MicroPython for the DXM Controller

Instruction Manual

Original Instructions
b_51151351 Rev. A
21 September 2021

Contents

1 MicroPython for the DXM Controllers

MicroPython is a compact version of the Python 3.4 language standard.

The DXM Controller can be programmed using action rules, read/write maps, a ScriptBasic program, or a MicroPython program.

This short tutorial describes how to use the DXM Configuration Software to upload a MicroPython script that uses the built-in modules and the DXM Custom MicroPython Module, 'pyb.'

Use any text editor tool to create a file. A good text editor that numbers lines and color codes the language syntax makes code easier to read and shows typing or copy/paste mistakes in the code. Examples include Visual Studio Code and PyCharm, but any editor that support syntax highlighting for Python is sufficient.

After the MicroPython program is created, save it with the extension **`.py`**. The filename needs to be **`filename.py`** for the DXM Controller to recognize the file as a MicroPython script.

For developers familiar with ScriptBasic scripts on the DXM Controller, the translation to MicroPython should be straightforward. The custom functions in ScriptBasic are: GETREG, SETREG, MULTISET, MULTIGET, FILEOUT, FILEIN, MBREGOUT, MBREGIN. The use of lists in MicroPython eliminated the need for MBREGOUT and MBREGIN, but all other custom functions listed are supported. The parameters for each function were kept in the same order in MicroPython to avoid confusion. See the DXM ScriptBasic Instruction Manual (p/n 191745).

For more information on MicroPython, go to docs.micropython.org. Many of the code examples in this document are from docs.micropython.org.

1.1 Upload the MicroPython Program to the DXM Controller

Use the DXM Configuration Software to upload the MicroPython file (*.py) to the DXM Controller before uploading the XML configuration file. Define which MicroPython program to run, save the XML file, then load the XML configuration file onto the DXM Controller.

Note that the DXM Controller runs only one script (either MicroPython or ScriptBasic) after it boots up.

Follow these steps to upload the MicroPython program using the DXM Configuration Software. Uploading a MicroPython program requires v4.1.11 or newer of the DXM Configuration Software and version 3.4.0 or newer of the DXM Controller firmware.

1. Launch the DXM Configuration Software.
2. Connect to the DXM.
   - To use serial, select the **COMM port**.
   - To use TCP/IP, enter the IP address. The IP of the DXM Controller can be found in the device's LCD menu system under the **System Info** > **Ethernet** screen.
3. Click **Connect**.
4. To access a saved XML configuration file, go to **File** > **Open** and select that file.
5. On the **Settings** > **Scripting** screen, click **Upload File** to upload the MicroPython program (*.py).
6. Click on the file name and then click **Add Selected to Startup**.
   This stores the startup script name in the XML configuration file.
7. Go to **File** > **Save** to save the XML configuration file.
8. Go to **DXM** > **Send Configuration to DXM** to upload the XML file to the device.
   The DXM automatically reboots.

After the DXM reboots, the program outputs to the USB (default console port). Use a serial terminal with the baud rate 115200. Any syntax errors are visible in the terminal. Any print statements also print in the terminal.

1.2 MicroPython Modules

MicroPython modules allow programmatic access to the DXM's application programmer interface.

1.2.1 pyb.api

Access some features of the DXM API. See parameters table for numerical values.

Command	pyb.api(cmd,val)
Returns	Command API response in string format, or "INVALID PA" on error.
Return type	String

Table 1: Parameters for pyb.api(cmd:int)

Command	Value	Definition
Register push	6	Invoke cellular register push
Clear HTTP Log	8	Clear HTTP log - no return
Get Default IP	16	returns string value of IP address
Get Default Gateway	18	Returns string value of gateway
Get Modbus Server SLID	28	returns string value of Modbus server ID
Get RTC Value	102	returns string value of current time
Get Firmware version	103	returns string value of firmware version
Get MAC Address	112	Returns string value of device MAC address
Get Model Number	113	Returns string value of device model number
Get Serial Number	114	Returns string value of device serial number
Reboot	200	Restarts the device
Update Cell Firmware	212	Cellular FOTA - Message to the cellular modem to look for updated firmware. Forced to look in FTP.SENSONIX.NET for delta file of new version. Running this command stops all operations on the controller and resets the controller after 5 minutes.

Example: Get RTC Value

```
import pyb
RTC = 102
current_time = pyb.api(RTC)
print(current_time)
```

Example: GET IP Address

```
import pyb
IP_addr = 16
IP_address = pyb.api(IP_addr)
print(IP_address)
```

1.2.2 File Operation Commands

Command	Description	Example
pyb.filein(fileindex, maxlen)	Read bytes from UART, file, TCP socket, or UDP socket. Returns a content read as a string.	
pyb.fileout(fileindex, length, flags, content, content2)	Write a string to a serial port, email, TCP Socket, UDP socket, or file. Returns an integer 0 on success or an integer code on failure.	
pyb.filepeek(fileindex)	Reads the number of bytes waiting in a network socket, the number of bytes in the serial port read buffer, or the size of a file on disc. Returns the size of file in bytes, unless otherwise noted.	

content

The content to write (string)

content2

Optional string; only used for email and SMS; SMS is not supported on the DXM700 models

fileindex

Values include:

Table 2: Parameters for the fileindex(int) command

Target	Value	Description
UART (generally serial console)	1	Number of bytes in serial out buffer
Email	2	
SMS	3	
TCP Server	4	Number of bytes in TCP server buffer
TCP Client	5	Number of bytes in TCP client buffer
UDP socket	6	Number of bytes in UDP buffer
File 1	10	
File 2	11	
File 3	12	
File 4	13	
File 5	14	

flags

Defines the file operation (integer)

- 0 - append file
- 1 - Overwrite the file, if it exists

length

Length of content to write (integer), input 0 for auto-detect

maxlen

Returns the maximum number of characters read (integer); default value 200. If the number of desired characters is greater than 200, use this command to specify.

Examples for pout

Task	Command	Result
Write data to the RS-232 UART	pyb.fileout(fileindex, length, flags, content)	Result = pyb.fileout(1,0,0,"This is sent through UART")
Send an email (the optional filename is specified by the file's file index)	pyb.fileout(fileindex, optionalFilename, 0, emailAdrs, message)	Result = pyb.fileout(2,10,0,"myEmail@host.com","Sending File")
Write data to Ethernet	pyb.fileout(fileindex, length, 0, content)	Result = pyb.fileout(4,0,0,"This is sent through Ethernet")
Append data to SbFile1.dat	pyb.fileout(fileindex, length, flags, content)	Result = pyb.fileout(10,0,0,"This is put into the file")

Example Code

```
import pyb
#Get the contents of File 1 and put in string file_content
File1 = 10
file_len_bytes = pyb.filepeek(File1)
#If file_len_bytes not specified (0 is used), 200 bytes will be returned
file_content = pyb.filein(File1,file_len_bytes)
```

1.2.3 Register Access Commands

Command	Description
pyb.getreg(reg, sid, mbtype)	Returns a register value.
pyb.multiget(reg, count, sid, mbtype)	Returns a list of register values. The list will be of length count.
pyb.multiset(reg, values, sid, mbtype)	Returns 0 on success or an error code on failure.
pyb.setreg(reg, value, sid, mbtype)	Returns 0 on success or an error code on failure. Even on invalid register set (for example, non-writeable registers).

The data type of the register (integer or floating point) is determined by the register address and Modbus Slave ID (sid). For example, if reading from internal floating-point registers (register IDs 1001 through 5000, inclusive) from SID 199, the result is a float. Otherwise the result is a 16-bit integer. Refer to the DXM instruction Manual for Modbus Register data types.

count

The number of registers to get [1,100] (integer)

mbtype

The Modbus Register Type (integer). Ignored by function but reserved for future use.

- 0 = holding register (all DXM local registers are holding registers
- 3 = coil
- 4 = input
- 5 = input register
- 6 = single coil
- 7 = single register

reg

The register address (integer)

sid

The Modbus Slave ID (integer) of the device

- 0-198 = External Modbus slave devices
- 199 = Internal Local Registers
- 200 = I/O board registers
- 201 = Display board registers
- 203 = On-board I/O (refer to the device datasheet for available I/O board, display board, and on-board I/O)

values

multiset: The list of register values (integer)to set. Length of list must be 1 to 100 values. Values in the list must match the register type: Uint16 for offboard Modbus registers, int32/uint32 for SID 199 internal registers, and SEM32 format for SID 199 floating point registers.

setreg: Union[int,float]. Integer values can be written to any register; floating point values can only be written to floating point registers

This call may take some time. The delay is from networking delays when external devices are polled.

Example: getreg

This example code gets the value of local register 1.

```
reg1 = pyb.getreg(1, 199, 0)
```

Example: multiget

This example gets the value of local registers 1 through 5. The value of "my_local_regs" will be a list of the results from the register read.

```
my_local_regs = pyb.multiget(1, 5, 199, 0)
```

Example: setreg

This example code sets the value of local register 1 to 10.

```
pyb.setreg(1, 10, 199, 0)
```

This example code, specific to the DXM700 models, turns on the red LED 1 on the DXM700 display board.

```
pyb.setreg(1102,1,201,0)
```

Example: multiset

This example code sets the value of local registers 1 through 5 with the values 1 through 5, respectively.

```
pyb.multiset(1, [1,2,3,4,5], 199, 0)
```

Error Codes

Return Value	Register Access Error
70001	API not available
80001	Script timeout
80003	Queue full
80004	API parameter issue
80005	Not authorized
80006	General failure
80007	Out of memory
80008	Unsupported function
80009	Syntax error
80010	Bad command
80011	General process error
80012	Result process error
80021	Transport failure
80022	Transport timeout

Reading invalid registers or registers from an invalid range returns a 0, -1, or an error code. If more than 100 registers are requested, the contents of indices beyond 100 of the returned list will likely be 0.

1.2.4 gc (garbage collector)

This module exposes memory management functions and statistics. The garbage collector is enabled by default. Disabling the garbage collector is highly discouraged.

Function	Result
gc	Garbage collector
gc.collect()	Run a garbage collection
gc.enable()	Enable automatic garbage collection
gc.mem_free()	Return the number of bytes of available heap RAM, or -1 if this amount is not known.
gc.disable()	Disable automatic garbage collection. Heap memory can still be allocated, and garbage collection can still be initiated manually using gc.collect().
gc.isenabled	
gc.mem_alloc()	Returns the number of bytes of heap RAM that are allocated

For ongoing scripts, it is recommended to periodically call gc.collect() to manage the memory contents in the DXM generated by the script. Understand that gc.collect() can be a time consuming function. More information is found at: https://docs.micropython.org/en/latest/library/gc.html

1.2.5 math

This is a list of included python math functions and constants in this implementation.

Math Function	Command	Result
acos	math.acos(x)	Returns the inverse cosine of x
asin	math.sin(x)	Returns the inverse sine of x
atan	math.atan(x)	Returns the inverse tangent of x
atan2	math.atan2(y,x)	Returns the principal value of the inverse tangent of y/x
ceil	math.ceil(x)	Returns an integer, x, rounded towards positive infinity
copysign	math.copysign(x,y)	Returns x with the sign of y
cos	math.cos(x)	Returns the cosine of x
degrees	math.degrees(x)	Returns radians x converted to degrees
e	math.e	Constant that refers to e, the base of the natural logarithm function
exp	math.exp(x)	Returns the exponential of x
fabs	math.fabs(x)	Returns the absolute value of x
floor	math.floor(x)	Returns an integer, x, rounded towards negative infinity
fmod	math.fmod(x,y)	Returns the remainder of x/y
frexp	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.
isfinite	math.isfinite(x)	Returns true if x is finite
isinf	math.isinf(x)	Returns true if x is infinite
isnan	math.isnan(x)	Returns true if x is not a number
ldexp	math.ldexp(x, exp)	Returns x * 2^{exp}
log	math.log(x)	Returns the natural logarithm of x
modf	math.modf(x)	Returns a tuple of two floats, being the fractional and integral parts of x. Both return values have the same sign as x.
pi	math.pi	Constant that refers to the ratio of a circle's circumference to its diameter, pi
pow	math.pow(x,y)	Returns x to the power of y
radians	math.radians(x)	Returns degrees x converted into radians
sin	math.sin(x)	Returns the sine of x
sqrt	math.sqrt(x)	Returns the square root of x
tan	math.tan(x)	Returns the tangent of x
trunc	math.trunc(x)	Returns an integer x rounded towards zero (0)

For more information, see https://docs.micropython.org/en/latest/library/math.html.

1.2.6 micropython

Access and control for MicroPython internals.

Command	Description

micropython.const(expr)	Use to declare the expression is a constant so the compile can optimise it. Example:
	`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. If a constant begins with an underscore it is hidden (not available as a global variable) and does not take up memory during execution.

micropython.heap_lock()	
micropython.heap_unlock()	
micropython.mem_info([verbose])	Print information about currently used memory. If the verbose argument is given then extra information is printed. The printed information 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.opt_level([level])	If `level` is given, this function sets the optimization level for subsequent compilation of scripts and returns None. Otherwise it returns the current optimization level. The optimization level controls the following compilation features: • 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 is True; at levels 1 and higher it is False. • Source-code line numbers: at levels 0, 1 and 2 source-code line number are stored along with the bytecode so exceptions can report the line number they occurred at; at levels 3 and higher line numbers are not stored. The default optimization level is usually level 0.
micropython.pystack_use	
micropython.qstr_info([verbose])	Prints 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()	Returns an integer representing the current amount of stack being used. The absolute value of this is not particularly useful and should be used to compute differences in stack use at different points.

More documentation is found at: https://docs.micropython.org/en/latest/library/micropython.html.

Avoid using heap_lock and heap_unlock. These functions can have unexpected effects on the MicroPython environment.

The most useful function in this library is const. The const function is used to declare optimized constants in a namespace. This can be used with most MicroPython datatypes (bytes, bytearrays, strings, numbers, lists, etc.) and allows the creation of read-only variables that are stored in the most efficient manner possible in their given namespace. This is a simple const demo.

```
>>> class constDemo:
...     publicNumber = micropython.const(1)
...     _privateNumber = micropython.const(-1)
...     publicString = micropython.const("I am Constant")
...     _privateString = micropython.const("I am private")
...
...     def __init__(self, args):
...         self.arg = args
...         print("Made a const demo with args: " + str(args))
...
...
...
>>> a = constDemo(1)
Made a const demo with args: 1
>>> b = constDemo(2)
Made a const demo with args: 2
>>> a._privateNumber
-1
>>> b._privateString
'I am private'
>>> a.publicString is b.publicString
True
```

1.2.7 string (str)

These functions generally match the behavior of their desktop Python counterparts. They can be called on from an instance or from a class method.

Command	Description
str.count(substring)	Returns the number of occurrences of the substring in the given string.
str.endswith(suffix[, start[, end]])	Returns true if the string ends with the specified `suffix`, otherwise returns False optionally restricting the matching with the given indices `start` and `end`. • `suffix`—Could be a string or a tuple of suffixes • `start`—Slice begins here • `end`—Slice ends here
str.find(sub[, start[, end]])	Returns the index of first occurrence of the substring (if found). If not found, it returns -1.
str.format	Format strings contain "replacement fields" surrounded by curly braces {}. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. Search online Python documentation for examples and guidelines.
str.index(sub[, start[, end]])	Returns the index of a substring inside the string (if found). If the substring is not found, it raises an exception.
str.isalpha()	Returns true if all characters in the string are alphabet characters.
str.isdigit()	Returns true if all characters in the string are numerical digits.
str.islower()	Checks whether all the case-based characters (letters) of the string are lowercase. Returns true if all cased characters in the string are lowercase and there is at least one cased character. Returns false otherwise.
str.isspace()	Returns true if all characters in the string are whitespace characters, otherwise, returns false. Use this function to check if the argument contains all whitespace characters such as: • ' ' - Space • \t - Horizontal tab • \n - Newline • \v - Vertical tab • \f - Feed • \r - Carriage return
str.isupper()	Returns true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.
str.join(iterable)	Returns a string by joining all the elements of an iterable (such as list, string, and tuple), separated by a string separator.
str.lower()	Returns a string that converted any uppercase characters to lowercase characters
str.lstrip()	Returns a copy of the string with leading characters removed (based on the string argument passed).
str.replace(old, new[, max])	Returns a copy of the string in which the occurrences of `old` have been replaced with `new`, optionally restricting the number of replacements to `max`.
str.rfind(sub[, start[, end]])	Returns the highest index of the substring (if found). If not found, it returns -1. • sub—It's the substring to be searched in the str string • start and end (optional)—substring is searched within str[start:end]
str.rindex(sub[, start[, end]])	Returns the highest index of the substring inside the string (if found). If the substring is not found, it raises an exception. • sub—substring to be searched in the str string • start and end (optional)—substring is searched within str[start:end]

str.rsplit()	Split a string into a list, using comma, followed by a space (,) as the separator
	`txt = "apple, banana, cherry"`
	`x = txt.rsplit(", ")`
	`print(x)`
	Result: ['apple', 'banana', 'cherry']

str.rstrip()	Returns a copy of the string with trailing characters removed (based on the string argument passed). • chars (optional)—a string specifying the set of characters to be removed. If chars argument is not provided, all whitespaces on the right are removed from the string. `string.rstrip([chars])`

Command	Description

str.split()	Split a string into a list where each word is a list item
	`txt = "welcome to the jungle"`
	`x = txt.split()`
	`print(x)`
	Result: ['welcome', 'to', 'the', 'jungle']

str.startswith()	Python string method startswith() checks whether string starts with `str`, optionally restricting the matching with the given indices `start` and `end`.
str.strip()	Returns a copy of the string by removing both the leading and the trailing characters (based on the string argument passed). • chars (optional)—a string specifying the set of characters to be removed from the left and right part of the string. `string.strip([chars])`
str.upper()	Returns a string that converted any lowercase characters to uppercase characters

```
# instance
>>> a = '1,2,3,4,5,6,7,8'
>>> a.split(",")
['1', '2', '3', '4', '5', '6', '7', '8']
# class call
>>> str.split(a,',')
['1', '2', '3', '4', '5', '6', '7', '8']
```

1.2.8 uarray

The uarray function provides fast, memory efficient access to typed data. This is useful for data conversion (ie: converting Modbus registers values to SEM32 values) and data transmission (sending data over a socket).

Array slice assignment is not supported in this build.

Table 3: Supported uarray specifiers

Type Specifier	C Type	Python type	Standard size in bytes
b	Signed char	Integer	1
B	Unsigned char	Integer	1
h	Short	Integer	2
H	Unsigned short	Integer	2
i	Int	Integer	4
I	Unsigned int	Integer	4
l	Long	Integer	4
L	Unsigned long	Integer	4
q	Long long	Integer	8
Q	Unsigned long long	Integer	8
s	Char[]	bytes	
P	Void *	Integer	
f	Float	Float	4
d	Double	Float	8

uarray.array(typecode, [iterable])— Returns a new array object. Type code must be from a given type specifier.

- typecode is the type of data to store in the array.
- iterable optional, if specified add these elements to the array.

uarray.extend(iterable)— Attempts to add the elements from iterable to the end of the instance calling uarray.

```
>>> a = uarray.array("H",[0xFFFF, 0, 1, 2])
>>> b = uarray.array("h",[-1, 0, 1,2])
>>> c = uarray.array("H", [3,4,5])
```

```
>>> a.extend(c)
>>> a
array('H', [65535, 0, 1, 2, 3, 4, 5])
>>> a.extend(b)
>>> a
array('H', [65535, 0, 1, 2, 3, 4, 5, 65535, 0, 1, 2])
```

uarray.append(val)— Add val to a uarray, growing the array. Upython will attempt to convert val to the appropriate value for the array. If that is not possible, append will throw an exception.

```
>>> a = uarray.array("L",[0,1,2])
>>> b = uarray.array('f',[0, 1.0, 2.0])
>>> a
array('L', [0, 1, 2])
>>> b
array('f', [0.0, 1.0, 2.0])
>>> a.append(3)
>>> b.append(3.0)
>>> a
array('L', [0, 1, 2, 3])
>>> b
array('f', [0.0, 1.0, 2.0, 3.0])
>>> b.append(3)
>>> b
array('f', [0.0, 1.0, 2.0, 3.0, 3.0])
>>> a.append(4.0)
Traceback (most recent call last):
  File "<stdin>", in <module>
TypeError: can't convert float to int
```

1.2.9 ubinascii

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

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

1.2.10 ucollections

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

Command	Description
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. After 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)— Adds x to the right side of the deque. Raises IndexError if overflow checking is enabled and there is no more room left. • deque.popleft()— Removes and returns an item from the left side of the deque. Raises IndexError if no items are present.

ucollections.namedtuple(name, fields)	Creates a new namedtuple type with a specific name and set of fields. A namedtuple is a subclass of tuple that accesses 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 string with a space-separated field named (but this is less efficient).
	`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]`

Command	Description

ucollections.OrderedDict(..)	Remembers and preserves the order of keys added. When ordered dict is iterated over, keys/items are returned in the order they were added:
	
	`from ucollections import OrderedDict`
	`# To make benefit of ordered keys, OrderedDict should be initialized`
	`# from sequence of (key, value) pairs.`
	`d = OrderedDict([("z", 1), ("a", 2)])`
	`# More items can be added as usual`
	`d["w"] = 5`
	`d["b"] = 3`
	`for k, v in d.items():`
	`  print(k, v)`

1.2.11 ujson

This modules converts between Python objects and the JSON data format.

Commands	Description
ujson.dump(obj, stream)	Serializes `obj` to a JSON string, writing it to the given stream.
ujson.dumps(obj)	Returns `obj` represented as a JSON string.
ujson.load(stream)	Parses the given stream, interpreting it as a JSON string and deserializing 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)	Parses the JSON `str` and returns an object. Raises ValueError if the string is not correctly formed.

1.2.12 urandom

Use this module to generate random numbers.

Commands	Description
urandom.getrandbits(n)	Returns an integer with n random bits where n may be between 1 and 32 (inclusive).
urandom.seed(n)	Initializes the random number generator with a known integer n; provides reproducibly deterministic randomness from a given starting state (n). Typical use of this function is to seed the generator with something that varies over time, such as the LSB of a timer, such that the randomness starts from a different place every run.
urandom.randint(n)	Return a random integer N such that a ≤ N ≤ b. Alias for randrange(a, b+1).
urandom.randrange(stop)	Return a randomly selected integer between zero and up to (but not including) `stop`.
urandom.randrange(start, stop)	Return a randomly selected integer from range(start, stop).
urandom.randrange(start, stop, step)	Return a randomly selected element from range(start, stop, step).
urandom.choice(seq)	Return a random element from the non-empty sequence seq. If seq is empty, raises IndexError.
urandom.random()	Return the next random floating point number in the range (0.0, 1.0)
urandom.uniform(a,b)	Return a random floating point number N such that a ≤ N ≤ b for a ≤ b and b ≤ N ≤ a for b < a.

1.2.13 ure

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).

Commands	Description
ure.compile(regex_str[, flags])	Compiles regular expression, returns `regex` object.

Commands	Description
ure.match(regex_str, string)	Compiles `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 searches `string` for the first position that 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 stops after this many substitutions are made. The `flags` argument is ignored.

1.2.14 ustruct

Functions for packing and unpacking data into well-formatted c-style structures.

Table 4: Supported number format specifiers

Format Specifier	C Type	Python type	Standard size in bytes
b	Signed char	Integer	1
B	Unsigned char	Integer	1
h	Short	Integer	2
H	Unsigned short	Integer	2
i	Int	Integer	4
I	Unsigned int	Integer	4
l	Long	Integer	4
L	Unsigned long	Integer	4
q	Long long	Integer	8
Q	Unsigned long long	Integer	8
s	Char[]	bytes	
P	Void *	Integer	
f	Float	Float	4
d	Double	Float	8

Table 5: Supported byte ordering prefixes

Prefix	Description
@	Native byte ordering
<	Little-endian
>	Big-endian
!	Network ordering (big-endian)

Example for ustruct.calcsize (fmt)

Returns the byte size of a given format specifier where fmt is a string of the form: "bf1f2f3.."
b is optional. If specified, it must be in the supported byte ordering prefixes.
f1 ... fn format specifier. Must be in the table of supported format specifiers.

```
calcsize( "<LL" ) == 8
"<LL" format string generates 2 unsigned longs -> 2*4 bytes = 8.
```

Example for ustruct.pack(fmt, v1, v2, v3 …)

Returns a bytes object composed of arguments v1 .. vn formatted by fmt, where fmt is a string of the form: "bf1f2f3.."

b is optional. If specified, it must be in the supported byte ordering prefixes.

f1 … fn format specifier to apply to argument vn. Must be in the table of supported format specifiers.

v1 … vn arguments to format

```
pack(">Ih",0x10A0B0C0, -1)
b'\x10\xa0\xb0\xc0\xff\xff'
 ">" output is in big-endian format.
"I" format the first argument as an unsigned 32-bit int.
"h" format the second argument as a signed 16-bit int.
```

```
pack("<II",0x10A0B0C0, 0xFFFF)
b'\xc0\xb0\xa0\x10\xff\xff\x00\x00'
"<" output is in little-endian format.
"I" format the first argument as an unsigned 32-bit int.
"I" format the second argument as an unsigned 32-bit int.
```

```
pack("H",65536) Equivalently, pack("H",0x10000)
b'\x00\x00'
Use platform default endian-ness.
"H" format the first argument as an unsigned 16-bit value
```

Example for ustruct.pack_into(fmt, buffer, offset, v1,v2,v3 ..)

Packs bytes into buffer, starting at offset. Bytes are composed of v1 .. vn, formatted by fmt, where fmt is a string of the form: "bf1f2f3.."

b is optional. If specified, it must be in the supported byte ordering prefixes.

f1 … fn format specifier to apply to argument vn. Must be in the table of supported format specifiers.

Buffer is any object that implements the buffer protocol, typically array or bytearray. It is up to the user to ensure that buffer is large enough to handle the output from pack_into.

offset is the offset, in bytes, to begin writing the pack values. May be negative to start writing from the end of the buffer.

v1 … vn arguments to format

```
>>> a = bytearray(10)
>>> pack_into("<Hh", a, 0, 0x0102, -2)
>>> a
bytearray(b'\x02\x01\xfe\xff\x00\x00\x00\x00\x00\x00')
>>> pack_into("<Hh", a, -4, 0x0102, -2)
>>> a
bytearray(b'\x02\x01\xfe\xff\x00\x00\x02\x01\xfe\xff')
```

```
>>> a = array('L',[0,1])
>>> pack_into(">HH",a,0,0x0102, 0x0304)
>>> hex(a[0])
'0x4030201'
>>> a
array('L', [67305985, 1])
>>> pack_into(">HH",a,-4,0x0102, 0x0304)
>>> a
array('L', [67305985, 67305985])
>>> hex(a[1])
'0x4030201'
>>> a[1] = 1
>>> pack_into(">H",a,-2,0x0102)
>>> a
array('L', [67305985, 33619969])
>>> hex(a[1])
'0x2010001'
```

Example for ustruct.unpack(fmt, arg)

Returns a tuple where each element is formatted according to fmt with bytes found in arg, where fmt is a string of the form: "bf1f2f3.."

b is optional. If specified, it must be in the supported byte ordering prefixes.

f1 … fn format specifier to apply to argument vn. Must be in the table of supported format specifiers.

arg is any object that implements the buffer protocol, typically array or bytearray. It is up to the user to ensure that the buffer is large enough to handle the format string.

```
>>> a = bytes([01,02,03,04])
>>> a
b'\x01\x02\x03\x04'
>>> unpack("Hh",a)
(513, 1027)
```

Example for struct.unpack_from(fmt, buffer, offset)

Returns a tuple where each element is formatted according to fmt with bytes found in buffer, starting at index offset, where fmt is a string of the form: "bf1f2f3.."

b is optional. If specified, it must be in the supported byte ordering prefixes.

f1 ... fn format specifier to apply to argument vn. Must be in the table of supported format specifiers.

buffer is any object that implements the buffer protocol, typically array or byte-array. It is up to the user to ensure that buffer is large enough to handle the output from pack_into.

offset is the offset, in bytes, to begin writing the pack values. May be negative to start writing from the end of the buffer.

```
>>> a = bytes([0, 1, 2, 3, 4])
>>> a
b'\x00\x01\x02\x03\x04'
>>> unpack_from('hH',a, 1)
(513, 1027)
```

1.2.15 utime

Use this module to add a sleep function or for timing events or functions.

Commands	Description
utime.sleep(seconds)	Sleep for the given number of seconds, should be positive or 0.
utime.sleep_ms(ms)	Delay for given number of milliseconds, should be positive or 0.
utime.ticks_ms()	Returns an increasing millisecond counter with an arbitrary reference point, that wraps around after some value. Only non-negative values are used. For the most part, treat values returned by these functions as opaque. The only operations available for them are ticks_diff() and ticks_add() functions described below. Performing standard mathematical operations (+, -) or relational operators (<, <=, >, >=) directly on these value leads to invalid results. Performing mathematical operations and then passing their results as arguments to ticks_diff() or ticks_add() also leads to invalid results from the latter functions.
utime.ticks_us()	Just like ticks_ms(), but in microseconds.
utime.ticks_cpu()	Similar to ticks_ms() and ticks_us(), but with the highest possible resolution in the system. This is usually in CPU clocks. The exact timing unit (resolution) of this function is not specified on utime module level, but documentation for a specific port may provide more specific information. This function is intended for very fine benchmarking or very tight real-time loops. Avoid using it in portable code.

utime.ticks_add(ticks, delta)	Offset ticks value by a given number (either positive or negative).
	Given a ticks value, this function calculates ticks value delta ticks before or after it, following modular-arithmetic definition of tick values (see ticks_ms() above). The 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()). Delta can be an arbitrary integer number or numeric expression. ticks_add() is useful for calculating deadlines for events/tasks. (Use ticks_diff() function to work with deadlines.)
	`# 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))`

Commands	Description

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 in between, 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 returns -TICKS_PERIOD/2 instead. For example, the result value wraps around to the negative range of possible values.
	ticks_diff() is designed to accommodate various use patterns, including:
	• Polling with timeout. The order of events is known and the function deals 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. 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 that **`sleep_us`** is not implemented. The **`sleep`** feature accepts a floating point number for realizing sleep time of less than one second. For example, **`sleep(0.345)`** translates to sleep for 345 ms. The finest sleep granularity/resolution possible is approximately 3 ms.