MicroPython Documentation

Release 1.11

Damien P. George, Paul Sokolovsky, and contributors

May 29, 2019

CONTENTS

CHAPTER

ONE

MICROPYTHON LIBRARIES

Warning: Important summary of this section

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

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

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

Note about the availability of the modules and their contents: This documentation in general aspires to describe all modules and functions/classes which are implemented in MicroPython project. However, MicroPython is highly configurable, and each port to a particular board/embedded system makes available only a subset of MicroPython libraries. For officially supported ports, there is an effort to either filter out non-applicable items, or mark individual descriptions with "Availability:" clauses describing which ports provide a given feature.

With that in mind, please still be warned that some functions/classes in a module (or even the entire module) described in this documentation **may be unavailable** in a particular build of MicroPython on a particular 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. Some modules below use a standard Python name, but prefixed with "u", e.g. `ujson` instead of `json`. This is to signify that such a module is micro-library, i.e. implements only a subset of CPython module functionality. By naming them differently, a user has a choice to write a Python-level module to extend functionality for better compatibility with CPython (indeed, this is what done by the *`micropython-lib`* project mentioned above).

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

1.1.1 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
See CPython documentation: `OSError`. MicroPython doesn't implement `errno` attribute, instead use the standard way to access exception arguments: `exc.args[0]`.

exception RuntimeError

exception StopIteration

exception SyntaxError

exception SystemExit
See CPython documentation: `SystemExit`.

exception TypeError
See CPython documentation: `TypeError`.

exception ValueError

exception ZeroDivisionError

1.1.2 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.

1.1.3 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 circle's circumference to its diameter

1.1.4 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.

1.1.5 `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 circle's circumference to its diameter

1.1.6 `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.`**`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`.

Constants

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

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

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

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

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

Difference to CPython

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

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

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

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

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

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

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

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

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

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

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

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

1.1.7 ubinascii – 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

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

Difference to CPython

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

---

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

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

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

1.1.8 ucollections – 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

ucollections.**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.

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

```
from ucollections import namedtuple

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

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

```
from ucollections import OrderedDict

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

Output:

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

1.1.9 uerrno – system error codes

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

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

Constants

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

```
try:
    uos.mkdir("my_dir")
except OSError as exc:
```

```
    if exc.args[0] == uerrno.EEXIST:
        print("Directory already exists")
```

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

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

1.1.10 uhashlib – 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 interoperatiability will try to provide this.
- MD5 - A legacy algorithm, not considered cryptographically secure. Only selected boards, targeting interoperatibility with legacy applications, will offer this.

Constructors

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

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

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

Methods

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

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

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

1.1.11 uheapq – 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 heap queue algorithm.

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

Functions

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

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

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

1.1.12 uio – input/output streams

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

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

Conceptual hierarchy

Difference to CPython

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

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

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

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

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

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

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

Functions

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

Classes

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

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

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

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

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

class `uio.`**`StringIO`**(*alloc_size*)

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

Difference to CPython

These constructors are a MicroPython extension.

1.1.13 ujson – 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

ujson.**dump**(*obj*, *stream*)
Serialise *obj* to a JSON string, writing it to the given *stream*.

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

ujson.**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.

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

1.1.14 uos – 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 uos module contains functions for filesystem access and mounting, terminal redirection and duplication, and the uname and urandom functions.

General functions

uos.**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)

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

Filesystem access

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

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

uos.**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 don't have such a notion.
- Some platforms may return a 4-tuple that includes the entry's *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.

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

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

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

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

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

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

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

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

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

- `f_namemax` – maximum filename length

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

`uos.`**`sync`**`()`
Sync all filesystems.

Terminal redirection and duplication

`uos.`**`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 `uio.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.

`uos.`**`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 it's 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.

`uos.`**`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 `uos.`**`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*.

Block devices

A block device is an object which implements the block protocol, which is a set of methods described below by the *AbstractBlockDev* class. A concrete implementation of this class will usually allow access to the memory-like functionality a piece of hardware (like flash memory). A block device can be used by a particular filesystem driver to store the data for its filesystem.

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

readblocks(*block_num*, *buf*)
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.

writeblocks(*block_num*, *buf*)
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.

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)

By way of example, the following class will implement a 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 uos

bdev = RAMBlockDev(512, 50)
```

```
uos.VfsFat.mkfs(bdev)
vfs = uos.VfsFat(bdev)
uos.mount(vfs, '/ramdisk')
```

1.1.15 `ure` – simple regular expressions

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

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

Supported operators and special sequences are:

`.` Match any character.

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

`^` Match the start of the string.

`$` Match the end of the string.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

NOT SUPPORTED:

- counted repetitions (`{m,n}`)
- named groups (`(?P<name>...)`)
- non-capturing groups (`(?:...)`)

- more advanced assertions (\b, \B)
- special character escapes like \r, \n - use Python's own escaping instead
- etc.

Example:

```
import ure

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

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

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

Functions

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

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

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

ure.**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*.

ure.**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 *ure.compile()*.

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

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

Match objects

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

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

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

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

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

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

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

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

1.1.16 `uselect` – 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

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

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

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

class `Poll`

Methods

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

- `uselect.POLLIN` - data available for reading

- `uselect.POLLOUT` - more data can be written

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

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

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

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

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

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

Returns list of (`obj`, `event`, ...) tuples. There may be other elements in tuple, depending on a platform and version, so don't assume that its size is 2. The `event` element specifies which events happened with a stream and is a combination of `uselect.POLL*` constants described above. Note that flags `uselect.POLLHUP` and `uselect.POLLERR` can be returned at any time (even if were not asked for), and must be acted on accordingly (the corresponding stream unregistered from poll and likely closed), because otherwise all further invocations of *`poll()`* may return immediately with these flags set for this stream again.

In case of timeout, an empty list is returned.

Difference to CPython

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

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

If *flags* is 1, one-shot behavior for events is employed: streams for which events happened will have their event masks automatically reset (equivalent to `poll.modify(obj,0)`), so new events for such a stream won't be processed until new mask is set with *`poll.modify()`*. This behavior is useful for asynchronous I/O schedulers.

Difference to CPython

This function is a MicroPython extension.

1.1.17 `usocket` – 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 `usocket` module is an opaque data type returned by *getaddrinfo* function, which must be used to resolve textual address (including numeric addresses):

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

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

However, `socket` module (note the difference with native MicroPython `usocket` module described here) provides CPython-compatible way to specify addresses using tuples, as described below. 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 *usocket.getaddrinfo()*.
- IPv6: *(ipv6_address, port, flowinfo, scopeid)*, where *ipv6_address* is a string with colon-notation numeric IPv6 address, e.g. `"2001:db8::1"`, and *port* is an integer port number in the range 1-65535. *flowinfo* must be 0. *scopeid* is the interface scope identifier for link-local addresses. Note the domain names are not accepted as *ipv6_address*, they should be resolved first using *usocket.getaddrinfo()*. Availability of IPv6 support depends on a *MicroPython port*.

Functions

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

Create a new socket using the given address family, socket type and protocol number. 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)
```

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

Recommended use of filtering params:

```
s = usocket.socket()
# Guaranteed to return an address which can be connect'ed to for
# stream operation.
s.connect(usocket.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 doesn't have `socket.gaierror` and raises OSError directly. Note that error numbers of *getaddrinfo()* form a separate namespace and may not match error numbers from the *uerrno* module. To distinguish *getaddrinfo()* errors, they are represented by negative numbers, whereas standard system errors are positive numbers (error numbers are accessible using `e.args[0]` property from an exception object). The use of negative values is a provisional detail which may change in the future.

---

usocket.**inet_ntop**(*af*, *bin_addr*)

Convert a binary network address *bin_addr* of the given address family *af* to a textual representation:

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

usocket.**inet_pton**(*af*, *txt_addr*)

Convert a textual network address *txt_addr* of the given address family *af* to a binary representation:

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

Constants

usocket.**AF_INET**
usocket.**AF_INET6**

Address family types. Availability depends on a particular *MicroPython port*.

usocket.**SOCK_STREAM**
usocket.**SOCK_DGRAM**

Socket types.

usocket.**IPPROTO_UDP**
usocket.**IPPROTO_TCP**
IP protocol numbers. Availability depends on a particular *MicroPython port*. Note that you don't need to specify these in a call to *usocket.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()*.

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

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

Constants specific to WiPy:

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

class socket

Methods

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

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

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

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

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

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

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

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

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

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

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

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

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

socket.**settimeout**(*value*)
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 *uselect.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 = uselect.poll()
poller.register(s, uselect.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'). CPython's arguments: *encoding*, *errors* and *newline* are not supported.

Difference to CPython

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

Difference to CPython

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

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

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

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

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

Return value: the line read.

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

Return value: number of bytes written.

exception `usocket.`**`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 `ussl` – 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

`ussl.`**`wrap_socket`**`(`*sock*, *server_side=False*, *keyfile=None*, *certfile=None*, *cert_reqs=CERT_NONE*, *ca_certs=None*`)`
Takes a *`stream`* *sock* (usually usocket.socket instance of `SOCK_STREAM` type), and returns an instance of

ssl.SSLSocket, which wraps the underlying stream in an SSL context. Returned object has the usual *stream* interface methods like `read()`, `write()`, etc. In MicroPython, the returned object does not expose socket interface and methods like `recv()`, `send()`. In particular, a server-side SSL socket should be created from a normal socket returned from *accept()* on a non-SSL listening server socket.

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

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

Exceptions

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

Constants

`ussl.`**`CERT_NONE`**
`ussl.`**`CERT_OPTIONAL`**
`ussl.`**`CERT_REQUIRED`**
Supported values for *cert_reqs* parameter.

1.1.19 `ustruct` – 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).

Functions

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

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

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

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

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

1.1.20 `utime` – 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 `utime` 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

`utime.`**`localtime`**`(`[*secs*]`)`

Convert a time 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.

- 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

`utime.`**`mktime`**`()`

This is inverse function of localtime. It's 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.

`utime.`**`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.

`utime.`**`sleep_ms`**`(`*ms*`)`

Delay for given number of milliseconds, should be positive or 0.

`utime.`**`sleep_us`**`(`*us*`)`

Delay for given number of microseconds, should be positive or 0.

utime.**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.

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

utime.**ticks_cpu**()
Similar to *ticks_ms()* and *ticks_us()*, but with the highest possible resolution in the system. This is usually CPU clocks, and that's why the function is named that way. But it doesn't 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 utime 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.

utime.**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))
```

utime.**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*] (that's a typical range definition for two's-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 don't 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 behavior: don't 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()
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 .

utime.**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, use *ticks_ms()* and *ticks_us()* functions, if you need calendar time, *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 doesn't 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).

1.1.21 `uzlib` – 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

`uzlib.`**`decompress`**(*data*, *wbits=0*, *bufsize=0*)

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

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

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

Difference to CPython

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

1.1.22 `_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 `btree` – simple BTree database

The `btree` module implements a simple key-value database using external storage (disk files, or in general case, a random-access *`stream`*). Keys are stored sorted in the database, and besides efficient retrieval by a key value, a database also supports efficient ordered range scans (retrieval of values with the keys in a given range). On the

application interface side, BTree database work as close a possible to a way standard *dict* type works, one notable difference is that both keys and values must be *bytes* objects (so, if you want to store objects of other types, you need to serialize them to *bytes* first).

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

Example:

```
import btree

# First, we need to open a stream which holds a database
# This is usually a file, but can be in-memory database
# using uio.BytesIO, a raw flash 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 port's memory usage and/or performance.
- *cachesize* - Suggested memory cache size in bytes. For a board with enough memory using larger values may improve performance. Cache policy is as follows: entire cache is not allocated at once; instead, accessing a new page in database will allocate a memory buffer for it, until value specified by *cachesize* is reached. Then, these buffers will be managed using LRU (least recently used) policy. More buffers may still be allocated if needed (e.g., if a database contains big keys and/or values). Allocated cache buffers aren't reclaimed.
- *minkeypage* - Minimum number of keys to store per page. Default value of 0 equivalent to 2.

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

Methods

btree.**close**()

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

btree.**flush**()

Flush any data in cache to the underlying stream.

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

Standard dictionary methods.

btree.**__iter__**()

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

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

These methods are similar to standard dictionary methods, but also can take optional parameters to iterate over a key sub-range, instead of the entire database. Note that for all 3 methods, *start_key* and *end_key* arguments represent key values. For example, `values()` method will iterate over values corresponding to they key range given. None values for *start_key* means "from the first key", no *end_key* or its value of None means "until the end of database". By default, range is inclusive of *start_key* and exclusive of *end_key*, you can include *end_key*

in iteration by passing *flags* of `btree.INCL`. You can iterate in descending key direction by passing *flags* of `btree.DESC`. The flags values can be ORed together.

Constants

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

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

1.2.2 `framebuf` — Frame buffer manipulation

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

class FrameBuffer

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

For example:

```
import framebuf

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

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

Constructors

class `framebuf.`**`FrameBuffer`**(*buffer*, *width*, *height*, *format*, *stride=width*)
Construct a FrameBuffer object. The parameters are:

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

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

Drawing primitive shapes

The following methods draw shapes onto the FrameBuffer.

`FrameBuffer.`**`fill`**(*c*)
Fill the entire FrameBuffer with the specified color.

`FrameBuffer.`**`pixel`**(*x*, *y*[, *c*])
If *c* is not given, get the color value of the specified pixel. If *c* is given, set the specified pixel to the given color.

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

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

`FrameBuffer.`**`line`**(*x1*, *y1*, *x2*, *y2*, *c*)
Draw a line from a set of coordinates using the given color and a thickness of 1 pixel. The `line` method draws the line up to a second set of coordinates whereas the `hline` and `vline` methods draw horizontal and vertical lines respectively up to a given length.

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

`FrameBuffer.`**`fill_rect`**(*x*, *y*, *w*, *h*, *c*)
Draw a rectangle at the given location, size and color. The `rect` method draws only a 1 pixel outline whereas the `fill_rect` method draws both the outline and interior.

Drawing text

`FrameBuffer.`**`text`**(*s*, *x*, *y*[, *c*])
Write text to the FrameBuffer using the the coordinates as the upper-left corner of the text. The color of the text can be defined by the optional argument but is otherwise a default value of 1. All characters have dimensions of 8x8 pixels and there is currently no way to change the font.

Other methods

`FrameBuffer.`**`scroll`**(*xstep*, *ystep*)
Shift the contents of the FrameBuffer by the given vector. This may leave a footprint of the previous colors in the FrameBuffer.

`FrameBuffer.`**`blit`**(*fbuf*, *x*, *y*[, *key*])
Draw another FrameBuffer on top of the current one at the given coordinates. If *key* is specified then it should be a color integer and the corresponding color will be considered transparent: all pixels with that color value will not be drawn.

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

Constants

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

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

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

framebuf.**RGB565**
Red Green Blue (16-bit, 5+6+5) color format

framebuf.**GS2_HMSB**
Grayscale (2-bit) color format

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

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

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

Interrupt related functions

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

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

Power related functions

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

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

machine.**sleep**()

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 timout, 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.**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
p0.mode(p0.IN)

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

Constructors

class machine.**Pin**(*id*, *mode=-1*, *pull=-1*, ***, *value*, *drive*, *alt*)

Access the pin peripheral (GPIO pin) associated with the given id. If additional arguments are given in the constructor then they are used to initialise the pin. Any settings that are not specified will remain in their previous state.

The arguments are:

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

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

Methods

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

Re-initialise the pin using the given parameters. Only those arguments that are specified will be set. The rest of

the pin peripheral state will remain unchanged. See the constructor documentation for details of the arguments.

Returns `None`.

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

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

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

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

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

When setting the value this method returns `None`.

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

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

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

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

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

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

Not all ports implement this method.

Availability: WiPy.

`Pin.`**`irq`**`(`*handler=None*, *trigger=(Pin.IRQ_FALLING | Pin.IRQ_RISING)*, *, *priority=1*, *wake=None*, *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 OR'ed together to trigger on multiple events.
- `priority` sets the priority level of the interrupt. The values it can take are port-specific, but higher values always represent higher priorities.
- `wake` selects the power mode in which this interrupt can wake up the system. It can be `machine.IDLE`, `machine.SLEEP` or `machine.DEEPSLEEP`. These values can also be OR'ed together to make a pin generate interrupts in more than one power mode.
- `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*.

This method returns a callback object.

Constants

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

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

`Pin.`**`PULL_UP`**
`Pin.`**`PULL_DOWN`**
`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
# 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 usecases above and the architecture of MicroPython: Pin offers the lowest overhead, which may be important when bit-banging protocols. But Signal adds additional flexibility on top of Pin, at the cost of minor overhead (much smaller than if you implemented active-high vs active-low device differences in Python manually!). Also, Pin is 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 doesn't work for them simply because their LEDs or relays are wired in a slightly different way.

Constructors

class machine.**Signal**(*pin_obj*, *invert=False*)
class machine.**Signal**(*pin_arguments...*, *, *invert=False*)
Create a Signal object. There're two ways to create it:

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

The arguments are:

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

Methods

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

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

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

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

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

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

Activate signal.

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

Deactivate signal.

class ADC – analog to digital conversion

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.`**`ADC`**`(`*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

ADC.**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')
```

ADC.**init**()
Enable the ADC block.

ADC.**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 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.
- *txbuf* specifies the length in characters of the TX buffer.
- *rxbuf* specifies the length in characters of the RX buffer.

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.

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

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

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

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

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

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

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

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

`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 (master side)

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

Constructors

class machine.**SPI**(*id*, ...)

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

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

Methods

SPI.**init**(*baudrate=1000000*, ***, *polarity=0*, *phase=0*, *bits=8*, *firstbit=SPI.MSB*, *sck=None*, *mosi=None*, *miso=None*, *pins=(SCK*, *MOSI*, *MISO)*)

Initialise the SPI bus with the given parameters:

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

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

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

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

class I2C – a two-wire serial protocol

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

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

Printing the I2C object gives you information about its configuration.

Example usage:

```
from machine import I2C

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

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

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

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

Constructors

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

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

General Methods

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

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

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

Availability: WiPy.

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

Primitive I2C operations

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

These methods are available on software I2C only.

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 slave 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 master read and write operations that target a given slave device.

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

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

The method returns `None`.

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

I2C.**writevto**(*addr*, *vector*, *stop=True*)
Write the bytes contained in *vector* to the slave 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 don't 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 slave address and the memory address. The following methods are convenience functions to communicate with such devices.

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

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

The method returns `None`.

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

The method returns `None`.

class RTC – real time clock

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

Example usage:

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

Constructors

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

Methods

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

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

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

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

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

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

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

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

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

Constants

RTC.**ALARM0**
irq trigger source

class Timer – control hardware timers

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

See discussion of *important constraints* on Timer callbacks.

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

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).

Methods

Timer.**init**(*, *mode=Timer.PERIODIC*, *period=-1*, *callback=None*)
Initialise the timer. Example:

```
tim.init(period=100)                            # periodic with 100ms period
tim.init(mode=Timer.ONE_SHOT, period=1000)      # one shot firing after 1000ms
```

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.

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

Constants

Timer.**ONE_SHOT**
Timer.**PERIODIC**
Timer operating mode.

class WDT – watchdog timer

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

Example usage:

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

Availability of this class: pyboard, WiPy.

Constructors

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

Methods

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

class SD – secure digital memory card

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.

1.2.4 `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**()

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

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

If the REPL becomes active with the heap locked then it will be forcefully unlocked.

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

1.2.5 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 *usocket* module.

For example:

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

# now use usocket as usual
```

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

Common network adapter interface

This section describes an (implied) abstract base class for all network interface classes implemented by *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 (behavior 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 identifer types; b) provide additional connection parameters. For various medium types, there are different sets of predefined/recommended parameters, among them:

  - WiFi: *bssid* keyword to connect 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.

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

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

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

There are five values for authmode:

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

and two for hidden:

- 0 – visible
- 1 – hidden

WLAN.**status**([*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

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())

# 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 busses 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 busses 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.

1.2.6 `ucryptolib` – cryptographic ciphers

Classes

class `ucryptolib.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 `ucryptolib.MODE_ECB` if it exists) for Electronic Code Book (ECB).
  - `2` (or `ucryptolib.MODE_CBC` if it exists) for Cipher Block Chaining (CBC).
  - `6` (or `ucryptolib.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.

1.2.7 uctypes – access binary data in a structured way

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

Warning: `uctypes` module allows access to arbitrary memory addresses of the machine (including I/O and control registers). Uncareful usage of it may lead to crashes, data loss, and even hardware malfunction.

See also:

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

Usage examples:

```
import uctypes

# Example 1: Subset of ELF file header
# https://wikipedia.org/wiki/Executable_and_Linkable_Format#File_header
ELF_HEADER = {
    "EI_MAG": (0x0 | uctypes.ARRAY, 4 | uctypes.UINT8),
    "EI_DATA": 0x5 | uctypes.UINT8,
    "e_machine": 0x12 | uctypes.UINT16,
}

# "f" is an ELF file opened in binary mode
buf = f.read(uctypes.sizeof(ELF_HEADER, uctypes.LITTLE_ENDIAN))
header = uctypes.struct(uctypes.addressof(buf), ELF_HEADER, uctypes.LITTLE_ENDIAN)
```

```
assert header.EI_MAG == b"\x7fELF"
assert header.EI_DATA == 1, "Oops, wrong endianness. Could retry with uctypes.BIG_
↪ENDIAN."
print("machine:", hex(header.e_machine))


# Example 2: In-memory data structure, with pointers
COORD = {
    "x": 0 | uctypes.FLOAT32,
    "y": 4 | uctypes.FLOAT32,
}

STRUCT1 = {
    "data1": 0 | uctypes.UINT8,
    "data2": 4 | uctypes.UINT32,
    "ptr": (8 | uctypes.PTR, COORD),
}

# Suppose you have address of a structure of type STRUCT1 in "addr"
# uctypes.NATIVE is optional (used by default)
struct1 = uctypes.struct(addr, STRUCT1, uctypes.NATIVE)
print("x:", struct1.ptr[0].x)


# Example 3: Access to CPU registers. Subset of STM32F4xx WWDG block
WWDG_LAYOUT = {
    "WWDG_CR": (0, {
        # BFUINT32 here means size of the WWDG_CR register
        "WDGA": 7 << uctypes.BF_POS | 1 << uctypes.BF_LEN | uctypes.BFUINT32,
        "T": 0 << uctypes.BF_POS | 7 << uctypes.BF_LEN | uctypes.BFUINT32,
    }),
    "WWDG_CFR": (4, {
        "EWI": 9 << uctypes.BF_POS | 1 << uctypes.BF_LEN | uctypes.BFUINT32,
        "WDGTB": 7 << uctypes.BF_POS | 2 << uctypes.BF_LEN | uctypes.BFUINT32,
        "W": 0 << uctypes.BF_POS | 7 << uctypes.BF_LEN | uctypes.BFUINT32,
    }),
}

WWDG = uctypes.struct(0x40002c00, WWDG_LAYOUT)

WWDG.WWDG_CFR.WDGTB = 0b10
WWDG.WWDG_CR.WDGA = 1
print("Current counter:", WWDG.WWDG_CR.T)
```

Defining structure layout

Structure layout is defined by a "descriptor" - a Python dictionary which encodes field names as keys and other properties required to access them as associated values:

```
{
    "field1": <properties>,
    "field2": <properties>,
    ...
}
```

Currently, `uctypes` requires explicit specification of offsets for each field. Offset are given in bytes from the structure

start.

Following are encoding examples for various field types:

- Scalar types:

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

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

- Recursive structures:

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

  i.e. value is a 2-tuple, first element of which is an offset, and second is a structure descriptor dictionary (note: offsets in recursive descriptors are relative to the structure it defines). Of course, recursive structures can be specified not just by a literal dictionary, but by referring to a structure descriptor dictionary (defined earlier) by name.

- Arrays of primitive types:

```
"arr": (offset | uctypes.ARRAY, size | uctypes.UINT8),
```

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

- Arrays of aggregate types:

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

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

- Pointer to a primitive type:

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

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

- Pointer to an aggregate type:

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

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

- Bitfields:

```
"bitf0": offset | uctypes.BFUINT16 | lsbit << uctypes.BF_POS | bitsize << uctypes.
↪BF_LEN,
```

  i.e. value is a type of scalar value containing given bitfield (typenames are similar to scalar types, but prefixes with `BF`), ORed with offset for scalar value containing the bitfield, and further ORed with values for bit position and bit length of the bitfield within the scalar value, shifted by BF_POS and BF_LEN bits, respectively. A bitfield position is counted from the least significant bit of the scalar (having position of 0), and is the number

of right-most bit of a field (in other words, it's a number of bits a scalar needs to be shifted right to extract the bitfield).

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

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

Module contents

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

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

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

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

`uctypes.`**`sizeof`**(*struct*, *layout_type=NATIVE*)
Return size of data structure in bytes. The *struct* argument can be either a structure class or a specific instantiated structure object (or its aggregate field).

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

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

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

`uctypes.`**`UINT8`**
`uctypes.`**`INT8`**
`uctypes.`**`UINT16`**
`uctypes.`**`INT16`**
`uctypes.`**`UINT32`**
`uctypes.`**`INT32`**
`uctypes.`**`UINT64`**
`uctypes.`**`INT64`**
Integer types for structure descriptors. Constants for 8, 16, 32, and 64 bit types are provided, both signed and unsigned.

`uctypes.`**`FLOAT32`**

uctypes.**FLOAT64**
Floating-point types for structure descriptors.

uctypes.**VOID**
VOID is an alias for UINT8, and is provided to conviniently define C's void pointers: (uctypes.PTR,uctypes.VOID).

uctypes.**PTR**
uctypes.**ARRAY**
Type constants for pointers and arrays. Note that there is no explicit constant for structures, it's implicit: an aggregate type without PTR or ARRAY flags is a structure.

Structure descriptors and instantiating structure objects

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

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

Structure objects

Structure objects allow accessing individual fields using standard dot notation: my_struct.substruct1.field1. If a field is of scalar type, getting it will produce a primitive value (Python integer or float) corresponding to the value contained in a field. A scalar field can also be assigned to.

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

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

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

Limitations

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

- Avoid accessing nested structures. For example, instead of mcu_registers.peripheral_a.register1, define separate layout descriptors for each peripheral, to be accessed as peripheral_a.register1. Or just cache a particular peripheral: peripheral_a = mcu_registers.peripheral_a. If a register consists of multiple bitfields, you would need to cache references to a particular register: reg_a = mcu_registers.peripheral_a.reg_a.
- Avoid other non-scalar data, like arrays. For example, instead of peripheral_a.register[0] use peripheral_a.register0. Again, an alternative is to cache intermediate values, e.g. register0 = peripheral_a.register[0].

2. Range of offsets supported by the `uctypes` module is limited. The exact range supported is considered an implementation detail, and the general suggestion is to split structure definitions to cover from a few kilobytes to a few dozen of kilobytes maximum. In most cases, this is a natural situation anyway, e.g. it doesn't make sense to define all registers of an MCU (spread over 32-bit address space) in one structure, but rather a peripheral block by peripheral block. In some extreme cases, you may need to split a structure in several parts artificially (e.g. if accessing native data structure with multi-megabyte array in the middle, though that would be a very synthetic case).

1.3 Libraries specific to the pyboard

The following libraries are specific to the pyboard.

1.3.1 `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 busses 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 *uos.mount()* and *uos.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 *uos.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*], *vid=0xf055*, *pid=0x9801*, *hid=pyb.hid_mouse*)
If called with no arguments, return the current USB mode as a string.

If called with `modestr` provided, attempts to set USB mode. This can only be done when called from `boot.py` before *pyb.main()* has been called. 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)

For backwards compatibility, `'CDC'` is understood to mean `'VCP'` (and similarly for `'CDC+MSC'` and `'CDC+HID'`).

The `vid` and `pid` parameters allow you to specify the VID (vendor id) and PID (product id).

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.

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 ADC's.

It reads analog values from multiple ADC's 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 ADC's.

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 ADC's:

```
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 ADC's being read. In testing two ADC's were sampled at a timer rate of 210kHz without overrun. Samples were missed at 215kHz. For three ADC's 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, don't forget that the MCU's 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
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 busses 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*)

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

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 controller's 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 entered the Error Warning state (wrapped around to 0 after 65535)
- number of times the controller entered the Error Passive state (wrapped around to 0 after 65535)
- number of times the controller entered 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 can't 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')

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
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 5KΩ. Otherwise it has an output impedance of 15KΩ maximum: consequently to achieve a 1% accuracy without buffering requires the applied load to be less than 1.5MΩ. 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 the frequency of the repeating triangle wave itself is 2048 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 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.MASTER)               # create and init as a master
i2c.init(I2C.MASTER, baudrate=20000) # init as a master
i2c.init(I2C.SLAVE, addr=0x42)         # init as a slave with given address
i2c.deinit()                           # turn off the peripheral
```

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 master must specify the recipient's address:

```
i2c.init(I2C.MASTER)
i2c.send('123', 0x42)        # send 3 bytes to slave with address 0x42
i2c.send(b'456', addr=0x42)  # keyword for address
```

Master also has other methods:

```
i2c.is_ready(0x42)             # check if slave 0x42 is ready
i2c.scan()                     # scan for slaves on the bus, returning
                               #   a list of valid addresses
i2c.mem_read(3, 0x42, 2)       # read 3 bytes from memory of slave 0x42,
                               #   starting at address 2 in the slave
i2c.mem_write('abc', 0x42, 2, timeout=1000) # write 'abc' (3 bytes) to memory of␣
↪slave 0x42
                                            # starting at address 2 in the slave,␣
↪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 busses 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.MASTER` or `I2C.SLAVE`
- `addr` is the 7-bit address (only sensible for a slave)
- `baudrate` is the SCL clock rate (only sensible for a master)
- `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 master 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 master 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 master 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 master 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 master 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 master mode.

Constants

`I2C.`**`MASTER`**
for initialising the bus to master mode

`I2C.`**`SLAVE`**
for initialising the bus to slave 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*, *af=-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.
- when mode is `Pin.AF_PP` or `Pin.AF_OD`, then af can be the index or name of one of the alternate functions associated with a pin.

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**
don't 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 and 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 master-driven serial protocol

SPI is a serial protocol that is driven by a master. 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.MASTER, baudrate=600000, polarity=1, phase=0, crc=0x7)
```

Only required parameter is mode, SPI.MASTER or SPI.SLAVE. 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 busses 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.MASTER` or `SPI.SLAVE`.
- `baudrate` is the SCK clock rate (only sensible for a master).
- `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`.
- `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.`**`MASTER`**

`SPI.`**`SLAVE`**
for initialising the SPI bus to master or slave 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 can't be allocated during a callback (an interrupt) and so exceptions raised within a callback don't 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*)

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 timer's 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 timer's 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 doesn't 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. `bus` can be 1-6, or 'XA', 'XB', '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 busses 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 as above except:

- `UART(2)` is on: `(TX,RX) = (X1,X2) = (PA2,PA3)`

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 UART's `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 Pyboard's `nCTS` signal should be connected to the target's `nRTS` and the Pyboard's `nRTS` to the target's `nCTS`.

CTS: target controls Pyboard transmitter

If CTS flow control is enabled the write behaviour is as follows:

If the Pyboard's `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 target's 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**
Create a new USB_VCP object.

Methods

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.

1.3.2 `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)
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 LCD's power/enabled pin.
- *i2c* is an I2C object connected to the LCD's I2C interface.
- *spi* is an SPI object connected to the LCD's 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 it's 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 don't 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 display's 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 OR'ed together, used by *LCD160CR.set_startup_deco()*.

1.4 Libraries specific to the WiPy

The following libraries and classes are specific to the WiPy.

1.4.1 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).

1.4.2 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. MicroPython's Timer class defines a baseline operation of executing a callback with a given period (or once after some delay), and allow specific boards to define more non-standard behavior (which thus won't be portable to other boards).

See discussion of *important constraints* on Timer callbacks.

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

Constructors

class machine.**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`. It's a percentage (0.00-100.00). Since the WiPy doesn't 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 there's 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.

1.4.3 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 behavior 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. It's a percentage (0.00-100.00). Since the WiPy doesn't 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.

1.5 Libraries specific to the ESP8266 and ESP32

The following libraries are specific to the ESP8266 and ESP32.

1.5.1 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=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.

1.5.2 `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.

The 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.

CHAPTER

TWO

THE MICROPYTHON LANGUAGE

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) OS, for example an *MCU*-based system. When running on a baremetal system, MicroPython effectively becomes its user-facing OS with a command interpreter (REPL).

board A PCB board. Oftentimes, the term is used to denote a particular model of an *MCU* system. Sometimes, it is used to actually refer to *MicroPython port* to a particular board (and then may also refer to "boardless" ports like *Unix port*).

callee-owned tuple A tuple returned by some builtin function/method, containing data which is valid for a limited time, usually until next call to the same function (or a group of related functions). After next call, data in the tuple may be changed. This leads to the following restriction on the usage of callee-owned tuples - references to them cannot be stored. The only valid operation is extracting values from them (including making a copy). Callee-owned tuples is a MicroPython-specific construct (not available in the general Python language), introduced for memory allocation optimization. The idea is that callee-owned tuple is allocated once and stored on the callee side. Subsequent calls don't require allocation, allowing to return multiple values when allocation is not possible (e.g. in interrupt context) or not desirable (because allocation inherently leads to memory fragmentation). Note that callee-owned tuples are effectively mutable tuples, making an exception to Python's rule that tuples are immutable. (It may be interesting why tuples were used for such a purpose then, instead of mutable lists - the reason for that is that lists are mutable from user application side too, so a user could do things to a callee-owned list which the callee doesn't expect and could lead to problems; a tuple is protected from this.)

CPython CPython is the reference implementation of Python programming language, and the most well-known one, which most of the people run. It is however one of many implementations (among which Jython, IronPython, PyPy, and many more, including MicroPython). As there is no formal specification of the Python language, only CPython documentation, it is not always easy to draw a line between Python the language and CPython its particular implementation. This however leaves more freedom for other implementations. For example, MicroPython does a lot of things differently than CPython, while still aspiring to be a Python language implementation.

GPIO General-purpose input/output. The simplest means to control electrical signals. With GPIO, user can configure hardware signal pin to be either input or output, and set or get its digital signal value (logical "0" or "1"). MicroPython abstracts GPIO access using `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).

interned string A string referenced by its (unique) identity rather than its address. Interned strings are thus can be quickly compared just by their identifiers, instead of comparing by content. The drawbacks of interned strings are that interning operation takes time (proportional to the number of existing interned strings, i.e. becoming slower and slower over time) and that the space used for interned strings is not reclaimable. String interning is done automatically by MicroPython compiler and runtimer when it's either required by the implementation (e.g. function keyword arguments are represented by interned string id's) or deemed beneficial (e.g. for short enough strings, which have a chance to be repeated, and thus interning them would save memory on copies). Most of string and I/O operations don't produce interned strings due to drawbacks described above.

MCU Microcontroller. Microcontrollers usually have much less resources than a full-fledged computing system, but 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*. Instead, there is a related, but separate project micropython-lib which provides implementations for many modules from CPython's standard library. However, large subset of these modules require POSIX-like environment (Linux, FreeBSD, MacOS, etc.; Windows may be partially supported), and thus would work or make sense only with *MicroPython Unix port*. Some subset of modules is however usable for *baremetal* ports too.

Unlike monolithic *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 MicroPython core functionality, would be described in the separate port-specific documentation.

MicroPython Unix port 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 importance of Unix port lies in the fact that while there are many different *boards*, so two random users unlikely have the same board, almost all modern OSes have some level of POSIX compatibility, so Unix port serves as a kind of "common ground" to which any user can have access. So, Unix port is used for initial prototyping, different kinds of testing, development of machine-independent features, etc. All users of MicroPython, even those which are interested only in running MicroPython on *MCU* systems, are recommended to be familiar with Unix (or Windows) port, as it is important productivity helper and a part of normal MicroPython workflow.

port Either *MicroPython port* or *GPIO port*. If not clear from context, it's recommended to use full specification like one of the above.

stream Also known as a "file-like object". An 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 *uio* module.

upip (Literally, "micro pip"). A package manage for MicroPython, inspired by *CPython*'s pip, but much smaller and with reduced functionality. upip runs both on *Unix port* and on *baremetal* ports (those 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 you've entered. The following shows what you'd 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:
...         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 won't 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 doesn't intercept the KeyboardInterrupt exception.

For example:

```
>>> for i in range(1000000):
...     print(i)
...
0
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 you're 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, you'd 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, you'll 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

Raw mode is not something that a person would normally use. It is intended for programmatic use. It essentially behaves like paste mode with echo turned off.

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.

The `tools/pyboard.py` program uses the raw REPL to execute python files on the MicroPython board.

2.3 Writing interrupt handlers

On suitable hardware MicroPython offers the ability to write interrupt handlers in Python. Interrupt handlers - also known as interrupt service routines (ISR's) - 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.

2.3.1 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).

2.3.2 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)
```

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 LED's 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 function's 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

ISR's 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 ISR's can't use floating point arithmetic; this is because floats are Python objects. Similarly an ISR can't 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 it's 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 ISR's is unsafe because memory allocation may be attempted as the variable's 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".

2.3.3 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.

2.3.4 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, ISR's 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 ISR's 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 ISR's or between multiple ISR's. 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
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 ISR's. 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 ISR's, 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`.

2.4 Maximising MicroPython Speed

Contents

- *Maximising MicroPython Speed*
  - *Designing for speed*
    - *Algorithms*
    - *RAM Allocation*
    - *Buffers*
    - *Floating Point*
    - *Arrays*
  - *Identifying the slowest section of code*
  - *MicroPython code improvements*

        * *The const() declaration*
        * *Caching object references*
        * *Controlling garbage collection*
    - *The Native code emitter*
    - *The Viper code emitter*
    - *Accessing hardware directly*

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.

2.4.1 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 Python's 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. *memoryview* itself is allocated on heap, but is a small, fixed-size object, regardless of the size of slice it points too.

```
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 isn't 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()`.

2.4.2 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 *utime*. 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 = utime.ticks_us()
        result = f(*args, **kwargs)
        delta = utime.ticks_diff(utime.ticks_us(), t)
        print('Function {} Time = {:6.3f}ms'.format(myname, delta/1000))
        return result
    return new_func
```

2.4.3 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):
        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.

2.4.4 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 twices as fast as bytecode) is an increase in compiled code size.

2.4.5 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 with read-write capabilities (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).

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

2.4.6 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* instance's *value()* method. This overhead can be eliminated by performing a read/write to the relevant bit of the chip's 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

BIT14 = const(1 << 14)
machine.mem16[stm.GPIOA + stm.GPIO_ODR] ^= BIT14
```

2.5 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 non-volatile "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.

2.5.1 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.

2.5.2 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 *`ustruct`* 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)
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 *ujson* 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.

2.5.3 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.

2.5.4 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.

2.5.5 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`, `ustruct` 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.

2.6 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 user's environment. This chapter discusses how these actions are achieved. Some familiarity with Python packaging is recommended.

2.6.1 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.

2.6.2 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 contguous 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 aren't used by the installation process. In particular, *upip* package manager doesn't 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 isn't 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 MicroPython's target environments, which are highly resource constrained devices.

2.6.3 `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 don't have to match, and oftentimes they don't. 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 MicroPython's 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.]

2.6.4 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 depenency 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).

2.6.5 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 port's 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 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.

2.6.6 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

2.6.7 Application resources

A complete application, besides the source code, oftentimes also consists of data files, e.g. web page templates, game images, etc. It's 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.

Let's 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
```

2.6.8 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

2.7 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.

2.7.1 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 doesn't support concepts such as indirection. The pseudocode employed in such cases is described on the relevant page.

2.7.2 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 = 1's 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 two's 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 two's 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 two's 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 doesn't 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 core's 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 it's 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.

2.7.3 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 it's 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)

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 MicroPython's 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)))
```

2.7.4 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.)

CHAPTER

THREE

MICROPYTHON DIFFERENCES FROM CPYTHON

The operations listed in this section produce conflicting results in MicroPython when compared to standard Python. MicroPython implements Python 3.4 and some select features of Python 3.5.

3.1 Syntax

Generated Wed 29 May 2019 07:20:22 UTC

3.1.1 Spaces

uPy requires spaces between literal numbers and keywords, CPy doesn't

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

3.1.2 Unicode

Unicode name escapes are not implemented

Sample code:

```
print("\N{LATIN SMALL LETTER A}")
```

CPy output:	uPy output:
a	NotImplementedError: unicode name escapes

3.2 Core Language

Generated Wed 29 May 2019 07:20:22 UTC

3.2.1 Classes

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__	

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 don't 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)`

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__`

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>`

3.2.2 Functions

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:
`append() takes exactly one argument (0 given)`	`function takes 2 positional arguments but 1 were g`

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 10, in <module>`
	`AttributeError: 'function' object has no attribute`

3.2.3 Generator

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

3.2.4 Runtime

Local variables aren't included in locals() result

Cause: MicroPython doesn't maintain symbolic local environment, it is optimized to an array of slots. Thus, local variables can't be accessed by a name.

Sample code:

```
def test():
    val = 2
    print(locals())

test()
```

CPy output:	uPy output:
{'val': 2}	{'test': <function test at 0x7f17b5262100>, '__nam

Code running in eval() function doesn't have access to local variables

Cause: MicroPython doesn't maintain symbolic local environment, it is optimized to an array of slots. Thus, local variables can't 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

3.2.5 import

__path__ attribute of a package has a different type (single string instead of list of strings) in MicroPython

Cause: MicroPython does't support namespace packages split across filesystem. Beyond that, MicroPython's import system is highly optimized for minimal memory usage.

Workaround: Details of import handling is inherently implementation dependent. Don't rely on such details in portable applications.

Sample code:

```
import modules

print(modules.__path__)
```

CPy output:	uPy output:
['/home/micropython/micropython-docs/tests/import/modules']	test_subdir/foo.py test_subdir/__init__.py test_subdir/bar.py

Failed to load modules are still registered as loaded

Cause: To make module handling more efficient, it's 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`

MicroPython doesn't support namespace packages split across filesystem.

Cause: MicroPython's import system is highly optimized for simplicity, minimal memory usage, and minimal filesystem search overhead.

Workaround: Don't install modules belonging to the same namespace package in different directories. For MicroPython, it's 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 12, in <module>`
	`ImportError: no module named 'subpkg.bar'`

3.3 Builtin Types

Generated Wed 29 May 2019 07:20:22 UTC

3.3.1 Exception

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 attribut`

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 15, in <module>`
	`  File "<stdin>", line 13, in __init__`
	`AttributeError: type object 'Exception' has no att`

3.3.2 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`

3.3.3 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, don't 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 paramter, 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 i`

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`

3.3.4 float

uPy and CPython outputs formats may differ

Sample code:

```
print('%.1g' % -9.9)
```

CPy output:	uPy output:
`-1e+01`	`-10`

3.3.5 int

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 11, in <module>`
	`  File "<stdin>", line 8, in <lambda>`
	`TypeError: unsupported types for __radd__: 'int',`

3.3.6 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:`

3.3.7 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 i

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 'lju

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)

Instance of a subclass of str cannot be compared for equality with an instance of a str

Sample code:

```
class S(str):
    pass

s = S('hello')
print(s == 'hello')
```

CPy output:	uPy output:
`True`	`False`

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`

3.3.8 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`

3.4 Modules

Generated Wed 29 May 2019 07:20:22 UTC

3.4.1 array

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 8, 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 9, in <module>`
	`TypeError: 'array' object doesn't support item del`

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 9, in <module>`
	`NotImplementedError: only slices with step=1 (aka`

3.4.2 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 b`

3.4.3 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 8, in <module>`
	`TypeError: function missing 2 required positional`

3.4.4 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`

3.4.5 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`

3.4.6 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 8, in <module>`
	`AttributeError: 'module' object has no attribute '`

CHAPTER

FOUR

DEVELOPING AND BUILDING MICROPYTHON

This chapter describes some options for extending MicroPython in C. Note that it doesn't aim to be a complete guide for developing with MicroPython. See the getting started guide for further information.

4.1 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 can't be resolved with suggestions in *Maximising MicroPython Speed*, writing some or all of your module in C 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 you're 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.

4.1.1 Structure of an external C module

A MicroPython user C module is a directory with the following files:

- `*.c` and/or `*.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 it's 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 C files relative to your expanded copy of `$(USERMOD_DIR)` to `SRC_USERMOD`, eg `SRC_USERMOD += $(EXAMPLE_MOD_DIR)/example.c`

  If you have custom `CFLAGS` settings or include folders to define, these should be added to `CFLAGS_USERMOD`.

  See below for full usage example.