     time.sleep(0.5)
```

The above code uses an LED location for a FRDM-K64F board (port B, pin 21; following Zephyr conventions ports are identified by GPIO_x, where *x* starts from 0). You will need to adjust it for another board using the boards reference materials.

12.2.3 Filesystems and Storage

Storage modules support virtual filesystem with FAT and littlefs formats, backed by either Zephyr DiskAccess or FlashArea (flash map) APIs depending on which the board supports.

See os Filesystem Mounting.

Disk Access

The *zephyr.DiskAccess* class can be used to access storage devices, such as SD cards. This class uses Zephyr Disk Access API and implements the *os.AbstractBlockDev* protocol.

For use with SD card controllers, SD cards must be present at boot & not removed; they will be auto detected and initialized by filesystem at boot. Use the disk driver interface and a file system to access SD cards via disk access (see below).

Example usage of FatFS with an SD card on the mimxrt1050_evk board:

```
import os
from zephyr import DiskAccess
bdev = zephyr.DiskAccess('SDHC')        # create block device object using DiskAccess
os.VfsFat.mkfs(bdev)                    # create FAT filesystem object using the disk␣
↪storage block
os.mount(bdev, '/sd')                   # mount the filesystem at the SD card␣
↪subdirectory
with open('/sd/hello.txt','w') as f:    # open a new file in the directory
    f.write('Hello world')              # write to the file
print(open('/sd/hello.txt').read())     # print contents of the file
```

Flash Area

The *zephyr.FlashArea* class can be used to implement a low-level storage system or customize filesystem configurations. To store persistent data on the device, using a higher-level filesystem API is recommended (see below).

This class uses Zephyr Flash map API and implements the *os.AbstractBlockDev* protocol.

Example usage with the internal flash on the reel_board or the rv32m1_vega_ri5cy board:

```
import os
from zephyr import FlashArea
bdev = FlashArea(FlashArea.STORAGE, 4096)   # create block device object using FlashArea
os.VfsLfs2.mkfs(bdev)                       # create Little filesystem object using the␣
↪flash area block
os.mount(bdev, '/flash')                    # mount the filesystem at the flash storage␣
↪subdirectory
with open('/flash/hello.txt','w') as f:     # open a new file in the directory
    f.write('Hello world')                  # write to the file
print(open('/flash/hello.txt').read())      # print contents of the file
```

For boards such as the frdm_k64f in which the MicroPython application spills into the default flash storage partition, use the scratch partition by replacing `FlashArea.STORAGE` with the integer value 4.

12.2.4 GPIO Pins

Use *machine.Pin* to control I/O pins.

For Zephyr, pins are initialized using a tuple of port and pin number `(\"GPIO_x\", pin#)` for the `id` value. For example to initialize a pin for the red LED on a FRDM-k64 board:

```
LED = Pin(("GPIO_1", 22), Pin.OUT)
```

Reference your boards datasheet or Zephyr documentation for pin numbers, see below for more examples.

Table 1: Pin Formatting

Board	Pin	Format
frdm_k64f	Red LED = PTB22	(GPIO_1, 22)
96b_carbon	LED1 = PD2	(GPIOD, 2)
mimxrt685_evk_cm33	Green LED = PIO0_14	(GPIO0, 14)

Interrupts

The Zephyr port also supports interrupt handling for Pins using *machine.Pin.irq()*. To respond to Pin change IRQs run:

```
from machine import Pin

SW2 = Pin(("GPIO_2", 6), Pin.IN)            # create Pin object for switch 2
SW3 = Pin(("GPIO_0", 4), Pin.IN)            # create Pin object for switch 3

SW2.irq(lambda t: print("SW2 changed"))     # print message when SW2 state is changed␣
↪(triggers change IRQ)
SW3.irq(lambda t: print("SW3 changed"))     # print message when SW3 state is changed␣
↪(triggers change IRQ)

while True:                                 # wait
    pass
```

12.3 Running MicroPython

See the corresponding section of the tutorial: *Getting started with MicroPython on the ESP8266*.

12.4 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
```

12.5 Pins and GPIO

Use the *machine.Pin* class:

```
from machine import Pin

pin = Pin(("GPIO_1", 21), Pin.IN)   # create input pin on GPIO1
print(pin)                          # print pin port and number

pin.init(Pin.OUT, Pin.PULL_UP, value=1)     # reinitialize pin

pin.value(1)                        # set pin to high
pin.value(0)                        # set pin to low
```

(continues on next page)

(continued from previous page)

```
pin.on()                            # set pin to high
pin.off()                           # set pin to low

pin = Pin(("GPIO_1", 21), Pin.IN)   # create input pin on GPIO1

pin = Pin(("GPIO_1", 21), Pin.OUT, value=1)         # set pin high on creation

pin = Pin(("GPIO_1", 21), Pin.IN, Pin.PULL_UP)      # enable internal pull-up resistor

switch = Pin(("GPIO_2", 6), Pin.IN)                 # create input pin for a switch
switch.irq(lambda t: print("SW2 changed"))          # enable an interrupt when switch␣
↪state is changed
```

12.6 Hardware I2C bus

Hardware I2C is accessed via the *machine.I2C* class:

```
from machine import I2C

i2c = I2C("I2C_0")          # construct an i2c bus
print(i2c)                  # print device name

i2c.scan()                  # scan the device for available I2C slaves

i2c.readfrom(0x1D, 4)                # read 4 bytes from slave 0x1D
i2c.readfrom_mem(0x1D, 0x0D, 1)      # read 1 byte from slave 0x1D at slave memory 0x0D

i2c.writeto(0x1D, b'abcd')           # write to slave with address 0x1D
i2c.writeto_mem(0x1D, 0x0D, b'ab')   # write to slave 0x1D at slave memory 0x0D

buf = bytearray(8)                  # create buffer of size 8
i2c.writeto(0x1D, b'abcd')          # write buf to slave 0x1D
```

12.7 Hardware SPI bus

Hardware SPI is accessed via the *machine.SPI* class:

```
from machine import SPI

spi = SPI("SPI_0")          # construct a spi bus with default configuration
spi.init(baudrate=100000, polarity=0, phase=0, bits=8, firstbit=SPI.MSB) # set␣
↪configuration

# equivalently, construct spi bus and set configuration at the same time
spi = SPI("SPI_0", baudrate=100000, polarity=0, phase=0, bits=8, firstbit=SPI.MSB)
print(spi)                  # print device name and bus configuration

spi.read(4)                 # read 4 bytes on MISO
```

(continues on next page)

(continued from previous page)

```
spi.read(4, write=0xF)      # read 4 bytes while writing 0xF on MOSI

buf = bytearray(8)          # create a buffer of size 8
spi.readinto(buf)           # read into the buffer (reads number of bytes equal to the␣
↪buffer size)
spi.readinto(buf, 0xF)      # read into the buffer while writing 0xF on MOSI

spi.write(b'abcd')          # write 4 bytes on MOSI

buf = bytearray(4)                  # create buffer of size 8
spi.write_readinto(b'abcd', buf)    # write to MOSI and read from MISO into the buffer
spi.write_readinto(buf, buf)        # write buf to MOSI and read back into the buf
```

12.8 Disk Access

Use the *zephyr.DiskAccess* class to support filesystem:

```
import os
from zephyr import DiskAccess

block_dev = DiskAccess('SDHC')      # create a block device object for an SD card
os.VfsFat.mkfs(block_dev)           # create FAT filesystem object using the disk␣
↪storage block
os.mount(block_dev, '/sd')          # mount the filesystem at the SD card subdirectory

# with the filesystem mounted, files can be manipulated as normal
with open('/sd/hello.txt','w') as f:     # open a new file in the directory
    f.write('Hello world')                  # write to the file
print(open('/sd/hello.txt').read())      # print contents of the file
```

12.9 Flash Area

Use the *zephyr.FlashArea* class to support filesystem:

```
import os
from zephyr import FlashArea

block_dev = FlashArea(4, 4096)      # creates a block device object in the frdm-k64f␣
↪flash scratch partition
os.VfsLfs2.mkfs(block_dev)          # create filesystem in lfs2 format using the flash␣
↪block device
os.mount(block_dev, '/flash')       # mount the filesystem at the flash subdirectory

# with the filesystem mounted, files can be manipulated as normal
with open('/flash/hello.txt','w') as f:     # open a new file in the directory
    f.write('Hello world')                  # write to the file
print(open('/flash/hello.txt').read())      # print contents of the file
```

12.10 Sensor

Use the *zsensor.Sensor* class to access sensor data:

```
import zsensor
from zsensor import Sensor

accel = Sensor("FXOX8700")    # create sensor object for the accelerometer

accel.measure()               # obtain a measurement reading from the accelerometer

# each of these prints the value taken by measure()
accel.float(zsensor.ACCEL_X)  # print measurement value for accelerometer X-axis sensor␣
↪channel as float
accel.millis(zsensor.ACCEL_Y) # print measurement value for accelerometer Y-axis sensor␣
↪channel in millionths
accel.micro(zsensor.ACCEL_Z)  # print measurement value for accelerometer Z-axis sensor␣
↪channel in thousandths
accel.int(zsensor.ACCEL_X)    # print measurement integer value only for accelerometer X-
↪axis sensor channel
```

PYTHON MODULE INDEX