Digi MicroPython

Programming Guide
Revision history

Revision history—90002219

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>December 2020</td>
<td>Added <code>Idle device from MicroPython</code> and <code>apin.read_u16()</code>. Updated <code>Initiate sleep from MicroPython</code> for the non-cellular devices. Updated <code>Test the connection</code>. Updated PyCharm section.</td>
</tr>
<tr>
<td>W</td>
<td>February 2021</td>
<td>Added <code>Recover an XBee device</code>.</td>
</tr>
<tr>
<td>Y</td>
<td>June 2022</td>
<td>Added the <code>cellular.scan(deep, callback)</code> method.</td>
</tr>
<tr>
<td>Z</td>
<td>August 2022</td>
<td>Added note to code storage topic that main code will not be run on startup.</td>
</tr>
<tr>
<td>AA</td>
<td>September 2022</td>
<td>Added new value type to supported datapoint values, added <code>time.tz_offset()</code>, added <code>boot.py</code>, created new Control MicroPython from Remote Manager section.</td>
</tr>
</tbody>
</table>

Trademarks and copyright

Digi, Digi International, and the Digi logo are trademarks or registered trademarks in the United States and other countries worldwide. All other trademarks mentioned in this document are the property of their respective owners.
© 2017-2023 Digi International Inc. All rights reserved.

Disclaimers

Information in this document is subject to change without notice and does not represent a commitment on the part of Digi International. Digi provides this document “as is,” without warranty of any kind, expressed or implied, including, but not limited to, the implied warranties of fitness or merchantability for a particular purpose. Digi may make improvements and/or changes in this manual or in the product(s) and/or the program(s) described in this manual at any time.

Warranty

To view product warranty information, go to the following website:

www.digi.com/howtobuy/terms
Customer support

Gather support information: Before contacting Digi technical support for help, gather the following information:

- Product name and model
- Product serial number(s)
- Firmware version
- Operating system/browser (if applicable)
- Logs (from time of reported issue)
- Trace (if possible)
- Description of issue
- Steps to reproduce

Contact Digi technical support: Digi offers multiple technical support plans and service packages. Contact us at +1 952.912.3444 or visit us at www.digi.com/support.

Feedback

To provide feedback on this document, email your comments to

techcomm@digi.com

Include the document title and part number (Digi MicroPython Programming Guide, 90002219 AA) in the subject line of your email.
## Keyboard shortcuts

<table>
<thead>
<tr>
<th>Keyboard shortcuts</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select a previously typed statement</td>
<td>26</td>
</tr>
</tbody>
</table>

## Differences between MicroPython and other programming languages

<table>
<thead>
<tr>
<th>Memory management</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable types</td>
<td>28</td>
</tr>
<tr>
<td>Syntax</td>
<td>28</td>
</tr>
<tr>
<td>Curly braces and indentation</td>
<td>29</td>
</tr>
<tr>
<td>Semicolons</td>
<td>30</td>
</tr>
<tr>
<td>Increment operator</td>
<td>30</td>
</tr>
<tr>
<td>Logical operators</td>
<td>31</td>
</tr>
</tbody>
</table>

## Develop applications on an XBee device

<table>
<thead>
<tr>
<th>Space allocated to MicroPython</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code storage</td>
<td>33</td>
</tr>
<tr>
<td>Built-in modules embedded in XBee firmware (device flash)</td>
<td>33</td>
</tr>
<tr>
<td>Source code in .py files (file system)</td>
<td>33</td>
</tr>
<tr>
<td>Parsed and compiled code in .mpy files (file system)</td>
<td>33</td>
</tr>
<tr>
<td>Executable code on MicroPython heap (device RAM)</td>
<td>34</td>
</tr>
<tr>
<td>Compiled modules relocated from file system to device flash</td>
<td>34</td>
</tr>
<tr>
<td>How to organize your code</td>
<td>34</td>
</tr>
<tr>
<td>Run code at startup</td>
<td>34</td>
</tr>
<tr>
<td>boot.py</td>
<td>34</td>
</tr>
<tr>
<td>main.py</td>
<td>34</td>
</tr>
<tr>
<td>Monitor memory usage</td>
<td>35</td>
</tr>
<tr>
<td>The gc module</td>
<td>35</td>
</tr>
<tr>
<td>The micropython module</td>
<td>35</td>
</tr>
<tr>
<td>Efficient coding</td>
<td>38</td>
</tr>
<tr>
<td>Application evolution</td>
<td>38</td>
</tr>
<tr>
<td>One-liners in the REPL</td>
<td>38</td>
</tr>
<tr>
<td>Short blocks in paste mode</td>
<td>38</td>
</tr>
<tr>
<td>Flash upload mode</td>
<td>39</td>
</tr>
<tr>
<td>Modules stored as .py files</td>
<td>39</td>
</tr>
<tr>
<td>Compiled modules stored as .mpy files</td>
<td>39</td>
</tr>
<tr>
<td>Compiled modules via Flash upload mode</td>
<td>40</td>
</tr>
<tr>
<td>Compiled modules embedded in device flash</td>
<td>40</td>
</tr>
<tr>
<td>Digi modified slicing for bytes and strings operations</td>
<td>41</td>
</tr>
</tbody>
</table>

## Power management in MicroPython

<table>
<thead>
<tr>
<th>Prevent sleep from MicroPython</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBee Cellular Modem:</td>
<td>43</td>
</tr>
<tr>
<td>XBee 3 Zigbee RF Module, XBee 3 802.15.4 RF Module, XBee 3 DigiMesh RF Module</td>
<td>43</td>
</tr>
<tr>
<td>description:</td>
<td>43</td>
</tr>
<tr>
<td>Initiate sleep from MicroPython</td>
<td>44</td>
</tr>
<tr>
<td>XBee Cellular Modem:</td>
<td>44</td>
</tr>
<tr>
<td>XBee 3 Zigbee RF Module, XBee 3 802.15.4 RF Module, XBee 3 DigiMesh RF Module:</td>
<td>44</td>
</tr>
<tr>
<td>Sleeping with AT commands</td>
<td>45</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Idle a device from MicroPython</td>
<td>45</td>
</tr>
<tr>
<td>Enter and exit the idle state</td>
<td>45</td>
</tr>
<tr>
<td>Poll for data while the device is idle</td>
<td>46</td>
</tr>
<tr>
<td><strong>Control MicroPython from Remote Manager</strong></td>
<td></td>
</tr>
<tr>
<td>Run ATPY commands as a data service request</td>
<td>47</td>
</tr>
<tr>
<td><strong>Access the primary UART</strong></td>
<td></td>
</tr>
<tr>
<td>How to use the primary UART</td>
<td>51</td>
</tr>
<tr>
<td>sys.stdin limitations</td>
<td>51</td>
</tr>
<tr>
<td>Example: read bytes from the UART</td>
<td>51</td>
</tr>
<tr>
<td>Example: read the first 15 bytes from the UART</td>
<td>52</td>
</tr>
<tr>
<td><strong>REPL (Read-Evaluate-Print Loop) examples</strong></td>
<td></td>
</tr>
<tr>
<td>Ctrl+A: Enter raw REPL mode</td>
<td>54</td>
</tr>
<tr>
<td>Ctrl+B: Print the MicroPython banner</td>
<td>54</td>
</tr>
<tr>
<td>Print the banner</td>
<td>55</td>
</tr>
<tr>
<td>Print the banner and verify that the memory was not wiped</td>
<td>55</td>
</tr>
<tr>
<td>Ctrl+C: Regain control of the terminal</td>
<td>56</td>
</tr>
<tr>
<td>Ctrl+D: Reboot the MicroPython REPL</td>
<td>57</td>
</tr>
<tr>
<td>Ctrl+E: Enter paste mode</td>
<td>57</td>
</tr>
<tr>
<td>Paste one line of code</td>
<td>57</td>
</tr>
<tr>
<td>Paste a code segment</td>
<td>58</td>
</tr>
<tr>
<td>Ctrl+F: Upload code to flash</td>
<td>58</td>
</tr>
<tr>
<td>Load code to flash memory</td>
<td>59</td>
</tr>
<tr>
<td>Erase the code stored in flash memory</td>
<td>60</td>
</tr>
<tr>
<td>Flash memory and automatic code execution</td>
<td>60</td>
</tr>
<tr>
<td>Run stored code at start-up to flash LEDs</td>
<td>60</td>
</tr>
<tr>
<td>Disable code from running at start up</td>
<td>61</td>
</tr>
<tr>
<td>Ctrl+R: Run code in flash</td>
<td>62</td>
</tr>
<tr>
<td>Enable code to run at start-up</td>
<td>62</td>
</tr>
<tr>
<td>Perform a soft-reset or reboot</td>
<td>63</td>
</tr>
<tr>
<td><strong>Access file system in MicroPython</strong></td>
<td></td>
</tr>
<tr>
<td>Modify file system contents</td>
<td>65</td>
</tr>
<tr>
<td>uos.chdir(dir)</td>
<td>65</td>
</tr>
<tr>
<td>uos.getcwd()</td>
<td>65</td>
</tr>
<tr>
<td>uos.ilistdir(dir)</td>
<td>65</td>
</tr>
<tr>
<td>uos.listdir(dir)</td>
<td>65</td>
</tr>
<tr>
<td>uos.mkdir(dir)</td>
<td>65</td>
</tr>
<tr>
<td>uos.remove(file)</td>
<td>65</td>
</tr>
<tr>
<td>uos.rmdir(dir)</td>
<td>65</td>
</tr>
<tr>
<td>uos.rename(old_path, new_path)</td>
<td>65</td>
</tr>
<tr>
<td>uos.replace(old_path, new_path)</td>
<td>65</td>
</tr>
<tr>
<td>uos.sync()</td>
<td>65</td>
</tr>
<tr>
<td>uos.compile(source_file, mpy_file=None)</td>
<td>66</td>
</tr>
<tr>
<td>uos.format()</td>
<td>66</td>
</tr>
<tr>
<td>uos.hash(secure_file)</td>
<td>66</td>
</tr>
<tr>
<td>Access data in files</td>
<td>67</td>
</tr>
</tbody>
</table>
File object methods ........................................... 67
    read(size=-1) .................................................. 67
    readinto(b) .................................................. 67
    readline(size=-1) ........................................... 68
    readline() .................................................. 68
    write(b) .................................................. 68
    seek(offset, whence=0) ..................................... 68
    tell() .................................................. 68
    flush() .................................................. 68
    close() .................................................. 68
Import modules from file system .................................. 68
Reload a module ................................................ 69
Compiled MicroPython files ........................................ 69

Send and receive User Data Relay frames

Constants .................................................. 71
    Interfaces (always defined) .................................. 71
    Limits .................................................. 71
Methods .................................................. 71
    relay.receive() ........................................... 71
    relay.send(dest, data) ..................................... 71
Exceptions ................................................ 71
    relay.callback(my_callback) ................................ 71
Examples .................................................. 72

MicroPython libraries on GitHub

MicroPython modules

XBee-specific functions ........................................ 75
Standard modules and functions ................................ 75
Discover available modules ..................................... 76

Digi extensions to the utime module

time.tz_offset() .................................................. 77

Machine module

Reset-cause .................................................. 79
    Constants .................................................. 79
Random numbers ................................................ 79
Unique identifier ............................................. 79
Class PWM (pulse width modulation) .......................... 79
Class ADC: analog to digital conversion ....................... 80
    Constructors ............................................. 80
    Methods .................................................. 81
    Sample program ........................................... 81
Class I2C: two-wire serial protocol .......................... 82
    Constructors ............................................. 83
    General methods .......................................... 83
Standard bus operations methods .......................................................... 83
Memory operations methods ...................................................................... 83
Sample programs ......................................................................................... 84
Class Pin ........................................................................................................ 87
Class UART .................................................................................................... 87
Test the UART interface ............................................................................... 88
Use the UART class ...................................................................................... 88
Constructors ................................................................................................. 89
Methods ......................................................................................................... 89
Constants ....................................................................................................... 90
Class WDT: watchdog timer ........................................................................ 90
Access the XBee device’s I/O pins ................................................................. 90
Use the Pin() constructor ............................................................................ 92
Use mode() to configure a pin ....................................................................... 93
  Pin.DISABLED ............................................................................................ 93
  Pin.IN ......................................................................................................... 93
  Pin.OUT ...................................................................................................... 93
  Pin.ALT ....................................................................................................... 94
  Pin.ANALOG .............................................................................................. 94
  Pin.OPEN_DRAIN and Pin.ALTOPEN_DRAIN ........................................... 94
Use pull() to configure an internal pull up/down resistor ............................. 94

digi.ble module

Feature support ......................................................................................... 97
active() ....................................................................................................... 97
config() ..................................................................................................... 98
  Query a value .......................................................................................... 98
  Update configuration values .................................................................... 98
disconnect_code() .................................................................................... 99
gap_connect() .......................................................................................... 99
  <addr_type> .......................................................................................... 99
  <address> ............................................................................................. 99
  <timeout_ms> ......................................................................................... 100
  <interval_us>, <window_us> ................................................................. 100
  <onclose> .............................................................................................. 100
Return value ............................................................................................... 100
gap_connection methods .......................................................................... 100
gattc_services() ....................................................................................... 100
gattc_characteristics() ........................................................................... 101
gattc_descriptors() .................................................................................. 102
gattc_read_characteristic() ..................................................................... 102
gattc_configure() .................................................................................... 102
gattc_read_descriptor() .......................................................................... 103
gattc_write_characteristic() ................................................................... 103
gattc_write_descriptor() ......................................................................... 104
addr() ........................................................................................................ 104
close() ....................................................................................................... 104
config() ..................................................................................................... 104
isconnected() ........................................................................................... 106
secure() ...................................................................................................... 106
io_callbacks() ......................................................................................... 106
delete_bondings() ...................................................................................... 107
passkey_enter() ....................................................................................... 107
passkey_confirm() ..................................................................................... 107
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UUID()</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>&lt;value&gt;</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>Return value</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>gap_scan()</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>&lt;duration_ms&gt;</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>&lt;interval_us&gt;, &lt;window_us&gt;</td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>&lt;oldest&gt;</td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>Return value</td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>gap_scan methods</td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>get()</td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>any()</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>stop()</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>stopped()</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>gap_scan advertisement format</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>Use gap_scan as an iterator</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>Use gap_scan as a context manager</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>gap_advertise()</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>&lt;interval_us&gt;</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>&lt;adv_data&gt;</td>
<td></td>
<td>111</td>
</tr>
<tr>
<td>Return value</td>
<td></td>
<td>111</td>
</tr>
<tr>
<td>xbee_connect()</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>&lt;gap_connection&gt;</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>&lt;receive&gt;</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>&lt;password&gt;</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>&lt;timeout&gt;</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>Return value</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>xbee_connection methods</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>digi.ble samples</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>Generic gap advertising and gap scanning samples</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>Eddystone Beaconing samples</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>iBeacon samples</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>Troubleshooting</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>Fewer advertisements than expected when using gap_scan</td>
<td></td>
<td>113</td>
</tr>
</tbody>
</table>

**Cellular network configuration module**

Configure a specific network interface          | 115  |
class Cellular                                        | 115  |
| Constructors |                                                  | 115  |
| Cellular power and airplane mode method |                                          | 116  |
| Verify cellular network connection method |                                     | 116  |
| Cellular connection configuration method |                                    | 116  |
| Cellular active scan method |                                      | 116  |
| Send an SMS message method |                                      | 117  |
| Receive an SMS message method |                                      | 117  |
| Register an SMS Receive Callback method |                             | 118  |
| Cellular shutdown method |                                     | 118  |
| RSRP/RSRQ reporting in MicroPython |                                | 118  |

**XBee module**

AT commands that do not work in MicroPython | 121  |
class XBee on XBee Cellular Modem | 121  |
| Constructors |                                                  | 121  |
Methods ........................................................................................................... 122
XBee MicroPython module on the XBee 3 RF Modules ......................................... 122
Functions ........................................................................................................... 122
atcmd() ............................................................................................................. 122
discover() .......................................................................................................... 123
receive() ............................................................................................................ 124
receive_callback(rx_callback) ............................................................................. 125
transmit() .......................................................................................................... 125
modem_status .................................................................................................... 127
digi.cloud module
Create and upload data points ............................................................................. 129
class DataPoints .................................................................................................. 129
  Constructor ....................................................................................................... 129
  Optional parameter ............................................................................................ 129
  Add a data point method ................................................................................... 129
  Upload data to Digi Remote Manager method .................................................. 130
  Check the status of a DataPoints object .......................................................... 131
  The life-cycle of a DataPoints object ................................................................ 131
  Delete a DataPoints object .............................................................................. 131
Receive a Data Service Device Request .............................................................. 132
class device_request ............................................................................................. 132
  Use the read(size=-1) method ........................................................................ 133
  Use the readinto(b) method ............................................................................ 133
  Use the write(b) method .................................................................................. 133
  Use the close() method .................................................................................. 133
Use the API Explorer to send Device Requests .................................................... 133
digi.cloud.Console() object ................................................................................ 134
  isConnected() method ..................................................................................... 134
  read(size) method ............................................................................................ 134
  readinto(buf) method ...................................................................................... 134
  write(buf) method ............................................................................................ 134
  close() method ................................................................................................ 134
The ussl module
ussl on the XBee Cellular Modem .......................................................................... 136
Syntax .................................................................................................................. 136
Usage .................................................................................................................... 136
digi.gnss module
GNSS module methods ........................................................................................ 138
  single_acquisition(callback, timeout=60) ....................................................... 138
  raw_mode(callback) ......................................................................................... 138
GNSS examples ...................................................................................................... 139
Terminal redirection
dupterm(stream_obj, index=0) ............................................................................. 141
Use AWS IoT from MicroPython

Add an XBee Cellular Modem as an AWS IoT device .......................................................... 143
Create a policy for access control ......................................................................................... 143
Create a Thing ......................................................................................................................... 144
Install the certificates .............................................................................................................. 146
Test the connection .................................................................................................................. 147
Publish to a topic ...................................................................................................................... 149
Confirm published data ............................................................................................................ 150
Subscribe to updates from AWS .............................................................................................. 150

Time module example: get the current time

Retrieve the local time .............................................................................................................. 153
Retrieve time with a loop .......................................................................................................... 153
Delay and timing quick reference ............................................................................................ 154

Cellular network connection examples

Check the network connection ............................................................................................... 156
Check network connection with a loop ................................................................................... 156
Check network connection and print connection parameters ................................................... 157

Socket examples

Sockets ..................................................................................................................................... 160
Basic socket operations: sending and receiving data, and closing the network connection ...... 160
  Basic data exchange code sample ......................................................................................... 160
  Response header lines ........................................................................................................... 161
Specialized receiving: send received data to a specific memory location ............................... 162
DNS lookup ............................................................................................................................. 163
  DNS lookup code output ........................................................................................................ 164
Set the timeout value and blocking/non-blocking mode ........................................................... 164
Send an HTTP request and dump the response ........................................................................ 166
Socket errors ........................................................................................................................... 167
  ENOTCONN: Time out error .................................................................................................. 167
  ENFILE: No sockets are available ....................................................................................... 167
  ENXIO: No such device or address ...................................................................................... 167
Unsupported methods .............................................................................................................. 167

I/O pin examples

Change I/O pins ......................................................................................................................... 169
Print a list of pins ....................................................................................................................... 169
Change output pin values: turn LEDs on and off .................................................................. 170
Poll input pin values .................................................................................................................. 170
Check the configuration of a pin .............................................................................................. 171
Check the pull-up mode of a pin .............................................................................................. 173
Measure voltage on the pin (Analog to Digital Converter) ....................................................... 174
SMS examples

Send an SMS message ........................................................................................................ 177
Send an SMS message to a valid phone number ................................................................. 177
Check network connection and send an SMS message ......................................................... 177
Receive an SMS message .................................................................................................... 178
   Sample code ..................................................................................................................... 178
Receive an SMS message using a callback ........................................................................... 179

XBee device examples

Print the temperature of the XBee Cellular Modem ......................................................... 181
Print the temperature of the XBee 3 Zigbee RF Module .................................................... 181
Print a list of AT commands .............................................................................................. 182
xbee.discover() examples .................................................................................................. 184
   Handle responses as they are received ............................................................................. 184
   Gather all responses into a list ......................................................................................... 185
xbee.transmit() examples .................................................................................................. 185
   xbee.transmit() using constants ..................................................................................... 185
   xbee.transmit() using byte string .................................................................................. 185
Digi MicroPython Programming Guide

This guide introduces the MicroPython programming language by showing how to create and run a simple MicroPython program. It includes sample code to show how to use MicroPython to perform actions on a Digi device, particularly those devices with Digi-specific behavior. It also includes reference material that shows how MicroPython coding can be used with Digi devices.

You can code MicroPython to transform cryptic readings into useful data, filter out excess transmissions, directly employ modern sensors and actuators, and use operational logic to glue inputs and outputs together in an intelligent way.

The XBee Cellular Modem has MicroPython running on the device itself. You can access a MicroPython prompt from the XBee Cellular Modem when you install it in an appropriate development board (XBDB or XBIB), and connect it to a computer via a USB cable.

Reference material

MicroPython is an open-source programming language based on the Python 3 standard library. MicroPython is optimized to run on a microcontroller, cellular modem, or embedded system.

Refer to the Get started with MicroPython section of the appropriate user guide for information on how to enter the MicroPython environment and several simple examples to get you started:

- Digi XBee PyCharm IDE Plugin User Guide
- Digi XBee Cellular Embedded Modem User Guide
- Digi XBee Cellular 3G Global Embedded Modem User Guide
- Digi XBee 3 Cellular LTE Cat 1 Smart Modem User Guide
- Digi XBee 3 Cellular LTE-M Global Smart Modem User Guide
- XBee 3 802.15.4 RF Module User Guide
- XBee 3 DigiMesh RF Module User Guide
- XBee 3 Zigbee RF Module User Guide

This programming guide assumes basic programming knowledge. For help with programming knowledge, you can refer to the following sites for Python and MicroPython:

- MicroPython: micropython.org
- MicroPython documentation: docs.micropython.org
- MicroPython Wiki: wiki.micropython.org
- Python: python.org
Which features apply to my device?

MicroPython features and errors differ depending on the device you use. Unless specified, information in this document applies to all devices. This table covers which features apply to specific products:

<table>
<thead>
<tr>
<th>Feature</th>
<th>XBee 3 Cellular</th>
<th>XBee 3 Zigbee, DigiMesh, and 802.15.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital I/O</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I2C</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Power management</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Idle from MicroPython</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Digi Remote Manager</td>
<td>Yes(^1)</td>
<td>No</td>
</tr>
<tr>
<td>Secondary UART</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Real-time clock</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>File system</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>File system - concurrent file writes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>File system - rename</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>File system - Edit files after creation</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>File System - delete</td>
<td>Yes</td>
<td>No(^2)</td>
</tr>
<tr>
<td>File System - secure files</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>File System preserved across updates</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>GNSS on demand</td>
<td>Yes(^3)</td>
<td>No</td>
</tr>
</tbody>
</table>

\(^1\)Remote Manager features are only supported on XBee 3 Cellular devices, not XBee Cellular.

\(^2\)Files can be deleted, but doing so does not reclaim their space on the file system.

\(^3\)XBee 3 Global LTE-M/NB-IoT, XBee 3 Global LTE Cat 1, and XBee 3 North America LTE Cat 1 only.
Use MicroPython

Access the MicroPython environment .......................................................... 16
Enter MicroPython code ............................................................................. 16
Exit MicroPython ......................................................................................... 16
Display tools ............................................................................................... 16
Coding tips .................................................................................................. 16
Recover an XBee device ............................................................................... 17
Access the MicroPython environment

To begin using MicroPython on the XBee device, open XCTU and enter MicroPython mode. See Use XCTU to enter the MicroPython environment in the appropriate user guide.

Enter MicroPython code

You can use different methods to enter MicroPython code into the MicroPython Terminal on the XBee device.

- **Direct entry**: Manually type code into the MicroPython Terminal.
- **Paste mode**: Use the REPL paste mode to paste copied code into the MicroPython Terminal for immediate execution.
- **Flash mode**: Use the REPL flash mode to paste a block of code into the MicroPython Terminal and store it in flash memory.
- **Access file system in MicroPython**: Upload code to the file system.

Direct entry

From a serial terminal, you can type code at the MicroPython REPL prompt. When you press Enter, the line of code runs and another MicroPython prompt appears. Manually typing in code is the simplest method.

**Example**

1. Access the MicroPython environment.
2. At the MicroPython >>> prompt, type `print("This is a simple line of code")` and then press Enter. The phrase in quotes prints in the terminal: This is a simple line of code

Exit MicroPython

When you are done coding, exit MicroPython by closing the MicroPython terminal. Any code that has been executed will continue to run, even if the XBee device is set to Transparent or API mode. For additional instructions, see the Exit MicroPython mode section in the appropriate user guide.

Display tools

MicroPython mode requires echo to be turned off in terminal emulation. Command mode does not echo your input back to you. In order to see what you are typing, use the appropriate display tool:

- **MicroPython mode**: For MicroPython coding, use the XCTU MicroPython Terminal or configure your terminal emulator for "echo off."
- **Command mode**: For device configuration that is done in Command mode (initiated by sending +++ to the device), use the XCTU Serial Console or configure your terminal emulator for "echo on."

Coding tips

For all XBee devices:
Use the integer division operator (\(//\)) unless you need a floating point.

MicroPython's `struct_time` does not include the `tm_isdst` element in the tuple.

For the XBee Cellular Modem:

- The XBee Cellular Modem supports the use of hostnames in `socket.connect()` calls, unlike other MicroPython platforms that require an IP address obtained by doing a manual look-up using `socket.getaddrinfo()`.

For the XBee 3 Zigbee RF Module:

- The Micropython `time.time()` function returns the number of seconds since the epoch. The XBee 3 Zigbee RF Module does not have a realtime clock, so it does not support `time.time()`.

To track elapsed time, use `time.ticks_ms()`.

For XBee3 radio modules:

- The counter for the Micropython `ticks_us()` function will fall behind `ticks_ms()` by about 1 ms every 10 seconds.

- If you need a high level of accuracy over a long period of time, use `ticks_ms()`.

**Recover an XBee device**

If you are unable to communicate with an XBee device when a MicroPython script is running—for example, if your MicroPython code is changing settings such as `ATBD` to strange values—you can recover the XBee device by holding serial break—DIN line low—while the XBee is reset or powered up.

If serial break is held during reset/power-on, the XBee will enter Command mode at 9600 baud, and MicroPython will not execute until Command mode is exited.

While in Command mode you can then set `ATPS` to 0 to disable MicroPython autostart, for example. You can also query and set `ATBD` or other commands to assist in restoring communication with the XBee.

---

**Note** See Break control for related information.
MicroPython syntax

Syntax refers to rules that must be followed when entering code into MicroPython. If you do not follow the syntax rules when coding, errors are generated, and the code may not run as expected or not run at all.

For information about coding errors, see Errors and exceptions.

The following sections describe coding syntax rules.

Colons ................................................................. 19
Indentations ................................................................ 19
Functions ..................................................................... 20
Colons

MicroPython requires a colon (:) after you entered the following statement types:

- Function name and the arguments that function accepts, if any
- Condition statement
- Loop statement

**Defining a function**

A function consists of the following:

- `def` keyword
- Function name
- Any arguments the function takes, inside a set of parentheses. The parentheses remain empty if there are no passed arguments
- The function declaration must be followed by a colon

The code sample below is a basic function definition. Note that a colon is entered after the function name. This colon defines the following indented lines as part of the function. Indentation is equally important, and is discussed in [Indentations](#).

```python
def sample_function():
    print("I am a sample function!")
```

**After conditional statements and loop statements**

A colon is required after each conditional statement and loop statement. The code sample below shows how the colon is used for a conditional statement (`if True:`) and for a loop statement (`for x in range(10):`).

```python
>if True:
    print("Condition is true!")

for x in range(10):
    print("Current number: %d" % x)
```

**Indentations**

In MicroPython, an indentation tells the compiler which statements are members of a function, conditional execution block, or a loop. If a line is not indented, that line is not considered a part of the function, conditional execution block, or loop.

A function declaration, conditional execution block, or loop should be followed by a colon. All code after the colon that is meant to be part of that block must be indented. For more information about how colons are used in the code, see [Colons](#).

**FOR loop with one statement indented**

In this example, only one statement after the initial FOR loop statement (which ends in a colon) is indented. When the loop is executed, only line 2 of the code is executed. When the loop completes, the code at line 3 executes.
When this code executes, it prints "In the FOR loop, iteration # <number>" 10 times, where <number> is 0 in the first loop of the code, and 9 at the last loop. Line 3 of the code runs one time, after the loop completes, printing the phrase "Current number: 9" one time.

```
for x in range(10):
    print("In the FOR loop, iteration # %d % x)"
    print("Current number: %d % x)"
```

FOR loop with two statements indented

In this example, both statements after the initial FOR loop statement (which ends in a colon) are indented. When the loop is executed, both print statements are printed in each loop iteration.

As in the previous example, the code prints "In the FOR loop, iteration # <number>", where <number> is 0 in the first loop of the code, and 9 at the last loop. This time, however, line 3 of the code is run in each loop iteration, and prints the phrase "Current number: number". Both phrases are printed 10 times, with the <number> starting at 0 and increasing by one on each loop.

```
for x in range(10):
    print("In the FOR loop, iteration # %d % x)"
    print("Current number: %d % x)"
```

Functions

A function is an operation that performs an action and may return a value. A function consists of the following:

- **def** keyword. The def keyword is required, and is short for "define".
- Function name.
- Any arguments the function takes, defined by a set of parentheses. The parentheses remain empty if there are no passed arguments.
- The function statement must be followed by a colon. For more information, see Colons.

The code sample below is a basic function definition. Note that the colon is entered after the function name and parentheses. This colon defines that everything after that line that is indented is part of the function. Indentation is equally important, and is discussed in the Indentations section.

```
def example_function():
    print("I am a function!")
```

Function with arguments

This sample shows how to define a function and then how to call the function to perform an operation and return a value.

- Line 1: Define the function and define two arguments: x and y.
- Line 2: Define the variable that holds the sum of the arguments as sum_val.
- Line 3: Define a phrase that will be printed to the terminal including sum_val.
- Line 4: The function returns the value of its own variable sum_val. A returned value can be used and stored outside of the function.
- Line 6: Define the value of the variable global_sum to be the value returned by the function.
defined in line 1: `addition_function(3,4)`, which is equal to the returned variable `sum_val`.
- Line 7: Define that a phrase that includes `global_sum` is printed to the terminal.

```python
def addition_function(x, y):
    sum_val = x + y
    print("value of sum (x+y): %d" % sum_val)
    return sum_val

global_sum = addition_function(3,4)
print("Value of global_sum: %d" % global_sum)
```

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.
Errors and exceptions

If something goes wrong during compilation or during execution of code you have entered, you may get an error. The type of error that occurred and the line number that caused the error will print to the terminal. Errors can happen for many reasons, such as syntax errors, name errors (which generally means the variable or function you are referencing is not available), or other more specific errors.

**Note** Some exceptions have Error in their name and others have Exception.

Common types of errors include:

- Syntax error ................................................................. 23
- Name error ........................................................................ 23
- OSError ........................................................................... 24
- Socket errors .................................................................... 24
Syntax error
A syntax error occurs when a MicroPython code statement has the wrong syntax.

Example
In this example, the syntax is incorrect. A colon is missing after the word "True".

```python
if True print("Condition is true!")
```

When you press Enter to run the code it generates the following Exception describing the error (SyntaxError) and the execution path that led to it (line 1 of the code you entered).

```
Traceback (most recent call last):
File "<stdin>", line 1
SyntaxError: invalid syntax
```

The correct code syntax is:

```python
if True: print("Condition is true!")
```

Name error
A name error is generated when a name of an item, such as a variable or function, cannot be found. This can occur when:

- You typed the name into the code incorrectly.
- You are referencing a name that was never created.
- The name is defined, but is not in scope when you reference it. For example, if you defined the name in function A, but are referencing the name in function B.

Referencing a name that was not created
In this example, the name `deviation_factor` was not created. If you reference this name in the code, a NameError occurs in line 4, as the code references the `deviation_factor` name, which was not created.

```python
print("Assigning value to x...")
x = 17
print("Adding deviation_factor to x...")
x = x + deviation_factor
```

Referencing a name from one function that was created in a different function
In this example, a variable is created in the `example_func`. When you run the code, the NameError references line 8, where the code tries to print `local_variable`. The variable was created inside the function `example_func`, and the scope of that variable, meaning where it can be accessed, is in that function. The code references `local_variable` outside of that function.

```python
def example_func():
    print("Entering example function...")
    local_variable = "I'm a variable inside this function"
```
**Errors and exceptions**

---

```
print(local_variable)
```

```
example_func()
print(local_variable)
```

### OSError

MicroPython returns an OSError when a function returns a system-related error.

For example, if you try to send a message on a Zigbee network:

```
import xbee
xbee.transmit(xbee.ADDR_COORDINATOR, 'Hello!')
```

This code assumes that the device is associated to a network and able to send and receive data. If the device is not associated with a network, it produces an OS error:

**OSError: [Errno 7107] ENOTCONN.**

### Socket errors

**Note** This section only applies to the XBee Cellular Modem. See [Which features apply to my device?](#) for a list of the supported features.

This following socket errors may occur.

#### ENOTCONN: Time out error

If a socket stays idle too long, it will time out and disconnect. Attempting to send data over a socket that has timed out produces the OSError **ENOTCONN**, meaning "Error, not connected." If this happens, perform another `connect()` call on the socket to be able to send data again.

#### ENFILE: No sockets are available

The `socket.socket()` or `socket.connect()` method returns an OSError (**ENFILE**) exception if no sockets are available. If you are already using all of the available sockets, this error may occur in the few seconds between calling `socket.close()` to close a socket, and when the socket is completely closed and returned to the socket pool.

You can use the following methods to close sockets and make more sockets available:

- **Close abandoned sockets**: Initiate garbage collection (`gc.collect()`) to close any abandoned MicroPython sockets. For example, an abandoned socket could occur if a socket was created in a function but not returned. For information about the `gc` module, see the MicroPython garbage collection documentation.

- **Close all allocated sockets**: Press Ctrl+D to perform a soft reset of the MicroPython REPL to close all allocated sockets and return them to the socket pool.

#### ENXIO: No such device or address

OSError (**ENXIO**) is returned when DNS lookups fail from calling `usocket.getaddrinfo()`.
Keyboard shortcuts

This section includes keyboard shortcuts you can use to make coding with MicroPython easier.

Keyboard shortcuts ................................................................................................................26
Select a previously typed statement .....................................................................................26
Keyboard shortcuts

XCTU version 6.3.6.2 and higher works when the REPL is enabled. The MicroPython Terminal tool allows you to communicate with the MicroPython stack of your device through the serial interface. The MicroPython Terminal tool in XCTU supports the following control characters:

- **Ctrl+A**: Enter raw REPL mode. This is like a permanent paste mode, except that characters are not echoed back.
- **Ctrl+B**: Print the MicroPython banner. Leave raw mode and return to the regular REPL (also known as friendly REPL). Reprints the MicroPython banner followed by a REPL prompt.
- **Ctrl+C**: Regain control of the terminal. Interrupt the currently running program.
- **Ctrl+D**: Reboot the MicroPython REPL. Soft-reset MicroPython, clears the heap.
- **Ctrl+E**: Enter paste mode. Does not auto-indent and compiles pasted code all at once before execution. Uses a REPL prompt of `===`. Use **Ctrl-D** to compile uploaded code, or **Ctrl-C** to abort.
- **Ctrl+F**: Upload code to flash. Uses a REPL prompt of `^^^`. Use **Ctrl-D** to compile uploaded code, or **Ctrl-C** to abort.
- **Ctrl+R**: Run code in flash. Run code compiled in flash.

Note If PS is set to 1, code in flash automatically runs once at startup. Use **Ctrl-R** to re-run it.

Select a previously typed statement

You can use the UP and DOWN arrows on the keyboard to display a previously typed statement at the current MicroPython prompt.

Note This shortcut does not work: (1) while in paste mode (**Ctrl-E**) or on any code entered while in paste mode and (2) while in flash upload mode.

Arrow keys work to scroll back through previous commands, and to edit the current command. Some terminal emulators (like CoolTerm) might not work with scrollback.

1. Access the MicroPython environment.
2. At the MicroPython `>>>` prompt, type `print("statement 1")` and press Enter.
3. At the MicroPython `>>>` prompt, type `print("statement 2")` and press Enter.
4. At the MicroPython `>>>` prompt, type `print("statement 3")` and press Enter.
5. At the MicroPython `>>>` prompt, press the UP arrow key on the keyboard. The most recently typed statement displays at the prompt. In this example, the statement `print("statement 3")` displays.
6. You can press the UP arrow key on the keyboard to display the next most recently type statement, or press the DOWN arrow key on the keyboard to return the previously selected statement. Continue this process until the statement you want to use displays at the MicroPython `>>>` prompt. Use the Left and Right arrow keys and Backspace to make edits to the previous statement if desired.
7. Press Enter to execute the displayed statement.
Differences between MicroPython and other programming languages

You may have experience coding in another language, such as C or Java. You should be aware of the coding differences between other languages and MicroPython.

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory management</td>
<td>28</td>
</tr>
<tr>
<td>Variable types</td>
<td>28</td>
</tr>
<tr>
<td>Syntax</td>
<td>28</td>
</tr>
</tbody>
</table>
Memory management

In C, memory has to be allocated by the user for a variable or object before it can be used. For a variable in C, this is done by a declaration statement as shown in the code below. The first 2 lines create a floating-point (decimal-valued real number) type variable named salary and an integer named x. The last 2 lines assign values to each of those variables.

```c
float salary;
int x;

x = 9;
salary = 3.0 + x;
```

In MicroPython, a variable does not need to be declared before it can be used. For example, the MicroPython code shown below does the same thing as the C code shown in the example above. Each line does multiple things: creates the variable (the name), assigns it a type based on the assigned value, determines the space it needs in memory and allocates that space, and then assigns the value to it.

**Note** You can copy and paste code from the online version of the Digi MicroPython Programming Guide. Use caution with the PDF version, as it may not maintain essential indentations.

```python
x = 9
salary = x + 3.0
```

Variable types

In C, variables are "statically typed", meaning they are a certain type when they are created, and the type does not change. This also means the variable can only hold data appropriate for the type. In the C code sample shown below, an integer type variable named my_variable is created. An integer type variable can only hold integer values and the amount of memory allocated to this variable for storing its value is a fixed size- 4 bytes, limiting the range of values to -2,147,483,648 to 2,147,483,647 for a signed integer.

```c
int my_variable;
my_variable = 32;
```

In MicroPython, variables are dynamically (or automatically) assigned a variable type when the user assigns a value to the variable. In the code shown below, the variable big_number is assigned an integer type, allocated the appropriate amount of memory, and the value stored after the user assigns a value to the variable.

```python
big_number = 99999999999999999999
```

If a user changes the value of the variable to a text string, MicroPython stores the string and automatically changes the variable type to string.

```python
big_number = "This is a really big number!"
```

Syntax

Syntax refers to rules that you must follow when programming. The following sections explain the differences in syntax between MicroPython and other programming languages.
Curly braces and indentation

In C, a function or conditional statement is enclosed by curly braces, as shown in the code sample below.

```c
void action1(void) {
    printf("Function action1\n");
}

void action2(void) {
    printf("Function action2\n");
}

if condition {
    action1();
} else {
    action2();
}
```

In MicroPython, only a colon is required. Any statements that are part of the function must be indented. The C code sample shown above would be coded in MicroPython as shown below. After the function definitions and conditionals, the code to be executed is indented to make it a part of that block. Indentation is used in MicroPython to tell the compiler which lines are members of a certain structure.

```python
def action1():
    print("Function action1")

def action2():
    print("Function action2")

if condition:
    action1()
else:
    action2()
```

In C, all of the instructions to be executed for the function `some_function()` are contained within the curly braces. Most programmers indent all the instructions within the function for readability, but this is not required for the code to work.

```c
void some_function(void) {
    int x;
    x = 7;
    x = x + 1;
    printf("Incremented x\n");
    x = x + 2;
    printf("Incremented x by 2\n");
}
```

In MicroPython, indentation is required to tell the compiler what lines of code are to be executed for the function `some_function()`, as shown in the example below.
def some_function():
    x = 7
    x = x + 1
    print("Incremented x!")
    x = x + 2
    print("Incremented x by 2!")

When nesting conditions and functions, C relies on curly braces, as shown in the example below. Each level of code is indented to make it more readable, but it is not required for the code to run.

```c
void some_other_function(void) {
    if (condition) {
        do_something();
    }
}
```

In MicroPython, indentation is the only thing telling the compiler what instructions belong to what function or condition. The nested C code example shown above is coded in MicroPython in the example below:

```python
def some_other_function():
    if condition:
        do_something()
```

**Semicolons**

Statements in C are followed by a semicolon, as shown in the example below.

```c
int x;
x = 7 + 3;
action1();
```

In MicroPython, statements are ended by starting a new line. A special symbol or character is not needed.

```python
x = 7 + 3
action1()
```

**Increment operator**

C and Java have an "increment" operator, which lets the user increase the value of a variable by 1. See the following example:

```c
int x;
x = 1;
x++; // x is now 2
x++; // x is now 3
```

MicroPython does not have an "increment" operator. To do the equivalent in MicroPython the variable would have to have 1 explicitly added to it, or use the `+=` operator. The `+=` operator states that a variable equals itself plus a value. So, in the MicroPython code block below, line 3 is basically shorthand for line 2. They both do the same operation: set `x` equal to `x` plus 1.
Differences between MicroPython and other programming languages

Syntax

```plaintext
x = 1
x = x + 1  # x is now 2
x += 1    # x is now 3
```

Logical operators

In C, the logical operators AND, OR, and NOT are represented by `&&`, `||`, and `!` respectively. The C code block below shows the logical operators in use.

```c
// if it's sunny out, AND NOT cold outside
if (sunny_outside && !cold_outside) {
    // if you have a towel AND an umbrella
    if (have_towel && have_umbrella) {
        // if you have a bike OR a car
        if (have_bike || have_car) {
            // then you will go to the beach
            go_to_beach();
        }
    }
}
```

In MicroPython, the operators for AND, OR, and NOT are simply `and`, `or`, and `not`, which is much more intuitive. The MicroPython code shown below has the same function as the C code shown above.

```python
if sunny_outside and not cold_outside:
    if have_towel and have_umbrella:
        if have_bike or have_car:
            go_to_beach()
```
Develop applications on an XBee device

Space allocated to MicroPython ................................................................. 33
Code storage ......................................................................................... 33
How to organize your code ................................................................. 34
Run code at startup .............................................................................. 34
Monitor memory usage ....................................................................... 35
Efficient coding .................................................................................. 38
Application evolution .......................................................................... 38
Digi modified slicing for bytes and strings operations ...................... 41
Space allocated to MicroPython

The XBee device allocates space in various locations for use by MicroPython.

- **Heap** (32 kB of RAM): Area used for variables, objects and modules imported from .py and .mpy files in the file system.
- **Stack** (4 kB of RAM): RAM used by the MicroPython interpreter/task running as part of the XBee firmware. If your function has tail recursion, try to rewrite it as a loop to reduce stack use.
- **File System**: Storage area for .py and .mpy files, along with SSL/TLS certificates and other data files. File system is managed using ATFS commands, the MicroPython os module, and XCTU.
- **Frozen/bundled .mpy files** (32 kB of device flash): Storage area for compiled modules that can execute in place. Standard MicroPython builds for other hardware (like the pyboard) refer to these as "frozen" .mpy files but only support embedding them into the firmware at compile-time. The XBee device adds an os.bundle() method to freeze multiple .mpy files into the device flash so they can execute in place with a minimal impact on heap.

**Note** On XBee 3 Cellular devices with firmware ending in *15 or newer, the MicroPython heap has been increased to 64 kB and the MicroPython stack has been increased to 6 kB of RAM.

Code storage

The XBee device stores code in different formats.

**Built-in modules embedded in XBee firmware (device flash)**

Many of the modules you import into your program are actually implemented in compiled C code that exists as part of the MicroPython interpreter embedded in the XBee firmware and stored on the XBee device's flash. These modules only use heap space for variables and any objects you instantiate, like a `machine.Pin()` or `network.Cellular()` object.

**Source code in .py files (file system)**

You can create MicroPython modules and store them as .py files on the file system of the XBee device's SPI flash. Upload the modules over the serial port via YMODEM protocol using XCTU or a standard terminal emulator. When you import one of these files, MicroPython has to parse and compile it to a form that it can execute from the heap.

**Parsed and compiled code in .mpy files (file system)**

Parsing and compiling MicroPython source code requires heap space, and larger programs require more space than is available on the XBee device. XBee devices include the `os.compile()` method for compiling a .py file into a .mpy file. The maximum size for compiling a .py file on the device depends on its contents, but you may run out of memory trying to compile a 13 kB or larger file. In those cases, you can use mpy-cross on a PC (Mac, Linux, Windows) to pre-compile your source code and upload the resulting .mpy file instead.
Executable code on MicroPython heap (device RAM)
When you enter code in the REPL or import a module from the file system (a .py or .mpy file), MicroPython places it in the heap where it can execute in place. See documentation for the gc and micropython modules for methods to report on heap memory usage.

Compiled modules relocated from file system to device flash
Use os.bundle() to freeze/embed multiple .mpy files to an area of the XBee device's internal flash where they can execute in place. This can free up heap space for use by the running program.

---

Note Main code (bundled) in flash will not be run on startup or when CTRL-R is issued in the REPL. See Run code at startup for more information.

---

How to organize your code
To create the lib directory, format your device. The main execution program is always called main.py and should be located in /flash.

The modules and libraries you import should be located in /flash or /flash/lib.

You can load files using XCTU, any YMODEM compliant client or using PyCharm. PyCharm is the most user friendly for developers.

When you manually load files onto an XBee3 device for the first time, the MicroPython interpreter prompts you to format the file system. XCTU formats the file system using the AT FS FORMAT command, and then you can download the files to the device using XCTU or the tools mentioned above.

Run code at startup

boot.py
Unconditionally at boot, the XBee device automatically runs /flash/boot.py or /flash/boot.mpy (in that order) when the device powers up. It also tries to run that code after a soft reboot—for example, via CTRL-D in the "friendly" REPL but not the "raw" REPL, or calling machine.soft_reset() in your code. If any error is raised, boot.py exits immediately and main.py will not run even if the PS (Python Startup) command = 1.

boot.py could be useful to redirect the REPL to the secondary UART or to the Device Cloud console. See the cloud monitor sample for more information.

main.py
If you configure the PS (Python Startup) command = 1, the XBee device automatically tries to run /flash/main.py or /flash/main.mpy (in that order) when the XBee device powers up or resets. It also tries to run that code after a soft reboot—for example, via CTRL-D in the "friendly" REPL but not the "raw" REPL, or calling machine.soft_reset() in your code. During development, you can use CTRL-R to run the code as often as you'd like (for testing purposes), but if you replace /flash/main.py or /flash/main.mpy using a method other than Flash Upload Mode (for example, YMODEM upload), you will have to reset the REPL for it to reload code from those files. Each time you press CTRL-R it tells you if you are loading new code—and whether it is using main.py or main.mpy—or just running the same code as the last time you pressed CTRL-R.
Monitor memory usage

MicroPython provides various tools you can use to monitor memory usage in the heap (RAM allocated for MicroPython's use).

- The `gc` module
- The `micropython` module

The gc module

You can import `gc` for tools to initiate garbage collection (deletion of objects on the heap no longer in use) and measure heap usage. Use `gc.mem_free()` and `gc.mem_alloc()` for counts of available and used memory. The two values should always add up to the same number. Due to the overhead required by heap management, the 32 kB heap (32,768 bytes) only has 32,000 bytes available for allocation.

Use `gc.collect()` to force garbage collection of unreferenced objects in the heap. You should always do this before calling `gc.mem_free()` or `gc.mem_alloc()` in order to get an accurate value, or between successive calls to see how much space was released.

```python
>>> import gc
>>> gc.mem_free()
31232
>>> gc.mem_alloc()
896
>>> gc.mem_free() + gc.mem_alloc()
32000
>>> gc.collect()
>>> gc.mem_free()
31472
```

The micropython module

You can import `micropython` to get detailed information on heap memory usage, beyond the summaries provided by `gc.mem_free()` and `gc.mem_alloc()`.

`micropython.mem_info()`

Calling `mem_info()` without any parameters prints a summary of heap usage. Calling it with a parameter—for example, `micropython.mem_info(1)`—adds a detailed report of memory usage on the heap. Each line of the report starts with a memory offset into the heap, and then 64 characters representing 16-byte blocks with the following meanings:
Develop applications on an XBee device

Monitor memory usage

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>unused (available) block</td>
</tr>
<tr>
<td>h</td>
<td>start (head) of an allocation (unknown content)</td>
</tr>
<tr>
<td>=</td>
<td>continuation of allocation</td>
</tr>
<tr>
<td>A</td>
<td>start of array or bytearray</td>
</tr>
<tr>
<td>B</td>
<td>start of function bytecode</td>
</tr>
<tr>
<td>D</td>
<td>start of dict</td>
</tr>
<tr>
<td>F</td>
<td>start of float</td>
</tr>
<tr>
<td>L</td>
<td>start of list</td>
</tr>
<tr>
<td>M</td>
<td>start of module</td>
</tr>
<tr>
<td>S</td>
<td>start of string or bytes</td>
</tr>
<tr>
<td>T</td>
<td>start of tuple</td>
</tr>
</tbody>
</table>

The example below shows heap usage before and after importing a module (*urequests*) stored as an mpy file on the XBee device.

```python
>>> import micropython
>>> micropython.mem_info()
stack: 596 out of 3584
GC: total: 32000, used: 688, free: 31312
No. of 1-blocks: 9, 2-blocks: 14, max blk sz: 3, max free sz: 1950
>>> micropython.mem_info(1)
stack: 596 out of 3584
GC: total: 32000, used: 688, free: 31312
No. of 1-blocks: 9, 2-blocks: 14, max blk sz: 3, max free sz: 1950
GC memory layout; from 20001d10:
00000: =h=Bhh=Bh=h=h=h=Bh=hh=h=h=...h=.......... (30 lines all free)
000c0: ...........
>>> import urequests
>>> micropython.mem_info(1)
stack: 596 out of 3584
GC: total: 32000, used: 5168, free: 26832
No. of 1-blocks: 63, 2-blocks: 52, max blk sz: 45, max free sz: 1192
GC memory layout; from 20001d10:
00000: =h=Bhh=Bh=h=h=Bh=hh=h=h=...h=.......... (18 lines all free)```
Develop applications on an XBee device

Monitor memory usage

07c00: ..............
>>> import gc
>>> gc.collect()
>>> micropython.mem_info(1)
stack: 596 out of 3584
GC: total: 32000, used: 3952, free: 28048
No. of 1-blocks: 57, 2-blocks: 27, max blk sz: 45, max free sz: 1192
GC memory layout; from 20001d10:
00000: h=Bhhhh=........=h=...=h=..MD................
00400: DDSSSh=..BBSBhBhBBBBh===h===..BBh=======h=====B==h====BSh=h=h=..
00800: ...h=........Sh.......hBh=h=........h=..........................
00c00: .................h=======...............................h=======.
01000: ===..............Sh=................................h=======....
01400: ...........................................h==........hh=.......
01800: .................................................h=.............
02000: ....h.............................................Sh=======..
02c00: ...h============================================h=====hShShShShS
03000: hh=hh=hh=hh=hh=hh=hh==hh====hh=hh==h........................
07c00: ..............

**micropython.qstr_info()**

MicroPython stores identifiers (the names of things in your code - variables, methods, classes, and so forth) in pools as "QSTR" objects. In doing so, it can reference the full QSTR in bytecode by using a 16-bit index into the pool. The XBee firmware has a static QSTR pool embedded in it with names of built-in modules and their identifiers. Any Python code that runs on the XBee device can reference those existing names in its compiled bytecode. New identifiers go into dynamic QSTR pools allocated in MicroPython's heap.

You can use the `qstr_info()` method to report on the contents of those allocated pools. Without a parameter, you will just see summary usage information. With a parameter, it prints the contents of each QSTR stored in the pool.

<table>
<thead>
<tr>
<th>Information reported by micropython.qstr_info()</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_pool</td>
</tr>
<tr>
<td>n_qstr</td>
</tr>
<tr>
<td>n_str_data_bytes</td>
</tr>
<tr>
<td>n_total_bytes</td>
</tr>
</tbody>
</table>

At the beginning of the following example, MicroPython has not allocated any QSTR pools. In importing a module (`urequests`) stored as an mpy file on the XBee device, MicroPython allocated two pools, totaling 50 strings of 464 bytes and using a total of 736 bytes of the heap.

```python
>>> import micropython
>>> micropython.qstr_info(1)
qstr pool: n_pool=0, n_qstr=0, n_str_data_bytes=0, n_total_bytes=0
>>> import urequests
>>> micropython.qstr_info(1)
qstr pool: n_pool=2, n_qstr=50, n_str_data_bytes=464, n_total_bytes=736
```
Efficient coding


Feel free to use docstrings (string literals used to document code) in your programs, as the parser will ignore them and they are not included in compiled code or the .mpy file generated from the .py source.

Application evolution

As you work on your MicroPython application, you will likely take portions of it though a series of versions as it evolves from incomplete code (undergoing active development and debugging) to feature-complete, debugged modules that rarely change. The following topics provide some techniques you will use along the way to creating a production-ready application. If you are not already familiar with the Python concept of modules, you can learn about them at https://docs.python.org/3/tutorial/modules.html.

One-liners in the REPL

If you just want to test the syntax of a few lines of code, experimenting in the REPL (and even a Python3 interpreter on your PC) can be a good place to start.

Short blocks in paste mode

If you are working on a multi-line sequence or a complete function, you might do so in an editor on your computer, copy it to your clipboard, press Ctrl+E in the MicroPython REPL, paste the code, and then press Ctrl+D for immediate execution.
Flash upload mode

Flash upload mode is similar to paste mode, but stores the compiled code so you can run it more than once or automatically run it at startup. Press Ctrl+F in the MicroPython REPL, paste the code, and then press Ctrl+D to compile it. It stores the compiled code in `/flash/main.mpy` and you can then run it by pressing Ctrl+R. Set `ATPS = 1` to automatically run that code at startup. Flash upload mode prompts you about changing the current `ATPS` value; you can press Enter to accept the default of leaving it unchanged.

Storing compiled code requires the file system be formatted first, if the file system is not formatted, then the following error is generated: **OSError: [Errno 7019] ENODEV**

You can use the following method to format the file system from within MicroPython:

```python
import os
os.format()
```

**Note** When uploading code through flash upload mode, `/flash/main.mpy` will be deleted if it already exists. On file systems that do not support deleting files (see Which features apply to my device?), the space used by the existing `/flash/main.mpy` file is not reclaimed. While developing using flash upload mode on these devices, you may have to reformat the device if it runs out of space.

Modules stored as .py files

When you have a collection of related functions, you will probably want to combine them into a module that you can import into your main program and other modules. If you are going through lots of revisions, it might be easiest to edit a .py file on your computer and then upload it to the XBee device using XCTU or another terminal program. If you have previously loaded the module in MicroPython with the import statement, you need to perform a soft-reboot (press Ctrl+D at a REPL prompt) or use the following method to delete the old module and re-import it:

```python
import sys
def reload(mod):
    mod_name = mod.__name__
    del sys.modules[mod_name]
    return __import__(mod_name)
```

After running that code, you can type `reload(foo)` at a REPL prompt to reload a module from `foo.py` or `foo.mpy`.

Compiled modules stored as .mpy files

At some point, you may not have enough space in the MicroPython heap to compile and load multiple modules. In that case, you can pre-compile each .py file to a .mpy file to reduce the memory requirements of an import statement. Use the `os.compile()` method to create a .mpy file on the XBee device itself, or install `mpy-cross` on your PC and do it there before uploading to the XBee device. With `mpy-cross`, you will have the added benefit of identifying syntax errors on your computer before spending time uploading the file to the device.

The `os.compile()` process prints memory usage information to help identify when you are reaching the limit of the XBee device’s heap. In the example below, you can see that the parsing of `urequests.py` requires 7696 bytes (8336 - 640). The compilation step converts the parsed Python source code to compiled bytecode, and is usually the most memory-intensive step of creating the mpy file. But once it is complete, garbage collection releases most of that temporarily allocated memory and you see just the 3248 bytes (3888 - 640) required for the compiled code.
The final step saves the compiled module to the file system, but as you can see from the final `gc.mem_alloc()` call, there is still 608 bytes (1248-640) of heap in use. This is from the QSTR pools created when parsing and compiling the code. Since QSTR pools are permanent, the only way to recover that memory is to perform a soft reboot of the MicroPython REPL using Ctrl+D.

```
>>> import os
>>> os.chdir('lib')
>>> import gc
>>> gc.collect()
>>> gc.mem_alloc()
640
>>> os.compile('urequests.py')
stack: 644 out of 3584
GC: total: 32000, used: 640, free: 31360
No. of 1-blocks: 11, 2-blocks: 6, max blk sz: 8, max free sz: 1909
Parsing urequests.py...
stack: 644 out of 3584
GC: total: 32000, used: 8336, free: 23664
No. of 1-blocks: 19, 2-blocks: 11, max blk sz: 89, max free sz: 1407
Compiling...
stack: 644 out of 3584
GC: total: 32000, used: 3888, free: 28112
No. of 1-blocks: 44, 2-blocks: 34, max blk sz: 45, max free sz: 1225
Saving urequests.mpy...
>>> gc.collect()
>>> gc.mem_alloc()
1248
```

**Compiled modules via Flash upload mode**

A quick way to compile a module without having to use YMODEM is to use Flash upload mode, which saves the pasted code as `/flash/main.mpy`, and then use `os.replace('/flash/main.mpy', 'flash/lib/foo.mpy')` to replace the old module `foo` compiled code. This can only be done on modules that support renaming files—see Which features apply to my device?.

**Compiled modules embedded in device flash**

You can maximize your application size by writing your code as modules, cross-compiling them on a PC, uploading to the XBee device and then using `os.bundle()` to freeze/embed them into the flash where they can run in-place, with minimal heap usage.

Call `os.bundle()` without any parameters to get a list of modules embedded in the flash. Call `os.bundle(None)` to erase the modules embedded in the flash.

```
MicroPython v1.9.4-803-g4b0a8eada-dirty on 2018-06-21; XBC LTE Cat 1 Verizon with EFM32G
Type "help()" for more information.
>>> import os
>>> os.bundle()
['urequests', 'umqtt/simple']
>>> os.bundle(None)
Erased bundled modules.
>>> os.bundle()
[]
```

Call `os.bundle('mod1.mpy', 'mod2.mpy', 'package/mod3.mpy')` to embed modules `mod1`, `mod2`, and `package.mod3`. When you import a module, MicroPython checks for an embedded/frozen version of it before looking to the file system.
Develop applications on an XBee device

Digi modified slicing for bytes and strings operations

---

MicroPython v1.9.4-803-g4b0a8eada-dirty on 2018-06-21; XBC LTE Cat 1 Verizon with EFM32G
Type "help()" for more information.

```python
>>> import os
>>> os.chdir('lib')
>>> os.bundle('urequests.mpy', 'umqtt/simple.mpy')
bundling urequests.mpy...2196 bytes of raw code
bundling umqtt/simple.mpy...2916 bytes of raw code
Used 72/371 QSTR entries.
stack: 844 out of 3584
GC: total: 32000, used: 12288, free: 19712
   No. of 1-blocks: 114, 2-blocks: 77, max blk sz: 45, max free sz: 775
```

Embedded 2 module(s) to 5851/31152 bytes of flash.
soft reboot

MicroPython v1.9.4-803-g4b0a8eada-dirty on 2018-06-21; XBC LTE Cat 1 Verizon with EFM32G
Type "help()" for more information.

```python
>>> os.bundle()
['urequests', 'umqtt/simple']
>>> import urequests
>>> import umqtt.simple
```

---

**Digi modified slicing for bytes and strings operations**

Upstream MicroPython only allows bytes and string objects to be sliced with a step size of one. Digi has modified these types to allow standard slicing syntax with other values, including negative.
Power management in MicroPython

Prevent sleep from MicroPython ................................................................. 43
Initiate sleep from MicroPython ................................................................. 44
Sleeping with AT commands ................................................................. 45
Idle a device from MicroPython ................................................................. 45
Prevent sleep from MicroPython

**Note** This section only applies to devices that support the **Power Management** feature.

When the XBee device enters sleep mode, any MicroPython code currently executing is suspended until the device comes out of sleep. When the XBee device comes out of sleep mode, MicroPython execution continues where it left off.

If you use SM sleep, MicroPython can use `XBee().wake_lock` to force the device to stay awake during critical operations, for example, when the device is configured for one of the ATSM sleep options (excluding SM = 6 MicroPython Sleep). The following example shows how to use the `XBee().wake_lock`:

**Note** `wake_lock` is a context manager. See **Context Manager Documentation** for more instructions on usage.

```python
import xbee
xb = xbee.XBee()

# do things interruptable by sleep
with xb.wake_lock:
    # do important things

# back to things that are safe to interrupt
```

**XBee Cellular Modem:**

Upon entering sleep mode, the XBee Cellular Modem closes any active TCP/UDP connections and turns off the cellular component. As a result, any sockets that were opened in MicroPython prior to sleep report as no longer being connected. This behavior appears the same as a typical socket disconnection event.

The following is a summary of the behavior to expect from the main socket methods:

- `socket.send` raises **OSError**: ENOTCONN
- `socket.recv` returns an empty string, the traditional end-of-file return value

**Note** As of the x09 firmware, all time-related APIs include the time spent in sleep. Prior firmware versions paused the millisecond timer used by `time.sleep()`, `time.sleep_ms()` and `time.time()`, so having a 15-second SM (Sleep Mode)-triggered sleep occur during a MicroPython `time.sleep(30)` would result in a 45 second delay in execution. With the x09 firmware, it only delays for 30 seconds.

**XBee 3 Zigbee RF Module, XBee 3 802.15.4 RF Module, XBee 3 DigiMesh RF Module description:**

Upon entering sleep mode the device shuts down all peripheral resources for the lowest possible current consumption, then upon device wake the peripheral resources are restored and the device continues with MicroPython code execution.
Initiate sleep from MicroPython

Note This section only applies to devices that support the Power Management feature.

XBee Cellular Modem:
If you disable sleep modes by setting SM (Sleep Mode) to 0, you can use XBee().sleep_now() and XBee().wake_reason() to control when the module sleeps. When selecting sleep and wake times on the XBee Cellular Modem, take into consideration the time it takes to close network connections and shut down the cellular connection before sleeping, and then to restore the connection when waking back up.

XBee 3 Zigbee RF Module, XBee 3 802.15.4 RF Module, XBee 3 DigiMesh RF Module:
When setting SM (Sleep Mode) to 6, you can use XBee().sleep_now() and XBee().wake_reason() to control when the device sleeps. The device sleeps for the time period programmed with an optional early pin wake (DTR, commissioning button, or SPI_SSEL).

sleep_now(timeout_ms, pin_wake=False)

Sleeps for timeout_ms milliseconds and then wakes.

- If pin_wake is set to True, the device will wake before timeout_ms if DIO8 transitions from high to low—that is on a falling edge of the line.
- If timeout_ms is None then pin_wake must be set to True and the device will sleep indefinitely until a falling edge on DIO8 occurs.
- If a timeout is specified—timeout_ms is not None—the function will return the actual elapsed time in milliseconds after the device wakes.
- Throws an EALREADY OSError exception if SM is not configured correctly for MicroPython to control sleep.

Note The sleep time reported includes code execution overhead—several milliseconds.

wake_reason()

Returns either xbee.RTC_WAKE if the full timeout_ms elapsed, or xbee.PIN_WAKE when enabled and DIO8 woke the device early.

The following example shows power management with MicroPython:

```python
from machine import Pin
import time
import xbee

def read_switch(iopin = None):
    if iopin.value() == 0:
        print("SW2 has been pressed!")
        return True
    return False

# Configure DIO0(SW2) to put module to sleep
```
Sleeping read

Sleeping

Digi

Power

MicroPython

a

management

puts

Use

Enter

be

Note

idle

transmissions,

Normally, a XBee device comes into the idle state, which affects its current draw and the device will not receive any RF transmissions.

Note This only affects the device’s primary RF interface—if Bluetooth is enabled, the device will still be able to communicate over Bluetooth and the current reduction will be less.

Enter and exit the idle state

Use the `xbee.idle_radio()` function to put the device into a forced idle state. `xbee.idle_now(True)` puts the device into this state, and `xbee.idle_now(False)` returns it to normal operation.
Transmit from an idle device

While the device is idle, `xbee.transmit()` can be used to send packets just like in normal operation. The device will be enabled just long enough to complete the transmission, then return to the idle state.

Send data to an idling device

Because a device that uses the `idle_radio()` feature is not always able to receive transmissions, you must take care if data needs to be sent to an idling device. Functionally, sending data to an idling device is equivalent to sending data to a sleeping device. See the sections on sleep and indirect messaging in the user guide for the protocol you use for more information on how to transmit to a sleeping device.

Poll for data while the device is idle

This section only applies to the XBee 3 Zigbee RF Module. See Which features apply to my device? for a list of the supported features.

The Zigbee protocol uses a polling system to deliver messages to sleeping—or idling—end devices. Messages for a sleeping end device are stored on a nearby router—the "parent"—and the sleeping end device periodically queries the router to see if any messages are available. When the device is idled, this querying must be triggered by the MicroPython application instead of occurring automatically.

In order to receive data while the device is idled, the MicroPython application must periodically call `xbee.poll_now()` to check for incoming data. Calling `poll_now()` causes the device to check for messages stored by its parent, and will retrieve a single message from the parent if any are available. `xbee.poll_now()` does not return the retrieved packet—the packet is instead passed to MicroPython via the normal method, either by the application calling `xbee.receive()` or through a callback if one is registered.

**Note** `xbee.poll_now()` is a non-blocking call, meaning it will return before the polling is complete. It can take approximately 10 ms after calling `poll_now()` before a message is available to `xbee.receive()`.

We recommend registering a callback for received packets when using `xbee.poll_now()`. That way, once the packet is retrieved from the parent it is immediately handled by the application.

Since `xbee.poll_now()` only retrieves a single packet from the parent, it may need to be called more than once if more than one message may be received between polls. One way to do so is by calling `xbee.poll_now()` again every time a packet is received, or in the receive callback.

Configure the parent

You must configure two parameters on the network—including non-sleeping devices—in order for messages to be delivered correctly on the network.

- **SP** determines how long the parent will hold on to messages for a sleeping device. The parent will hold on to messages for three times the value of **SP**. If the end device always calls `poll_now()` more frequently than this, no data will be dropped.

- **ET** must be set to at least the longest time the sleeping device will go between calls to `poll_now()`. This value is used as a network timeout; if the end device goes longer than this without polling, it is considered to have left the network and will need to rejoin the next time it polls.
**Example**

The following code shows some basic usage examples of the `idle_radio` and `poll_now` functions. A more detailed sample application can be found in the [xbee-micropython github repository](https://github.com).

```python
import xbee
import time

# Idle the radio
xbee.idle_radio(True)

# Define a callback for received packets
def rx_callback(packet):
    print(packet)
    xbee.poll_now()
    xbee.receive_callback(rx_callback)

# Send a transmission with the radio idled
xbee.transmit(xbee.ADDR_COORDINATOR, "payload", tx_options=1)

# With the radio idled, we have to call poll_now() occasionally or risk dropping data
while True:
    # Application code, etc. can go here
    time.sleep(1)
    # Check for new messages
    xbee.poll_now()
    # rx_callback() will be called if the poll comes back with a message
```

---

**Control MicroPython from Remote Manager**

*Note* This section only applies to the XBee Cellular Modem. See [Which features apply to my device?](#) for a list of the supported features.

---

**Run ATPY commands as a data service request**

The **ATPY** command has six options. These command options are supported in Digi Remote Manager using the new **mPyControl** service tag:

- **version**: MicroPython version
- **bundled**: Report on bundled bytecode
- **erase**: Erase bundled bytecode (equivalent to `os.bundle(None)`)  
- **reboot**: Soft reboot (equivalent to `CTRL-D` in the REPL)
- **execute**: Execute code stored in `main.{py,mpy}` (`CTRL-R` in REPL)
- **interrupt**: Send `KeyboardInterrupt`

The following is an example of the `version` device request:

```xml
<sci_request version="1.0">
<data_service>
<targets>
```
Control MicroPython from Remote Manager
Run ATPY commands as a data service request

Example response:

```xml
<sci_reply version="1.0">
<data_service>
<device id="00010000-00000000-03592051-00000000">
<requests>
<device_request target_name="mPyControl" status="0">MicroPython v1.18-1705-g4f1ec2455 on 2022-07-20</device_request>
</requests>
</device>
</data_service>
</sci_reply>
```
Access the primary UART

How to use the primary UART .......................................................... 51
Example: read bytes from the UART ............................................. 51
Example: read the first 15 bytes from the UART .......................... 52
How to use the primary UART

MicroPython provides access to the primary UART via `sys.stdin` (see `sys.stdin limitations`) and `sys.stdout` (and `sys.stderr` as an alias to `sys.stdout`). Unlike Python3, MicroPython does not allow overriding `stdin`, `stdout` and `stderr` with other stream objects.

`sys.stdin` is a standard input stream. `sys.stdin` supports the standard stream methods `read` and `readline` in text mode, converting carriage return (``) to newline (`
`).

**Note**: Do not use the `stdin` methods `readlines` or `readinto` because they will be removed in future firmware.

Use `sys.stdin.buffer` (instead of `sys.stdin`) for binary mode without any line ending conversions. The `read()` method takes a single, optional parameter of the number of bytes to read. For a positive value, `read()` blocks until receiving that many bytes from the standard stream methods primary UART. For non-blocking, call `read()` without the parameter (or with a negative value) and it returns whatever characters are available or `None` if no bytes are waiting.

`sys.stdout` supports the `write()` method in text mode, sending an additional carriage return (``) before each newline (`
`). Use `sys.stdout.buffer` (instead of `sys.stdout`) for binary mode without any line ending conversions. The `write()` method buffers its output, and can return before sending all bytes out on the UART.

**sys.stdin limitations**

Note that `sys.stdin` provides access to a filtered input stream with the following limitations:

- Only works as long as `ATAP = 4`.
- You can only configure the primary serial port via AT commands (for example `ATBD` to set the baud rate) and not from MicroPython.
- Receiving a Ctrl-C character generates a `KeyboardInterrupt`.
  - You can change the interrupt character using `micropython.kbd_intr(ch)` where `ch` is the new character to use (3 corresponds to Ctrl-C) or -1 to disable the keyboard interrupt entirely.
  - MicroPython always resets the keyboard interrupt to Ctrl-C at the start of each REPL line, before executing code entered via paste mode, and when executing compiled code at startup or via Ctrl-R.
- The escape sequence (configured with `ATCC`, +++ by default) protected by a guard time (configured with `ATGT`, 1 second by default) of no data before and after the escape sequence will always enter Command mode.
  - Escape sequence handling can cause delays when reading from `sys.stdin`.
  - You can send `ATPY` in Command mode to force a `KeyboardInterrupt`, even if it was disabled via `micropython.kbd_intr(-1)`.

**Example: read bytes from the UART**

The following example reads bytes from the UART and prints it out one at a time until a keyboard interrupt occurs when you press Ctrl+C.

```python
from sys import stdin, stdout

```
Access the primary UART

Example: read the first 15 bytes from the UART

```python
while True:
    data = stdin.buffer.read(1)
```

Example: read the first 15 bytes from the UART

The following example reads the first 15 bytes from the UART and prints it out one at a time. Notice that keyboard interrupts are disabled.

```python
import micropython
from sys import stdin, stdout

interrupt_char = -1
micropython.kbd_intr(interrupt_char)

for _ in range(15):
    data = stdin.buffer.read(1)
    stdout.buffer.write(data)
```
REPL (Read-Evaluate-Print Loop) examples

A REPL is a language shell that accepts user input, evaluates the input, and then returns a result.
This section contains examples of specific MicroPython REPL commands on the XBee device. For
information about MicroPython REPL rules in general, see

Ctrl+A: Enter raw REPL mode ........................................................................54
Ctrl+B: Print the MicroPython banner ...................................................................54
Ctrl+C: Regain control of the terminal .................................................................56
Ctrl+D: Reboot the MicroPython REPL .................................................................57
Ctrl+E: Enter paste mode ....................................................................................57
Ctrl+F: Upload code to flash ............................................................................58
Flash memory and automatic code execution ....................................................60
Perform a soft-reset or reboot .........................................................................63
### Ctrl+A: Enter raw REPL mode

Use this command to enter raw REPL mode, which enables you to execute pasted code. This acts like a paste mode, but the characters are not echoed back.

This command is used for machine-to-machine communication.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

1. **Access the MicroPython environment.**
2. **Copy** the code you want to paste into the XBee device. For example:
   ```python
   print("Hello world")
   ```
3. **Press Ctrl+A** to enter raw REPL mode.

```
MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.
>>> raw REPL; Ctrl-B to exit
> OK
```

4. **Right-click** at the MicroPython > prompt and select the **Paste** option.
5. **Press Ctrl+D** to save the paste action. An "OK" confirmation and the pasted code displays in the line. The code is saved to the XBee device and immediately executed.

```
MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.
>>> raw REPL; Ctrl-B to exit
> OK Hello world
```

6. **Press Ctrl+B** to exit raw REPL mode.

```
MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.
>>> raw REPL; Ctrl-B to exit
> OK Hello world
> MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.
>>> Ok
```

### Ctrl+B: Print the MicroPython banner

Use this command to perform one of the following:

- If MicroPython is in raw REPL mode, press **Ctrl+B** to return to the regular REPL and print the MicroPython banner.
- If MicroPython is in the regular REPL mode, press **Ctrl+B** to print the banner.

The banner displays the MicroPython version you are using and the build date for that version. Pressing **Ctrl+B** does not reboot the REPL. If you need start a fresh REPL session, use the **Ctrl+D: Reboot the MicroPython REPL** command to reboot the REPL.
Print the banner
This example shows how to print the banner.

1. Access the MicroPython environment.
2. Press Ctrl+B to print the banner.

MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.

Print the banner and verify that the memory was not wiped
In this example, a variable "a" is assigned the value "test". When you press Ctrl+B, the banner is printed.

You can verify that the memory was not wiped by entering the variable "a" and seeing that the value "test" is the output.

1. Access the MicroPython environment.
2. At the MicroPython >>> prompt, type a = "test", then press Enter. This statement assigns the value "test" to the variable "a".
3. At the MicroPython >>> prompt, type a, then press Enter. The value assigned to the variable displays.
4. Press Ctrl+B to print the banner.
5. At the MicroPython >>> prompt, type a and press Enter. The assigned value for the variable is returned.

Note You can copy and paste code from the online version of the Digi MicroPython Programming Guide. Use caution with the PDF version, as it may not maintain essential indentations.
Ctrl+C: Regain control of the terminal

Use this command to interrupt the currently running program and regain control of the terminal. This is useful if running the code is taking longer than expected, such as if the code has an incorrectly coded never-ending loop.

In this example the code has an infinite loop. The code stops the code execution.

Note You can copy and paste code from the online version of the Digi MicroPython Programming Guide. Use caution with the PDF version, as it may not maintain essential indentations.

1. Access the MicroPython environment.
2. Copy the code you want to paste. This example uses the following code:

   ```python
   while True:
       pass  # This statement means "do nothing"
   ```

3. At the MicroPython >>> prompt, type Ctrl+E to enter paste mode. The terminal displays paste mode; Ctrl-C to cancel, Ctrl-D to finish.
4. At the MicroPython >>> prompt, right-click and the select the Paste option. The code appears in the terminal and each line is numbered, followed by ===. For example line 1 starts with 1===.
5. Press Ctrl+D to accept and run the pasted code. The code will run continuously until you cancel it.
6. Press Ctrl+C to stop the code execution. A KeyboardInterrupt exception message prints to the screen.
**Ctrl+D: Reboot the MicroPython REPL**

Use this command to reboot the REPL and clear any variable and function definitions.

1. Access the MicroPython environment.
2. At the MicroPython >>> prompt, type `a = "test"`, then press Enter. This statement assigns the value "test" to the variable "a".
3. At the MicroPython >>> prompt, type `a`, then press Enter. The value assigned to the variable displays.
4. Press Ctrl+D to reboot the REPL. The phrase "soft reboot" followed by the MicroPython banner prints.
5. At the MicroPython >>> prompt, type the variable "a" (no quotes) and press Enter. Since the memory was wiped, the variable is not found and the error NameError: name not defined prints in the output.

```
MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.
>>> a = 'test'
>>> a
'test'
>>> soft reboot
MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name not defined
```

**Ctrl+E: Enter paste mode**

Use this REPL command to enter paste mode. This enables you to paste a block of code into the terminal, rather than having to type in lines of code.

**Note** Paste mode evaluates each line in the pasted code block in order, as if the code had been typed into the REPL.

**Paste one line of code**

This example uses the following code to show how to copy one line of code and paste it into the MicroPython Terminal.

1. Access the MicroPython environment.
2. Copy the code you want to paste. This example uses the following code:

   ```python
   print("Hello world")
   ```

3. At the MicroPython >>> prompt type Ctrl+E to enter paste mode. The terminal displays paste mode; Ctrl-C to cancel, Ctrl-D to finish.
**REPL (Read-Evaluate-Print Loop) examples**

MicroPython v1.9.3-999-g000000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.

```python
>>> paste mode; Ctrl-C to cancel, Ctrl-D to finish
1===
```

4. At the MicroPython >>> prompt, right-click and select the **Paste** option.
5. The code appears in the terminal and each line is numbered, followed by ===. For example line 1 starts with 1===.
6. Press **Ctrl+D** to complete the paste process and run the pasted code.

---

**Paste a code segment**

This example uses the following code to show how to copy one line of code and paste it into the MicroPython Terminal.

1. **Access the MicroPython environment.**
2. **Copy the code you want to paste.** This example uses the following code:

```python
for x in range(10):
    print("Current number: %d" % x)
    if (x < 9):
        print("Next number will be: %d\n" % (x + 1))
    else:
        print("This is the last number!")
```

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

3. At the MicroPython >>> prompt type **Ctrl+E** to enter paste mode. The terminal displays **paste mode; Ctrl-C to cancel, Ctrl-D to finish**.
4. At the MicroPython >>> prompt, right-click and select the **Paste** option.
5. The code appears in the terminal and each line is numbered, followed by ===. For example line 1 starts with 1===.
6. Press **Ctrl+D** to complete the paste process and run the pasted code. In this example, you should see 10 statements print to the terminal that state the current number, and what the next number will be. The numbers are from 0 to 9.

---

**Ctrl+F: Upload code to flash**

You can use flash mode to paste a block of code into MicroPython and store it to flash memory. You can run the stored code at any time from the MicroPython prompt by pressing **Ctrl+R**.

---

*Digi MicroPython Programming Guide* 58
When the code is uploaded to the flash memory, the MicroPython volatile memory (RAM) is cleared of any previously executed code. The uploaded code is saved on the XBee device. This means that only the last code saved to the flash memory is available.

You can choose to automatically run the code currently stored in the flash memory when the XBee device boots up.

**Load code to flash memory**

Use this command to upload code to the flash compile mode.

Any code uploaded in the flash memory can be set to run automatically when the XBee Cellular Modem boots up. You can also press Ctrl+R to re-run the compiled code at any time.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

1. **Access the MicroPython environment.**
2. **Copy the code you want to paste into the XBee device.** For example:

   ```
   print("Hello world")
   ```
3. **Press Ctrl+F.**

   ```
   MicroPython v1.9.3-999-g000000000 on 2018-01-01; XBee Module with EFX32
   Type "help()" for more information.
   >>>
   flash compile mode; Ctrl-C to cancel, Ctrl-D to finish
   1^^^
   ```
4. **At the MicroPython 1^^ prompt, right-click and select the Paste option.**

   ```
   MicroPython v1.9.3-999-g000000000 on 2018-01-01; XBee Module with EFX32
   Type "help()" for more information.
   >>>
   flash compile mode; Ctrl-C to cancel, Ctrl-D to finish
   1^^^ print("Hello world")
   ```
5. **Press Ctrl+D to finish.** The code is compiled and stored in flash memory.

   ```
   Compiling 123 bytes of code...
   Used 0/150 QSTR entries.
   Compiled 123 bytes of code to 188/7544 bytes of flash.
   Automatically run this code at startup [Y/n]? 
   ```

   **Note** The compilation report includes the number of used/available QSTR entries. The QSTR pool is used to store string literals from uploaded code. If a piece of code contains too many string literals, compilation fails and reports a QSTR pool overflow.

6. **You can choose whether to have the code stored in the flash memory automatically run the next time the XBee device is started.** Press Enter to leave the setting unchanged (the default value shown as uppercase).

   − **Y**: Press Y to automatically run the code stored in flash memory upon startup. This sets the PS command to 1. Note that this example only works on startup if you have a
terminal open on that serial port and the **AP** command is set to 4.

- **N**: Press **N** to ensure that the code stored in flash memory is not run the next time the XBee device is started. This sets the **PS** command to 0.

### Erase the code stored in flash memory
You can erase the code stored in flash memory using one of the following methods.

**Note** This example assumes you have code stored to flash memory. For information about how to store code to flash memory, see Load code to flash memory.

**Ctrl+D**

1. Access the MicroPython environment.
2. At the MicroPython >>> prompt, press **Ctrl+F** to enter flash mode. Do not enter or paste any code.
3. At the MicroPython >>> prompt, press **Ctrl+D** to complete the process. A process message displays:
   
   Erasing stored code...
4. When the process is complete the MicroPython >>> prompt displays in the terminal.

**ATPYD command**
The **ATPYD** command erases stored code and performs a soft reboot. For instructions, see the MicroPython commands section in the appropriate user guide.

### Flash memory and automatic code execution

Flash memory is referred to as "non-volatile" memory, as it retains whatever is stored in it, even without any power. This allows code stored in the flash memory to be run when you start up the XBee device.

The sections below explain how to manage code stored in flash memory.

- Run stored code at start-up to flash LEDs (XBee Cellular Modem only)
- Disable code from running at start up
- Enable code to run at start-up

### Run stored code at start-up to flash LEDs

**Note** This section only applies to the XBee Cellular Modem. See Which features apply to my device? for a list of the supported features.

If you have stored code to the flash memory, you can choose to automatically run this code when the XBee device boots up.

1. Access the MicroPython environment.
2. Copy the code you want to paste. This example uses the following code, which automatically blinks the LED lights on the XBIB board every two seconds.
from machine import Pin
import time
dio10 = Pin("P0", Pin.OUT, value=0)
while True:
    time.sleep(1)
    dio10.toggle()  # Flash the LED on DIO10 (P0)

3. At the MicroPython >>> prompt, press Ctrl+F.
4. At the MicroPython 1^^^ prompt, right-click and select the Paste option.
5. The code appears in the terminal and each line is numbered, followed by ^^^. For example, line 1 starts with 1^^^.
6. Press Ctrl+D to finish.

Compiling 123 bytes of code...
Used 0/150 QSTR entries.
Compiled 123 bytes of code to 188/7544 bytes of flash.
Automatically run this code at startup [Y/n]?

7. Press the Y key to run the code at start-up.
8. You may want to test your code before power cycling the device.
9. Press Ctrl+R to run the code compiled in flash. If it is not working correctly, press Ctrl+C to interrupt it and upload a new version.
10. Once you are happy with the uploaded code, power down the XBee Cellular Modem.
    a. Unplug the USB cable from your computer.
    b. Disconnect the power supply from the XBIB board.
    c. Wait until the lights on the XBIB board turn off.
    d. Reconnect the power. The three LEDs on the XBIB board automatically start turning ON and OFF every 2 seconds.
11. Connect the USB cable to your computer.
12. Access the MicroPython environment. A MicroPython prompt does not display, as MicroPython is running the code to blink the LEDs.
13. The terminal seems unresponsive as the code loop executes. Note the three green LEDs to the right of the USB-B port on the XBIB development board. These LEDs turn ON then OFF every 2 seconds.
14. At the terminal, press Ctrl+C to stop code execution and regain control of the terminal. A MicroPython prompt displays and the LEDs stop flashing.

Disable code from running at start up

For code that you saved to the flash memory and specified that the code should run at start up, you can change your choice and choose not to automatically run the code at start up. You can change your choice without saving the code to the flash memory again.

1. Use Ctrl+F to save code to the flash memory and choose to run it at start up.
2. At the Serial Console, enter Command mode by sending +++ and receiving an OK response.
3. At the prompt, type **ATPS** and press **Enter**. The terminal should echo back 1, since the code in the flash memory is set to run at start up.

4. At the prompt, type **ATPS0** and press **Enter**. This statement disables automatic code execution at start up.

5. At the prompt, type **ATWR** and press **Enter**. This statement writes the change to the flash memory.

6. At the prompt, type **ATCN** and press **Enter**. This statement exits Command mode.

7. Disconnect the USB cable from your computer.

8. Close the Serial Console.

9. Disconnect the power from the XBIB board.

10. After the LEDs on the XBIB board have all turned off, reconnect the power to the XBIB board.

11. Connect the USB cable to your computer. Notice that the LEDs do not blink, which verifies that you have successfully disabled the automatic code execution at start up.

### Ctrl+R: Run code in flash

You can use this command to re-run the code in the flash memory.

1. Access the MicroPython environment.

2. Load code to flash memory.

3. Press **Ctrl+R** to re-run the code in the flash memory.

---

```plaintext
MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.
>>> Running 76 bytes of stored bytecode...
Hello world
```

### Enable code to run at start-up

For code that you saved to the flash memory and chosen not to run at start up, you can change your choice and enable the code to automatically run at start up. You can change your choice without saving the code to the flash memory again.

1. For this example, you need code stored in flash memory that will not automatically run at start-up. Use **Ctrl+F** to save code to the flash memory. You can either:
   - Press **N** and choose not to run it at start up.
   - Press **Y** to run the code in flash memory at start-up. If you chose **Yes**, for this example you should **Disable code from running at start up**.

Remember that in this example, when MicroPython is not set to automatically run at start-up, the LEDs do not blink on module start-up.

2. At the Serial Console, enter Command mode by sending +++ and receiving an **OK** response.

3. At the prompt, type **ATPS** and press **Enter**. The terminal should echo back 0, since the code in the flash memory is not set to run at start-up.
4. At the prompt, type **ATPS1** and press **Enter**. This statement enables automatic code execution at start up.

5. At the prompt, type **ATWR** and press **Enter**. This statement writes the change from the previous statement to the flash memory.

6. At the prompt, type **ATCN** and press **Enter**. This statement exits command mode.

7. Press the **Reset** button on the XBIB board.

8. Notice that the LEDs blink ON and OFF, which verifies that you have successfully enabled the automatic code execution at start up.

### Perform a soft-reset or reboot

If you want to soft-reset the REPL you can press **Ctrl+D** in the REPL, or run `machine.soft_reset()` to force a soft reset from code. If you want to reboot the entire XBee device:

- On Cellular devices, you can perform a safe shutdown and reboot using `network.Cellular().shutdown(reset=True)`
- On all devices, you can reboot the XBee using `xbee.atcmd('FR')` or `machine.reset()`.

*Note FR and `machine.reset` force an immediate reset of the XBee, and on Cellular should not be used unless preceded by a safe shutdown.*
Access file system in MicroPython

Note This section only applies to devices that support the File System feature.

Directory and file names follow the rules in Paths.

Modify file system contents ................................................................. 65
Access data in files ........................................................................... 67
File object methods ......................................................................... 67
Import modules from file system ...................................................... 68
Reload a module .............................................................................. 69
Compiled MicroPython files ............................................................... 69
Modify file system contents

The uos module contains the following methods to interact with the file system.

**uos.chdir(dir)**
Change the current working directory.

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

*Note* MicroPython maintains a separate working directory from the FS (File System) command processor.

**uos.listdir([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**: A string (or bytes if dir is a bytes object) and it is the name of the entry.
- **type**: An integer that specifies the type of the entry, with 0x4000 for directories and 0x8000 for regular files.
- **inode**: An integer corresponding to the inode of the file. On XBee devices, set to 0 for regular files and directories and -1 for secure files.
- **size**: An integer representing the size of the file or -1 if unknown. Its meaning is currently undefined for directory entries.

**uos.listdir([dir])**
Returns a list of files in the given directory. With no argument it uses the current working directory (.)

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

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

**uos.rmdir(dir)**
Remove a directory. Fails if dir is not empty.

**uos.rename(old_path, new_path)**
Rename or move a file or directory. Fails if new_path already exists.

*Note* This function is only available on modules that support renaming files.
uos.replace(old_path, new_path)
Replace a file or directory (new_path) with another (old_path).

**Note** This function is only available on modules that support renaming files.

uos.sync()
Sync all file systems.

uos.compile(source_file, mpy_file=None)
This is an XBee extension to uos. Compile Python source code in source_file and store in a file with an .mpy extension. Default is to remove the .py extension from source_file and append .mpy to generate mpy_file. See Import modules from file system for details on using .mpy files. Compilation involves three steps: parsing, compiling and saving to the file system. MicroPython prints information about heap usage before each step so you can monitor heap requirements for a device, and consider splitting it into two (or more) modules or compiling with the MicroPython cross compiler (mpy-cross) on your computer instead of compiling on the XBee device.

```python
>>> uos.compile('urequests.py')
stack: 644 out of 3584
GC: total: 32000, used: 688, free: 31312
  No. of 1-blocks: 12, 2-blocks: 7, max blk sz: 8, max free sz: 1716
Parsing urequests.py...
stack: 644 out of 3584
GC: total: 32000, used: 8000, free: 24000
  No. of 1-blocks: 20, 2-blocks: 12, max blk sz: 88, max free sz: 1415
Compiling...
stack: 644 out of 3584
GC: total: 32000, used: 3872, free: 28128
  No. of 1-blocks: 45, 2-blocks: 35, max blk sz: 42, max free sz: 1254
Saving urequests.mpy...
>>> list(uos.ilstdir())
[('urequests.py', 32768, 0, 3407), ('urequests.mpy', 32768, 0, 2657)]
```

uos.format()
This is an XBee extension to uos. Reformats the SPI flash and creates the default directory structure.

uos.hash([secure_file])
This is an XBee extension to uos. Returns a 32-byte bytes object with the sha256 hash digest of a secure file. You can use this value to verify that a secure file matches an unencrypted copy of the file. See FS HASH filename for more information on using this digest. If secure_file is not specified, it returns a string identifying the hash method (sha256). You can convert the 32-byte digest to a 64-character hexdigest with the following code snippet:

```python
>>> from ubinascii import hexlify
>>> digest = os.hash('cert/client.key')
>>> hexdigest = hexlify(digest)
>>> digest
```
Access data in files

The built-in method `open()` is an alias to `uio.open(file, mode='r')` which returns a file object—an `uio.FileIO` object for binary modes and an `uio.TextIOWrapper` object for text modes. If the file cannot be opened, an `OSError` is raised.

Parameter `file` is a path-like object giving the path—absolute or relative to the current working directory—of the file to be opened.

Parameter `mode` is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), `'x'` for exclusive creation and `'a'` for appending—all writes append to the end of the file regardless of the current seek position. The available modes are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'r'</code></td>
<td>Open for reading (default).</td>
</tr>
<tr>
<td><code>'w'</code></td>
<td>Open for writing, truncating file file. On modules that do not support editing files after creation, this will fail if the file already exists.</td>
</tr>
<tr>
<td><code>'x'</code></td>
<td>Open for exclusive creation, failing if the file already exists.</td>
</tr>
<tr>
<td><code>'a'</code></td>
<td>Open for writing, always appending to the end of the file. Only available on modules that support editing files after creation.</td>
</tr>
<tr>
<td><code>'b'</code></td>
<td>Binary mode.</td>
</tr>
<tr>
<td><code>'t'</code></td>
<td>Text mode (default).</td>
</tr>
<tr>
<td><code>'r+'</code></td>
<td>Open a disk file for updating (reading and writing). Only available on modules that support editing files after creation.</td>
</tr>
<tr>
<td><code>'*'</code></td>
<td>(XBee extension) open a secure file for writing. Only available on modules that support secure files.</td>
</tr>
</tbody>
</table>

The default mode is `'r'`—open for reading text, a synonym of `'rt'`. For binary read-write access, the mode `'w+b'` opens and truncates the file to 0 bytes. `'r+b'` opens the file without truncation.

Python distinguishes between binary and text I/O. Files opened in binary mode—including `'b'` in the mode argument—return contents as bytes objects without any decoding. In text mode—the default, or when `'t'` is included in the mode argument—the contents of the file are returned as `str`.

File object methods

The following methods interact with file objects.

```python
read(size=-1)
```

Read up to `size` bytes from the object and return them. As a convenience, if `size` is unspecified or `-1`, all bytes until end-of-file (EOF) are returned.
readinto(b)
Read bytes into a pre-allocated, writable bytes-like object b, and return the number of bytes read.

readline(size=-1)
Read and return one line from the stream. If size is specified, at most size bytes are read.

readlines()
Read and return a list of lines from the stream. MicroPython does not support Python3’s hint parameter.

Note It is already possible to iterate on file objects using for line in file: ... without calling file.readlines().

write(b)
Write the given bytes-like object, b, to the underlying raw stream, and return the number of bytes written.

seek(offset, whence=0)

Note Seeking is disabled when writing to secure files.

Change the stream position to the given byte offset. offset is interpreted relative to the position indicated by whence. The default value for whence is 0 (SEEK_SET). Values for whence are:

- 0 (SEEK_SET) - start of the stream (the default); offset should be zero or positive
- 1 (SEEK_CUR) - current stream position; offset may be negative
- 2 (SEEK_END) - end of the stream; offset is usually negative

Returns the new absolute stream position.

tell()
Return the current stream position.

flush()
Flush the write buffers of the stream if applicable. This does nothing for read-only streams.

close()
Flush and close the stream. This does nothing if the file is already closed.

Import modules from file system

Python code can access code in modules using the builtin import command. When executing the line import foo, MicroPython goes through each entry in sys.path looking for a module called foo. It first checks for a package by looking for the file __init__.py in the directory foo. It then checks for a
file `foo.py` and finally `foo.mpy` (a pre-compiled Python file) before moving to the next entry in `sys.path`.
On startup, the XBee device sets its `sys.path` to a default of `['', '/flash', '/flash/lib']`.

**Reload a module**

If you want to reload a module after uploading a revised source file, use the following method to discard the old module and re-import from the updated file.

**Note** This is also necessary if the previous import attempt failed due to a syntax error.

```python
import sys
def reload(mod):
    mod_name = mod.__name__
    del sys.modules[mod_name]
    return __import__(mod_name)
```

**Compiled MicroPython files**

With the file system, the XBee device supports compiled MicroPython code in the form of `.mpy` files. You can convert a `.py` file to a `.mpy` file on the XBee device using the `uos.compile()` method; see [Modify file system contents](#). The XBee device also supports `.mpy` files created with [mpy-cross](#), the MicroPython cross-compiler. You can download [mpy-cross](#) for Windows, Linux and MacOS from the mpy-cross project.

**Note** You should pass `-mno-unicode` and `-msmall-int-bits=31` to `mpy-cross` when cross-compiling for the XBee device.

The benefit of using a `.mpy` file is that MicroPython can load it to the heap with minimal overhead, unlike the parsing and compiling process which could require a 32 kB heap to create a 7 kB `.mpy` file.

Since MicroPython checks for `.py` files in a given directory before `.mpy` files, you need to organize your files so the `.mpy` comes up first during an import search. One technique is to keep the Python source in `lib/source/` and then compile to an `.mpy` file in `lib/` after uploading new files; for example, with `/flash/lib` as the current working directory, `uos.compile('source/foo.py', 'foo.mpy')`.

---

**Digi MicroPython Programming Guide**

69
Send and receive User Data Relay frames

**Note** This section applies to the XBee Cellular Modem and the XBee 3 Zigbee RF Module. See [Which features apply to my device?](#) for a list of the supported features.

You can send and receive User Data Relay Frames from MicroPython using the `relay` module from the `xbee` module. Import the module with the statement: `from xbee import relay`

### Constants

```
71
```

### Methods

```
71
```

### Examples

```
72
```
Send and receive User Data Relay frames

Constants

Interfaces (always defined)

- `relay.SERIAL`: 0
- `relay.BLUETOOTH`: 1
- `relay.MICROPYTHON`: 2

Limits

- `relay.MAX_DATA_LENGTH`: maximum length of data passed to `relay.send()`

Methods

- `relay.receive()`
  Returns `None` if a frame is not available, otherwise a dictionary with entries for the sender (one of the interfaces, for example, `relay.SERIAL`), and message (a bytes object).

- `relay.send(dest, data)`
  Pass one of `relay.SERIAL`, `relay.BLUETOOTH` or `relay.MICROPYTHON` (for loopback) as `dest`. Can use `sender` from the dictionary returned from `receive()` as `dest` parameter. The `data` parameter should be a `bytes` or `string` object, or any other object that implements the `buffer` protocol. You can send a maximum of `relay.MAX_DATA_LENGTH` bytes in a single frame.

Exceptions

The `send()` method throws exceptions in at least the following cases:

- `ValueError` or `TypeError` for invalid parameters.
- `OSError(ENOBUFS)` if unable to allocate a buffer for the frame.
- `OSError(ENODEV)` for invalid `dest` parameter.
- `OSError(ECONNREFUSED)` when destination is not accepting frames (for example, the serial interface is not in API mode, Bluetooth is not connected and unlocked, the queue is full or delivery failed).

- `relay.callback(my_callback)`

  **Note** This section only applies to the XBee 3 Cellular Modem firmware x15 or later. See [Which features apply to my device?](#) for a list of the supported features.

  Register a callback that will be called whenever a user data relay frame is received. The callback function must take one parameter:
A dictionary with the following keys:

- **message**: The received data in `bytearray` format.
- **sender**: The source interface.

**Note** When a callback is registered, using `relay.callback()` will raise an error as only one method of relay frame delivery is supported at a time.

---

**Examples**

Digi has provided example applications which demonstrate how to use User Data Relay frames from MicroPython.

You can read these examples on GitHub: [https://github.com/digidotcom/xbee-micropython/tree/master/samples/xbee/communication](https://github.com/digidotcom/xbee-micropython/tree/master/samples/xbee/communication)

If you use the Digi XBee MicroPython PyCharm Plugin you can load the examples using **File > Import XBee MicroPython Sample Project...** in the **XBEE / COMMUNICATION** folder.
MicroPython libraries on GitHub

**Note** This section only applies to the XBee Cellular Modem. See [Which features apply to my device?](#) for a list of the supported features.

On GitHub, we maintain modules and sample code for use on XBee devices with MicroPython. The code is available at [github.com/digidotcom/xbee-micropython](https://github.com/digidotcom/xbee-micropython). The samples include:

- Secure Sockets Layer (SSL) and Transport Layer Security (TLS). See [The ussl module](#).
- Amazon Web Services (AWS). These samples demonstrate how to connect to AWS IoT and publish and subscribe to topics using the [umqtt.simple module](#). See [Use AWS IoT from MicroPython](#).
- MQ Telemetry Transport (MQTT). MQTT client for publish/subscribe. See [Publish to a topic](#).
- Digi Remote Manager. An HTTP client for Digi Remote Manager.
MicroPython modules

You can use many MicroPython modules with the XBee device. You can obtain a list of the available modules and of the module properties from the REPL. For more information see Discover available modules.

XBee-specific functions ........................................................................................................75
Standard modules and functions ..........................................................................................75
Discover available modules .................................................................................................76
**XBee-specific functions**

The following functions are specifically for use with the XBee device.

- Machine module
- Cellular network configuration module
- XBee module
- digi.cloud module

**Standard modules and functions**

The table below describes the MicroPython modules that you can use with the XBee device. For some functions and classes, you can only use a subset of the functions and classes with the XBee device. The table specifies those that you can use.

For a complete description of the MicroPython libraries and the related functions, see MicroPython libraries.

---

**Note** The MicroPython modules starting with "u" have aliases to the standard Python module names.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MicroPython functions</td>
<td>Functions used to access and control MicroPython internals.</td>
</tr>
<tr>
<td><strong>Note</strong></td>
<td>The standard set of MicroPython functions work with the XBee device.</td>
</tr>
<tr>
<td>Builtin Functions</td>
<td>Basic functions built in to MicroPython.</td>
</tr>
<tr>
<td>gc</td>
<td>Functions that control the garbage collector.</td>
</tr>
<tr>
<td>sys</td>
<td>System-specific functions.</td>
</tr>
<tr>
<td></td>
<td><strong>sys.print_exception(exc, file=sys.stdout)</strong></td>
</tr>
<tr>
<td></td>
<td>Available constants:</td>
</tr>
<tr>
<td></td>
<td>- sys.argv</td>
</tr>
<tr>
<td></td>
<td>- sys.byteorder</td>
</tr>
<tr>
<td></td>
<td>- sys.implementation</td>
</tr>
<tr>
<td></td>
<td>- sys.maxsize</td>
</tr>
<tr>
<td></td>
<td>- sys.modules</td>
</tr>
<tr>
<td></td>
<td>- sys.path</td>
</tr>
<tr>
<td></td>
<td>- sys.platform</td>
</tr>
<tr>
<td></td>
<td>- sys.version</td>
</tr>
<tr>
<td></td>
<td>- sys.version_info</td>
</tr>
<tr>
<td>ubinascii</td>
<td>This module implements conversions between binary data and various encodings of it in ASCII form (in both directions).</td>
</tr>
</tbody>
</table>
Discover available modules

You can obtain a list of the available modules and of the module properties from the REPL.

**Note** The MicroPython modules starting with "u" have aliases to the standard Python module names.

1. Access the MicroPython environment.
2. At the MicroPython >>> prompt, type `help('modules')` and press Enter. A list of available modules displays.
3. You can display a list of a module's properties and methods. In these steps, (modulename) in the command should be replaced by the module you are interested in.
   a. At the MicroPython >>> prompt, type `import modulename`, and press Enter.
   b. At the MicroPython >>> prompt, type `help(modulename)` and press Enter. A list of the module's properties and methods displays.

Digi extensions to the utime module

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucryptolib</td>
<td>This module provides an Advanced Encryption Standard (AES) API in MicroPython to perform encryption and decryption of data or files.</td>
</tr>
<tr>
<td>uhashlib</td>
<td>This module implements binary data hashing algorithms.</td>
</tr>
<tr>
<td>uio</td>
<td>This module contains additional types of stream (file-like) objects and helper functions.</td>
</tr>
<tr>
<td>ujson</td>
<td>This module performs JSON encoding and decoding.</td>
</tr>
<tr>
<td>uselect</td>
<td>This module provides functions to efficiently wait for events on multiple streams (select streams which are ready for operations) This is currently only available on the XBee Cellular Modem with firmware x15 and later as it primarily applies to sockets.</td>
</tr>
<tr>
<td>usocket</td>
<td>(XBee Cellular Modem only) This module provides access to the BSD socket interface. See Sockets for samples of using sockets with the XBee Cellular Modem.</td>
</tr>
<tr>
<td>ustruct</td>
<td>This module provides functions to pack and unpack primitive data types.</td>
</tr>
<tr>
<td>utime</td>
<td>XBee Cellular Modem: This module provides functions for getting the current time and date, measuring time intervals, and for delays. XBee 3 Zigbee RF Module: This module provides functions for measuring time intervals, and for delays.</td>
</tr>
</tbody>
</table>
time.tz_offset()

*Note* This section only applies to the XBee Cellular Modem. See *Which features apply to my device?* for a list of the supported features.

The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK).
Machine module

The machine module contains specific functions related to the XBee device.
For a detailed description of the MicroPython machine functions, see the machine function section in the standard MicroPython documentation.

Reset-cause ................................................................. 79
Random numbers .......................................................... 79
Unique identifier .......................................................... 79
Class PWM (pulse width modulation) ............................... 79
Class ADC: analog to digital conversion ............................ 80
Class I2C: two-wire serial protocol ................................. 82
Class Pin ....................................................................... 87
Class UART .................................................................... 87
Class WDT: watchdog timer ............................................. 90
Access the XBee device’s I/O pins ................................... 90
Use the Pin() constructor ............................................... 92
Use mode() to configure a pin ......................................... 93
Use pull() to configure an internal pull up/down resistor ....... 94
Reset-cause

This function returns the cause of a reset. See Constants for possible return values.

machine.reset_cause()

Constants

These return values describe the cause of a reset.

machine.BROWNOUT_RESET
machine.LOCKUP_RESET
machine.PWRON_RESET
machine.HARD_RESET
machine.WDT_RESET
machine.DEEPSLEEP_RESET
machine.SOFT_RESET

Note Some devices do not support reporting all of these reset causes. To see a list of possible values, run:

import machine
help(machine)

Random numbers

The machine.rng() method returns a 30-bit random number that is generated by the software.
The uos.urandom(n) method returns a bytes object with n random bytes generated by the hardware random number generator.

Unique identifier

The machine.unique_id() function returns a 64-bit bytes object with a unique identifier for the processor on the XBee Cellular Modem.
In some MicroPython ports, the ID corresponds to the network MAC address.

Class PWM (pulse width modulation)

Note This section only applies to devices that support the Pin I/O feature.

This class is not supported on the XBee Cellular Modem.
You use this function to enable PWM on XBee devices using pin P0.
The duty cycle is between 0 and 1023, inclusive of the end points. PWM cannot read or write the frequency.
This function uses the machine.PWM class. For information about the MicroPython machine module, see machine — functions related to the hardware.

For XBee devices that support PWM1, change the instances of P0 to P1 in the example program.

```python
from machine import Pin, PWM

pwm0 = PWM(Pin('P0'))  # create PWM object from a pin
pwm0.duty()            # get current duty cycle
pwm0.duty(200)         # set duty cycle
pwm0.deinit()          # turn off PWM on the pin

pwm0 = PWM('P0', duty=512)  # create and configure in one go
```

The following REPL session makes use of the PWM class:

```python
>>> from machine import PWM
>>> pwm0 = PWM('P0')
>>> pwm0.freq()    # report the frequency (23.46kHz)
23460
>>> pwm0.freq(10000)  # can't change fixed frequency on XBee
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NotImplementedError: can't set PWM frequency

>>> pwm0.duty()    # report the duty cycle
0
>>> pwm0.duty(255) # set 25% duty cycle
>>> pwm0.duty(511) # set 50% duty cycle
>>> pwm0.duty(767) # set 75% duty cycle
>>> pwm0.duty(1023) # set 100% duty cycle
>>> pwm0.duty()    # report the duty cycle
1023
>>> pwm0.deinit()  # disable DIO10
```

**Note** Supported on XBee 3 cellular only. Not supported on XBee Cellular Cat 1 Verizon and XBee Cellular 3G.

---

**Class ADC: analog to digital conversion**

**Note** This section only applies to devices that support the Pin I/O feature.

Use this class to read analog values on a pin.

```python
import machine

apin = machine.ADC('D0')    # create an analog pin on D0
val = apin.read()           # read an analog value
print(val)                  # display analog value
```

**Constructors**

You can create an ADC object associated with the assigned pin. You can then read analog values on that pin.

```python
class machine.ADC('D0')
```
For the XBee Cellular Modem the ADC analog reference is 2.5 V and the pin input range is 0 - 2.5 V. The ADC reference voltage and input range for XBee 3 Zigbee, DigiMesh and 802.15.4 are based on the AV value which can be 0 = 1.25 V, 1 = 2.5 V or 2 = VDD.

The ADC reading value has a resolution of 12 bits with a range of 0 - 4095.

Methods

Read the analog value
This function allows you to read the ADC value.

`apin.read()`

Note `apin.read()` returns a raw ADC sample. Use the following equation to convert this value to mV:

\[
\text{sample mV} = \left(\frac{A/D \ \text{reading} \times \text{Vref mV}}{4095}\right)
\]

Read the analog value as a 16-bit number
This function allows you to read the ADC value and get the result as a 16-bit number—0 to 65535. This function is provided for compatibility with other MicroPython implementations. XBee devices only support 12-bit ADC readings. Readings from `read_u16()` will match those from `read()`, scaled to a 16-bit range.

`apin.read_u16()`

Note `apin.read_u16()` returns a raw ADC sample. Use the following equation to convert this value to mV:

\[
\text{sample mV} = \left(\frac{A/D \ \text{reading} \times \text{Vref mV}}{65535}\right)
\]

Note This function is available on XBee Cellular and XBee 3 Cellular products with firmware ending in *16 or newer, and XBee 3 DigiMesh, 802.15.4 and Zigbee devices with firmware ending in *0B or newer.

Sample program
The following sample program applies to the XBee 3 Zigbee, DigiMesh, and 802.15.4.

```python
import machine
import xbee

x = xbee.XBee()

print('Setting the analog reference to 1 (2500 mV)')
x.atcmd('AV', 1)
print('Analog reference set to %d' % x.atcmd('AV'))

# Take an analog measurement
apin = machine.ADC('D0')
raw_val = apin.read()
val_mv = (raw_val * 2500)/4095
```
Class I2C: two-wire serial protocol

**Note** This section only applies to devices that support the I2C feature.

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

When created, I2C objects are associated with a specific two wire bus. They can be initialized when created, or initialized later on.

Printing the I2C object gives you information about its configuration.

The XBee device can function as an I2C master controlled by MicroPython. This allows you to perform basic sensing and actuation with I2C devices such as sensors and actuators via MicroPython without an additional microcontroller.

The MicroPython API is the same as documented in the MicroPython library reference except that the XBee device does not support primitive operations or the deinit operation.

The I2C implementation is provided through hardware, so when you use machine.I2C to initialize I2C, use the id parameter to select the interface. The only valid value is 1, which uses DIO1 for SCL and DIO11 for SDA. Using the scl and sda parameters to select pins is not valid on the XBee device.

**Note** You are not required to configure the XBee I/O using AT commands prior to creating an I2C object. The appropriate I/O configuration will be performed automatically.

The following table shows the pin layout associated with the example below.

<table>
<thead>
<tr>
<th>Pin</th>
<th>AT command</th>
<th>SMT pin</th>
<th>MMT pin</th>
<th>TH pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C SCL (DIO1)</td>
<td>D1</td>
<td>32</td>
<td>30</td>
<td>19</td>
</tr>
<tr>
<td>I2C SDA (DIO11)</td>
<td>P1</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

An example of using I2C follows:

```python
from machine import I2C

i2c = I2C(1, freq=400000)  # create I2C peripheral at frequency of 400kHz

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
```
Constructors

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

- id identifies a particular I2C peripheral. This version of MicroPython supports a single peripheral with id 1 using DIO1 for SCL and DIO11 for SDA.
- freq should be an integer that sets the maximum frequency for SCL.

General methods

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

Standard bus operations methods

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.

Note buf should be a bytearray type object.

Memory operations methods

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`.

**Note** `buf` should be a bytearray type object.

**Sample programs**

The following sample program applies to the HDC1080 I2C temperature and humidity sensor. This sensor is available on the XBiB-CU-TH, XBiB-C-MMT, and XBiB-C-SMT XBee development boards.

**Note** Refer to the HDC1080 datasheet available at ti.com for detailed technical information.

Make the following connections for this example:

<table>
<thead>
<tr>
<th>XBee pin</th>
<th>Description</th>
<th>HDC1080 pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIO1</td>
<td>SCL</td>
<td>6</td>
</tr>
<tr>
<td>DIO11</td>
<td>SDA</td>
<td>1</td>
</tr>
<tr>
<td>VCC</td>
<td>VCC</td>
<td>5</td>
</tr>
<tr>
<td>GND</td>
<td>ND</td>
<td>2</td>
</tr>
</tbody>
</table>

```python
dataclass HDC1080:
    def __init__(self, i2c, slave_addr=64):
        # Initialize a HDC1080 temperature and humidity sensor.
        # Keyword arguments:
        # i2c -- The i2c object used to interact with the I2C sensor.
        # slave_addr -- The slave address of the sensor (default 64 or 0x40).
        self.i2c = i2c
        scan_result = self.i2c.scan()
        assert slave_addr in scan_result, "Did not find slave %d in scan: %s" % (slave_addr, scan_result)
```
```python
self.addr = slave_addr
# Sleep for 15 ms to allow the temperature and humidity
# sensors to start recording.
sleep(0.015)
# Set temperature and humidity readings for independent
# operation, 14 bit resolution.
setup_data = 0b00000000
data = bytearray(3)
data[0] = CONF_REG
data[1] = setup_data # Configuration Register [15:8]
data[2] = 0 # Configuration Register [7:0] (Reserved)
i2c.writeto(self.addr, data)

def read_temp(self, celsius=False):
    """Read the temperature
    Keyword arguments:
    celsius -- If True the temperature is returned in Celsius, else
    Fahrenheit (default False).
    """
    # Set the pointer register to point to the temperature register.
    data = bytearray([TEMP_REG])
    self.i2c.writeto(self.addr, data)
    # Wait for conversion.
    sleep(0.01)
data = self.i2c.readfrom(self.addr, 2) # Read two bytes.
    # Convert big-endian array of bytes to integer.
    value = int.from_bytes(data, "big")
    if celsius:
        value = (value / (2 ** 16)) * 165 - 40
    else:
        value = (1.8 * value / (2 ** 16)) * 165 - 40
    return value

def read_humidity(self):
    """Read the relative humidity """
    # Write to the pointer register, changing it to the humidity
    register.
    data = bytearray([HUMI_REG])
    self.i2c.writeto(self.addr, data)
    # Wait for conversion.
    sleep(0.01)
data = self.i2c.readfrom(self.addr, 2) # Read two bytes.
    # Convert big-endian array of bytes to integer.
    value = int.from_bytes(data, "big")
    return value / (2 ** 16)) * 100

x = HDC1080(I2C(1, freq=200000), 64) # This sets up an instance of this
class.

print('Humidity:', x.read_humidity()) # Display humidity.
print('Temperature (C):', x.read_temp(True)) # Display temp in Celsius.
print('Temperature (F):', x.read_temp(False)) # Display temp in Fahrenheit.
```

The following sample works with a DS1621 I2C temperature sensor. Make the following connections before testing the code:
Machine module

Class I2C: two-wire serial protocol

<table>
<thead>
<tr>
<th>Xbee pin</th>
<th>Description</th>
<th>DS1621 pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIO1</td>
<td>SCL</td>
<td>2</td>
</tr>
<tr>
<td>DIO11</td>
<td>SDA</td>
<td>1</td>
</tr>
<tr>
<td>VCC</td>
<td>VCC</td>
<td>8</td>
</tr>
<tr>
<td>GND</td>
<td>GND</td>
<td>4</td>
</tr>
</tbody>
</table>

In addition, connect the address pins of the DC1621 (5, 6 and 7) to ground, and a pullup resistor from the SDA line to VCC.

```python
# Simple DS1621 I2C Example
# Wiring Diagram:
# Xbee -> DS1621
# SCL 2
# SDA 1 (and connect via pullup resistor to Vcc)
# Vcc 8
# GND 4 (and address pins 5, 6 and 7)

import machine
import utime
import ustruct

i2c = machine.I2C(1)
slave_addr = 0x48  # 0b100_1000. Assumes A0-2 are low.

# The high/low temperature registers are 9-bit two's complement signed ints.
# Data is written MSB first, so as an example the value 1 (0b1) is represented
# as 0b00000000 10000000, or 0x0080.
REGISTER_FORMAT = '>h'
REGISTER_SHIFT = 7

# Read a 9-bit temperature from the DS1621. Values for <protocol>:
# b'0xAA' for Read Temperature
# b'0xA1' for TH Register
# b'0xA2' for TL Register
# Returns temperature in units of 0.5C. Fahrenheit = temp * 9 / 10 + 32

def read_temperature(protocol=b'\xAA'):
    i2c.writeto(slave_addr, protocol, False)
    data = i2c.readfrom(slave_addr, 2)
    value = ustruct.unpack(REGISTER_FORMAT, data)[0] >> REGISTER_SHIFT
    return value

def start_convert():
    i2c.writeto(slave_addr, '\xEE', True)

def stop_convert():
    i2c.writeto(slave_addr, '\x22', True)

def read_access_config():
    i2c.writeto(slave_addr, '\xAC', False)
    return i2c.readfrom(slave_addr, 1)
```

Digi MicroPython Programming Guide 86
def write_access_config(value):
    written = i2c.writeto(slave_addr, b'\xA1' + ustruct.pack('b', value))
    assert written == 2, "Access Config write returned %d ?" % written

def display_continuous():
    start_convert()
    try:
        while True:
            print('%.1fF' % (read_temperature() * 9 / 10 + 32))
            utime.sleep(2)
    except:
        stop_convert()
        raise

# Perform a scan and make sure we find the slave device we want to talk to.
devices = i2c.scan()
assert (slave_addr in devices, "Did not see slave device address %d in scan result: %s" % (slave_addr, devices))
display_continuous()

## Class Pin

**Note** This section only applies to devices that support the **Pin** I/O feature.

**Note** Only pins D0-P2 are accessible using the Pin class.

You can use the Pin class with the XBee device. For information, see Class Pin: Control I/O pins.

## Class UART

**Note** This section only applies to devices that support the **Secondary UART** feature.

MicroPython on the XBee Cellular Modem provides access to a 3-wire or 5-wire TTL-level serial port (referred to as `machineUART(1)`) on the following pins. The table also indicates the proper connections when testing with an FTDI TTL-232R cable. Note that the FTDI cable's pin 3 (VCC) remains unconnected.

<table>
<thead>
<tr>
<th>XBee Pin</th>
<th>Name</th>
<th>Description</th>
<th>Direction</th>
<th>FTDI TTL-232R</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>GND</td>
<td>Ground</td>
<td>N/A</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>DIO4</td>
<td>Transmit (TX)</td>
<td>XBee →</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>DIO12</td>
<td>Receive (RX)</td>
<td>XBee ←</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>DIO2</td>
<td>Ready to Receive (RTS)</td>
<td>XBee →</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>DIO3</td>
<td>Clear to Send (CTS)</td>
<td>XBee ←</td>
<td>6</td>
</tr>
</tbody>
</table>

## Machine module

Class Pin

Note: This section only applies to devices that support the Pin I/O feature.

Note: Only pins D0-P2 are accessible using the Pin class.

You can use the Pin class with the XBee device. For information, see Class Pin: Control I/O pins.

Class UART

Note: This section only applies to devices that support the Secondary UART feature.

MicroPython on the XBee Cellular Modem provides access to a 3-wire or 5-wire TTL-level serial port (referred to as `machineUART(1)`) on the following pins. The table also indicates the proper connections when testing with an FTDI TTL-232R cable. Note that the FTDI cable's pin 3 (VCC) remains unconnected.

<table>
<thead>
<tr>
<th>XBee Pin</th>
<th>Name</th>
<th>Description</th>
<th>Direction</th>
<th>FTDI TTL-232R</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>GND</td>
<td>Ground</td>
<td>N/A</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>DIO4</td>
<td>Transmit (TX)</td>
<td>XBee →</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>DIO12</td>
<td>Receive (RX)</td>
<td>XBee ←</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>DIO2</td>
<td>Ready to Receive (RTS)</td>
<td>XBee →</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>DIO3</td>
<td>Clear to Send (CTS)</td>
<td>XBee ←</td>
<td>6</td>
</tr>
</tbody>
</table>
Using the RTS and CTS pins for hardware flow control is optional. The XBee Cellular Modem can use RTS to signal the remote end to stop sending when its receive buffer is close to full, and it will conversely monitor the CTS signal and only send when the remote end asserts the signal. Both RTS and CTS are active low signals where 0 (GND) represents "asserted" (or "safe to send") and 1 (VCC) represents "deasserted" (or "wait to send").

**Test the UART interface**

Once you have the hardware set up:

1. Open a terminal window to the MicroPython REPL on your XBee Cellular Modem.
2. Open a second terminal window to the TTL-232R cable you connected to DIO4/DIO12.
3. Leave DIO2/DIO3 disconnected and configure the second terminal window without any flow control.
4. From the REPL prompt, press **Ctrl-E** to enter paste mode.
5. Paste the following test code (which uses the default baud rate of 115,200).

    ```python
    from machine import UART
    import time

    u = UART(1)
    u.write('Testing from XBee\n')

    while True:
        uart_data = u.read()
        if uart_data:
            print(str(uart_data, 'utf8'), end='')
        time.sleep_ms(5)
    ```

6. Press **Ctrl-D** on a blank line to execute it.
7. You should see the message **Testing from XBee** in the other terminal window, and anything you type there should appear in your MicroPython terminal.
8. From the MicroPython terminal, use **Ctrl-C** to send a **KeyboardInterrupt** and exit the **while** loop.

**Use the UART class**

UART implements the standard UART/USART duplex serial communications protocol. At the physical level it consists of at least two lines: RX and TX, with support for optional hardware flow control using RTS/CTS handshaking. The unit of communication is a character (not to be confused with a string character) which can be 5 to 8 bits wide.

Create UART objects using the **machine.UART()** class:

```python
from machine import UART
uart = UART(1, 9600)  # create with given baudrate
uart.init(9600, bits=8, parity=None, stop=1)  # reconfigure with given parameters
```

A UART object acts like a stream object and uses the standard stream methods for reading and writing.
Machine module

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uart.read(10)</td>
<td># read 10 characters, returns a bytes object</td>
</tr>
<tr>
<td>uart.read()</td>
<td># read all available characters</td>
</tr>
<tr>
<td>uart.readline()</td>
<td># read a line</td>
</tr>
<tr>
<td>uart.readinto(buf)</td>
<td># read and store into the given buffer</td>
</tr>
<tr>
<td>uart.write('abc')</td>
<td># write the 3 characters</td>
</tr>
</tbody>
</table>

To check if there is anything to be read, use:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uart.any()</td>
<td># returns the number of characters waiting</td>
</tr>
</tbody>
</table>

**Constructors**

class machine.UART(id, baudrate=115200, bits=8, parity=None, stop=1, *, flow=0, timeout=0, timeout_char=0)

- **id**: XBee Cellular supports a single UART, using the id 1.
- **baudrate**: Clock rate for serial data.
- **bits**: Bits per character, a value from 5 to 8.
- **parity**: An additional parity bit added to each byte, either None, 0 (even) or 1 (odd).
- **stop**: Number of stop bits after the optional parity bit, either 1 or 2.
- **flow**: Hardware flow control; either 0 for none, UART.RTS for RTS-only, UART.CTS for CTS-only or UART.RTS|UART.CTS for both.
- **timeout**: Number of milliseconds to wait for reading the first character.
- **timeout_char**: Number of milliseconds to wait between characters when reading.

You can pass parameters before the flow keyword without their names, for example: UART(1, 115200, 8, None, 1).

**Note** Unlike other MicroPython platforms, the XBee Cellular Modem uses a circular buffer to store serial data, and the timeout and timeout_char settings do not apply to writes.

**Methods**

UART.init(baudrate=0, bits=0, parity=-1, stop=0, *, flow=-1, timeout=-1, timeout_char=-1)

See Constructors for descriptions of each keyword. The default values (used if a keyword is not specified) leave the current setting unchanged. Calling UART.init() resets the port using the current settings.

UART.deinit()

Turn off the UART bus. After calling deinit(), attempts to write to the UART result in an OSError (EPERM) exception but reads continue to pull buffered bytes.

UART.any()

Returns an integer value of the number of bytes in the read buffer, or 0 if no bytes are available.

UART.read([nbytes])

Read characters. If nbytes is specified and a positive value, then read at most that many bytes, otherwise read as much data as possible.

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

UART.readinto(buf, nbytes)

Read bytes into the buf. If nbytes is specified then read at most that many bytes. Otherwise, read at most len(buf) bytes.
Return value: number of bytes read and stored into `buf` or `None` on timeout.

`UART.readline()`
Read a line, ending in a newline character.
Return value: the line read or `None` on timeout.

`UART.write(buf)`
Write the buffer of bytes to the bus.
Return value: number of bytes written.

**Constants**
Used to specify the flow control type.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UART.RTS</td>
<td></td>
</tr>
<tr>
<td>UART.CTS</td>
<td></td>
</tr>
</tbody>
</table>

**Class WDT: watchdog timer**
XBee 3 Cellular devices with firmware ending in *15* or newer and XBee3 DigiMesh, 802.15.4, or ZigBee devices with firmware ending in *0A* or newer contain the `machine.WDT()` object. It is primarily the same as documented at `wipy/library/machine.WDT`. One primary difference is the addition of a `response` parameter to the constructor. This allows you to select the behavior of the system when the watchdog timer expires without being fed.

On XBee 3 DigiMesh, 802.15.4, or Zigbee devices, the default timeout for the watchdog is five seconds. On XBee 3 Cellular devices, the default timeout has changed to one minute as blocking operations on cellular can normally take many seconds to complete and in an initial attempt to use the watchdog should not cause a reset.

The `sleep_now` and `clean_shutdown` operations which can be very lengthy will not trigger a watchdog reset while in progress and the watchdog timer will have the full timeout upon coming out of sleep.

Valid choices are:
- `SOFT_RESET`: resets only the MicroPython interpreter as if the `soft_reset` method in the device had been called.
- `HARD_RESET`: Reboots the entire XBee device.
- `CLEAN_SHUTDOWN`: Shuts down the cellular component and then reboots. If the cellular component cannot be cleanly shut down in two minutes it is reset anyway.

**Note** The `CLEAN_SHUTDOWN` option is only available on XBee 3 Cellular devices.

**Access the XBee device's I/O pins**
You can access the XBee device's I/O pins using the `Pin` class from the `machine` module. To get started, import that class and use the `help()` function to display available methods and constants.

The REPL sessions below all assume you have started with `from machine import Pin`.

```python
>>> from machine import Pin
>>> help(Pin)
object <class 'Pin'> is of type type
```
To see a list of pins available on your hardware, get help on the Pin.board class:

```python
>>> help(Pin.board)
object <class 'board'> is of type type
   D0 -- Pin(Pin.board.D0, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF0_COMMISSION)
   D1 -- Pin(Pin.board.D1, mode=Pin.DISABLED)
   D2 -- Pin(Pin.board.D2, mode=Pin.DISABLED)
   D3 -- Pin(Pin.board.D3, mode=Pin.DISABLED)
   D4 -- Pin(Pin.board.D4, mode=Pin.DISABLED)
   D5 -- Pin(Pin.board.D5, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF5_ASSOC_IND)
   D6 -- Pin(Pin.board.D6, mode=Pin.DISABLED)
   D7 -- Pin(Pin.board.D7, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF7_CTS)
```
Use the Pin() constructor

Use `Pin(name, mode, pull=None, *, value, alt)` to create a new `Pin` object with a specific configuration. The `name` parameter can be a string (for example, `D0`) or reference to an existing

```python
>>> button = Pin.board.D0
>>> button.mode(Pin.IN)
>>> button
Pin(Pin.board.D0, mode=Pin.IN, pull=Pin.PULL_UP)
>>> button.mode(Pin.IN)
>>> button.value()
1
>>> button.value()
0
```

The names `button` and `Pin.board.D0` both show the new configuration after using the `mode()` method to make it an input. The example keeps the configuration of an internal pull up to Vcc to simplify the button wiring—just short the pin to ground when you press the button. You can check the status of the button using the `value()` method. It returns 0 when pressed (shorted to ground) and 1 otherwise (pulled up to Vcc).

From the list above, you can see the current configuration of all the pins. Note that pins `P0` through `P9` have aliases of `D10` through `D19`. Also, through-hole XBee 3 RF products (802.15.4, DigiMesh and Zigbee) still list pins `P5` through `P9` even though they are only accessible on the surface-mount products.

You can assign any of the `Pin.board` objects to a variable that is easier to type (for example, `d0 = Pin.board.D0`) or more descriptive (for example, `status_led = Pin.board.D3`). Multiple names for a pin all reference the same physical pin, so changes made through one name appear in all other names. For example, to change pin D0 from operating as a commissioning button, you could do the following:

```plaintext
D8 -- Pin(Pin.board.D8, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF8_SPI_MASTER)
D9 -- Pin(Pin.board.D9, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF9_MASTER)
P0 -- Pin(Pin.board.P0, mode=Pin.DISABLED)
P1 -- Pin(Pin.board.P1, mode=Pin.DISABLED)
P2 -- Pin(Pin.board.P2, mode=Pin.DISABLED)
P3 -- Pin(Pin.board.P3, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF13_DOUT)
P4 -- Pin(Pin.board.P4, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF14_DIN)
P5 -- Pin(Pin.board.P5, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF15_SPI_MASTER)
P6 -- Pin(Pin.board.P6, mode=Pin.ALT, alt=Pin.AF16_SPI_MOSI)
P7 -- Pin(Pin.board.P7, mode=Pin.ALT, alt=Pin.AF17_SPI_SSEL)
P8 -- Pin(Pin.board.P8, mode=Pin.ALT, alt=Pin.AF18_SPI_SCLK)
P9 -- Pin(Pin.board.P9, mode=Pin.ALT, alt=Pin.AF19_SPI_ATTN)
D10 -- Pin(Pin.board.P0, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF10_SCK)
D11 -- Pin(Pin.board.P1, mode=Pin.DISABLED)
D12 -- Pin(Pin.board.P2, mode=Pin.DISABLED)
D13 -- Pin(Pin.board.P3, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF13_DOUT)
D14 -- Pin(Pin.board.P4, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF14_DIN)
D15 -- Pin(Pin.board.P5, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF15_SPI_MASTER)
D16 -- Pin(Pin.board.P6, mode=Pin.ALT, alt=Pin.AF16_SPI_MOSI)
D17 -- Pin(Pin.board.P7, mode=Pin.ALT, alt=Pin.AF17_SPI_SSEL)
D18 -- Pin(Pin.board.P8, mode=Pin.ALT, alt=Pin.AF18_SPI_SCLK)
D19 -- Pin(Pin.board.P9, mode=Pin.ALT, alt=Pin.AF19_SPI_ATTN)
```
Use mode() to configure a pin

Note By default pull is set to None and will disable a pull up/down resistor already configured for a given pin.

The documentation for mode(), pull(), and value() also apply to those parameters in the Pin() constructor. See Pin.ALT for usage of the alt parameter.

Use mode() to configure a pin

Note Using the Pin() constructor to change the mode() of a pin will automatically update the corresponding AT command value to match and vice-versa. For example, setting pin D11 to disabled sets the P1 AT command to 0.

Pin.DISABLED
If you are not using a pin, configure it as Pin.DISABLED.

Pin.IN
Pin acts as an input that you can read with the value() method, which returns 1 for high and 0 for low. See the pull() method for configuring an internal pull up/down resistor on input pins.

```python
>>> button = Pin.board.D0
>>> button.mode(Pin.IN)
>>> button.pull(Pin.PULL_UP)
>>> # or: button = Pin('D0', mode=Pin.IN, pull=Pin.PULL_UP)
>>> button
Pin(Pin.board.D0, mode=Pin.IN, pull=Pin.PULL_UP)
>>> button(0)
1
>>> # hold button and then read value again
>>> button.value()
0
```

Pin.OUT
Pin acts as an output that you can set by passing a parameter to the value() method. Any value that evaluates to True sets the pin high (Vcc) and all other values set it low (ground). Pin objects also support the on() and off() methods as shortcuts for value(1) and value(0) respectively, and toggle() to toggle the current value. For example, you can override the association indicator normally configured for D5 and control it manually:

```python
>>> d5 = Pin.board.D5
>>> d5.mode(Pin.OUT)
>>> # turn LED off
>>> d5.value(0)
>>> # turn LED on
>>> d5.value(1)
>>> # turn LED off
>>> d5.off()
```
Use pull() to configure an internal pull up/down resistor

You typically only enable an internal pull up/down resistor on an input to keep it from floating. Enabling, disabling, or changing the state of a pull up/down resistor using the Pin() constructor will
automatically update the PR and PD parameter values and vice-versa. The Pin.pull() method and pull parameter to the Pin() constructor take a single parameter:

- **None**: disable the internal resistor
- **Pin.PULL_DOWN**: enable a pull down to ground
- **Pin.PULL_UP**: enable a pull up to Vcc
digi.ble module

The `digi.ble` module provides interaction with the Bluetooth Low Energy (BLE) functionality of the XBee device.

You can import the `digi.ble` module as follows:

```python
from digi import ble
```

If you prefer, you can also use the `digi.ble` module as follows:

```python
import digi

digi.ble.active(False)
```

Feature support

- active() .......................................................... 97
- config() ......................................................... 97
- disconnect_code() ............................................. 98
- gap_connect() .................................................. 99
- gap_connection methods .................................... 100
- UUID() ................................................................ 107
- gap_scan() .......................................................... 108
- gap_scan methods ............................................... 108
- gap_scan advertisement format .............................. 109
- Use gap_scan as an iterator .................................... 109
- Use gap_scan as a context manager .......................... 110
- gap_advertise() .................................................. 110
- xbee_connect() .................................................... 111
- xbee_connection methods .................................... 112
- digi.ble samples ............................................... 112
- Troubleshooting .................................................. 113
## Feature support

The following table shows which devices support the `digi.ble` module.

<table>
<thead>
<tr>
<th>Feature</th>
<th>XBee 3 Cellular</th>
<th>XBee 3 Zigbee</th>
<th>XBee 3 802.15.4</th>
<th>XBee 3 DigiMesh</th>
<th>XBee Cellular</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>active()</code></td>
<td>Firmware ending in *15 or higher</td>
<td>Firmware 1009 or later</td>
<td>Firmware 200A and later</td>
<td>Firmware 300A and later</td>
<td>Not supported</td>
</tr>
<tr>
<td><code>config()</code></td>
<td>Firmware ending in *15 or higher</td>
<td>1009: <code>config('mac')</code> 100A or later: <code>config('mac')</code>, updating configuration</td>
<td>Firmware 200A and later</td>
<td>Firmware 300A and later</td>
<td>Not supported</td>
</tr>
<tr>
<td><code>delete_bondings()</code></td>
<td>Firmware ending in *15 or higher</td>
<td>100A</td>
<td>Firmware 200A and later</td>
<td>Firmware 300A and later</td>
<td>Not supported</td>
</tr>
<tr>
<td><code>gap_scan()</code></td>
<td>Firmware ending in *15 or higher</td>
<td>1009</td>
<td>Firmware 200A and later</td>
<td>Firmware 300A and later</td>
<td>Not supported</td>
</tr>
<tr>
<td><code>gap_advertise()</code></td>
<td>Firmware ending in *15 or higher</td>
<td>1009</td>
<td>Firmware 200A and later</td>
<td>Firmware 300A and later</td>
<td>Not supported</td>
</tr>
<tr>
<td><code>gap_connect()</code></td>
<td>Firmware ending in *15 or higher</td>
<td>100A</td>
<td>Firmware 200A and later</td>
<td>Firmware 300A and later</td>
<td>Not supported</td>
</tr>
<tr>
<td><code>io_callbacks()</code></td>
<td>Firmware ending in *15 or higher</td>
<td>100A</td>
<td>Firmware 200A and later</td>
<td>Firmware 300A and later</td>
<td>Not supported</td>
</tr>
<tr>
<td><code>passkey_confirm()</code></td>
<td>Firmware ending in *15 or higher</td>
<td>100A</td>
<td>Firmware 200A and later</td>
<td>Firmware 300A and later</td>
<td>Not supported</td>
</tr>
<tr>
<td><code>passkey_enter()</code></td>
<td>Firmware ending in *15 or higher</td>
<td>100A</td>
<td>Firmware 200A and later</td>
<td>Firmware 300A and later</td>
<td>Not supported</td>
</tr>
<tr>
<td><code>secure()</code></td>
<td>Firmware ending in *15 or higher</td>
<td>100A</td>
<td>Firmware 200A and later</td>
<td>Firmware 300A and later</td>
<td>Not supported</td>
</tr>
<tr>
<td><code>xbee_connect()</code></td>
<td>Firmware ending in *16 or higher</td>
<td>100B and later</td>
<td>200B and later</td>
<td>300B and later</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

### active()

Use this function to set or query whether BLE functionality is enabled on the XBee device. This method is equivalent to the `ATBT` command.
ble.active([mode])

Without parameters:
- Returns `True` if BLE is enabled on the XBee device (`ATBT = 1`).
- Returns `False` if BLE is disabled on the XBee device (`ATBT = 0`).

With parameters:
- `True`: Enable BLE functionality.
- `False`: Disable BLE functionality.

config()
Query a BLE configuration value by name, or update one or more BLE configuration values.

ble.config(name)
ble.config([[interval_ms=...], [latency=...], [timeout_ms=...]])

Query a value
To query a BLE configuration value, pass the name of the value as a string. Currently supported values are:
- "mac": Returns the device BLE MAC address, as a `bytes` object.
- "interval_ms": Initial connection interval to use on future GAP connections, as an integer.
- "latency": Initial slave latency value to use on future GAP connections, as an integer.
- "timeout_ms": Initial connection supervision timeout to use on future GAP connections, as an integer.

Update configuration values
To update one or more BLE configuration values, pass the value(s) as keyword argument(s). `config` returns `None`—in other words it has no return value—when updating settings.

<interval_ms>
Update the initial connection interval to use on future GAP connections—see `gap_connect()`.
The connection interval is the time between two data transfer events on the GAP connection. The value will be rounded down to the nearest multiple of 1.25 milliseconds. `interval_ms` may be between 8 and 4000 (4 seconds).
Default value (restored at XBee power-up): 50 milliseconds.

<latency>
Update the initial slave latency to use on future GAP connections—see `gap_connect()`.
The slave latency is the number of consecutive connection events that the connected peripheral is allowed to skip before the connection is dropped. `latency` may be between 0 and 500.
Default value (restored at XBee power-up): 0.
<timeout_ms>
Update the initial connection supervision timeout to use on future GAP connections—see `gap_connect()`.

The connection supervision timeout value is the time that the central device (in this case, the XBee) will wait for a data transfer before assuming that the connection is lost. `timeout_ms` may be between 100 and 32000 (32 seconds). `timeout_ms` must be larger than `2 * interval_ms * (latency + 1)`.

Default value (restored at XBee power-up): 1000 milliseconds (1 second).

disconnect_code()
When called on a connection which has been closed, returns a value from the Bluetooth Core specification Vol 2, Part D (Error Codes) indicating the reason for the disconnect. Calling this on an open connection returns zero.

The most common values to see here include:

- 8 - Connection timeout
- 19 - Remote user terminated
- 22 - Connection terminated by local host

gap_connect()
Connect to a BLE device. The address type and the address are required arguments.

```python
collection = ble.gap_connect(addr_type, address[, timeout_ms=5000][, interval_us][, window_us][, onclose])
```

<addr_type>
The `<addr_type>` parameter specifies the address type.

The possible values are defined as constants on the `digi.ble` module:

<table>
<thead>
<tr>
<th>Constants</th>
<th>Applicable firmwares</th>
</tr>
</thead>
<tbody>
<tr>
<td>ble.ADDR_TYPE_PUBLIC</td>
<td>802.15.4, Zigbee, DigiMesh, Cellular devices</td>
</tr>
<tr>
<td>ble.ADDR_TYPE_RANDOM</td>
<td>802.15.4, Zigbee, DigiMesh, Cellular devices</td>
</tr>
<tr>
<td>ble.ADDR_TYPE_PUBLIC_IDENTITY</td>
<td>802.15.4, Zigbee, DigiMesh, Cellular devices</td>
</tr>
<tr>
<td>ble.ADDR_TYPE_RANDOM_IDENTITY</td>
<td>802.15.4, Zigbee, DigiMesh, Cellular devices</td>
</tr>
</tbody>
</table>

* `ble.ADDR_TYPE_PUBLIC_IDENTITY` and `ble.ADDR_TYPE_RANDOM_IDENTITY` are removed in 802.15.4’s 0x200B release and removed in Zigbee’s 0x300B release. These constants are going to be removed from future releases of DigiMesh and Cellular firmwares so we do not advise using them.

<address>
The `<address>` parameter is a bytes object which represents the BLE MAC address that is the target of the connection.
<timeout_ms>
The `<timeout_ms>` parameter specifies the timeout before giving up on a connection. When a connection times out, **OSErr**or **ETIMEDOUT** is raised.

**Note** The connection attempt will automatically time out if the remote peripheral does not respond to a connection request within six connection intervals—see the `<interval_ms>` parameter.

<interval_us>, <window_us>
Use `<interval_us>` and `<window_us>` to optionally configure the duty cycle to scan for the remote device. The scanner will run for `<window_us>` microseconds every `<interval_us>` microseconds. The default interval and window are 20 milliseconds and 11.25 milliseconds, respectively. Both values must be at least 2,500 microseconds (2.5 milliseconds) and no more than approximately 40.96 seconds (40,959,375 microseconds).

<onclose>
The `<onclose>` parameter assigns a function as a callback to be triggered on receiving a close event on the connection. The onclose function will be called with two arguments, the `ble_connection` object that received the disconnect event and the disconnect code.

**Note** The supplied disconnect code is also stored on the `ble_connection` object itself, see `disconnect_code()` for more information on disconnect codes.

Return value
If the GAP connect operation is started successfully, a `gap_connection` object is returned. If the GAP connect operation is not successful, an **OSErr**or is raised.

gap_connection methods
The methods available on a `gap_connection` object—returned by the `gap_connect()`—function—are as follows.

gattc_services()
Discover Generic Attribute Profile (GATT) services in the remote device’s database. A specific service can be discovered by specifying the UUID of the service.

```python
connection.gattc_services([, uuid])
```

<uuid>
The `<uuid>` parameter is either a **UUID** object or a value that can construct a **UUID** object. When specified, the iterator only returns the service with that UUID, otherwise all services are returned.

**Return value**
If the GATT service discovery operation is successful, an iterator is returned containing tuples with the following information about each discovered service:
(handle, uuid)

<handle>
The <handle> is an integer used to reference the service.

<uuid>
The <uuid> is an UUID object.
If the GATT service discovery operation is not able to be started, an OSError is raised.

gattc_characteristics()
Discover GATT characteristics of a service in the remote device's database. A specific characteristic can be discovered by specifying the UUID of the characteristic.

collection.gattc_characteristics(service[, uuid])

<service>
The <service> parameter is a service handle discovered from gattc_services().

<uuid>
The <uuid> parameter is either a UUID object or a value that can construct a UUID object. When specified, the iterator or only returns the characteristic(s) with that UUID, otherwise all characteristics are returned.

Return value
If the GATT characteristic discovery operation is successful, an iterator is returned containing tuples with the following information about each discovered characteristic:

(handle, uuid, properties)

<handle>
The <handle> is an integer used to reference the characteristic.

<uuid>
The <uuid> is an UUID object.

<properties>
<properties> is an integer containing the property flags. These flags are defined as constants in the digi.ble module.
They are the following:

- PROP_BROADCAST
- PROP_READ
- PROP_WRITE_NO_RESP
- PROP_WRITE
- PROP_NOTIFY
- PROP_INDICATE
- PROP_AUTH_SIGNED_WR
If the GATT characteristic discovery operation is not able to be started, an OSError is raised.

**gattc_descriptors()**
Return an iterator of all GATT descriptors of a characteristic in the remote device's database. Note that this returns an iterator and the descriptor discovery will not be completed until the iterator is emptied.

```python
connection.gattc_descriptors(characteristic)
```

**<characteristic>**
The `<characteristic>` parameter is a characteristic handle discovered from `gattc_characteristics`.

**Return value**
If the GATT descriptor discovery operation is successful, an iterator is returned containing tuples with the following information about each discovered descriptor:

```python
(handle, uuid)
```

**<handle>**
The `<handle>` is an integer used to reference the descriptor.

**<uuid>**
The `<uuid>` is an UUID object.

If the GATT descriptor discovery operation is not able to be started, an OSError is raised.

**gattc_read_characteristic()**
Issue a remote read to the connected peripheral to the specified characteristic.

```python
connection.gattc_read_characteristic(characteristic_handle)
```

**<characteristic_handle>**
The `<characteristic_handle>` parameter is a characteristic handle discovered from `gattc_characteristics`.

**Return value**
`gattc_read_characteristic` returns a bytes object containing the characteristic attribute value. If the characteristic passed in is invalid or the connection to the peripheral device is lost, an OSError is raised. If the required read permissions for the characteristic are not met then an empty bytes object is returned.

**gattc_configure()**
Enable or disable notifications/indications for a given characteristic. This configures the remote server to send notifications/indications and registers the passed callback to be called when one is received.
connection.gattc_configure(characteristic_handle, [callback=None], notification=False)

**<characteristic_handle>**
The `<characteristic_handle>` parameter is a characteristic handle discovered from `gattc_characteristics`, whose characteristic has the notify property—`ble.PROPERTY_NOTIFY`—or the indicate property—`ble.PROPERTY_INDICATE`.

**<callback>**
The `<callback>` parameter is a user defined callback that is called whenever a notification or indication is received from the passed characteristic.

This callback should have two parameters. The first is the data, a bytes object. The second is an integer indicating the offset of the data.

If `None` is passed as the `<callback>` parameter or `<callback>` is not specified, notifications/indications are disabled for the characteristic.

**<notification>**
The `<notification>` parameter is an optional parameter used to distinguish between using notifications instead of indications. By default, indications are used. If `<notification>` is set to `True`, notifications are used instead of indications.

*Note* Notifications are unacknowledged by the client and do not guarantee delivery of the data.

gattc_read_descriptor()
Issue a remote read to the connected peripheral to the specified descriptor.

connection.gattc_read_descriptor(descriptor_handle)

**<descriptor_handle>**
The `<descriptor_handle>` parameter is a descriptor handle discovered from `gattc_descriptors`.

**Return value**
`gattc_read_descriptor` returns a bytes object containing the descriptor attribute value.

If the descriptor handle passed in is invalid or the connection to the peripheral device is lost, an `OSError` is raised.

If the required read permissions for the descriptor are not met then an empty bytes object will be returned.

gattc_write_characteristic()
Issue a remote write to the connected peripheral to the specified characteristic.

connection.gattc_write_characteristic(characteristic_handle, data)

**<characteristic_handle>**
The `<characteristic_handle>` parameter is a characteristic handle discovered from `gattc_characteristics`. 
The `<data>` parameter specifies the data to be written to the remote characteristic.

**Return value**

`gattc_write_characteristic` returns `None`—in other words it has no return value. If the characteristic handle passed in is invalid or the connection to the peripheral device is lost, an `OSError` is raised.

```python
connection.gattc_write_characteristic(characteristic_handle, data)
```

The `<descriptor_handle>` parameter is a descriptor handle discovered from `gattc_descriptors`.

The `<data>` parameter specifies the data to be written to the remote descriptor.

**Return value**

`gattc_write_descriptor` returns `None`—in other words it has no return value. If the descriptor handle passed in is invalid or the connection to the peripheral device is lost, an `OSError` is raised.

```python
connection.gattc_write_descriptor(descriptor_handle, data)
```

Returns the BLE peripheral device's address and address type.

```python
address_type, address = connection.addr()
```

**Return value**

`addr` A 2-tuple containing the BLE addressing information of:

- BLE address type of the peripheral device, as an `int` type.
- BLE address of the peripheral device, formatted as a ``bytes`` object.

Close the connection. The connection object will no longer be usable.

```python
connection.close()
```

Query a BLE connection configuration value by name, or update one or more BLE connection configuration values.
connection.config(name)
connection.config([interval_ms=..., ] [latency=..., ] [timeout_ms=..])

All parameters must be specified as keyword argument, for example:

**Query a value**
To query a connection’s BLE configuration value, pass the name of the value as a string. Currently supported values are:

- "interval_ms": BLE connection interval, as an integer.
- "latency": BLE slave latency, as an integer.
- "mtu": BLE MTU size, as an integer.
- "timeout_ms": BLE connection supervision timeout, as an integer.

To control these timing parameters before opening a connection, see `config()`.

**Update configuration values**
To update one or more of the BLE timing parameters for this connection, use `config()` with keyword arguments. For example:

```
connection.config(interval_ms=100, timeout_ms=1000)
```

**<interval_ms>**
The connection interval is the time between two data transfer events on the GAP connection. The value will be rounded down to the nearest multiple of 1.25 milliseconds. `interval_ms` may be between 8 and 4000 (4 seconds).

Default value (restored at XBee power-up): 50 milliseconds.

**<latency>**
The slave latency is the number of consecutive connection events which the connected peripheral is allowed to skip before the connection is dropped. `latency` may be between 0 and 500.

Default value (restored at XBee power-up): 0.

**<timeout_ms>**
The connection supervision timeout value is the time that the central device—in this case, the XBee—will wait for a data transfer before assuming that the connection is lost. `timeout_ms` may be between 100 and 32000 (32 seconds). `timeout_ms` must be larger than `2 * timeout_ms * (latency + 1)`.

Default value—restored at XBee power-up: 1000 milliseconds (1 second).
If the configuration could not be updated, an `OSError` is raised.

**security**
The security argument takes a bit mask of the following values:

- **PAIRING_REQUIRE_MITM**: If set, MITM protection must be used during the pairing process. It is an error to attempt to set this flag when no I/O callbacks have been set yet with the `io_callbacks()` method.
- **PAIRING_REQUIRE_BONDING**: Require the use of bonding to enable encryption. When connections are secured bonding table entries are created to remember the negotiated keys.
for future sessions. The device will remember up to 13 bonding table entries. If an attempt to bond with additional devices occurs when the bonding table is full the least recently used bonding table entry is dropped to allow insertion of the new entry.

- **PAIRING_DISABLE_LEGACY**: Disables the use of legacy pairing when securing the connection, only LE Secure Connections will be used.

### isconnected()

Determines whether BLE is connected to a BLE peripheral device.

```python
connection.isconnected()
```

**Return value**

`isconnected` returns `True` if the BLE is connected to BLE peripheral device, `False` otherwise.

### secure()

Performs pairing/bonding on a connection.

```python
secure(secure_cb)
```

**<secure_cb>**

A callback which is called upon completion of the pairing operation. It is passed the value zero if the pairing succeeded, otherwise it is passed an error code as documented in the BLE Core specification. Values between 0-127 are from *Vol 11, Part F*. Values from 128-255 are Pairing Errors and can be seen in *Vol 3, Part H*, section 3.5.5 in Table 3.7.

See also the `security` argument to `ble.config` to guide the behavior of the pairing/bonding operation.

### io_callbacks()

Provide callbacks that define IO capabilities for pairing.

```python
io_callbacks([display_cb=None], [confirm_cb=None], [request_cb=None])
```

**<display_cb>**

Callback to be used to present a passkey to the user.

**<confirm_cb>**

Callback to be used when the user must confirm (Y/N) a passkey. The passkey is provided as an argument. The passkey should be presented to the user and the user's input should be fed back using `ble.passkey_confirm`.

**Note** If you are providing a `display_cb` and `request_cb`, you must provide a `confirm_cb`. Based on the I/O capabilities of the peer it may be necessary to perform either input or confirmation of a passkey.

**<request_cb>**

Callback to indicate that the user must input a passkey value. The user should be prompted to enter a passkey and the passkey provided by the user should be fed back using `ble.passkey_`
**enter.**

*Note* The BLE standards recommend that the passkey be presented to the user as a six digit number padded with leading zeros.

### delete_bondings()
Remove all stored bonding table entries.

```python
def delete_bondings()
```

This function deletes the contents of the bonding table. This puts the device back into the initial state and can be used to provision a device for redeployment or during development for device testing.

### passkey_enter()
Allows user entry of a passkey value.

```python
def passkey_enter(passkey):
```

*<passkey>*  
The numeric value of the passkey provided by the user.

### passkey_confirm()
Allows user confirmation of BLE pairing passkey.

```python
def passkey_confirm(confirmation):
```

*<confirmation>*  
Provide `True` if the passkey provided by the `confirm` is correct, `False` otherwise.

### UUID()
Create a UUID container.

```python
ble.UUID(value)
```

*<value>*  
The `value` parameter can be either:

- A 16-bit integer. For example `0x2893`.
- A 128-bit UUID string. For example `'eb76d48b-a885-4059-b70e-adfc7f33d255'`.

**Return value**
A UUID object containing the passed UUID.  
To read the UUID value, convert the UUID object into a `bytes` or `bytearray` object:
uuid = ble.UUID(0x1010)
uuid_value = bytes(uuid)
# b'\x10\x10'

The size, in bytes, of the UUID value can be determined using the built-in \texttt{len} function:

assert len(ble.UUID(0x1234)) == 2  # 16 bits
assert len(ble.UUID('eb76d48b-a885-4059-b70e-adfc7f33d255')) == 16  # 128 bits

gap\_scan()  
Run a GAP scan/discovery operation to collect advertisements from nearby BLE devices.

\texttt{ble.gap\_scan(duration\_ms[, interval\_us[, window\_us[, oldest=False]]])}

\texttt{<duration\_ms>}

The \texttt{<duration\_ms>} parameter specifies the duration of the GAP scan operation, in milliseconds. To scan indefinitely, set \texttt{<duration\_ms>} to 0.

\texttt{<interval\_us>, <window\_us>}

Use \texttt{<interval\_us>} and \texttt{<window\_us>} to optionally configure the duty cycle. The scanner will run for \texttt{<window\_us>} microseconds every \texttt{<interval\_us>} microseconds. The default interval and window are 1.28 seconds and 11.25 milliseconds, respectively. Both values must be at least 2,500 microseconds (2.5 milliseconds) and no more than approximately 40.96 seconds (40,959,375 microseconds).

\texttt{<oldest>}

Received GAP advertisements are stored in order from oldest (earliest-received) to newest (most-recently-received). When the internal queue of advertisements is full, the default behavior is to discard the oldest advertisement in order to make room for the newly-received advertisement. If your application depends on retaining older advertisements at the expense of losing newer advertisements, set the \texttt{oldest} argument to \texttt{True}. This will cause new advertisements to be discarded if the internal queue is full.

Return value

If the GAP scan operation is started successfully, a \texttt{gap\_scan} object is returned. If the GAP scan operation is not able to be started, an \texttt{OSError} is raised.

gap\_scan methods

The methods available on a \texttt{gap\_scan} object—returned by the \texttt{gap\_scan()} function—are as follows.

\texttt{get()}

Return a list of all received GAP advertisements currently in the internal queue. This list may be empty.
If the GAP scan has timed out—see `<duration_ms>` argument—or `stop()` has been called, this method will return any remaining advertisements, but no new advertisements will be stored.

**any()**
Returns `True` if there are any GAP advertisements in the internal queue, otherwise returns `False`.

**stop()**
Stop the ongoing GAP scan operation. Any already-received GAP advertisements will be retained—see `get()`.

**stopped()**
Returns `True` if the GAP scan operation has been stopped—using `stop()`—or has timed out, otherwise returns `False`.

### gap_scan advertisement format
Received GAP advertisements are formatted as a dictionary, whose entries are as follows:

- **address**: The BLE MAC address of the received advertisement. Formatted as a bytes object.
- **addr_type**: The type of address contained in the `address` field. The possible values are defined as constants on the `digi.ble` module:

<table>
<thead>
<tr>
<th>Constants</th>
<th>Applicable firmwares</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ble.ADDR_TYPE_PUBLIC</code></td>
<td>802.15.4, Zigbee, DigiMesh, Cellular devices</td>
</tr>
<tr>
<td><code>ble.ADDR_TYPE_RANDOM</code></td>
<td>802.15.4, Zigbee, DigiMesh, Cellular devices</td>
</tr>
<tr>
<td><code>ble.ADDR_TYPE_PUBLIC_IDENTITY</code></td>
<td>802.15.4, Zigbee, DigiMesh, Cellular devices</td>
</tr>
<tr>
<td><code>ble.ADDR_TYPE_RANDOM_IDENTITY</code></td>
<td>802.15.4, Zigbee, DigiMesh, Cellular devices</td>
</tr>
</tbody>
</table>

* `ble.ADDR_TYPE_PUBLIC_IDENTITY` and `ble.ADDR_TYPE_RANDOM_IDENTITY` are removed in 802.15.4’s 0x200B release and removed in Zigbee’s 0x300B release. These constants are going to be removed from future releases of DigiMesh and Cellular firmwares so we do not advise using them.

- **connectable**: `True` if the advertising device indicates that BLE central-mode devices may connect to it, `False` otherwise.
- **rssi**: The received signal strength of the advertisement, in dBm.
- **payload**: The raw advertisement payload. Formatted as a bytes object.

### Use gap_scan as an iterator
Instead of calling the `get()` method repeatedly to access received GAP advertisements, the `gap_scan` object may be used as an iterator, in other words, as the target of a for-loop.

Using the `gap_scan` object as an iterator is the preferred means to access the received advertisements, because calling `get()` requires allocating a list and filling the list with each
advertisement, whereas iterating over the \texttt{gap\_scan} object only requires creating one dictionary at a time.

See the following example.

\begin{verbatim}
scan = ble.gap_scan(duration_ms=10000)  # 10 seconds
for advertisement in scan:
    print(advertisement)
\end{verbatim}

\section*{Use \texttt{gap\_scan} as a context manager}

Instead of needing to remember to call \texttt{stop()} when you wish to end a GAP scan operation, you may instead use a \texttt{gap\_scan} object as a context manager. This way, when the \texttt{with} block is exited, the GAP scan operation is automatically stopped. This approach uses less code and is more elegant.

For example, if you want to run a GAP scan until any advertisement whose payload contains a particular byte string is found, you can do this without a context manager as shown here:

\begin{verbatim}
def find_advertisement(search_string):
    scan = ble.gap_scan(duration_ms=0)
    try:
        for adv in scan:
            if search_string in adv["payload"]:  
                return adv
    finally:
        # Make sure to call stop(), even if an exception is raised.
        scan.stop()

adv = find_advertisement(b"Hello, XBee")
\end{verbatim}

If you instead use \texttt{gap\_scan} as a context manager, using the \texttt{with} statement, you do not need a try/finally block, nor do you need to call \texttt{stop()} yourself.

\begin{verbatim}
def find_advertisement(search_string):
    with ble.gap_scan(duration_ms=0) as scan:
        for adv in scan:
            if search_string in adv["payload"]:  
                return adv

adv = find_advertisement(b"Hello, XBee")
\end{verbatim}

\section*{\texttt{gap\_advertise()}}

Start or stop GAP advertisements from the XBee device.

\begin{verbatim}
ble.gap_advertise(interval_us, adv_data=None)
\end{verbatim}

\section*{\texttt{<interval\_us>}}

Start advertising at the specified interval, in microseconds. This value will be rounded down to the nearest multiple of 625 microseconds. The interval, if not \texttt{None}, must be at least 20,000 microseconds (20 milliseconds) and no larger than approximately 40.96 seconds (40,959,375 microseconds).

To stop advertising, set \texttt{<interval\_us>} to \texttt{None}. 
<adv_data>
This is the payload that will be included in GAP advertisement broadcasts. <adv_data> can be a bytes or bytearray object up to 31 bytes in length, or None.
If <adv_data> is empty—for example b""—then GAP advertising will return to the default XBee behavior, which is to advertise the product name—such as XBee3 Zigbee, or the value in ATBI.
If <adv_data> is None or not specified, then the data passed to the previous call to gap_advertise is used, unless there was no previous call or the previous data was empty, in which case the behavior will be as if an empty value—b""—was passed.
Otherwise, <adv_data> should consist of one or more Advertising Data (AD) elements, as defined in the Bluetooth Core Specification Supplement, Part A Section 1.

- Each AD element consists of a length byte, a data type byte, and one or more bytes of data. The length byte indicates how long the rest of the element is, for example a Complete Local Name element with value My XBee would have a length byte 0x08 - 1 byte for type plus 7 bytes for the value.
- The Bluetooth SIG provides the list of defined Advertising Data element types here: https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile/

Be aware that <adv_data> must be formatted as one or more Advertising Data elements in order to be interpreted as a valid Bluetooth Low Energy advertisement by other devices. For example, to advertise the name My XBee:

ble.gap_advertise(200000, b"\x08\x09My XBee")

Return value

gap_advertise returns None—i.e. it has no return value.
If BLE functionality is not currently active—see active()—gap_advertise will raise an OSError.

xbee_connect()

Authenticates to a connected XBee 3 over BLE and provides access to the API Service running on the remote XBee.

xbee_connection = ble.xbee_connect(gap_connection, receive, password[, timeout=10])

<gap_connection>
The <gap_connection> parameter takes the result of calling ble.gap_connect previously and the connected device should be another XBee 3 device offering the BLE API Service.

CAUTION! If using xbee_connect refrain from using characteristics from the API service directly as well because that will corrupt the state maintained by the xbee_connection object.

For more information on the API Service see the user guide for your device, for example the BLE Unlock API frame and XBee API BLE Service.
<receive>
The <receive> parameter takes a callable MicroPython function. The function should accept a single argument and return None. After authentication with the remote XBee device, this callback will be called whenever an API frame from the peer is ready to be processed. The callback will be called with the entire frame as an object of type bytes, including initial delimiter and checksum. The checksum is validated by the internal logic. Bad data from the peer will be dropped with no calls to the callback.

<pASSWORD>
The <password> parameter is the password to be used to authenticate with the remote XBee device.

<timeout>
The <timeout> parameter is a value in seconds. If the authentication has not completed within <timeout> seconds an exception of type OSError will be raised with the value ETIMEDOUT. Acceptable values are within the range of 5-60 seconds.

Return value
On success an object of type xbee_connection is returned.

xbee_connection methods
Currently the xbee_connection object only offers a single method. The send method takes a bytes object as an argument and transmits it to the connected remote peer over the encapsulated gap_connection object. The bytes object should constitute a legal API frame for the remote peer to process. The send method does not validate transmit data.
To ensure that an API frame has the correct format, consult the product user guide, for example: API frame format.

digi.ble samples
On the digidotcom github there are Bluetooth sample programs found here. Read the accompanying README for each sample for guidance on usage.

Generic gap advertising and gap scanning samples
There are two simple Bluetooth samples available. Both of these do not require any additional libraries to run.
The first sample, gap_advertise advertises a new local name for the XBee specified in the sample.
The second sample, gap_scan shows how to scan and interact with advertisements, printing all found advertisements.

Eddystone Beaconing samples
There are two samples available for Eddystone beaconing. Both of which require the provided Eddystone Library.
The first sample, `eddystone_advertise`, which forms and advertises the three types of beacons the library supports.
The second sample, `eddystone_parse` scans for beacons and prints out any Eddystone beacons it discovers.

**iBeacon samples**
There are two samples available for iBeacon, both of which require the provided iBeacon Library.
The first sample is `iBeacon_advertise`, which forms and advertises iBeacon beacons.
The second sample is `iBeacon_scan`, which scans for and prints out any iBeacon beacons it discovers.

**Troubleshooting**

**Fewer advertisements than expected when using gap_scan**

When using the `digi.ble.gap_scan()` function, the default interval and window are 1.28 seconds and 11.25 milliseconds, respectively. This quote from the Bluetooth Core Specification explains these parameters:

> During scanning, the Link Layer listens on a primary advertising channel index for the duration of the scan window, `scanWindow`. The scan interval, `scanInterval`, is defined as the interval between the start of two consecutive scan windows.

In other words, with the default scan interval and window parameters, the XBee device only listens for advertisements for 11.25 milliseconds every 1.28 seconds, which is an effective duty cycle of about 0.9% (11250 / 1280000).

To increase the likelihood of observing any particular advertisement, you may increase the window, decrease the interval, or both. To continuously listen for advertisements, set both parameters to the same value.

Be aware that increasing the BLE GAP scan duty cycle will increase the power consumption of the XBee device, and can have negative impacts on the performance of Zigbee interactions—on XBee 3 Zigbee RF Modules. The `gap_scan` parameters must be adjusted to each deployment's energy consumption and specific networking requirements.
Cellular network configuration module

**Note** This section only applies to the XBee Cellular Modem. See *Which features apply to my device?* for a list of the supported features.

The network configuration module provides network drivers for specific hardware, which you can use to configure the hardware network interfaces.

Configure a specific network interface ................................................................. 115
class Cellular .................................................................................................... 115
Configure a specific network interface

Network services provided by the configured interfaces are available for use from the socket module. For more information about the socket module, see the MicroPython documentation: socket module.

**Note** The Digi version of MicroPython differs from MicroPython regarding the SSL API. The XBee Cellular Modem supports secure sockets via the `usocket.IPPROTO_SEC` option to the `usocket.socket()` constructor, but does not include the `ussl` module for wrapping sockets and providing certificates and keys.

This example shows how to configure a specific network interface:

```python
from machine import UART
import sys, time

def uart_init():
    u = UART(1)
    u.write('Testing from XBee\n')
    return u

def uart_relay(u):
    while True:
        uart_data = u.read(-1)
        if uart_data:
            sys.stdout.buffer.write(uart_data)
        stdin_data = sys.stdin.buffer.read(-1)
        if stdin_data:
            u.write(stdin_data)

        time.sleep_ms(5)

u = uart_init()
uart_relay(u)
```

For information about the cellular class, which provides a driver for the Cellular modem in the XBee, see class Cellular.

### class Cellular

**Note** This section only applies to the XBee Cellular Modem. See Which features apply to my device? for a list of the supported features.

This class provides a driver for the cellular modem in the XBee device. For example:

```python
import network
import time
cellular = network.Cellular()
while not cellular.isconnected():
    time.sleep_ms(50)
print(cellular.ifconfig())

# now use socket as usual
...
```
Constructors
Use the constructor to create an XBee Cellular object.

```
class network.Cellular()
```

Cellular power and airplane mode method
This method determines whether the XBee Cellular Modem is powered on or in airplane mode.

```
cellular.active([mode])
```

Without parameters:
- Returns `True` if the XBee Cellular Modem is powered on.
- Returns `False` if the XBee Cellular Modem is in airplane mode.

With parameters:
- `False`: XBee Cellular Modem enters airplane mode.
- `True`: XBee Cellular Modem leaves airplane mode.

**Note** No changes to the XBee Cellular Modem are made if the parameter matches the current mode.

Verify cellular network connection method
This method determines whether the XBee Cellular Modem is connected to a network.

```
cellular.isconnected()
```

- `True`: The XBee Cellular Modem is connected to a cellular network and has a valid IP address.
- `False`: Otherwise.

Cellular connection configuration method
The `ifconfig()` method reports on the IP addressing. See Check the network connection for details. The `config()` method reports on and allows configuration of the network interface. See Check network connection and print connection parameters for an example.
For additional information about network configuration, see the MicroPython network configuration documentation.

Cellular active scan method
```
cellular.scan(callback=None, *, deep=False)
```

The parameter `deep` is Boolean and optional.
- `deep=True`: A more comprehensive search, but may take more time to run.
- `deep=False`: A quick check, but may return fewer results.
**Note** Refer to the documentation for the **AS** command on your Cellular device for more details on how a "deep" scan may affect your network connectivity.

The parameter **callback** is optional.  
If specified, this **scan()** call is non-blocking, and will return the results in the user-provided callback function.  
The callback function must take one parameter:

- A list of dictionaries with the following keys:
  - **serving_cell**: True if the current Serving Cell, False otherwise.
  - **mcc**: Mobile Country Code.
  - **mnc**: Mobile Network Code.
  - **signal**: Receiving level in dBm.
  - **area**: Location Area Code.
  - **cell_id**: Cell ID.

If the callback is not specified, the **scan()** call is blocking, and will return when the results are retrieved.  
The results will be list of dictionaries as listed above for the callback results.

### Send an SMS message method

This method sends a message to a phone using SMS.

```
cellular.sms_send(phone, message)
```

where:

- **phone**: The phone number of the device to which the message should be sent. This variable can be a string or an integer.
- **message**: The contents of the message. The message should be a string or a bytes object of 7-bit ASCII characters.

Possible return values:

- None: The cellular network acknowledges receipt of the message. The method throws a **ValueError** for invalid parameters.

Throws an **OSError** exception:

- **ENOTCONN**: The cellular mode has not connected.
- **ETIMEDOUT**: If the network doesn’t acknowledge the message in a reasonable amount of time.
- **EIO**: If there was some other error in sending the messages.

### Receive an SMS message method

You can use the **sms_receive()** method on the **network.Cellular()** class to receive any SMS messages that have been sent.
cellular.sms_receive()

This class returns one of the following:

- **None**: There is no message.
- A dictionary with the following keys:
  - **message**: The message text, which is converted to a 7-bit ASCII with extended Unicode characters changed to spaces.
  - **sender**: The phone number from which the message was sent.
  - **timestamp**: The number of seconds since 1/1/2000, which is passed to `time.localtime()` and then converted into a tuple of datetime elements.

**Register an SMS Receive Callback method**

You can use the `sms_callback()` method on the `network.Cellular()` class to register a callback.

```python
        cellular.sms_callback(my_callback)
```

The callback will be called whenever an SMS message is received.

**Note** When a callback is registered, using `cellular.sms_receive()` will raise an error as only one method of SMS delivery is supported at a time.

The callback function must have one parameter:

- A dictionary with the following keys:
  - **message**: The message text, which is converted to a 7-bit ASCII with extended Unicode characters changed to spaces.
  - **sender**: The phone number from which the message was sent.
  - **timestamp**: The number of seconds since 1/1/2000, which is passed to `time.localtime()` and then converted into a tuple of datetime elements.

**Cellular shutdown method**

This method will properly and safely shut down the cellular modem.

```python
        cellular.shutdown(reset=[reset])
```

Where `reset` can be `True` or `False`.

If `reset` is set to `True`, the XBee Cellular will be rebooted after the cellular modem has been shut down.
If `reset` is set to `False`, the XBee Cellular will not be rebooted, but the cellular modem will have been shut down.
If `False`, you would typically use a `machine.reset()` after this command to emulate the `reset=True` option.

**RSRP/RSRQ reporting in MicroPython**

The `network.Cellular()` object contains a `signal()` function.
When called `network.Cellular()` will return a dictionary containing up to three signal quality indicators if they are currently available. The possible entries in the dictionary and their meaning are in the table below.

<table>
<thead>
<tr>
<th>key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsrp</td>
<td>Reference Signal Received Power in dBm</td>
</tr>
<tr>
<td>rsrq</td>
<td>Reference Signal Received Quality in dB</td>
</tr>
<tr>
<td>rssi</td>
<td>Received Signal Strength Indicator in dB</td>
</tr>
</tbody>
</table>

If the module is in a PSM dormant state, the `signal()` function will throw an `ENOTCONN` error. It is possible for the device to be in a state where signal information is not provided in a timely fashion, so the function will raise an OSErrort with value `ETIMEDOUT` should it take longer than five seconds to attempt to retrieve the signal values.

This applies to all XBee Cellular LTE variants, but not to the XBee Cellular 3G Global Embedded Modem.
XBeemodule

The functions in this section are specific to the XBeem device hardware.

AT commands that do not work in MicroPython ......................................................... 121
class XBeemoonXBeeeCellular Modem ........................................................................ 121
XBeemoonmicropython module on the XBeee 3 RF Modules ........................................ 122
AT commands that do not work in MicroPython

The following commands cannot be performed with the `atcmd` function due to the fact that they trigger an action that does not complete immediately.

The AT commands and alternatives listed here only apply to products on which that AT command is available, refer to the user guide for your device.

Pending commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS - Active Scan</td>
<td><code>cellular.scan()</code> method.</td>
</tr>
<tr>
<td>DB - RSSI</td>
<td><code>networking.Cellular().signal()</code> method.</td>
</tr>
<tr>
<td>DN - Discover Node</td>
<td></td>
</tr>
<tr>
<td>ED - Energy Detect</td>
<td></td>
</tr>
<tr>
<td>FS - Filesystem access</td>
<td>Native Python file system methods.</td>
</tr>
<tr>
<td>IS - IO sampling</td>
<td>The <code>Pin class</code>.</td>
</tr>
<tr>
<td>LA - DNS lookup</td>
<td>Perform <code>getaddrinfo()</code> call.</td>
</tr>
<tr>
<td>ND - Network Discovery</td>
<td><code>xbee.discover()</code> function.</td>
</tr>
<tr>
<td>PG - Ping</td>
<td></td>
</tr>
<tr>
<td>SD - Shutdown</td>
<td><code>Cellular shutdown method</code>.</td>
</tr>
<tr>
<td>SW - RSRP</td>
<td><code>networking.Cellular().signal()</code> method.</td>
</tr>
<tr>
<td>IR - Cell module reset</td>
<td></td>
</tr>
</tbody>
</table>

class XBee on XBee Cellular Modem

Note This section only applies to the XBee Cellular Modem. See Which features apply to my device? for a list of the supported features.

Use this function to output information about the XBee device that is hosting MicroPython.

```python
import xbee
x = xbee.XBee()  # Create an XBee object
print(x.atcmd('MY'))
```

Constructors

Use this class to create an XBee Cellular object for the XBee Cellular Modem that is hosting MicroPython.

```python
class xbee.XBee()
```
Methods
Use this method to send an AT command to the XBee Cellular Modem.

```
x.atcmd(cmd[, value])
```

**<cmd>**
The `<cmd>` parameter is a two-character string that represents the command.

For detailed information about the AT commands that you can use with the XBee device, see the AT commands section in the appropriate user guide.

**<value>**
The `<value>` parameter is optional.

- If the `<value>` parameter is NOT set: The function executes the AT command and, depending on the command, returns the result as either a string, bytes object, an integer, or None. Some commands simply return a value; other AT commands, such as special commands and execution commands, change the behavior of the XBee device. For example, `FR` resets the device.
- If the `<value>` parameter is set: You can specify a value in a string, bytearray, or integer format. The function passes the value to set the AT command.

For examples of how to use the AT commands with the XBee device, see XBee device examples.

---

**XBee MicroPython module on the XBee 3 RF Modules**

**Note** This section only applies to the XBee 3 Zigbee RF Module, XBee 3 802.15.4 RF Module, and XBee 3 DigiMesh RF Modules. See Which features apply to my device? for a list of the supported features.

Functions
The `xbee` MicroPython module supports the following functions:

```
xbee.atcmd(cmd[, value])
```

**<cmd>**
The `<cmd>` parameter is a two-character string that represents the command.

**Note** Some AT commands do not work in MicroPython. See AT commands that do not work in MicroPython.

For detailed information about the AT commands that you can use with the XBee device, see the AT commands section in the appropriate user guide.
<value>
The <value> parameter is optional.

- If the <value> parameter is not set: The function executes the AT command and, depending on the command, returns the result as either a string, bytes object, an integer, or None. Some commands simply return a value; other AT commands, such as special commands and execution commands, change the behavior of the XBee device. For example, FR resets the device.
- If the <value> parameter is set: You can specify a value in a string, bytes object, or integer format. The function passes the value to set the AT command.

For examples of how to use the AT commands with the XBee device, see XBeedeviceexamples.

discover()
Use this function to perform a network discovery, which is equivalent to issuing the ND command. The timeout for the discovery is determined by the N? command.

```
xbee.discover()
```

This function accepts no parameters, and returns an iterator yielding a dictionary for each discovered node.

**Note** xbee.discover() returns immediately, but querying the resulting iterator will block execution until a response is available or the discovery times out (as determined by N?). See the xbee.discover() examples for more information.

Discovered node dictionaries can contain the following fields:

- **sender_nwk**: 16-bit network address
- **sender_eui64**: 8-byte bytes object with EUI-64 address
- **parent_nwk**: set to 0xFFFE on the coordinator and routers, otherwise the network address of the end device's parent
- **node_id**: the device's NI value (a string of up to 20 characters, also referred to as Node Identification)
- **node_type**: Value of 0, 1 or 2 for coordinator, router or end device.
- **device_type**: the device's 32-bit DD value (also referred to as Digi Device Type)
- **rssi**: RSSI of the node discovery request packet received by the sending node

**Note** Some of these fields may be excluded depending on what protocol the XBee device is running.

Example output on a Zigbee network:

```
{
    'rssi': -20,
    'node_id': ' ',
    'device_type': 1179648,
    'parent_nwk': 65534,
    'sender_nwk': 41334,
    'sender_eui64': b'\x00\x13\xa2\x00\x92w%',
```
'node_type': 1
}

Example output on a 802.15.4 network:

{
    'rssi': -20,
    'node_id': ' ',
    'sender_nwk': 41334,
    'sender_eui64': b'\x00\x13\xa2\x00\x92w%`',
}

Example output on a DigiMesh network:

{
    'rssi': -20,
    'node_id': ' ',
    'device_type': 1179648,
    'sender_eui64': b'\x00\x13\xa2\x00\x92w%`',
    'node_type': 1
}

**receive()**

The XBee device has a MicroPython receive queue that stores up to four incoming packets. If the device is operating in MicroPython REPL (AP is set to 4) and the receive queue is full, it silently rejects any additional incoming packets:

- On the XBee 3 Zigbee device, the sending node receives a transmission status of 0x24 (Address not found).
- On the XBee 3 DigiMesh or XBee 3 802.15.4 device, the sending node receives a transmission status of 0x00 (Success) in this case.

**Note** DigiMesh does not acknowledge packets at the application level, so if a packet is delivered to the device while the receive queue is full the sender still sees it as a successful transmission. If you need to verify that MicroPython has received data, you need to send an acknowledgment using `xbee.transmit()` and look for that acknowledgment on the sender.

**Note** We recommend calling the `receive()` function in a loop so no data is lost. On devices where there is a high volume of network traffic, there could be data lost if the messages are not pulled from the queue fast enough.

`receive()` is a non-blocking method. Use this function to return a single entry from the receive queue. The format and fields are equivalent to receiving a 0x91 Explicit Rx API frame.

`xbee.receive()`

This function accepts no parameters, and returns either `None` when there is no packet waiting, or a dictionary containing the following entries:

- **sender_nwk**: the 16-bit network address of the sending node. This field is absent on DigiMesh devices, as devices on a DigiMesh network do not have 16 bit addresses.
- `sender_eui64`: the 64-bit address (as a bytearray) of the sending node. If no 64-bit address is present, such as when the sending device on an 802.15.4 network is using 16 bit addressing, this field will have the value `None`.
- `source_ep`: the source endpoint as an integer
- `dest_ep`: the destination endpoint as an integer
- `cluster`: the cluster id as an integer
- `profile`: the profile id as an integer
- `broadcast`: either True or False depending on whether the frame was broadcast or unicast
- `payload`: a bytes object of the payload (intentional selection of bytes object over string since the payload can contain binary data)

Example output:

```python
{
    'cluster': 17,
    'dest_ep': 232,
    'broadcast': False,
    'source_ep': 232,
    'payload': b'Sample payload',
    'profile': 49413,
    'sender_nwk': 63941,
    'sender_eui64': b'\x00\13\xa2\00\92\w%`
}
```

**receive_callback(rx_callback)**

Register a callback function that will be called whenever an RF transmission is received by the device. The callback function should take a single parameter, a dictionary in the same format used by the `receive()` method. Calling `receive_callback(None)` de-registers the receive callback if one is registered.

**Note** The `receive()` method cannot be called while a callback is registered. Attempting to do so raises an exception.

**transmit()**

Use this function to transmit a packet to a specified destination address. This function either succeeds and returns `None`, or raises an exception. Here is a partial list of the exceptions to expect:

- `TypeError`: invalid type for either `<dest>` or `<payload>`
- `ValueError`: Payload is too long. Maximum length depends on whether you are making a unicast or broadcast transmission with or without encryption. Note that application-level encryption is not available in current builds.
- `OSError(EAGAIN)`: Device is not joined to a network (`AI` returns a non-zero value)
- `OSError(EAGAIN)`: temporary issue preventing sending, for example, insufficient buffers, packet already queued for target
- `OSError(EIO)`: general error message for unable to send
xbee.transmit(dest, payload[, source_ep][, dest_ep][, cluster][, profile][, bcast_radius][, tx_options])

**<dest>**
The `<dest>` parameter is the destination address of the message, and accepts any of the following:
- an integer for 16-bit addressing (only available on the XBee 3 Zigbee and 802.15.4)
- an 8-byte bytes object for 64-bit addressing
- the constant `xbee.ADDR_BROADCAST` to indicate a broadcast destination
- the constant `xbee.ADDR_COORDINATOR` to indicate the coordinator (only available on the XBee 3 Zigbee and 802.15.4)

There are multiple ways to create the 8-byte bytes object for 64-bit addressing:
- as a bytestring: `b'\x00\x13\xa2\x00\x41\x74\x07\xa6'`
- using the `bytes()` constructor with a list of decimal values: `bytes([0, 19, 162, 0, 65, 116, 7, 166])`
- using the `bytes()` constructor with a tuple of hex values: `bytes((0x00, 0x13, 0xa2, 0x00, 0x41, 0x74, 0x07, 0xa6))`

**Note** You can also pass a list of hex values or a tuple of decimal values to `bytes()`.

**<payload>**
The `<payload>` parameter should be a string (for example, 'Hello World!') or bytes object (useful for sending binary data).

**<source_ep>**
Optional 8-bit Source Endpoint for the transmission, defaulting to `xbee.ENDPOINT_DIGI_DATA`.

**<dest_ep>**
Optional 8-bit Destination Endpoint for the transmission, defaulting to `xbee.ENDPOINT_DIGI_DATA`.

**<cluster>**
Optional 16-bit Cluster ID for the transmission, defaulting to `xbee.CLUSTER_DIGI_SERIAL_DATA`.

**<profile>**
Optional 16-bit Cluster ID for the transmission, defaulting to `xbee.PROFILE_DIGI_XBEE`.

**<bcast_radius>**
Optional 8-bit value to set the maximum number of hops a broadcast transmission can traverse. Default is 0.

**<tx_options>**
Optional 8-bit bitfield that configures advanced transmission options. See the protocol-specific user guide for TX Options usage.
**Note** All of the optional parameters are keyword-only, and require the following firmware versions or higher:
* XBee 3 Zigbee: version 1007
* XBee 3 802.15.4: version 2004 - Endpoints, cluster ID, profile ID, and broadcast radius can be used on 802.15.4 but are effectively non-functional.
* XBee 3 DigiMesh 2.4: version 3003

**modem_status**

The `modem_status` module provides ways to handle modem status messages within MicroPython. All modem statuses are reported to MicroPython with the exception of hardware reset/watchdog reset. See the documentation for the Modem Status (0x8A) API frame of the device you use for a list of possible status values.

**Note** Hardware/watchdog reset modem statuses are not sent to MicroPython. Use `machine.reset_cause()` instead.

**Note** Modem statuses sent to MicroPython are stored on a queue with limited space. If modem statuses are to be handled `receive()` should be called frequently or a callback should be registered soon after the device boots to avoid missing data.

**modem_status.receive()**

If a modem status is available, the `receive()` function returns its status code. If no modem status is available, the function returns `None`.

The following code polls for modem statuses and prints them as they are received:

```python
import xbee

while True:
    status = xbee.modem_status.receive()
    if status is not None:
        print("Received status: {:02X}".format(status))
```

**modem_status.callback(my_callback)**

Registers a callback that will be called whenever a modem status is received. The callback function must take one parameter, an integer containing the status code.

The following code registers a callback to print modem statuses when they are received:

```python
import xbee

def status_cb(status):
    print("Received status: {:02X}".format(status))

xbee.modem_status.callback(status_cb)
**digi.cloud module**

*Note* This section only applies to the XBee Cellular Modem with firmware version ending in *11* or newer. See *Which features apply to my device?* for a list of the supported features.

The **digi.cloud** module provides interaction with Digi Remote Manager.

- Create and upload data points ........................................................................................................................................... 129
- class DataPoints ................................................................................................................................................................. 129
- Receive a Data Service Device Request .............................................................................................................................. 132
- class device_request .......................................................................................................................................................... 132
- Use the API Explorer to send Device Requests .................................................................................................................. 133
- digi.cloud.Console() object .............................................................................................................................................. 134
Create and upload data points

You can use the `DataPoints` class to create and upload new data streams and data points to your Digi Remote Manager account. To learn more about data streams, see the Digi Remote Manager User Guide.

This example creates a single data point and uploads it to Digi Remote Manager.

```python
from digi import cloud
data = cloud.DataPoints()
data.add("example-stream", 123)
data.send()
```

You can add multiple data points to a stream, and/or uploads points to multiple streams, in a single request.

```python
data = cloud.DataPoints()
data.add("stream1", 1234)
data.add("stream1", 2345)
data.add("stream2", "value")
data.send()
```

If you prefer, you can also use the `digi.cloud` module as follows:

```python
import digi

data = digi.cloud.DataPoints()
```

class DataPoints

Constructor

Use the constructor to create a DataPoints object.

```python
cloud.DataPoints([transport])
```

Optional parameter

- `transport`: The transport method used to deliver the data points. Acceptable values are `digi.cloud.TRANSPORT_TCP` (the default transport) and `digi.cloud.TRANSPORT_UDP`.

If the Digi Remote Manager feature is disabled (bit 0 of ATDO is cleared), this will raise a TypeError indicating that the Remote Manager feature is disabled.

If there are not enough resources available in the system to create the DataPoints object, or your application has created too many DataPoints objects without allowing some to be garbage-collected, an OSError will be raised with an error code of `ENOBUS`. 

Note DataPoints objects using the `digi.cloud.TRANSPORT_UDP` transport are limited to one data point per DataPoints container.

Add a data point method

This method creates a new data point entry inside the DataPoints container.
data.add(stream_name, value[, units][, quality][, description][, location])

**Required parameters**
- **stream_name**: Specifies the data stream name to which this data point is added.
- **value**: The value to assign to this data point. Currently the only supported types are integer, float, and string.

**Optional keyword parameters**
- **units**: A string, specifying the units associated with data on this data stream. If this value is specified, it will overwrite the `units` field of the data stream in Digi Remote Manager. Individual data points do not have units associated with them.
- **quality**: A user-defined 32-bit integer value indicating the quality of the data in the data point.
- **description**: A string, specifying a description of the data point.
- **location**: A tuple of three floating point numbers, indicating the geo-location information of the data point. Geo-location is represented as (latitude in degrees, longitude in degrees, elevation in meters).

**Return value**
None.

If any of the parameters values are of an inappropriate type (such as an integer for stream name), a `TypeError` or `ValueError` is raised indicating the problem.

This method will raise an `OSError` with the error code `ENOSPC` when there is not enough room to add the data point to the upload buffer. The amount of space each data point consumes in the buffer varies based on the length of the string value and how many of the optional parameters are specified.

**Note** DataPoints objects that use the `TRANSPORT_UDP` transport are limited to one data point per `DataPoints` container.

### Upload data to Digi Remote Manager method

This method performs a data point upload of all data that has been created using the `add()` method. This method takes no parameters, and blocks until the data has been uploaded or the specified timeout expires.

```python
data.send([timeout=30])
```

**Optional keyword parameters**
- **timeout**: An integer, number of seconds that the send call is allowed to block. If nonzero and this timeout elapsed without the data being sent, an `OSError` `ETIMEDOUT` is raised, but the object is still considered to be "locked" and new data points cannot be added. See Check the status of a DataPoints object.

**Return value**
None.
If the Digi Remote Manager feature is disabled (bit 0 of ATDO is cleared), this raises a TypeError indicating that the Remote Manager feature is disabled.

If there is no data to be uploaded, an OSError EINVAL is raised.

If a blocking upload fails (due to a network issue or command timeout), an OSError is raised.

If the "Sleepy Digi Remote Manager" feature is being used (ATMO bit 0 is cleared) and the transport selected is TRANSPORT_TCP, this method causes a temporary TCP/SSL connection to be created.

Check the status of a DataPoints object

This method returns a value indicating the status of the most recent send call on a DataPoints object. This method takes no parameters.

```python
data.status()
```

**Return value**

- `digi.cloud.IDLE`: `send` has never been called.
- `digi.cloud.SENDING`: The most recent `send` call is still being processed.
- `digi.cloud.SUCCESS`: The most recent `send` call has succeeded.
- Any other value is a negative `uerrno` value for the most recent `send` call. For example, `uerrno.EIO`.

The life-cycle of a DataPoints object

DataPoints objects are single use, meaning: after calling `send()` on a DataPoints object, the data inside has been erased and the object will no longer be usable beyond the `status()` call. This means you will need to call `DataPoints()`, instantiating a new DataPoints object, each time you have data points to upload.

If calling `send()` on a DataPoints object raises OSError with error code ETIMEDOUT, then for TRANSPORT_TCP data points, the XBee device will hold on to the data and attempt to retry the send internally. The status of the send can be checked with the `status()` call. See Check the status of a DataPoints object. For TRANSPORT_UDP, after a ETIMEDOUT, the send is not retried. You can know that the send has finished by calling `status()` and seeing a value other than `digi.cloud.SENDING`.

Delete a DataPoints object

To free up the resources held by the DataPoints object (if you are seeing OSError exceptions with ENOBUFS), ensure that your MicroPython application has no references to the object anymore so that it can be garbage-collected. Usually you can use the `del` statement to do this.

```python
data.send()
# success
del data
```

The `del` statement is not necessary if the DataPoints object goes out of scope. One example of this is when the DataPoints object was created inside of a function:

```python
def upload_data(value):
    data = cloud.DataPoints()
```
**digi.cloud module**

**Receive a Data Service Device Request**

```python
    data.add("my_stream", value)
data.send()
    # data automatically gets deleted here

upload_data(123)

Note that the DataPoints object will not be garbage-collected if another object holds a reference to the DataPoints object. Placing the DataPoints object inside a container such as a list, tuple, or dictionary will cause this.

```python
    data = cloud.DataPoints()
    my_list = []
    my_list.append(data)  # my_list now keeps 'data' alive
    my_dict = {"data": data}  # my_dict now keeps 'data' alive
    my_tuple = (data, 123)  # same here
```

**Receive a Data Service Device Request**

Using the `device_request_receive()` function, poll for device requests from your Digi Remote Manager account. You can learn more about device requests in the *Digi Remote Manager Programmer Guide* under data_service.

This example is a typical use case. It receives a device request and forms a response to send back to Digi Remote Manager.

```python
    import time
    from digi import cloud

    while 1:
        request = cloud.device_request_receive()
        if request is not None:
            body = request.read()

            # Process the request
            data = b"my data to send back"

            request.write(data)
            request.close()
        time.sleep(5)
```

If you prefer, you can also use the `digi.cloud` module as follows:

```python
    import digi

    request = digi.cloud.device_request_receive()
```

**class device_request**

This class is returned from `digi.cloud.device_request_receive()`.

**Note** If there is no pending request, `device_request_receive()` returns `None`.

The `device_request` class is a file-like object containing the payload of the request that can be read.

A response can be sent back to Digi Remote Manager using the `write` method.
Use the read(size=-1) method
This method reads the payload received from Digi Remote Manager. Read the payload of the request, returns up to size bytes.
If the argument is omitted, None, or negative, data is read and returned until EOF is reached. An empty bytes object is returned if the stream is already at EOF. This call can block.

```python
request.read(size=-1)
```

Use the readinto(b) method
This method reads the payload received from Digi Remote Manager. Read the payload of the request into a pre-allocated, bytearray() object b, and return the number of bytes read.
This call, like read() is a blocking call.

```python
request.readinto(b)
```

Use the write(b) method
This method will allow a response to be written back to Digi Remote Manager. Write the given bytes() or bytearray() object, b, and return the number of bytes written (always equal to the length of b in bytes).
After finishing writing a response, close() should be called to complete the transaction.

```python
Note All data from the request should have been read before issuing a write.
```

```python
request.write(b)
```

Use the close() method
This method is required to finish the device request transaction. Finish the device request transaction, sending a response to Digi Remote Manager.

```python
Note The request should not be written to or read after a close call.
```

```python
request.close()
```

Use the API Explorer to send Device Requests
You can send a Device Request via Web Services within Digi Remote Manager. Follow these steps:

1. Log in to your Remote Manager account (https://remotemanager.digi.com).
2. Go to Documentation > API Explorer.
3. Select Examples > SCI > Data Service > Send Request. Remote Manager automatically creates the necessary code.
4. Replace the device id value with the ID of your device.
digicloud module digicloud.Console() object

5. Replace the target_name value with micropython and enter the data for the device request between the <device_request> XML tags.

```xml
<sci_request version="1.0">
  <data_service>
    <targets>
      <device id="00000000-00000000-00000000-00000000"/>
    </targets>
    <requests>
      <device_request target_name="micropython">
        This is a Device Request sample
      </device_request>
    </requests>
  </data_service>
</sci_request>
```

6. Click Send.

digicloud.Console() object

Note This applies to XBee 3 Cellular Modem modules in firmware x18 and newer.

Class providing access to the Digi Remote Manager console interface. To use the Digi Remote Manager console you must establish a TCP session to Digi Remote Manager.

isconnected() method

Returns a boolean indicating whether a console session is attached from Digi Remote Manager.

read(size) method

Reads from the Digi Remote Manager console. Reads up to size bytes from the object and returns them. As a convenience, if size is unspecified or -1, all bytes until end-of-file (EOF) are returned. Returns the data read, or None if no data is available.

readinto(buf) method

Reads from the Digi Remote Manager console into a buffer object. Reads bytes into a pre-allocated writable bytes-like object and returns the number of bytes read or None if no data is available. This call is non-blocking.

write(buf) method

Write to the Digi Remote Manager console. Send the data in buf to a connected Digi Remote Manager console session. If no console is currently attached the data is discarded. Returns the number of bytes written.

close() method

Close the session tracked by the current Console() object.
The ussl module

Note This section only applies to the XBee Cellular Modem. See Which features apply to my device? for a list of the supported features.

ussl on the XBee Cellular Modem ................................................................. 136
Syntax .............................................................................................................. 136
ussl on the XBee Cellular Modem

The XBee Cellular Modem's implementation of MicroPython provides a stripped-down version of Python3's ssl module using the name ussl. It consists of a single method, wrap_socket(), which you can use to authenticate servers—ensuring they have a certificate signed by given CA—or provide client authentication via a client certificate and key to the server. Some important differences from wrap_socket() on Python3 are:

- You can only wrap a socket created with protocol IPPROTO_SEC. Python3 uses IPPROTO_TCP.
- You can only wrap a socket before calling the connect() method. Python3 allows for opening a socket, performing unencrypted communications, and then upgrading the connection to use TLS, for example, via the STARTTLS command supported in some protocols.
- In Python3, wrap_socket() creates a new ssl.HTTPSConnection object and the original socket.socket remains intact. MicroPython on the XBee Cellular Modem converts the original socket.socket to a ussl.HTTPSConnection with the same methods.
- Python3 allows for including the key with the device's certificate in a single file for the certfile keyword parameter, but MicroPython on the XBee Cellular Modem requires separate files for the certificate and key.
- If specifying a device certificate, you must also provide a ca_certs file.

Syntax

Usage

ussl.wrap_socket(sock, keyfile=None, certfile=None, ca_certs=None, server_side=False, server_hostname=None)

- **sock:** Socket object created with IPPROTO_SEC and not already wrapped.
- **keyfile:** Name of a file containing the private key for certfile (also stored as a Base64 PEM file).
- **certfile:** Name of a file containing this device's public X.509 certificate as a Base64 PEM file. When specifying certfile, you must also specify keyfile and ca_certs.
- **ca_certs:** Name of a file containing a single public X.509 certificate of the trusted certificate authority (CA) for the remote host. Connections with remote devices only succeed if they have a certificate signed by the CA listed in ca_certs. Unlike Python3, which supports multiple certificates in ca_certs, MicroPython on the XBee Cellular Modem only supports a single certificate in this file. In order to authenticate a server not participating in a PKI (using CAs) the server must present a self-signed certificate. That certificate can be used in the ca_certs field to authenticate that single server.
- **server_side:** currently ignored.
- **server_hostname:** reserved for future support of Server Name Indication (SNI).

wrap_socket() returns the wrapped socket object as a SSLSocket object. Filenames are relative to MicroPython’s current working directory, which defaults to /flash and changes via the uos.chdir() method. Use an absolute path like /flash/cert/server.pem to ignore the current working directory when resolving the filename.
digi.gnss module

This section only applies to the XBee 3 Global LTE-M/NB-IoT. See Which features apply to my device? for a list of the supported features.

The `digi.gnss` module provides an interface to use the modem's Global Navigation Satellite System (GNSS) capabilities.

GNSS module methods ........................................................................................................ 138
GNSS examples .................................................................................................................... 139
GNSS module methods

**single_acquisition(callback, timeout=60)**
Request the GNSS location data to be acquired. This returns that location data using the `location_cb` callback registered in the GNSS class constructor.

When this times out the callback is called with **None**.

**Required parameter**

**callback**
A user callback that will be called whenever new location data is requested. See the class methods that follow for details on requesting location data.

The callback function must take one parameter:
A dictionary with the following keys:

- `longitude`: The longitude in decimal degrees.
- `latitude`: The latitude in decimal degrees.
- `altitude`: The altitude in meters.
- `satellites`: The number of satellites used to get this fix.
- `timestamp`: Seconds since 2000.

**Note** On the XBee 3 Global LTE-M/NB-IoT modules, when gathering non-cached location data there will be interruptions to the cellular network because the radio is shared.

**Optional parameter**

**timeout**
The amount of time to try to acquire a location before giving up.

If the timeout is zero, this will callback the cached location value or **None** if a cached value is not available.

**Return value**
This always returns **None**.

If there is already a request active, an **OSError EBUSY** is raised.

**raw_mode(callback)**
Request access to NMEA 0183 data stream.

When a callback has been provided, the callback will be called periodically with NMEA 0183 sentences containing location information.

**CAUTION!** On the XBee 3 Global LTE-M/NB-IoT modules, access to GNSS disables access to the cellular network. The MicroPython application needs to enable and disable raw mode as necessary.
Parameters

Callback

Callback provided by the user, **None** to disable raw mode. The callback function should accept a single bytes-type parameter which provides full NMEA 0183 sentences.

Errors

- **NSError**
  
  **EBUSY** – Error when GNSS is already in use for other functionality – **one_shot**, and so forth.

GNSS examples

We have provided example applications which demonstrate how to use the GNSS module from MicroPython. You can read these examples on GitHub:

- [https://github.com/digidotcom/xbee-micropython/tree/master/samples/gnss_raw](https://github.com/digidotcom/xbee-micropython/tree/master/samples/gnss_raw)
- [https://github.com/digidotcom/xbee-micropython/tree/master/samples/gnss_single](https://github.com/digidotcom/xbee-micropython/tree/master/samples/gnss_single)

This example shows a typical acquisition of location data:

```python
from digi import gnss

def my_location_callback(data):
    if data is None:
        print("The attempt to get a location timed out")
    else:
        print("Got a location: {}".format(data))

gnss.single_acquisition(my_location_callback)
```
Terminal redirection

dupterm(stream_obj, index=0)
**dupterm(stream_obj, index=0)**

---

**Note** This applies to XBee 3 Cellular Modem modules in firmware x18 and newer.

Switch the MicroPython terminal—the REPL—on the given stream-like object. The `stream_obj` 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.

When the XBee AP parameter is not 4—MicroPython—all terminal output is directed to this stream, and any input that is available on the stream is passed on to the terminal input.

The `index` parameter is present for compatibility up stream, but only slot zero is provided.

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

The function returns the previous stream-like object in the given slot.
Use AWS IoT from MicroPython

**Note** This section only applies to the XBee Cellular Modem. See *Which features apply to my device?* for a list of the supported features.

You can use MicroPython to connect an XBee Cellular Modem to the Amazon Web Services (AWS) IoT cloud.

Add an XBee Cellular Modem as an AWS IoT device .............................................................. 143
Create a policy for access control ......................................................................................... 143
Create a Thing ....................................................................................................................... 144
Install the certificates ............................................................................................................. 146
Test the connection ................................................................................................................ 147
Publish to a topic ................................................................................................................... 149
Confirm published data ......................................................................................................... 150
Subscribe to updates from AWS ......................................................................................... 150
Add an XBee Cellular Modem as an AWS IoT device

First, log in to AWS. To do this:

1. If you do not already have one, sign up for a Basic AWS account with twelve months of free tier access.
2. You can add devices and generate certificates, but they might not be able to connect until you receive an email from Amazon confirming that your AWS account is ready.

Create a policy for access control

Once you have an AWS account, log into the AWS IoT Console.

Use the following policy as a starting point for testing. It allows any device with a valid certificate to connect and perform various actions, which you will use for testing your client certificate via HTTPS.

In the left navigation pane, choose Secure, and then Policies. On the You don't have a policy yet page, choose Create a policy; see Create an AWS IoT Policy.

Once there, you can create a policy and enter advanced mode to paste in the following open policy.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "iot:Connect",
            "iot:GetThingShadow",
            "iot:Publish",
            "iot:Receive",
            "iot:Subscribe"
         ],
         "Resource": ["*"]
      }
   ]
}
```

Once you have things working, you can switch to a more restrictive policy that limits a Thing to connecting with its ThingName as its ClientId, and publishing and subscribing only to topics under its type/name in the topic hierarchy.

The client ARNs follow this format:


Note Replace the region and account numbers in the following sample code with your own information.
Create a Thing

From the AWS services page, choose IoT Core.
Use AWS IoT from MicroPython

Create a Thing

In AWS IoT:

1. Click Manage > Things.
2. Click the CREATE button.
3. On the page that says You don’t have any things yet, choose Register a thing.
4. On the Creating AWS IoT things page, choose Create a single thing.
5. In the Name field, give a unique name to your device.
6. In the Thing Type field, choose Create a type.
7. Type XBee_Cellular in the Name field.
8. In the Attribute key field, type IMEI. You can use this IMEI attribute key to identify a specific device if you add multiple devices to the AWS account. Use the ATIM command to get the XBee device’s IMEI.
9. Choose Create thing type.
10. Choose Next to add your device to the registry.
11. Choose Create certificate to use One-click certificate creation to generate a certificate, public key and private key for your device.
12. Download the certificate, public key and private key for this specific device. You will not use the public key file, but this is your only opportunity to download it–you can generate new certificates for your device if you somehow misplace them. You also need the root CA for AWS IoT, this file should be identical for all devices you connect to your account.
13. Once you have downloaded all of the files, choose Attach a Policy to attach the policy created previously. Note that Amazon now recommends using Amazon Trust Service endpoints and recommends using intermediate Root CAs. Some devices such as the XBee 3 Cellular LTE-M Global Smart Modem only work with the originating end of chain Root CA, so use that one instead. Specifically, for ATS endpoints we recommend using the Starfield Services Root Certificate from amazontrust.com/repository/.
14. In the left navigation pane, choose Manage, and then choose Certificates. If the certificate says Inactive on its row, click Activate in the drop-down menu on the right side of the certificate’s row to activate it.

---

1ATS endpoints include -ats as part of the hostname. ATS endpoint <host_prefix>-ats.iot.<aws_region>.amazonaws.com where <host_prefix>-ats is the full hostname and <aws_region> is the region of your endpoint. Legacy endpoints omit the -ats postfix string so the endpoint becomes <host_prefix>.iot.<aws_region>.amazonaws.com.
Install the certificates

Place the downloaded certificates into a folder with a name to match your Thing's name or the 10-character ID used in the filenames that correspond to the start of the certificate's ID shown in the AWS IoT console.

To simplify file management on the XBee device and to allow re-use of the same code on multiple devices, give the files shorter names.

<table>
<thead>
<tr>
<th>Original name</th>
<th>New name</th>
</tr>
</thead>
<tbody>
<tr>
<td>9770fec281-certificate.pem.crt</td>
<td>aws.crt</td>
</tr>
<tr>
<td>9770fec281-private.pem.key</td>
<td>aws.key</td>
</tr>
<tr>
<td>9770fec281-public.pem.key</td>
<td>(unused)</td>
</tr>
<tr>
<td>SFSRootCAG2.pem</td>
<td>aws.ca</td>
</tr>
</tbody>
</table>

Use XCTU or ATFS commands in a terminal emulator to upload the three files to the **cert** directory on the XBee device. For security, use ATFS XPUT to upload the aws.key as a secure file. We recommend using the Starfield Services Root Certificate from [amazontrust.com/repository/](https://amazontrust.com/repository/) as the intermediate CA certificates provided by Amazon do not work on some cellular modules. Note the Verisign certificate is now considered legacy by Amazon.

Many of the intermediate root certificate authorities on the AWS repository ([amazontrust.com/repository/](https://amazontrust.com/repository/)) do not work with the TLS implementation on the XBee Cellular Modems. To ensure that you have success, you need to use a specific Starfield Technologies Root Certificate Authority depending on which XBee you are using.

If you are using one of the CAT 1 XBee Cellular devices, use the **Starfield Class 2 Certification Authority Root Certificate** instead of the ones recommended by Amazon. If you are using the LTE-M/NB-IoT or 3G devices, you need to use the **Starfield Services Root Certificate Authority** certificate. Note that the Amazon certificates are in the trust chain of these two certificates. It has the following SHA-256 thumbprint and can be obtained from Starfield Technologies.

<table>
<thead>
<tr>
<th>XBee 3 Cellular Cat 1 AT&amp;T</th>
<th>Starfield Class 2 Certification Authority Root Certificate</th>
<th>sf-class2-root.crt (PEM)</th>
<th>1465fa205397b876faa6f0a9958e5590e40fc7faa4fb7c2c8677521fb5fb658</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBee 3 Cellular Cat 1 AT&amp;T</td>
<td>Starfield Class 2 Certification Authority Root Certificate</td>
<td>sf-class2-root.crt (PEM)</td>
<td>1465fa205397b876faa6f0a9958e5590e40fc7faa4fb7c2c8677521fb5fb658</td>
</tr>
<tr>
<td>XBee Cellular Cat 1 Verizon</td>
<td>Starfield Services Root Certificate</td>
<td>PEM</td>
<td>2b071c59a0a0ae76b0eadb2bad23bad4580b69c3601b630c2ef0613afa83f92</td>
</tr>
<tr>
<td>LTE-M/3G products</td>
<td>Starfield Services Root Certificate</td>
<td>PEM</td>
<td>2b071c59a0a0ae76b0eadb2bad23bad4580b69c3601b630c2ef0613afa83f92</td>
</tr>
</tbody>
</table>
Test the connection

Update the following samples with settings for your AWS account and the Thing you are testing with, and use it to test your certificates. All samples use the same settings so you can easily paste your configuration to the top of each sample. You can identify the elements of the AWS endpoint (such as host, region, account) and the elements of this Thing (such as Thing type, Thing name).

You can first test with the following code on your computer with Python3 (run from the command line `python aws_https_pc.py`):

```python
# Test code to run from Python3 on a PC

# AWS IoT Account for this Thing
host = b'ABCDEFG1234567-ats'
region = b'us-east-1'
aws_endpoint = b'%s.iot.%s.amazonaws.com' % (host, region)

# This Thing's type and name
thing_type = b'XBee_Cellular'
thing_name = b'IMEI_63890'

import socket, ssl

s = socket.socket()
w = ssl.wrap_socket(s,
    keyfile='cert/aws.key',
    certfile='cert/aws.crt',
    ca_certs='cert/aws.ca')
w.connect((aws_endpoint, 8443))
w.write(b'GET /things/%s/shadow HTTP/1.0\r\nHost: %s\n\r\n' % (thing_name, aws_endpoint))
print(str(w.read(1024), 'utf-8'))
w.close()
```

You should see sample output something like this on your computer:

```
HTTP/1.1 200 OK
content-type: application/json
content-length: 61
date: Thu, 05 Jul 2018 01:24:15 GMT
x-amzn-Requestid: 37e93081-06f5-0bc2-1384-5a129eb0ac30
connection: keep-alive

{"state":{},"metadata":{},"version":1,"timestamp":1530753855}
```

Once you confirm that the certificates and policy on your AWS account are correct, you can test on the XBee device with the following code. It configures the socket as non-blocking in order to return any amount of data read instead of blocking until receiving the full byte count (for exrample, 1024 below).

**Note** It is easiest to use paste mode by pressing **CTRL-E** from the REPL.
# AWS IoT Account for this Thing
host = b'ABCDEFG1234567'
region = b'us-east-1'
aws_endpoint = b'%'s.iot.%s.amazonaws.com' % (host, region)

# This Thing’s type and name
type_type = b'XBee_Cellular'
type_name = b'IMEI_63890'

import usocket, ssl, errno, time

s = usocket.socket(usocket.AF_INET, usocket.SOCK_STREAM, usocket.IPPROTO_SEC)
s.setblocking(False)
w = ssl.wrap_socket(s,
    keyfile='cert/aws.key',
    certfile='cert/aws.crt',
    ca_certs='cert/aws.ca')

while True:
    try:
        w.connect((aws_endpoint, 8443))
        break  # connection complete
    except OSError as e:
        # Nonblocking socket will raise OSError EINPROGRESS
        # until the connection is complete or an error occurs.
        if e.args[0] == errno.EINPROGRESS:
            # Sleep for a moment before checking connection status again.
            time.sleep_ms(100)
        else:
            # Some other error (such as ETIMEDOUT) occurred.
            raise

w.write(b'GET /things/%s/shadow HTTP/1.0\r\nHost: %s\r\n\r
' % (type_name, aws_endpoint))

while True:
    data = w.read(1024)
    if data:
        print(str(data, 'utf-8'))
    break
w.close()

The XBee device includes additional blank lines because the HTTP response uses CRLF for line endings, and starts with the return value of the write() call (in this case, 92 bytes written):

92
HTTP/1.1 200 OK
content-type: application/json
content-length: 61
date: Thu, 05 Jul 2018 19:28:03 GMT
x-amzn-RequestID: 0744caf6-2162-1d4f-c4f9-67a2d7ff2ce9
Publish to a topic

You can use the `umqtt.simple` module to publish data to a topic. This code demonstrates publishing to a topic based on the Thing type and name.

```python
from umqtt.simple import MQTTClient
import time, network

# AWS endpoint parameters
host = b'FILL_ME_IN-at's'  # ex: b'abcdefg1234567'
region = b'FILL_ME_IN'  # ex: b'us-east-1'
aws_endpoint = b'%s.iot.%s.amazonaws.com' % (host, region)
ssl_params = {'keyfile': '/flash/cert/aws.key', 'certfile': '/flash/cert/aws.crt', 'ca_certs': '/flash/cert/aws.ca'}  # ssl certs

conn = network.Cellular()
while not conn.isconnected():
    print("waiting for network connection...")
    time.sleep(4)
print("network connected")

def publish_test(clientId="clientId", hostname=aws_endpoint, sslp=ssl_params):
    # "clientId" should be unique for each device connected
```
c = MQTTClient(clientId, aws_endpoint, ssl=True, ssl_params=sslparams)  
print("connecting...")  
c.connect()  
print("connected")

# topic: "sample/xbee"  
# message: {message: AWS Samples are cool!}  
print("publishing message...")  
c.publish("sample/xbee", '{"message": "AWS Sample Message"}')  
print("published")  
c.disconnect()  
print("DONE")

publish_test()

**Confirm published data**

From the AWS IoT Console, choose Test and subscribe to the topic # to see all messages pushed to your account.

*Note* You will not see old messages, so open the Test console before running the sample code on your device.

You can also navigate to your Thing and choose Activity to monitor when your Thing makes an MQTT connection and then disconnects it.

**Subscribe to updates from AWS**

The XBee Cellular Modem can subscribe to topics published on the AWS server.

---

Copyright (c) 2018, Digi International, Inc.  
Sample code released under MIT License.

**Instructions:**

- Ensure that the umqtt/simple.py module is in the /flash/lib directory on the XBee Filesystem
- Ensure that the SSL certificate files are in the /flash/cert directory on the XBee Filesystem
  - "ssl_params" shows which ssl parameters are required, and gives examples for referencing the files
  - If needed, replace the file paths to match the certificates you’re using
- The policy attached to the SSL certificates must allow for publishing, subscribing, connecting, and receiving
- The host and region need to be filled in to create a valid AWS endpoint to connect to
- The loop that checks for incoming traffic will end after it receives "msg_limit" messages
- Send this code to your XBee module using paste mode (CTRL-E)

- If you want to change any of the params in the method, call the method again
  and pass in the params you want
from umqtt.simple import MQTTClient
import time, network

# AWS endpoint parameters
host = b'FILL_ME_IN-ats'  # ex: b'abcdefg1234567'
region = b'FILL_ME_IN'  # ex: b'us-east-1'

aws_endpoint = b'%s.iot.%s.amazonaws.com' % (host, region)
ssl_params = {'keyfile': '/flash/cert/aws.key',
              'certfile': '/flash/cert/aws.crt',
              'ca_certs': '/flash/cert/aws.ca'}

msgs_received = 0
conn = network.Cellular()
while not conn.isconnected():
    print("waiting for network connection...")
    time.sleep(4)
print("network connected")

# Received messages from subscriptions will be delivered to this callback
def sub_cb(topic, msg):
    global msgs_received
    msgs_received += 1
    print(topic, msg)

def subscribe_test(clientId="clientId", hostname=aws_endpoint, sslp=ssl_params, msg_limit=2):
    # "clientId" should be unique for each device connected
    c = MQTTClient(clientId, hostname, ssl=True, ssl_params=sslp)
    c.set_callback(sub_cb)
    print("connecting...")
    c.connect()
    print("connected")
    c.subscribe("sample/xbee")
    print("subscribed")
    print('waiting...')
    global msgs_received
    msgs_received = 0
    while msgs_received < msg_limit:
        c.check_msg()
        time.sleep(1)
    c.disconnect()
    print("DONE")

subscribe_test()
Time module example: get the current time

Note This section only applies to devices that support the Real Time Clock feature.

Use the time module to get the current time on the cellular network. The XBee Cellular Modem must be connected to the cellular network.
The following examples describe coding the time module.

Retrieve the local time ................................. 153
Retrieve time with a loop .................................. 153
Delay and timing quick reference .......................... 154
Retrieve the local time

This code sample shows how to retrieve the local time. The time format is: year, month, day, hour, second, day of week, day of year.

Note Day of week is 0 - 6 for Monday - Sunday and day of year is 1 - 366.

1. Access the MicroPython environment.
2. At the MicroPython >>> prompt, type import time and press Enter.
3. At the MicroPython >>> prompt, type time.localtime() and press Enter. The current time prints.

Retrieve time with a loop

In this example, you can use the time module to get the current time every five seconds. The code executes in a loop, for a total of five loop iterations. In each iteration, the current local time is printed to the terminal and then pauses for five seconds.

The time format is: year, month, day, hour, second, day of week, day of year.

Note You can copy and paste code from the online version of the Digi MicroPython Programming Guide. Use caution with the PDF version, as it may not maintain essential indentations.

1. Access the MicroPython environment.
2. Copy the sample code shown below:

```python
import time
print("Preparing to print the current time 5 times, once every 5 seconds.")
print("The time format is (year, month, day, hour, second, day, yearday)\n")
for _ in range(5):  # Loop 5 times.
    print(time.localtime())  # Print out the current time.
    print("Pause 5 seconds")
    time.sleep(5)
print("Done!")
```

3. At the MicroPython >>> prompt, press Ctrl+E to enter paste mode.
4. At the MicroPython >>> prompt, right-click and select the Paste option.
5. Once pasted, the code should execute immediately. The sample output below shows the five loops that iterate every five seconds.

Preparing to print the current time 5 times, once every 5 seconds.
The time format is (year, month, day, hour, second, day, yearday)

(2017, 5, 10, 11, 30, 55, 2, 130)
Pause 5 seconds
(2017, 5, 10, 11, 31, 0, 2, 130)
Delay and timing quick reference

The table below contains additional time commands that you can use.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

```python
import time

time.sleep(1)  # sleep for 1 second
time.sleep_ms(500)  # sleep for 500 milliseconds
time.sleep_us(10)  # sleep for 10 microseconds

start = time.ticks_ms()  # get value of millisecond counter
delta = time.ticks_diff(time.ticks_ms(), start)  # compute time difference
```
Cellular network connection examples

**Note** This section only applies to the XBee Cellular Modem. See Which features apply to my device? for a list of the supported features.

You can use MicroPython code to check network connection on the XBee Cellular Modem. The coding samples in the sections below show different methods you can use to check the network connection.

- Check the network connection ............................................................... 156
- Check network connection with a loop .............................................. 156
- Check network connection and print connection parameters ..................... 157
Check the network connection

The `ifconfig()` method returns connection elements: IP address, subnet mask, default gateway and DNS server.

Because cellular connections are point-to-point, the subnet mask and default gateway are always 255.255.255.255 and 0.0.0.0. The XBee Cellular Modem reports 0.0.0.0 for its IP address and DNS server until it completes a connection to the cellular network.

In this sample, the return value options for the `isconnected()` method are:

- **False**: The XBee Cellular Modem is not connected to the cellular network. The IP address reported by `ifconfig()` is 0.0.0.0.
- **True**: The XBee Cellular Modem is connected to the cellular network. All connection elements should be populated.

Note that the connection elements that print depend on the XBee Cellular Modem network configuration.

---

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

---

1. **Access the MicroPython environment.**
2. At the MicroPython `>>>` prompt, type `import network` and press Enter.
3. At the MicroPython `>>>` prompt, type `c = network.Cellular()` and press Enter.
4. At the MicroPython `>>>` prompt, type `c.isconnected()` and press Enter.
   - If the return value is **False**, the cellular connection is not complete. Wait until the red LED on the XBIB board is flashing (or if you have a different board, wait 5 to 10 seconds), and run the command again.
   - If the return value is **True**, the cellular connection is complete.
5. Once the cellular connection is complete, you can print the IP settings. At the MicroPython `>>>` prompt, type `c.ifconfig()` and press Enter to print the settings.

```python
>>> import network
>>> c = network.Cellular()
>>> c.isconnected()
True
>>> c.ifconfig()
('100.96.17.xx', '255.255.255.255', '0.0.0.0', '100.96.17.xx')
```

---

Check network connection with a loop

The code in this example waits for the module to connect to the cellular network and then prints the connection message and module network configuration information.

Note that the connection elements that print depend on the XBee Cellular Modem network configuration.

---

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.
1. Access the MicroPython environment.
2. Copy the sample code shown below:

```python
import network
import time
c = network.Cellular()  # initialize Cellular object
while not c.isconnected():  # return if the module is connected to cellular network
    time.sleep_ms(100)  # delay
    print("It is now connected")
    print("My IP address is", c.ifconfig()[0])
```

3. Press Ctrl+E to enter paste mode.
4. At the MicroPython `>>>` prompt, right-click and select the Paste option. Once pasted, the code should execute immediately.

It is now connected
My IP address is 166.184.xxx.xxx

---

Check network connection and print connection parameters

The code in this example waits for the module to connect to the cellular network and then prints the connection message and the XBee Cellular Modem's connection parameters. Note that the connection elements that print depend on the XBee Cellular Modem network configuration.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

1. Access the MicroPython environment.
2. Copy the sample code shown below:

```python
import network
import time
c = network.Cellular()  # Initialize Cellular object.

# Wait until the module is connected to the cellular network.
while not c.isconnected():
    print("Waiting to be connected to the cellular network...")
    time.sleep_ms(1500)  # Pause 1.5 seconds between checking connection
    print("Module is now connected to cellular network")
    print("Here is a summary of module status:")
    print("IP address:", c.ifconfig()[0])
    print("SIM card number:", c.config('iccid'))
    print("International Mobile Equipment Identity:", c.config('imei'))
    print("Network operator:", c.config('operator'))
    print("Phone number:", c.config('phone'))
```
3. Press Ctrl+E to enter paste mode.
4. At the MicroPython >>> prompt, right-click and select the Paste option. Once pasted, the code should execute immediately.

Waiting to be connected to the cellular network...
Module is now connected to cellular network
Here is a summary of module status:
IP address: 166.184.xxx.xxx
SIM card number: 89014103278193xxxxxx
International Mobile Equipment Identity: 357520070xxxxxx
Network operator: AT&T
Phone number: 1612xxxxxxx
Socket examples

The following sections include code samples for using sockets with the XBee Cellular Modem.

Note This section only applies to the XBee Cellular Modem. See Which features apply to my device? for a list of the supported features.

Sockets ................................................................. 160
Basic socket operations: sending and receiving data, and closing the network connection ......................... 160
Specialized receiving: send received data to a specific memory location .................................................. 162
DNS lookup .................................................................. 163
Set the timeout value and blocking/non-blocking mode ............................................................................. 164
Send an HTTP request and dump the response ......................................................................................... 166
Socket errors .................................................................... 167
Unsupported methods ............................................................................................................................. 167
Sockets

A socket provides a reliable data stream between connected network devices. You must import the usocket module so that you can create and use socket objects.

If you are trying different socket examples and you have not power-cycled the XBee Cellular Modem or cleared the MicroPython volatile memory (RAM), it is not necessary to re-type the following code, as it remains in the memory.

Basic socket operations: sending and receiving data, and closing the network connection

A socket opens a network connection, so that data can be requested by the XBee Cellular Modem. The request is sent to the specified destination, and then received by the module. Once the data communication is complete, you should close the socket to close the network connection.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

Basic data exchange code sample

The following example shows basic data exchange between a computer and a website.

1. Access the MicroPython environment.
2. Copy the sample code shown below.
3. Press Ctrl+E to enter paste mode.
4. At the MicroPython >>> prompt, right-click and select the Paste option.
5. Once pasted, the code should execute immediately.

```
# Import the socket module.
# This allows the creation/use of socket objects.
import usocket

# Create a TCP socket that can communicate over the internet.
socketObject = usocket.socket(usocket.AF_INET, usocket.SOCK_STREAM)

# Create a "request" string, which is how we "ask" the web server for data.
request = "GET /ks/test.html HTTP/1.1\r\nHost: www.micropython.org\r\n\r
"

# Connect the socket object to the web server
socketObject.connect(("www.micropython.org", 80))

# Send the "GET" request to the MicroPython web server.
# A "GET" request asks the server for the web page data.
bytessent = socketObject.send(request)

print("\r\nSent %d byte GET request to the web server."
      % bytessent)

# Single lines can be read from the socket,
# useful for separating headers or reading other data line-by-line.
# Use the "readline" call to do this.
# Calling it a few times will show the
# first few lines from the server's response.
socketObject.readline()
```
socketObject.readline()
socketObject.readline()
# The first 3 lines of the server's response
# will be received and output to the terminal.

print("\nPrinting the remainder of the server's response: \n")
# Use a "standard" receive call, "recv",
# to receive a specified number of
# bytes from the server, or as many bytes as are available.
# Receive and output the remainder of the page data.
socketObject.recv(512)

# Close the socket's current connection now that we are finished.
socketObject.close()
print("Socket closed.")

Response header lines
The first three lines received using the readline() call should look like the following output sample. Note that the date reflects the current system date and time. These three lines are the "response headers" of the server's reply, and include relevant data about the server and the content of the data in the reply.

HTTP/1.1 200 OK
Server: nginx/1.8.1
Date: Tue, 28 Mar 2017 21:31:22 GMT

First line
The first line in the response depends on whether a valid request was sent.

- **Valid request**: If a valid request was sent and it was processed correctly, the first line should always be "HTTP/1.1 200 OK".
- **Invalid request**: If an invalid request was sent, a response similar to "HTTP/1.1 400 Bad Request" is received. This can occur if a typographical error is the original request, or if you do not specify the host in the request with the line "Host: www.example.com".

recv() call
The recv() call receives the remainder of the page data. In this example, the requested page is small, so all of the data remaining after the 3 readline() calls is received in this one call.

Several more "response headers" are visible in the reply to this call, followed by some HTML tags, such as "<!DOCTYPE>" and "<head>". The web page being requested in the example consists only of a header that reads "Test", with text underneath it reading "It's working if you can read this!" This content is visible within the response, all of the content is inside of "<body>" tags, and the header is inside of "<h1>" tags, also visible in the response.

Additional examples
If you want to try this example on other web servers, and see the different responses, you can repeat the previous steps, but replace the following:

- /ks/test.html: This is inside the "request" variable and you can replace it with with "/" or a specific path on a server.
Socket examples

Specialized receiving: send received data to a specific memory location

- **www.micropython.org**: This is inside the "request" variable AND inside the "address" variable and you can replace it with the address of the site you want to test.

**Note** If you have not power-cycled the XBee Cellular Modem, and have not cleared the MicroPython volatile memory (RAM) with a soft reboot, you do not need to re-type lines 2 or 4 of the above example, since you already imported `usocket` and created the socket object. If you power off the XBee Cellular Modem, however, or clear the MicroPython heap with a soft reboot, you need to import `usocket` again and create the socket object again. Any variables you created will also no longer be in memory.

### Specialized receiving: send received data to a specific memory location

You can use the `readinto()` method to receive data from a socket and save it to a buffer, which is a specific memory location for reading and writing data. This is useful for processing data, since processing operations can simply read from the buffer. You must create a buffer object to which the `readinto()` method can write the data.

This method receives data from a socket in the same manner as the `recv()` method, but allows you to specify a buffer location.

In this example, the `readinto()` method performs a read on the socket, and puts the data into buffer that is specified by the user.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

The following example shows how to receive data from a socket and save it to a buffer. The `readinto()` method performs a read on the socket, as can be done with `recv()`, but puts the data into a buffer specified by the user. This is useful for processing data since you can reuse a dedicated buffer for received data, and processing operations can simply read from that buffer.

1. Access the MicroPython environment.
2. Copy the sample code shown below.
3. Press Ctrl+E to enter paste mode.
4. At the MicroPython `>>>` prompt, right-click and select the Paste option.
5. Once pasted, the code should execute immediately.

```python
# Import the usocket module.
import usocket
# Create socket object.
socketObject = usocket.socket(usocket.AF_INET, usocket.SOCK_STREAM)
# Create address variable.
address = ("www.micropython.org", 80)
# Create request variable.
request = "GET /ks/test.html HTTP/1.1\r\nHost:
www.micropython.org\r\n\r\n"
# Create a blank array of bytes in memory, which can be used as a buffer.
buff = bytes object(1024)
# Connect the socket object to the web server specified in "address".
socketObject.connect(address)
# Send the GET request to the MicroPython web server.
```

Digi MicroPython Programming Guide
bytessent = socketObject.send(request)
print("\nSent %d byte GET request to server\n" % bytessent)

print("Waiting on server response...\n")
# Read data from the socket and put it into the buffer we created.
# "readinto" will read as many bytes as fit in the buffer, in this case 1024.
bytesread = socketObject.readinto(buff)
print("%d bytes written to buffer!" % bytesread)
# Print the contents of the buffer, showing that the "readinto" call wrote
# the web server's response into memory.
print("Contents of buffer: \n")
print(str(buff[:bytesread], 'utf8'))
# Close the socket.
socketObject.close()
print("Socket closed.")

DNS lookup
You can use the getaddrinfo() function in the socket module to perform a DNS lookup of a domain name, or retrieve information about a domain name or IP address.

In this example, this code imports the socket module and uses getaddrinfo() to perform a DNS lookup on www.micropython.org. The target port is 80.

For detailed information about getaddrinfo(), see micropython.org/resources/docs/en/latest/wipy/library/usocket.html.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

1. Access the MicroPython environment.
2. Copy the sample code shown below.
3. Press Ctrl+E to enter paste mode.
4. At the MicroPython >>> prompt, right-click and select the Paste option.

```python
import socket
# Return tuple (family, type, proto, canonname, sockaddr)
print(\\nCalling getaddrinfo() for micropython.org on port 80,\\n")
print("this will return information about the host address in the")
print("following format:")
print(\\[family, type, proto, canonname, sockaddr\\n")
print(socket.getaddrinfo('www.micropython.org', 80))

# Return sockaddr, which consists of an IP address and port
print(\\nCalling getaddrinfo(), but returning only the address/port tuple")
print("(\\sockaddr\\") via indexing the output of getaddrinfo().\\n")
print(socket.getaddrinfo('www.micropython.org', 80)[0][-1])

# Return the IP address only
print("Finally, returning ONLY the IP address, via more specific")
print("indexing.\\n")
print(socket.getaddrinfo('www.micropython.org', 80)[0][-1][0])
```
5. Once pasted, the code should execute immediately. The output should be similar to the output shown below.

Calling getaddrinfo() for micropython.org on port 80, this will return information about the host address in the following format:

```
[(2, 1, 0, '', ('176.58.119.26', 80))]
```

Calling getaddrinfo(), but returning only the address/port tuple ("sockaddr") via indexing the output of getaddrinfo().

```
('176.58.119.26', 80)
```

Finally, returning ONLY the IP address, via more specific indexing.

```
176.58.119.26
```

**DNS lookup code output**

The output of the `getaddrinfo()` method call is in the following form: `(family, type, protocol, canonname, sockaddr)`

In the output sample, the fourth line of text includes the output of the `getaddrinfo()` method call.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td><code>&lt;family&gt;</code> An integer that represents the type of connection the socket is using. Represents the <code>usocket.AF_INET</code>, meaning an internet family of connection.</td>
</tr>
<tr>
<td>1</td>
<td><code>&lt;type&gt;</code> An integer that represents the type of connection the socket is using. Represents <code>usocket.SOCK_STREAM</code>, meaning a TCP connection.</td>
</tr>
<tr>
<td>0</td>
<td><code>&lt;protocol&gt;</code> An integer that represents the type of connection the socket is using. Represents <code>usocket.IPPROTO_IP</code>, meaning the IP protocol.</td>
</tr>
<tr>
<td>empty string</td>
<td><code>&lt;canonname&gt;</code> A string that represents the &quot;canonical&quot; name of the host, if it has one. If the host does not have a &quot;canonical&quot; name, an empty string is used.</td>
</tr>
<tr>
<td>176.58.119.26, 80</td>
<td><code>&lt;sockaddr&gt;</code> The IP address and port number of the machine you queried.</td>
</tr>
</tbody>
</table>

**Set the timeout value and blocking/non-blocking mode**

You can set the socket's timeout value using the `settimeout()` module. The timeout value is the amount of time the socket waits for data to become available to read.

The value can be set to one of the following:
- **Non-negative integer**: Defines the length of time for the timeout value. The time is measured in seconds.
- **Floating-point value**: Defines the length of time for the timeout value. The time is measured in seconds.
- **0 (zero)**: Makes the socket non-blocking. The socket returns immediately, regardless of whether there is anything to read.
- **None**: Makes the socket blocking. The socket waits indefinitely for data to become available to read, or waits up until the socket times out or closes.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

The code below shows examples of all these options:

```python
# Import the socket module.
import usocket
# Create socket object.
socketObject = usocket.socket(usocket.AF_INET, usocket.SOCK_STREAM)
# Create address variable.
address = ("www.micropython.org", 80)
# Connect to the server specified in "address".
socketObject.connect(address)
print("Setting socket timeout to 5 seconds.")
# Set the timeout to 5 seconds.
socketObject.settimeout(5)
print("Calling RECV- this will timeout since no data was requested.
")
# Call "recv", even though no data has been requested from the host yet,
# meaning none will be received.
try:
    socketObject.recv(1024)
except OSError as error:
    print("Socket timed out!
")
except:
    print("An error occurred.")

# After 5 seconds, there will be an "ETIMEDOUT" OSError, meaning
# the read timed out. This will not print to the screen since it is
captured by the "except" block.
print("Setting socket timeout to zero (non-blocking)."")
# Set the socket to be non-blocking, by setting the timeout to 0.
socketObject.settimeout(0)
print("Calling RECV- should return immediately with no data.
")
# Call "recv".
try:
    socketObject.recv(1024)
except OSError:
    print("No data to read!
")
except:
    print("An error occurred.")

# The call will return right away.
print("Setting socket mode to "Blocking" meaning it will wait for
# data.")
# NOTE: the method "setblocking" is a shorthand way of setting blocking:
# calling "socketObject.setblocking(False)" is shorthand for calling
# "socketObject.settimeout(0)".
```
# This call will set the socket to be blocking:
socketObject.setblocking(True)
print("Calling RECV with a blocking socket.")
print("This will wait for data, until it either receives it,")
print("the socket times out, or the user cancels the call.
")
print("This call will time out after approximately 60 seconds. If you
don't")
print("feel like waiting to see that happen, feel free to")
print("press Ctrl-C to cancel the RECV call and return to a prompt...")
# Call "recv".
socketObject.recv(1024)

Send an HTTP request and dump the response

You can use the http_get() command to send an HTTP request and then dump the response. You can use the dump_socket() method with any open socket, and it will automatically exit when the remote end closes the connection.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

1. Access the MicroPython environment.
2. Copy the sample code shown below. This code splits a URL into the hostname and path, connects to the server at the host name, and sends a request for the page at the path. The code then prints the response to the screen.

```python
import socket
def http_get(url):
    scheme, _, host, path = url.split('/', 3)
    s = socket.socket()
    try:
        s.connect((host, 80))
        request=bytes('GET /%s HTTP/1.1\nHost: %s\n\n' % (path, host), 'utf8')
        print("Requesting /%s from host %s" % (path, host))
        s.send(request)
        while True:
            print(str(s.recv(500), 'utf8'), end = '')
        finally:
            s.close()

http_get('http://www.micropython.org/ks/test.html')
```

3. At the MicroPython >>> prompt, press Ctrl+E to enter paste mode.
4. At the MicroPython 1=== prompt, right-click and select the Paste option.
5. After pasting the code, press Ctrl+D to finish. You can now retrieve URLs a the MicroPython >>> prompt.
Socket errors

Note This section only applies to the XBee Cellular Modem. See Which features apply to my device? for a list of the supported features.

This following socket errors may occur.

ENOTCONN: Time out error
If a socket stays idle too long, it will time out and disconnect. Attempting to send data over a socket that has timed out produces the OSError ENOTCONN, meaning "Error, not connected." If this happens, perform another connect() call on the socket to be able to send data again.

ENFILE: No sockets are available
The socket.socket() or socket.connect() method returns an OSError (ENFILE) exception if no sockets are available. If you are already using all of the available sockets, this error may occur in the few seconds between calling socket.close() to close a socket, and when the socket is completely closed and returned to the socket pool.
You can use the following methods to close sockets and make more sockets available:

- Close abandoned sockets: Initiate garbage collection (gc.collect()) to close any abandoned MicroPython sockets. For example, an abandoned socket could occur if a socket was created in a function but not returned. For information about the gc module, see the MicroPython garbage collection documentation.
- Close all allocated sockets: Press Ctrl+D to perform a soft reset of the MicroPython REPL to close all allocated sockets and return them to the socket pool.

ENXIO: No such device or address
OSError(ENXIO) is returned when DNS lookups fail from calling usocket.getaddrinfo().

Unsupported methods
The following methods are standard features of the Python socket interface that are not supported on this version of the XBee Cellular Modem.

- setsockopt()
I/O pin examples

**Note** This section only applies to devices that support the Pin I/O feature.

The following sections include code samples for changing the XBee device’s pins.

- Change I/O pins .............................................................. 169
- Print a list of pins ............................................................ 169
- Change output pin values: turn LEDs on and off ......................... 170
- Poll input pin values ......................................................... 170
- Check the configuration of a pin ........................................... 171
- Check the pull-up mode of a pin ........................................... 173
- Measure voltage on the pin (Analog to Digital Converter) ............. 174
Change I/O pins

You can use MicroPython to change the pins on the XBee device. By initializing a pin object, you can change the pin to be an input pin or an output pin.

- If a pin is set up as an output, a pin's output value can be set on or off.
- If the pin is set up as a digital input, you can read the digital value on it.

When initializing a pin, the first argument must be an object within the `machine.Pin.board` module, or a string that matches one of these objects.

For example, in the line of code below, the identifier `P0` refers to the DIO10/PWM0 pin:

```python
dio10 = Pin("P0", Pin.OUT)
```

**Note** You can replace `P0` with `Pin.board.P0` as `P0` is a quoted string and `Pin.board.P0` is an object reference. `Pin.board.P0` only works if you have previously entered `from machine import Pin`.

**Note** MicroPython does not currently support identifying a pin with an integer ID.

The pins available to the system can be seen after importing the `machine` module by typing `dir(machine.Pin.board)`.

Print a list of pins

You can use the `help(Pin.board)` command to print a list of the pins available on the XBee device.

1. Access the MicroPython environment.
2. At the MicroPython >>> prompt, type `from machine import Pin` and press Enter.
3. At the MicroPython >>> prompt, type `help(Pin.board)` and press Enter. The following is a list of available pins.

```python
>>> from machine import Pin
>>> help(Pin.board)
object <class 'board'> is of type type
D0 = Pin(Pin.board.D0, mode=Pin.IN, pull=Pin.PULL_UP)
D1 = Pin(Pin.board.D1, mode=Pin.IN, pull=Pin.PULL_UP)
D2 = Pin(Pin.board.D2, mode=Pin.IN, pull=Pin.PULL_UP)
D3 = Pin(Pin.board.D3, mode=Pin.IN, pull=Pin.PULL_UP)
D4 = Pin(Pin.board.D4, mode=Pin.IN, pull=Pin.PULL_UP)
D5 = Pin(Pin.board.D5, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF5_ASSOC_IND)
D6 = Pin(Pin.board.D6, mode=Pin.IN, pull=Pin.PULL_UP)
D7 = Pin(Pin.board.D7, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF7_CTS)
D8 = Pin(Pin.board.D8, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF8_SLEEP_REQ)
D9 = Pin(Pin.board.D9, mode=Pin.ALT, pull=Pin.PULL_UP, alt=Pin.AF9_ON_SLEEP)
P0 = Pin(Pin.board.P0, mode=Pin.OUT)
P1 = Pin(Pin.board.P1, mode=Pin.IN, pull=Pin.PULL_UP)
P2 = Pin(Pin.board.P2, mode=Pin.IN, pull=Pin.PULL_UP)
```

**Note** The pin list may vary between XBee devices that have different I/O capabilities.
Change output pin values: turn LEDs on and off

You can change the output value of a pin on the XBee device using an "active high" configuration. This means that turning the pin ON turns the LEDs ON, not OFF.

For example, you can change the value of a pin that is connected to some of the LEDs on an XBIB-U-DEV board. The change in pin state is shown by the LEDs being illuminated or not. The pin in the example is connected to three green LEDs in an "active high" configuration.

Note You can copy and paste code from the online version of the Digi MicroPython Programming Guide. Use caution with the PDF version, as it may not maintain essential indentations.

1. Access the MicroPython environment.
2. Copy the sample code shown below.
3. At the MicroPython >>> prompt, press Crti+E to enter paste mode.
4. At the MicroPython >>> prompt, right-click and select the Paste option.
5. Once pasted, the code should execute immediately. The print statements in the code block below print to the terminal.

```
# Import the Pin module
from machine import Pin
print("\nTake note of the 3 green LEDs to the right of the USB port on the"")
print("XBIB-UDEV board, they normally turn on during boot-up."")
print("Creating a pin object for the pin these LEDs are connected to...\n")
# Set up a Pin object to represent pin 6 (PWM0/RSSI/DIO10).
# The second argument, Pin.OUT, sets the pin's mode to be an OUTPUT.
# The third argument sets the initial value, which is 0 here, meaning OFF.
dio10 = Pin("P0", Pin.OUT, value=0)
print("The LEDs should now be OFF, since we set the pin to output \"0\"")
print("For verification, we will check the value of the pin:"")
# After running the above command, the green LEDs should now all be OFF.
# Verify the value of the pin's output by calling the "value" method without
# any parameters.
pinval = dio10.value()
# This should return "0", which is correct given that the LEDs are OFF,
# they are active high, and we set the initial value to be 0.
print("Pin value (retrieved using the \"value()\" method): %d\n" % pinval)
_ = input("Press Enter to change the pin value from 0 to 1.\n")
print("Turning the LEDs ON by setting the pin to 1 with the value() method...")
# Turn the LEDs on.
dio10.value(1)
# The LEDs should turn on and stay on.
print("The LEDs should now be ON!")
```

Poll input pin values

You can use the value() method to check the present value on a pin set up to be in input mode. With polling, you can use MicroPython code to monitor the value of a pin. During polling, the system constantly checks the value of the pin. MicroPython can then perform an action when the value on the pin changes.
I/O pin examples

Check the configuration of a pin

The following example demonstrates a simple loop that waits for the user to press a button on the XBIB board, which is connected to a pin on the XBee device. This sample uses the `value()` method to return the current value on an input pin, and uses polling to monitor a pin.

1. **Access the MicroPython environment.**
2. **Copy the code sample below.** This code imports the `pin` module from the `machine` module and creates a pin object `ad0` to represent pin 20.
   ```python
   from machine import Pin
   ad0 = Pin("D0", Pin.IN, Pin.PULL_UP)
   ```
3. **At the MicroPython `>>>` prompt, press Ctrl+E to enter paste mode.**
4. **At the MicroPython `>>>` prompt, right-click and select the Paste option.**
5. **Copy the code sample below.** This code returns the current value of pin 20. This pin is pulled up on the development board and will read 1 until the SW2 button on the development board is pressed.
   ```python
   ad0.value()
   ```
6. **Press Ctrl+E to enter paste mode.**
7. **At the MicroPython `>>>` prompt, right-click and select the Paste option.**
8. **Copy the code sample below.** This code waits for the SW2 button to be pressed, prints a message, and then exits the program.
   ```python
   while True:
       if ad0.value() == 0:
           print("SW2 has been pressed!")
           break
   ```
9. **Press Ctrl+E to enter paste mode.**
10. **At the MicroPython `>>>` prompt, right-click and select the Paste option.**
11. **Press Enter until "..." is no longer displayed on the left.** The code that was entered is now running. It is waiting for the value of the pin to go from 1 to 0.
12. **Press the SW2 button on the XBIB board.** It is below and left of the RESET button, with the USB port facing you. The terminal should output **SW2 has been pressed!**, then go back to the MicroPython `>>>` prompt on a new line.

Check the configuration of a pin

You can check the configuration of a pin using the `mode()` method when the pin is set up as an input, output, analog, or other function.

The following example shows the basics of these modes.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

1. **Access the MicroPython environment.**
2. **Copy the sample code shown below:**

---

*Digi MicroPython Programming Guide* 171
# Import the Pin module from the machine module from machine import Pin

print("\nChecking the mode of AD0/DIO pin...")
pinmode = Pin.board.D0.mode()
# This should return "0", meaning it is in input mode.
print("AD0/DIO0 is in mode: %d\n" % pinmode)

print("Checking the mode of Associate/DIO5 pin...")
pinmode = Pin.board.D5.mode()
# This should return "2", meaning it is in "ALT" mode by default,
# meaning an alternative function, generally board or port-specific.
print("ASSOC/DIO5 is in mode: %d\n" % pinmode)

print("Creating a pin object for ASSOC/DIO5, setting it as an input...")
d5 = Pin("D5", Pin.IN, Pin.PULL_UP)
print("Checking DIO5's mode using the \"mode\" method...")
pinmode = d5.mode()
# This should return "0", meaning it is an input, which is how it was
# initialized when d5 was created.
print("DIO5 is in mode: ", pinmode)
print("(Note the fact that this pin started out in either ALT or OUTPUT
mode)
print("(value 2 or 1) and is now in input mode (value 0).\n")

print("The modes can be seen by printing the values of the main pin
modes:")
print("Pin.IN: ", Pin.IN) # This should print "0", this is input mode.
print("Pin.OUT: ", Pin.OUT) # This should print "1", this is output mode.
print("Pin.ALT: ", Pin.ALT) # This should print "2", this is ALT mode.
# ALT stands for "alternate", and is usually a port-specific function.
print("Pin.OPEN_DRAIN: ", Pin.OPEN_DRAIN) # This should print "17".
# Open Drain is an output configuration referring to the circuit
positioning
# of the drive transistor.
print("Pin.ANALOG: %d\n" % Pin.ANALOG)
# This should print "3", this is analog mode.

print("Changing the pin DIO5 to be an output, rather than an input, using
the")
print("\"mode\" method...")
d5.mode(Pin.OUT) # Set to output
print("Checking the mode of the pin after the change...")
pinmode = d5.mode() # This should return "1".
print("DIO5 is in mode: %d\n" % pinmode)
print("(Note that value of 1 corresponds to an output, as we set it.\n")
# This means the pin is an output, just as we defined it.
print("(Note that pin DIO5 has held at least 2 different mode values in
this")
print("example, showing the different pin modes and how they can be
changed."")

3. At the MicroPython >>> prompt type Ctrl+L to enter paste mode. The terminal displays paste mode; Ctrl-C to cancel, Ctrl-D to finish.
4. At the MicroPython >>> prompt, right-click and select the Paste option.
5. Once pasted, the code should execute immediately. You should see output showing the different values generated by the print and mode commands.
Check the pull-up mode of a pin

You can use the `pull()` method to check the pull-up mode of a pin. The mode options are:

- **Pin.PULL_UP**: The pin has a default "high" value by connecting it to voltage using a resistor: "pulling up".
- **Pin.PULL_DOWN**: The pin has a default "low" value by connecting it to ground with a resistor: "pulling down".

The following example demonstrates how to check the pull direction of one of the pins on the XBee device and the resultant values on the pin.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

1. **Access the MicroPython environment.**
2. **Copy the sample code shown below.**

```python
# Import the pin module
from machine import Pin

# print("\nChecking the default pull-direction of the AD0/DIO0 pin...")
pinpull = Pin.board.D0.pull()  # This call should return "1", meaning it is set to "PULL_UP".
print("AD0/DIO0 is set to: %d\n" % pinpull)

# print("The two different values for pull direction can be viewed:")
print("Pin.PULL_UP: %d\n" % Pin.PULL_UP)  # This should return "1".
print("Pin.PULL_DOWN: %d\n" % Pin.PULL_DOWN)  # This should return "2".

# Now, make a pin object for pin AD0/DIO0, set as an input, and pulled down to ground (0).
print("Creating a pin object for AD0/DIO0, pulled DOWN...")
d0 = Pin("D0", Pin.IN, Pin.PULL_DOWN)
print("Checking the pull direction of this pin...")
pinpull = d0.pull()  # This should return "2", since it was just set to "PULL_DOWN".
print("Checking the value present on the pin...")
pinval = d0.value()
print("Value on AD0/DIO0: %d\n" % pinval)
print("This should return 0, since the pin is pulled down to ground.\n")

print("Changing the pin mode to be PULL_UP.")
d0.pull(Pin.PULL_UP)
print("Checking the pull direction of this pin...")
pinpull = d0.pull()  # Pull direction of AD0/DIO0: %d\n" % pinpull
print("This should return 1, since it was just set to PULL_UP.\n")
pinval = d0.value()  # Value on AD0/DIO0: %d\n" % pinval
print("This should now return 1 now, instead of 0. This means the pin was")
print("successfully \"pulled up\" to Vdd, or a logic 1.\n")
```

*Digi MicroPython Programming Guide*
# Now that DIO0 is pulled up, we can examine how a pulled-up input works.
# Holding down the button "SW2"/"DIO0", check the value on the pin again.
print("Now we can examine how a pulled-up pin acts when connected to
ground.")
_ = input("Press and hold SW2 on the XBIB board, then press Enter.")
pinval = d0.value()
print("\nValue on DIO0/DIO0: %d" % pinval)
print("The value should now be 0. This is because SW2 connected the pin
to");
print("ground, causing current to flow through the pull-up resistor,
which")
print("dropped the voltage to 0.")

3. **At the MicroPython >>> prompt, type Ctrl+E to enter paste mode.** The terminal displays 
paste mode; Ctrl-C to cancel, Ctrl-D to finish.
4. **At the MicroPython >>> prompt, right-click and select the Paste option.**
5. **Once pasted, the code should execute immediately.** You should see output showing the
different values generated by the pull and value commands.

### Measure voltage on the pin (Analog to Digital Converter)

The device has four ADC inputs available to the user. These channels allow measurement of a
voltage on the pin. The voltage measurement is represented and returned as a 12-bit value, which
is a number between 0 and 4095, where 0 represents 0 V and 4095 represents 2.5 V (XBee
Cellular Modem). For Zigbee, DigiMesh, and 802.15.4 XBee Devices 4095 can represent 1.25 V,
2.5 V or VDD with using the ATAV command (default is 1.25 V)

The following example shows the basics of using ADC.

- **The first `read()` call produces a high value, even though the pin is not connected to anything.**
  This is known as "floating" pin. The high value is caused by voltage being generated at the
  pin from electromagnetic waves coming from other circuits on the board as well as the
electrical power at your location. If a multimeter that is set to measure DC voltage is
connected between the pin and ground, the `read()` method returns a low value, between 0
and 500. Generally a low value is under 100.

- **The second `read()` call is almost always 0, or very close to 0.** This is because the pin is
connected directly to ground by the SW2 button. A multimeter has a high input impedance,
compared to the low (almost zero) impedance of a switch or button.

This example can be repeated with AD1, AD2, and AD3. Just replace "D0" with "D1", "D2", or "D3",
respectively. The button for AD1 is SW3 (DIO1), for AD2 is SW4 (DIO2), and for AD3 is SW5
(DIO3). All four ADC channels work the same way and can all be used at the same time.

**Note** You can copy and paste code from the online version of the *Digi MicroPython Programming
Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

1. **Access the MicroPython environment.**
2. **Copy the sample code shown below:**

   ```python
   # Import ADC from machine, for simpler syntax.
   from machine import ADC

   # Create an ADC object for pin AD0.
   ```
I/O pin examples

Measure voltage on the pin (Analog to Digital Converter)

```python
def main():
    # Perform a read of the analog voltage value present at the pin.
    adc0 = ADC("D0")
    adc_value = adc0.read()
    print("ADC read #1 value: %d" % adc_value)
    print("This will generally return a high value, around 4095,")
    print("but can return any value, since the pin is not connected")
    print("to anything, called "floating").")

    # Now, holding down the SW2/DIO0 button, perform another read.
    _ = input("Press and hold SW2, then press Enter on your keyboard.
")
    adc_value = adc0.read()
    print("ADC read #2 value: %d" % adc_value)
    # This should return a low value, around 0.
    print("This is because the pin was connected to ground, which is")
    print("generally recognized as a 0 volt reference."))

    # If something that output a variable voltage was connected to pin AD0, such as
    # a sensor or transducer, it could be measured by taking the value it
    # returned,
    # dividing it by 4095, and multiplying by the reference voltage.
    
    # For example, if the reference voltage is 2.5VDC, and a 1.0VDC signal is
    # present on the pin, a "read()" call would return approximately 1638,
    # which is
    # equal to (1.0/2.5)*4095.
```

3. At the MicroPython >>> prompt type Ctrl+E to enter paste mode. The terminal displays paste mode; Ctrl-C to cancel, Ctrl-D to finish.
4. At the MicroPython >>> prompt, right-click and select the Paste option.
5. Once pasted, the code should execute immediately. You should see output showing the different values generated by the ADC read commands.
SMS examples

Note This section only applies to the XBee Cellular Modem. See Which features apply to my device? for a list of the supported features.

You can use MicroPython code to send and receive short message service (SMS) messages. You can specify a phone number and send a message of up to 160 characters. A received message includes the phone number from which the message was sent and the message text.

The following sections include code samples for sending and receiving an SMS message from and to the XBee Cellular Modem.

Send an SMS message ................................................................. 177
Send an SMS message to a valid phone number ................................................. 177
Check network connection and send an SMS message ........................................ 177
Receive an SMS message ............................................................................. 178
Receive an SMS message using a callback ......................................................... 179
Send an SMS message

Before you begin sending SMS messages, verify that the XBee Cellular Modem is connected to the cellular network. For information on checking the network connection, see Cellular network connection examples.

You can use the `network.Cellular()` class to send an SMS message from the XBee Cellular Modem. The message consists of the following:

- **Phone number**: The phone number of the device that should receive the message. The phone number can be either a string, such as `'19525551212'` or `'+19525551212'`, or an integer (19525551212).
- **Message**: A message of up to 160 characters.

If the message is sent successfully, `sms_send()` returns `None`. If the message fails, an error message is returned.

Send an SMS message to a valid phone number

The code in this example sends a message to the specified phone number.

**Note** In the example below, replace the sample phone number `1123456789` with a valid mobile telephone number.

1. Access the MicroPython environment.
2. At the MicroPython `>>>` prompt, type `import network` and press `Enter`.
3. At the MicroPython `>>>` prompt, type `c=network.Cellular()` and press `Enter`.
4. At the MicroPython `>>>` prompt, type `c.sms_send('1123456789', 'MicroPython on XBee Cellular is the best!')` and press `Enter`.

Check network connection and send an SMS message

The code in this example waits for the module to connect to the cellular network and then send out the SMS message.

1. Access the MicroPython environment.
2. Copy the sample code shown below:

   The number "***" in the example code must be replaced with the 10-digit mobile telephone number to which you wish to send an SMS message.

   ```python
   import network
   import time

   number = "***" # please fill in the target number
   message = "MicroPython on XBee Cellular is the best!" # Message to sent out

   c = network.Cellular()
   while not c.isconnected():
       print("waiting to be connected to the cellular network...")
       time.sleep_ms(1500) # Pause 1.5 seconds between checking connection
   print("The module is connected to the cellular network. Now send the
   ```
try:
    c.sms_send(number, message)
    print("Message sent successfully to " + number)
except Exception as e:
    print("Send failure: " + str(e))

3. At the MicroPython >>> prompt type Ctrl+E to enter paste mode. The terminal displays paste mode; Ctrl-C to cancel, Ctrl-D to finish.
4. At the MicroPython >>> prompt, right-click and select the Paste option.
5. Once pasted, the code should execute immediately. If the SMS message sends successfully, a message prints.

The module is connected to the cellular network. Now send the message Message sent successfully to "xxxxxxxxxx"

Receive an SMS message
You can use the sms_receive() method on the network.Cellular() class to receive any SMS messages that have been sent. This class returns one of the following:

- None: There is no message.
- Message entry consisting of:
  - message: The message text, which is converted to a 7-bit ASCII with extended Unicode characters changed to spaces.
  - sender: The phone number from which the message was sent.
  - timestamp: The number of seconds since 1/1/2000, which is passed to time.localtime() and then converted into a tuple of datetime elements.

MicroPython only buffers a single received SMS message. If two messages arrive between successive calls to sms_receive(), you will receive only the most recent message.
Before you can receive an SMS message, you should verify that the XBee Cellular Modem is connected to the cellular network. For information on checking the network connection, see Cellular network connection examples.

Sample code
The code in this example commands the device to wait for and then output the incoming SMS message.

Note You can copy and paste code from the online version of the Digi MicroPython Programming Guide. Use caution with the PDF version, as it may not maintain essential indentations.

1. Access the MicroPython environment.
2. Copy the code sample below.

    import network
    import time

    c = network.Cellular()  # Initialize the network parameter object.
def timestamp(t = None):  # Obtain and output the current time.
    return "%04u-%02u-%02u %02u:%02u:%02u" % time.localtime(t)[0:6]

# Check for incoming sms message, output the message if there is any.
def check_sms():
    # Return the incoming message, or "None" if there isn't one.
    msg = c.sms_receive()
    if msg:
        print('SMS received at %s from %s:
             %s' %
             (timestamp(msg['timestamp']), msg['sender'], msg['message']))
        return msg

def wait_for_sms():
    while not check_sms():  # Wait until a message arrives.
        print("Waiting for message...")
        time.sleep_ms(1500)

    wait_for_sms()

3. At the MicroPython >>> prompt type Ctrl+E to enter paste mode. The terminal displays
   paste mode; Ctrl-C to cancel, Ctrl-D to finish.
4. At the MicroPython >>> prompt, right-click and select the Paste option.
5. Press Ctrl+D to compile and run the code. The device starts waiting for an incoming
   message.
6. Once this is running, an SMS message must be sent to the 10-digit phone number
   associated with the XBee Cellular Modem for a message to be received. The received
   message prints, including the time the message was received and the phone number from
   which the message was sent.

Waiting for message...
Waiting for message...
Waiting for message...
SMS received at 2017-05-09 16:53:39 from 2125550199:
hello world

Receive an SMS message using a callback
The code in this example registers a callback to be called when an SMS is received.

import network

def my_callback(sms):
    print("SMS received from %s >> %s" % (sms['sender'], sms['message']))

    cellular = network.Cellular()  # Initialize the network parameter object.
    cellular.sms_callback(my_callback)
XBee device examples

AT commands control the XBee device. The "AT" is an abbreviation for "attention," and the prefix "AT" notifies the modem about the start of a command line. For detailed information about the AT commands that you can use with the XBee device, see the AT commands section in the appropriate user guide.

The \texttt{atcmd()} method first appeared in the \texttt{xbee.XBee()} class on the XBee Cellular products. For the XBee3 Zigbee products and XBee Cellular firmware versions of x0B and later, it is accessible directly from the \texttt{xbee} module, for example, \texttt{xbee.atcmd()}. The \texttt{atcmd()} method can have two parameters.

- The first parameter is the 2-character AT command. If a second parameter is not specified, the command executes the first command and returns the result as an integer, string, or bytes object, depending on the settings in the internal XBee command table.
- Use an optional second parameter to set an AT value to an integer, bytes object or string.

\textbf{Note} See AT commands that do not work in MicroPython for commands that the \texttt{xbee().atcmd()} method does not support on XBee Cellular Modem and XBee 3 Zigbee RF Module.

To perform a network discovery equivalent to an ND command, use the \texttt{xbee.discover()} function.

The following sections include MicroPython AT command code samples you can use with the XBee device.

Print the temperature of the XBee Cellular Modem ................................................................. 181
Print the temperature of the XBee 3 Zigbee RF Module ........................................................... 181
Print a list of AT commands ........................................................................................................ 182
\texttt{xbee.discover()} examples ............................................................................................... 184
\texttt{xbee.transmit()} examples ............................................................................................... 185
Print the temperature of the XBee Cellular Modem

You can use `atcmd()` to read or set AT command parameter values. In this example, the MicroPython code prints the temperature of the XBee Cellular Modem, reports the current IP address of the device, and assigns a value to the DL parameter.

Note You can copy and paste code from the online version of the Digi MicroPython Programming Guide. Use caution with the PDF version, as it may not maintain essential indentations.

1. Access the MicroPython environment.
2. Copy the sample code shown below:

   ```python
   import xbee
   x = xbee.XBee()
   # AT command 'MY' records the current IP address assigned to the module.
   print("Current IP address on module: " + x.atcmd('MY'))
   
   # set 'DL' (destination address parameter) to be "52.43.121.77".
   print("Now set ATDL value to 52.43.121.77.")
   x.atcmd('DL', "52.43.121.77")
   print("Setup succeeds. The default target IP address is: " + x.atcmd('DL'))
   
   # 'TP' records the current temperature measure on the module
   tp = xbee.atcmd('TP')
   if tp > 0x7FFF:
       tp = tp - 0x10000
   print("The XBee is %.1F degrees" % (tp * 9.0 / 5.0 + 32.0))
   ```

3. At the MicroPython `>>>` prompt type Ctrl+E to enter paste mode. The terminal displays paste mode; Ctrl-C to cancel, Ctrl-D to finish.
4. At the MicroPython `>>>` prompt, right-click and select the Paste option.
5. Once pasted, the code should execute immediately. You should see a list of the items generated by the print command:

   - Current IP address on module: 100.65.176.112
   - Now set ATDL value to 52.43.121.77.
   - Setup succeeds. The default target IP address is: 52.43.121.77
   - The XBee Cellular is 111.1

Print the temperature of the XBee 3 Zigbee RF Module

You can use `atcmd()` to read or set AT command parameter values. In this example, the MicroPython code prints the temperature of the XBee Cellular Modem, reports the current address of the device, and assigns a value to the DL parameter.

Note You can copy and paste code from the online version of the Digi MicroPython Programming Guide. Use caution with the PDF version, as it may not maintain essential indentations.

1. Access the MicroPython environment.
2. Copy the sample code shown below:
import xbee
# AT commands 'SH' + 'SL' combine to form the module's 64-bit address.
addr64 = xbee.atcmd('SH') + xbee.atcmd('SL')
print("64-bit address: " + repr(addr64))

# AT Command 'MY' is the module's 16-bit network address.
print("16-bit address: " + repr(xbee.atcmd('MY')))
XBee device examples

Print a list of AT commands

import xbee
at_cmds = {
    "01. Network": ["CE", "ID", "ZS", "CR", "NJ", 
                     "NW", "JV", "JN", "DO", "DC"],
    "02. Operating Network": ["AI", "OP", "OI", "CH", "NC"],
    "03. Security": ["EE", "EO", "KY", "NK", "KT", "I?"],
    "04. Addressing": ["SH", "SL", "MY", "MP", "DH", 
                        "DI", "NI", "NH", "BH", "AR", 
                        "DD", "NT", "NO", "NP"],
    "05. Zigbee Addressing": ["TO", "SE", "DE", "CI"],
    "06. RF Interfacing": ["PL", "PP", "SC", "SD", "DB"],
    "07. UART Interface": ["BD", "NB", "SB", "AP", "A0", 
                            "RO", "D6", "D7", "P3", "P4"],
    "08. AT Command Options": ["CT", "GT", "CC"],
    "09. MicroPython Options": ["PS"],
    "10. Sleep Modes": ["SM", "SP", "ST", "SN", "SO", 
                         "WH", "PO"],
    "11. I/O Settings": ["DO", "D1", "D2", "D3", "D4", 
                          "D5", "D6", "D7", "D8", "D9", 
                          "P0", "P1", "P2", "P3", "P4", 
                          "P5", "P6", "P7", "P8", "P9", 
                          "PB", "PD", "LT", "RP"],
    "12. I/O Sampling": ["IR", "IC", "V+"],
}

print("Here is a summary of all AT values:
")
for category, cmds in sorted(at_cmds.items()):
    print("\n\n{}: ".format(category))
    for cmd in cmds:
        try:
            value = xbee.atcmd(cmd)
        except KeyError:
print("Invalid command:", cmd)
else:
    if (type(value) is int) and (value > 0xF):
        print(cmd, '=', hex(value))
    else:
        if type(value) is bytes:
            # Format Bytes as colon-delimited
            value = ':'.join('%02x' % b for b in value)
        print(cmd, '=', value)

3. At the MicroPython >>> prompt type Ctrl+E to enter paste mode. The terminal displays paste mode; Ctrl-C to cancel, Ctrl-D to finish.

4. At the MicroPython >>> prompt, right-click and select the Paste option.

5. After you press Ctrl+D to compile and run the code, a list of AT commands and I/O parameter values is printed:

Here is a summary of all AT values:
PH = xxx
S# = xxx
IM = xxx
MN = Verizon
MV = xxx
DB = 93
AM = 0
(...) [truncated for brevity]

**xbee.discover() examples**

Since the call to xbee.discover() returns an iterator which will block each time it is queried, the way that elements in the returned list are accessed can affect the timing of the application. The following examples shows two ways you can use xbee.discover() (the examples assume an N? time of 10 seconds).

**Handle responses as they are received**

Using xbee.discover() as the iterator in a for loop will handle each response as it is received.

1. Access the MicroPython environment.
2. Copy the sample code shown below.

```python
import xbee
for i in xbee.discover():
    print(i)
```

3. At the MicroPython >>> prompt type Ctrl+E to enter paste mode. The terminal displays paste mode; Ctrl-C to cancel, Ctrl-D to finish.

4. At the MicroPython >>> prompt, right-click and select the Paste option.

5. Press Ctrl+D to run the code.

Running the above code prints out each response as it is received over the course of 10 seconds. Keep the processing for each response (in other words the code in the for loop) to a minimum to avoid missing responses.
Gather all responses into a list
Calling `list(xbee.discover())` will block until the discovery completes and return a list of all responses found.

Access the MicroPython environment.

1. Copy the sample code shown below.

```python
>import xbee
for i in list(xbee.discover()):
    print(i)
```

2. At the MicroPython `>>>` prompt type `Ctrl+E` to enter paste mode. The terminal displays paste mode; `Ctrl-C` to cancel, `Ctrl-D` to finish.
3. At the MicroPython `>>>` prompt, right-click and select the Paste option.
4. Press `Ctrl+D` to run the code.

Running the above code will wait for 10 seconds then print out a list of all the responses that were received during that time. This method has less chance of missing a response due to processing, but uses more memory at run time as it has to keep track of all the responses at once.

**xbee.transmit() examples**

**xbee.transmit() using constants**
In this example, the MicroPython code transmits a broadcast message using the `xbee.ADDRESS_BROADCAST` constant.

`Note` You can copy and paste code from the online version of the *Digi MicroPython Programming Guide*. Use caution with the PDF version, as it may not maintain essential indentations.

1. Access the MicroPython environment.
2. Copy the sample code shown below:

```python
import xbee
test_data = 'Hello World!'
xbee.transmit(xbee.ADDR_BROADCAST,test_data)
```

3. At the MicroPython `>>>` prompt type `Ctrl+E` to enter paste mode. The terminal displays paste mode; `Ctrl-C` to cancel, `Ctrl-D` to finish.
4. At the MicroPython `>>>` prompt, right-click and select the Paste option.
5. Once pasted, the code should execute immediately. If the transmission attempt is successful, the MicroPython prompt is returned or else the appropriate error message is displayed.

**xbee.transmit() using byte string**
In this example, the MicroPython code transmits a broadcast message using the `xbee.ADDRESS_BROADCAST` constant.
1. Access the MicroPython environment.
2. Copy the sample code shown below:

```python
import xbee
test_data = 'Hello World!'
xbee.transmit(b'\x00\x13\xa2\xff\xad\x95\x5a\xa8', test_data)
```

3. At the MicroPython >>> prompt type Ctrl+E to enter paste mode. The terminal displays paste mode; Ctrl-C to cancel, Ctrl-D to finish.
4. At the MicroPython >>> prompt, right-click and select the Paste option.
5. Once pasted, the code should execute immediately. If the transmission attempt is successful, the MicroPython prompt is returned or else the appropriate error message is displayed.