3. REQUIREMENTS

3.1. General

The following conventions are used throughout this document.

Each requirement has a unique identification in the context of the MicroPython project, as defined by [AD02]. This identification respects the following template:

<subsystem id> - <requirement category id> - <sequential number>

In the present document, there shall be one unique subsystem id: "MPVM", for MicroPython VM.

The following requirement categories are present:

- FC – Functional Requirement
- IF – Interface requirement
- INIT – Initialisation requirement
- OBSE – Observability requirement
- PERF – Performance requirement
- FDIR – FDIR requirement
- DVP – Development (design/test/…) requirement

The following verification methods can be applicable to the requirements:

- T – Test
- R – Review of design
- A – Analysis
- I – Inspection

The comments to the requirements are given in italic.

The MicroPython VM requirements are tracing specifications from ESA's statement of work [RD01]. An additional effort is put on the OBCP use case, as MicroPython VM requirements are also covering and tracing a subset[1] of ECSS-E-ST-70-01C requirements [AD01], as tailored by ESA's statement of work [RD01].

[1] Other ECSS-E-ST-70-01C requirements are covered in the scope of the *OBCP engine*, the software component that controls the µPython VM on-board.

3.2. Functional requirements

MPVM-FC-010	OBCP-51i, OBCP-51j, OBCP-522a, OBCP-522b, OBCP-524a, OBCP-524b, OBCP-524c, OBCP-524d, OBCP-525a, OBCP-526a, OBCP-526b, OBCP-526c, OBCP-526h, OBCP-527a1, OBCP-527a2, OBCP-527a3, OBCP-527b, OBCP-5210a, OBCP-531e, OBCP-5447a, OBCP-624a	T	
The syntax and semantic of MicroPython is defined as a subset of Python 3.4, with some restrictions.			
The language subset and restrictions are elaborated in the following requirements. Note that the standard library of Python 3.4 is mostly excluded from the present requirements; the small subset of modules that are covered, at least partially, are specified, in other requirements (see below).			

MPVM-FC-020	OBCP-523a, OBCP-523c	T	
MicroPython shall support the following built-in types, as defined in Python 3.4: `bool int float str object super type`			
The mapping from Python types to ECSS-E-ST-70-31C' s simple types is as follows: `bool` *Boolean* `int` *Integer, Unsigned Integer, bit string, octet string* `float` *Real* *These can be used also as functions for type-casting constructs.* *The "value type data types" can be made using Python inheritance, with* `object` *and* `super` *types.* *Note that, compared to ECSS-E-ST-70-31C, engineering units are not covered and syntax may differ.* *Note also that the following simple types are not covered:* *- the types for packet fields identified in [AD01] (i.e. the PUS),* *- absolute and relative time (Python module datetime),* *- enumerated types and enumerated set (Python module enum).*			

MPVM-FC-030	OBCP-523b, OBCP-526b	T	
MicroPython shall support the following built-in types, as defined in Python 3.4: `bytearray bytes tuple dict list`			
Note that a record (see [AD01]) can be represented by the `dict` type (dictionary), which is the approach taken for user-defined classes. Note however that unlike usual records, dictionary can add/remove fields at runtime; the same applies for classes since the `__slots__` class attribute is not supported. This peculiarity can be avoided by using `namedtuple` to represent records : the `namedtuple` type is closest to the concept of record because it defines fixed fields with fixed names (see MPVM-FC-110).			

MPVM-FC-040		T	
MicroPython shall support the following built-in types, as defined in Python 3.4: `filter memoryview set range reversed zip enumerate map`			
Note that the afore-mentioned types are not required by [AD01]. Note also that the following types present in [RD02] have been left out: `property, complex, frozenset`.			

MPVM-FC-050		T	

MicroPython shall support the following built-in functions, as defined in Python 3.4:
`abs(x) -- absolute value`
`all(iterable) -- True if all values in iterable are True`
`any(iterable) -- True if any values in iterable are True`
`bin(x) -- format number in binary representation`
`chr(x) -- character corresponding to value x`
`dir([obj]) -- list of attributes in object`
`divmod(x, y) -- a tuple with division and modulus`
`globals() -- dictionary of global objects`
`hasattr(obj, attr) -- check for an attribute`
`hash(obj) -- hash value of an object`
`hex(x) -- format number in hexadecimal representation`
`id(obj) -- unique number corresponding to object`
`isinstance(obj, cls) -- True if obj is instance of cls`
`iter(obj) -- create an iterator from the given object`
`len(iterable) -- number of elements in iterable`
`max(x, ...) -- maximum of all arguments`
`min(x, ...) -- minimum of all arguments`
`next(iterator) -- get next item from iterator`
`oct(x) -- format number in octal representation`
`pow(x, y[, mod]) -- power of arguments`
`print(...) -- print all arguments`
`round(x[, digits]) -- round a float`
`sorted(iterable[, key][, reverse]) -- sort an iterable`			

```
sum(iterable[, start]) -- sum all values in an iterable
```

Note that the following functions present in [RD02] have been left out: `callable, getattr, attr, setattr, repr, issubclass, ord, locals`

MPVM-FC-060	OBCP-5447a	T	

MicroPython shall support the following built-in exception class hierarchy, as defined in Python 3.4:

```
BaseException
  GeneratorExit
  KeyboardInterrupt
  SystemExit
  Exception
    ArithmeticError
      OverflowError
      ZeroDivisionError
    AssertionError
    AttributeError
    EOFError
    ImportError
    LookupError
      IndexError
      KeyError
    MemoryError
    NameError
    OSError
    RuntimeError
      NotImplementedError
    StopIteration
    SyntaxError
      IndentationError
    TypeError
    ValueError
      UnicodeError
```

MPVM-FC-070		T	

MicroPython shall support the following classes, as defined in Python 3.4:

```
classmethod
staticmethod
```

Note that the following class present in [RD02] have been left out: `Ellipsis`

MPVM-FC-080	OBCP-533a2, OBCP-5441b3	T	

MicroPython shall support the following functions from the module `gc`, as defined in Python 3.4:
`gc.collect()`
`gc.disable()`
`gc.enable()`
`gc.isenabled()`
`gc.mem_free()`
`gc.mem_alloc()`			

MPVM-FC-090	OBCP-51g	T	
MicroPython shall support the following function from the module `sys`, as defined in Python 3.4: `sys.exit([code])` If `code` argument is missing, then the default value `0` shall be taken. Upon completion with `sys.exit`, a SystemExit exception shall be raised and the C function starting the VM execution shall return the given `code` value bitwise ORed with 128.			
Note that the following constants present in [RD02] have been left out: `sys.byteorder`, `sys.implementation`, `sys.maxsize`, `sys.platform`, `sys.version`, `sys.version_info`. The rationale is that the dependencies with the target platform are already taken into account by the MicroPython cross-compiler (hence, not by the MicroPython source script).			

MPVM-FC-100	OBCP-526b	T	
MicroPython shall support the following function from the module `array`, as defined in Python 3.4: `array.array(typecode[, initialiser])`			

MPVM-FC-110	OBCP-523b	T	
MicroPython shall support the following functions from the module `ucollections`, as defined in Python 3.4 under the name `collections`: `ucollections.namedtuple`			
Note that `namedtuple` is the Python type closer to the concept of "record" (see comments in			

MPVM-FC-030). The `OrderedDict` function present in [RD02] has been left out.

MPVM-FC-120	OBCP-526a	T	

MicroPython shall support the following functions from the module `math`, as defined in Python 3.4:
`e acos(x) frexp(x)`
`pi asin(x) ldexp(x, i)`
`sqrt(x) atan(x) modf(x)`
`pow(x, y) atan2(y, x) isfinite(x)`
`exp(x) ceil(x) isinf(x)`
`log(x[, base]) copysign(x, y) isnan(x)`
`cos(x) fabs(x) trunc(x)`
`sin(x) floor(x) radians(x)`
`tan(x) fmod(x, y) degrees(x)`			

MPVM-FC-130	OBCP-523b, OBCP-5291a, OBCP-5291b	T	

MicroPython shall support the following functions from the module `struct`, as defined in Python 3.4:
`calcsize(fmt)`
`pack(fmt, ...)`
`pack_into(fmt, buf, offset, ...)`
`unpack(fmt, buf)`
`unpack_from(fmt, buf, offset)`			

MPVM-FC-140		T	

MicroPython shall support the following functions from the module `binascii`, as defined in Python 3.4:
`hexlify(data)`
`unhexlify(hexstr)`
`a2b_base64(string)`
`b2a_base64(data)`			

This module is used to convert between binary data and various ASCII encodings.			

MPVM-FC-150	OBCP-5351a, OBCP-5441b2, OBCP-5441b3	T	

MicroPython shall provide the module `micropython`, with the following functions:

`micropython.stack_use()`
Return the current amount of C stack that is used.

`micropython.pystack_use()`
Return the current amount of Python stack that is used. This function is available only if the VM has been compiled in a configuration which uses a separate stack for Python function calls.

`micropython.heap_lock()`
Lock the heap so that memory cannot be allocated. Any allocation that is attempted after calling this function will raise a MemoryError exception.

`micropython.heap_unlock()`
Unlock the heap.

The function `heap_lock()` should be called after initialization so that heap use is totally deterministic (i.e. does not change). The `mem_info()` and `qstr_info()` functions present in [RD02] have been left out.

MPVM-FC-160	OBCP-51e, OBCP-51f, OBCP-526c, OBCP-526g, OBCP-526i, OBCP-526j, OBCP-528a1, OBCP-528a2, OBCP-528a3, OBCP-528a4, OBCP-528a5, OBCP-528b, OBCP-528c, OBCP-528d, OBCP-5292a, OBCP-5292b, OBCP-5292c, OBCP-5292d, OBCP-5292e, OBCP-5292g, OBCP-5292h, OBCP-5292i, OBCP-5292j, OBCP-5292k, OBCP-5292l, OBCP-5423a3	R	

MicroPython shall allow for using custom modules developed in C.

Dedicated MicroPython modules can then be developed to access, in a safe way, the services and resources of OS and target platform.

MPVM-FC-170	OBCP-51g, OBCP-51h, OBCP-	T	

	5423c, OBCP-5423e		
The MicroPython VM shall provide a C function to start execution of MicroPython bytecode at a given address in memory. When the bytecode execution ends, the function shall return the return code (0 if no error code is specified). The MicroPython VM shall not modify or remove the bytecode.			

MPVM-FC-180	OBCP-5441b4, OBCP-5443d	T	
If the MicroPython VM cannot allocate required memory during execution, it shall stop and it shall raise a specific exception.			

MPVM-FC-190	OBCP-5446a , OBCP-5447a, OBCP-5447b, OBCP-5447d, OBCP-5447e	T	
If the MicroPython VM catches an exception during execution (uncaught by the MicroPython bytecode itself), it shall stop, return a specific error code and provide a means to retrieve the line number in the MicroPython source code where the exception was raised.			
The Python execution model handles any error occurring in as an exception. Such exception can be caught and treated by the MicroPython procedure itself. If not, the exception is propagated to the VM, which shall then halt.			

MPVM-FC-200	OBCP-51c1, OBCP-51c2, OBCP-51c3, OBCP-51c4, OBCP-51c5, OBCP-51d, OBCP-51g, OBCP-51h, OBCP-5210a, OBCP-528b, OBCP-528c	T	
The MicroPython language shall support object-orientation, as defined in Python 3.4.			
The object-orientation allows using the" template method" design pattern to enforce a common structure and execution logic. The idea is to define once for all an abstract class with methods for the different bodies (e.g. precondition, main, confirmation, contingency); these methods have a default empty implementation; the actual class inherits the abstract one and overloads the required methods.			

MPVM-FC-300		T	

A MicroPython cross-compiler shall be provided. It shall generate a bytecode file, having `.mpy` extension, from a given MicroPython script that is syntactically valid.
The words "syntactically valid" mean that the script conforms to the grammar of MicroPython. As per the Python language itself, there is no further static check, like availability of referred names (variable, function, method, class, module, etc). *The MicroPython cross-compiler is not formally qualified; it is not part of the flight code and it is not required to comply with CAT-B software.*

MPVM-FC-310	OBCP-543b	T	
The SDE shall provide a tool to calculate for a given bytecode, the ISO checksum, as defined in Annex A.2 of [RD03]. The tool shall take a `.mpy` file as input and produce a `.bin` file with the content of the `.mpy` file appended with the ISO checksum (2 bytes).			

MPVM-FC-320	OBCP-532a1	T	
If the given script has an invalid syntax, the MicroPython cross-compiler shall report an error, indicating the first invalid line. No bytecode file shall be generated in such case.			

MPVM-FC-330		T	
After being loaded in memory, the byte code generated by the MicroPython cross-compiler shall be executable by the MicroPython VM, with the semantic specified by the MicroPython source script.			
See MPVM-FC-190			

MPVM-FC-500	OBCP-526g, OBCP-526j, OBCP-531e, OBCP-532a2	T	
The MicroPython SDE shall provide a preprocessing tool called "Import Expander". The Import Expander shall take a given MicroPython script in input and it shall generate a semantically equivalent MicroPython script by replacing import statements by the content of the corresponding MicroPython source files, provided that these modules are not C extensions. This process shall be recursive, in case the imported modules themselves contain import statements.			
One of the goal of the Import Expander is to be able to produce one single bytecode for one given			

MicroPython procedure, whatever the imported modules in this procedure. This is meant to simplify the transport and loading of bytecode in the MicroPython VM.

Note that this preprocessing is formally required because the loading and registering of module bytecode in the VM is not in the scope of the present SRS.

MPVM-FC-510	OBCP-526g, OBCP-526j, OBCP-531e, OBCP-532a2	T	

The Import Expander shall be able to treat both flavours of MicroPython import:

```
from m1 import …
```

or

```
import m2, m3
```

If the `as` keyword is found, in any of these two flavours, then the tool shall report an error telling that it cannot handle such case.

For the first flavour, in order to limit complexity, it shall be accepted to treat

```
from m1 import a, b
```

like

```
from m1 import *
```

Notes:

- Import using package notation are not treated (e.g. `import m1.m2.m3` is not expanded).

- The use of asterisk may cause unwanted variable rebinding in case of name collision. This can be limited by using the smart expansion feature described in MPVM-FC-520. Independently of the tool described here, the use of the `from import` flavour is generally not recommended in Python scripts, for the sake of maintainability.

- If advanced features of Python are used (e.g. introspection, locals/globals dictionaries, etc), side-effects may be observed due to the "class embedding" trick.

- In case of uncaught exception (see MPVM-FC-190), the developer shall take care that the reported line number is given in the expanded source code file.

MPVM-FC-512	OBCP-526g, OBCP-526j, OBCP-531e, OBCP-532a2	T	

The Import Expander shall detect when an import statement refers to a module already imported, whether directly or through cascaded imports; in such situation, it shall apply the following rules:

1. If `import m` is found in scope S_2 while `import m` has already been encountered in scope S_1, then
   - if $S_2 = S_1$, then the latest import statement shall be skipped,

- if $S_2 \neq S_1$, then the import statement in S_2 shall be replaced by an assignment that creates an alias so that the objects of `m` can be referred also using S_2 scope.

2. If `from m import ...` is found in scope S_2 while `from m import ...` has already been encountered in scope S_1, then
   - if $S_2 = S_1$, then the latest import statement shall be skipped,
   - if $S_2 \neq S_1$, then the new tool shall report an error telling that it cannot handle such case.
3. If `import m` and `from m import ...` flavours are mixed at different places for the same given module `m` (whatever their scopes), then the tool shall report an error telling that it cannot handle such case.

These rules are meant to mimic as closely as possible the native import mechanism of Python, so to avoid side effects due to static expansion. In particular, a given module should not be expanded twice, wherever it is imported; therefore, an object that is imported at different places cannot be created twice (this is particularly important for mutable object). The most complex cases, which usually reveal bad programming practices, are not supported and reported as such.

MPVM-FC-520	OBCP-526g, OBCP-526j, OBCP-531e, OBCP-532a2, REQ-OBCPE-100	T	

The Import Expander shall be able to do "smart expansion": for a given list of modules, it shall analyse the importing MicroPython procedure and only copy the subset of assignments of variables that are actually used in this procedure.

The rationale is to have a MicroPython module containing all symbols defined in a database, as a set of MicroPython assignments (this should be automated by a dedicated tool). For example, the following script could be produced from the engineering data repository:

```
# sdb.py
camera_1_id = 1
camera_2_id = 2
...
```

A MicroPython procedure can then import this file and use the required subset of variables. Since this imported module is arbitrarily large, it is sensible to require that the Import Expander copies only the required subset of assignments.

This solution assumes that there are no interdependencies between variables in the imported file. For example, the assignment

```
# sdb.py
...
camera_1_controller_id = camera_1_id * 100 + 4
```

would fail if `camera_1_id` is not referred in the importing module. This is why such mechanism is not required by default for all imported modules: only the modules declared by the user are assumed to be simple assignments and can then be expanded without mistakes.

MPVM-FC-530	OBCP-533a1, REQ-OBCPE-100	T	
The Import Expander shall be able to report all the names that have been retained by the "smart expansion".			
This allows for getting the list names of variables imported from a database and actually used by the OBCP procedure.			

MPVM-FC-540	OBCP-533a4	I	
The SDE shall provide a static analysis tool to display violation to coding standard for a given MicroPython procedure and possibly detect some errors.			
Note that some displays the code quality metrics.			

MPVM-FC-550	OBCP-533b	I	
The SDE shall provide a static analysis tool to display the call tree of a given MicroPython procedure.			

MPVM-FC-600	REQ-VM-030	T	

MicroPython shall provide the module `mem`, with the following functions:
 	
`mem.read_u32(addr)`
Return the 32-bit value at the given `addr` address. `addr` must be word aligned.
 	
`mem.write_u32(addr, value)`
Write `value` to the given `addr` address. `addr` must be word aligned. Returns `None`.
 	
`mem.read_buf(addr, buf)`
Read a memory region starting at the address `addr` into the given buffer. `buf` must be a `bytearray` or an `array` and its length specifies the length of the read. The read is done using 32-bit words so `addr` must be word aligned and the length of `buf` must be a multiple of 4. Returns `None`.
 	
`mem.write_buf(addr, buf)`
Write the given buffer to the memory address `addr`. `buf` must be a `bytearray` or an `array` and its length specifies the length of the write. The write is done using 32-bit words so `addr` must be word aligned and the length of `buf` must be a multiple of 4. Returns `None`.			

MPVM-FC-610	REQ-VM-030	T	

MicroPython shall provide the module `rtems`, with the following function:

```
rtems.script_id()
```
Return an integer being the identifier of the VM instance that this script is running in. The first VM has an identifier of 0.

and the following objects (integer constants):

DEFAULT_ATTRIBUTES,
WAIT,
NO_WAIT,
SEARCH_ALL_NODES,
SEARCH_OTHER_NODES,
SEARCH_LOCAL_NODE.

with same meaning as defined in RTEMS.

The handling of errors that may be reported by the underlying RTEMS functions is specified in MPVM-FC-660.

MPVM-FC-620	REQ-VM-030	T	

MicroPython shall provide the module `rtems.queue` for creating and using message queues. It shall provide the following functions:

```
rtems.queue.create(name, count, max_size,
attr=rtems.DEFAULT_ATTRIBUTES)
```
Create a new RTEMS queue and return a queue object that can be used to send and receive messages on the queue. The `name` argument must be a string of four characters long. `count` is the maximum number of messages the queue can hold before getting full. `max_size` is the maximum size of a message, in bytes. The `attr` argument specifies the attributes of the queue and defaults to `rtems.DEFAULT_ATTRIBUTES`.
Corresponding C function: rtems_message_queue_create().
Note: this function uses the heap to create the queue object.

```
rtems.queue.ident(name, node=rtems.SEARCH_ALL_NODES)
```
Access an existing RTEMS queue and returns a queue object corresponding to that queue. The `name` argument must be a string of four characters long. `node` specifies the node to search for the queue on and can be one of `rtems.SEARCH_ALL_NODES`, `rtems.SEARCH_OTHER_NODES` or `rtems.SEARCH_LOCAL_NODE`.
Corresponding C function: rtems_message_queue_ident().
Note: this function uses the heap to create the queue object.

The above functions return a `queue` object which has the following methods to access the underlying RTEMS queue.

`queue.delete()`
Deletes the queue. Returns `None`.
Corresponding C function: rtems message queue delete().

`queue.send(buf, option=rtems.NO_WAIT, timeout=0)`
Send a message on the queue. `buf` must be a `str` object, `bytes` object, `bytearray` or `array`. `options` is one of `rtems.WAIT` or `rtems.NO_WAIT`. If `rtems.WAIT` is specified as the `option`, then the `timeout` argument specifies how long to wait (in ticks), with a negative or zero value meaning to wait forever. The default is to not wait. Returns `None`.
Corresponding C function: rtems message queue send().
Note: the blocking behaviour using `rtems.WAIT` is a MicroPython extension to the RTEMS API. If this option is given then the function `rtems_message_queue_send` will continue to be called while it returns `RTEMS_TOO_MANY` (meaning that the queue is full).

`queue.urgent(buf, option=rtems.NO_WAIT, timeout=0)`
This method has the same arguments and behaviour as `queue.send` except that the message has urgent priority.
Corresponding C function: rtems message queue urgent().

`queue.broadcast(buf)`
Broadcasts a message on the queue. buf must be a `str` object, `bytes` object, `bytearray` or `array`. Returns number of tasks that received the message.

`queue.receive(buf, option=rtems.WAIT, timeout=0)`
Receive a message from the queue and store it in the given buffer. The buffer argument buf must be a `bytearray` or `array`. `options` is one of `rtems.WAIT` or `rtems.NO_WAIT`. If `rtems.WAIT` is specified as the option then the timeout argument specifies how long to wait (in ticks), with a negative or zero value meaning to wait forever. The default is to wait forever. This method returns the number of bytes written into `buf`.
Corresponding C function: rtems message queue receive().
Note: `buf` must have enough room for the message.

`queue.flush()`
This method flushes the message queue, and returns the count.
Corresponding C function: rtems message queue flush().

`queue.get_number_pending()`
This method returns the number of messages pending on the queue.
Corresponding C function: rtems message queue get number pending().

The handling of errors that may be reported by the underlying RTEMS functions is specified in MPVM-FC-660.

MPVM-FC-630	REQ-VM-030	T	

MicroPython shall provide the module `rtems.sem` for creating and accessing RTEMS semaphores. It shall provide the following functions:

`rtems.sem.create(name, count=1, attr=rtems.DEFAULT_ATTRIBUTES, prio=0)`
Create a new RTEMS semaphore and returns a semaphore object that can be used to control concurrent access to a resource. The `name` argument must be a string of four characters long. `count` is the number of units that the initial semaphore holds (defaults to one unit). The `attr` argument specifies the attributes of the semaphore and defaults to `rtems.DEFAULT_ATTRIBUTES`. `prio` is the priority ceiling of the semaphore.
Corresponding C function: rtems semaphore create().
Note: this function uses the heap to create the semaphore object.

`rtems.sem.ident(name, node=rtems.SEARCH_ALL_NODES)`
This function accesses an existing RTEMS semaphore and returns a semaphore object corresponding to that semaphore. The name argument must be a string of four characters long. `node` specifies the node to search for the semaphore on, and can be one of `rtems.SEARCH_ALL_NODES`, `rtems.SEARCH_OTHER_NODES` or `rtems.SEARCH_LOCAL_NODE`.
Corresponding C function: rtems semaphore ident().
Note: this function uses the heap to create the semaphore object.

The above functions return a semaphore object (`sem`) which has the following methods to access the underlying RTEMS semaphore.

`sem.delete()`
Deletes the semaphore. Returns `None`.
Corresponding C function: rtems semaphore delete().

`sem.obtain(option=rtems.WAIT, timeout=0)`
Obtains the semaphore. If `rtems.WAIT` is specified as the option then the `timeout` argument specifies how long to wait (in ticks), with a negative or zero value meaning to wait forever. The default is to wait forever. Returns `None`.
Corresponding C function: rtems semaphore obtain().

`sem.release()`
Releases the semaphore. Returns `None`.
Corresponding C function: rtems semaphore release().

`sem.flush()`
Flushes the semaphore. Returns `None`.
Corresponding C function: rtems semaphore flush().

The handling of errors that may be reported by the underlying RTEMS functions is specified in

MPVM-FC-660.

MPVM-FC-640	REQ-VM-030	T	

MicroPython shall provide the module `rtems.task`, with the following functions:
 	
`task.wake_when((year, month, day, hour, minute, second))`
Take a tuple of 6 integers specifying a date and time. The script will sleep until that point in time. Returns None.
Corresponding C function: rtems_task_wake_when().
 	
`task.wake_after(ticks)`
Take an integer being a number of ticks, and pauses the script for that duration. Returns `None`.
Corresponding C function: rtems_task_wake_after().
Note: it is recommended to use `time.sleep(sec)` instead because that is the standard Python way of pausing execution.
 	
`task.get_note(note_id)`
Take an integer and returns an (unsigned) integer which is the value of the RTEMS notepad with the given identifier (between 0 and 15 inclusive).
Corresponding C function: rtems_task_get_note().
 	
`task.set_note(note_id, note_val)`
Take an integer for the notepad identifier, and an integer value, and stores the latter in the RTEMS notepad. The `note_val` argument is truncated to 32 bits. Returns `None`.
Corresponding C function: rtems_task_set_note().
Note: notepad 0, 1, 2 and 3 are used by the MicroPython VM manager.			

The handling of errors that may be reported by the underlying RTEMS functions is specified in MPVM-FC-660.			

MPVM-FC-650	REQ-VM-030	T	

MicroPython shall provide the module `time`, with the following functions:
 	
`time.localtime()`
Return a 6-tuple containing: `(year, month, mday, hour, minute, second)`. `month` is in the range 1-12 (inclusive), `mday` is in 1-31, `hour` is in 0-23, `minute` is in 0-59 and `second` is in 0-59.
 	
`time.localtime_into(lst)`
Store into the given list the following items: `(year, month, mday, hour, minute, second)`. The range of these values is the same as in the `time.localtime` function.
 	
`time.time()`			

Return the number of seconds since the Epoch, as a float. Has sub-second precision. `time.sleep(seconds)` Sleep for the given number of seconds. `seconds` can be a float to sleep for a fractional number of seconds.
Note: the `time.localtime_into` function shall be used in place of `time.localtime` if the heap is locked (see MPVM-FC-150). *The handling of errors that may be reported by the underlying RTEMS functions is specified in MPVM-FC-660.*

MPVM-FC-660	REQ-VM-030	T	
Each function provided by the `rtems` module and its sub-modules (see above) shall check the returned status code of the underlying RTEMS C functions. If this return code is not `RTEMS_SUCCESSFUL`, then the function shall raise a Python `OSError` exception with a single argument, which is the string corresponding to the `rtems_status_code`.			

3.3. Performance requirements

NA

3.4. Interface requirements

MPVM-IF-010	OBCP-51a, OBCP-531a, OBCP-531b, OBCP-531c, OBCP-531d, OBCP-532a1, OBCP-532a2, OBCP-534a1, OBCP-534a2, OBCP-534a3, OBCP-534a4, OBCP-534a5, OBCP-534a6, OBCP-534a7, OBCP-534a8, OBCP-534a9, OBCP-534b, OBCP-534c, OBCP-624a	I	
A Software Development Environment (SDE) shall be provided for developing MicroPython procedure. This SDE shall provide functions for edition, syntax checking, code completion, execution and debugging. For these functions, the MicroPython script under development shall be assimilated to a Python 3.4 script.			

The SDE should be a standard SDE as PyDev or PyCharm.

Since MicroPython is in essence a subset of Python 3.4, it is expected

- *that the edition, syntax check works as expected,*
- *that the code completion works as expected for all standard Python 3.4 standard modules / builtin functions ported in MicroPython.*

For all C modules developed specifically for MicroPython or for interfacing with an OS or an OBSW, the developer should implement stub modules in MicroPython. Under the assumption that all required modules are stubbed, the SDE can execute/debug the MicroPython script using the host Python 3.4 interpreter. This execution is however not fully representative of an execution on the MicroPython VM because:

- *the C modules are stubbed;*
- *MicroPython VM implementation may differ from Python 3.4 (offline compilation, numerical accuracy,...);*
- *the memory is expected to be more limited in MicroPython VM than in host Python 3.4. The memory management is different also;*
- *imported modules could be have been expanded inline (see 3.2).*

MPVM-IF-020		I	
The MicroPython SDE shall provide the user a simple means to call the MicroPython cross-compiler on any MicroPython procedure to generate the associated bytecode file (extension .mpy). The messages output by the cross-compiler shall be visible in the SDE.			
The error reporting of MicroPython cross-compiler is helpful to check that the written script conforms to MicroPython language, which is stricter than a Python 3.4 (e.g. unavailability of some builtin functions, disabling of complex numbers, etc.).			

MPVM-IF-030		I	
If the SDE is configured to do so, the Import Expander shall be called automatically before calling the cross-compiler (preprocessing of source code). The messages output by the Import Expander shall be visible in the SDE.			
The error reporting of Import Expander is helpful to check that the written script conforms to the specific limitations of "import", which are stricter than a Python 3.4 (e.g. import rules).			

MPVM-IF-040	OBCP-543b	T	
The SDE shall provide a simple means to call the ISO checksum tool in order to produce the `.bin` file for a given `.mpy` file.			

MPVM-IF-050	REQ-VM-070	I	
The SDE shall provide a simple means to load and execute a given bytecode file on a given LEON2 target.			

MPVM-IF-060	REQ-VM-070	I	
The VM test suite shall be automated.			
The objective is to be able to re-run the VM test suite easily on a different target.			

3.5. Operational requirements

NA

3.6. Resources requirements

MPVM-PERF-010		I	
The MicroPython VM executable code shall be designed to have a minimal size, which shall not exceed 200 kbytes.			
The MicroPython VM executable can be stored on RAM or, preferably, on ROM. Memory efficiency of the MicroPython VM is important because it is a limited resource in a spacecraft and because it is used also by OBSW. *Possible solutions can be found in [RD01] (chapter 4).*			

MPVM-PERF-020		R	
The MicroPython cross-compiler shall be designed to produce a bytecode with a minimal size.			
This requirement impacts also the design and performance of MicroPython VM. *Possible solutions can be found in [RD01] (chapter 4).*			

MPVM-PERF-030		R	

The MicroPython VM shall be designed to execute the bytecode using a good trade-off between memory consumption and execution time. For this choice, the minimisation of memory consumption should be prioritized.
See rationale in MPVM-PERF-010. *Possible solutions can be found in [RD01] (chapter 4).*

MPVM-PERF-040		I	
The MicroPython cross-compiler source code and/or compile chain shall contain configuration parameters to account for constraints of specific hardware target (e.g. ROM / RAM size) or for selecting a specific trade-off for a given target.			
See rationale in MPVM-PERF-010.			

MPVM-PERF-050		I	
The MicroPython VM source code and/or compile chain shall contain configuration parameters to account for constraints of specific hardware target (e.g. ROM / RAM size) or for selecting a specific trade-off for a given target.			
See rationale in MPVM-PERF-010.			

3.7. Design requirements and implementation constraints

MPVM-DVP-010	REQ-VM-060	I	
The MicroPython environment is made up of a SDE (development of MicroPython script), a cross-compiler (production of bytecode from MicroPython script), a VM (execution of the bytecode).			

MPVM-DVP-020		I	
In the C source code of MicroPython cross-compiler and VM, any language feature out of scope of the requirements defined in the present document shall be disabled by C preprocessor directives.			
The goal is here to avoid branching in the configuration of MicroPython. The rationale is to be able to integrate bug fixes and evolutions made on the development trunk.			

MPVM-DVP-030	REQ-OBCPE-110	I	
The MicroPython VM and compile chain shall support two configurations for MicroPython object representation: 1. 32 bits model – floating-point numbers require heap allocation, 2. 64 bits model "NaN Boxing" – floating numbers do not require heap allocation.			
Both configurations are subject to the qualification. See description and rationale in section 5.5.1 of [RD01].			

MPVM-DVP-040		I	
The MicroPython VM and compile chain shall allow building MicroPython VM without support of `mem`, `time` and `rtems` modules/submodules (see MPVM-FC-600, MPVM-FC-610, MPVM-FC-620, MPVM-FC-600, MPVM-FC-630, MPVM-FC-640, MPVM-FC-650, MPVM-FC-660).			
This option is not part of the qualification, which covers a configuration with all the afore-mentioned modules included.			

3.8. Security and privacy requirements

NA

3.9. Portability requirements

MPVM-DVP-100	REQ-VM-060	T	
The MicroPython VM shall run on LEON 2, on top of Edisoft RTEMS 4.8.			
The development is meant to be compatible also with Edisoft RTEMS 4.10 and 4.11, as well as LEON 3. This portability is however out of scope of the present SRS.			

MPVM-DVP-110		I	
The MicroPython SDE, including the Import Expander, shall run on a Unix system compatible with Python 3.4.			

MPVM-DVP-120		I	
The MicroPython VM build toolchain, the MicroPython cross-compiler and its build toolchain shall run on a Unix system, with the following software components installed: Python 3.4, gcc, cross-compiler for SPARC V8 and Edisoft RTEMS 4.8.			
The chosen Unix version should then be compatible with the afore-mentioned software components.			

3.10. Software quality requirements

MPVM-DVP-200	REQ-VM-030, REQ-VM-050	I	
The MicroPython VM shall be qualified with requirements of ECSS CAT-B, as defined in [AD02] and [AD03].			

3.11. Software reliability requirements

NA

3.12. Software maintainability requirements

NA

3.13. Software safety requirements

MPVM-DVP-310	OBCP-5446b, OBCP-5446d	T	
The MicroPython VM shall raise an exception in case it is no more possible to allocate memory on the heap.			

MPVM-DVP-320	OBCP-5446b, OBCP-5446d	T	
The MicroPython VM shall raise an exception in case it is no more possible to allocate memory on the stack.			

3.14. Software configuration and delivery requirements

MPVM-DVP-410	REQ-VM-030, REQ-VM-040	I	
The MicroPython VM source, configuration, tool chain and test bench files shall be maintained in configuration control.			

3.15. Data definition and database requirements

NA

3.16. Human factors related requirements

NA

3.17. Adaptation and installation requirements

NA