# MicroPython Lego Rl5 Documentation

*Release 1.11*

**Damien P. George, Paul Sokolovsky, StrangeButUntrue and contri**

**May 20, 2021**

# 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.
- Additions/deletions/modifications from the base Micropython version are indicated within the document.

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 this 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.

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).

In the RI5 port, since 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**

> **Difference for RI5**
>
> As with the array class, in RI5 this has methods `append()`, `extend()` and `decode()` that isn't in standard Micropython.

**class bytes**

See CPython documentation: `bytes`.

It's missing a lot of the more complicated or specialized functions of that class though.

**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**()

**execfile**()
Not in Python 3, but it does show up in Micropython.

**filter**()

**class float**
In MicroPython, this class doesn't have any methods.

**class frozenset**

**getattr**()

**globals**()

**hasattr**()

**hash**()

**help**()

**hex**()

**id**()

---

**Difference for RI5**

The input() function has been removed in RI5 - not unreasonably!

---

**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**
In MicroPython, this doesn't have any methods and can only be used with indices and slicing.

**min**()

**next**()

**class object**

**oct**()

**open**(*file*, *mode='r'*, *buffering=- 1*, *encoding=None*)
On RI5, this allows one to four arguments, so not as many as the corresponding CPython function.

**ord**()

**pow**()

**print**()

**Difference for RI5**

On the RI5, this function has been overwritten by an alias to a new builtin *spikeprint()*

**class property**

**range**()

**repr**()

**reversed**()

**round**()

**class set**

**setattr**()

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

**sorted**()

**spikeprint**()

**Difference for RI5**

A new function that overwrites *print()*, presumably so that print responses can be successfully sent back over the link to the controlling app.

**staticmethod**()

**class str**
See CPython documentation: `str`.

It's missing a lot of the more complicated or specialized functions of that class though.

**sum**()

**super**()

**class tuple**

**class type**

zip()

### Exceptions

exception BaseException

exception ArithmeticError

exception AssertionError

exception AttributeError

exception EOFError

exception Exception

exception GeneratorExit

exception ImportError

exception IndentationError

exception IndexError

exception KeyboardInterrupt

exception KeyError

exception LookupError

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 OverflowError

exception RuntimeError

exception StopAsyncIteration

exception StopIteration

exception SyntaxError

exception SystemExit
: See CPython documentation: SystemExit.

exception TypeError
: See CPython documentation: TypeError.

exception UnicodeError

exception ValueError

exception ZeroDivisionError

### Constants

**Ellipsis**
The same as the ellipsis literal "...". Special value used mostly in conjunction with extended slicing syntax for user-defined container data types.

**NotImplemented**
Special value which should be returned by the binary special methods (e.g. __eq__(), __lt__(), __add__(), __rsub__(), etc.) to indicate that the operation is not implemented with respect to the other type; may be returned by the in-place binary special methods (e.g. __imul__(), __iand__(), etc.) for the same purpose. It should not be evaluated in a boolean context.

Note: When a binary (or in-place) method returns NotImplemented the interpreter will try the reflected operation on the other type (or some other fallback, depending on the operator). If all attempts return NotImplemented, the interpreter will raise an appropriate exception. Incorrectly returning NotImplemented will result in a misleading error message or the NotImplemented value being returned to Python code.

See Implementing the arithmetic operations for examples.

Note: NotImplementedError and NotImplemented are not interchangeable, even though they have similar names and purposes. See NotImplementedError for details on when to use it.

## 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.

**decode**()
Outputs a string of the decoded array. Seems to treat each element as a byte and use ASCII encoding only?

---

**Difference for RI5**

This function is an extension from the base MicroPython version.

---

### 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.

#### 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. This is on by default for RI5 programs.

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.

---

gc.**isenabled**()
Returns true if automatic collection is enabled.

---

**Difference for RI5**

This function is an extension from the base MicroPython version.

---

### 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 RI5, early calculations show you get about 6 sig fig of precision for floats.

#### 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.

math.**isclose**(*a*, *b*, *, *rel_tol=1e-09*, *abs_tol=0.0*)
Return True if the values a and b are close to each other and False otherwise.

Whether or not two values are considered close is determined according to given absolute and relative tolerances.

rel_tol is the relative tolerance it is the maximum allowed difference between a and b, relative to the larger absolute value of a or b. For example, to set a tolerance of 5%, pass rel_tol=0.05. The default tolerance is 1e-09, which assures that the two values are the same within about 9 decimal digits. rel_tol must be greater than zero.

abs_tol is the minimum absolute tolerance useful for comparisons near zero. abs_tol must be at least zero.

If no errors occur, the result will be: abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol).

The IEEE 754 special values of NaN, inf, and -inf will be handled according to IEEE rules. Specifically, NaN is not considered close to any other value, including NaN. inf and -inf are only considered close to themselves.

---

**Difference for RI5**

This function is an extension from the base MicroPython version.

---

math.**factorial**(*x*)
Return x factorial. Raises ValueError if x is not integral or is negative.

---

**Difference for RI5**

This function is an extension from the base MicroPython version.

---

#### 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. Generally none for RI5 programs run normally.

`sys.byteorder`
The byte order of the system (`"little"` or `"big"`). Is little on the RI5

`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, 11, 0)
- *mpy* - number e.g. 517

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).

On RI5, it's 2147483647 (=0x7FFFFFFF).

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 RI5, this doesn't include Python-builtin modules but does include modules loaded from the filesystem.

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

On RI5, by default it's ['', '/flash', '/flash/lib']

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.

On RI5 it's "LEGO Learning System Hub".

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. E.g. "3.4.0".

sys.**version_info**
Python language version that this implementation conforms to, as a tuple of ints. E.g. (3, 4, 0)

### 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)])
```

(continues on next page)

(continued from previous page)

```
# 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")
```

**EPERM = 1**

**ENOENT = 2**

**EIO = 5**

**EBADF = 9**

**EAGAIN = 11**

**ENOMEM = 12**

**EACCES = 13**

**EEXIST = 17**

**ENODEV = 19**

**EISDIR = 21**

**EINVAL = 22**

**EOPNOTSUPP = 95**

**EADDRINUSE = 98**

**ECONNABORTED = 103**

**ECONNRESET = 104**

**ENOBUFS = 105**

**ENOTCONN = 107**

**ETIMEDOUT = 110**

**ECONNREFUSED = 111**

**EHOSTUNREACH = 113**

**EALREADY = 114**

**EINPROGRESS = 115**

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. RI5 implements:

- SHA256 - The current generation, modern hashing algorithm (of SHA2 series). It is suitable for cryptographically-secure purposes.

#### Constructors

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

---

**Difference for RI5**

Classes sha1 and md5 from the base MicroPython version are not implemented on the RI5.

---

#### 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'*, *buffering=-1*, *encoding=None*)
Open a file. Builtin `open()` function is aliased to this function.

### Classes

**class** uio.**IOBase**(...)
(Python abstract base class for IO objects still technically exists, although isinstance doesn't think the ones below are instances of it.)

**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 basic file methods (`read()`, `readinto()`, `readline()`, `readlines()`, `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` function.

#### 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. "Lego Technic Large Hub"
- `nodename` – the network name. "LEGO Learning System Hub"
- `release` – the version of the underlying system. The MicroPython release number on which this is based.
- `version` – a more exact MicroPython version and build date
- `machine` – an identifier for the underlying hardware (eg board, CPU). "Lego Technic Large Hub with STM32F413xx"

**Difference for RI5**

The function uos.urandom() from the base MicroPython version is not implemented on the RI5

### Constants

uos.**sep**
Filesystem path separator ''

**Difference for RI5**

This constant is no longer noted in the base MicroPython docs version.

### 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.
- 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. unlink() is also available and is semantically identical to this.

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. 4096 for the RI5.
- `f_frsize` – fragment size. 4096 for the RI5.
- `f_blocks` – size of fs in f_frsize units. 7936 for the RI5.
- `f_bfree` – number of free blocks.
- `f_bavail` – number of free blocks for unpriviliged users
- `f_files` – number of inodes. 0 for the RI5.
- `f_ffree` – number of free inodes. 0 for the RI5.
- `f_favail` – number of free inodes for unpriviliged users. 0 for the RI5.
- `f_flag` – mount flags. 0 for the RI5.
- `f_namemax` – maximum filename length. 255 for the RI5.

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.VfsLfs1`(*block_dev*)
Create a filesystem object that uses the littlefs v1 filesystem format. Storage of the littlefs filesystem is provided by *block_dev*. Objects created by this constructor can be mounted using *mount()*.

`static mkfs`(*block_dev*)
Build a littlefs filesystem on *block_dev*.

---

**Difference for RI5**

The base MicroPython version uses a VfsFat filesystem. The RI5 replaces that with VfsLfs1. This also means that block devices below use the extended interface so need to implement the offset parameter when doing reads and writes.

---

### 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*, *offset*)
Starting at the block given by the index *block_num* and from *offset* bytes into that block, read blocks from the device into *buf* (an array of bytes). The number of blocks to read is given by the length of *buf*.

`writeblocks`(*block_num*, *buf*, *offset*)
Starting at the block given by the index *block_num* and from *offset* bytes into that block, write blocks from *buf* (an array of bytes) to the device. The number of blocks to write is given by the length of *buf*.

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

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

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, offset):
        for i in range(len(buf)):
            buf[i] = self.data[block_num * self.block_size + offset + i]

    def writeblocks(self, block_num, buf, offset):
        for i in range(len(buf)):
            self.data[block_num * self.block_size + offset + 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
        if op == 6: # erase block
            self.writeblocks(arg, bytearray(self.block_size), 0)
```

It should be able to be used as follows, although this currently seems to fail with a TypeError on RI5 (so maybe something else is missing from the above):

```
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.

**Difference for RI5**

The base MicroPython version has a ure.DEBUG flag value that the RI5 doesn't have.

### 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.

**Difference for RI5**

The base MicroPython version has various methods that the RI5 doesn't: match.groups(), match.start(), match.end(), match.span()

## 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 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`.

#### 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.

### 1.1.18 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**: In contrast to Unix (and the Unix port of Micropython) using 1970-01-01 00:00:00 UTC, embedded ports including the RI5 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 like the RI5 however system time depends on `machine.RTC()` object. The current calendar time may be set using `machine.RTC().datetime(tuple)` function.

On the RI5, the RTC seems to keep running while the Hub is either switched on or plugged into a USB power source. When not powered in this way it resets to the value 473385600 seconds, or 2015-01-01 00:00:00 UTC. Also interesting

is that after a reset, the RTC only seems to start counting when it is first used to get the time since Epoch. (Use of the sleep/ticks functions don't count, but `machine.RTC()` functions `datetime()`, `wakeup()` and `calibration()` do start it going.)

The RI5 doesn't keep very exact time since its RTC is just based on CPU clock-cycles. The `machine.RTC().calibration()` function can be used to better approximate real time, but note that this may require tuning for a particular system, and the functions below should probably not be relied upon for exact timekeeping without access to an external time source.

### 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

Note that the RTC time can be successfully set to certain invalid values - I haven't experimented in detail with the behaviour of the RTC or this function after such events.

`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. *seconds* may be a floating-point number to sleep for a fractional number of seconds.

`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. On the RI5 it is 0x40000000. 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: for RI5 this is the CPU clock at 100MHz. This function is intended for very fine benchmarking or very tight real-time loops.

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 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).

---

### 1.1.19 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.

Current methods of the stream are `read()`, `readinto()` and `readline()`.

---

These libraries do exist in MicroPython, but aren't in the base docs.

### 1.1.20 urandom – random number generation

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

---

**Difference to CPython**

The pseudorandom algorithm used in Micropython is the Yasmarang algorithm , instead of the Mersenne Twister used in CPython.

---

This module allows generation of pseudo-random numbers, including setting of a seed. (These are probably not suitable for cryptographic purposes.)

On the RI5, the random generators do work without calling `seed()`. It's not immediately clear what seed is used by the module in this case, but there's no obvious repetition in outputs. If you want an explicit seed whose value is relatively difficult to predict, consider something like `utime.ticks_cpu()`.

### Functions

urandom.**seed**(*a*)
Initialize the pseudorandom number generator with integer a.

urandom.**randrange**(*stop*)

urandom.**randrange**(*start*, *stop*[, *step* ])
Return a randomly selected element from range(stop) or range(start, stop, step).

urandom.**randint**(*a*, *b*)
Return a random integer N between a and b (inclusive).

urandom.**getrandbits**(*n*)
Return an integer with n random bits.

urandom.**choice**(*seq*)
Return a random element of the given sequence.

urandom.**random**()
Return a float between 0 and 1 (not including 1).

urandom.**uniform**(*a*, *b*)
Return a float between a and b (b may or may not be included depending on floating-point rounding).

## 1.2 MicroPython-specific libraries

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

### 1.2.1 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.

The RI5 version of this module seems to be a lot like the STM32 port of Micropython, as you might expect given that the RI5 runs on a STM32F413. A lot of the RI5 specifics here line up with the code for that port.

A note of callbacks used by functions and class methods of machine module: all these callbacks should be considered as executing in an interrupt context. This is true for both physical devices with IDs >= 0 and "virtual" devices with negative IDs like -1 (these "virtual" devices are still thin shims on top of real hardware and real hardware interrupts). See *Writing interrupt handlers*.

#### Reset related functions

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

machine.**soft_reset**()
On RI5, resets to menu with a SystemExit, as though calling sys.exit(). Doesn't seem to change the reset_cause() below.

---

**Difference for RI5**

This function isn't in the base MicroPython, at least not the version RI5 was branched from. It's also not clear it does precisely the same thing as the soft_reset function in the latest base MicroPython version.

machine.**reset_cause**()
Get the reset cause. See *constants* for the possible return values. On RI5 it doesn't seem to report SOFT_RESET, and PWRON_RESET is only reported when the Hub has been both powered off and unplugged and then powered up. But the others all seem to work as you'd expect.

### 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.

On the RI5, the return value seems to be True if interrupt requests were successfully disabled, and False if they weren't (which you get for example if you try to disable twice without enabling in between).

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.

On the RI5, the parameter seems to determine whether the system actually tries to enable interrupts or not - i.e. if False it does nothing. This allows the user to nest calls to disable/enable, but to overlap them in other ways you would have to keep track of how many disable calls there have been yourself. Note that trying to enable twice with True parameters in succession seems to crash the system.

### Power related functions

machine.**freq**()

With no parameters, returns CPU frequency in hertz, or sets it with a parameter.

**Difference for RI5**

See below for RI5-specifics.

In RI5 this actually returns a tuple (S, H, P1, P2) of the various clock speed frequencies of the board. These can be set by passing one to four parameters to the function. Any unsupplied parameters are set in proportion to their default relationship to S, so setting freq(S/2) will divide everything by 2.

See STM32F413 board documentation for full details on the four clocks here, but in terms of RI5 observations:

- S = System Clock frequency. On my system, it seems to default to 96000000. Presumably this dictates how fast the CPU runs.
- H = AHB (Advanced High-Performance Bus) Clock frequency. On my system, it seems to default to 96000000. Possibly controls some aspect of USB, because setting this too low (e.g. 24MHz) seems to break the connection between RI5 and the app and lead to debug logs showing json messages with characters doubled or missing. And potentially other dangers - set at your peril!
- P1 = APB1 (Advanced Peripheral Bus 1) Clock frequency. On my system, it seems to default to 24000000. This seems to control most of the output systems - halving it causes the LED to flicker, and lights and sound go half as fast. It also has some impact on USB though and maybe some quite key systems too as a further

reduction broke the app connection and had me nervous I'd broken things more seriously for a while. Set at your peril!

- P2 = APB2 (Advanced Peripheral Bus 2) Clock frequency. On my system, it seems to default to 48000000. No obvious effect discovered yet.

Frequency changes persist between programs, but go back to defaults on a hard reset.

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).

Similar to other low-power functions, it's not clear how much use this is on the RI5. It doesn't turn off lights etc.

machine.**sleep**([*time_ms*])

---

**Note:** This function is deprecated, use *lightsleep()* instead.

---

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.

---

**Difference for RI5**

See below for the specifics of how this works in practice for RI5.

---

On the RI5, it's not clear that these are going to be particularly useful.

- Lightsleep seems to potentially cut off the USB connection if you're running via the Mindstorms app, requiring you to restart the Hub and then the app to regain connection between the two. It's possible this is just a bug.
- Deepsleep obviously restarts the Hub from its startup script which will put you back in the menu unless you've done extensive customization. Although when you run a program after that it is then possible to check for the DEEPSLEEP reset cause.
- It's not clear exactly how power-saving these modes are on the RI5. In tests, both seemed to turn the Hub LED red (or on one weird and memorable occasion, green) for the duration of the sleep.

---

**Difference for RI5**

Function `wake_reason()` is not implemented for the RI5.

### Miscellaneous functions

`machine.info([verbose])`

**Difference for RI5**

A function specific to the RI5.

Prints various information, including:

- ID = The hex of the `unique_id()`
- S = System Clock frequency. See `freq()` above.
- H = AHB (Advanced High-Performance Bus) Clock frequency. See `freq()` above.
- P1 = APB1 (Advanced Peripheral Bus 1) Clock frequency. See `freq()` above.
- P2 = APB2 (Advanced Peripheral Bus 2) Clock frequency. See `freq()` above.
- "qstr" section with similar information to `micropython.qstr_info()`
- "GC" section with similar information to `micropython.mem_info()`
- If the `verbose` parameter is defined then it also prints the GC memory layout that you get from `mem_info()`'s verbose mode.

`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).

**Difference for RI5**

Function `rng()` is not implemented for the RI5.

### Memory Access

machine.**mem8**
machine.**mem16**
machine.**mem32**

Supports machine memory access in 1-byte, 2-byte or 4-byte chunks. Access is via indexing, where the index is the memory address of the beginning of the chunk. (Attempts to use a wrongly aligned address for the chunk size cause a ValueError.)

Beware that slicing doesn't work properly and may cause a system failure!

On the RI5, be aware that 0 is a valid address: addressable memory goes from 0 to 1572863 (=0x17FFFF) inclusive, representing 1.5 MiB. Attempting to reference addresses outside of this causes a system restart. Addresses from 0x08a670 seem to all return 0xFF values though so I'm not sure how useful anything above this is…

The system lets you attempt to set address contents with `memX[index] = value`, but it doesn't seem to be effective - subsequent reads just show the old value again.

### Constants

**Difference for RI5**

IRQ wake value constants and wake-up reason constants are not present on the RI5.

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

Reset causes.

### 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 form of the identifier in the RI5 case is a string that specifies the board-name or cpu-name of the pin. Pins can also be selected via the cpu and board convenience classes within Pin.

Usage Model:

```
from machine import Pin

# create an output pin on pin X0
p0 = Pin('X0', Pin.OUT)

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

(continues on next page)

(continued from previous page)

```
p0.value(1)

# create an input pin on pin X1, with a pull up resistor
p2 = Pin('X1', Pin.IN, Pin.PULL_UP)

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

# reconfigure pin X0 in input mode
p0.mode(p0.IN)

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

### Constructors

**class** machine.**Pin**(*id*)

**class** machine.**Pin**(*id*, *mode*, *pull=None*, *af=-1*, ** value*, *alt=-1*)

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. If no settings are specified they will remain in their previous state.

The arguments are:

- `id` is mandatory and can be either an existing Pin object or a string that identifies one. String identifiers can be either the cpu-name or the board-name of a Pin.
- `mode` specifies the pin mode, which can be one of:
  - `Pin.IN` - Pin is configured for input (i.e. for getting values from the outside world into the CPU). If viewed as an output the pin is in high-impedance state.
  - `Pin.OUT` - Pin is configured for (normal) output (i.e. for letting the program set values).
  - `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. Some pins may not implement this mode.
  - `Pin.ALT` - Pin is configured to perform an alternative function. Available alt-functions are specific to particular pins. For a pin configured in such a way any other Pin methods (except *Pin.init()*) are not necessarily applicable (calling them will lead to undefined, or a hardware-specific, result).
  - `Pin.ALT_OPEN_DRAIN` - The Same as `Pin.ALT`, but the pin is configured as open-drain. Not all ports implement this mode.
  - `Pin.ANALOG` - The pin is configured as an analog input/output instead of digital, and voltages will be read instead of binary 0/1 states. Use the *ADC* class instead of the functions below on pins in this mode. Some pins may not implement this mode.
- `pull` specifies if the pin has a (weak) pull resistor attached, and can be one of:
  - `None`/`Pin.PULL_NONE` - No pull up or down resistor.
  - `Pin.PULL_UP` - Pull up resistor enabled.
  - `Pin.PULL_DOWN` - Pull down resistor enabled.

- `af` is a legacy (positional) alternative to `alt` with exactly the same values - the other parameter takes precedence over it.
- `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.
- `alt` specifies an alternate function for the pin. The values it can take are 0-15, although whether these will do anything is pin-dependent. The alt argument is valid only for `Pin.ALT` and `Pin.ALT_OPEN_DRAIN` modes and must be specified for those modes. For other modes it is ignored.

As specified above, the Pin class allows to set an alternate function or analog mode for a particular pin, but it does not specify any further operations on such a pin. Pins configured in alternate-function or analog mode are usually not used as GPIO but are instead driven by other hardware peripherals. Sometimes they may be usable with other machine subclasses. The only Pin 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*, *pull=None*, *af=-1 **, *value*, *alt=-1*`)`

Re-initialise the pin using the given parameters. If `value` is unspecified, it will not be set, but the other arguments are always set (to the defaults where suitable).

See the constructor documentation for details of the arguments.

Returns `None`.

Note that since this is modifying hardware state, changing parameters here will be reflected in any other objects referencing the same physical pin.

`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`.

Behaviour is undefined always for pins in ALT or ANALOG modes.

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**()

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

Pin.**off**()

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

Pin.**mode**()
Returns the pin mode in numeric form. See the constructor documentation for details of the `mode` argument. Unlike base MicroPython, the STM32 port doesn't let you set mode with this method.

Pin.**pull**()
Returns the pin pull state in numeric form. See the constructor documentation for details of the `pull` argument. Unlike base MicroPython, the STM32 port doesn't let you set pull with this method.

Pin.**irq**(*handler=None*, *trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING*, *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 (all optional and can be positonal or named) 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.

  These values can be OR'ed together to trigger on multiple events.
- `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 `None`. Since it doesn't return any kind of IRQ object, there's no way to turn the IRQ off again (presumably even when the program exits) so this function will probably be of limited use on the RI5 unless you want permanent control of a pin until the system is reset. And that's assuming you can make this work - all my attempts at using this function either did nothing (on pins that had constant value) or crashed the system!

Pin.**name**()
Returns the pin name. (This will be the CPU-based form.)

Pin.**names**()
Returns a list containing the cpu- and board-based names for the pin in that order.

Pin.**af_list**()
Returns an array of alternate functions available for this pin, in the form of constants like Pin.AF1_TIM1 and Pin.AF5_I2S4.

Pin.**port**()
Returns the pin port in numeric form. (A=0 to H=7)

Pin.**pin**()
Returns the pin number. (The 0-15 number of the pin that comes after the port letter.)

Pin.**gpio**()
: Returns the base address of the GPIO block associated with this pin.

Pin.**af**()
: Returns the currently configured alternate function of the pin in numeric form. This will match one of the allowed constants for the `af` argument to `init()`.

### Class methods

---

**Difference for RI5**

These three functions define global behaviour which persists past the end of a program back into the hub menu, and so could conceivably cause future programs to break.

---

Pin.**mapper**([*map_function*])
: Given a parameter, stores a global mapping function, which should be a function that takes a single string and returns a pin object. (ValueError is thrown if it returns something else.) Once specified this function takes precedence over the default way of mapping strings to Pins. It may return None, at which point other lookup types are attempted.

  With no parameter it returns the current mapper function.

Pin.**dict**(*mapping_dict*)
: Given a parameter, specifies a global dictionary to be used to map strings to pin objects. If specified, this method of mapping takes precedence over the default way of mapping strings to Pins. But the mapper function takes precedence over this.

  Note that nothing checks the output of this mapping to make sure it's actually returning a Pin object.

  With no parameter it returns the current mapping dictionary.

Pin.**debug**(*debug_info*)
: Sets global debug information on/off according to the boolean `debug_info`. Debug information (printed to stsandard output stream) tells you more about how a given string is mapped to a Pin object.

### Classes

**class** machine.**board**

**class** machine.**cpu**
: Two classes containing constant objects for all the pins on the system. In the cpu class these take the raw form cpu.A0 to cpu.H1. (Port letter plus pin number.) In the board class most are just named board.PA0 to board.PH1, but some have been translated into more meaningful names: A1 = `board.BUTTON3_SW`, A11 = `board.USB_DM`, A12 = `board.USB_DP`, A14 = `board.TEST_LED`.

  ---

  **Difference for RI5**

  The more meaningful board names here appear to be unique to the RI5, or at least I couldn't find other sources for them all online.

  ---

### Constants

The following constants are used to configure the pin objects.

**Pin.IN = 0**
**Pin.OUT = 1**
**Pin.OPEN_DRAIN = 17**
**Pin.ALT = 2**
**Pin.ALT_OPEN_DRAIN = 18**
**Pin.ANALOG = 3**
Selects the pin mode.

**Pin.PULL_NONE = 0**
**Pin.PULL_UP = 1**
**Pin.PULL_DOWN = 2**
Selects whether there is a pull up/down resistor.

**Pin.IRQ_FALLING = 0x10210000**
**Pin.IRQ_RISING = 0x10110000**
Selects the IRQ trigger type.

**Pin.OUT_PP = 1**
**Pin.OUT_OD = 17**
**Pin.AF_PP = 2**
**Pin.AF_OD = 18**
Legacy constants (synonyms for pin modes above).

**Pin.AFx_***
Lots of constants describing the possible alternate functions. These match the alternate function numbers described at https://github.com/micropython/micropython/blob/master/ports/stm32/boards/stm32f413_af.csv or at least the subset that this board supports.

### 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)
```

(continues on next page)

(continued from previous page)

```
# 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.

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

Read analog values from a pin.

---

**Difference for RI5**

Functions `channel()` (and associated class), `init()` and `deinit()` are not implemented for the RI5. Instead the `read_u16()` function allows analog reading, and the constants below are added too.

---

### Constructors

`class machine.ADC(channel)`
Create an ADC object with the given `channel` (a signed integer). This allows you to then read analog values on that channel. It's not clear to me yet what sources most of these channels represent - every one I tried did receive data from somewhere… The constants below seem to represent a few important ones though.

`class machine.ADC(pin)`
Create an ADC object from the given Pin object/pin name string.

On the RI5, this seems to be possible only on pins A0, B0, B1, C0-5 (although using it on C2 or C3 seems to conflict with something in the firmware, causing runtime errors after use).

### Methods

machine.**read_u16**()
Read the current channel value.

### Constants

Channel numbers to get particular information from.

**machine.ADC.VREF = 65535**
According to the STM32 port code, this channel just constantly reports the maximum ADC value, which seems to be 65535.

**machine.ADC.CORE_VREF = 17**
Perhaps this is core voltage data? In a test it seemed to get a variety of values of the form 0x3??3 (from 0x3B23 to 0x3CB3) and 0x4??4 (from 0x4274 to 0x45A4).

**machine.ADC.CORE_TEMP = 16**
Presumably gets core temperature sensor data. Running at room temperature I was seeing integer values around 11026 (0x2B12), varying in multiples of 16 to around +-200. Possibly the last hex digit is not relevant given the fact that it always seems to be the same as the first in these readings?

**machine.ADC.CORE_VBAT = 18**
Presumably this gets battery charge or voltage level data? In a test it seemed to get integer values around 2768 (0x0AD0), varying in multiples of 16 to around +-150. Possibly the last hex digit is not relevant given the fact that it always seems to be the same as the first in these readings?

### Other Experiments

Experiments often showed slightly different initial values settling down to a more steady range of values. I also found the sources for the rest of the channels below 16, apart from channel 1:

A0 for a while reported values from 0x1341 to 0x1501. Printing the ADC showed it's probably equal to channel 0 (i.e. ADC(0)).

A2 for a while reported values from 0x1001 to 0x1041. It presents as channel 2.

A3 for a while reported values from 0x11A1 to 0x12B1. It presents as channel 3.

A4 for a while reported values from 0x1041 to 0x1101. It presents as channel 4.

A5 for a while reported values from 0x1181 to 0x12A1. It presents as channel 5.

A6 for a while reported values from 0x0FB0 to 0x1011. It presents as channel 6.

A7 for a while reported values from 0x1241 to 0x1391. It presents as channel 7.

B0 for a while reported values from 0x25C2 to 0x2672. It presents as channel 8.

B1 for a while reported values from 0x11C1 to 0x1471. It presents as channel 9.

C0 for a while reported values from 0x10F1 to 0x1221. It presents as channel 10.

C1 for a while reported values from 0x3003 to 0x3173 and on another occasion reported values from 0xB23B to 0xBA8B. It presents as channel 11.

C2 for a while reported values from 0x1331 to 0x1EA1. It presents as channel 12. Although if you access channel 12 directly you tend to get much lower values it seems, and no runtime errors…

C3 gave two much higher readings of 0xAFEA and 0x2152, then settled down into lower readings between 0x1241 and 0x1551. It presents as channel 13, but if you access channel 12 directly you tend to get much lower values it seems, and no runtime errors…

C4 for a while reported values from 0xA0AA to 0xCA9C. It presents as channel 14.

C5 gave one much higher reading of 0x4E94, then for a while reported values from 0x0860 to 0x0AB0. It presents as channel 15.

### 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 different boards.

On the RI5, it appears that no UARTs are actually available - at least I couldn't find an ID that would allow me to create one. So available parameters are unknown and the rest of this section is a little light on RI5-specific detail as it's difficult to do experiments when you can't create the base object! It's based on the STM32 port code instead since that'll probably be quite similar…

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. Any additional parameters are passed on to the init function.

#### 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. Can be 8 or 9.
- *parity* is the parity, None, 0 (even) or 1 (odd).
- *stop* is the number of stop bits, 1 or 2.
- *timeout* is the timeout in milliseconds to wait for the first character.

- *timeout_char* is the timeout in milliseconds to wait between characters.
- *flow* is RTS | CTS where RTS == 256, CTS == 512
- *read_buf_len* is the character length of the read buffer (0 to disable).

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.**readchar**()
Receive a single character, and return it as an integer (or -1 on timeout).

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

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

UART.**writechar**(*char*)
Write a single character to the bus. `char` is the integer to write. Returns `None`.

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=0*, *hard=False*, *handler=None*)
Create a callback to be triggered when data is received on the UART.

- *trigger* can only be `UART.IRQ_RXIDLE`
- *hard* seems to specify whether it's a hardware or software interrupt?
- *handler* an optional function to be called when new characters arrive.

---

**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.

### Constants

**UART.RTS = 256**
Flow parameter setting for initialization.

**UART.CTS = 512**
Flow parameter setting for initialization.

**UART.IRQ_RXIDLE = 16**
IRQ flag "idle". (Replaces UART.RX_ANY from the base documentation.)

## 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=-1*, ...)
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. In this case sck/mosi/miso parameters must be specified.

Any extra parameters are passed to `init()`.

---

**Difference for RI5**

Unlike the specification in the base MicroPython docs, this constructor seems to always call `init()`, whether or not more parameters are provided to it.

---

---

**Difference for RI5**

The hardware SPI blocks on the RI5 and their default details are shown below. Pin details aren't printed when printing the object, so they've been deduced from machine.Pin() printing and https://github.com/micropython/micropython/blob/master/ports/stm32/boards/stm32f413_af.csv.

---

All SPIs have default baudrate=375000, polarity=0, phase=0, bits=8. Be careful with altering or deinitializing any of these, as any changes will persist beyond the life of your program, and can cause system failure.

SPI(1) - MISO=A6, MOSI=A7, SCK=A5, NSS=A4 or A15(?). No obvious effect if you slow it down or speed it up, but if you deinit it, the system slows to a crawl... A basic read gets the following set of bytes followed by zeros: 0,0,0,0,0,0,0x2F,0xC0,0,0,0,0,0x11,0x80,0x23,0,0x7F,0x80.

SPI(2) - MISO=C2, MOSI=C3, SCK=B13, NSS=A11 or B9 or B12(?). Seems to affect lots of systems like the screen/sound/lights/program loading if you slow it down... A basic read gets all zeros.

SPI(3) - MISO=B4, MOSI=B5, SCK=B3 or B12 or C10(?), NSS=A4 or A15(?). No obvious effect if you slow it down or deinit it. A basic read gets one zero and times out if you read more than one byte at a time.

### Methods

SPI.**init**(*baudrate=500000*, *, *polarity=0*, *phase=0*, *bits=8*, *firstbit=SPI.MSB*, *sck=None*, *mosi=None*, *miso=None*)

Initialise the SPI bus with the given parameters:

- `baudrate` is the SCK clock rate. Note that the default value isn't an available hardware baudrate, so those actually default to 375000 in practice.
- `polarity` can be 0 or 1, and is the level the idle clock line sits at. (Although it can actually take any byte value.)
- `phase` can be 0 or 1 to sample data on the first or second clock edge respectively. (Although it can actually take any byte value.)
- `bits` is the width in bits of each transfer. Must be 8.
- `firstbit` must be `SPI.MSB` (0).
- `sck`, `mosi`, `miso` are pins (machine.Pin) objects (or strings naming pins) 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).

The actual clock frequency may be lower than the requested baudrate. They are rounded down to the nearest available one. On RI5:

Hardware baudrates: 12000000, 6000000, 3000000, 1500000, 750000, 375000, 187500, 93750

Software baudrates = 16000000/(32*N) = 500000, 250000, 166666, 125000, 100000, 83333, ..., 1

The actual rate may be determined by printing the SPI object.

SPI.**deinit**()

Turn off the SPI bus.

SPI.**read**(*nbytes*, *write=0*)

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=0*)

Read into the buffer specified by `buf` while continuously writing the single byte given by `write`. Returns `None`.

SPI.**write**(*buf*)

Write the bytes contained in `buf`. Returns `None`.

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`.

### Constants

**SPI.MSB = 0**
set the first bit to be the most significant bit

**SPI.LSB = 128**
Set the first bit to be the least significant bit. (Cannot be used on RI5.)

### 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
                                # (in this port, actually scl and sda parameters also␣
↪required)

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*, *timeout=?*)
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* on the RI5 are 1 and 3 (see below) - in these cases specifying scl/sda is not allowed, and freq seems to be ignored.
- *scl* should be a pin object/string specifying the pin to use for SCL.
- *sda* should be a pin object/string specifying the pin to use for SDA.
- *freq* should be an integer which sets the maximum frequency for SCL.
- the *timeout* parameter isn't documented in base MicroPython, so it's not quite clear what it's meant to do or what the default is.

Printing I2C(1) or I2C(3) shows its pin/freq details, printing a custom I2C shows nothing much.

Setting a custom I2C on the same pins as a pre-existing one is accepted, but it does seem to modify behaviour of the pre-existing one. (Though so far that's meant just different error codes in my experiments - ETIMEDOUT rather than ENODEV/EINVAL.)

### I2C ids on the RI5

- I2C(1, scl=B6, sda=B7, freq=480000)
- I2C(3, scl=A8, sda=C9, freq=480000)

### General Methods

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

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

It's not clear whether user-defined I2Cs on RI5 can actually have parameters successfully changed with this function. Certainly attempting to use this function on I2C(1) or I2C(3) causes system failure.

---

**Difference for RI5**

Function `deinit()` to turn off the bus is not available on RI5.

---

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.

On RI5, scans of I2C(1) and I2C(3) seem to just return [].

### 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. They tend to throw an OSError of ETIMEDOUT if things aren't set up correctly to accept I2C commands.

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.

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.

The method returns `None`.

### class RTC – real time clock

The RTC is an independent clock that keeps track of the date and time. As noted in the utime module, it defaults to 2015-01-01 00:00:00 UTC and only seems to start running when it's used (calling any of the functions below except `info()` will start it running.)

Example usage:

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

### Constructors

**class** machine.**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

Very little error-checking seems to be done on the input values so be aware that the RTC will try to work with whatever you give it, with potentially undefined results. It won't successfully set years below 2000 though.

RTC.**wakeup**(*timeout*, *callback=None*)
Set the RTC wakeup timer to trigger repeatedly at every `timeout` milliseconds.

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. It's not clear what this does - it appears to be uninitialised on entry.

RTC.**info**()
Get information about the startup time and reset source. If the RTC hasn't yet been started, this seems to take value 1056964608 (= 0x3F000000). Otherwise, it's not quite clear what the value means - in experiments it always seems to have high bit 0x20000000 set, and then some value in the lower 0xFFFF. Base MicroPython docs claim:

- 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

But on the RI5 I've seen lower bit values of 16658 and 27119 which don't seem to correspond to a startup time…

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

Default calibration on the RI5 is 0, although it's presumably likely to vary by system (and maybe even vary over time?) what value is best to make the RTC approximate real time.

### 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.

---

**Difference for RI5**

This Timer class seems to be a bit broken in RI5, or at least it doesn't play nicely with the firmware - system failures that require pulling the battery and USB seem almost inevitable when trying to use `init()`, `deinit()` or `Timer.PERIODIC`. (Basically the only thing that will work is a Timer that's initialized on construction, runs once and then is discarded.) Periodic Timers will work, but of course you can't shut them off successfully and they will still try to pop after the program has ended, causing (you guessed it) system failure.

What you'd use instead seems a bit dependent on other requirements of your program - there's no obvious drop-in replacement for all circumstances.

---

### Constructors

**class** machine.**Timer**(*id=-1*, ...)

Construct a new virtual timer object. Id must equal -1 or be omitted. If more arguments are provided, this runs the `init()` function with them, causing the timer to be started. See `init()` below for a list.

---

**Difference for RI5**

Base MicroPython allows for various positive id values, specifying particular hardware timers. The RI5 seems to only allow -1 for a virtual timer, and the number may alternatively be omitted entirely.

---

In RI5 the only way to run `init()` successfully seems to be via this constructor - running the constructor with no extra parameters seems to leave the Timer in a weird half-initialized state that will cause system failure if you try to run `init()` or `deinit()` on it subsequently.

### Methods

Timer.**init**(*, *mode=Timer.PERIODIC*, *period=-1*, *callback=None*, *freq=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.
- `freq`
- `period`

  If specified, `freq` indicates the Hz frequency of when the timer will pop. Otherwise, `period` sets the number of milliseconds until the pop. A negative or zero value period or frequency causes an immediate pop.
- `callback`

  Specifies a function to run when the timer pops. This function must have one positional argument, which will be passed the Timer when it pops.

---

**Difference for RI5**

The STM32 port of Micropython (on which this seems to be based) has some extra possible arguments for this function for fuller control over how the Timer works.

---

---

**Difference for RI5**

Note as above - calling this function on any existing Timer on the RI5 (whether it's stopped or started) causes system failure! Use the constructor only, or give up on this class for the moment.

---

Timer.**deinit**()

Deinitialises the timer. Stops the timer.

---

**Difference for RI5**

Note as above - calling this function on any existing Timer on the RI5 (whether it's stopped or started) causes system failure! If you need a Timer that can be halted, you'll have to find something else for the moment.

---

### Constants

**Timer.ONE_SHOT = 1**
**Timer.PERIODIC = 2**
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.

Note that on the RI5, the WDT will continue running after a program ends, and it will still reset the system if it expires during another program or in the menu system. A hard reset will get rid of it though.

Example usage:

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

### 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.

On the RI5, note that if you've used a particular WDT before ever, feeding it starts it running, even if it wasn't running after a reset! The system seems to remember that a WDT has been used even after a full power off, although it may reset to the default timeout.

### Classes from default Micropython not present on the Hub

- class ADCChannel - read analog values from internal or external sources
- class SD - secure digital memory card

### 1.2.2 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.**pystack_use**()
An undocumented base-Micropython function. Return an integer presumably representing the current amount of stack being used by the "Pystack". I'm not sure whether that's a subset of the other stack or separate though! Or what the difference is…

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.

### 1.2.3 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)
```

(continues on next page)

(continued from previous page)

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

WWDG = uctypes.struct(0x40002c00, WWDG_LAYOUT)

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

### Defining structure layout

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

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

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

Following are encoding examples for various field types:

- Scalar types:

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

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

- Recursive structures:

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

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

- Arrays of primitive types:

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

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

- Arrays of aggregate types:

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

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

- Pointer to a primitive type:

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

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

- Pointer to an aggregate type:

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

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

- Bitfields:

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

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

uctypes.**BFUINT8**
uctypes.**BFINT8**

uctypes.**BFUINT16**
uctypes.**BFINT16**
uctypes.**BFUINT32**
uctypes.**BFINT32**
uctypes.**BFUINT64**
uctypes.**BFINT64**
uctypes.**BF_POS**
uctypes.**BF_LEN**
Types for bitfield operation - see above.

### 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).

These libraries do exist in MicroPython, but aren't in the base docs.

### 1.2.4 `utimeq` – heap queue with times

**Difference to CPython**

This is a MicroPython specific module. It's based on `heapq` but its implementation is more specialized than that, and it's not possible to use list operations like indexing on it.

This module uses the heap queue algorithm (priority queue algorithm) to create a heap queue where entries are popped based on which has the earliest time.

#### Classes

**class** utimeq.**utimeq**(*n*)

Create a utimeq heap queue with space for n entries. This size is static, and an attempt to push too many entries onto it will throw an IndexError with message 'queue overflow'.

The queue sorts itself by entries' *time* parameters, and then by the order in which entries were pushed for entries with equal times. So entries with lower *time* parameters get popped first, and for entries with the same *time* the first only that was pushed gets popped first.

The heap queue can be tested for non-emptiness with "if(heap)".

**push**(*time*, *obj*, *userdata*)

Push an entry onto the heap queue.

- The *time* parameter should be a number of ticks (see the utime module) compatible with `utime.ticks_diff()`.
- The `obj` and `userdata` parameters are not used internally, so the user can set them to anything. (The underlying MicroPython code suggests a callback and its arguments.)

**pop**(*list*)

Takes the entry with lowest *time* off the heap queue. Populates the first three items of the given list (which must already exist) with:

- The entry's *time*.
- The entry's `obj`.
- The entry's `userdata`.

Returns None.

**peektime**()

Returns the *time* of the current top item (i.e. the one that will be popped next) but without popping it.

### 1.2.5 _onewire – OneWire Protocol

This module allows sending and receiving of data on a Pin according to the OneWire protocol.

Functions which take a `pin` argument expect an argument that can be used to reference a Pin, i.e. something you can feed to the `machine.Pin()` function. In the RI5's case, this is a str.

#### Functions

`_onewire.`**`reset`**(*pin*)
Does a OneWire reset on the given pin. Returns true if a device presence pulse was detected, otherwise false.

`_onewire.`**`readbit`**(*pin*)
Reads and returns a single bit from the given pin, assuming the Onewire protocol.

`_onewire.`**`readbyte`**(*pin*)
Reads 8 bits from the given pin using the Onewire protocol, then returns them as an integer. (Assumes a low-bit-first model.)

`_onewire.`**`writebit`**(*pin*, *bitval*)
Writes a single bit with value `bitval` to the given pin, assuming the Onewire protocol.

`_onewire.`**`writebyte`**(*pin*, *byteval*)
From byte `byteval`, writes 8 bits to the given pin using the Onewire protocol. (Assumes a low-bit-first model.)

`_onewire.`**`crc8`**(*bytearray*)
Computes the 8-bit CRC-remainder of the given bytearray (or other buffered object). As expected for the Onewire protocol, it seems to use the CRC-8/MAXIM version.

## 1.3 MicroPython default libraries unavailable

Some default MicroPython functionality is missing from the Hub:

- usocket
- ussl
- _thread
- btree
- framebuf
- network
- ucryptolib

And some undocumented MicroPython modules that aren't in RI5:

- bluetooth
- lwip
- uasyncio (but note that the RI5 does have the async keyword)
- uwebsocket
- webrepl

## 1.4 Libraries specific to the Technic Hub

**Difference for RI5**

The following libraries are not found in default MicroPython. As such, documentation is mainly based on experimentation and internet sources since no code sources are available.

The following libraries are specific to the Technic Hub and are built into its Micropython.

### 1.4.1 hub – hub brick functionality

Classes and functions related to simple parts and abilities of the Hub brick itself, like buttons, ports, and internal sensors.

These are lower-level functions than the API ones (the API seems to import this module a lot) and as such they provide greater control, although with a corresponding amount of risk of things going wrong if you use them improperly.

#### Constants

hub.**__version__ = v1.0.06.0034-b0c335b**

#### Functions

hub.**info**(*???*)
???

hub.**power_off**(*???*)
???

hub.**repl_restart**(*???*)
???

hub.**status**(*???*)
???

hub.**led**(*???*)
???

hub.**temperature**(*???*)
???

hub.**file_transfer**(*???*)
???

### Imports

- Class *util.constants.Image*

### Objects and Classes

It's not quite clear from MicroPython help text whether these are classes or instances of classes. For the moment I'm assuming both, since some have different names from the object they point to. But if that's the case it's not clear where the class lives!

hub.**port**

**class** hub.**Port**(*???*)
???

**Constants**

**DETACHED = 0**
???

**ATTACHED = 1**
???

**A = Port(A)**

**B = Port(B)**

**C = Port(C)**

**D = Port(D)**

**E = Port(E)**

**F = Port(F)**
???

**MODE_DEFAULT = 0**
???

**MODE_FULL_DUPLEX = 1**
???

**MODE_HALF_DUPLEX = 2**
???

**MODE_GPIO = 3**
???

**Port(X)**

These are objects in their own right, with the following contents:

**Methods** .. method:: callback(???)

???

**info**(*???*)
???

**mode**(*???*)
???

**pwm**(*???*)
???

**Variables** .. data:: device

???? Observed value: None

**motor**

???? Observed value: None

hub.**display**

**class** hub.**Display**(*???*)

???

**pixel**(*???*)

???

**show**(*???*)

???

**callback**(*???*)

???

**clear**(*???*)

???

**rotation**(*???*)

???

hub.**button**

**class** hub.**Button**(*???*)

???

**Members**

**center = center**

**left = left**

**right = right**

**connect = connect**

Represent each of the four buttons on the Hub (the main button in the center, left, right, and the bluetooth connect button). The values are objects with the following contents:

**is_pressed**(*???*)

???

**was_pressed**(*???*)

???

**presses**(*???*)

???

**callback**(*???*)

???

**on_change**(*???*)

???

hub.**sound**

**class** hub.**Sound**(*???*)

???

**Methods**

**volume**(*???*)
???

**beep**(*???*)
???

**play**(*???*)
???

**callback**(*???*)
???

**Constants**

**SOUND_SIN = 0**
???

**SOUND_SQUARE = 1**
???

**SOUND_TRIANGLE = 2**
???

**SOUND_SAWTOOTH = 3**
???

hub.**motion**

**class** hub.**Motion**(*???*)
???

**Methods**

**gyroscope**(*???*)
???

**gyroscope_filter**(*???*)
???

**accelerometer**(*???*)
???

**accelerometer_filter**(*???*)
???

**position**(*???*)
???

**reset_yaw**(*???*)
???

**preset_yaw**(*???*)
???

**orientation**(*???*)
???

**gesture**(*???*)
???

**was_gesture**(*???*)
???

**callback**(*???*)
???

**Constants**

NONE = NULL
???

LEFTSIDE = leftside
???

RIGHTSIDE = rightside
???

DOWN = down
???

UP = up
???

FRONT = front
???

BACK = back
???

TAPPED = tapped
???

DOUBLETAPPED = doubletapped
???

SHAKE = shake
???

FREEFALL = freefall
???

hub.**battery**

**class** hub.**Battery**(*???*)
???

**Methods**

**voltage**(*???*)
???

**current**(*???*)
???

**temperature**(*???*)
???

**charger_detect**(*???*)
???

**info**(*???*)
???

**capacity_left**(*???*)
???

**Constants**

BATTERY_NO_ERROR = 0
???

**BATTERY_HUB_TEMPERATURE_CRITICAL_OUT_OF_RANGE = -2**
???

**BATTERY_TEMPERATURE_OUT_OF_RANGE = -2**
???

**BATTERY_TEMPERATURE_SENSOR_FAIL = -3**
???

**BATTERY_BAD_BATTERY = -4**
???

**BATTERY_VOLTAGE_TOO_LOW = -5**
???

**USB_CH_PORT_NONE = 0**
???

**USB_CH_PORT_SDP = 1**
???

**USB_CH_PORT_CDP = 2**
???

**USB_CH_PORT_DCP = 3**
???

**CHARGER_STATE_FAIL = -1**
???

**CHARGER_STATE_DISCHARGING = 0**
???

**CHARGER_STATE_CHARGING_ONGOING = 1**
???

**CHARGER_STATE_CHARGING_COMPLETED = 2**
???

hub.**bluetooth**

**class** hub.**bt**(*???*)
???

**info**(*???*)
???

**discoverable**(*???*)
???

hub.**ble**

**class** hub.**bluetooth**(*???*)
???

**rssi**(*???*)
???

**mac**(*???*)
???

**scan**(*???*)
???

**scan_result**(*???*)
???

**connect**(*???*)
???

**callback**(*???*)
???

hub.**supervision**

**class** hub.**supervision**(*???*)
???

**info**(*???*)
???

hub.**BT_VCP**

**class** hub.**BT_VCP**(*???*)
???

**setinterrupt**(*???*)
???

**isconnected**(*???*)
???

**any**(*???*)
???

**send**(*???*)
???

**recv**(*???*)
???

**read**(*???*)
???

**readinto**(*???*)
???

**readline**(*???*)
???

**readlines**(*???*)
???

**write**(*???*)
???

**close**(*???*)
???

**__del__**(*???*)
???

**__enter__**(*???*)
???

**__exit__**(*???*)
???

**callback**(*???*)
???

hub.**USB_VCP**

**class** hub.**USB_VCP**(*???*)
??? Has all the same contents as BT_VCP, with these extras:

**Methods**

**init**(*???*)
???

**Constants**

**RTS = 1**
???

**CTS = 2**
???

### 1.4.2 firmware – Firmware information and loading

**Warning:** By definition, a lot of the functions in here are probably quite dangerous to run as they can rewrite firmware. I haven't experimented with all of them so I can't even tell you which will cause system crashes, and which will be worse... Presumably this is the sort of thing you'll be wanting to run only if you want to load on a custom firmware. If you're doing that, feel free to add details on the functions and the exact dangers of using them!

See https://github.com/gpdaniels/spike-prime/issues/7 for some more (but currently still quite limited) details.

#### Functions

firmware.**info**()
Returns a dictionary containing various details about the firmware. For the version in this branch with no changes, it returns:

{'appl_checksum': 1231192444, 'new_appl_image_stored_checksum': 0, 'appl_calc_checksum': 1231192444, 'new_appl_valid': False, 'new_appl_image_calc_checksum': 0, 'new_image_size': 0, 'currently_stored_bytes': 0, 'upload_finished': True, 'spi_flash_size': '32 MBytes', 'valid': 0}

firmware.**appl_checksum**()
Returns a checksum of the firmware. For the version in this branch with no changes, that's 1231192444.

firmware.**appl_image_initialise**(*???params???*)
(Unknown - I wasn't confident I could run it safely. I suspect this might potentially be quite dangerous, at least if you've stored something in the image store at any point beforehand. Seems to take up to 1 positional argument.)

firmware.**appl_image_store**(*???params???*)
(Unknown - I wasn't confident I could run it safely. Seems to take up to 1 positional argument.)

firmware.**appl_image_read**(*start_pos?*)
Returns a bytearray - seems to return empty ones by default for all integer inputs. My hypothesis is that you're supposed to store an image in here with the appl_image_store() function and then appl_image_initialise() it to start running with it, but is this the whole firmware or just some particular necessary part of it that's different to what you might write with flash_write()?

I'm also hypothesising based on flash_read that the parameter is probably the byte to start reading at, and it returns empty bytearrays when there's no bytes stored to read.

firmware.**flash_read**(*start_pos*)
: Returns a bytearray of 32 bytes of the flash memory, starting at the given byte.

firmware.**flash_write**(*???*)
: (Unknown - I wasn't confident I could run it safely. Some internet research suggests this is the particularly dangerous one, so maybe don't try this at home unless you've got the cash for another hub! Seems to take 2 positional arguments - perhaps position to write and data to write in some order?)

firmware.**ext_flash_read_length**()
: Seems to crash the hub or at least make it unresponsive until after a battery pull or two. Not sure what it was trying to do!

firmware.**ext_flash_erase**()
: (Unknown - I wasn't confident I could run it safely. Probably another quite dangerous one?)

firmware.**erase_superblock**()
: (Unknown - I wasn't confident I could run it safely. The link at the top says that this erases the filesystem from the hub.)

firmware.**bootloader_version**()
: Returns a string specifying bootloader version. For the version in this branch with no changes, that's 'v0.5.01.0002-f75d82d'.

The following libraries are specific to the Technic Hub and are found in its filesystem.

### 1.4.3 _api – user API

This module contains most of the API functions that you're meant to call to operate the system as a user. The module itself is actually a backend to modules *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API*, which are nearly identical and seem to exist mainly in order to correctly brand the Hub in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

One interesting note is that submodule `large_technic_hub` only seems to appear in this module after you've imported either *mindstorms – Mindstorms branding of the user API* or *spike – Spike Prime branding of the user API*, so it's probably better to just use one of those unless you have very odd requirements.

All of the classes within the submodules (except LargeTechnicHub) are aliased in the main namespace for convenience.

#### Submodules

##### _api.distancesensor – distance sensor functions API

This module contains the API functions for user interaction with a distance sensor brick.

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

### DistanceSensor Class

class _api.distancesensor.**DistanceSensor**(*???*)

???

**Methods**

**_set_mode**(*???*)

???

**_set_range_mode**(*???*)

???

**_is_distance_sensor**(*???*)

???

**get_distance_cm**(*???*)

???

**get_distance_inches**(*???*)

???

**get_distance_percentage**(*???*)

???

**wait_for_distance_farther_than**(*???*)

???

**wait_for_distance_closer_than**(*???*)

???

**light_up**(*???*)

???

**light_up_all**(*???*)

???

**Constants**

**PERCENT = %**

???

**CM = cm**

???

**IN = in**

???

**_LONG_RANGE_MODE = (0, [(0, 0)])**

???

**_SHORT_RANGE_MODE = (1, [(1, 0)])**

???

**_LIGHT_MODE = (5, [(5, 0), (5, 1), (5, 2), (5, 3)])**

???

### Imports

- Function *_api.util.newSensorDisconnectedError*
- Function *utime.sleep_ms*
- Function *util.scratch.clamp*
- Function *util.sensors.is_type*
- Constant *util.constants.LPF2_FLIPPER_DISTANCE* = 62
- Constant *util.constants.PORTS* = {'C': Port(C), 'B': Port(B), 'D': Port(D), 'E': Port(E), 'A': Port(A), 'F': Port(F)}

## _api.forcesensor – force sensor functions API

This module contains the API functions for user interaction with a force sensor brick. (Included with Spike Prime but not in the Robot Inventor kit.)

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

### Functions

_api.forcesensor.**_get_port_device**(*???*)
???

_api.forcesensor.**_is_force_sensor**(*???*)
???

### ForceSensor Class

**class** _api.forcesensor.**ForceSensor**(*???*)
???

**Methods**

**wait_until_released**(*???*)
???

**is_pressed**(*???*)
???

**wait_until_pressed**(*???*)
???

**_is_pressed**(*???*)
???

**get_force_percentage**(*???*)
???

**get_force_newton**(*???*)
???

### Imports

- Function *_api.util.newSensorDisconnectedError*
- Function *utime.sleep_ms*
- Function *util.sensors.is_type*
- Constant *util.constants.LPF2_FLIPPER_FORCE* = 63
- Constant *util.constants.PORTS* = {'C': Port(C), 'B': Port(B), 'D': Port(D), 'E': Port(E), 'A': Port(A), 'F': Port(F)}

## _api.colorsensor – color sensor functions API

This module contains the API functions for user interaction with a color sensor brick.

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

### Functions

_api.colorsensor.**_get_port_device**(*???*)
???

_api.colorsensor.**_is_color_sensor**(*???*)
???

### Constants

_api.colorsensor.**_COLORLIST = ['black', 'violet', None, 'blue', 'cyan', 'green', None, 'yellow', None, 'red', 'white']**
A list to allow easy mapping from color values to color names.

_api.colorsensor.**_AMBIENT_MODE = (2, [(2, 0)])**
???

_api.colorsensor.**_LIGHT_MODE = (3, [(3, 0), (3, 1), (3, 2)])**
???

_api.colorsensor.**_COMBI_MODE = ([(1, 0), (0, 0), (5, 0), (5, 1), (5, 2), (5, 3)],)**
???

### ColorSensor Class

**class** _api.colorsensor.**ColorSensor**(*???*)
???

**Methods**

**light_up_all**(*???*)
???

**light_up**(*???*)
???

**get_reflected_light**(*???*)
???

**get_rgb_intensity**(*???*)
???

**get_red**(*???*)
???

**get_green**(*???*)
???

**get_blue**(*???*)
???

**get_ambient_light**(*???*)
???

**get_color**(*???*)
???

**_get_color**(*???*)
???

**_set_mode**(*???*)
???

**wait_until_color**(*???*)
???

**wait_for_new_color**(*???*)
???

### Imports

- Function *_api.util.newSensorDisconnectedError*
- Function *utime.sleep_ms*
- Function *util.scratch.clamp*
- Function *util.sensors.get_sensor_value*
- Function *util.sensors.is_type*
- Constant *util.constants.LPF2_FLIPPER_COLOR* = 61
- Constant *util.constants.PORTS* = {'C': Port(C), 'B': Port(B), 'D': Port(D), 'E': Port(E), 'A': Port(A), 'F': Port(F)}

## _api.motionsensor – motion sensor functions API

This module contains the API functions for user interaction with the motion sensor inside the Hub.

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

### MotionSensor Class

class _api.motionsensor.**MotionSensor**(*???*)
???

**Methods**

**get_pitch_angle**(*???*)
???

**get_roll_angle**(*???*)
???

**get_yaw_angle**(*???*)
???

**get_orientation**(*???*)
???

**get_gesture**(*???*)
???

**reset_yaw_angle**(*???*)
???

**was_gesture**(*???*)
???

**wait_for_new_orientation**(*???*)
???

**wait_for_new_gesture**(*???*)
???

**Constants**

**FALLING = falling**
???

**SHAKEN = shaken**
???

**TAPPED = tapped**
???

**DOUBLE_TAPPED = doubletapped**
???

**LEFT_SIDE = leftside**
???

**RIGHT_SIDE = rightside**
???

**FRONT = front**
???

**BACK = back**
???

**UP = up**
???

**DOWN = down**
???

### Imports

- Module *hub – hub brick functionality*
- Function *utime.sleep_ms*

### _api.statuslight – status light functions API

This module contains the API functions for user interaction with the status light around the main button on the Hub.

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

### Constants

**_api.statuslight._COLORMAP = {'white': 10, 'pink': 1, 'blue': 3, 'yellow': 7, 'orange': 8, 'violet': 2, 'azure': 4, 'red': 9, 'green': 6, 'cyan': 5, 'black': 0}**
Dictionary to map from names of colors to their values.

### StatusLight Class

**class** _api.statuslight.**StatusLight**(*???*)
???

**Methods**

**on**(*???*)
???

**off**(*???*)
???

### Imports

- Module *hub – hub brick functionality*

### _api.motor – motor functions API

This module contains the API functions for user interaction with a motor brick.

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

### Functions

_api.motor.**_is_motor**(*???*)
???

### Motor Class

**class** _api.motor.**Motor**(*???*)
???

**Methods**

**set_degrees_counted**(*???*)
???

**set_default_speed**(*???*)
???

**set_stop_action**(*???*)
???

**set_stall_detection**(*???*)
???

**get_position**(*???*)
???

**get_speed**(*???*)
???

**get_degrees_counted**(*???*)
???

**get_default_speed**(*???*)
???

**run_to_degrees_counted**(*???*)
???

**run_to_position**(*???*)
???

**run_for_degrees**(*???*)
???

**run_for_rotations**(*???*)
???

**run_for_seconds**(*???*)
???

**was_stalled**(*???*)
???

**was_interrupted**(*???*)
???

**start_at_power**(*???*)
???

**start**(*???*)
???

**stop**(*???*)
???

**Constants**

**BRAKE = brake**
???

**HOLD = hold**
???

**COAST = coast**
???

### Imports

- Module *hub – hub brick functionality*
- Function *_api.util.newSensorDisconnectedError*
- Function *_api.util.wait_for_async*
- Function *utime.sleep_ms*
- Function *util.motor.clamp_power*
- Function *util.motor.clamp_speed*
- Function *util.sensors.is_type*
- Constant *system.system* = <Main System object>
- Constant `util.constants.MOTOR_TYPES = (65, 48, 49, 75, 76)
- Constant *util.constants.PORTS* = {'C': Port(C), 'B': Port(B), 'D': Port(D), 'E': Port(E), 'A': Port(A), 'F': Port(F)}

## _api.button – button functions API

This module contains the API functions for user interaction with buttons. In particular the left and right buttons on the hub.

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

### Button Class

**class** _api.button.**Button**(*???*)
???

**Methods**

**was_pressed**(*???*)
???

**wait_until_pressed**(*???*)
???

**wait_until_released**(*???*)
???

**is_pressed**(*???*)
???

### _api.motorpair – paired motor functions API

This module contains the API functions for user interaction with pairs of motors. (Motors linked together in code to make dual-wheel vehicles easy to operate.)

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

#### Functions

_api.motorpair.**clamp_steering**(*???*)
???

_api.motorpair.**_is_motor**(*???*)
???

#### Constants

_api.motorpair.**_DISCONNECTED_ERROR = One or both of the motors has been disconnected.**
???

_api.motorpair.**_MOTOR_PAIRING_ERROR = The motors could not be paired.**
???

#### MotorPair Class

**class** _api.motorpair.**MotorPair**(*???*)
???

**Methods**

**get_default_speed**(*???*)
???

**set_default_speed**(*???*)
???

**set_stop_action**(*???*)
???

**set_motor_rotation**(*???*)
???

**start**(*???*)
???

**start_at_power**(*???*)
???

**start_tank**(*???*)
???

**start_tank_at_power**(*???*)
???

**stop**(*???*)
???

**move**(*???*)
???

**move_tank**(*???*)
???

**_move_with_speed**(*???*)
???

**was_interrupted**(*???*)
???

**Constants**

**BRAKE = brake**
???

**HOLD = hold**
???

**COAST = coast**
???

**CM = cm**
???

**IN = in**
???

**DEGREES = degrees**
???

**SECONDS = seconds**
???

**ROTATIONS = rotations**
???

### Imports

- Function *_api.util.wait_for_async*
- Function *system.movewrapper.from_steering*
- Function *util.motor.clamp_power*
- Function *util.motor.clamp_speed*
- Constant *system.system* = <Main System object>
- Constant *util.constants.PORTS* = {'C': Port(C), 'B': Port(B), 'D': Port(D), 'E': Port(E), 'A': Port(A), 'F': Port(F)}

### _api.lightmatrix – 5x5 display functions API

This module contains the API functions for user interaction with the 5x5 display on the front of the Hub.

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

#### LightMatrix Class

**class** _api.lightmatrix.**LightMatrix**(*???*)
???

**Methods**

**show_image**(*???*)
???

**off**(*???*)
???

**set_pixel**(*???*)
???

**write**(*???*)
???

#### Imports

- Module *hub – hub brick functionality*

### _api.util – utility functions for the API code

This module contains functions used by the API functions. (I don't believe they're designed to be a part of the API as they're not obviously documented anywhere.)

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

#### Functions

_api.util.**wait_for_async**(*???*)
???

_api.util.**newSensorDisconnectedError**(*???*)
???

### Imports

- Module *utime – time related functions*

## _api.speaker – speaker functions API

This module contains the API functions for user interaction with the speaker inside the Hub.

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

### Speaker Class

**class** _api.speaker.**Speaker**(*???*)

???

**Methods**

**beep**(*???*)

???

**start_beep**(*???*)

???

**stop**(*???*)

???

**get_volume**(*???*)

???

**set_volume**(*???*)

???

### Imports

- Module *hub – hub brick functionality*
- Function *_api.util.wait_for_async*
- Constant *system.system* = <Main System object>

## _api.app – application functions API

This module contains the API functions for user interaction with the controlling application (i.e. computer or phone).

This module is imported by the *mindstorms – Mindstorms branding of the user API* and *spike – Spike Prime branding of the user API* modules so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

### Constants

_api.app.**_NOT_CONNECTED_ERROR = The programming app is not connected to the hub.**
Error text for when the app and the hub have become disconnected.

### App Class

**class** _api.app.**App**(*???*)
???

**Methods**

**play_sound**(*???*)
???

**_play_sound**(*???*)
???

**start_sound**(*???*)
???

### Imports

- Class *protocol.ujsonrpc.JSONRPC*
- Function *utime.ticks_diff*
- Function *utime.ticks_ms*
- Constant *util.constants.BT_VCP* = BT_VCP(0)
- Constant *util.constants.USB_VCP* = USB_VCP(0)

## _api.large_technic_hub – central hub API

This module contains the specific instances of other API classes for user interaction with various aspects of the central hub brick.

The class in this module is superclassed by the *mindstorms.MSHub* and *spike.PrimeHub* classes so that the API can be branded appropriately in documentation (i.e. so that Mindstorms docs can tell you to import mindstorms, and Spike Prime docs can tell you to import spike).

### LargeTechnicHub Class

**class** _api.large_technic_hub.**LargeTechnicHub**(*???*)
???

**Properties**

**property status_light**
???

**property light_matrix**
???

**property left_button**
???

**property right_button**
???

**property motion_sensor**
???

**property speaker**
???

**Constants**

**PORT_A = A**

**PORT_B = B**

**PORT_C = C**

**PORT_D = D**

**PORT_E = E**

**PORT_F = F**
Constants to specify specific ports on the Hub.

**_status_light**
A reference to the specific *_api.statuslight.StatusLight* object representing the status light under the main button on the Hub.

**_light_matrix**
A reference to the specific *_api.lightmatrix.LightMatrix* object representing the 5x5 display on the Hub.

**_left_button**
A reference to the specific *_api.button.Button* object representing the left button on the Hub.

**_right_button**
A reference to the specific *_api.button.Button* object representing the right button on the Hub.

**_motion_sensor**
A reference to the specific *_api.motionsensor.MotionSensor* object representing the motion sensor in the Hub.

**_speaker**
A reference to the specific *_api.speaker.Speaker* object representing the speaker in the Hub.

### Imports

- Module *hub – hub brick functionality*
- Function *_api.button.Button*
- Function *_api.lightmatrix.LightMatrix*
- Function *_api.motionsensor.MotionSensor*
- Function *_api.speaker.Speaker*
- Function *_api.statuslight.StatusLight*

### 1.4.4 commands – commands module

Module containing a lot of high-level concepts, but no obvious theme beyond that. Mostly organised into submodules *_methods containing *Methods classes, which are all also aliased in the main module for convenience and seem to implement the *commands.abstract_handler.AbstractHandler* class.

#### Submodules

##### commands.abstract_handler – base class for handler classes

Contains an abstract base class implemented by the various *Methods classes in the *commands – commands module* module.

#### AbstractHandler Class

**class** commands.abstract_handler.**AbstractHandler**(*???*)
???

**Methods**

**abstract get_methods**(*???*)
Implemented by all the classes that implement this class. ???

**Constants**

**_rpc = None**
???

##### commands.linegraphmonitor_methods – ???

???

#### LinegraphMonitorMethods Class

**class** commands.linegraphmonitor_methods.**LinegraphMonitorMethods**(*???*)
???

**Methods**

**__init__**(*???*)
Closure function. ???

**get_methods**(*???*)
???

**handle_delete_file**(*???*)
???

**handle_get_linegraph_monitor_info**(*???*)
???

**_error_if_running**(*???*)
???

**handle_get_linegraph_monitor_package**(*???*)
???

### Imports

- Module *commands.abstract_handler.AbstractHandler*
- Module *math – mathematical functions*
- Module *uos – basic "operating system" services*
- Function *micropython.const*
- Function *utime.sleep_ms*
- Function *utime.ticks_diff*
- Function *utime.ticks_ms*
- Constant *util.constants.LINEGRAPH_DIR* = /data/linegraph
- Constant `util.storage.ENOENT` = 2

## commands.sound_methods – ???

???

### SoundMethods Class

**class** commands.sound_methods.**SoundMethods**(*???*)

???

**Methods**

**__init__**(*???*)

Closure function. ???

**get_methods**(*???*)

???

**handle_sound_beep_for_time**(*???*)

???

**handle_play_sound**(*???*)

???

**handle_sound_off**(*???*)

???

**handle_sound_beep**(*???*)

???

### Imports

- Module *commands.abstract_handler.AbstractHandler*
- Module *hub – hub brick functionality*

## commands.light_methods – ???

???

### LightMethods Class

**class** commands.light_methods.**LightMethods**(*???*)
???

**Methods**

**__init__**(*???*)
Closure function. ???

**get_methods**(*???*)
???

**handle_display_rotate_direction**(*???*)
???

**handle_display_rotate_orientation**(*???*)
???

**handle_ultrasonic_light_up**(*???*)
???

**handle_display_clear**(*???*)
???

**handle_display_animation**(*???*)
???

**handle_center_button_lights**(*???*)
???

**handle_display_set_pixel**(*???*)
???

**handle_display_sync**(*???*)
???

**handle_display_image_for**(*???*)
???

**show_frames**(*???*)
???

**handle_display_image**(*???*)
???

**handle_display_text**(*???*)
???

**static _merge_display_params**(*???*)
???

**Constants**

**DEFAULT_DISPLAY_PARAMS = {'fade': 0, 'delay': 500, 'wait': False, 'loop': False, 'clear': False}**

### Imports

- Module *commands.abstract_handler.AbstractHandler*
- Module *hub – hub brick functionality*
- Function *util.rotation.rotate_hub_display*
- Function *util.rotation.rotate_hub_display_to_value*
- Function util.scratch.number_color_to_rgb
- Function *util.scratch.percent_to_int*
- Function *util.sensors.set_display_sync*
- Constant *util.constants.NO_STATUS* = -1
- Constant *util.constants.PORTS* = {'C': Port(C), 'B': Port(B), 'D': Port(D), 'E': Port(E), 'A': Port(A), 'F': Port(F)}

## commands.program_methods – ???

???

### Constants

commands.program_methods.**_PRINT_OVERRIDE = "from util.print_override import spikeprint;print = spikeprint\n"**
Code to override the regular print statement with the special RI5 one.

commands.program_methods.**_TRANSFER_HANDLE = {}**
???

### ProgramMethods Class

**class** commands.program_methods.**ProgramMethods**(*???*)
???

**Methods**

**__init__**(*???*)
Closure function. ???

**get_methods**(*???*)
???

**handle_write_package**(*???*)
???

**handle_program_reset_time**(*???*)
???

**handle_program_start_time**(*???*)
???

**handle_soft_reset**(*???*)
???

**_handle_write_print_override**(*???*)
???

**handle_program_execute**(*???*)
???

**handle_start_write_program**(*???*)
???

**handle_remove_project**(*???*)
???

**handle_program_terminate**(*???*)
???

**handle_program_modechange**(*???*)
???

**handle_program_get_time**(*???*)
???

**handle_storage_status**(*???*)
???

**handle_move_project**(*???*)
???

### Imports

- Module *commands.abstract_handler.AbstractHandler*
- Module *protocol.notifications – ???*
- Module *sys – system specific functions*
- Module *urandom – random number generation*
- Module *util.storage – storage utility module*
- Module *utime – time related functions*
- Function *micropython.const*
- Function *ubinascii.a2b_base64*
- Function *util.time.get_time*
- Function *util.time.reset_time*
- Function *util.time.start_time*

## commands.motor_methods – ???

???

### MotorMethods Class

**class** commands.motor_methods.**MotorMethods**(*???*)

???

**Methods**

**__init__**(*???*)

Closure function. ???

**get_methods**(*???*)

???

**handle_motor_pwm**(*???*)

???

**handle_motor_go_direction_to_position**(*???*)

???

**handle_motor_run_timed**(*???*)

???

**handle_motor_stop**(*???*)

???

**handle_motor_start**(*???*)

???

**handle_motor_run_for_degrees**(*???*)

???

**handle_motor_set_position**(*???*)

???

**handle_motor_go_to_relative_position**(*???*)

???

**handle_motor_adjust_offset**(*???*)

???

**handle_motor_position**(*???*)

???

### Imports

- Module *commands.abstract_handler.AbstractHandler*
- Module *hub – hub brick functionality*
- Function *event_loop.get_event_loop*
- Constant *util.constants.NO_STATUS* = -1
- Constant *util.constants.PORTS* = {'C': Port(C), 'B': Port(B), 'D': Port(D), 'E': Port(E), 'A': Port(A), 'F': Port(F)}

## commands.hub_methods – ???

???

### HubMethods Class

class commands.hub_methods.**HubMethods**(*???*)
???

**Methods**

**__init__**(*???*)
Closure function. ???

**get_methods**(*???*)
???

**handle_trigger_current_state**(*???*)
???

**handle_set_port_mode**(*???*)
???

**handle_set_hub_name**(*???*)
???

**handle_get_hub_info**(*???*)
???

**handle_reset_yaw**(*???*)
???

### Imports

- Module *commands.abstract_handler.AbstractHandler*
- Module *hub – hub brick functionality*
- Module *protocol.notifications – ???*
- Module *version – version module*
- Function *ubinascii.a2b_base64*
- Function *util.storage.write_local_name*
- Constant util.storage.ENOENT = 2

## commands.wait_methods – ???

???

### WaitMethods Class

**class** commands.wait_methods.**WaitMethods**(*???*)
???

**Methods**

**__init__**(*???*)
Closure function. ???

**get_methods**(*???*)
???

**handle_when_sensor_changed**(*???*)
???

**handle_when_sensor_force_bumped**(*???*)
???

**handle_when_sensor_force_released**(*???*)
???

**handle_wait_gesture**(*???*)
???

### Imports

- Module *commands.abstract_handler.AbstractHandler*

## commands.move_methods – ???

???

### MoveMethods Class

**class** commands.move_methods.**MoveMethods**(*???*)
???

**Methods**

**__init__**(*???*)
Closure function. ???

**get_methods**(*???*)
???

**handle_move_tank_degrees**(*???*)
???

**handle_move_tank_time**(*???*)
???

**handle_move_start_powers**(*???*)
???

**handle_move_stop**(*???*)
???

**handle_move_start_speeds**(*???*)
???

### Imports

- Module *commands.abstract_handler.AbstractHandler*
- Constant *util.constants.NO_STATUS* = -1

## 1.4.5 event_loop – event_loop module

Home of the Event loop class - this seems to be the main scheduler on the Hub, called by the *hub_runtime – Hub main module* module's *start()* procedure. The EventLoop class (along with the imports) seem to technically live inside an event_loop.event_loop submodule, but the class is also available inside event_loop itself, so the submodule isn't necessary to know about.

### Functions

event_loop.**get_event_loop**(*???*)
???

### Constants

event_loop.**_EVENT_LOOP**
Reference to the main event loop object (type EventLoop).

### Class EventLoop

**class** event_loop.**EventLoop**(*???*)
???

**_discard**(*???*)
???

**step**(*???*)
Generator function. ???

**cancel**(*???*)
???

**call_soon**(*???*)
???

**run_forever**(*???*)
???

### Imports

- Module *ucollections – collection and container types*
- Module *utimeq – heap queue with times*
- Module *utime – time related functions*

## 1.4.6 `mindstorms` – Mindstorms branding of the user API

This module seems to be purely designed as a frontend for the *_api – user API* module, so that Robot Inventor Mindstorms documentation can tell its users to import a module with a related name.

See *_api – user API* for the majority of submodules and classes, all of which can also be called from the mindstorms module by using it as a synonym for "_api".

### Classes

**class** mindstorms.**MSHub**

The actual class appears to have no methods or functions of its own, but it seems to be a superclass of *_api.large_technic_hub.LargeTechnicHub* so inherits everything from there.

The API expects most access to the central Technic Hub and its functions to use an instance of this class.

## 1.4.7 `programrunner` – run user programs

This module handles the running of user programs (i.e. the Scratch/Python programs that live in the program slots on the system).

### Functions

programrunner.**filter_dict_len**(*???*)

???

programrunner.**map_dirty**(*???*)

???

programrunner.**filter_vm_vars**(*???*)

???

programrunner.**sum_list_len**(*???*)

???

programrunner.**filter_vm_lists**(*???*)

???

programrunner.**setup_vm**(*???*)

Generator function. ???

programrunner.**untuple_vm_vars**(*???*)

Generator function. ???

### Constants

programrunner._EMPTY_DICT = {}
???

### Class ProgramRunner

class programrunner.ProgramRunner(*???*)
???

**Methods**

vm_has_extension(*???*)
???

start_program(*???*)
???

is_running(*???*)
???

start_notify_loop(*???*)
Generator function. ???

notify_all_state(*???*)
???

stop_all(*???*)
???

**Constants**

IDLE = 0
???

RUNNING_NONBLOCKING = 1
???

RUNNING_BLOCKING = 2
???

### Imports

- Module *util.sensors – sensors utility module*
- Module *sys – system specific functions*
- Module *hub – hub brick functionality*
- Module *gc – control the garbage collector*
- Module *protocol.notifications – ???*
- Class *runtime.virtualmachine.VirtualMachine*
- Function *micropython.const*
- Function *event_loop.get_event_loop*
- Function *util.resetter.wait_until_ready_after_restart*
- Function *util.rotation.rotate_hub_display_to_orientation*

- Function *util.storage.get_path*
- Function *util.storage.set_force_reset*
- Function *util.storage.get_program_project_id*
- Function *util.storage.get_program_type*
- Function *util.time.reset_time*
- Function *util.time.stop_time*
- Constant *util.constants.LPF2_FLIPPER_DISTANCE* = 62
- Constant *util.constants.TIMER_PACE_LOW* = 48
- Constant *util.constants.TIMER_PACE_HIGH* = 16
- Constant *util.storage.PROGRAM_TYPE_PYTHON* = python
- Constant *util.storage.PROGRAM_TYPE_SCRATCH* = scratch
- Constant *util.error_handler.PROGRAM_EXECUTION_ERROR* = 0
- Constant *util.error_handler.PROGRAM_EXECUTION_MEMORY_ERROR* = 1
- Constant *util.error_handler.error_handler* = <Main ErrorHandler object>

### 1.4.8 protocol – RI5 communication protocol

This module handles the communication protocol that the RI5 uses when talking over USB/Bluetooth to a controller app. The protocol uses a specific json format. The base module has three submodules, and aliases protocol.rpc_protocol.RPCProtocol as protocol.RPCProtocol for convenience.

#### Submodules

#### protocol.notifications – ???

???

#### Functions

protocol.notifications.**notify_error_event**(*???*)
???

protocol.notifications.**notify_sensor_data**(*???*)
???

protocol.notifications.**notify_storage_status**(*???*)
???

protocol.notifications.**notify_stack_start**(*???*)
???

protocol.notifications.**notify_battery_status**(*???*)
???

protocol.notifications.**notify_program_running**(*???*)
???

protocol.notifications.**notify_gesture_event**(*???*)
???

protocol.notifications.**notify_info_status**(*???*)
???

protocol.notifications.**notify_vm_state**(*???*)
???

protocol.notifications.**notify_stack_stop**(*???*)
???

protocol.notifications.**notify_linegraph_timer_reset**(*???*)
???

protocol.notifications.**notify_debug_event**(*???*)
???

protocol.notifications.**notify_button_event**(*???*)
???

protocol.notifications.**notify_gesture_status**(*???*)
???

### Constants

protocol.notifications.**_RQ_LEN = run_queue_len**
???

protocol.notifications.**_MEM = mem_alloc**
???

protocol.notifications.**_D = mem_delta**
???

protocol.notifications.**_DEBUG_PAYLOAD = {'wait_queue_len': 0, 'mem_delta': 0, 'run_queue_len': 0, 'mem_alloc': 0}**
???

protocol.notifications.**_WQ_LEN = wait_queue_len**
???

### Imports

- Module *hub – hub brick functionality*
- Function *gc.mem_alloc*
- Function *micropython.const*
- Function *ubinascii.b2a_base64*
- Function *util.storage.get_storage_information*
- Function *util.storage.read_local_name*
- Variable *util.sensors.battery_status*
- Variable *util.sensors.sensor_data*

### protocol.rpc_protocol – ???

Aside from importing various functions from elsewhere, this module just contains the RPCProtocol class.

#### RPCProtocol Class

**class** protocol.rpc_protocol.**RPCProtocol**(*???*)
???

**register_method_handlers**(*???*)
???

**_register_method_handler**(*???*)
???

**looper**(*???*)
Generator method. ???

**property stream**
???

#### Imports

- Function *event_loop.get_event_loop*
- Function *micropython.const*
- Function *protocol.notifications.notify_battery_status*
- Function *protocol.notifications.notify_debug_event*
- Function *protocol.notifications.notify_sensor_data*
- Function *util.sensors.update_battery_status*
- Function *util.sensors.update_sensor_data*
- Class *protocol.ujsonrpc.JSONRPC*

### protocol.ujsonrpc – ???

???

#### Constants

protocol.ujsonrpc.**_ID_PREFIX = b'{"i":'**
???

protocol.ujsonrpc.**_PARAMS = b',"p":'**
???

protocol.ujsonrpc.**_ID = b',"i":'**
???

protocol.ujsonrpc.**_ERROR = b',"e":'**
???

protocol.ujsonrpc._SUFFIX = b'}\r'
???

protocol.ujsonrpc._SUSPENDED_MSG_PATH_ = ./suspended_msg
???

protocol.ujsonrpc._CARRIAGE_RETURN = b'\r'
???

protocol.ujsonrpc.NO_RESPONSE = {}
???

protocol.ujsonrpc._METHOD_PREFIX = b'{"m":'
???

protocol.ujsonrpc._RESPONSE = b',"r":'
???

### JSONRPC Class

class protocol.ujsonrpc.JSONRPC(*???*)
???

*Methods*

_pop_suspend_message(*???*)
???

clear_methods(*???*)
???

emit_large(*???*)
???

reply(*???*)
???

_handle_message(*???*)
???

suspend_current_message(*???*)
???

error(*???*)
???

add_method(*???*)
???

parse_chunk(*???*)
???

cancel_call(*???*)
???

parse_buffer(*???*)
???

emit(*???*)
???

**call**(*???*)
???

**resume_suspended_msg**(*???*)
???

**property stream**
???

*Fields* .. data:: pending

??? Default value = {}

*Method Dictionary*

**methods = {'scratch.display_animation': <bound_method>, 'scratch.motor_pwm': <bound_method>, 'set_hub_name': <bound_method>, 'scratch.play_sound': <bound_method>, 'get_linegraph_monitor_info': <bound_method>, 'reset_program_time': <bound_method>, 'set_port_mode': <bound_method>, 'scratch.motor_go_direction_to_position': <bound_method>, 'sync_display': <bound_method>, 'scratch.reset_yaw': <bound_method>, 'scratch.when_sensor_changed': <bound_method>, 'scratch.motor_run_timed': <bound_method>, 'scratch.move_stop': <bound_method>, 'program_execute': <bound_method>, 'scratch.when_sensor_force_released': <bound_method>, 'remove_project': <bound_method>, 'start_write_program': <bound_method>, 'get_storage_status': <bound_method>, 'scratch.sound_beep': <bound_method>, 'scratch.sound_off': <bound_method>, 'scratch.display_set_pixel': <bound_method>, 'scratch.ultrasonic_light_up': <bound_method>, 'scratch.motor_start': <bound_method>, 'delete_linegraph_file': <bound_method>, 'program_terminate': <bound_method>, 'scratch.display_rotate_direction': <bound_method>, 'scratch.display_image': <bound_method>, 'scratch.move_start_powers': <bound_method>, 'scratch.sound_beep_for_time': <bound_method>, 'get_program_time': <bound_method>, 'move_project': <bound_method>, 'get_hub_info': <bound_method>, 'scratch.center_button_lights': <bound_method>, 'scratch.motor_position': <bound_method>, 'scratch.move_start_speeds': <bound_method>, 'program_modechange': <bound_method>, 'scratch.move_tank_degrees': <bound_method>, 'scratch.motor_go_to_relative_position': <bound_method>, 'write_package': <bound_method>, 'scratch.display_rotate_orientation': <bound_method>, 'scratch.display_image_for': <bound_method>, 'scratch.when_sensor_force_bumped': <bound_method>, 'scratch.move_tank_time': <bound_method>, 'scratch.wait_gesture': <bound_method>, 'scratch.motor_run_for_degrees': <bound_method>, 'trigger_current_state': <bound_method>, 'scratch.display_clear': <bound_method>, 'scratch.motor_stop': <bound_method>, 'scratch.motor_adjust_offset': <bound_method>, 'start_program_time': <bound_method>, 'scratch.motor_set_position': <bound_method>, 'scratch.display_text': <bound_method>, 'get_linegraph_monitor_package': <bound_method>}**
Seems to be a lookup table binding scratch API commands to specific methods?

#### Imports

- Module *ujson – JSON encoding and decoding*
- Module *uos – basic "operating system" services*
- Module *urandom – random number generation*
- Function *ubinascii.b2a_base64*
- Function *utime.sleep_ms*
- Constant uerrno.ENOENT = 2
- Class *uio.StringIO*

### 1.4.9 runtime – runtime module

Contains mainly stack and virtual machine details needed to run programs on the hub.

Classes *runtime.stack.Stack*, *runtime.virtualmachine.VirtualMachine* and *runtime.multimotor.MultiMotor* have shortcut aliases in the main namespace.

#### Submodules

##### runtime.dirty_dict – ???

Contains the DirtyDict class. ???

#### DirtyDict Class

**class** runtime.dirty_dict.**DirtyDict**(*???*)
???

**__delitem__**(*???*)
Closure function. ???

**_mark_dirty**(*???*)
???

**dict_get**(*???*)
???

**list_insert**(*???*)
???

**__init__**(*???*)
Closure function. ???

**list_del**(*???*)
???

**list_clear**(*???*)
???

**setitem**(*???*)
Closure function. ???

**dict_set**(*???*)
???

**list_append**(*???*)
???

**list_set**(*???*)
???

**dirty_items**(*???*)
Generator function. ???

**clear**(*???*)
Closure function. ???

### runtime.multimotor – ???

Contains the MultiMotor class. ???

#### MultiMotor Class

**class** runtime.multimotor.**MultiMotor**(*???*)
???

**await_all**(*???*)
Generator function. ???

**run**(*???*)
???

### runtime.stack – ???

Contains the Stack class. ???

#### Stack Class

**class** runtime.stack.**Stack**(*???*)
???

**Methods** .. method:: is_active(???)

???

**should_start**(*???*)
???

**_check_condition**(*???*)
Generator function. ???

**stop**(*???*)
???

**restart**(*???*)
???

**start**(*???*)
???

**Constants** .. data:: STATUS_RUNNING

**value** 10

???

**STATUS_IDLE = 20**
???

**STATUS_WAITING = 30**
???

**ON_START = 0**
???

**ON_BROADCAST = 1**
???

**ON_BUTTON = 2**
???

**ON_GESTURE = 3**
???

**ON_CONDITION = 4**
???

### Imports

- Function *micropython.const*
- Function *protocol.notifications.notify_stack_start*
- Function *protocol.notifications.notify_stack_stop*

## runtime.timer – ???

???

### Functions

runtime.timer.**reset**(*???*)
???

runtime.timer.**get**(*???*)
???

### Constants

runtime.timer.**START_TIME** = **0**
???

### Imports

- Module *utime – time related functions*

## runtime.virtualmachine – ???

Contains the VirtualMachine class. ???

### VirtualMachine Class

**class** runtime.virtualmachine.**VirtualMachine**(*???*)
???

**Methods**

**register_on_condition**(*???*)
???

**register_on_start**(*???*)
???

**register_callback**(*???*)
???

**register_on_gesture**(*???*)
???

**register_on_button**(*???*)
???

**register_on_broadcast**(*???*)
???

**reset_time**(*???*)
???

**shutdown**(*???*)
???

**reset_timer**(*???*)
???

**stop_stacks**(*???*)
???

**schedule_coroutine**(*???*)
???

**broadcast**(*???*)
???

**check_all_conditions**(*???*)
Generator function. ???

**get_time**(*???*)
???

**start**(*???*)
???

### Imports

- Module *hub – hub brick functionality*
- Module *runtime.timer – ???*
- Class *protocol.ujsonrpc.JSONRPC*
- Class *runtime.dirty_dict.DirtyDict*
- Class *runtime.stack.Stack*
- Class *runtime.vm_store.VMStore*
- Class *system.System*
- Function *util.time.get_time*
- Function *util.time.reset_time*
- Constant *util.constants.PORTS* = {'C': Port(C), 'B': Port(B), 'D': Port(D), 'E': Port(E), 'A': Port(A), 'F': Port(F)}

## runtime.vm_store – ???

???

### Functions

runtime.vm_store.**add_prop**(*???*)
???

runtime.vm_store.**add_port_prop**(*???*)
???

### Constants

runtime.vm_store.**_STOP = 1**
???

runtime.vm_store.**_STALL = True**
???

runtime.vm_store.**_PCALIB = 17.5**
???

runtime.vm_store.**_LOC = Billund**
???

runtime.vm_store.**_PAIR = ('A', 'B')**
???

runtime.vm_store._STAT = 0
???

runtime.vm_store._ACCEL = (None, None)
???

### VMStore Class

class runtime.vm_store.VMStore(*???*)
???

**Methods**

move_speed(*???*)
Closure function. ???

move_last_status(*???*)
Closure function. ???

move_stop(*???*)
Closure function. ???

move_calibration(*???*)
Closure function. ???

move_acceleration(*???*)
Closure function. ???

move_pair(*???*)
Closure function. ???

motor_acceleration(*???*)
Closure function. ???

motor_stall(*???*)
Closure function. ???

motor_last_status(*???*)
Closure function. ???

motor_speed(*???*)
Closure function. ???

motor_stop(*???*)
Closure function. ???

music_tempo(*???*)
Closure function. ???

music_instrument(*???*)
Closure function. ???

sound_pitch(*???*)
Closure function. ???

sound_volume(*???*)
Closure function. ???

sound_pan(*???*)
Closure function. ???

**weather_location**(*???*)
Closure function. ???

**weather_offset**(*???*)
Closure function. ???

**display_brightness**(*???*)
Closure function. ???

#### Imports

- Class *runtime.dirty_dict.DirtyDict*
- Function `micropython.const
- Constant *util.constants.BRAKE* = 1
- Constant *util.constants.SUCCESS* = 0

### 1.4.10 spike – Spike Prime branding of the user API

This module seems to be purely designed as a frontend for the *_api – user API* module, so that Spike Prime documentation can tell its users to import a module with a related name.

See *_api – user API* for the majority of submodules and classes, all of which can also be called from the spike module by using it as a synonym for "_api".

#### Classes

**class** spike.**PrimeHub**
The actual class appears to have no methods or functions of its own, but it seems to be a superclass of *_api.large_technic_hub.LargeTechnicHub* so inherits everything from there.

The API expects most access to the central Technic Hub and its functions to use an instance of this class.

### 1.4.11 system – system module

Module containing a lot of high-level concepts, but no obvious theme beyond that.

Classes *system.callbacks.Callbacks*, *system.display.DisplayWrapper*, *system.move.Movement*, *system.motors.Motors*, *system.sound.SoundWrapper* have aliases in the main namespace for convenience.

#### Constants

system.**system**
Reference to the main System object.

### System Class

**class** system.**System**(*???*)
???

**reset**(*???*)
???

### Submodules

#### system.callbacks – ???

???

The main namespace contains an alias for *system.callbacks.customcallbacks.CustomSensorCallbackManager* for convenience.

### Submodules

#### system.callbacks.customcallbacks – ???

???

### Classes

**class** system.callbacks.customcallbacks.**CustomSensorCallbackManager**(*???*)
???

**Methods**

**static is_less_than**(*???*)
???

**static did_bump**(*???*)
???

**static did_change**(*???*)
???

**until**(*???*)
Generator function. ???

**until_less_than**(*???*)
Generator function. ???

**until_force_bumped**(*???*)
Generator function. ???

**until_changed**(*???*)
Generator function. ???

**wait_until_less_than**(*???*)
???

**wait_until_force_bumped**(*???*)
???

**wait_until_changed**(*???*)
???

**_start_test_task**(*???*)
???

**remove_task**(*???*)
???

**clear_tasks**(*???*)
???

**Variables**

**_active_tasks**
??? Observed value: []

### Imports

- Module *utime – time related functions*
- Function *event_loop.get_event_loop*
- Function *micropython.const*
- Function *util.sensors.get_sensor_value*
- Constant *util.constants.LPF2_FLIPPER_FORCE* = 63

### Classes

**class** system.callbacks.**Callbacks**(*???*)
???

**reset**(*???*)
???

**hard_reset**(*???*)
???

**class** system.callbacks.**ButtonCallbacks**(*???*)
???

**reset**(*???*)
???

**hard_reset**(*???*)
???

**register_rpc_handlers**(*???*)
???

**__getitem__**(*???*)
???

**class** system.callbacks.**PortCallbacks**(*???*)
???

**reset**(*???*)
???

**hard_reset**(*???*)
???

**init_attach**(*???*)
???

**__getitem__**(*???*)
???

class system.callbacks.**CallbackHandler**(*???*)
???

**reset**(*???*)
???

**hard_reset**(*???*)
???

**callback**(*???*)
???

**register**(*???*)
???

**register_single**(*???*)
???

**register_persistent**(*???*)
???

class system.callbacks.**ConnectionCallbacks**(*???*)
???

**__uch**(*???*)
???

**check_state**(*???*)
???

### Imports

- Module *hub – hub brick functionality*
- Function *protocol.notifications.notify_button_event*
- Function *util.schedule.mp_schedule*
- Constant *util.constants.BT_VCP* = BT_VCP(0) <Bluetooth connection object>
- Constant *util.constants.USB_VCP* = USB_VCP(0) <USB connection object>
- Constant *util.error_handler.error_handler* = <Main ErrorHandler object>

## system.motors – ???

???

### Constants

system.motors.**_PORT_TO_IDX = ['A', 'B', 'C', 'D', 'E', 'F']**
A list to help mapping hub port names to index values.

### Motors Class

**class** system.motors.**Motors**(*???*)
???

**Methods**

**register_port_callback_handlers**(*???*)
???

**on_port**(*???*)
???

**is_motor**(*???*)
???

**_update**(*???*)
???

**Variables**

**wrappers**
??? Observed value: {}

### Imports

- Module *hub – hub brick functionality*
- Class *system.motorwrapper.MotorWrapper*
- Constant *util.constants.PORTS* = {'C': Port(C), 'B': Port(B), 'D': Port(D), 'E': Port(E), 'A': Port(A), 'F': Port(F)}

## system.motorwrapper – ???

???

### Functions

system.motorwrapper.**_shortest**(*???*)
???

system.motorwrapper.**_calc_degrees**(*???*)
???

system.motorwrapper.**_clockwise**(*???*)
???

system.motorwrapper.**_counterclockwise**(*???*)
???

### MotorWrapper Class

**class** system.motorwrapper.**MotorWrapper**(*???*)
???

**Methods**

**__init__**(*???*)
Closure function. ???

**run_at_speed**(*???*)
???

**run_at_speed_async**(*???*)
Generator function. ???

**run_for_degrees**(*???*)
???

**run_for_degrees_async**(*???*)
Generator function. ???

**run_to_position**(*???*)
???

**run_to_position_async**(*???*)
Generator function. ???

**run_to_relative_position**(*???*)
???

**run_to_relative_position_async**(*???*)
Generator function. ???

**run_for_time**(*???*)
???

**run_for_time_async**(*???*)
Generator function. ???

**pwm**(*???*)
???

**stop**(*???*)
???

**brake**(*???*)
???

**hold**(*???*)
???

**get**(*???*)
???

**preset**(*???*)
???

**float**(*???*)
???

**Variables**

**motor**
??? Observed value: None

### Imports

- Class *system.abstractwrapper.AbstractWrapper*
- Function *micropython.const*
- Constant *util.constants.SUCCESS* = 0
- Constant *util.constants.FLOAT* = 0
- Constant *util.constants.BRAKE* = 1
- Constant *util.constants.HOLD* = 2

## system.sound – ???

???

### SoundWrapper Class

**class** system.sound.**SoundWrapper**(*???*)
???

**Methods**

**__init__**(*???*)
Closure function. ???

**beep**(*???*)
???

**beep_async**(*???*)
Generator function. ???

**play**(*???*)
???

**play_async**(*???*)
Generator function. ???

### Imports

- Module *hub – hub brick functionality*
- Class *system.abstractwrapper.AbstractWrapper*
- Function *util.scratch.note_to_frequency*

## system.move – ???

???

### Movement Class

**class** system.move.**Movement**(*???*)
???

**Methods**

**on_pair**(*???*)
???

**Variables**

**_pairs**
??? Observed value: {}

### Imports

- Class *system.movewrapper.MoveWrapper*
- Constant *util.constants.PORTS* = {'C': Port(C), 'B': Port(B), 'D': Port(D), 'E': Port(E), 'A': Port(A), 'F': Port(F)}

## system.movewrapper – ???

???

### Functions

system.movewrapper.**from_steering**(*???*)
???

### MoveWrapper Class

class system.movewrapper.**MoveWrapper**(*???*)
???

**Methods**

**__init__**(*???*)
Closure function. ???

**move_for_time**(*???*)
???

**move_for_time_async**(*???*)
Generator function. ???

**move_differential_speed**(*???*)
???

**move_differential_speed_async**(*???*)
Generator function. ???

**move_at_power**(*???*)
???

**start_at_speeds**(*???*)
???

**start_at_powers**(*???*)
???

**stop**(*???*)
???

**brake**(*???*)
???

**float**(*???*)
???

**hold**(*???*)
???

**_direction_to_steering**(*???*)
???

**from_direction**(*???*)
???

**from_steering**(*???*)
???

**is_valid**(*???*)
???

**unpair**(*???*)
???

**Variables**

**pair**
??? Observed value: None

### Imports

- Class *system.abstractwrapper.AbstractWrapper*
- Constant *util.constants.SUCCESS* = 0
- Constant *util.constants.FLOAT* = 0
- Constant *util.constants.BRAKE* = 1
- Constant *util.constants.HOLD* = 2

## system.abstractwrapper – ???

???

### AbstractWrapper Class

**class** system.abstractwrapper.**AbstractWrapper**(*???*)

???

**Methods**

**await_callback**(*???*)

Generator function. ???

**_callback**(*???*)

???

**_register**(*???*)

???

**cancel**(*???*)

???

### Imports

- Function *event_loop.get_event_loop*
- Function *micropython.const*
- Constant *util.constants.SUCCESS* = 0
- Constant *util.constants.INTERRUPTED* = 1

## system.display – ???

???

### Functions

system.display.**sanitize**(*???*)
???

### DisplayWrapper Class

**class** system.display.**DisplayWrapper**(*???*)
???

**Methods**

**__init__**(*???*)
Closure function. ???

**show**(*???*)
???

**show_async**(*???*)
Generator function. ???

**write**(*???*)
???

**write_async**(*???*)
Generator function. ???

**clear**(*???*)
???

**pixel**(*???*)
???

### Imports

- Module *hub – hub brick functionality*
- Class *system.abstractwrapper.AbstractWrapper*
- Constant *util.constants.SUCCESS* = 0

### Imports

- Module *hub – hub brick functionality*
- Function *event_loop.get_event_loop*

### 1.4.12 ui.hubui – menu system

Runs the menu system that you see when you boot up the Hub (and between running programs). The entire functionality is contained in submodule ui.hubui, although the main module also aliases the ui.hubui.HubUI class as ui.HubUI.

#### Functions

ui.hubui.**reset**(*???*)
???

ui.hubui.**user_interaction**(*???*)
???

#### Variables

ui.hubui.**_latest_activity**
???

#### Constants

ui.hubui.**INACTIVE_SHUTDOWN_BT_MS = 1200000**
???

ui.hubui.**INACTIVE_SHUTDOWN_MS = 300000**
???

```
ui.hubui.DEFAULT_IMAGE = (Image('09090:99999:99999:09990:00900:'),
Image('09000:09900:09990:09900:09000:'))
```
???

```
ui.hubui.SLOTS_IMAGE = (Image('09990:09090:09090:09090:09990:'),
Image('00900:09900:00900:00900:09990:'),  Image('09990:00090:09990:09000:09990:'),
Image('09990:00090:09990:00090:09990:'),  Image('09090:09090:09990:00090:00090:'),
Image('09990:09000:09990:00090:09990:'),  Image('09990:09000:09990:09090:09990:'),
Image('09990:00090:00900:09000:09000:'),  Image('09990:09090:09990:09090:09990:'),
Image('09990:09090:09990:00090:09990:'),  Image('90999:90909:90909:90909:90999:'),
Image('09009:99099:09009:09009:09009:'),  Image('90999:90009:90999:90900:90999:'),
Image('90999:90009:90999:90009:90999:'),  Image('90909:90909:90999:90009:90009:'),
Image('90999:90900:90999:90009:90999:'),  Image('90999:90900:90999:90909:90999:'),
Image('90999:90009:90090:90900:90900:'),  Image('90999:90909:90999:90909:90999:'),
Image('90999:90909:90999:90009:90999:'))
```
???

#### Class HubUI

**class** ui.hubui.**HubUI**(*???*)
???

**Methods**

**__bt_disconnect**(*???*)
???

**__toggle_program**(*???*)
???

**__change_slot**(*???*)
Closure function. ???

**_program_start**(*???*)
Generator function. ???

**stop_all**(*???*)
Closure function. ???

**change_execution_mode**(*???*)
Closure function. ???

**start_program**(*???*)
Closure function. ???

**__cancel_animations**(*???*)
???

**__start_autoshutdown**(*???*)
???

**__on_center_button**(*???*)
Closure function. ???

**_program_stop**(*???*)
Generator function. ???

**__get_slot_image**(*???*)
???

**__on_connect_button**(*???*)
Closure function. ???

**__bt_connect**(*???*)
???

**will_stop_restart**(*???*)
???

**__shutdown_timer**(*???*)
???

**on_connection**(*???*)
Closure function. ???

*Properties*

**property idle**
???

### Imports

- Module *hub – hub brick functionality*
- Module *utime – time related functions*
- Class *ProgramRunner*
- Class *Sounds*
- Class *system.System*
- Function *event_loop.get_event_loop*

- Function *micropython.const*
- Function *util.animations.bootup_animation*
- Generator function *util.animations.bt_animation*
- Function *util.animations.download_animation*
- Generator function *util.animations.led_fade_to*
- Function *util.animations.shutdown_animation*
- Function *util.animations.streaming_animation*
- Generator function *util.animations.shift_in_from_bottom*
- Generator function *util.animations.shift_out_to_bottom*
- Function *util.storage.get_used_slots*
- Constant *util.color.DIM_WHITE* = (135, 25, 10)
- Constant *util.color.WHITE* = (255, 70, 35)

### 1.4.13 `util` – misc utility module

Contains various miscellaneous utility sub-modules.

#### Submodules

#### `util.resetter` – resetting utility module

???

#### Functions

util.resetter.**wait_until_ready_after_restart**(*???*)
???

#### Variables

util.resetter.**_STARTED_AT**
??? Observed value: 3380

#### RTTimer Class

**class** util.resetter.**RTTimer**(*???*)
???

**__repl_reset**(*???*)
???

**repl_reset**(*???*)
???

**start**(*???*)
???

### Imports

- Module *hub – hub brick functionality*
- Function *micropython.schedule*
- Function *utime.sleep_ms*
- Function *utime.ticks_diff*
- Function *utime.ticks_ms*

## util.color – color utility module

???

### Functions

util.color.**color_percentage**(*???*)
???

util.color.**get_color_percentage**(*???*)
???

util.color.**rgb_percentage**(*???*)
???

util.color.**get_rgb_percentage**(*???*)
???

### Constants

util.color.**BLACK = (0, 0, 0)**

util.color.**BLUE = (0, 0, 80)**

util.color.**AZURE = (0, 57, 57)**

util.color.**GREEN = (0, 195, 0)**

util.color.**DIM_WHITE = (135, 25, 10)**

util.color.**RED = (255, 0, 0)**

util.color.**VIOLET = (255, 8, 23)**

util.color.**YELLOW = (255, 35, 0)**

util.color.**WHITE = (255, 70, 35)**
Convenience constants for various colors in (R,G,B)-tuple format.

### util.animations – animation utility module

???

#### Functions

util.animations.**shift_left**(*???*)
???

util.animations.**shift_right**(*???*)
???

util.animations.**shift_out_to_top**(*???*)
Generator function. ???

util.animations.**shift_in_from_right**(*???*)
Generator function. ???

util.animations.**shift_in_from_bottom_left**(*???*)
Generator function. ???

util.animations.**shift_out_to_bottom**(*???*)
Generator function. ???

util.animations.**shift_out_to_left**(*???*)
Generator function. ???

util.animations.**shift_in_from_left**(*???*)
Generator function. ???

util.animations.**shift_in_from_bottom**(*???*)
Generator function. ???

util.animations.**shift_out_to_right**(*???*)
Generator function. ???

util.animations.**shift_in_from_top_right**(*???*)
Generator function. ???

util.animations.**shift_in_from_top**(*???*)
Generator function. ???

util.animations.**streaming_animation**(*???*)
???

util.animations.**download_animation**(*???*)
???

util.animations.**bootup_animation**(*???*)
???

util.animations.**shutdown_animation**(*???*)
???

util.animations.**bt_animation**(*???*)
Generator function. ???

util.animations.**led_fade_to**(*???*)
Generator function. ???

util.animations.**led_fade_in_out**(*???*)
Generator function. ???

util.animations.**chain_animations**(*???*)
Generator function. ???

### Constants

util.animations.**DISPLAY_WIDTH = 5**

util.animations.**DISPLAY_HEIGHT = 5**
Constants for the display dimensions.

util.animations.**BOOTUP_FRAMES = (Image('00000:00000:09000:00000:00000:'),**
**Image('00000:00000:07000:00000:00000:'), Image('00000:00000:07000:00009:00000:'),**
**Image('00000:00000:07000:00007:00000:'), Image('00000:00000:07000:90007:00000:'),**
**Image('00000:00000:07000:70007:00000:'), Image('00000:90000:07000:70007:00000:'),**
**Image('00000:70000:07000:70007:00000:'), Image('00000:70000:07000:70007:00900:'),**
**Image('00000:70000:07000:70007:00700:'), Image('00000:70900:07000:70007:00700:'),**
**Image('00090:70800:07000:70007:00700:'), Image('00080:70800:07000:79007:00700:'),**
**Image('00080:70700:07090:78007:00700:'), Image('00070:70700:07080:78007:90700:'),**
**Image('09070:70700:07070:77007:80700:'), Image('08070:70700:07070:77007:80709:'),**
**Image('08079:70700:07070:77007:70708:'), Image('07078:70700:07070:77907:70708:'),**
**Image('07078:79700:07070:77707:70707:'), Image('07077:78700:07079:77707:70707:'),**
**Image('07077:78700:07078:77707:79707:'), Image('07977:78700:07078:77707:78707:'),**
**Image('07877:77700:07078:77797:78707:'), Image('07877:77709:07077:77787:78707:'),**
**Image('07877:77708:97077:77787:77707:'), Image('07777:77708:87077:77787:77797:'),**
**Image('07777:77798:87077:77777:77787:'), Image('97777:77787:87077:77777:77787:'),**
**Image('87777:77787:87977:77777:77787:'), Image('99999:99999:99999:99999:99999:'),**
**Image('77777:77777:77777:77777:77777:'), Image('66669:66669:66669:66669:66669:'),**
**Image('55599:55595:55595:55595:55599:'), Image('44999:44949:44949:44949:44999:'),**
**Image('39993:39393:39393:39393:39993:'), Image('09990:09090:09090:09090:09990:'))**
The set of image objects that are displayed on the bootup animation.

util.animations.**SHUTDOWN_FRAMES = (Image('99999:90009:90009:90009:99999:'),**
**Image('55555:57775:57075:57775:55555:'), Image('00000:09990:09090:09990:00000:'),**
**Image('00000:05550:05750:05550:00000:'), Image('00000:00000:00900:00000:00000:'),**
**Image('00000:00000:00500:00000:00000:'), Image('00000:00000:00000:00000:00000:'))**
The set of image objects that are displayed on the shutdown animation.

### Imports

- Module *hub – hub brick functionality*
- Module *utime – time related functions*
- Class *util.constants.Image*
- Function *util.color.color_percentage*
- Function *util.color.get_color_percentage*
- Constant *util.color.BLACK* = (0, 0, 0)
- Constant *util.color.DIM_WHITE* = (135, 25, 10)

### util.motor – motor utility module

???

#### Functions

util.motor.**clamp_speed**(*???*)
???

util.motor.**clamp_power**(*???*)
???

util.motor.**dir_to_speed**(*???*)
???

### util.scratch – scratch utility module

???

#### Functions

util.scratch.**to_number**(*???*)
???

util.scratch.**to_boolean**(*???*)
???

util.scratch.**orientation_to_number**(*???*)
???

util.scratch.**number_to_orientation**(*???*)
???

util.scratch.**percent_to_int**(*???*)
???

util.scratch.**percent_to_frequency**(*???*)
???

util.scratch.**color_to_number**(*???*)
???

util.scratch.**number_to_color**(*???*)
???

number_color_to_rgb – <function number_color_to_rgb at 0x200213e0>

util.scratch.**note_to_frequency**(*???*)
???

util.scratch.**pitch_to_freq**(*???*)
???

util.scratch.**sanitize_ports**(*???*)
???

util.scratch.**sanitize_movement_ports**(*???*)
???

util.scratch.**clamp**(*???*)
???

util.scratch.**wrap_clamp**(*???*)
???

util.scratch.**partition_image_str**(*???*)
???

util.scratch.**convert_image**(*???*)
???

util.scratch.**convert_animation_frame**(*???*)
???

util.scratch.**convert_brightness**(*???*)
???

util.scratch.**adjust_brightness**(*???*)
???

util.scratch.**compare**(*???*)
???

util.scratch.**tan**(*???*)
???

util.scratch.**is_int**(*???*)
???

util.scratch.**get_variable**(*???*)
???

### Constants

util.scratch.**PAIR_REGEX** = <regex object>
???

util.scratch.**ORIENTATIONS** = ('', 'front', 'back', 'up', 'down', 'rightside', 'leftside')
???

### Imports

- Module *math – mathematical functions*
- Module *ure – simple regular expressions*
- Module *util.color – color utility module*
- Constant *util.constants.NO_KEY* = -1
- Constant *util.constants.NUMBER* = 0
- Constant *util.constants.BOOLEAN* = 2
- Constant *util.constants.VAR_DEFAULTS* = {0: 0, 1: '', 2: False}

### util.storage – storage utility module

???

#### Functions

util.storage.**get_path**(*???*)
???

util.storage.**get_storage_information**(*???*)
???

util.storage.**generate_project_id**(*???*)
???

util.storage.**read_local_name**(*???*)
???

util.storage.**write_local_name**(*???*)
???

util.storage.**_ensure_folder_exists**(*???*)
???

util.storage.**_get_metadata**(*???*)
???

util.storage.**_set_metadata**(*???*)
???

util.storage.**get_used_slots**(*???*)
???

util.storage.**clear_slot**(*???*)
???

util.storage.**move_slot**(*???*)
???

util.storage.**_move_slot_lookup**(*???*)
???

util.storage.**_file_to_slotid**(*???*)
???

util.storage.**get_program_type**(*???*)
???

util.storage.**get_program_project_id**(*???*)
???

util.storage.**open_program**(*???*)
???

util.storage.**read_program**(*???*)
???

util.storage.**close_program**(*???*)
???

util.storage.**set_force_reset**(*???*)
???

util.storage.**pop_force_reset**(*???*)
???

### Constants

util.storage.**_BT_PREFIX = LEGO Hub@**
???

util.storage.**__FORCE_RESET_PATH__ = ./reset**
???

util.storage.**__STORAGE_PATH__ = ./projects**
???

util.storage.**__META_PATH__ = ./projects/.slots**
???

util.storage.**__PROGRAM_PATH__ = ./projects/{0}**
???

util.storage.**__PROGRAM_PATH_EXT__ = ./projects/{0}.py**
???

util.storage.**PROGRAM_TYPE_PYTHON = python**
???

util.storage.**PROGRAM_TYPE_SCRATCH = scratch**
???

### Imports

- Module *uos – basic "operating system" services*
- Module *urandom – random number generation*
- Module *ure – simple regular expressions*
- Constant uerrno.ENOENT = 2
- Constant uerrno.EEXIST = 17
- Constant *util.constants.LOCAL_NAME* = /local_name.txt

## util.sensors – sensors utility module

???

### Functions

util.sensors.**register_ports**(*???*)
???

util.sensors.**is_type**(*???*)
???

util.sensors.**_is_motor**(*???*)
???

util.sensors.**_type_change_handler**(*???*)
???

util.sensors.**get_sensor_value**(*???*)
???

util.sensors.**update_sensor_data**(*???*)
???

util.sensors.**update_battery_status**(*???*)
???

util.sensors.**reset_to_default_mode**(*???*)
???

util.sensors.**set_display_sync**(*???*)
???

util.sensors.**current_motion**(*???*)
???

### Variables

util.sensors.**battery_status**
??? Observed value: [8.36, 100, True]

util.sensors.**sensor_data**
??? Observed value: [[0, ()], [0, ()], [0, ()], [0, ()], [0, ()], [0, ()], (0, -805, 585), (3, 3, 0), (-3, 0, 54), '', 0]

### Constants

util.sensors.**_PORTS = [Port(A), Port(B), Port(C), Port(D), Port(E), Port(F)]**
List of the six port objects of the six ports on the Hub. See *hub.Port*.

util.sensors.**_REVERSE_MODES = {48: [3, 0, 1, 2], 65: [3, 0, 1, 2], 49: [3, 0, 1, 2], 75: [3, 0, 1, 2], 76: [3, 0, 1, 2], 61: [1, 0, 2, 3, 4], 62: [0], 63: [0, 1, -1, -1, 2]}**
???

util.sensors.**_EVENT_MODE = [[], [], [], [], [], []]**
???

util.sensors.**_PORT_INDEX_MAP = ['A', 'B', 'C', 'D', 'E', 'F', 'ACCELEROMETER', 'GYROSCOPE', 'POSITION', 'ORIENTATION', 'TIMER']**
???

util.sensors.**_PORT_TYPE = [0, 0, 0, 0, 0, 0]**
???

util.sensors.**_NO_DATA = ()**
???

util.sensors.**_SYNC_DISPLAY = False**
???

util.sensors.**_DEFAULT_MODE = {48: [(1, 0), (2, 2), (3, 1), (0, 0)], 65: [(1, 0), (2, 2), (3, 1), (1, 0)], 49: [(1, 0), (2, 2), (3, 1), (0, 0)], 75: [(1, 0), (2, 2), (3, 1), (0, 0)], 76: [(1, 0), (2, 2), (3, 1), (0, 0)], 61: [(1, 0), (0, 0), (5, 0), (5, 1), (5, 2)], 62: [(0, 0)], 63: [(0, 0), (1, 0), (4, 0)]}**
???

util.sensors.**_MOTOR_TYPES = [65, 48, 49, 75, 76]**
List of the LPF2 type IDs that correspond to motors.

### Imports

- Module *hub – hub brick functionality*
- Function *micropython.const*
- Function *util.scratch.orientation_to_number*
- Function *util.time.get_time*
- Constant *util.constants.LPF2_FLIPPER_MOTOR_MEDIUM* = 48
- Constant *util.constants.LPF2_FLIPPER_MOTOR_LARGE* = 49
- Constant *util.constants.LPF2_FLIPPER_COLOR* = 61
- Constant *util.constants.LPF2_FLIPPER_DISTANCE* = 62
- Constant *util.constants.LPF2_FLIPPER_FORCE* = 63
- Constant *util.constants.LPF2_FLIPPER_MOTOR_SMALL* = 65
- Constant *util.constants.LPF2_STONE_GREY_MOTOR_MEDIUM* = 75
- Constant *util.constants.LPF2_STONE_GREY_MOTOR_LARGE* = 76

## util.time – time utility module

???

### Functions

util.time.**reset_time**(*???*)
???

util.time.**get_time**(*???*)
???

util.time.**start_time**(*???*)
???

util.time.**stop_time**(*???*)
???

### Variables

util.time.**_STOPPED_AT**
???? Observed value: 0

util.time.**_STARTED_AT**
???? Observed value: 1680

util.time.**_RUNNING**
???? Observed value: False

### Imports

- Function *utime.ticks_diff*
- Function *utime.ticks_ms*

## util.error_handler – error handling utility module

???

### Constants

util.error_handler.**error_handler = <Main ErrorHandler object>**

util.error_handler.**PROGRAM_EXECUTION_ERROR = 0**
???

util.error_handler.**PROGRAM_EXECUTION_MEMORY_ERROR = 1**
???

### ErrorHandler Class

**class** util.error_handler.**ErrorHandler**(*???*)
???

**handle_user_program_error**(*???*)
???

**handle_notify_error**(*???*)
???

**handle_runtime_error**(*???*)
???

**initialize**(*???*)
???

**_emit_runtime_error**(*???*)
???

**_handle_error**(*???*)
???

### Imports

- Module *hub – hub brick functionality*
- Module *protocol.notifications – ???*
- Module *sys – system specific functions*
- Module *uio – input/output streams*
- Module *ure – simple regular expressions*
- Module *version – version module*
- Function *event_loop.get_event_loop*
- Function *micropython.const*
- Function *ubinascii.b2a_base64*
- Function *util.log.log_critical_error*
- Constant *util.color.BLACK* = (0, 0, 0)
- Constant *util.color.RED* = (255, 0, 0)

## util.log – log utility module

???

### Functions

util.log.**log_to_file**(*???*)
???

util.log.**log_critical_error**(*???*)
???

util.log.**clear_log**(*???*)
???

util.log.**_write_to_log**(*???*)
???

util.log.**timed_function**(*???*)
???

util.log.**cat_log**(*???*)
???

### Constants

util.log.**_LOG_FILE = ./runtime.log**
Location of the log file for logging.

### Variables

util.log.**timed_fn_buffer**
??? Observed value: []

### Imports

- Module *gc – control the garbage collector*
- Module *sys – system specific functions*
- Module *uio – input/output streams*
- Module *uos – basic "operating system" services*
- Module *utime – time related functions*

## util.schedule – scheduling utility module

???

### Functions

util.schedule.**mp_schedule**(*???*)
???

### Imports

- Module *micropython – access and control MicroPython internals*

## util.print_override – remote printing module

???

### Functions

util.print_override.**spikeprint**(*???*)
???

### Constants

util.print_override._NOT_CONNECTED_ERROR = The programming app is not connected to the hub.
Error message for when there's no connection to the app (computer or phone).

### Imports

- Module *Builtin functions and exceptions*
- Module *uio – input/output streams*
- Class *protocol.ujsonrpc.JSONRPC*
- Function *ubinascii.b2a_base64*
- Function *utime.ticks_diff*
- Function *utime.ticks_ms*
- Constant *util.constants.BT_VCP* = BT_VCP(0)
- Constant *util.constants.USB_VCP* = USB_VCP(0)

## util.constants – constants module

???

### Constants

util.constants.LPF2_FLIPPER_MOTOR_MEDIUM = 48

util.constants.LPF2_FLIPPER_MOTOR_LARGE = 49

util.constants.LPF2_ACCELERATION = 57

util.constants.LPF2_GYRO = 58

util.constants.LPF2_ORIENTATION = 59

util.constants.LPF2_FLIPPER_COLOR = 61

util.constants.LPF2_FLIPPER_DISTANCE = 62

util.constants.LPF2_FLIPPER_FORCE = 63

util.constants.LPF2_FLIPPER_MOTOR_SMALL = 65

util.constants.LPF2_STONE_GREY_MOTOR_MEDIUM = 75

util.constants.LPF2_STONE_GREY_MOTOR_LARGE = 76
Constants to represent various types of input device. They correspond to official PoweredUp/SpikePrime Type IDs - see for example https://github.com/pybricks/technical-info/blob/master/assigned-numbers.md

util.constants.MOTOR_TYPES = (65, 48, 49, 75, 76)
A tuple specifying which of the above types are motors.

util.constants.**DEFAULT_IMAGE = (Image('09090:99999:99999:09990:00900:'), Image('09000:09900:09990:09900:09000:'))**

Two Image objects that are the default initial display screens on the Spike Prime and the RI5 firmwares respectively.

util.constants.**SLOTS_IMAGE = (Image('09990:09090:09090:09090:09990:'), Image('00900:09900:00900:00900:09990:'), Image('09990:00090:09990:09000:09990:'), Image('09990:00090:09990:00090:09990:'), Image('09090:09090:09990:00090:00090:'), Image('09990:09000:09990:00090:09990:'), Image('09990:09000:09990:09090:09990:'), Image('09990:00090:00900:09000:09000:'), Image('09990:09090:09990:09090:09990:'), Image('09990:09090:09990:00090:09990:'), Image('90999:90909:90909:90909:90999:'), Image('09009:99099:09009:09009:09009:'), Image('90999:90009:90999:90900:90999:'), Image('90999:90009:90999:90009:90999:'), Image('90909:90909:90999:90009:90009:'), Image('90999:90900:90999:90009:90999:'), Image('90999:90900:90999:90909:90999:'), Image('90999:90009:90090:90900:90900:'), Image('90999:90909:90999:90909:90999:'), Image('90999:90909:90999:90009:90999:'))**

Image objects shown in the menu system when navigating between slots (they are the numbers 0 - 19).

util.constants.**PORTS = {'C': Port(C), 'B': Port(B), 'D': Port(D), 'E': Port(E), 'A': Port(A), 'F': Port(F)}**

Dictionary mapping port names to the corresponding Port objects (see *hub.Port*).

util.constants.**FLOAT = 0**

util.constants.**BRAKE = 1**

util.constants.**HOLD = 2**

Modes of operation for stopping a motor. FLOAT simply removes power and allows coasting, while BRAKE reverses power to stop the motor as soon as possible, and HOLD mode will deliberately try to return to the braked point if it is moved away from it.

util.constants.**USB_VCP = USB_VCP(0)**

util.constants.**BT_VCP = BT_VCP(0)**

Aliases for the main USB and Bluetooth objects on the Hub. See *hub.USB_VCP* and *hub.BT_VCP*.

util.constants.**NO_KEY = -1**

util.constants.**NUMBER = 0**

util.constants.**STRING = 1**

util.constants.**BOOLEAN = 2**

util.constants.**VAR_DEFAULTS = {0: 0, 1: '', 2: False}**

Imported by *util.scratch – scratch utility module*. Seems to be representing basic scratch data types numerically, with a dictionary to look up their default values.

util.constants.**TIMER_PACE_LOW = 48**

util.constants.**TIMER_PACE_HIGH = 16**

Imported by *programrunner – run user programs* and *hub_runtime – Hub main module*. ???

util.constants.**INACTIVE_SHUTDOWN_MS = 300000**

util.constants.**INACTIVE_SHUTDOWN_BT_MS = 1200000**

Imported by *ui.hubui*. Presumably they represent the length of inactive time before the system shuts down. Perhaps when running alone, and when connected to bluetooth?

util.constants.**LONG_PRESS_MS = 3000**

Not obviously used anywhere. ???

util.constants.**SUCCESS = 0**

util.constants.**INTERRUPTED = 1**
Seems to represent return codes of some function somewhere. Success code is imported in various places. ???

util.constants.**STALLED = 2**
Not obviously used anywhere. May belong to the previous group? ???

util.constants.**NO_STATUS = -1**
Imported by various methods submodules in the *commands – commands module* module. ???

util.constants.**DATA_DIR = /data**

util.constants.**LINEGRAPH_DIR = /data/linegraph**

util.constants.**LOCAL_NAME = /local_name.txt**
Important things in the local filesystem. ???

### Sounds Class

**class** util.constants.**Sounds**(*???*)
???

**Constants**

Looks like filesystem locations of sounds associated with certain system operations.

**NAVIGATION = sounds/menu_click**
???

**NAVIGATION_FAST = sounds/menu_fastback**
???

**STARTUP = sounds/startup**
???

**SHUTDOWN = sounds/menu_shutdown**
???

**PROGRAM_STOP = sounds/menu_program_stop**
???

**PROGRAM_START = sounds/menu_program_start**
???

### Image Class

**class** util.constants.**Image**(*???*)
??? I'm not quite clear whether this class principally lives here or in *hub – hub brick functionality*...

**Methods**

**width**(*???*)
???

**height**(*???*)
???

**get_pixel**(*???*)
???

**set_pixel**(*???*)
???

**shift_left**(*???*)
???

**shift_right**(*???*)
???

**shift_up**(*???*)
???

**shift_down**(*???*)
???

**Constants**

**HEART = Image('09090:99999:99999:09990:00900:')**

**HEART_SMALL = Image('00000:09090:09990:00900:00000:')**

**HAPPY = Image('00000:09090:00000:90009:09990:')**

**SMILE = Image('00000:00000:00000:90009:09990:')**

**SAD = Image('00000:09090:00000:09990:90009:')**

**CONFUSED = Image('00000:09090:00000:09090:90909:')**

**ANGRY = Image('90009:09090:00000:99999:90909:')**

**ASLEEP = Image('00000:99099:00000:09990:00000:')**

**SURPRISED = Image('09090:00000:00900:09090:00900:')**

**SILLY = Image('90009:00000:99999:00909:00999:')**

**FABULOUS = Image('99999:99099:00000:09090:09990:')**

**MEH = Image('09090:00000:00090:00900:09000:')**

**YES = Image('00000:00009:00090:90900:09000:')**

**NO = Image('90009:09090:00900:09090:90009:')**

**CLOCK12 = Image('00900:00900:00900:00000:00000:')**

**CLOCK1 = Image('00090:00090:00900:00000:00000:')**

**CLOCK2 = Image('00000:00099:00900:00000:00000:')**

**CLOCK3 = Image('00000:00000:00999:00000:00000:')**

**CLOCK4 = Image('00000:00000:00900:00099:00000:')**

**CLOCK5 = Image('00000:00000:00900:00090:00090:')**

**CLOCK6 = Image('00000:00000:00900:00900:00900:')**

**CLOCK7 = Image('00000:00000:00900:09000:09000:')**

**CLOCK8 = Image('00000:00000:00900:99000:00000:')**

**CLOCK9 = Image('00000:00000:99900:00000:00000:')**

**CLOCK10 = Image('00000:99000:00900:00000:00000:')**

**CLOCK11 = Image('09000:09000:00900:00000:00000:')**

**ARROW_N = Image('00900:09990:90909:00900:00900:')**

**ARROW_NE = Image('00999:00099:00909:09000:90000:')**

```
ARROW_E = Image('00900:00090:99999:00090:00900:')
ARROW_SE = Image('90000:09000:00909:00099:00999:')
ARROW_S = Image('00900:00900:90909:09990:00900:')
ARROW_SW = Image('00009:00090:90900:99000:99900:')
ARROW_W = Image('00900:09000:99999:09000:00900:')
ARROW_NW = Image('99900:99000:90900:00090:00009:')
GO_RIGHT = Image('09000:09900:09990:09900:09000:')
GO_LEFT = Image('00090:00990:09990:00990:00090:')
GO_UP = Image('00000:00900:09990:99999:00000:')
GO_DOWN = Image('00000:99999:09990:00900:00000:')
TRIANGLE = Image('00000:00900:09090:99999:00000:')
TRIANGLE_LEFT = Image('90000:99000:90900:90090:99999:')
CHESSBOARD = Image('09090:90909:09090:90909:09090:')
DIAMOND = Image('00900:09090:90009:09090:00900:')
DIAMOND_SMALL = Image('00000:00900:09090:00900:00000:')
SQUARE = Image('99999:90009:90009:90009:99999:')
SQUARE_SMALL = Image('00000:09990:09090:09990:00000:')
RABBIT = Image('90900:90900:99990:99090:99990:')
COW = Image('90009:90009:99999:09990:00900:')
MUSIC_CROTCHET = Image('00900:00900:00900:99900:99900:')
MUSIC_QUAVER = Image('00900:00990:00909:99900:99900:')
MUSIC_QUAVERS = Image('09999:09009:09009:99099:99099:')
PITCHFORK = Image('90909:90909:99999:00900:00900:')
XMAS = Image('00900:09990:00900:09990:99999:')
PACMAN = Image('09999:99090:99900:99990:09999:')
TARGET = Image('00900:09990:99099:09990:00900:')
TSHIRT = Image('99099:99999:09990:09990:09990:')
ROLLERSKATE = Image('00099:00099:99999:99999:09090:')
DUCK = Image('09900:99900:09999:09990:00000:')
HOUSE = Image('00900:09990:99999:09990:09090:')
TORTOISE = Image('00000:09990:99999:09090:00000:')
BUTTERFLY = Image('99099:99999:00900:99999:99099:')
STICKFIGURE = Image('00900:99999:00900:09090:90009:')
GHOST = Image('99999:90909:99999:99999:90909:')
SWORD = Image('00900:00900:00900:09990:00900:')
GIRAFFE = Image('99000:09000:09000:09990:09090:')
```

```
SKULL = Image('09990:90909:99999:09990:09990:')
```

```
UMBRELLA = Image('09990:99999:00900:90900:09900:')
```

```
SNAKE = Image('99000:99099:09090:09990:00000:')
```

These are all Image objects containing the pictures suggested by their names.

```
ALL_CLOCKS = (Image('00900:00900:00900:00000:00000:'),
Image('00090:00090:00900:00000:00000:'),  Image('00000:00099:00900:00000:00000:'),
Image('00000:00000:00999:00000:00000:'),  Image('00000:00000:00900:00099:00000:'),
Image('00000:00000:00900:00090:00090:'),  Image('00000:00000:00900:00900:00900:'),
Image('00000:00000:00900:09000:09000:'),  Image('00000:00000:00900:99000:00000:'),
Image('00000:00000:99900:00000:00000:'),  Image('00000:99000:00900:00000:00000:'),
Image('09000:09000:00900:00000:00000:'))
```

```
ALL_ARROWS = (Image('00900:09990:90909:00900:00900:'),
Image('00999:00099:00909:09000:90000:'),  Image('00900:00090:99999:00090:00900:'),
Image('90000:09000:00909:00099:00999:'),  Image('00900:00900:90909:09990:00900:'),
Image('00009:00090:90900:99000:99900:'),  Image('00900:09000:99999:09000:00900:'),
Image('99900:99000:90900:00090:00009:'))
```

A couple of tuples of sets of images you might want to iterate though.

### Imports

- Module *hub – hub brick functionality*
- Function *micropython.const*

## util.rotation – rotation utility module

???

### Functions

util.rotation.**dir_to_rotation**(*???*)

???

util.rotation.**rotate_hub_display**(*???*)

???

util.rotation.**rotate_hub_display_to_value**(*???*)

???

util.rotation.**rotate_hub_display_to_orientation**(*???*)

???

### Variables

util.rotation.**_CURRENT_ROTATION**
??? Observed value: 0

### Imports

- Module *hub – hub brick functionality*

## 1.4.14 hub_runtime – Hub main module

Seems to be the main module on the Hub, in charge of startup and error handling.

### Functions

hub_runtime.**__connection_changed**(*???*)
???

hub_runtime.**init**(*???*)
???

hub_runtime.**start**(*???*)
???

### Imports

- Module *hub – hub brick functionality*
- Module *runtime – runtime module*
- Module *util.scratch – scratch utility module*
- Module *util.sensors – sensors utility module*
- Class commands.LinegraphMonitorMethods
- Class commands.SoundMethods
- Class *programrunner.ProgramRunner*
- Class *ui.hubui.HubUI*
- Class commands.LightMethods
- Class commands.ProgramMethods
- Class commands.MotorMethods
- Class commands.HubMethods
- Class protocol.RPCProtocol
- Class commands.WaitMethods
- Class *util.resetter.RTTimer*
- Class *class Timer – control hardware timers*
- Class commands.MoveMethods

- Function *event_loop.get_event_loop*
- Function *protocol.notifications.notify_gesture_event*
- Function *util.sensors.register_ports*
- Function *util.storage.pop_force_reset*
- Constant *system.system* = <Main System object>
- Constant *util.constants.BT_VCP* = BT_VCP(0) <Bluetooth connection object>
- Constant *util.constants.USB_VCP* = USB_VCP(0) <USB connection object>
- Constant *util.constants.TIMER_PACE_HIGH* = 16
- Constant *util.error_handler.error_handler* = <Main ErrorHandler object>

### 1.4.15 `version` – version module

Just contains the version!

#### Constants

**`__version__ = 2.1.4-mindstorms.13`**

File main.py is also found in the filesystem, but do not import it as it will restart the hub and require a battery removal/reinsert to get the hub working again! You can technically "import" boot, projects, sounds, extra_files, but by default they're empty of Python content so they do nothing.

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 cpython_diffs 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):
```

(continues on next page)

(continued from previous page)

```
    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**



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 nonvolatile "disk" (flash memory) storage is limited. This tutorial offers ways to make the most of the limited resources. Because MicroPython runs on controllers based on a variety of architectures, the methods presented are generic: in some cases it will be necessary to obtain detailed information from platform specific documentation.

### 2.5.1 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

(The Technic Hub uses an ARM Cortex-M4 processor which uses Thumb2 instructions, so this section of the Micropython docs has been left in, but it's left as an exercise for the reader how to actually get started doing assembler programming on the Hub!)

This document assumes some familiarity with assembly language programming and should be read after studying a Thumb2 tutorial like the Pyboard Assembler Tutorial in the main Micropython docs. 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)
```

(continues on next page)

(continued from previous page)

```
    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

# 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.

## 3.1 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.

### 3.1.1 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.

### 3.1.2 Basic Example

This simple module named `example` provides a single function `example.add_ints(a, b)` which adds the two integer args together and returns the result.

Directory:

```
example/
├── example.c
└── micropython.mk
```

example.c

```
// Include required definitions first.
#include "py/obj.h"
#include "py/runtime.h"
#include "py/builtin.h"

// This is the function which will be called from Python as example.add_ints(a, b).
STATIC mp_obj_t example_add_ints(mp_obj_t a_obj, mp_obj_t b_obj) {
    // Extract the ints from the micropython input objects
    int a = mp_obj_get_int(a_obj);
    int b = mp_obj_get_int(b_obj);

    // Calculate the addition and convert to MicroPython object.
    return mp_obj_new_int(a + b);
}
// Define a Python reference to the function above
STATIC MP_DEFINE_CONST_FUN_OBJ_2(example_add_ints_obj, example_add_ints);

// Define all properties of the example module.
// Table entries are key/value pairs of the attribute name (a string)
// and the MicroPython object reference.
// All identifiers and strings are written as MP_QSTR_xxx and will be
// optimized to word-sized integers by the build system (interned strings).
STATIC const mp_rom_map_elem_t example_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_example) },
    { MP_ROM_QSTR(MP_QSTR_add_ints), MP_ROM_PTR(&example_add_ints_obj) },
};
STATIC MP_DEFINE_CONST_DICT(example_module_globals, example_module_globals_table);

// Define module object.
const mp_obj_module_t example_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&example_module_globals,
};

// Register the module to make it available in Python
MP_REGISTER_MODULE(MP_QSTR_example, example_user_cmodule, MODULE_EXAMPLE_ENABLED);
```

micropython.mk

```
EXAMPLE_MOD_DIR := $(USERMOD_DIR)

```

(continues on next page)

(continued from previous page)

```
# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(EXAMPLE_MOD_DIR)/example.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(EXAMPLE_MOD_DIR)
```

Finally you will need to define `MODULE_EXAMPLE_ENABLED` to 1. This can be done by adding `CFLAGS_EXTRA=-DMODULE_EXAMPLE_ENABLED=1` to the `make` command, or editing `mpconfigport.h` or `mpconfigboard.h` to add

```
#define MODULE_EXAMPLE_ENABLED (1)
```

Note that the exact method depends on the port as they have different structures. If not done correctly it will compile but importing will fail to find the module.

### 3.1.3 Compiling the cmodule into MicroPython

To build such a module, compile MicroPython (see getting started) with an extra `make` flag named `USER_C_MODULES` set to the directory containing all modules you want included (not to the module itself). For example:

Directory:

```
my_project/
├── modules/
│   └──example/
│       ├──example.c
│       └──micropython.mk
└── micropython/
    ├──ports/
   ... ├──stm32/
       ...
```

Building for stm32 port:

```
cd my_project/micropython/ports/stm32
make USER_C_MODULES=../../../modules CFLAGS_EXTRA=-DMODULE_EXAMPLE_ENABLED=1 all
```

### 3.1.4 Module usage in MicroPython

Once built into your copy of MicroPython, the module implemented in `example.c` above can now be accessed in Python just like any other builtin module, eg

```
import example
print(example.add_ints(1, 3))
# should display 4
```

CHAPTER

FOUR

# MICROPYTHON LICENSE INFORMATION

# PYTHON MODULE INDEX

## V

# INDEX

## Symbols

## A



## B

## C

## D



## E

## F



## G

## H

## I

## J



## K



## L

## M

## N

## O



## P

## Q



## R

## S

## T



## U

## V



## W

## Y



## Z