Digi MicroPython

Programming Guide

Revision history—90002219

Revision	Date	Description
F	June 2018	Removed some information for the Cellular 0B firmware release. Added file system support for XBee Cellular devices. Documented the OSError**(ENXIO)**.
G	July 2018	Added AWS support for Cellular devices.
H	October 2018	Updated **Receive an SMS message method**. Fixed a syntax error in the Pizza Cert example.
J	November 2018	Added information on the relay module.
K	November 2018	Added instructions to develop applications.
L	December 2018	Updated AWS instructions to point to Starfield certificates.

Trademarks and copyright

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 F) in the subject line of your email.

Contents

Keyboard shortcuts

Differences between MicroPython and other programming languages

Develop applications on an XBee device

Power management with MicroPython

Access the primary UART

REPL (Read-Evaluate-Print Loop) examples

Access file system in MicroPython

Send and receive User Data Relay frames

MicroPython libraries on GitHub

The ussl module

Use AWS IoT from MicroPython

Time module example: get the current time

Cellular network connection examples

Socket examples

I/O pin examples

SMS examples

AT command examples

MicroPython modules

Machine module

Cellular network configuration module

XBee module

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 Cellular Embedded Modem User Guide*
- *Digi XBee Cellular 3G Global Embedded Modem User Guide*
- *Digi XBee3 Cellular LTE Cat 1 Smart Modem User Guide*
- *Digi XBee3 Cellular LTE-M Global Smart Modem User Guide*
- *XBee3 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. This table covers which features apply to specific products:

Feature	XBee/XBee3 Cellular	XBee3 Zigbee
Syntax error	Applicable	Applicable
Name error	Applicable	Applicable
OSError	Applicable	Not applicable
OSError on XBee3 Zigbee RF Module	Not applicable	Applicable
Socket errors	Applicable	Not applicable
Power management with MicroPython	Supported	Not supported
Access the primary UART	Supported	Supported[1]
REPL (Read-Evaluate-Print Loop) examples	Supported	Supported
Run stored code at start-up to flash LEDs	Supported	Not supported
Access file system in MicroPython	Supported	Not supported
Send and receive User Data Relay frames	Supported	Supported
MicroPython libraries on GitHub	Supported	Not supported
The ussl module	Supported	Not supported
Use AWS IoT from MicroPython	Supported	Not supported
Time module example: get the current time	Supported	Not supported
Cellular network connection examples	Supported	Not supported
Socket examples	Supported	Not supported
I/O pin examples	Supported	Not supported
SMS examples	Supported	Not supported

[1]The example in this section is not supported.

Feature	XBee/XBee3 Cellular	XBee3 Zigbee
AT command examples	Supported	Supported
MicroPython modules	Supported	Supported
Machine module	Supported	Supported
Class ADC: analog to digital conversion	Supported	Not supported
Class I2C: two-wire serial protocol	Supported	Not supported
Class Pin	Supported	Not supported
Class UART	Supported	Not supported
Cellular network configuration module	Supported	Not supported
XBee module	Supported	Supported

Use MicroPython

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.

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 tabs instead of spaces when indenting lines of code to minimize source code byte count.
- 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 XBee3 Zigbee RF Module:

- The Micropython **time.time()** function returns the number of seconds since the epoch. The XBee3 Zigbee RF Module does not have a realtime clock, so it does not support **time.time()**. To track elapsed time, use **time.ticks_ms()**.

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

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.

```
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):**).

```
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.

```
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

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".

```
    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:

```
    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.

```
    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.

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

```
        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

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.

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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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:

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

Semicolons

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

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

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

```
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 excample:

```
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.

```
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.

```
// 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.

```
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

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 section requires the ***10** firmware release or later.

Space allocated to MicroPython

The XBee Cellular Modem 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 (on external SPI flash): 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.

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.

Run code at startup

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**.

```
>>> # press CTRL-RLoading /flash/main.mpy...
Running bytecode...
Hello, world!

>>> # press CTRL-R
Running bytecode...
Hello, world!
```

As you can see above, it loaded from **/flash/main.mpy** the first time, but the second time it re-ran the same code.

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.

```
>>> 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:

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

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

```
>>> 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=Bhhhh=Bh=h=h=Bh=hhh=h=h==Bh=h=h=h==.h=h=......h=..............
       (30 lines all free)
07c00: .................
>>> 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=Bhhhh=Bh=h=h=Bh=Bhh=h=h==Bh=h=h=h==Bh=h=h=h=MDh=h=Bh=hh=h=Bh==
00400: DDSSSh=h=BBSBhBhBBBBh===h===T=BBh=======h=====B==h====BSh=h=h=h=
00800: =h=......h=..S.....................................................
00c00: .................h=======.............................h=======
01000: ===.............Sh=.............................h=======....
01400: ............................Sh=h=...h=h=...h=.....h=..........
01800: .....h=.................................h=h==........hh=.......
01c00: ...............................................h=.............
02000: ...h=..h.............................................Sh=======..
02400: ...................................................h=........
02800: .......h=.....h=.....h=.............................h=====h===
02c00: =h=h=============================================h=====hShShShShS
03000: hh=hh=hh=hh=hh=hh=hh==hh==hh====hh=hh==h.......................
       (18 lines all free)
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==h=h=..h=.h=........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: ............................S.............................
01800: .............................................h==........hh=.......
01c00: ...............................................h=.............
02000: .......h.............................................Sh=======..
       (2 lines all free)
02c00: ...h=============================================h=====hShShShShS
03000: hh=hh=hh=hh=hh=hh=hh==hh==hh====hh=hh==h.......................
       (18 lines all free)
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.

Information reported by micropython.qstr_info()	
n_pool	number of QSTR pools allocated
n_qstr	number of QSTRs allocated
n_str_data_bytes	combined size of QSTR contents
n_total_bytes	total bytes used by the QSTR contents and pool overhead

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.

```
>>> 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
Q(port)
Q(proto)
Q(https:)
Q(:)
Q(s)
Q(wrap_params)
Q(Host)
Q(Host: %s

)
Q(k)
Q(: )
# [...30 deleted QSTR entries...]
Q(method)
Q(url)
Q(data)
Q(headers)
Q(stream)
Q(verify)
Q(cert)
Q(scheme)
Q(host)
Q(http:)
```

Efficient coding

Follow recommendations from the MicroPython documentation on Maximising MicroPython Speed.

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.

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:

```
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 limitation 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.

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.

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

Power management with MicroPython

Sleeping with AT commands: 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.

When the XBee Cellular Modem enters deep sleep mode, any MicroPython code currently executing is suspended until the device comes out of sleep. When the XBee Cellular Modem 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, uploading data to a web server. The following example shows how to use **XBee().wake_lock**:

```
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
```

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.

Initiate sleep 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.

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.

```
sleep_now(timeout_ms, pin_wake=False)
```

Returns the number of milliseconds elapsed. If **pin_wake** is set to **True**, the device only goes to sleep if DIO8 is pulled high. The device wakes up early if DIO8 goes low before **timeout_ms** elapsed.

Throws an **EALREADY** OSError exception if **SM** is already configured for sleep (set to something other than **0**).

```
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:

```
# Sample showing MicroPython sleeping to conserve power.  Only initiates
# cellular connection if there's been an alarm condition or it's been
# an hour since the last connection.
from machine import Pin
import network
import time
import xbee

alarm_enabled = False
alarm = False
last_upload = 0

def read_sensors():
       # add code here to actually read sensors
       # might set the alarm for unusual readings
       print("read sensors")

def set_alarm_condition():
       global alarm
       print("alarm!")
       alarm = True

def send_data():
       global alarm, last_upload
       # add code here to connect to a server to upload sensor readings
       print("sent data and reported any alarms")
       alarm = False
       last_upload = time.ticks_ms()

def time_for_upload():
       global alarm, last_upload

       # upload if there's been an alarm, or it's been an hour since last upload
       return alarm or (time.ticks_diff(time.ticks_ms(), last_upload) > 60 * 60 *
1000)

def upload_then_sleep():
       c.active(True)           # establish cellular connection
       print("waiting for connection...")
       while not c.isconnected():
              time.sleep(1)        # wait for cellular connection
       send_data()
       c.active(False)          # shut down cellular connection

c = network.Cellular()
x = xbee.XBee()

x.atcmd('SM', 0)     # make sure MicroPython fully controls sleep

# must configure DIO8 as an input with pullup if we want to read and use it
for waking
dio8 = Pin('D8', Pin.IN, Pin.PULL_UP)
```

```
# upload data then put module to sleep
upload_then_sleep()

while True:
       read_sensors()
       if time_for_upload():
              upload_then_sleep();

       if not alarm_enabled and dio8():
              print("DIO8 high, re-enabling alarm")
              alarm_enabled = True

       # sleep for 60 seconds, wake early if DIO8 is low
       print("sleeping for 60 seconds")
       sleep_ms = x.sleep_now(60000, alarm_enabled)

       print("slept for %u ms" % sleep_ms)
       if x.wake_reason() is xbee.PIN_WAKE:
              print("woke early on DIO8 low")
              set_alarm_condition()
              alarm_enabled = False
```

Sleeping with AT commands: XBee3 Zigbee RF Module

Note This section only applies to the XBee3 Zigbee RF Module. See Which features apply to my device? for a list of the supported features.

When the XBee3 Zigbee RF Module enters deep sleep mode, any MicroPython code currently executing is suspended until the device comes out of sleep. When the XBee3 Zigbee RF Module comes out of sleep mode, MicroPython execution continues where it left off.

Access the primary UART

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 sys.stdin supports standard stream methods read and readline in text mode, converting carriage return (**\r**) to newline (**\n**).

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 (**\r**) before each newline (**\n**). 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

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 sample code is handy for debugging the secondary UART. It simply relays data between the primary and secondary UARTs.

```
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)
```

You only need to call **uart_init()** once.

Call **uart_relay()** to pass data between the UARTs.

Send **Ctrl-C** to exit relay mode.

When done, call **u.close()** to close the secondary UART.

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 http://docs.micropython.org/en/latest/pyboard/reference/repl.html.

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:

```
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
>
```

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

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 REPLcommand 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.

```
MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.
>>> a = "test"
>>> a
'test'
>>> <Ctrl-B>
MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.
>>> a
'test'
>>>
```

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:

```
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.
7. A MicroPython **>>>** prompt displays on a new line.

```
MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
   1=== while True:
   2===     pass # This statement means "do nothing"
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt:
>>>
```

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:

```
    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**.

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

```
MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
Type "help()" for more information.
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
   1=== print("Hello world")
Hello world
```

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:

```
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.

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-g00000000 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-g00000000 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]?
```

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 **Crtl+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.

```
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, run **xbee.atcmd('FR')**.

Access file system in 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.

Directory and file names follow the rules in Paths.

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.ilistdir([*dir*])

This function returns an iterator which then yields tuples corresponding to the entries in the directory that it is listing. With no argument it lists the current directory, otherwise it lists the directory given by **dir**. The tuples have the form (**name**, **type**, **inode**, **size**):

- **name**: 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.

uos.replace(*old_path*, *new_path*)

Replace a file or directory (**new_path**) with another (**old_path**).

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.

```
>>> 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.ilistdir())
[('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:

```
>>> from ubinascii import hexlify
>>> digest = os.hash('cert/client.key')
>>> hexdigest = hexlify(digest)
>>> digest

b'\r\x85\xdbY\x0b\xfd\r\x00\x1aI\x08\xb8\x19\xd3\xb8\xa0\x03f\x85\x0fh\xb9\xc9\x1
f\x92;\xd8\xab\xa2\x0f\xfb\x16'
>>> hexdigest
'0d85db590bfd0d001a4908b819d3b8a00366850f68b9c91f923bd8aba20ffb16'
```

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:

Character	Meaning
'r'	Open for reading (default)
'w'	Open for writing, truncating **file** file
'x'	Open for exclusive creation, failing if the file already exists
'a'	Open for writing, always appending to the end of the file
'b'	Binary mode
't'	Text mode (default)
'+'	Open a disk file for updating (reading and writing)
'*'	(XBee extension) open a secure file for writing

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.

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.

```
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 Cellular Modem 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 Cellular Modem 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 Cellular Modem.

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')**.

Send and receive User Data Relay frames

Note This section applies to the XBee Cellular Modem and the XBee3 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

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).

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. 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.
- File Transfer Protocol (FTP). Micro File Transfer Protocol client.
- MQ Telemetry Transport (MQTT). MQTT client for publish/subscribe. See Publish to a topic.
- Digi Remote Manager. An HTTP client for Digi Remote Manager.

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

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.SSLSocket** object and the original **socket.socket** remains intact. MicroPython on the XBee Cellular Modem converts the original **socket.socket** to a **ussl.SSLSocket** 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.

Sample code

This sample code makes use of a CloudFlare demonstration page using the hostname **auth.pizza**. You can read about it on this page. The steps to connect with the client certificate and retrieve the JSON response are:

1. Download the **pizza.pem** file from this page, (download link) and split it into two separate files: **pizza-client.pem** for the public certificate and **pizza-client.key** for the secret key.

Note You can download a zip file of the following three certificate files here and use them instead of copying and pasting from this example.
Use Ctrl+E: Enter paste mode to make copying and pasting easier.

pizza-client.pem

-----BEGIN CERTIFICATE-----

```
MIIEBDCCAuygAwIBAgIUBz8Dd38XnixDcZrowRqrdueQ7sUwDQYJKoZIhvcNAQEL
BQAwdjELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcT
DVNhbiBGcmFuY2lzY28xEzARBgNVBAoTCkNsb3VkRmxhcmUxJTAjBgNVBAsMHENs
b3VkRmxhcmUgVGVzdCBDbGllbnQgQ0EgIzEwIBcNMTcwMzIyMTYzMjAwWhgPMjEx
NzAyMjYxNjMyMDBaMIGIMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5p
YTEWMBQGA1UEBxMNU2FuIEZyYW5jaXNjbzEZMBcGA1UEChMQQ2xvdWRGbGFyZSwg
SW5jLjEZMBcGA1UECxMQVGVzdCBDZXJ0aWZpY2F0ZTEWMBQGA1UEAwwNU1NMIENs
aWVudCAjMTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMi7up5GuGop
kJXaXXQuQtncsX/SJWVOEyN1ZXJOLkkL1kSG8u5/mm5fWKzwQLTn5UBsVMyIzS0z
zLMpwP7QhSMl/lgdIwkvTX8Gpqe7Cqs2nhHsuYFtrEqOzBTqdVHDUhy8N95R7+S+
OMsbnyFq4TXP5zIGRj4dQUd0JudNDVnLcll04m6AF5UCAxFCkBuepU3hD27lC9NP
7wRoFM+3hwteQOdX4XHinHGn+imjlyHv9oBBE5/lO4JpO6TxQ45BSI9/4drQAvl7
iR+dNO1avKCGEXu2S7lygqfHU3MTMGu7tBg76hPn2X/IscaHZX157ROcTRZkeADx
My3ypS+qXA0CAwEAAaN1MHMwDgYDVR0PAQH/BAQDAgWgMBMGA1UdJQQMMAoGCCsG
AQUFBwMCMAwGA1UdEwEB/wQCMAAwHQYDVR0OBBYEFFa0NhueHsUoCJ21+a85slq8
hqRKMB8GA1UdIwQYMBaAFEX9Gvx1vPLPFgfhTZLV3eQYygkQMA0GCSqGSIb3DQEB
CwUAA4IBAQCYcBneR2+rIbfNPtPawixkgKIbqdaEV6xpFQAD0cF6ldbFsZwN3Yzx
XDOYx9K0c5FHR6smoNNpcoOU6hjSP+v3Bj06gY6qdgSjkHhTYDBV84w83jMy3FoR
FhjSCfE+dXJeLBYJWPrbEkc2VI8bWk8G7KM1x7c8cHsznW4ddqoZJDfx/tsUkJO8
y55C9gE8XM0/OTj5r1MCxEogM6Doq/RvF18ZV/MnaJA0B/KakD+lv+k0tSKM+tW9
waDaqY8JZP6n8jaUhQlGkwaufkgiaoy1U0b9CwkR+E/RQhUi1ahdgOiMN/HxVtI8
hkh4mSojjVuMKJQs7oWEgW+kbTmhCoT9
```

-----END CERTIFICATE-----

pizza-client.key

-----BEGIN RSA PRIVATE KEY-----

```
MIIEpQIBAAKCAQEAyLu6nka4aimQldpddC5C2dyxf9IlZU4TI3Vlck4uSQvWRIby
7n+abl9YrPBAtOflQGxUzIjNLTPMsynA/tCFIyX+WB0jCS9Nfwamp7sKqzaeEey5
gW2sSo7MFOp1UcNSHLw33lHv5L44yxufIWrhNc/nMgZGPh1BR3Qm500NWctyWXTi
boAXlQIDEUKQG56lTeEPbuUL00/vBGgUz7eHC15A51fhceKccaf6KaOXIe/2gEET
n+U7gmk7pPFDjkFIj3/h2tAC+XuJH5007Vq8oIYRe7ZLuXKCp8dTcxMwa7u0GDvq
E+fZf8ixxodlfXntE5xNFmR4APEzLfKlL6pcDQIDAQABAoIBADJifKsx0SREnpge
oYqB+iG5NYyB8QUGneMumnIgkZmgMP4uaVfYC6lcoWN3Qqal9nM/PeHBDM8ly2HF
Pz42lNSHutnfJmYty2PxBW/gkQL8yJxzMPT91Fs6kJtHZn9JaZjw3Y0eP/rIjHTe
0AiRTUo2jy+NR6Bbs4D99K3mN02sQ40EDYHP/hzuOe67Je34K7rYVxjfQRrW0iBG
ujz/vI6VvZAiPAb3r+7vx1IYnSz8+Br1hinQPvFo0q4GvdWE/aTt0VvoXt4ag6MP
zMxa+n767MHbdiIewrxOc8ok/JXl+4E8iKQA5vsLz2WnuUcqQbKvl/UAUPzcWD97
Uk0WUsECgYEA3y8mZz0CsHSOpp7+xc4a3Zn0FDX1BBg4HwH9WwqctojAqyNGDenB
0eVKTz5ICjo3GgUvsGRL2ZyEpGiFl0lAEa/XxlsLoR1IEG8BXVCe5vYNJf1UcKvk
1HYO0jmHBBLRC0obc2QnWiKH4qWhkZYd5WPwv06+QPKRdmJ2ktB3GzkCgYEA5j+C
```

```
M8OwopdvTS4trnXB4N3pZyTLZZnqUw7idD3mWGUyu13MGsnWK+V667kHtkamYyh4
DN9O66k0HN82HBCr4jdH5ZB8OSxDYVJnPmoY8j0r8+qRagvD8s6x363BOraP0N/t
80UgSKXivlxUDn9hHvvfzVO8p/U8VN9ZTxbzQ3UCgYEA1rKf6e6KGL93vxfylMzJ
kWYZpBun4VF/I20hkaQqz3nMhpV/PcEif81oZ8TNPnF0MmbM0o4ZXXSxMQuPf9Kq
fJlBJILPNCVb/tsaX+8/fYUzbtk9ksn5bt1HMrq+hI+pY4fd0mqFZLMVL1YQkGHt
zo8OuKqCYS43+r+Lu34pJhkCgYEAsTyR7WblwULw2miBoZnj9ETeNheMulkQl38N
so0zkzfTJYe+ZaCYzJGJXYrA9P7NdebhAoejKKTHCLKJ5HCw3yHOplfiR+Bavb/A
2bqyGUYdX1VhOlB0NaQGKDiIVbnSWucFNA4yG7oWIJLR4ZcMG92nGVcsEd4k4vBK
vybbe0ECgYEAotGxz7CvjwHP14rfQeK0jzmQk+ettccFsbTl0smZHgB4pY1wAIeV
a84vrGyzt4Id1V8ucQPWiLSasXLhgKIedoWmGdRNidIPQsa8Q8oH0OJwAZgQ/Pnk
6UVP9oaoE/clzjlIz0hDLVEsxBnTRxV4uNumrnW3ZLTDy3aJ07fKZ9Y=
```

-----END RSA PRIVATE KEY-----

2. Retrieve a CA certificate from the certification path of https://auth.pizza/.
    - Use a web browser to visit the page and view its certificate. Web browsers vary, but you can usually click on the secure lock icon next to the URL.
    - Find the "Certification Path" for the certificate and select the top CA of the path. In the sample code below, we trust any certificate with that CA at the top of its certification path.
    - Save the CA Certificate to your computer as a Base64 encoded X.509 file called **pizza-server-ca.pem**.

pizza-server-ca.pem

-----BEGIN CERTIFICATE-----

```
MIIENjCCAx6gAwIBAgIBATANBgkqhkiG9w0BAQUFADBvMQswCQYDVQQGEwJTRTEU
MBIGA1UEChMLQWRkVHJ1c3QgQUIxJjAkBgNVBAsTHUFkZFRydXN0IEV4dGVybmFs
IFRUUCBOZXR3b3JrMSIwIAYDVQQDExlBZGRUcnVzdCBFeHRlcm5hbCBDQSBSb290
MB4XDTAwMDUzMDEwNDgzOFoXDTIwMDUzMDEwNDgzOFowbzELMAkGA1UEBhMCU0Ux
FDASBgNVBAoTC0FkZFRydXN0IEFCMSYwJAYDVQQLEx1BZGRUcnVzdCBFeHRlcm5h
bCBUVFAgTmV0d29yazEiMCAGA1UEAxMZQWRkVHJ1c3QgRXh0ZXJuYWwgQ0EgUm9v
dDCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALf3GjPm8gAELTngTlvt
H7xsD821+iO2zt6bETOXpClMfZOfvUq8k+0DGuOPz+VtUFrWlymUWoCwSXrbLpX9
uMq/NzgtHj6RQa1wVsfwTz/oMp50ysiQVOnGXw94nZpAPA6sYapeFI+eh6FqUNzX
mk6vBbOmcZSccbNQYArHE504B4YCqOmoaSYYkKtMsE8jqzpPhNjfzp/haW+710LX
a0Tkx63ubUFfclpxCDezeWWkWaCUN/cALw3CknLa0Dhy2xSoRcRdKn23tNbE7qzN
E0S3ySvdQwAl+mG5aWpYIxG3pzOPVnVZ9c0p10a3CitlttNCbxWyuHv77+ldU9U0
WicCAwEAAaOB3DCB2TAdBgNVHQ4EFgQUrb2YejS0Jvf6xCZU7wO94CTLVBowCwYD
VR0PBAQDAgEGMA8GA1UdEwEB/wQFMAMBAf8wgZkGA1UdIwSBkTCBjoAUrb2YejS0
Jvf6xCZU7wO94CTLVBqhc6RxMG8xCzAJBgNVBAYTAlNFMRQwEgYDVQQKEwtBZGRU
cnVzdCBBQjEmMCQGA1UECxMdQWRkVHJ1c3QgRXh0ZXJuYWwgVFRQIE5ldHdvcmsx
IjAgBgNVBAMTGUFkZFRydXN0IEV4dGVybmFsIENBIFJvb3SCAQEwDQYJKoZIhvcN
AQEFBQADggEBALCb4IUlwtYj4g+WBpKdQZic2YR5gdkeWxQHIzZlj7DYd7usQWxH
YINRsPkyPef89iYTx4AWpb9a/IfPeHmJIZriTAcKhjW88t5RxNKWt9x+Tu5w/Rw5
6wwCURQtjr0W4MHfRnXnJK3s9EK0hZNwEGe6nQY1ShjTK3rMUUKhemPR5ruhxSvC
Nr4TDea9Y355e6cJDUCrat2PisP29owaQgVR1EX1n6diIWgVIEM8med8vSTYqZEX
c4g/VhsxOBi0cQ+azcgOno4uG+GMmIPLHzHxREzGBHNJdmAPx/i9F4BrLunMTA5a
mnkPIAou1Z5jJh5VkpTYghdae9C8x49OhgQ=
```

-----END CERTIFICATE-----

3. Upload the three files to the **cert/** directory using XCTU or **FS** commands and YMODEM in a terminal emulator. Since the **.pem** files contain public information, you can upload them with the **FS PUT** command. In this example the key file is not much of a secret, but in a typical installation you would upload the key file with the **FS XPUT** command to limit access to its

contents.

4. Now you can use the following code snippet to test the connection to **auth.pizza**. Run it on a computer with Python3 installed to verify that the files are set up correctly, and then try it in MicroPython on an XBee Cellular Modem. If you try it without the **keyfile**, **certfile** and **ca_certs** parameters to **wrap_socket()**, it will still connect but will get a different JSON response.

Note Saving the certificate with a **.cer** extension is acceptable.

You can copy and paste the following samples into the terminal.

pizza.py

```
# Test code for CloudFlare's https://auth.pizza page
# https://blog.cloudflare.com/introducing-tls-client-auth/

import sys
if sys.platform == 'xbee-cellular':
    import usocket, ussl
    proto = usocket.IPPROTO_SEC
else:
    import socket as usocket
    import ssl as ussl
    proto = usocket.IPPROTO_TCP
print('Creating socket...')
s = usocket.socket(usocket.AF_INET, usocket.SOCK_STREAM, proto)
w = ussl.wrap_socket(s,
    keyfile='cert/pizza-client.key',
    certfile='cert/pizza-client.pem',
    ca_certs='cert/pizza-server-ca.pem')
print('Opening connection...')
w.connect(('auth.pizza', 443))
w.write(b'GET / HTTP/1.0\r\nHost: auth.pizza\r\nAccept:
application/json\r\n\r\n')
print(str(w.read(4096), 'utf-8'))
w.close()
```

5. Make use of **urequests.py** for an easier interface to make web page requests and parse the responses.

auth_pizza-ureqests.py

```
# Test code for CloudFlare's https://auth.pizza page
# https://blog.cloudflare.com/introducing-tls-client-auth/

import sys

if sys.platform == 'xbee-cellular':
    import urequests
else:
    import requests as urequests

print('Sending request...')
r = urequests.get('https://auth.pizza',
                  headers={'Accept': 'application/json'},
                  verify='cert/pizza-ca.pem',
                  cert=('cert/pizza-client.pem', 'cert/pizza-
client.key'))
```

```
print(r.text)
```

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

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.

```
{
  "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:

arn:aws:iot:your-region:your-aws-account:client/<my-client-id>

Note Replace the region and account numbers in the following sample code with your own information.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "*",
      "Condition": {
        "Bool": {
          "iot:Connection.Thing.IsAttached": [
            "true"
          ]
        },
        "StringEquals": {
          "iot:ClientId": "${iot:Connection.Thing.ThingName}"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Receive"
      ],
      "Resource": [
        "arn:aws:iot:us-east-
1:123456789012:topic/${iot:Connection.Thing.ThingTypeName}/${iot:Connec
tion.Thing.ThingName}",
        "arn:aws:iot:us-east-
1:123456789012:topic/${iot:Connection.Thing.ThingTypeName}/${iot:Connec
tion.Thing.ThingName}/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Subscribe"
      ],
      "Resource": [
        "arn:aws:iot:us-east-
1:123456789012:topicfilter/${iot:Connection.Thing.ThingTypeName}/${iot:
Connection.Thing.ThingName}",
        "arn:aws:iot:us-east-
1:123456789012:topicfilter/${iot:Connection.Thing.ThingTypeName}/${iot:
Connection.Thing.ThingName}/*"
      ]
    }
  ]
}
```

Create a Thing

From the **AWS services** page, choose **IoT Core**.

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 XBee3 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[1] 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.

---

[1]ATS 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**.

dInstall 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.

Original name	New name
9770fec281-certificate.pem.crt	aws.crt
9770fec281-private.pem.key	aws.key
9770fec281-public.pem.key	(*unused*)
SFSRootCAG2.pem	aws.ca

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/ 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.

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**):

```
# 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\r\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 you 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 rexample, 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
thing_type = b'XBee_Cellular'
thing_name = b'IMEI_63890'

import usocket, ussl

s = usocket.socket(usocket.AF_INET, usocket.SOCK_STREAM, usocket.IPPROTO_
SEC)
s.setblocking(False)
w = ussl.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\r\n\r\n' % (thing_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 **w.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

connection: keep-alive


{"state":{},"metadata":{},"version":1,"timestamp":1530818883}
```

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.

```
"""
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
 - 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"}  # 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=sslp)
    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 recieves
   "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"}  # ssl certs

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 the XBee Cellular Modem. See Which features apply to my device? for a list of the supported features.

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

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.

```
MicroPython v1.9.3-999-g00000000 on 2018-01-01; XBee Module with EFX32
>>> import time
>>> time.localtime()
(2017, 1, 13, 14, 51, 18, 4, 13)
```

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:

```
import time
print("\nPreparing 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 **Crtl+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)
```

```
Pause 5 seconds
(2017, 5, 10, 11, 31, 5, 2, 130)
Pause 5 seconds
(2017, 5, 10, 11, 31, 10, 2, 130)
Pause 5 seconds
(2017, 5, 10, 11, 31, 15, 2, 130)
Pause 5 seconds
Done!
```

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.

```
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

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.

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

```
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 **Crtl+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:

```
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 **Crtl+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

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 **Crtl+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\n"
# 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)

print("Printing first 3 lines of server's response: \r\n")
# 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.
- **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 **Crtl+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 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.
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 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 **Crtl+E** to enter paste mode.
4. At the MicroPython **>>>** prompt, right-click and select the **Paste** option.

```
import socket
# Return tuple (family, type, proto, canonname, sockaddr)
print("\nCalling getaddrinfo() for micropython.org on port 80,")
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("\nFinally, 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:
[family, type, proto, canonname, sockaddr]

[(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.

Value	Description
2	<*family*> An integer that represents the type of connection the socket is using. Represents the **usocket.AF_INET**, meaning an internet family of connection.
1	<*type*> An integer that represents the type of connection the socket is using. Represents **usocket.SOCK_STREAM**, meaning a TCP connection.
0	<*protocol*> An integer that represents the type of connection the socket is using. Represents **usocket.IPPROTO_IP**, meaning the IP protocol.
empty string	<*canonname*> A string that represents the "canonical" name of the host, if it has one. If the host does not have a "canonical" name, an empty string is used.
176.58.119.26, 80	<*sockaddr*> The IP address and port number of the machine you queried.

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 of these options:

```
            # 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("\nSetting 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.\n")
# 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!\n")
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 caught
# 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.\n")
# Call "recv".
try:
    socketObject.recv(1024)
except OSError:
    print("No data to read!\n")
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.\n")
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)
# The call will not return until the server sends data (which won't happen
in
# this case, since none was requested), or the socket times out.
```

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.

```
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\r\nHost: %s\r\n\r\n' % (path, host),
'utf8')
        print("Requesting /%s from host %s\n" % (path, host))
        s.send(request)
        while True:
            print(str(s.recv(500), 'utf8'), end = '')
    finally:
        s.close()
```

3. At the MicroPython **>>>** prompt, press **Crtl+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.

```
>>> http_get('http://www.micropython.org/ks/test.html')
```

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

The following sections include code samples for changing the XBee Cellular Modem pins.

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

Change I/O pins

You can use MicroPython to change the pins on the XBee Cellular Modem .

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:

```
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 Cellular Modem.

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.

```
>>> 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)
```

Change output pin values: turn LEDs on and off

You can change the output value of a pin on the XBee Cellular Modem 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 **Crtl+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.

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 Cellular Modem. 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.

```
from machine import Pin
ad0 = Pin("D0", Pin.IN)
```

3. At the MicroPython **>>>** prompt, press **Crtl+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.

```
ad0.value()
```

6. Press **Crtl+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.

```
while True:
    if ad0.value() == 0:
        print("SW2 has been pressed!")
        break
```

9. Press **Crtl+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:

```
# Import the Pin module from the machine module
from machine import Pin

print("\nChecking the mode of pin 20 (AD0/DIO0)...")
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 pin 15 (Associate/DIO5)...")
```

```
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)
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" % 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+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 **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 Cellular Modem 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.

```
# 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" % 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()
print("Pull direction of AD0/DIO0: %d\n" % pinpull)
# 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" % 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()
print("Pull direction of AD0/DIO0: %d" % pinpull)
print("This should return 1, since it was just set to PULL_UP.\n")
print("Checking the value on the pin again...")
pinval = d0.value()
print("Value on AD0/DIO0: %d" % 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")

# 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 AD0/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 XBee Cellular Modem 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.

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:

```
# Import ADC from machine, for simpler syntax.
from machine import ADC

# Create an ADC object for pin AD0.
print("\nCreating an ADC object for pin AD0...")
adc0 = ADC("D0")
# Perform a read of the analog voltage value present at the pin.
print("Reading the ADC value on the pin...")
adc_value = adc0.read()
print("ADC read #1 value: %d\n" % 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.\n")
print("Reading ADC0 again...")
adc_value = adc0.read()
print("ADC read #2 value: %d" % adc_value)
# This should return a low value, around 0.
print("Note that this value is low, it should be 0 or close to 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

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.

```
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
message")
```

```
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"
```

Send to an invalid phone number

The code in this example sends a message to an invalid phone number. An invalid phone number error message is returned.

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('1', 'test')** and press **Enter**.

```
>>> c.sms_send('1', 'test')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid SMS phone number
```

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:\n%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
```

AT command 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 **atcmd()** method first appeared in the **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 **xbee** module, for example, **xbee.atcmd()**. The **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.

Note For the XBee Cellular Modem, the **xbee().atcmd()** method does not support the following AT commands: **AS**, **FS**, **LA** and **IS**.
For the XBee3 Zigbee RF Module, the **xbee.atcmd()** function does not support the following AT commands: **IS**, **ED**, **AS**, **ND** and **DN**. To perform a network discovery equivalent to an **ND** command, use the **xbee.discover()** function.

The following sections include MircroPython AT command code samples you can use with the XBee device.

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:

```
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
print("The XBee Cellular is %.1F" % (x.atcmd('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 XBee3 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')))

# Set the Network Identifier of the radio
xbee.atcmd("NI", "XBee3 module")

# Configure a destination address using two different data types
xbee.atcmd("DH", 0x0013A200)          # Hex
xbee.atcmd("DL", b'\x12\x25\x89\xF5') # Bytes

dest = xbee.atcmd("DH") + xbee.atcmd("DL")
formatted_dest = ':'.join('%02x' % b for b in dest)
print("Destination address set to: " + formatted_dest)

# 'TP' records the current temperature measure on the module
print("The XBee is %.1F degrees" % (xbee.atcmd('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:

```
64-bit address: 1754658623
16-bit address: 65534
Destination address set to: 00:13:a2:00:12:25:89:f5
The XBee is 78.8 degrees
```

Print a list of AT commands

You can read and show output for multiple AT commands and I/O parameter values.

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 appropriate sample code shown below. For XBee Cellular Modem:

```
import xbee
x = xbee.XBee()

def dump_atcmds():  # This function outputs multiple AT parameter values.
    print("Here is a summary of all AT values:")
    print()
    for cmd in ['PH', 'S#', 'IM', 'MN', 'MV', 'DB', 'AM', 'IP', 'TL', 'TM',
        'DO', 'DL', 'DE', 'MY', 'BD', 'NB', 'SB', 'RO', 'TD', 'FT', 'AP',
        'D8', 'TP', 'SM', 'SP', 'ST', 'CC', 'CT', 'GT', 'VL']:
        print(cmd, '=', x.atcmd(cmd))
    print("The following AT values are in HEX format:")
    for hexcmd in ['VR', 'HV', 'AI', 'DI', 'CI', 'HS', 'CK']:
        print(hexcmd, '=', hex(x.atcmd(hexcmd)))


def dump_iocmds():  # This function outputs multiple IO parameter values.
    print("Here is a summary of all IO values:")
```

```
        for cmd in ['D0', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D9',
            'P0', 'P1']:
            print(cmd, '=', x.atcmd(cmd))
        print("The following IO values are in HEX format:")
        for hexcmd in ['PR', 'PD']:
            print(hexcmd, '=', hex(x.atcmd(hexcmd)))


dump_atcmds()
print()
dump_iocmds()
```

For XBee3 Zigbee RF Module:

```
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",
              "DL", "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", "AO",
              "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": ["D0", "D1", "D2", "D3", "D4",
              "D5", "D6", "D7", "D8", "D9",
              "P0", "P1", "P2", "P3", "P4",
              "P5", "P6", "P7", "P8", "P9",
              "PR", "PD", "LT", "RP"],
       "12. I/O Sampling": ["IR", "IC", "V+"],
       "13. Diagnostics": ["VR", "VH", "HV", "%V", "TP", "CK"]
       }

print("Here is a summary of all AT values:\n")
for category, cmds in sorted(at_cmds.items()):
       print("\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]
```

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

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

- Machine module
- Cellular network configuration module
- XBee 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.

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

Function	Description
uio	This module contains additional types of stream (file-like) objects and helper functions.
ujson	This module performs JSON encoding and decoding.
usocket	(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.
ustruct	This module provides functions to pack and unpack primitive data types.
utime	XBee Cellular Modem: This module provides functions for getting the current time and date, measuring time intervals, and for delays. XBee3 Zigbee RF Module: This module provides functions for measuring time intervals, and for delays.

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.

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

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

```
machine.reset_cause()
```

Constants

These return values describe the cause of a reset.

```
machine.PWRON_RESET
```

```
machine.HARD_RESET
```

```
machine.WDT_RESET
```

```
machine.DEEPSLEEP_RESET
```

```
machine.SOFT_RESET
```

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 the XBee Cellular Modem. See Which features apply to my device? for a list of the supported features.

You use this function to enable PWM on XBee Cellular Modem 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.

```
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

pwm2 = PWM('P1', duty=512) # create and configure in one go
```

The following REPL session makes use of the PWM class:

```
>>> 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 PWM1 is not supported currently.

Class ADC: analog to digital conversion

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 class to read analog values on a pin.

```
import machine

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

Constructors

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

```
class machine.ADC('D0')
```

Note The ADC analog pin input range is 0 - 2.5 V.

Methods

Read the analog value

This function allows you to read the ADC value.

```
apin.read()
```

Class I2C: two-wire serial protocol

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

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.

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

Printing the I2C object gives you information about its configuration.

The XBee Cellular Modem 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 Cellular Modem 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 D1 for SCL and D11 for SDA. Using the **scl** and **sda** parameters to select pins is not valid on the XBee Cellular Modem.

An example of using I2C follows:

```
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 slave
i2c.writeto_mem(42, 2, b'\x10') # write 1 byte to memory of slave 42
                                #   starting at address 2 in the slave
```

Constructors

class machine.I2C(id, *, 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 those that respond. A device responds if it pulls the SDA line low after its address (including a write bit) is sent on the bus.

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.

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**.

Sample program

The following sample works with a DS1621 I2C Temperature Sensor. Make the following connections before testing the code:

XBee pin	Description	DS1621 pin
19 (DIO1)	SCL	2
7 (DIO11)	SDA	1

XBee pin	Description	DS1621 pin
1	VCC	8
10	GND	4

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.

```
# Simple DS1621 I2C Example
# Wiring Diagram:
# XBee -> DS1621
#  19  SCL  2
#   7  SDA  1 (and connect via pullup resistor to Vcc)
#   1  Vcc  8
#  10  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)

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 the XBee Cellular Modem. See Which features apply to my device? for a list of the supported features.

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

Class UART

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

MicroPython on the XBee Cellular Modem provides access to a 3-wire or 5-wire TTL-level serial port (referred to as **machine.UART(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.

XBee				FTDI TTL-232R	
Pin	Name	Description	Direction	Pin	Name
10	GND	Ground	N/A	1	GND
11	DIO4	Transmit (TX)	XBee →	5	RXD
4	DIO12	Receive (RX)	XBee ←	4	TXD
18	DIO2	Ready to Receive (RTS)	XBee →	2	CTS#
17	DIO3	Clear to Send (CTS)	XBee ←	6	RTS#

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).

```
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:

```
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.

```
uart.read(10)        # read 10 characters, returns a bytes object
uart.read()          # read all available characters
uart.readline()      # read a line
```

```
uart.readinto(buf)    # read and store into the given buffer
uart.write('abc')     # write the 3 characters
```

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

```
uart.any()            # returns the number of characters waiting
```

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.

`UART.RTS`

`UART.CTS`

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

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:

```
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:

```
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.

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.

XBee module

The functions in this section are specific to the XBee device hardware.

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.

```
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.

```
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, btyearray, 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 AT command examples.

XBee MicroPython module on the XBee3 Zigbee RF Module

Note This section only applies to the XBee3 Zigbee RF Module. See Which features apply to my device? for a list of the supported features.

Functions

The **xbee** MicroPython module supports the following functions:

atcmd()

Use this function to set or query an AT command on the XBee3 Zigbee RF Module.

```
xbee.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, btyearray, 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 AT command examples.

discover()

Use this function to perform a network discovery, which is equivalent to issuing the **ND** command.

```
xbee.discover()
```

This function accepts no parameters, and returns a dictionary for each discovered node that contains the following entries:

- **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

Example output:

```
{
  'rssi': -20,
  'node_id': ' ',
  'device_type': 1179648,
  'parent_nwk': 65534,
  'sender_nwk': 41334,
```

```
  'sender_eui64': b'\x00\x13\xa2\x00\x92w%`',
  'node_type': 1
}
```

receive()

The XBee3 Zigbee RF Module 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; the sending node will receive a transmission status of **0x24** (Address not found) in this case.

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.

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 a dictionary containing the following entries:

- **sender_nwk**: the 16-bit network address of the sending node
- **sender_eui64**: the 64-bit address (as a bytearray) of the sending node
- **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:

```
{
  'cluster': 17,
  'dest_ep': 232,
  'broadcast': False,
  'source_ep': 232,
  'payload': b'Sample payload',
  'profile': 49413,
  'sender_nwk': 63941,
  'sender_eui64': b'\x00\x13\xa2\x00\x92w%`'
}
```

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(ENOTCONN)**: 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)
```

<dest>

The **<dest>** parameter is the destination address of the message, and accepts any of the following:

- an integer for 16-bit addressing
- 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

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).