Digi XBee® Cellular LTE Cat 1

Embedded Modem

User Guide

Revision history—90001525

Revision	Date	Description
N	February 2019	Added information for XBIB-C-TH and XBIB-C-GPS development boards
P	April 2019	Added information for Digi Remote Manager®
R	June 2019	Added information for Clean shutdown and securing the connection between and XBee and Remote Manager. Added FCC publication 996369 related information. Updated the FCC and IC labeling information.
S	July 2019	Updated the FCC and IC labeling information.
T	July 2019	Added socket information. Additional edits.
	September	Added firmware update information.

Trademarks and copyright

Disclaimers

Warranty

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

www.digi.com/howtobuy/terms

Send comments

Documentation feedback: To provide feedback on this document, send your comments to techcomm@digi.com.

Customer support

Digi Technical Support: Digi offers multiple technical support plans and service packages to help our customers get the most out of their Digi product. For information on Technical Support plans and pricing, contact us at +1 952.912.3444 or visit us at www.digi.com/support.

Contents

Socket behavior

Extended Socket frames

Transport Layer Security (TLS)

AT commands

Regulatory information

Digi XBee Cellular LTE Cat 1 Embedded Modem User Guide

The XBee Cellular Modem is an embedded Long-Term Evolution (LTE) Category 1 cellular module that provides original equipment manufacturers (OEMs) with a simple way to integrate cellular connectivity into their devices.

The XBee Cellular Modem enables OEMs to quickly integrate cutting edge 4G cellular technology into their devices and applications without dealing with the painful, time-consuming, and expensive FCC and carrier end-device certifications.

With the full suite of standard XBee API frames and AT commands, existing XBee customers can seamlessly transition to this new device with only minor software adjustments. When OEMs add the XBee Cellular Modem to their product, they create a future-proof design with flexibility to switch between wireless protocols or frequencies as needed.

You can read some frequently asked questions here.

Applicable firmware and hardware

This manual supports the following firmware:

- 100A

It supports the following hardware:

- XBC-V1-UT-xxx

SIM cards

If you order the wrong type of SIM card it will not work with the XBee Cellular Modem.

Verizon recommends SIM SKU: **M2MTRI-NONRUG-GT-A** or an equivalent that must include a 4FF punch out. This SKU is in triple punch, so devices with 2FF/3FF or 4FF can use this SIM SKU.

Bulk SIMs for M2M/IoT are available from:

National distributor	Network	Contact	Phone number	Email
Reliance Communications	Verizon direct	Raja Ali	917-517-7282	raja.ali@reliance.us
Ingram Micro - Sales	Verizon direct	Lesli Reeves	317-707-2371	lesli.reeves@ingrammicro.com

National distributor	Network	Contact	Phone number	Email
Ingram Micro - Sales	Verizon direct	Steve Kreiger	317-707-2474	steve.kreiger@ingrammicro.com
Ice Mobility	Verizon direct	Tom Puchala	847-876-1768	tom.puchala@icemobility.com
KORE	Verizon MVNO	Genesis Crowder	877-710-5673	gcrowder@korewireless.com
KORE	Verizon MVNO	Mike Basso	877-710-5673	mbasso@korewireless.com

Get started with the XBee Cellular Modem Development Kit

This section describes how to connect the hardware in the XBee Cellular Modem Development Kit, and provides some examples you can use to communicate with the device.

You should perform all of the steps below in the order shown.

1. Identify the kit contents
2. Connect the hardware
3. Review the development board
4. Set up cellular service
5. Update the firmware on your XBee
6. Use one of the following methods to verify your cellular connection:
   - Connect to the Echo server
   - Connect to the ELIZA server
   - Connect to the Daytime server

Optional steps

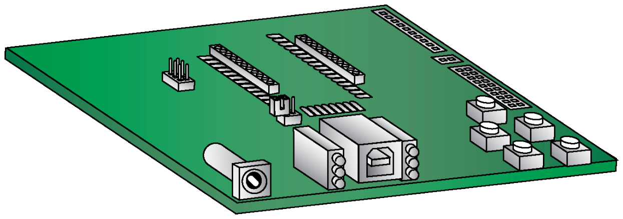
You can review the information in these steps for more XBee connection examples and examples of how to use MicroPython.

1. Review additional connection examples to help you learn how to use the device. See XBee connection examples.
2. Review introductory MicroPython examples. You can use MicroPython to enhance the intelligence of the XBee to enable you to do edge-computing by adding business logic in MicroPython, rather than using external components.
   - Example: hello world
   - Example: turn on an LED

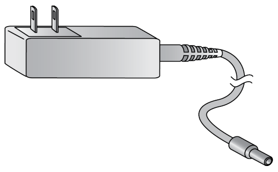
Identify the kit contents

The Developer's kit includes the following:

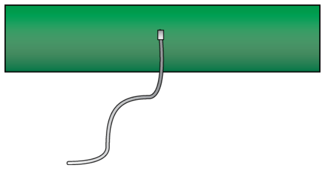
One XBIB-U-DEV board



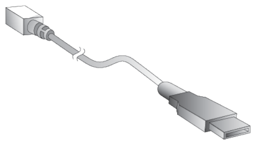
One 12 V power supply



Two cellular antennas with U.FL connectors

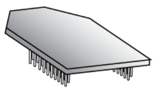


One USB cable



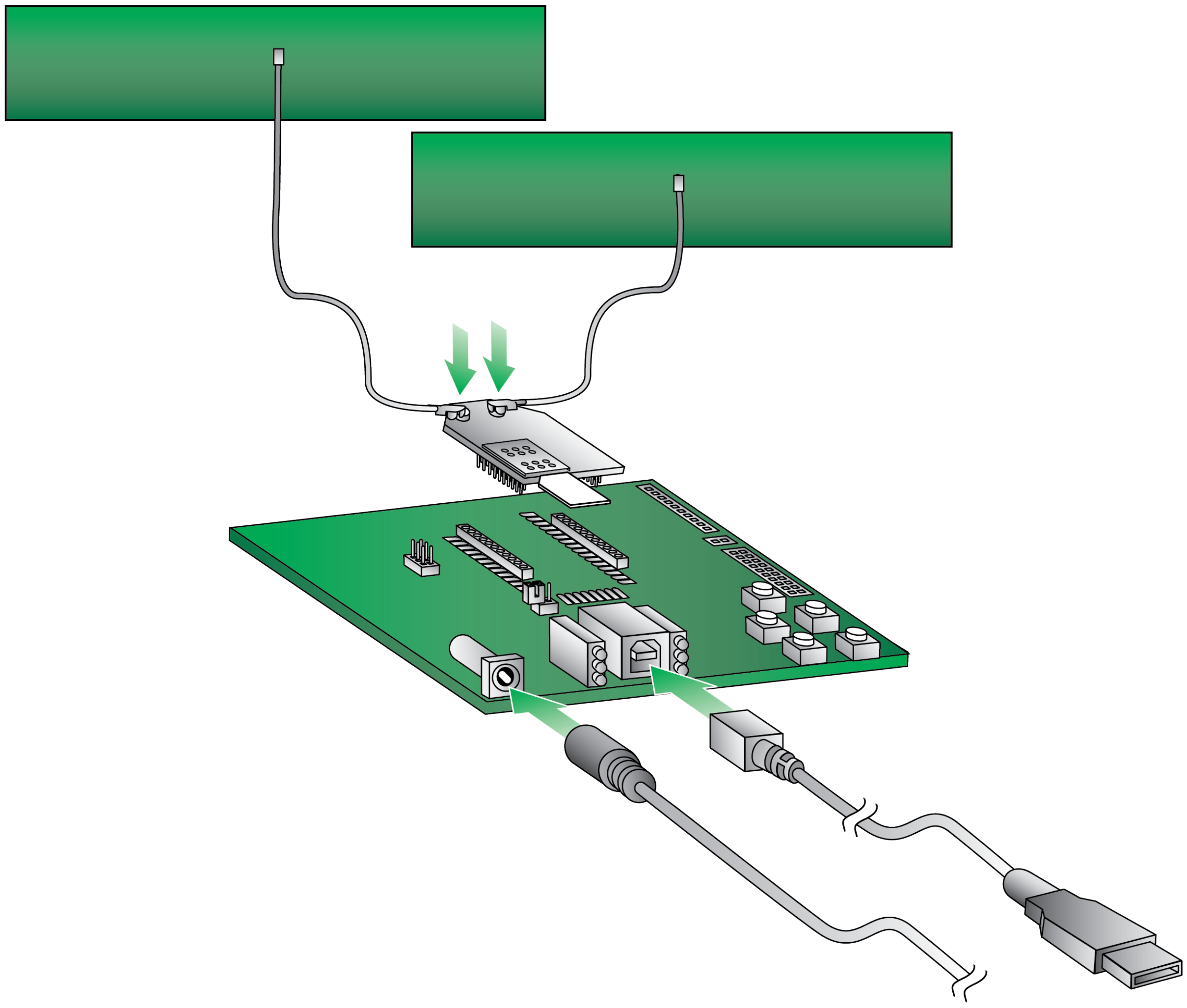
One XBee Cellular Modem

Note The XBee Cellular Modem comes attached to the board in ESD wrap.



One SIM card

Connect the hardware



1. The XBee Cellular Modem should already be plugged into the XBIB-U-DEV board.
2. The SIM card should be already be inserted into the XBee Cellular Modem. If not, install the SIM card into the XBee Cellular Modem.

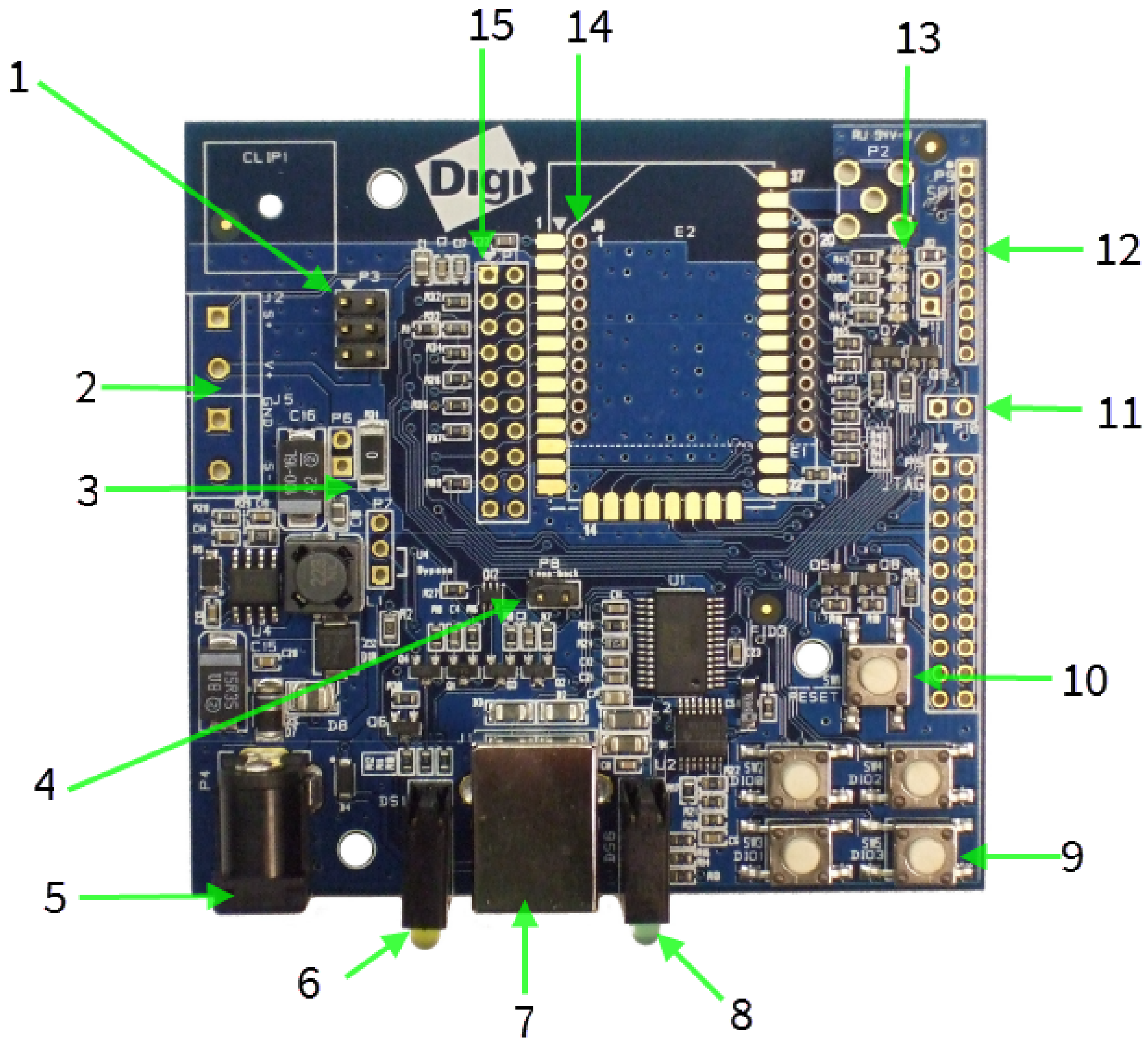
WARNING! Never insert or remove the SIM card while the device is powered!

3. Connect the antennas to the XBee Cellular Modem. Align the U.FL connectors carefully, then firmly press straight down to seat the connector. You should hear a snap when the antenna attaches correctly. U.FL is fragile and is not designed for multiple insertions, so exercise caution when connecting or removing the antennas. We recommend using a U.FL removal tool.

4. Plug the 12 V power supply to the power jack on the development board.
5. Connect the USB cable from a PC to the USB port on the development board. The computer searches for a driver, which can take a few minutes to install.

XBIB-U-DEV reference

This picture shows the XBee USB development board and the table that follows explains the callouts in the picture.



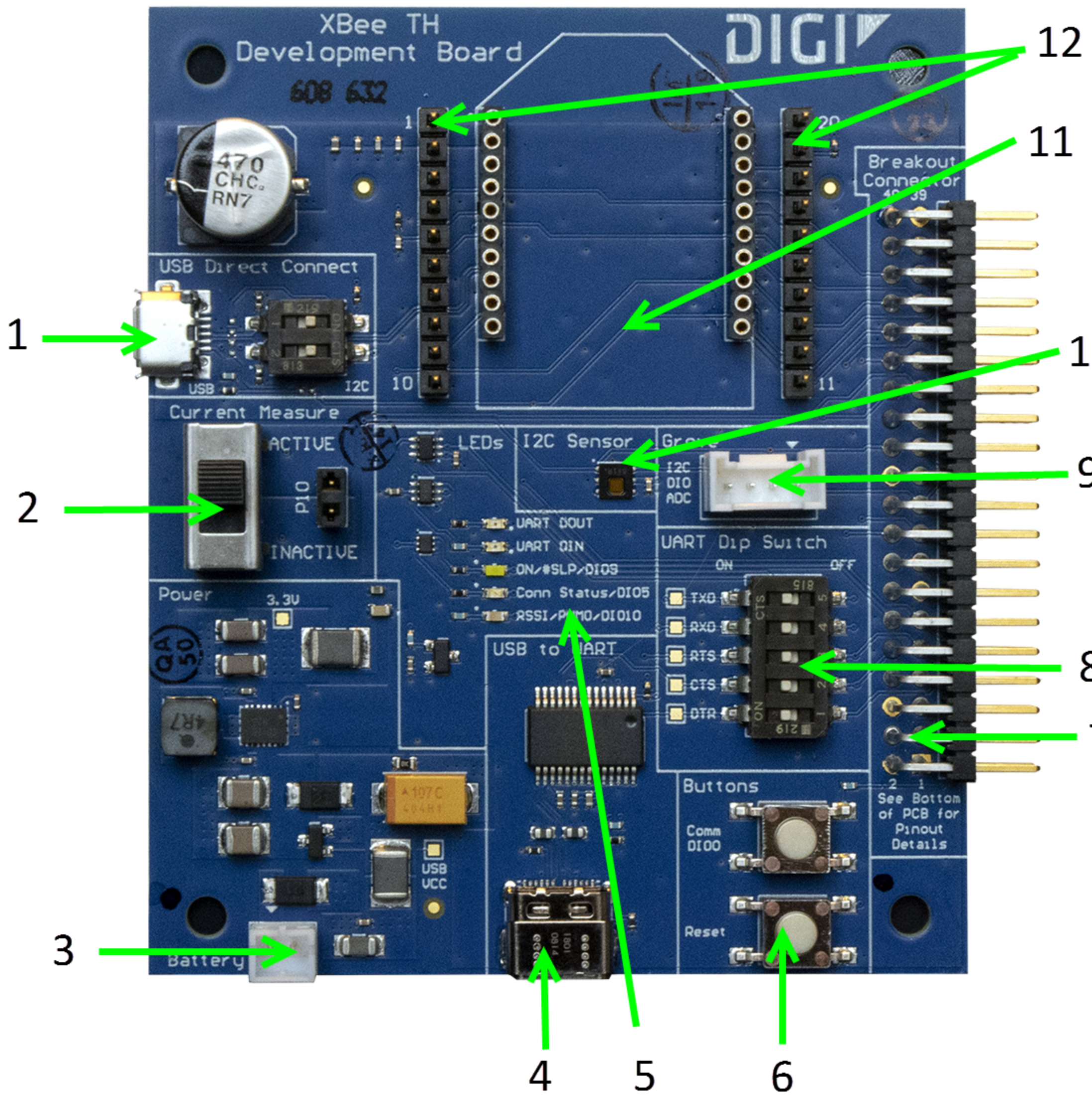
Number	Item	Description
1	Programming header	Header used to program XBee programmable devices.
2	Self power module	Advanced users only—voids the warranty. Depopulate R31 to power the device using V+ and GND from J2 and J5. You can connect sense lines to S+ and S- for sensing power supplies. ⚠ CAUTION: Voltage is not regulated. Applying the incorrect voltage can cause fire and serious injury.[1]
3	Current testing	Depopulating R31 allows a current probe to be inserted across P6 terminals. The current though P6/R31 powers the device only. Other supporting circuitry is powered by a different trace.
4	Loopback jumper	Populating P8 with a loopback jumper causes serial transmissions both from the device and from the USB to loopback.
5	DC barrel plug: 6-20 V	Greater than 500 mA loads require a DC supply for correct operation. Plug in the external power supply prior to the USB connector to ensure that proper USB communications are not interrupted.
6	LED indicator	Yellow: Modem sending serial/UART data to host. Green: Modem receiving serial/UART data from host. Red: Associate.
7	USB	Connects to your computer.
8	RSSI indicator	See RSSI PWM. On the XBIB-U, more lights are better.
9	User buttons	Connected to DIO lines for user implementation.
10	Reset button	Press the reset button to reset the device to the default configuration.
11	SPI power	Connect to the power board from 3.3 V.
12	SPI	Only used for surface-mount devices.
13	Indicator LEDs	DS5: ON/$\overline{\text{SLEEP}}$ DS2: DIO12, the LED illuminates when driven low. DS3: DIO11, the LED illuminates when driven low. DS4: DIO4, the LED illuminates when driven low.
14	Through-hole XBee sockets	
15	20-pin header	Maps to standard through-hole XBee pins. Male, Samtec header, part number: TSW-110-26-L-D. 2.54 mm / .100" pitch and row spacing.

[1]Powering the board with J2 and J5 without R31 removed can cause shorts if the USB or barrel plug power are connected. Applying too high a voltage destroys electronic circuitry in the device and other board components and/or can cause injury.

XBIB-CU-TH reference

This picture shows the XBee-CU-TH development board and the table that follows explains the callouts in the picture.

Note This module is sold separately.



Number	Item	Description
1	Secondary USB (USB MICRO B) and DIP Switch	Secondary USB Connector for direct programming of modules on some XBee units. Flip the Dip switches to the right for I2C access to the board; flip Dip switches to the left to disable I2C access to the board. The USB_P and USB_N lines are always connected to the XBee, regardless of Dip switch setting. This USB port is not designed to power the module or the board. Do not plug in a USB cable here unless the board is already being powered through the main USB-C connector. Do not attach a USB cable here if the Dip switches are pushed to the right. **WARNING!** Direct input of USB lines into XBee units or I2C lines not designed to handle 5V can result in the destruction of the XBee or I2C components. Could cause fire or serious injury. Do not plug in a USB cable here if the XBee device is not designed for it and do not plug in a USB cable here if the Dip switches are pushed to the right.
2	Current Measure	Large switch controls whether current measure mode is active or inactive. When inactive, current can freely flow to the VCC pin of the XBee. When active, the VCC pin of the XBee is disconnected from the 3.3 V line on the development board. This allows current measurement to be conducted by attaching a current meter across the jumper P10.
3	Battery Connector	If desired, a battery can be attached to provide power to the development board. The voltage can range from 2 V to 5 V. The positive terminal is on the left. If the USB-C connector is connected to a computer, the power will be provided through the USB-C connector and not the battery connector. **Note** See modem specifications for minimum voltage requirements for your modem.
4	USB-C Connector	Connects to your computer and provides the power for the development board. This is connected to a USB to UART conversion chip that has the five UART lines passed to the XBee. The UART Dip Switch can be used to disconnect these UART lines from the XBee. **Note** Requires USB 3.0 to supply required current.
5	LED indicator	Red: UART DOUT (modem sending serial/UART data to host) Green: UART DIN (modem receiving serial/UART data from host) White: ON/SLP/DIO9 Blue: Connection Status/DIO5 Yellow: RSSI/PWM0/DIO10
6	User Buttons	Comm DIO0 Button connects the Commissioning/DIO0 pin on the XBee Connector through to a 10 Ω resistor to GND when pressed. $\overline{\text{RESET}}$ Button Connects to the $\overline{\text{RESET}}$ pin on the XBee Connector to GND when pressed.

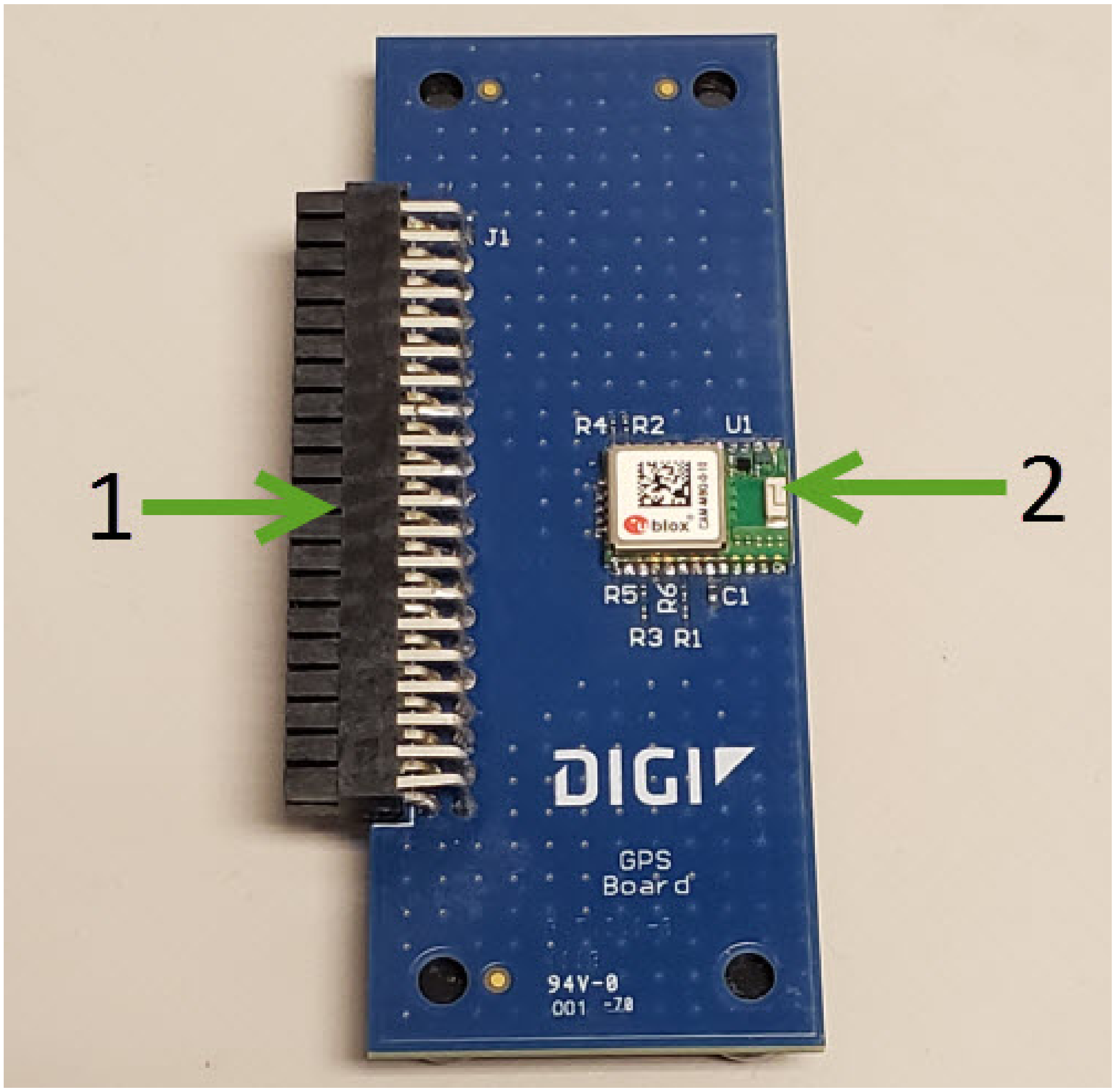
Number	Item	Description
7	Breakout Connector	This 40 pin connector can be used to connect to various XBee pins as shown on the silkscreen on the bottom of the board.
8	UART Dip Switch	This dip switch allows the user to disconnect any of the primary UART lines on the XBee from the USB to UART conversion chip. This allows for testing on the primary UART lines without the USB to UART conversion chip interfering. Push Dip switches to the right to disconnect the USB to UART conversion chip from the XBee.
9	Grove Connector	This connector can be used to attach I2C enabled devices to the development board. Note that I2C needs to be available on the XBee in the board for this functionality to be used. Pin 1: I2C_CLK/XBee DIO1 Pin2: I2C_SDA/XBee DIO11 Pin3: VCC Pin4: GND
10	Temp/Humidity Sensor	This as a Texas Instruments HDC1080 temperature and humidity sensor. This part is accessible through I2C. Be sure that the XBee that is inserted into the development board has I2C if access to this sensor is desired.
11	XBee Socket	This is the socket for the XBee (TH form factor).
12	XBee Test Point Pins	Allows easy access for probes for all 20 XBee TH pins. Pin 1 is shorted to Pin 1 on the XBee and so on.

XBIB-C-GPS reference

This picture shows the XBIB-C-GPS module and the table that follows explains the callouts in the picture.

Note This module is sold separately. You must also have purchased an XBIB-CU-TH development board.

Note For a demo of how to use MicroPython to parse some of the GPS NMEA sentences from the UART, print them and report them to Digi Remote Manager, see Run the MicroPython GPS demo.

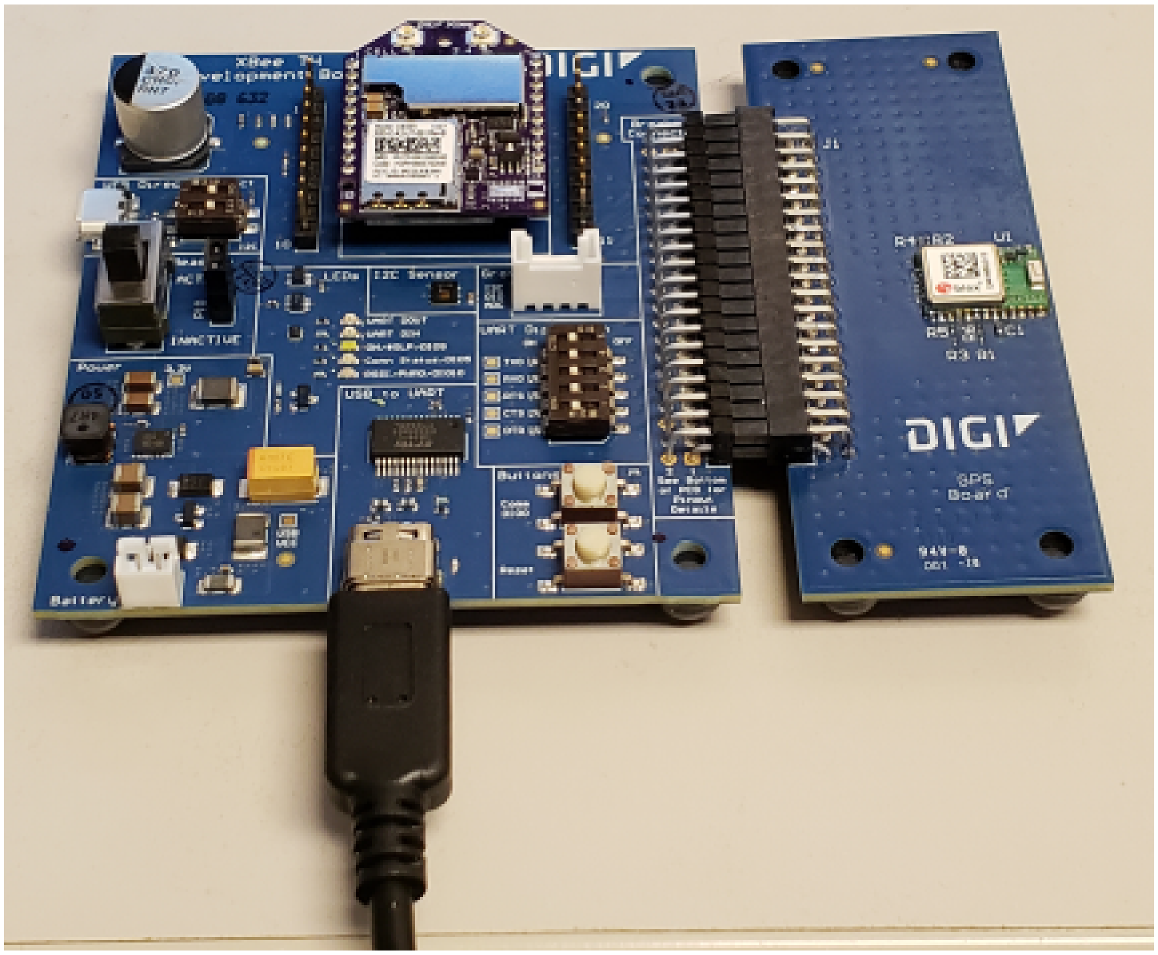


Number	Item	Description
1	40-pin header	This header is used to connect the XBIB-C-GPS board to a compatible XBIB development board. Insert the XBIB-C-GPS module slowly with alternating pressure on the upper and lower parts of the connector. If added or removed improperly, the pins on the attached board could bend out of shape.
2	GPS unit	This is the CAM-M8Q-0-10 module made by u-blox. This is what makes the GPS measurements. Proper orientation is with the board laying completely flat, with the module facing towards the sky.

Interface with the XBIB-C-GPS module

The XBee Cellular Modem can interface with the XBIB-C-GPS board through the large 40-pin header. This header is designed to fit into XBIB-C development board. This allows the XBee Cellular Modem in the XBIB-C board to communicate with the XBIB-C-GPS board—provided the XBee device used has MicroPython capabilities (see this link to determine which devices have MicroPython capabilities). There are two ways to interface with the XBIB-C-GPS board: through the host board's Secondary UART or through the I2C compliant lines.

The following picture shows a typical setup:



I^{2}C communication

There are two I2C lines connected to the host board through the 40-pin header, SCL and SDA. I2C communication is performed over an I2C-compliant Display Data Channel. The XBIB-C-GPS module operates in slave mode. The maximum frequency of the SCL line is 400 kHz. To access data through the I2C lines, the data must be queried by the connected XBee Cellular Modem.

For more information about I2C Operation see the **I2C** section of the *Digi Micro Python Programming Guide*.

For more information on the operation of the XBIB-C-GPS board see the CAM-M8 datasheet. Other CAM-M8 documentation is located here.

UART communication

There are two UART pins connected from the XBIB-C-GPS to the host board by the 40-pin header: RX and TX. By default, the UART on the XBIB-C-GPS board is active and sends GPS readings to the connected device's secondary UART pins. Readings are transmitted once every second. The baud rate of the UART is 9600 baud.

For more information about using Micro Python to communicate to the XBIB-C-GPS module, see Class UART.

Run the MicroPython GPS demo

The Digi MicroPython github repository contains a GPS demo program that parses some of the GPS NMEA sentences from the UART, prints them and also reports them to Digi Remote Manager.

Note If you are unfamiliar with MicroPython on XBee you should first run some of the tutorials earlier in this manual to familiarize yourself with the environment. See Get started with MicroPython. For more detailed information, refer to the *Digi MicroPython Programming Guide*.

Step 1: Create a Remote Manager developer account

You must have a Remote Manager developer account to be able to use this program. Make sure you know the user name and password for this account.

If you don't currently have a Remote Manager developer account, you can create a free developer account.

Step 2: Download or clone the XBee MicroPython repository

1. Navigate to: **https://github.com/digidotcom/xbee-micropython/**
2. Click **Clone or download**.
3. You must either clone or download a zip file of the repository. You can use either method.
   - **Clone**: If you are familiar with GIT, follow the standard GIT process to clone the repository.
   - **Download**
     a. Click **Download zip** to download a zip file of the repository to the download folder of your choosing.
     b. Extract the repository to a location of your choosing on your hard drive.

Step 3: Edit the MicroPython file

1. Navigate to the location of the repository zip file that you created in Step 2.
2. Navigate to: **samples/gps**
3. Open the MicroPython file: *gpsdemo1.py*
4. Using the editor of your choice, edit the MicroPython file. At the top of the file, enter the user name and password for your Remote Manager developer account. The correct location is indicated in the comments in the file.

Step 4: Run the program

1. Rename the file you edited in Step 3 from *gpsdemo1.py* to *main.py*.
2. Copy the renamed file onto your device's root filesystem directory.
3. Copy the following three modules from the locations specified below into your device's **/lib** directory:
   - From the **/lib** directory of the Digi xbee-micropython repository: *urequest.py* and *remotemanager.py*
   - From the **/lib/sensor** directory of the Digi xbee-micropython repository: *hdc1080.py*

   Note These modules are required to be able to run the *gpsdemo1.py*.

4. Open XCTU and use the MicroPython Terminal to run the demo.
5. Type <CTRL>-R from the MicroPython prompt to run the code.

Cellular service

Digi now offers Cellular Bundled Service plans. This service includes pre-configured cellular data options that are ideal for IoT applications, bundled together with Digi Remote Manager for customers who want to remotely monitor and manage their devices.

To learn more, or obtain the plan that is right for your needs, contact us:

- By phone: 1-877-890-4014 (USA/toll free) or +1-952-912-3456 (International). Select the **Wireless Plan Support** or **Activation** option in the menu.
- By email: Data.Plan.QuoteDesk@digi.com.

WARNING! Digi Remote Manager is enabled by default on the XBee device. You should configure the device to avoid excess cellular data usage. For more information, see Configure Remote Manager keepalive interval.

The XBee Cellular kit includes six months of free cellular service. Six months of free cellular service assumes a rate of 5 MB/month. If you exceed a limit of 30 MB during the six month period your SIM will be deactivated.

Update the firmware on your XBee

You should update your XBee to the latest device firmware and cellular modem firmware.

Update modem firmware

- From XCTU. See Configure and update your XBee with XCTU.
- Update the modem firmware for XBee Cellular devices. See Update the modem firmware for XBee devices.

Configure and update your XBee with XCTU

You should update your XBee to the latest firmware available for the device.

XBee Configuration and Test Utility (XCTU) is a multi-platform program developed by Digi that enables users to interact with Digi radio frequency (RF) devices through a graphical interface. The application includes built-in tools that make it easy to set up, configure, and test Digi RF devices.

XCTU does not work directly over an SPI interface.

For instructions on downloading and using XCTU, see the *XCTU User Guide*.

Note If you are on a macOS computer and encounter problems installing XCTU, see Correct a macOS Java error.

Update the device firmware

You can use XCTU to update the firmware.

1. To use XCTU, you may need to install FTDI Virtual COM port (VCP) drivers onto your computer. Click here to download the drivers for your operating system.
2. Upgrade XCTU to version 6.4.2 or later. This step is required.
3. You must add a device to XCTU before you can update the device's firmware from XCTU.
4. Update to the latest modem firmware from XCTU.

Check for cellular registration and connection

You should verify proper cellular network registration and address assignment.

Add a device

These instructions show you how to add the XBee Cellular Modem to XCTU.

If XCTU does not find your serial port, see Cannot find the serial port for the device and Enable Virtual COM port (VCP) on the driver.

1. Launch XCTU.

   Note XCTU's **Update the radio module firmware** dialog box may open and will not allow you to continue until you click **Update** or **Cancel** on the dialog.

2. Click **Help** > **Check for XCTU Updates** to ensure you are using the latest version of XCTU.
3. Click the **Discover radio modules** button in the upper left side of the XCTU screen.
4. In the **Discover radio devices** dialog, select the serial ports where you want to look for XBee modules, and click **Next**.
5. In the **Set port parameters** window, maintain the default values and click **Finish**.

6. As XCTU locates radio modules, they appear in the **Discovering radio modules** dialog box.
7. Select the device(s) you want to add and click **Add selected devices**.

If your module could not be found, XCTU displays the **Could not find any radio module** dialog providing possible reasons why the module could not be added.

Update to the latest modem firmware from XCTU

Firmware is the program code stored in the device's persistent memory that provides the control program for the device. Use XCTU to update the firmware.

Note If you have already updated the firmware in a previous step, this process is not necessary.

1. Launch XCTU .
2. Click the **Configuration working modes** button .
3. Select a local XBee module from the **Radio Modules** list.
4. Click the **Update firmware** button to ensure you have the most current firmware.

   The **Update firmware** dialog displays the available and compatible firmware for the selected XBee module.
5. Make sure you check the **Force the module to maintain its current configuration** box and then click **Update**.
6. Select the product family of the XBee module, the function set, and the latest firmware version.
7. Click **Update**. A dialog displays update progress. Click **Show details** for details of the firmware update process.

See How to update the firmware of your modules in the *XCTU User Guide* for more information.

Check for cellular registration and connection

In the following examples, proper cellular network registration and address assignment must occur successfully. The LED on the development board blinks when the XBee Cellular Modem is registered to the cellular network; see Associate LED functionality. If the LED remains solid, registration has not occurred properly.

Registration can take several minutes.

Note Make sure you are in an area with adequate cellular network reception or the XBee Cellular Modem will not make the connection.

Note Check the antenna connections if the device has trouble connecting to the network.

In addition to the LED confirmation, you can check the AT commands below in XCTU to check the registration and connection.

To view these commands:

1. Open XCTU.
2. Add a device.
3. Click the **Configuration working mode** button.

4. Select a device from the **Radio Modules** list. XCTU displays the current firmware settings for that device.
5. Update to the latest modem firmware from XCTU.

Note To search for an AT command in XCTU, use the search box .

The relevant commands are:

- **AI (Association Indication)** reads **0** when the device successfully registers to the cellular network. If it reads **23** it is connecting to the Internet; **22** means it is registering to the cellular network.
- **MY (Module IP Address)** should display a valid IP address. If it reads **0.0.0.0**, it has not registered yet.

Note To read a command's value, click the **Read** button next to the command.

Update the modem firmware for XBee devices

This process explains how to update the modem firmware for XBee Cellular devices.

Update the modem firmware

1. Make sure you have the correct version of the modem firmware for your XBee device.
2. Enter programming (bootloader) mode. Use one of the following methods: AT commands or hardware signaling.
   - **AT commands**
     a. Send the %P command. The %P command must be sent an argument derived from the SL parameter of the module being updated. The argument is the value of SL added to the value 0xDB8A and then masked by performing a bitwise-AND with 0x3FFF.
        i. Run ATSL to get the address value, which is in hex.

        ```
        ATSL
        123456
        ```

        ii. Add bitwise-AND with 0x3FFF.

        ```
        (0xDB8A + 0x123456) & 0x3FFF= 0x0FE0
        ```

        iii. Send the command AT%PFE0.

        ```
        AT%PFE0
        ```

     b. You will receive an error, which is expected.
     c. Send the FR command to reboot and enter into bootloader.
   - **Invoke the bootloader with hardware signaling**
     a. De-Assert RTS (pin 16).
     b. Assert DTR (pin 9).
     c. Put DIN in a low state (break) (pin 3).
     d. Reset the module (pin 5).
     e. Release the break on DIN (pin 3) The module should now be in bootloader at 38400 baud.
3. Once the module is in programming (bootloader) mode, configure the local serial port to 38400/8/N/1.

4. Get the hardware version of the radio module from the bootloader.
    a. Send the V command. The response to that command has the following format:

XXXXYYYYZZAABBBBCCCCCCCCCCCCCCCC	■ **XXXX**: The hardware version. See ATHV, little endian. ■ **YYYY**: The hardware revision. See AT%R, little endian. ■ **ZZ**: The hardware compatibility number. See AT%C. ■ **AA**: Unused and should be 0. ■ **BBBB**: The hardware series. See ATHS, little endian. ■ **CCCCCCCCCCCCCCCC**: The serial number.

5. If possible, change the baud rate of the serial port to optimize the firmware update process. Send the X command to the bootloader.
    - The bootloader answers with the maximum supported baud rate (in ASCII) and, just after that, the bootloader changes its baud rate to that value. Change your baud rate to match the max supported rate.
    - If the bootloader does not answer to this command, remain at the current rate.
6. Send the I command (initialization command). This command erases the current firmware from the device.
7. Transfer the firmware to the device using the transfer protocol shown below.

Transfer the firmware to the device

1. You must split the file into 512 byte blocks.
2. Transfer each block using the following structure, with block index and CRC16 sent in little endian byte:

   ```
   P [2 bytes for block index] [block data with page size length] [2 bytes for
   CRC16]
   ```

   Note CRC16 is calculated only with the bytes of the page to be sent, and is initialized with 0x0000. The polynomial used for the CRC16 is 0x8005.

3. After each block is transfered, wait for a response. Options are:
   - 0x55 - ACK: This is the expected answer.
   - 0x12: Checksum/CRC16 error.
   - 0x13: Flash write/verify error.

   Note If an error occurs, you may try to transfer each block up to three times.

4. Verify and write the firmware to flash.
   a. Send the C command (verify) to verify and write the firmware to the flash.
   b. Verify that the answer to this command is 0x55 (ACK). Any other result is an error.
5. Wait a couple of seconds for the firmware to be installed and start running.

XBee connection examples

The following examples provide some additional scenarios you can try to get familiar with the XBee Cellular Modem. These examples are focused on inter-operating with a host processor to drive the XBee.

If you are interested in using the intelligence built into the XBee, see Get started with MicroPython.

Note Some carriers restrict your internet access. If access is restricted, running some of these examples may not be possible. Check with your carrier provider to determine whether internet access is restricted.

Connect to the Echo server

This server echoes back the messages you type.

Note For help with debugging, see Debugging.

The following table explains the AT commands that you use in this example.

At command	Value	Description
IP (IP Protocol)	1	Set the expected transmission mode to TCP communications.
TD (Text Delimiter)	D (0x0D)	The text delimiter to be used for Transparent mode, as an ASCII hex code. No information is sent until this character is entered, unless the maximum number of characters has been reached. Set to **0** to disable text delimiter checking. Set to **D** for a carriage return.
DL (Destination Address)	52.43.121.77	The target IP address of the echo server.
DE (Destination Port)	0x2329	The target port number of the echo server.

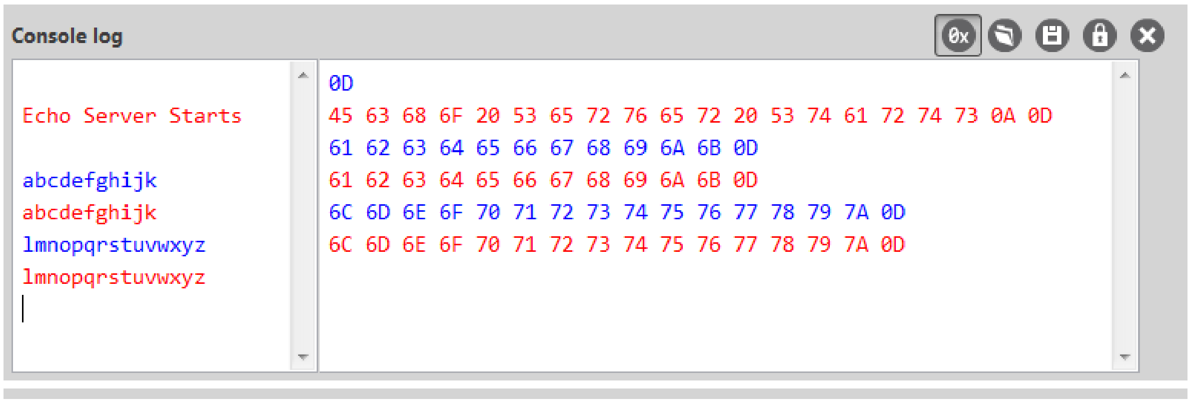
To communicate with the Echo server:

1. Ensure that the device is set up correctly with the SIM card installed and the antennas connected as described in Connect the hardware.
2. Open XCTU and Add a device.
3. Click the **Configuration working mode** button.
4. Select a device from the **Radio Modules** list. XCTU displays the current firmware settings for that device.
5. To switch to TCP communication, in the **IP** field, select 1 and click the **Write** button .
6. To enable the XBee Cellular Modem to recognize carriage return as a message delimiter, in the **TD** field, type **D** and click the **Write** button.
7. To enter the destination address of the echo server, in the **DL** field, type **52.43.121.77** and click the **Write** button.
8. To enter the destination IP port number, in the **DE** field, type **2329** and click the **Write** button.

   Note XCTU does not follow the standard hexadecimal numbering convention. The leading 0x is not needed in XCTU.

9. Click the **Consoles working mode** button on the toolbar to open a serial console to the device. For instructions on using the Console, see the AT console topic in the *XCTU User Guide*.

10. Click the **Open** button Open to open a serial connection to the device.
11. Click in the left pane of the **Console log**, then type in the Console to talk to the echo server. The following screenshot provides an example of this chat.



```
Echo Server Starts

abcdefghijk
abcdefghijk
lmnopqrstuvwxyz
lmnopqrstuvwxyz
```

```
0D
45 63 68 6F 20 53 65 72 76 65 72 20 53 74 61 72 74 73 0A 0D
61 62 63 64 65 66 67 68 69 6A 6B 0D
61 62 63 64 65 66 67 68 69 6A 6B 0D
6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 0D
6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 0D
```

Connect to the ELIZA server

You can use the XBee Cellular Modem to chat with the ELIZA Therapist Bot. ELIZA is an artificial intelligence (AI) bot that emulates a therapist and can perform simple conversations.

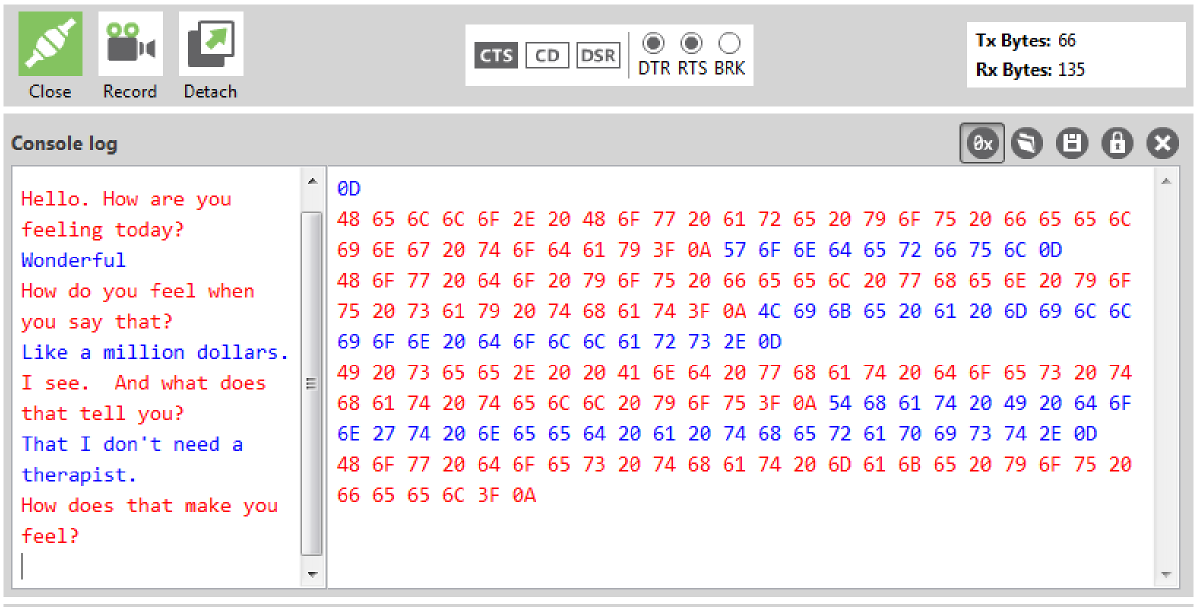
Note For help with debugging, see Debugging.

The following table explains the AT commands that you use in this example.

At command	Value	Description
IP (IP Protocol)	1	Set the expected transmission mode to TCP communications.
DL (Destination Address)	52.43.121.77	The target IP address of the ELIZA server.
DE (Destination Port)	0x2328	The target port number of the ELIZA server.

To communicate with the ELIZA Therapist Bot:

1. Ensure that the device is set up correctly with the SIM card installed and the antennas connected as described in Connect the hardware.
2. Open XCTU and Add a device.
3. Click the **Configuration working mode** button.
4. Select a device from the **Radio Modules** list. XCTU displays the current firmware settings for that device.
5. To switch to TCP communication, in the **IP** field, select 1 and click the **Write** button.
6. To enter the destination address of the ELIZA Therapist Bot, in the **DL** field, type **52.43.121.77** and click the **Write** button.
7. To enter the destination IP port number, in the **DE** field, type **2328** and click the **Write** button.
8. Click the **Consoles working mode** button on the toolbar to open a serial console to the device. For instructions on using the Console, see the AT console topic in the *XCTU User Guide*.
9. Click the **Open** button to open a serial connection to the device.
10. Click in the left pane of the **Console log**, then type in the Console to talk to the ELIZA Therapist Bot. The following screenshot provides an example of this chat with the user's text in blue.



```
Hello. How are you
feeling today?
Wonderful
How do you feel when
you say that?
Like a million dollars.
I see.  And what does
that tell you?
That I don't need a
therapist.
How does that make you
feel?
```

```
0D
48 65 6C 6C 6F 2E 20 48 6F 77 20 61 72 65 20 79 6F 75 20 66 65 65 6C
69 6E 67 20 74 6F 64 61 79 3F 0A 57 6F 6E 64 65 72 66 75 6C 0D
48 6F 77 20 64 6F 20 79 6F 75 20 66 65 65 6C 20 77 68 65 6E 20 79 6F
75 20 73 61 79 20 74 68 61 74 3F 0A 4C 69 6B 65 20 61 20 6D 69 6C 6C
69 6F 6E 20 64 6F 6C 6C 61 72 73 2E 0D
49 20 73 65 65 2E 20 20 41 6E 64 20 77 68 61 74 20 64 6F 65 73 20 74
68 61 74 20 74 65 6C 6C 20 79 6F 75 3F 0A 54 68 61 74 20 49 20 64 6F
6E 27 74 20 6E 65 65 64 20 61 20 74 68 65 72 61 70 69 73 74 2E 0D
48 6F 77 20 64 6F 65 73 20 74 68 61 74 20 6D 61 6B 65 20 79 6F 75 20
66 65 65 6C 3F 0A
```

Connect to the Daytime server

The Daytime server reports the current Coordinated Universal Time (UTC) value responding to any user input.

Note For help with debugging, see Debugging.

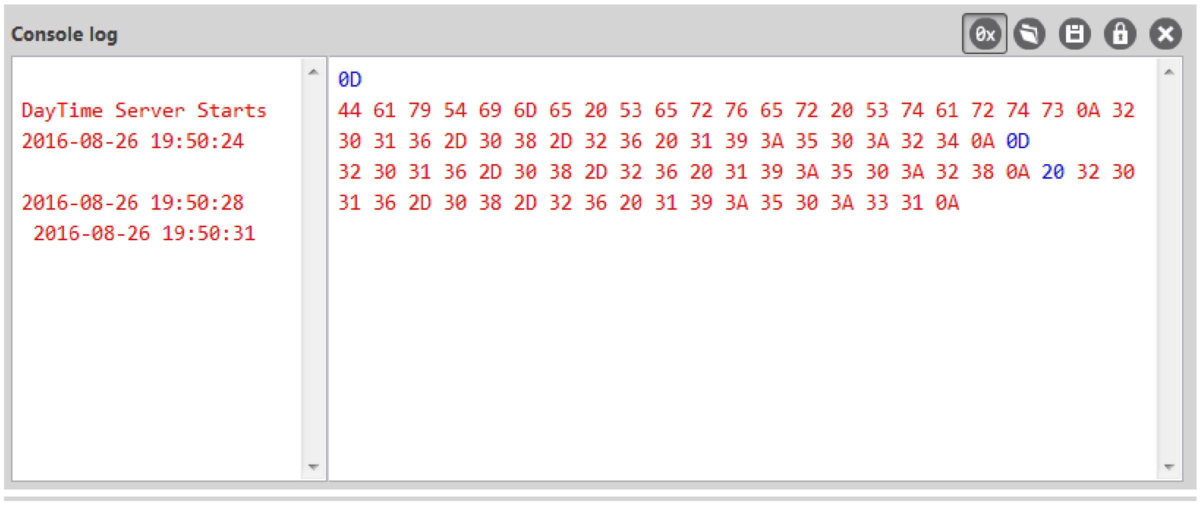
The following table explains the AT commands that you use in this example.

At command	Value	Description
IP (IP Protocol)	1	Set the expected transmission mode to TCP communications.
DL (Destination Address)	52.43.121.77	The target IP of the Daytime server.
DE (Destination Port)	0x232A	The target port number of the Daytime server.
TD (Text Delimiter)	0	The text delimiter to be used for Transparent mode, as an ASCII hex code. No information is sent until this character is entered, unless the maximum number of characters has been reached. Set to zero to disable text delimiter checking.

To communicate with the Daytime server:

1. Ensure that the device is set up correctly with the SIM card installed and the antennas connected as described in Connect the hardware.
2. Open XCTU and Add a device.
3. Click the **Configuration working mode** button.
4. Select a device from the **Radio Modules** list. XCTU displays the current firmware settings for that device.
5. To switch to TCP communication, in the **IP** field, select 1 and click the **Write** button .
6. To enter the destination address of the daytime server, in the **DL** field, type **52.43.121.77** and click the **Write** button.
7. To enter the destination IP port number, in the **DE** field, type **232A** and click the **Write** button.
8. To disable text delimiter checking, in the **TD** field, type **0** and click the **Write** button.
9. Click the **Consoles working mode** button on the toolbar to open a serial console to the device. For instructions on using the Console, see the AT console topic in the *XCTU User Guide*.
10. Click the **Open** button Open to open a serial connection to the device.

11. Click in the left pane of the **Console log**, then type in the Console to query the Daytime server. The following screenshot provides an example of this chat.



```
DayTime Server Starts
2016-08-26 19:50:24

2016-08-26 19:50:28
 2016-08-26 19:50:31
```

```
0D
44 61 79 54 69 6D 65 20 53 65 72 76 65 72 20 53 74 61 72 74 73 0A 32
30 31 36 2D 30 38 2D 32 36 20 31 39 3A 35 30 3A 32 34 0A 0D
32 30 31 36 2D 30 38 2D 32 36 20 31 39 3A 35 30 3A 32 38 0A 20 32 30
31 36 2D 30 38 2D 32 36 20 31 39 3A 35 30 3A 33 31 0A
```

Send an SMS message to a phone

The XBee Cellular Modem can send and receive Short Message Service (SMS) transmissions (text messages) while in Transparent mode. This allows you to send and receive text messages to and from an SMS capable device such as a mobile phone.

Note For help with debugging, see Debugging.

The following table explains the AT commands that you use in this example.

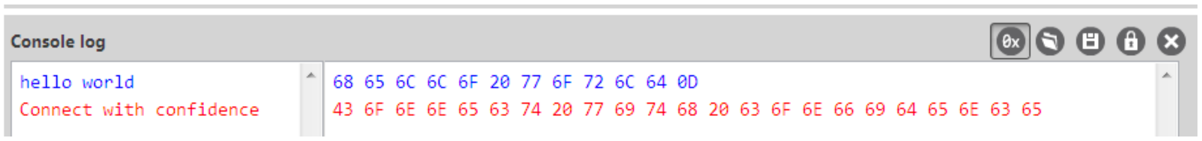
Command	Value	Description
AP (API Enable)	0	Set the device's API mode to Transparent mode.
IP (IP Protocol)	2	Set the expected transmission mode to SMS communication.
P# (Destination Phone Number)	<Target phone number>	The target phone number that you send to, for example, your cellular phone. See P# (Destination Phone Number) for instructions on using this command.
TD (Text Delimiter)	D (0x0D)	The text delimiter to be used for Transparent mode, as an ASCII hex code. No information is sent until this character is entered, unless the maximum number of characters has been reached. Set to **0** to disable text delimiter checking. Set to **D** for a carriage return.
PH (Module's SIM phone number)	Read only	The value that represents your device's phone number as supplied by the SIM card. This is used to send text messages to the device from another cellular device.

1. Ensure that the device is set up correctly with the SIM card installed and the antennas connected as described in Connect the hardware.
2. Open XCTU and Add a device.
3. Click the **Configuration working mode** button.
4. Select a device from the **Radio Modules** list. XCTU displays the current firmware settings for that device.
5. To switch to SMS communication, in the **IP** field, select **2** and click the **Write** button .
6. To enter your cell phone number, in the **P#** field, type the <**target phone number**> and click the **Write** button. Type the phone number using only numbers, with no dashes. You can use the **+** prefix if necessary. The target phone number is the phone number you wish to send a text to.
7. In the **TD** field, type **D** and click the **Write** button.
8. Note the number in the **PH** field; it is the XBee Cellular Modem phone number, which you see when it sends an SMS to your phone.
9. Click the **Consoles working mode** button on the toolbar to open a serial console to the device. For instructions on using the Console, see the AT console topic in the *XCTU User Guide*.

10. Click the **Open** button to open a serial connection to the device.
11. Click in the left pane of the **Console log**, type **hello world** and press **Enter**. The XBee Cellular Modem sends the message to the destination phone number set by the **P#** command.

Note If you are receiving individual characters, verify that you set **TD** correctly.

12. When the phone receives the text, you can see that the sender's phone number matches the value reported by the XBee Cellular Modem with the **PH** command.
13. On the phone, reply with the text **connect with confidence** and the XBee Cellular Modem outputs this reply from the UART.



```
hello world                 68 65 6C 6C 6F 20 77 6F 72 6C 64 0D
Connect with confidence     43 6F 6E 6E 65 63 74 20 77 69 74 68 20 63 6F 6E 66 69 64 65 6E 63 65
```

Perform a (GET) HTTP request

You can use the XBee Cellular Modem to perform a GET Hypertext Transfer Protocol (HTTP) request using XCTU. HTTP is an application-layer protocol that runs over TCP. This example uses httpbin.org/ as the target website that responds to the HTTP request.

Note For help with debugging, see Debugging.

To perform a GET request:

1. Ensure that the device is set up correctly with the SIM card installed and the antennas connected as described in Connect the hardware.
2. Open XCTU and Add a device.
3. Click the **Configuration working mode** button.
4. Select a device from the **Radio Modules** list. XCTU displays the current firmware settings for that device.
5. To enter the destination address of the target website, in the **DL** field, type **httpbin.org** and click the **Write** button .
6. To enter the HTTP request port number, in the **DE** field, type **50** and click the **Write** button. Hexadecimal **50** is 80 in decimal.
7. To switch to TCP communication, in the **IP** field, select **1** and click the **Write** button.
8. To move into Transparent mode, in the **AP** field, select **0** and click the **Write** button.
9. Wait for the **AI** (Association Indication) value to change to **0** (Connected to the Internet).
10. Click the **Consoles working mode** button on the toolbar.
11. From the AT console, click the **Add new packet button** in the Send packets dialog. The **Add new packet** dialog appears.
12. Enter the name of the data packet.
13. Type the following data in the **ASCII** input tab:

    ```
    GET /ip HTTP/1.1
    Host: httpbin.org
    ```
14. Click the **HEX** input tab and add **0A** (zero A) after each **0D** (zero D), and add an additional **0D 0A** at the end of the message body. For example, copy and past the following text into the **HEX** input tab:

    ```
    47 45 54 20 2F 69 70 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 68 74 74 70 62 69 6E
    2E 6F 72 67 0D 0A 0D 0A
    ```

Note The HTTP protocol requires an empty line (a line with nothing preceding the CRLF) to terminate the request.

15. Click **Add packet**.
16. Click the **Open** button .
17. Click **Send selected packet**.
18. A GET HTTP response from httpbin.org appears in the Console log.

Get started with CoAP

Constrained Application Protocol (CoAP) is based on UDP connection and consumes low power to deliver similar functionality to HTTP. This guide contains information about sending GET, POST, PUT and DELETE operations by using the Coap Protocol with XCTU and Python code working with the XBee Cellular Modem and Coapthon library (Python 2.7 only).

The Internet Engineering Task Force describes CoAP as:

> The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation. CoAP provides a request/response interaction model between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URIs and Internet media types. CoAP is designed to easily interface with HTTP for integration with the Web while meeting specialized requirements such as multicast support, very low overhead, and simplicity for constrained environments (source).

CoAP terms

When describing CoAP, we use the following terms:

Term	Meaning
Method	COAP's method action is similar to the HTTP method. This guide discusses the GET, POST, PUT and DELETE methods. With these methods, the XBee Cellular Modem can transport data and requests.
URI	URI is a string of characters that identifies a resource served at the server.
Token	A token is an identifier of a message. The client uses the token to verify if the received message is the correct response to its query.
Payload	The message payload is associated with the POST and PUT methods. It specifies the data to be posted or put to the URI resource.
Message ID	The message ID is also an identifier of a message. The client matches the message ID between the response and query.

CoAP quick start example

The following diagram shows the message format for the CoAP protocol; see ISSN: 2070-1721 for details:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  TKL  |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Token (if any, TKL bytes) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|    Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This is an example GET request:

```
44 01 C4 09 74 65 73 74 B7 65 78 61 6D 70 6C 65
```

The following table describes the fields in the GET request.

Field	HEX	Bits	Meaning
Ver	44	01	Version 01, which is mandatory here.
T		00	Type 0: confirmable.
TKL		0100	Token length: 4.
Code	01	000 00001	Code: 0.01, which indicates the GET method.
Message ID	C4 09	2 Bytes equal to hex at left	Message ID. The response message will have the same ID. This can help out identification.
Token	74 65 73 74	4 Bytes equal to hex at left	Token. The response message will have the same token. This can help out identification.
Option delta	B7	1011	Delta option: 11 indicates the option data is Uri-Path.
Option length		0111	Delta length: 7 indicates there are 7 bytes of data following as a part of this delta option.
Option value	65 78 61 6D 70 6C 65	7 Bytes equal to hex at left	Example.

Configure the device

1. Ensure that the device is set up correctly with the SIM card installed and the antennas connected as described in Connect the hardware.
2. Open XCTU and click the **Configuration working mode** button.
3. Add the XBee Cellular Modem to XCTU; see Add a device.
4. Select a device from the **Radio Modules** list. XCTU displays the current firmware settings for that device.
5. To switch to UDP communication, in the **IP** field, select **0** and click the **Write** button .
6. To set the target IP address that the XBee Cellular Modem will talk to, in the **DL** field type **52.43.121.77**and click the **Write** button . A CoAP server is publicly available at address 52.43.121.77.
7. To set the XBee Cellular Modem to send data to port 5683 in decimal, in the **DE** field, type **1633** and click the **Write** button.
8. To move into Transparent mode, in the **AP** field, select **0** and click the **Write** button.
9. Wait for the **AI** (Association Indication) value to change to **0** (Connected to the Internet). You can click **Read** to get an update on the **AI** value.

Example: manually perform a CoAP request

Follow the steps in Configure the device prior to this example. This example performs the CoAP GET request:

- Method: GET
- URI: example
- Given message token: test

1. Click the **Consoles working mode** button on the toolbar to add a customized packet.
2. From the AT console, click the **Add new packet button** in the Send packets dialog. The **Add new packet** dialog appears.
3. Click the **HEX** tab and type the name of the data packet: **GET_EXAMPLE**.
4. Copy and past the following text into the **HEX** input tab:
   44 01 C4 09 74 65 73 74 B7 65 78 61 6D 70 6C 65
   This is the CoAP protocol message decomposed by bytes to perform a GET request on an example URI with a token test.
5. Click **Add packet**.
6. Click the **Open** button .
7. Click **Send selected packet**. The message is sent to the public CoAP server configured in Configure the device. A response appears in the Console log. Blue text is the query, red text is the response.

The payload is **Get to uri: example**, which specifies that this is a successful CoAP GET to URI end example, which was specified in the query.
Click the **Close** button to terminate the serial connection.

Example: use Python to generate a CoAP message

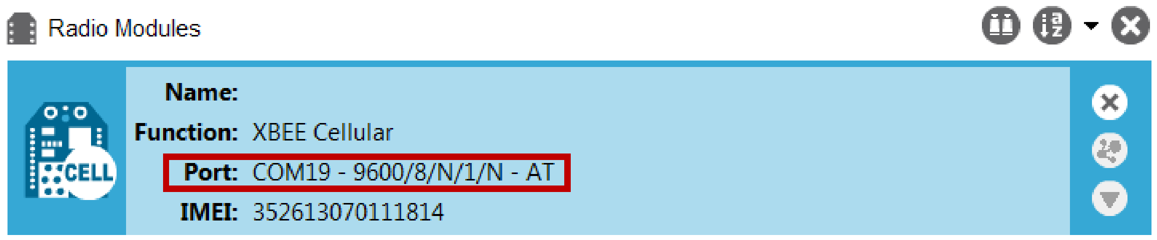
This example illustrates how the CoAP protocol can perform GET/POST/PUT/DELETE requests similarly to the HTTP protocol and how to do this using the XBee Cellular Modem. In this example, the XBee Cellular Modem talks to a CoAP Digi Server. You can use this client code to provide an abstract wrapper to generate a CoAP message that commands the XBee Cellular Modem to talk to the remote CoAP server.

Note It is crucial to configure the XBee Cellular Modem settings. See Configure the device and follow the steps. You can target the IP address to a different CoAP public server.

1. Install Python 2.7. The Installation guide is located at: python.org/downloads/.
2. Download and install the CoAPthon library in the python environment from pypi.python.org/pypi/CoAPthon.
3. Download these two .txt files: Coap.txt and CoapParser.txt. After you download them, open the files in a text editor and save them as .py files.
4. In the folder that you place the Coap.py and CoapParser.py files, press **Shift** + **right-click** and then click **Open command window**.
5. At the command prompt, type **python Coap.py** and press **Enter** to run the program.

6. Type the USB port number that the XBee Cellular Modem is connected to and press **Enter**. Only the port number is required, so if the port is COM19, type 19.

Note If you do not know the port number, open XCTU and look at the XBee Cellular Modem in the **Radio Modules** list. This view provides the port number and baud rate, as in the figure below where the baud rate is 9600 b/s.

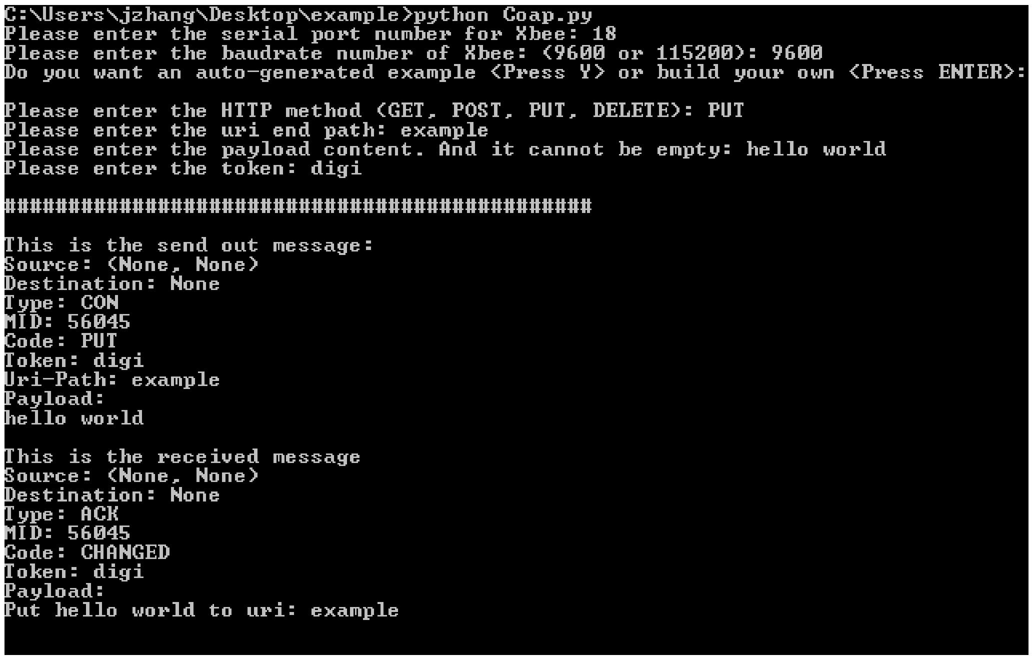


Name:
Function: XBEE Cellular
Port: COM19 - 9600/8/N/1/N - AT
IMEI: 352613070111814

7. Type the baud rate and press **Enter**. You must match the device's current baud rate.
   XCTU provides the current baud rate in the **BD Baud Rate** field. In this example you would type **9600**.
8. Press **Y** if you want an auto-generated example. Press **Enter** to build your own CoAP request.
9. If you press **Y** it generates a message with:
   - Method: POST
   - URI: example
   - payload: hello world
   - token: test

The send and receive message must match the same token and message id. Otherwise, the client re-attempts the connection by sending out the request.

In the following figure, the payload contains the server response to the query. It shows the results for when you press **Enter** rather than **Y**.



```
C:\Users\jzhang\Desktop\example>python Coap.py
Please enter the serial port number for Xbee: 18
Please enter the baudrate number of Xbee: (9600 or 115200): 9600
Do you want an auto-generated example (Press Y) or build your own (Press ENTER):

Please enter the HTTP method (GET, POST, PUT, DELETE): PUT
Please enter the uri end path: example
Please enter the payload content. And it cannot be empty: hello world
Please enter the token: digi

##############################################

This is the send out message:
Source: (None, None)
Destination: None
Type: CON
MID: 56045
Code: PUT
Token: digi
Uri-Path: example
Payload:
hello world

This is the received message
Source: (None, None)
Destination: None
Type: ACK
MID: 56045
Code: CHANGED
Token: digi
Payload:
Put hello world to uri: example
```

Connect to a TCP/IP address

The XBee Cellular Modem can send and receive TCP messages while in Transparent mode; see Transparent operating mode.

Note You can use this example as a template for sending and receiving data to or from any TCP/IP server.

Note For help with debugging, see Debugging.

The following table explains the AT commands that you use in this example.

Command	Value	Description
IP (IP Protocol)	1	Set the expected transmission mode to TCP communication.
DL (Destination IP Address)	<Target IP address>	The target IP address that you send and receive from. For example, a data logging server's IP address that you want to send measurements to.
DE (Destination Port)	<Target port number>	The target port number that the device sends the transmission to. This is represented as a hexadecimal value.

To connect to a TCP/IP address:

1. Ensure that the device is set up correctly with the SIM card installed and the antennas connected as described in Connect the hardware.
2. Open XCTU and Add a device.
3. Click the **Configuration working mode** ⚙ button.
4. Select a device from the **Radio Modules** list. XCTU displays the current firmware settings for that device.
5. In the **IP** field, select 1 and click the **Write** button ✎.
6. In the **DL** field, type the <**target IP address**> and click the **Write** button. The target IP address is the IP address that you send and receive from.
7. In the **DE** field, type the <**target port number**>, converted to hexadecimal, and click the **Write** button.
8. Exit Command mode.

After exiting Command mode, any UART data sent to the device is sent to the destination IP address and port number after the RO (Packetization Timeout) occurs.

Get started with MQTT

MQ Telemetry Transport (MQTT) is a messaging protocol that is ideal for the Internet of Things (IoT) due to a light footprint and its use of the publish-subscribe model. In this model, a client connects to a broker, a server machine responsible for receiving all messages, filtering them, and then sending messages to the appropriate clients.

The first two MQTT examples do not involve the XBee Cellular Modem. They demonstrate using the MQTT libraries because those libraries are required for Use MQTT over the XBee Cellular Modem with a PC.

The examples in this guide assume:

- Some knowledge of Python.
- An integrated development environment (IDE) such as PyCharm, IDLE or something similar.

The examples require:

- An XBee Cellular Modem.
- A compatible development board, such as the XBIB-U.
- XCTU. See Configure and update your XBee with XCTU.
- That you install Python on your computer. You can download Python from: https://www.python.org/downloads/.
- That you install the **pyserial** and **paho-mqtt** libraries to the Python environment. If you use Python 2, install these libraries from the command line with **pip install pyserial** and **pip install paho-mqtt**. If you use Python 3, use **pip3 install pyserial** and **pip3 install paho-mqtt**.
- The full MQTT library source code, which includes examples and tests, which is available in the paho-mqtt github repository at https://github.com/eclipse/paho.mqtt.python. To download this repository you must have Git installed.

Example: MQTT connect

This example provides insight into the structure of packets in MQTT as well as the interaction between the client and broker. MQTT uses different packets to accomplish tasks such as connecting, subscribing, and publishing. You can use XCTU to perform a basic example of sending a broker a connect packet and receiving the response from the server, without requiring any coding. This is a good way to see how the client interacts with the broker and what a packet looks like. The following table is an example connect packet:

	Description	Hex value
CONNECT packet fixed header		
byte 1	Control packet type	0x10
byte 2	Remaining length	0x10
CONNECT packet variable header		
Protocol name		

	Description	Hex value
byte 1	Length MSB (0)	0x00
byte 2	Length LSB (4)	0x04
byte 3	(M)	0x4D
byte 4	(Q)	0x51
byte 5	(T)	0x54
byte 6	(T)	0x54
Protocol level		
byte 7	Level (4)	0x04
Connect flags		
byte 8	CONNECT flags byte, see the table below for the bits.	0X02
Keep alive		
byte 9	Keep Alive MSB (0)	0X00
byte 10	Keep Alive LSB (60)	0X3C
Client ID		
byte 11	Length MSB (0)	0x00
byte 12	Length LSB (4)	0x04
byte 13	(D)	0x44
byte 14	(I)	0x49
byte 15	(G)	0x47
byte 16	(I)	0x49

The following table describes the fields in the packet:

Field name	Description
Protocol Name	The connect packet starts with the protocol name, which is MQTT. The length of the protocol name (in bytes) is immediately before the name itself.
Protocol Level	Refers to the version of MQTT in use, in this case a value of 4 indicates MQTT version 3.1.1.
Connect Flags	Indicate certain aspects of the packet. For simplicity, this example only sets the Clean Session flag, which indicates to the client and broker to discard any previous session and start a new one.
Keep Alive	How often the client pings the broker to keep the connection alive; in this example it is set to 60 seconds.

Field name	Description
Client ID	The length of the ID (in bytes) precedes the ID itself. Each client connecting to a broker must have a unique client ID. In the example, the ID is DIGI. When using the Paho MQTT Python libraries, a random alphanumeric ID is generated if you do not specify an ID.

The following table provides the CONNECT flag bits from byte 8, the CONNECT flags byte.

CONNECT Flag Bit(s)	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
User name flag	0							
Password flag		0						
Will retain			0					
Will QoS				0	0			
Will flag						0		
Clean session							1	
Reserved								0

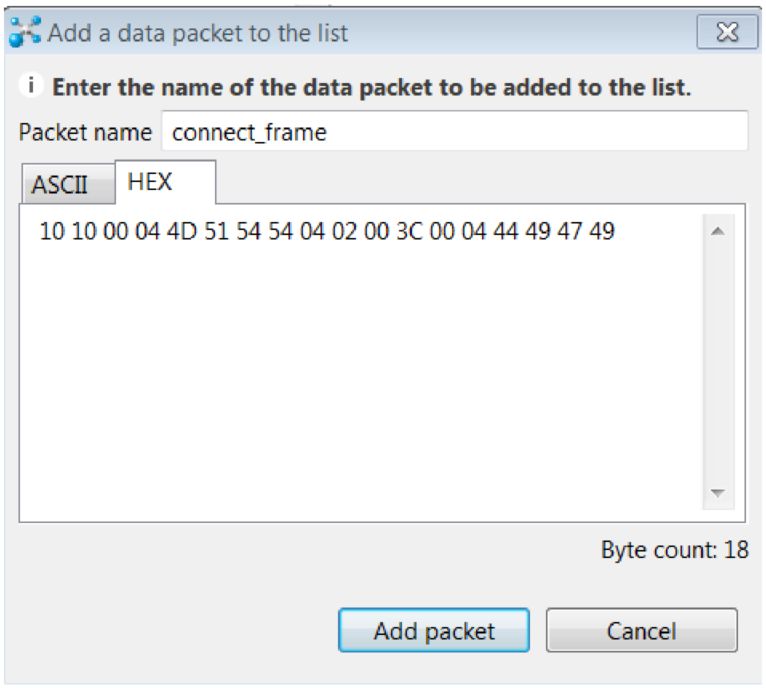
Send a connect packet

Now that you know what a connect packet looks like, you can send a connect packet to a broker and view the response. Open XCTU and click the Configuration working mode button.

1. Ensure that the device is set up correctly with the SIM card installed and the antennas connected as described in Connect the hardware.
2. Open XCTU and click the **Configuration working mode** button.
3. Add the XBee Cellular Modem to XCTU. See Add a device.
4. Select a device from the **Radio Modules** list. XCTU displays the current firmware settings for that device.
5. In the **AP** field, set **Transparent Mode** to **[0]** if it is not already and click the **Write** button.
6. In the **DL** field, type the IP address or the fully qualified domain name of the broker you wish to use. This example uses test.mosquitto.org.
7. In the **DE** field, type **75B** and set the port that the broker uses. This example uses **75B**, because the default MQTT port is 1883 (0x75B).
8. Once you have entered the required values, click the **Write** button to write the changes to the XBee Cellular Modem.
9. Click the **Consoles working mode** button on the toolbar to open a serial console to the device. For instructions on using the Console, see the AT console topic in the *XCTU User Guide*.
10. Click the **Open** button to open a serial connection to the device.

11. From the AT console, click the **Add new packet button** in the **Send packets** dialog. The **Add new packet** dialog appears.
12. Enter the name of the data packet. Name the packet **connect_frame** or something similar.
13. Click the **HEX** input tab and type the following (these values are the same values from the table in Example: MQTT connect):

    10 10 00 04 4D 51 54 54 04 02 00 3C 00 04 44 49 47 49

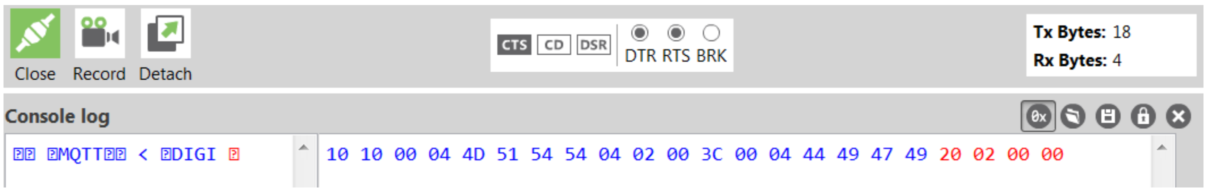

Add a data packet to the list
Enter the name of the data packet to be added to the list.
Packet name
ASCII
HEX
Add packet
Cancel

Packet name: connect_frame

```
10 10 00 04 4D 51 54 54 04 02 00 3C 00 04 44 49 47 49
```

Byte count: 18

14. Click **Add packet**. The new packet appears in the **Send packets** list.
15. Click the packet in the **Send packets** list.
16. Click **Send selected packet**.
17. A CONNACK packet response from the broker appears in the **Console log**. This is a connection acknowledgment; a successful response should look like this:



Tx Bytes: 18
Rx Bytes: 4

```
 MQTT < DIGI      10 10 00 04 4D 51 54 54 04 02 00 3C 00 04 44 49 47 49 20 02 00 00
```

You can verify the response from the broker as a CONNACK by comparing it to the structure of a CONNACK packet in the MQTT documentation, which is available at http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718081).

Example: send messages (publish) with MQTT

A basic Python example of a node publishing (sending) a message is:

```
mqttc = mqtt.Client("digitest")  # Create instance of client with client ID
“digitest”
mqttc.connect("m2m.eclipse.org", 1883)  # Connect to (broker, port,
keepalive-time)
mqttc.loop_start()  # Start networking daemon
mqttc.publish("digitest/test1", "Hello, World!")  # Publish message to
“digitest /test1” topic
mqttc.loop_stop()  # Kill networking daemon
```

Note You can easily copy and paste code from the online version of this guide. Use caution with the PDF version, as it may not maintain essential indentations.

This example imports the MQTT library, allowing you to use the MQTT protocol via APIs in the library, such as the **connect()**, **subscribe()**, and **publish()** methods.

The second line creates an instance of the client, named **mqttc**. The client ID is the argument you passed in: **digitest** (this is optional).

In line 3, the client connects to a public broker, in this case **m2m.eclipse.org**, on port **1883** (the default MQTT port, or 8883 for MQTT over TLS). There are many publicly available brokers available, you can find a list of them here: https://github.com/mqtt/mqtt.github.io/wiki/brokers.

Line 4 starts the networking daemon with **client.loop_start()** to handle the background network/data tasks.

Finally, the client publishes its message **Hello, World!** to the broker under the topic **digitest/backlog/test1**. Any nodes (devices, phones, computers, even microcontrollers) subscribed to that same topic on the same broker receive the message.

Once no more messages need to be published, the last line stops the network daemon with **client.loop_stop()**.

Example: receive messages (subscribe) with MQTT

This example describes how a client would receive messages from within a specific topic on the broker:

```
import paho.mqtt.client as mqtt


def on_connect(client, userdata, flags, rc):  # The callback for when the
client connects to the broker
    print("Connected with result code {0}".format(str(rc)))  # Print result
of connection attempt
    client.subscribe("digitest/test1")  # Subscribe to the topic
“digitest/test1”, receive any messages published on it


def on_message(client, userdata, msg):  # The callback for when a PUBLISH
message is received from the server.
    print("Message received-> " + msg.topic + " " + str(msg.payload))  #
Print a received msg


client = mqtt.Client("digi_mqtt_test")  # Create instance of client with
client ID “digi_mqtt_test”
client.on_connect = on_connect  # Define callback function for successful
connection
client.on_message = on_message  # Define callback function for receipt of a
```

```
message
# client.connect("m2m.eclipse.org", 1883, 60)  # Connect to (broker, port,
keepalive-time)
client.connect('127.0.0.1', 17300)
client.loop_forever()  # Start networking daemon
```

Note You can easily copy and paste code from the online version of this guide. Use caution with the PDF version, as it may not maintain essential indentations.

The first line imports the library functions for MQTT.

The functions **on_connect** and **on_message** are callback functions which are automatically called by the client upon connection to the broker and upon receiving a message, respectively.

The **on_connect** function prints the result of the connection attempt, and performs the subscription. It is wise to do this in the callback function as it guarantees the attempt to subscribe happens only after the client is connected to the broker.

The **on_message** function prints the received message when it comes in, as well as the topic it was published under.

In the body of the code, we:

- Instantiate a client object with the client ID **digi_mqtt_test**.
- Define the callback functions to use upon connection and upon message receipt.
- Connect to an MQTT broker at **m2m.eclipse.org**, on port **1883** (the default MQTT port, or 8883 for MQTT over TLS) with a keepalive of 60 seconds (this is how often the client pings the broker to keep the connection alive).

The last line starts a network daemon that runs in the background and handles data transactions and messages, as well as keeping the socket open, until the script ends.

Use MQTT over the XBee Cellular Modem with a PC

To use this MQTT library over an XBee Cellular Modem, you need a basic proxy that transfers a payload received via the MQTT client’s socket to the serial or COM port that the XBee Cellular Modem is active on, as well as the reverse; transfer of a payload received on the XBee Cellular Modem’s serial or COM port to the socket of the MQTT client. This is simplest with the XBee Cellular Modem in Transparent mode, as it does not require code to parse or create API frames, and not using API frames means there is no need for them to be queued for processing.

1. To put the XBee Cellular Modem in Transparent mode, set **AP** to **0**.
2. Set **DL** to the IP address of the broker you want to use.
3. Set **DE** to the port to use, the default is 1883 (0x75B). This sets the XBee Cellular Modem to communicate directly with the broker, and can be performed in XCTU as described in Example: MQTT connect.
4. You can make the proxy with a dual-threaded Python script, a simple version follows:

```
import threading
import serial
import socket


def setup():
    """
    This function sets up the variables needed, including the serial port,
    and it's speed/port settings, listening socket, and localhost adddress.
    """
    global clisock, cliaddr, svrsock, ser
    # Change this to the COM port your XBee Cellular module is using.  On
    # Linux, this will be /dev/ttyUSB#
    comport = 'COM44'
    # This is the default serial communication speed of the XBee Cellular
    # module
    comspeed = 115200
    buffer_size = 4096  # Default receive size in bytes
    debug_on = 0  # Enables printing of debug messages
    toval = None  # Timeout value for serial port below
    # Serial port object for XBCell modem
    ser = serial.Serial(comport,comspeed,timeout=toval)
    # Listening socket (accepts incoming connection)
    svrsock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    # Allow address reuse on socket (eliminates some restart errors)
    svrsock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    clisock = None
    cliaddr = None  # These are first defined before thread creation
    addrtuple = ('127.0.0.1', 17300)  # Address tuple for localhost
    # Binds server socket to localhost (allows client program connection)
    svrsock.bind(addrtuple)
    svrsock.listen(1)  # Allow (1) connection


def ComReaderThread():
    """
    This thread listens on the defined serial port object ('ser') for data
    from the modem, and upon receipt, sends it out to the client over the
    client socket ('clisock').
    """
    global clisock
    while (1):
        resp = ser.read()  ## Read any available data from serial port
        print("Received {} bytes from modem.".format(len(resp)))

        clisock.sendall(resp)  # Send RXd data out on client socket
        print("Sent {} byte payload out socket to client.".format(len
(resp)))


def SockReaderThread():
    """
    This thread listens to the MQTT client's socket and upon receiving a
    payload, it sends this data out on the defined serial port ('ser') to
the
    modem for transmission.
    """

    global clisock
    while (1):
        data = clisock.recv(4096)  # RX data from client socket
        # If the RECV call returns 0 bytes, the socket has closed
        if (len(data) == 0):
            print("ERROR - socket has closed.  Exiting socket reader
thread.")
```

```
                return 1  # Exit the thread to avoid a loop of 0-byte receptions
            else:
                print("Received {} bytes from client via socket.".format(len
(data)))
                print("Sending payload to modem...")
                bytes_wr = ser.write(data)  # Write payload to modem via
UART/serial
                print("Wrote {} bytes to modem".format(bytes_wr))

def main():
    setup()  # Setup the serial port and socket
    global clisock, svrsock
    if (not clisock):  # Accept a connection on 'svrsock' to open 'clisock'
        print("Awaiting ACCEPT on server sock...")
        (clisock,cliaddr) = svrsock.accept()  # Accept an incoming
connection
        print("Connection accepted on socket")
    # Make thread for ComReader
    comthread = threading.Thread(target=ComReaderThread)
    comthread.start()  # Start the thread
    # Make thread for SockReader
    sockthread = threading.Thread(target=SockReaderThread)
    sockthread.start()  # Start the thread

main()
```

Note This script is a general TCP-UART proxy, and can be used for other applications or scripts that use the TCP protocol. Its functionality is not limited to MQTT.

Note You can easily copy and paste code from the online version of this guide. Use caution with the PDF version, as it may not maintain essential indentations.

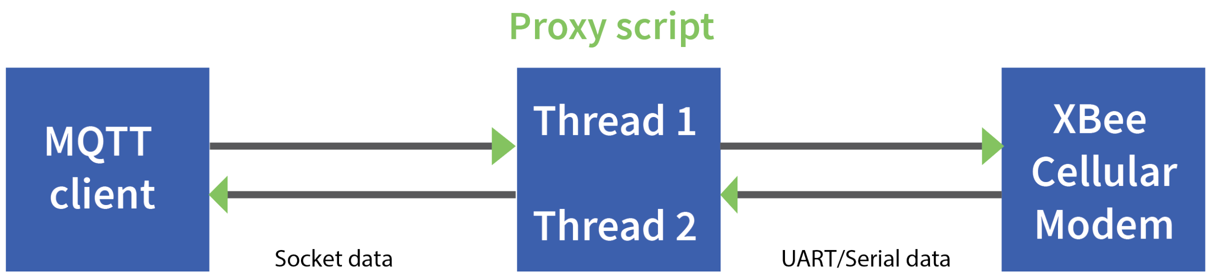
This proxy script waits for an incoming connection on localhost (**127.0.0.1**), on port **17300**. After accepting a connection, and creating a socket for that connection (**clisock**), it creates two threads, one that reads the serial or COM port that the XBee Cellular Modem is connected to, and one that reads the socket (**clisock**), that the MQTT client is connected to.

With:

- The proxy script running
- The MQTT client connected to the proxy script via localhost (**127.0.0.1**)
- The XBee Cellular Modem connected to the machine via USB and properly powered
- **AP**, **DL**, and **DE** set correctly

the proxy acts as an intermediary between the MQTT client and the XBee Cellular Modem, allowing the MQTT client to use the data connection provided by the device.

Think of the proxy script as a translator between the MQTT client and the XBee Cellular Modem. The following figure shows the basic operation.



The thread that reads the serial port forwards any data received onward to the client socket, and the thread reading the client socket forwards any data received onward to the serial port. This is represented in the figure above.

The proxy script needs to be running before running an MQTT publish or subscribe script.

1. With the proxy script running, run the subscribe example from Example: receive messages (subscribe) with MQTT, but change the connect line from **client.connect("m2m.eclipse.org", 1883, 60)** to **client.connect("127.0.0.1", port=17300, keepalive=20**). This connects the MQTT client to the proxy script, which in turn connects to a broker via the XBee Cellular Modem's internet connection.
2. Run the publish example from Example: send messages (publish) with MQTT in a third Python instance (while the publish script is running you will have three Python scripts running at the same time).

The publish script runs over your computer's normal Internet connection, and does not use the XBee Cellular Modem. You are able to see your published message appear in the subscribe script's output once it is received from the broker via the XBee Cellular Modem. If you watch the output of the proxy script during this process you can see the receptions and transmissions taking place.

The proxy script must be running before you run the subscribe and publish scripts. If you stop the subscribe script, the socket closes, and the proxy script shows an error. If you try to start the proxy script after starting the subscribe script, you may also see a socket error. To avoid these errors, it is best to start the scripts in the correct order: proxy, then subscribe, then publish.

Software libraries

One way to communicate with the XBee device is by using a software library. The libraries available for use with the XBee Cellular Modem include:

- XBee Java library
- XBee Python library

The XBee Java Library is a Java API. The package includes the XBee library, its source code and a collection of samples that help you develop Java applications to communicate with your XBee devices.

The XBee Python Library is a Python API that dramatically reduces the time to market of XBee projects developed in Python and facilitates the development of these types of applications, making it an easy process.

Debugging

If you experience problems with the settings in the examples, you can load the default settings in XCTU.

Note If you load the default settings, you will need to reapply any configuration settings that you have previously made.

1. On the Configuration toolbar, click the **Default** button to load the default values established by the firmware, and click **Yes** to confirm.
2. Factory settings are loaded but not written to the device. To write them, click the **Write** button on the toolbar.

Get started with MicroPython

This section provides an overview and simple examples of how to use MicroPython with the XBee Cellular Modem. You can use MicroPython to enhance the intelligence of the XBee to enable you to do edge-computing by adding business logic in MicroPython, rather than using external components.

Note For in-depth information and more complex code examples, refer to the *Digi MicroPython Programming Guide*.

About MicroPython

MicroPython is an open-source programming language based on Python 3, with much of the same syntax and functionality, but modified to fit on small devices with limited hardware resources, such as microcontrollers, or in this case, a cellular modem.

Why use MicroPython

MicroPython enables on-board intelligence for simple sensor or actuator applications using digital and analog I/O. MicroPython can help manage battery life. Cryptic readings can be transformed into useful data, excess transmissions can be intelligently filtered out, modern sensors and actuators can be employed directly, and logic can glue inputs and outputs together in an intelligent way.

For more information about MicroPython, see www.micropython.org.

For more information about Python, see www.python.org.

MicroPython on the XBee Cellular Modem

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.

Note MicroPython does not work with SPI.

The examples in this guide assume:

- You have XCTU on your computer. See Configure and update your XBee with XCTU.
- You have a terminal program installed on your computer. We recommend using the Use the MicroPython Terminal in XCTU. This requires XCTU 6.3.7 or higher.
- You have an XBee Cellular Modem installed in an appropriate development board such as an XBIB-U-DEV.

Note Most examples in this guide require the XBIB-U-DEV board.

- The XBee Cellular Modem is connected to the computer via a USB cable and XCTU recognizes it.
- The board is powered by an appropriate power supply, 12 VDC and at least 1.1 A.

Use XCTU to enter the MicroPython environment

To use the XBee Cellular Modem in the MicroPython environment:

1. Use XCTU to add the device(s); see Configure and update your XBee with XCTU and Add a device.
2. The XBee Cellular Modem appears as a box in the **Radio Modules** information panel. Each module displays identifying information about itself.
3. Click this box to select the device and load its current settings.

4. Set the device's baud rate to 115200 b/s, in the **BD** field select **115200 [7]** or higher and click the **Write** button. We recommend using flow control to avoid data loss, especially when pasting large amounts of code/text.
5. Put the XBee Cellular Modem into MicroPython mode, in the **AP** field select **MicroPython REPL [4]** and click the **Write** button.
6. Note what COM port(s) the XBee Cellular Modem is using, because you will need this information when you use terminal communication. The **Radio Modules** information panel lists the COM port in use.

Use the MicroPython Terminal in XCTU

You can use the MicroPython Terminal to communicate with the XBee Cellular Modem when it is in MicroPython mode.[1] This requires XCTU 6.3.7 or higher. To enter MicroPython mode, follow the steps in Use XCTU to enter the MicroPython environment. To use the MicroPython Terminal:

1. Click the **Tools** drop-down menu and select **MicroPython Terminal**. The terminal opens.
2. Click **Open**. If you have not already added devices to XCTU:
   a. In the **Select the Serial/USB port** area, click the COM port that the device uses.
   b. Verify that the baud rate and other settings are correct.
3. Click **OK**. The **Open** icon changes to **Close**, indicating that the device is properly connected.
4. Press **Ctrl**+**B** to get the MicroPython version banner and prompt.

You can now type or paste MicroPython commands at the **>>>** prompt.

Troubleshooting

If you receive **No such port: 'Port is already in use by other applications.'** in the **MicroPython Terminal** close any other console sessions open inside XCTU and close any other serial terminal programs connected to the device, then retry the MicroPython connection in XCTU.

If the device seems unresponsive, try pressing **Ctrl+C** to end any running programs.

You can use the **+++** escape sequence and look for an **OK** for confirmation that you have the correct baud rate.

Example: hello world

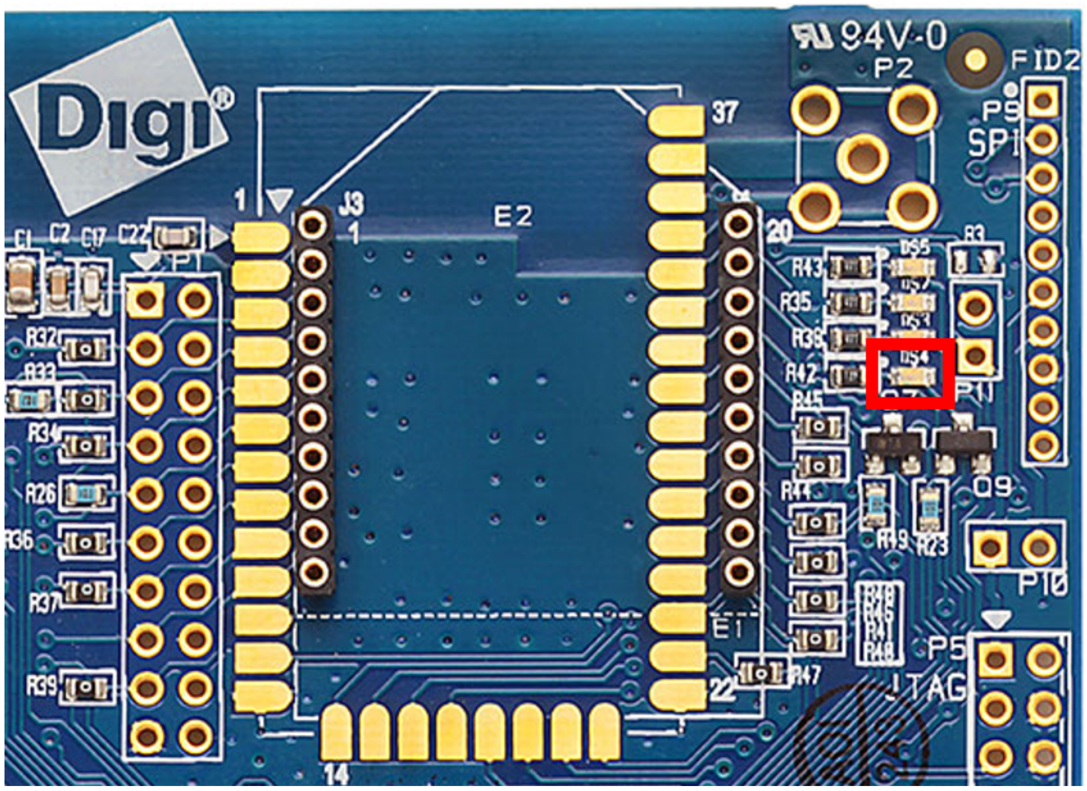
Before you begin, you must have previously added a device in XCTU. See Add a device.

1. At the MicroPython **>>>** prompt, type the Python command: **print("Hello, World!")**
2. Press **Enter** to execute the command. The terminal echos back **Hello, World!**.

1See Other terminal programs if you do not use the MicroPython Terminal in XCTU.

Example: turn on an LED

1. Note the **DS4** LED on the XBIB board. The following image highlights it in a red box. The LED is normally off.



2. At the MicroPython **>>>** prompt, type the commands below, pressing **Enter** after each one. After entering the last line of code, the LED illuminates. Anything after a **#** symbol is a comment, and you do not need to type it.

Note You can easily copy and paste code from the online version of this guide. Use caution with the PDF version, as it may not maintain essential indentations.

```
import machine
from machine import Pin
led = Pin("D4", Pin.OUT, value=0)  # Makes a pin object set to output 0.
# One might expect 0 to mean OFF and 1 to mean ON, and this is normally the
case.
# But the LED we are turning on and off is setup as what is# known as
"active low".
# This means setting the pin to 0 allows current to flow through the LED and
then through the pin, to ground.
```

3. To turn it off, type the following and press **Enter**:

```
led.value(1)
```

You have successfully controlled an LED on the board using basic I/O.

Example: code a request help button

This example provides a fast, deep dive into MicroPython designed to let you see some of the powerful things it can do with minimal code. It is not meant as a tutorial; for in-depth examples refer to the *Digi MicroPython Programming Guide*.

Many stores have help buttons in their aisles that a customer can press to alert the store staff that assistance is required in that aisle. You can implement this type of system using the Digi XBee Cellular Modem, and this example provides the building blocks for such a system. This example, based on SMS paging, can have many other uses such as alerting someone with a text to their phone if a water sensor in a building detects water on the floor, or if a temperature sensor reports a value that is too hot or cold relative to normal operation.

Enter MicroPython paste mode

In the following examples it is helpful to know that MicroPython supports paste mode, where you can copy a large block of code from this user guide and paste it instead of typing it character by character. To use paste mode:

1. Copy the code you want to run. For example, copy the following code that is the code from the LED example:

```
from machine import Pin
led = Pin("D4", Pin.OUT, value=0)
```

Note You can easily copy and paste code from the online version of this guide. Use caution with the PDF version, as it may not maintain essential indentations.

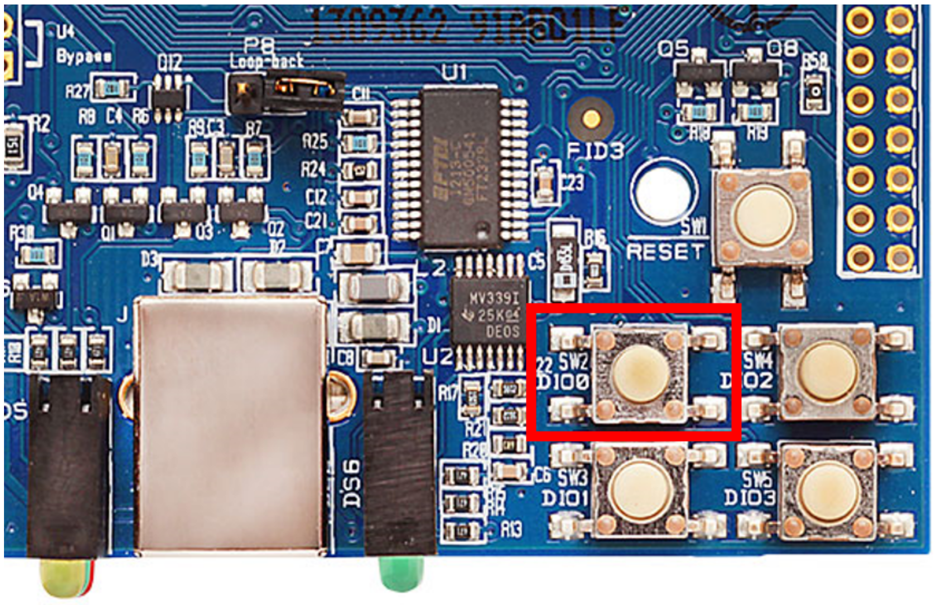
2. In the terminal, at the MicroPython **>>>** prompt type **Ctrl**+**E** to enter paste mode. The terminal displays **paste mode; Ctrl-C to cancel, Ctrl-D to finish**.
3. The code appears in the terminal occupying four lines, each line starts with its line number and three **=** symbols. For example line 1 starts with **1===**.
4. If the code is correct, press **Ctrl**+**D** to run the code and you should once again see the **DS4** LED turn on. If you get a **Line 1 SyntaxError: invalid syntax** error, see Syntax error at line 1.
   If you wish to exit paste mode without running the code, for example, or if the code did not copy correctly, press **Ctrl**+**C** to cancel and return to the normal MicroPython **>>>** prompt.
5. Next turn the LED off. Copy the code below:

```
from machine import Pin
led = Pin("D4", Pin.OUT, value=1)
print("DS4 LED now OFF!")
print("Paste Mode Successful!")
```

6. Press **Ctrl**+**E** to enter paste mode.
7. Press **Ctrl** + **Shift** + **V** or right-click in the Terminal and select **Paste** to paste the copied code.
8. If the code is correct, press **Ctrl**+**D** to run it. The LED should turn off and you should see two confirmation messages print to the screen.

Catch a button press

For this part of the example, you write code that responds to a button press on the XBIB-U-DEV board that comes with the XBee Cellular Modem Development Kit. The code monitors the pin connected to the button on the board labeled **SW2**.



On the board you see **DIO0** written below **SW2**, to the left of the button. This represents the pin that the button is connected to.

In MicroPython, you will create a pin object for the pin that is connected to the **SW2** button. When you create the pin object, the **DIO0** pin is called **D0** for short.

The loop continuously checks the value on that pin and once it goes to **0** (meaning the button has been pressed) a **print()** call prints the message **Button pressed!** to the screen.

At the MicroPython **>>>** prompt, copy the following code and enter it into MicroPython using paste mode (**Ctrl**+**E**), right-click in the Terminal, select **Paste** to paste the copied code, and press **Ctrl**+**D** to run the code.

```
# Import the Pin module from machine, for simpler syntax.
from machine import Pin


# Create a pin object for the pin that the button "SW2" is connected to.
dio0 = Pin("D0", Pin.IN, Pin.PULL_UP)
# Give feedback to inform user a button press is needed.
print("Waiting for SW2 press...")
# Create a WHILE loop that checks for a button press.
while (True):
    if (dio0.value() == 0):  # Once pressed.
        print("Button pressed!")  # Print message once pressed.
        break  # Exit the WHILE loop.

# When you press SW2, you should see "Button pressed!" printed to the
screen.
# You have successfully performed an action in response to a button press!
```

Note You can easily copy and paste code from the online version of this guide. Use caution with the PDF version, as it may not maintain essential indentations.

Note If you have problems pasting the code, see Syntax error at line 1. For SMS failures, see Error Failed to send SMS.

Send a text (SMS) when the button is pressed

After creating a while loop that checks for a button press, add sending an SMS to your code. Instead of printing **Button pressed!** to the screen, this code sends **Button pressed** to a cell phone as a text (SMS) message.

To accomplish this, use the **sms_send()** method, which sends a string to a given phone number. It takes the arguments in the following order:

1. **<phone number>**
2. **<message-to-be-sent>**

Before you run this part of the example, you must create a variable that holds the phone number of the cell phone or mobile device you want to receive the SMS.

1. To do this, at the MicroPython **>>>** prompt, type the following command, replacing **1123456789** with the full phone number (no dashes, spaces, or other symbols) and press **Enter**:

```
ph = 1123456789
```

2. After you create this **ph** variable with your phone number, copy the code below and enter it into MicroPython using paste mode (**Ctrl**+**E**) and then run it.

```
from machine import Pin
import network # Import network module
import time


c = network.Cellular() # initialize cellular network parameter
dio0 = Pin("D0", Pin.IN, Pin.PULL_UP)
while not c.isconnected():  # While no network connection.
    print("Waiting for connection to cell network...")
    time.sleep(5)
print("Connected.")
# Give feedback to inform user a button press is needed.
print("Waiting for SW2 press...")
while (True):
    if (dio0.value() == 0):
        # When SW2 is pressed, the module will send an SMS
        # message saying "Button pressed" to the given target cell phone
number.
        try:
            c.sms_send(ph, 'Button Pressed')
            print("Sent SMS successfully.")
        except OSError:
            print("ERROR- failed to send SMS.")
        # Exit the WHILE loop.
        break
```

Note You can easily copy and paste code from the online version of this guide. Use caution with the PDF version, as it may not maintain essential indentations.

Note If you have problems pasting the code, see Syntax error at line 1. For SMS failures, see Error Failed to send SMS.

Add the time the button was pressed

After you add the ability to send an SMS to the code, add functionality to insert the time at which the button was pressed into the SMS that is sent. To accomplish this:

1. Create a UDP socket with the **socket()** method.
2. Save the IP address and port of the time server in the **addr** variable.
3. Connect to the time server with the **connect()** method.
4. Send **hello** to the server to prompt it to respond with the current date and time.
5. Receive and store the date/time response in the **buf** variable.
6. Send an SMS in the same manner as before using the **sms_send()** method, except that you add the time into the SMS message, such that the message reads: **[Button pressed at: YYYY-MM-DD HH:MM:SS]**

To verify that your phone number is still in the memory, at the MicroPython **>>>** prompt, type **ph** and press **Enter**.

If MicroPython responds with your number, copy the following code and enter it into MicroPython using paste mode and then run it. If it returns an error, enter your number again as shown in Send a text (SMS) when the button is pressed. With your phone number in memory in the **ph** variable, copy the code below and enter it into MicroPython using paste mode (**Ctrl**+**E**) and then run it.

```
from machine import Pin
import network
import usocket
import time


c = network.Cellular()
dio0 = Pin("D0", Pin.IN, Pin.PULL_UP)
while not c.isconnected():  # While no network connection.
    print("Waiting for connection to cell network...")
    time.sleep(5)
print("Connected.")
# Give feedback to inform user a button press is needed.
print("Waiting for SW2 press...")
while (1):
    if (dio0.value() == 0):
        # When button pressed, now the module will send "Button Press" AND
        # the time at which it was pressed in an SMS message to the given
        # target cell phone number.
        socketObject = usocket.socket(usocket.AF_INET, usocket.SOCK_DGRAM)
        # Connect the socket object to the web server specified in
"address".
        addr = ("52.43.121.77", 10002)
        socketObject.connect(addr)
        bytessent = socketObject.send("hello")
        print("Sent %d bytes on socket" % bytessent)
```

```
                buf = socketObject.recv(1024)
                # Send message to the given number.  Handle error if it occurs.
                try:
                    c.sms_send(ph, 'Button Pressed at: ' + str(buf))
                    print("Sent SMS successfully.")
                except OSError:
                    print("ERROR- failed to send SMS.")
                # Exit the WHILE loop.
                break
```

Note You can easily copy and paste code from the online version of this guide. Use caution with the PDF version, as it may not maintain essential indentations.

Now you have a system based on the XBee Cellular Modem that sends an SMS in response to a certain input, in this case a simple button press.

Note If you have problems pasting the code, see Syntax error at line 1. For SMS failures, see Error Failed to send SMS.

Example: debug the secondary UART

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.

Exit MicroPython mode

To exit MicroPython mode:

1. In the XCTU MicroPython Terminal, click the green **Close** button .
2. Click **Close** at the bottom of the terminal to exit the terminal.
3. In XCTU's Configuration working mode , change **AP API Enable** to another mode and click the **Write** button . We recommend changing to Transparent mode **[0]**, as most of the examples use this mode.

Other terminal programs

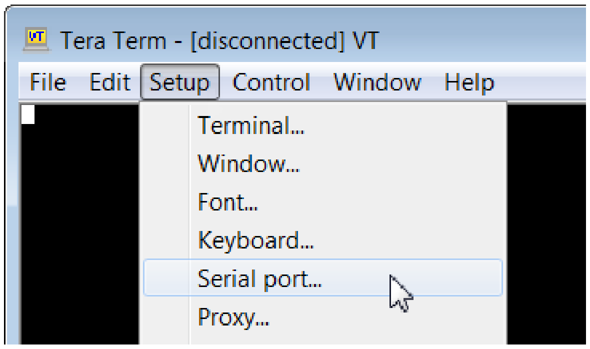
If you do not use the MicroPython Terminal in XCTU, you can use other terminal programs to communicate with the XBee Cellular Modem. If you use Microsoft Windows, follow the instructions for Tera Term, if you use Linux, follow the instructions for picocom. To download these programs:

- Tera Term for Windows; see https://ttssh2.osdn.jp/index.html.en.
- Picocom for Linux; see https://developer.ridgerun.com/wiki/index.php/Setting_up_Picocom_-_Ubuntu and for the source code and in-depth information https://github.com/npat-efault/picocom.

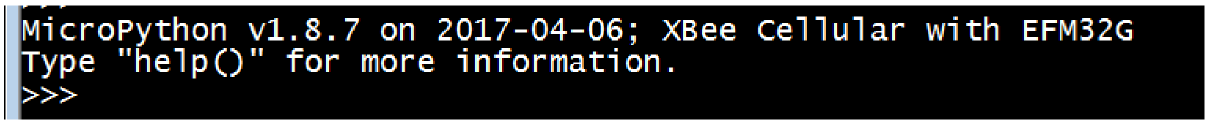
Tera Term for Windows

With the XBee Cellular Modem in MicroPython mode (**AP** = **4**), you can access the MicroPython prompt using a terminal.

1. Open Tera Term. The **Tera Term: New connection** window appears.
2. Click the **Serial** radio button to select a serial connection.
3. From the **Port:** drop-down menu, select the COM port that the XBee Cellular Modem is connected to.
4. Click **OK**. The **COMxx - Tera Term VT** terminal window appears and Tera Term attempts to connect to the device at a baud rate of 9600 b/s. The terminal will not allow communication with the device since the baud rate setting is incorrect. You must change this rate as it was previously set to 115200 b/s.
5. Click **Setup** and **Serial Port**. The **Tera Term: Serial port setup** window appears.


Tera Term - [disconnected] VT
File Edit Setup Control Window Help
Terminal...
Window...
Font...
Keyboard...
Serial port...
Proxy...

6. In the **Tera Term: Serial port setup** window, set the parameters to the following values:
   - **Port**: Shows the port that the XBee Cellular Modem is connected on.
   - **Baud rate**: 115200
   - **Data**: 8 bit
   - **Parity**: none
   - **Stop**: 1 bit
   - **Flow control**: hardware
   - **Transmit delay**: N/A
7. Click **OK** to apply the changes to the serial port settings. The settings should go into effect right away.
8. To verify that local echo is not enabled and that extra line-feeds are not enabled:
   a. In Tera Term, click **Setup** and select **Terminal**.
   b. In the **New-line** area of the **Tera Term: Serial port setup** window, click the **Receive** drop-down menu and select **CR** if it does not already show that value.
   c. Make sure the **Local echo** box is not checked.
9. Click **OK**.
10. Press **Ctrl**+**B** to get the MicroPython version banner and prompt.



```
MicroPython v1.8.7 on 2017-04-06; XBee Cellular with EFM32G
Type "help()" for more information.
>>>
```

Now you can type MicroPython commands at the **>>>** prompt.

Use picocom in Linux

With the XBee Cellular Modem in MicroPython mode (**AP** = **4**), you can access the MicroPython prompt using a terminal.

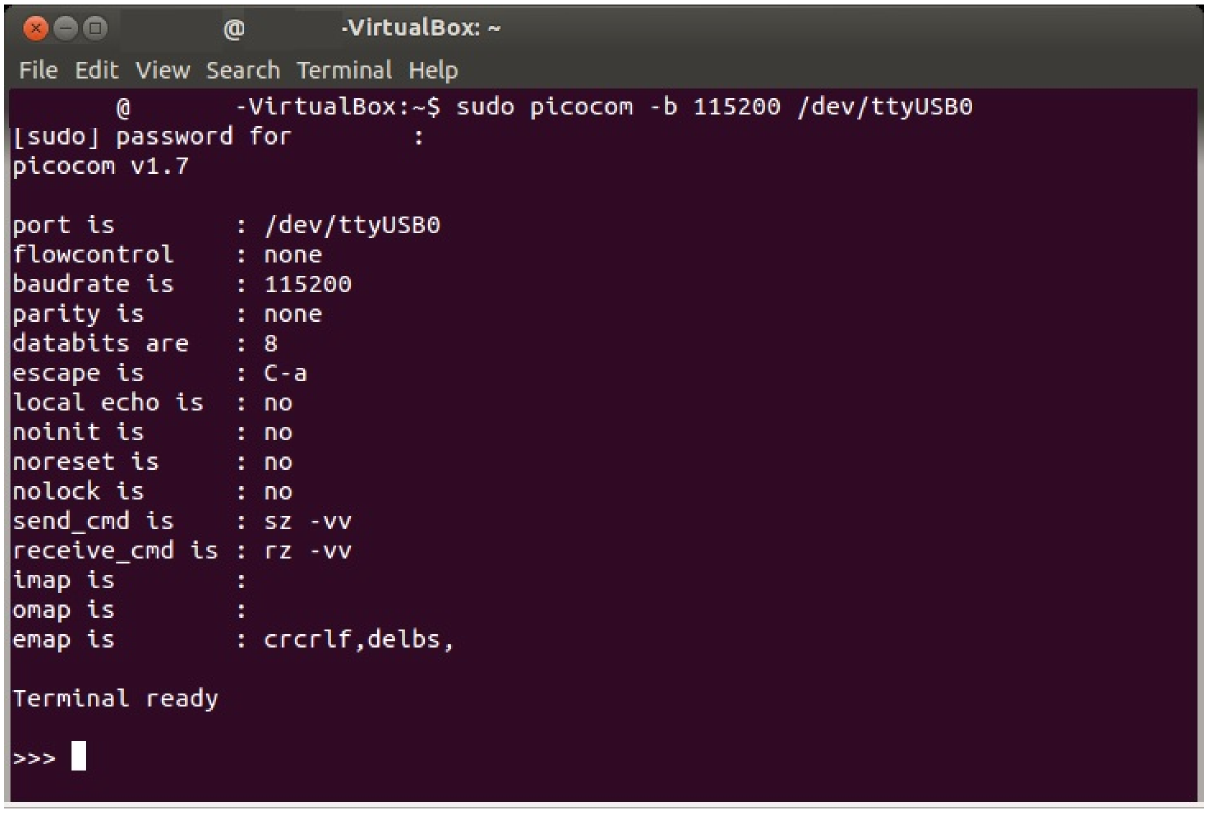
Note The user must have read and write permission for the serial port the XBee Cellular Modem is connected to in order to communicate with the device.

1. Open a terminal in Linux and type **picocom -b 115200 /dev/ttyUSB0**. This assumes you have no other USB-to-serial devices attached to the system.
2. Press **Ctrl**+**B** to get the MicroPython version banner and prompt. You can also press **Enter** to bring up the prompt.

If you do have other USB-to-serial devices attached:

1. Before attaching the XBee Cellular Modem, check the directory **/dev/** for any devices named **ttyUSBx**, where **x** is a number. An easy way to list these is to type: **ls /dev/ttyUSB***. This produces a list of any device with a name that starts with **ttyUSB**.
2. Take note of the devices present with that name, and then connect the XBee Cellular Modem.

3. Check the directory again and you should see one additional device, which is the XBee Cellular Modem.
4. In this case, replace **/dev/ttyUSB0** at the top with **/dev/ttyUSB<number>**, where **<number>** is the new number that appeared.
5. It should connect and show Terminal ready.


[redacted]@[redacted]-VirtualBox: ~
File Edit View Search Terminal Help

```
       @       -VirtualBox:~$ sudo picocom -b 115200 /dev/ttyUSB0
[sudo] password for        :
picocom v1.7

port is         : /dev/ttyUSB0
flowcontrol     : none
baudrate is     : 115200
parity is       : none
databits are    : 8
escape is       : C-a
local echo is   : no
noinit is       : no
noreset is      : no
nolock is       : no
send_cmd is     : sz -vv
receive_cmd is  : rz -vv
imap is         :
omap is         :
emap is         : crcrlf,delbs,

Terminal ready

>>> 
```

Now you can type MicroPython commands at the **>>>** prompt.

Get started with Digi Remote Manager

Digi Remote Manager® is a cloud-based device and data management platform that you can use to configure and update a device, and view and manage device data.

The sections below describe how to create a Remote Manager account, upgrading your device, configure your device, and manage data in Remote Manager.

1. Create a Remote Manager account and add devices
2. To ensure that all Remote Manager features are available, you should upgrade your device to the latest firmware. See Update the firmware from Remote Manager or Update the firmware using web services in Remote Manager.
3. Configure your device in Remote Manager

   To be able to configure your device in Remote Manager, the device must be connected to Remote Manager. You can connect to and configure your device in Remote Manager using one of the following methods:

   - **Scheduled connection**: In this method, you create a list of tasks that you want to perform on the device, and then start the operation. This is the recommended method, and is the best choice for low data usage. See Configure Remote Manager features by scheduling tasks.
   - **Always connected**: This method can be used for initial configuration, or when you are not concerned with low data usage. See Configure XBee settings within Remote Manager.
4. Secure the connection between an XBee and Remote Manager with server authentication.
5. Manage data in Remote Manager
6. Remote Manager reference

Create a Remote Manager account and add devices

To be able to use Remote Manager, you must create a Remote Manager account and add your XBee devices to the device list. You should also verify that the device is enabled to connect to Remote Manager.

1. Create a Remote Manager account.
2. Add an XBee Cellular Modem to Remote Manager.
3. Verify the connection between a device and Remote Manager

Create a Remote Manager account

Digi Remote Manager is an on-demand service with no infrastructure requirements. Remote devices and enterprise business applications connect to Remote Manager through standards-based web services. This section describes how to configure and manage an XBee using Remote Manager. For detailed information on using Remote Manager, refer to the *Remote Manager User Guide*, available via the **Documentation** tab in Remote Manager.

Before you can manage an XBee with Remote Manager, you must create a Remote Manager account. To create a Remote Manager account:

1. Go to https://www.digi.com/products/cloud/digi-remote-manager.
2. Click **30 DAY FREE TRIAL/LOGIN**.
3. Follow the online instructions to complete account registration. You can upgrade your Developer account to a paid account at any time.

When you are ready to deploy multiple XBee Cellular Modems in the field, upgrade your account to access additional Remote Manager features.

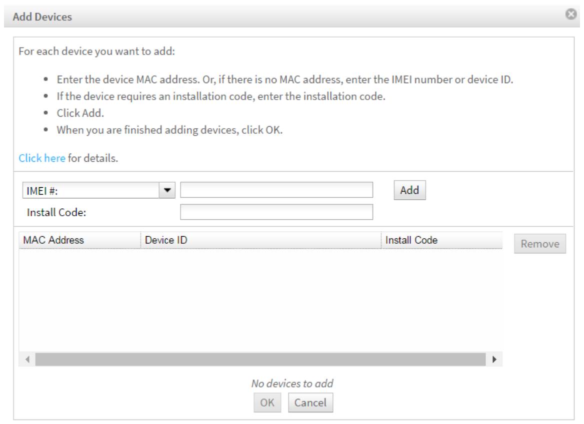
Add an XBee Cellular Modem to Remote Manager

Each XBee Cellular Modem must be added to the Remote Manager account inventory list.

Before adding an XBee to your Remote Manager account inventory, you need to determine the International Mobile Equipment Identity (IMEI) number for the device. Use XCTU to view the IMEI number by querying the **IM** parameter.

To add an XBee to your Remote Manager account inventory, follow these steps:

1. Log into Remote Manager.
2. Click **Device Management** > **Devices**.
3. Click **Add Devices**. The **Add Devices** dialog appears.
4. Select **IMEI #**, and type or paste the IMEI number of the XBee you want to add. The IM (IMEI) command provides this number.


Add Devices

For each device you want to add:

- Enter the device MAC address. Or, if there is no MAC address, enter the IMEI number or device ID.
- If the device requires an installation code, enter the installation code.
- Click Add.
- When you are finished adding devices, click OK.

Click here for details.

MAC Address	Device ID	Install Code

No devices to add

5. Click **Add** to add the device. The XBee is added to your inventory.
6. Click **OK** to close the **Add Devices** dialog and return to the **Devices** view.

Verify the connection between a device and Remote Manager

By default, the XBee is configured to enable a connection to Remote Manager. The connection between XBee and Remote Manager is maintained using periodic UDP operations.

You should verify the default settings to ensure that the connection will work as desired.

1. Launch XCTU.
2. Verify that the MO command is set to **6**, which is the default.
3. Configure the frequency of polls for Remote Manager activity using the DF command. The default is 1440 minutes (24 hours).
4. Enable the SM/UDP feature in Remote Manager for each device. See Enable SM/UDP.

Configure Remote Manager features by scheduling tasks

Remote Manager provides tools to perform common management and maintenance tasks on your XBee device. A Remote Manager task is a sequence of commands that can be performed on one or more XBee Cellular devices. Tasks can then be assigned to a schedule. When a scheduled task is run it becomes an active operation and can be monitored for status and completion.

Note You must upgrade your device to the latest firmware for this feature to be available. See Update the firmware from Remote Manager or Update the firmware using web services in Remote Manager.

Some typical examples of useful things that can be done with scheduled tasks include:

- Change configuration
- Update your MicroPython application and libraries to add features and capabilities
- Update your security certificates
- Perform a data service device request
- Send an SMS message to your device

Scheduled tasks can be created and performed through the following methods:

- Remote Manager **Schedules** user interface.
- Remote Manager **API Explorer** user interface
- Programming web service calls

Note For any of these methods to work properly, you must have SM/UDP enabled. See Enable SM/UDP.

Overview: Create a schedule for a set of tasks

When using the most current firmware version, the XBee Cellular devices are designed to poll Remote Manager once per day over the SM/UDP protocol to check for any active operations. In order to perform a set of tasks, the device needs to be told to connect to Remote Manager, perform the sequence of tasks, and then told to disconnect.

The following provides a template of how to create a schedule for an XBee to connect, perform a set of tasks and then disconnect:

1. Make sure that SM/UDP is enabled. See Enable SM/UDP.
2. Log into Remote Manager.

3. Click **Device Management > Schedules**.
4. Click **New Schedule**. The **New Schedule** page displays.

   Note The **Steps to schedule a task** wizard may display. Click the **x** in the upper left corner to close the wizard. See Schedule walk-through feature in the *Digi Remote Manager® User Guide* for more information.

5. In the **Description** field, enter a name for the schedule, such "Read Settings."
6. Add the following tasks:
   a. Click **SM/UDP > SM/UDP Request Connect**. A task is added to the dialog.
   b. Add other tasks as needed. For examples, refer to the Examples section.
   c. Click **Device > Disconnect**. A task is added to the dialog.
7. Click **Schedule** in the lower right corner of the dialog to schedule the tasks to run. The schedule screen displays.

   Note You can also click **Save as** to save this schedule for future use.

8. Select the device(s) on which you want to run this schedule. You can add more than one device.
9. Click **Run Now**.

Examples

The examples in the following sections assume you are using the Digi Remote Manager Schedule wizard. However, you should be aware that operations can be created and performed programmatically via web service calls or via the API explorer. The XML web service calls provide more options than are available in the GUI dashboard for some tasks.

Example: Read settings and state using Remote Manager

In order to configure devices you will need to know the structure of the XML for your XBee's settings. The easiest way to obtain this is to perform a `query_setting` RCI request against your device.

Note You must upgrade your device to the latest firmware for this feature to be available. See Update the firmware from Remote Manager or Update the firmware using web services in Remote Manager.

Note To obtain the state of the device, you can perform the same operations in the example below, but replace `query_setting` with `query_state`.

1. Log into Remote Manager.
2. Click **Device Management > Schedules**.
3. Click **New Schedule**. The **New Schedule** page displays.

   Note The **Steps to schedule a task** wizard may display. Click the **x** in the upper left corner to close the wizard. See Schedule walk-through feature in the *Digi Remote Manager® User Guide* for more information.

4. In the **Description** field, enter a name for the schedule, such "Read Settings."

5. Add the following tasks:
   a. Click **SM/UDP > SM/UPD Request Connect**. A task is added to the dialog.
   b. Click **Device > RCI Command**. A task is added to the dialog.

      Change the RCI command to the following:

      ```
      <rci_request>
          <query_setting/>
      </rci_request>
      ```

   c. Click **Device > Disconnect**. A task is added to the dialog.
6. Click **Schedule** in the lower right corner of the dialog to schedule the tasks to run. The schedule screen displays.

   Note You can also click **Save as** to save this schedule for future use.

7. Select the device(s) on which you want to run this schedule. You can add more than one device.
8. Click **Run Now**.
9. Click **Device Management > Operations** to view information about the operation. See Operations in the *Digi Remote Manager® User Guide* for more information about this page.

After your operation completes you can click **Response** to view the XML for all of the settings that your XBee reports. This XML structure has the same settings that you will use in the `set_setting` command to configure your XBee as shown in this example: Example: Configure a device from Remote Manager using XML.

Example: Configure a device from Remote Manager using XML

You can configure each XBee device from Remote Manager, using XML. The devices must be in the Remote Manager inventory device list and be active.

Note You must upgrade your device to the latest firmware for this feature to be available. See Update the firmware from Remote Manager or Update the firmware using web services in Remote Manager.

In this configuration example, you are changing the device to poll four times a day instead of just once. In this case, you should change the **DF** parameter to 360 minutes.

1. Log into Remote Manager.
2. Click **Device Management > Schedules**.
3. Click **New Schedule**. The **New Schedule** page displays.

   Note The **Steps to schedule a task** wizard may display. Click the **x** in the upper left corner to close the wizard. See Schedule walk-through feature in the *Digi Remote Manager® User Guide* for more information.

4. In the **Description** field, enter a name for the schedule, such as "Configure Reporting Frequency."

5. Add the following tasks:
   a. Click **SM/UDP > SM/UPD Request Connect**. A task is added to the dialog.
   b. Click **Device > RCI Command**. A task is added to the dialog.

      Change the RCI command to the following:

```
<rci_request>
    <set_setting>
        <remote_manager>
            <DF>360</DF>
        </remote_manager>
     </set_setting>
</rci_request>
```

   c. Click **Device > Disconnect**. A task is added to the dialog.
6. Click **Schedule** in the lower right corner of the dialog to schedule the tasks to run. The schedule screen displays.

   Note You can also click **Save as** to save this schedule for future use.

7. Select the device(s) on which you want to run this schedule. You can add more than one device.
8. Click **Run Now**.
9. Click **Device Management > Operations** to view information about the operation. See Operations in the *Digi Remote Manager® User Guide* for more information about this page.

Example: Update XBee firmware using Remote Manager

You can use a scheduled task to update the XBee Cellular firmware. Since the device is configured by default to poll Remote Manager once a day, you need to be able to set up a scheduled task to update the device's firmware to take advantage of new features and fixes. To update the firmware to a new version you will need to obtain the .ebin file for the new firmware from our support site. This file is one of the files in the .zip (for example, XBXC-31011.zip) archive that you can download for the product.

Note You must upgrade your device to the latest firmware for this feature to be available. See Update the firmware from Remote Manager or Update the firmware using web services in Remote Manager.

To upgrade using a scheduled task perform the following steps:

Step 1:

1. Download the updated firmware file for your device from Digi's support site.
   a. Go to the Digi XBee Cellular LTE CAT 1 support page.
   b. Scroll down to the **Firmware Updates** section.
   c. Locate and click **XBee Cellular LTE Cat 1 Verizon Firmware** to download the zip file.
   d. Unzip the file.

2. Log into Remote Manager.
3. Make sure that you have enabled SM/UDP. See Enable SM/UDP.
4. Click **Device Management > Schedules**.
5. Click **New Schedule**. The **New Schedule** page displays.

   Note The **Steps to schedule a task** wizard may display. Click the **x** in the upper left corner to close the wizard. See Schedule walk-through feature in the *Digi Remote Manager® User Guide* for more information.

6. In the **Description** field, enter a name for the schedule, such as "Update XBee Firmware."
7. Add the following tasks:
   a. Click **SM/UDP > SM/UDP Request Connect**. A task is added to the dialog.
   b. Click **Device > Gateway Firmware Update**.
   c. Click **Browse** and select the .ebin file (for example, XBXC-1011.ebin) for the new firmware to update.
   d. Click **Device > Disconnect**. A task is added to the dialog.
8. Click **Schedule** in the lower right corner of the dialog to schedule the tasks to run. The schedule screen displays.

   Note You can also click **Save as** to save this schedule for future use.

9. Select the device(s) on which you want to run this schedule. You can add more than one device.
10. Click **Run Now**.
11. Click **Device Management > Operations** to view information about the operation. See Operations in the *Digi Remote Manager® User Guide* for more information about this page.

Example: Update MicroPython from Remote Manager using XML

You can use the API Explorer in Remote Manager to create a schedule that enables you to update the MicroPython application. In this example, you want to add FTP client capability to the MicroPython application. You will need to add the library *uftp.py* and then update the *main.py* application.

This example is done following these steps: upload the MicroPython files to Remote Manager, create an XML file with the tasks that you want to perform, upload the XML file, and then schedule an operation to upload the files onto your device.

Note You must upgrade your device to the latest firmware for this feature to be available. See Update the firmware from Remote Manager or Update the firmware using web services in Remote Manager.

Step 1: Upload the MicroPython files

1. Log into Remote Manager.
2. Click **Data Services > Data Files**.

3. Upload the MicroPython application *main.py* file.
    a. Click **New Folder**. The **New Folder** dialog displays.
    b. In the **Folder name** field, enter a descriptive name, such as "MicroPython."
    c. Click **Create**. The new file is added to the list of files.
    d. Find the "MicroPython" folder in the folder list.
    e. Click **Upload Files**. The **Upload Files** dialog displays.
    f. Browse for the *main.py* file. Check with your system administrator for the location of the application file.
    g. Click **OK**.
4. Upload the MicroPython library *uftp.py* file.
    a. Find the "MicroPython" folder in the folder list.
    b. Click **Upload Files**. The **Upload Files** dialog displays.
    c. Browse for the *uftp.py* file. The library *uftp.py* file is found on the GitHub repository: https://github.com/digidotcom/xbee-micropython
    d. Click **OK**.

Step 2: Create an XML file with the tasks that you want to perform

This XML file will contain a list of commands for the operation that you will schedule in Step 3.

Note The RCI commands to *set_settings* in the task may fail to execute because of disconnects after changing the value for **MO**.

1. Open the editor of your choice.
2. Create a new file named "updatemicropython.xml."
3. Copy the XML below and paste it into the new file.
4. Save the file.

```
<task>
   <description>Update MicroPython</description>
   <command>
      <name>SM/UDP Request Connect</name>
      <event>
         <on_error>
            <end_task/>
         </on_error>
      </event>
      <sci>
         <send_message reply="none"  >
            <sm_udp>
               <request_connect/>
            </sm_udp>
         </send_message>
      </sci>
   </command>
   <command>
      <name>RCI Command</name>
      <event>
```

```
         <on_error>
            <continue/>
         </on_error>
      </event>
      <sci>
         <send_message cache="false" allowOffline="true"  >
            <!-- Disable Python Auto-start and enable TCP connection for
remainder of commands-->
            <rci_request>
               <set_setting>
                  <micropython>
                    <PS>0</PS>
                  </micropython>
                  <remote_manager>
                    <MO>7</MO>
                  </remote_manager>
               </set_setting>
             </rci_request>
         </send_message>
      </sci>
   </command>
   <command>
          <!-- Reboot to stop MicroPython -->
       <name>Reboot</name>
       <event>
         <on_error>
            <continue/>
         </on_error>
       </event>
       <sci>
          <reboot allowOffline="true" waitForReconnect="true"/>
       </sci>
   </command>
      <!-- Update MicroPython application-->
   <command>
      <name>Upload Files</name>
      <event>
         <on_error>
            <continue/>
         </on_error>
      </event>
      <sci>
         <file_system  allowOffline="true" >
            <commands>
               <put_file path="/flash/main.py">
                  <file>~/MicroPython/main.py</file>
               </put_file>
            </commands>
         </file_system>
      </sci>
   </command>
   <command>
      <name>Upload Files</name>
      <event>
         <on_error>
            <continue/>
         </on_error>
      </event>
      <sci>
         <file_system  allowOffline="true" >
```

```
            <commands>
               <put_file path="/flash/lib/uftp.py">
                  <file>~/MicroPython/uftp.py</file>
               </put_file>
            </commands>
         </file_system>
      </sci>
   </command>
   <command>
      <name>RCI Command</name>
      <event>
         <on_error>
            <continue/>
         </on_error>
      </event>
      <sci>
         <send_message cache="false" allowOffline="true">
               <!-- Enable Python Auto-start -->
            <rci_request>
               <set_setting>
                  <micropython>
                     <PS>1</PS>
                  </micropython>
                  <remote_manager>
                     <MO>6</MO>
                  </remote_manager>
               </set_setting>
            </rci_request>
         </send_message>
      </sci>
   </command>
         <!-- Reboot to start the program -->
   <command>
      <name>Reboot</name>
         <event>
            <on_error>
               <end_task/>
            </on_error>
         </event>
         <sci>
            <reboot  allowOffline="true"  waitForReconnect="false"/>
         </sci>
   </command>
</task>
```

Step 3: Upload the XML to Remote Manager

In this step you will upload the file you just created (*updatemicropython.xml*) to Remote Manager.

1. Log into Remote Manager.
2. Click **Data Services > Data Files**.

3. Upload the XML file you just created: *updatemicropython.xml*
   a. Find the "~/my_tasks" folder in the folder list.
   b. Click **Upload Files**. The **Upload Files** dialog displays.
   c. Browse for the *updatemicropython.xml* file.
   d. Click **OK**.

Step 4: Schedule an operation to upload the files

1. Log into Remote Manager.
2. Click **Documentation > API Explorer**.
3. Click **SCI Targets**. The **Select devices to be used in examples** dialog appears.
   a. From the **Add Targets** list box, search for the IMEI (device ID) of the device that you want to update.
   b. Click **Add**. The device is added to the device list.
   c. Click **OK**.
4. Click the **Examples** drop-down list button.
5. Click **Scheduled Operation > Create immediate running schedule**.
6. Update the XML to refer to the *updatemicropython.xml* file you created previously.

```
<!-- Runs immediately -->
<Schedule on="IMMEDIATE">
   <targets>
   <device id="00010000-00000000-03588320-70372440"/>
   </targets>

   <task path="~/my_tasks/updatemicropython.xml"/>
</Schedule>
```

7. Click **Send** to schedule the task.
8. Click **Device Management > Operations** to view information about the operation. See Operations in the *Digi Remote Manager® User Guide* for more information about this page.

Restore persistent connection to a remote XBee

The default connectivity to Remote Manager in the most recent firmware polls once a day using SM/UDP, which means that your XBee will always appear in a disconnected state and will use significantly less data.

If needed, you can restore the default connectivity to use the former behavior, where the device is continually connected using TCP. To do this, you will need to set bit 0 of the **MO** setting. The suggested value is **7** to connect securely over TLS, or you can use **1** for no security, which is the legacy value.

You can make the change using one of the following methods:

- **Local access**: If you have local access to the device you can use XCTU to change the **MO** setting back to the former default value.
- **Remote access**: If you only have remote access to your XBee you can change the device to maintain a persistent connection to Remote Manager. To do this you can set up a scheduled operation in Remote Manger for your device, as shown below.

Note You must upgrade your device to the latest firmware for this feature to be available. See Update the firmware from Remote Manager or Update the firmware using web services in Remote Manager.

To set up a scheduled operation to maintain a persistent connection:

1. Log into Remote Manager.
2. Make sure that you have enabled SM/UDP. See Enable SM/UDP.
3. Click **Device Management > Schedules**.
4. Click **New Schedule**. The **New Schedule** page displays.

   Note The **Steps to schedule a task** wizard may display. Click the **x** in the upper left corner to close the wizard. See Schedule walk-through feature in the *Digi Remote Manager® User Guide* for more information.

5. In the **Description** field, enter a name for the schedule, such as "Restore Persistent."
6. Add the following tasks:
   a. Click **SM/UDP > SM/UPD Request Connect**. A task is added to the dialog.
   b. Click **Device > RCI Command**. A task is added to the dialog.

   Change the RCI command to the following:

```
<rci_request>
    <set_setting>
        <remote_manager>
<MO>7</MO>
        </remote_manager>
     </set_setting>
</rci_request>
```

7. Click **Schedule** in the lower right corner of the dialog to schedule the tasks to run. The schedule screen displays.

   Note You can also click **Save as** to save this schedule for future use. The XML for your task is saved in the **~\my_tasks** directory on **Data Services > Data Files** in Remote Manager.

8. Select the device(s) on which you want to run this schedule. You can add more than one device.
9. Click **Run Now**. Within the next 24 hours, which is the default polling period for querying Remote Manager, your device will connect and will remain connected, as specified by the change to the **MO** setting.

10. Click **Device Management > Operations** to view information about the operation. See Operations in the *Digi Remote Manager® User Guide* for more information about this page.

Manage data in Remote Manager

You can view and manage XBee data in Remote Manager.

Review device status information from Remote Manager

You can view address, BLE, cellular, firmware, and I/O sampling status information for a XBee device in Remote Manager. The device must be in the Remote Manager inventory device list and be active.

Note You must upgrade your device to the latest firmware for this feature to be available. See Update the firmware from Remote Manager or Update the firmware using web services in Remote Manager.

1. Set up a persistent connection to connect the device to Remote Manager using one of the following methods:
   - **Remote Manager**: A persistent connection can be set up in Remote Manager. This option should be used when you have many deployed devices and no local access. See Restore persistent connection to a remote XBee.
   - **XCTU**: This option allows immediate access, and should be used when you have local access, such as when using a development kit or in a lab environment.
2. Log into Remote Manager.
3. Click **Device Management > Devices**.
4. Select the device that you want to configure.
5. Click **Properties** in the toolbar. As an alternative, click **Properties > Edit Device Configuration**. The configuration **Home** page appears.
6. Click **Status** in the toolbar to display the status sub-menus.
7. Click on the status group that has information you want to display. The status information is related to AT commands. For information about each AT command in the categories, click on the appropriate link below.
   - Addressing
   - Cellular
   - Firmware Version/Information
   - I/O
8. Click **Home** to return to the configuration **Home** page.
9. When all changes are complete, disconnect the device from Remote Manager.

Update the firmware from Remote Manager

XBee Cellular Modem supports Remote Manager firmware updates.

If you are upgrading the device firmware to a version listed below, your connection will disconnect and by default, your device will query Remote Manager only once a day. If you wish to restore the persistent connection behavior that was the default in prior firmware versions, see Restore persistent connection to a remote XBee.

After you have upgraded to the new firmware version, it is recommended that you keep the polling frequency low to reduce data usage. In order to upgrade firmware in the future, refer to Example: Update XBee firmware using Remote Manager.

Module	Upgrade firmware version
XBee CAT 1 Verizon	1011
XBee 3G	11311
XBee3 LTE-M	11411
XBee3 CAT 1	31011

To perform a firmware update:

1. Download the updated firmware file for your device from Digi's support site.
    a. Go to the Digi XBee Cellular LTE CAT 1 support page.
    b. Scroll down to the **Firmware Updates** section.
    c. Locate and click **XBee Cellular LTE Cat 1 Verizon Firmware** to download the zip file.
    d. Unzip the file.
2. Set up a persistent connection to connect the device to Remote Manager using one of the following methods:
    - **Remote Manager**: A persistent connection can be set up in Remote Manager. This option should be used when you have many deployed devices and no local access. See Restore persistent connection to a remote XBee.
    - **XCTU**: This option allows immediate access, and should be used when you have local access, such as when using a development kit or in a lab environment.
3. Log into Remote Manager.
4. In your Remote Manager account, click **Device Management > Devices**.
5. Select the first device you want to update.
6. To select multiple devices (must be of the same type), press the Control key and select additional devices.
7. Click **More** in the Devices toolbar and select **Update Firmware** from the Update category of the More menu. The **Update Firmware** dialog appears.
8. Click **Browse** to select the downloaded .ebin file that you unzipped earlier.
9. Click **Update Firmware**. The updated devices automatically reboot when the updates are complete.

    Note The update is immediately rejected and an error is returned if the device is going into sleep mode or is being shut down. See Clean shutdown.

10. When all changes are complete, disconnect the device from Remote Manager.

Update the cellular component firmware using Remote Manager

You can update the firmware for a device's cellular component using Remote Manager.

Prerequisites

- Remote Manager account created and an XBee cellular device added.
- XBee cellular device must be connected to Remote Manager to initiate update.
- The device ID of the XBee cellular device that you want to update.

Determine the update location

You must first determine the location of the firmware version to which you want to update. Digi provides updates by hosting them on an FTP server: **ftp1.digi.com**. If the FTP location is not accessible to your XBee Cellular, such as if you are using a VPN, the files may be retrieved and hosted separately on a server that it can reach.

Firmware is provided in the form of delta images which will migrate the cellular component from a known source to a given target version. You can verify the firmware version level of the cellular component using the MV (Modem Version) AT command. Check documentation and release notes for your XBee Cellular variant to determine the necessary upgrade path for your product.

You will need:

- The FTP hostname or IP address, which for Digi hosted files is: **ftp1.digi.com**
- The port running the FTP server, which is typically 21
- Username. For **ftp1.digi.com**, use: anonymous
- Password. For **ftp1.digi.com**, use your email address.
- Directory path containing update file.
- Update image filename.

Form the update request

A request to perform an update is communicated to the XBee Cellular through Remote Manager by using the Data Services Device Request feature. The device request should be sent to the **FTP_OTA** target and the payload of the request is the concatenation of the six fields identifying the full FTP location of the update file using the NUL byte as a delimiter. We recommend using the base64 encoded binary transport option to avoid issues representing the request in XML.

For example, you want to update a module with the file **sample.bin** in the **support/example** directory on Digi's FTP server.

The full body of the request:

ftp1.digi.com␀21␀anonymous␀example@digi.com␀support/example␀sample.bin

Note The ␀ character represents a byte in the date with the value zero.

The base64 encoded representation of the payload in turn:

ZnRwMS5kaWdpLmNvbQAyMQBhbm9ueW1vdXMAZXhhbXBsZUBkaWdpLmNvbQBzdXBwb3J0L2
V4YW1wbGUAc2FtcGxlLmJpbg==

The full Remote Manager device request is as shown below. Make sure to replace the **Device ID** attribute with the ID for your device.

```
<sci_request version="1.0">
  <data_service>
    <targets>
      <device id="Your device ID here"/>
    </targets>
    <requests>
      <device_request target_name="FTP_OTA" format="base64">

ZnRwMS5kaWdpLmNvbQAyMQBhbm9ueW1vdXMAZXhhbXBsZUBkaWdpLmNvbQBzdXBwb3J0L2V4YW1wbGUAc
2FtcGxlLmJpbg==
      </device_request>
    </requests>
  </data_service>
</sci_request>
```

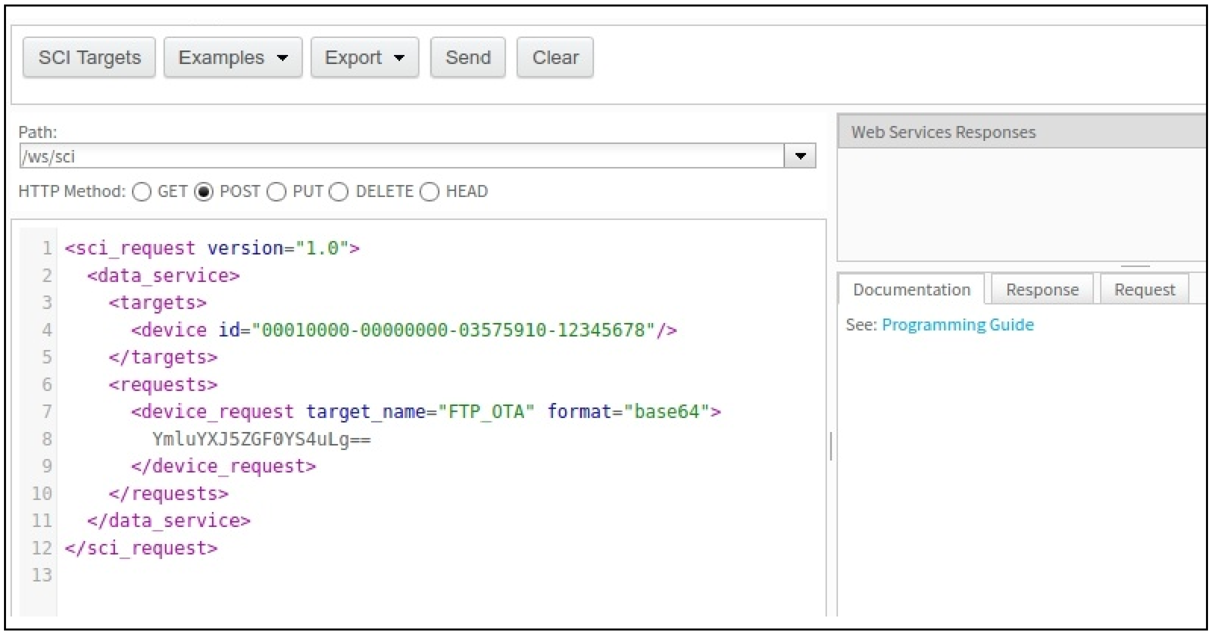
Perform the update

Once the update details have been established and the device request body written, the update is performed by doing an HTTP **POST** operation to the **/ws/sci** API endpoint of Remote Manager.

You can do this manually from the Remote Manager API Explorer.

1. Log into Remote Manager.
2. Select **Documentation** > **API Explorer**. The **API Explorer** page appears.
3. In the **Path** field, select or type: **/ws/sci**
4. Select the **POST** HTTP method option.
5. Copy the full Remote Manager device request you created in the previous step: Form the update request.

6. Paste the copied SCI request into the window below the HTTP Method selection section.



Path: /ws/sci

HTTP Method: GET [✓ POST] PUT DELETE HEAD

```
<sci_request version="1.0">
  <data_service>
    <targets>
      <device id="00010000-00000000-03575910-12345678"/>
    </targets>
    <requests>
      <device_request target_name="FTP_OTA" format="base64">
        YmluYXJ5ZGF0YS4uLg==
      </device_request>
    </requests>
  </data_service>
</sci_request>
```

7. Click **Send** to initiate the update.

Note Do not be alarmed if Remote Manager indicates that the device has disconnected. This is normal, as performing the update requires a reboot, and the network connection is temporarily disconnected during the reboot.

Validate the update

After the update has been triggered, it may take up to 30 minutes for the update to be applied and for the module to be connected to the network once more. If the XBee is not configured to automatically connect to [[[Undefined variable Digi_Standard_Variables.DigiRemoteManager_product_name]]], you will need to reconnect to Remote Manager to perform validation.

You can check that the update process has succeeded by reading the MV parameter value. After the update is complete, the version should reflect the desired target version.

Update the firmware using web services in Remote Manager

Remote Manager supports both synchronous and asynchronous firmware update using web services. The following examples show how to perform an asynchronous firmware update. See the Remote Manager documentation for more details on firmware updates.

Note You must use XCTU to update the cellular component's firmware.

1. Download the updated firmware file for your device from Digi's support site.
   a. Go to the Digi XBee Cellular LTE CAT 1 support page.
   b. Scroll down to the **Firmware Updates** section.
   c. Locate and click **XBee Cellular LTE Cat 1 Verizon Firmware** to download the zip

file.

d. Unzip the file.

2. Unzip the file and locate the .ebin file inside the unzipped directory.
3. Send an HTTP SCI request to Remote Manager with the contents of the downloaded file converted to base64 data; see the following examples:

   Examples for .ebin:

   - Example: update the XBee .ebin firmware synchronously with Python 3.0
   - Example: use the device's .ebin firmware image to update the XBee firmware synchronously

Example: update the XBee .ebin firmware synchronously with Python 3.0

```
import base64
import requests

# Location of firmware image
firmware_path = 'XBXC.ebin'

# Remote Manager device ID of the device being updated
device_id = '00010000-00000000-03526130-70153378'

# Remote Manager username and password
username = "my_Remote_manager_username"
password = "my_remote_manager_password"

url = 'https://remotemanager.digi.com/ws/sci'

# Get firmware image
fw_file = open(firmware_path, 'rb')
fw_data = fw_file.read()
fw_data = base64.encodebytes(fw_data).decode('utf-8')

# Form update_firmware request
data = """
<sci_request version="1.0">
  <update_firmware filename="firmware.ebin">
    <targets>
      <device id="{}"/>
    </targets>
    <data>{}</data>
  </update_firmware>
</sci_request>
""".format(device_id, fw_data)

# Post request
r = requests.post(url, auth=(username, password), data=data)
if (r.status_code != 200) or ("error" in r.content.decode('utf-8')):
    print("firmware update failed")
else:
    print("firmware update success")
```

Example: use the device's .ebin firmware image to update the XBee firmware synchronously

To update the XBee firmware synchronously with Python 3.0, but using the device firmware image already uploaded to Remote Manager, upload the device's *.ebin firmware to Remote Manager:

1. Download the updated firmware file for your device from Digi's support site. This is a zip file containing .ebin and .mxi files for import.
2. Unzip the file and locate the .ebin inside the unzipped directory.
3. Log in to Remote Manager.
4. Click the **Data Services** tab.
5. Click **Data Files**.
6. Click **Upload Files**; browse and select the *.ebin firmware file to upload it.
7. Send an HTTP SCI request to Remote manager with the path of the .ebin file; see the example below.

```
import base64
import requests

# Location of firmware image on Remote Manager
firmware_path = '~/XBXC.ebin'

# Remote Manager device ID of the device being updated
device_id = '00010000-00000000-03526130-70153378'

# Remote Manager username and password
username = "my_remote_manager_username"
password = "my_remote_manager_password"

url = 'https://remotemanager.digi.com/ws/sci'

# Form update_firmware request
data = """
<sci_request version="1.0">
  <update_firmware filename="firmware.ebin">
    <targets>
      <device id="{}"/>
    </targets>
    <file>{}</file>
  </update_firmware>
</sci_request>
""".format(device_id, firmware_path)

# Post request
r = requests.post(url, auth=(username, password), data=data)
if (r.status_code != 200) or ("error" in r.content.decode('utf-8')):
    print("firmware update failed")
else:
    print("firmware update success")
```

Manage secure files in Remote Manager

You can interact with files on the XBee device from Remote Manager, using either the SCI (Server command interface) or in the **File Management** view.

You can securely upload files by appending a hash sign (#) to the end of the file name. After the upload, the hash sign (#) is not retained as part of the file name. For example, you could upload a file named *my-cert.crt* appended with a hash sign (#): *my-cert.crt#*. After the upload is complete, the file is named *my-cert.crt*.

Note Uploading secure files in Remote Manager has the same result as doing an ATFS XPUT locally. See Secure files for more information.

SCI (Server command interface)

You can use the SCI (Server command interface) `file_system` command to securely upload a file. For more information, see the file_system section in the *Digi Remote Manager Programming Guide*.

File Management view

You can upload and manage files in the Remote Manager **File Management** view.

1. Prepare the file that you want to upload.
   a. Find the file on your hard drive.
   b. Rename the file and append a hash sign (#) to the end of the file name.
2. Set up a persistent connection to connect the device to Remote Manager using one of the following methods:
   - **Remote Manager**: A persistent connection can be set up in Remote Manager. This option should be used when you have many deployed devices and no local access. See Restore persistent connection to a remote XBee.
   - **XCTU**: This option allows immediate access, and should be used when you have local access, such as when using a development kit or in a lab environment.
3. Log into Remote Manager.
4. Click **Device Management > Devices**.
5. Select the device that you want to configure.
6. Click **Properties** in the toolbar. As an alternative, double-click on the device name. The **Properties** page appears.
7. Click **File Management**. The **File Management** view appears.
8. Click the upload icon. The **Upload File** dialog appears.
   a. Click **Browse** to browse for the file you want to upload. The selected file displays in the **File** field. Make sure that the file name is appended by a hash sign (#).
   b. Click **OK**. The uploaded file displays in the **File Management** view. Note that the file name is no longer appended by a hash sign (#).
9. When all changes are complete, disconnect the device from Remote Manager.

Remote Manager reference

Enable SM/UDP

You can use the SM/UDP feature to leverage the very small data footprint of Remote Manager SM protocol over UDP.

1. Log into Remote Manager.
2. Click **Device Management > Devices**.
3. Select the device that you want to configure.
4. Click **More > SM/UDP > Configure**. The **SM/UDP** dialog appears.
5. Verify that the **Battery Operated Mode** is not selected.

   This mode is not supported with Remote Manager and if enabled, the connectivity between XBee and Remote Manager may not work as expected.
6. Select **SM/UDP Service Enabled** to enable SM/UDP.
7. Click **Save**.

Note Do not enable battery-operated mode. This mode is not supported with Remote Manager and if enabled, the connectivity between XBee and Remote Manager may not work as expected.

Disconnect

The TCP connection remains open and periodic polling occurs until you manually disconnect the TCP connection. After you have disconnected the TCP connection, Remote Manager is no longer updated.

You can disconnect the TCP connection using either of the following methods:

- From the **Devices** page in Remote Manager: See Disconnect a device in the *Digi Remote Manager® User Guide*.
- Using web services in Remote Manager: See Request connect SM/UDP support in the *Digi Remote Manager® Programming Guide*.

Configure XBee settings within Remote Manager

You can configure the device settings to use features with Remote Manager. For more information, see Example: Read settings and state using Remote Manager.

Configure device settings in Remote Manager

You can configure each XBee device from Remote Manager. The devices must be in the Remote Manager inventory device list and be active.

1. Set up a persistent connection to connect the device to Remote Manager using one of the following methods:
   - **Remote Manager**: A persistent connection can be set up in Remote Manager. This option should be used when you have many deployed devices and no local access. See Restore persistent connection to a remote XBee.

- **XCTU**: This option allows immediate access, and should be used when you have local access, such as when using a development kit or in a lab environment.

2. Log into Remote Manager.
3. Click **Device Management > Devices**.
4. Select the device that you want to configure.
5. Click **Properties** in the toolbar. As an alternative, click **Properties > Edit Device Configuration**. The configuration **Home** page appears.
6. Click **Config** in the toolbar to display the settings sub-menus.
7. Click on the settings category that you want to configure. The settings in that category appear.
8. Make the desired configuration changes. See AT commands for information about each setting in the categories.
9. As you finish configuring in each setting category, click **Apply** to save the changes. If the changes are valid, Remote Manager writes them to non-volatile memory and applies them.
10. When all changes are complete, disconnect the device from Remote Manager.

Configure Remote Manager keepalive interval

Managing the data usage and the keepalive interval is important if you have the MO (Remote Manager Options) command bit 0 set to 1 or if you have enabled the Request connect feature in Remote Manager.

Digi Remote Manager is enabled on the XBee by default and has a 60 second keepalive interval, which can result in excessive cellular data usage, depending on your plan. The K1 and K2 commands can be used to tune the keepalive interval. Your carrier will disconnect an inactive socket automatically if there is no activity, so you need to tune this value based on your carrier’s disconnect timeout.

You can further reduce your data usage by periodically duty cycling your Remote Manager connection, either from MicroPython or your host processor. For example, you could enable the Remote Manager connection for 2 hours a day and then disable the connection for 22 hours. Your host processor or MicroPython program would need to keep track of the time to ensure the time interval.

Technical specifications

Interface and hardware specifications

The following table provides the interface and hardware specifications for the device.

Specification	Value
Dimensions	2.438 x 3.294 cm (0.960 x 1.297 in)
Weight	5 g (0.18 oz)
Operating temperature	-40 to +80 °C
Antenna connector	U.FL for primary and secondary antennas
Digital I/O	13 I/O lines
ADC	4 12-bit analog inputs

RF characteristics

The following table provides the RF characteristics for the device.

Specification	Value
Modulation	LTE/4G – QPSK, 16 QAM
Transmit power	23 dBm
Receive sensitivity	-102 dBm
Over-the-air maximum data rate	10 Mb/s

Networking specifications

The following table provides the networking and carrier specifications for the device.

Specification	Value
Addressing options	TCP/IP and SMS
Carrier and technology	Verizon 4G LTE Cat 1
Supported bands	4 and 13
Security	TLS

Power requirements

The following table provides the power requirements for the device.

Specification	Value
Supply voltage	3.0 to 5.5 V
Extended voltage range	2.7 to 5.5 VDC

Power consumption

The peak current was measured from multiple tested units.

Specification	State	Average current	Measured peak current
Tx+RX current	Active transmit, 23 dBm @ 3.3 V	860 mA	1020 mA
Tx+RX current	Active transmit, 23 dBm @ 5.0 V	555 mA	630 mA
TX Only current	Active transmit, 23 dBm @ 3.3 V	680 mA	N/A
Rx + ACK current	Active receive @ 3.3 V	530 mA	N/A
Rx + ACK current	Active receive @ 5 V	360 mA	N/A
RX Only current	Active receive @ 3.3 V	300 mA	N/A
Idle current	Idle/connected, listening @ 3.3 V	143 mA	N/A
Idle current	Idle/connected, listening @ 5 V	100 mA	N/A
Sleep current	Not connected, Deep Sleep @ 3.3 V	10 µA	N/A

Electrical specifications

The following table provides the electrical specifications for the XBee Cellular Modem.

Symbol	Parameter	Condition	Min	Typical	Max	Units
VCCMAX	Maximum limits of VCC line		0		5.5	V
VDD_IO	Internal supply voltage for I/O	While in deep sleep and during initial power up	Min (VCC-0.3, 3.3)		3.3	V
VDD_IO	Internal supply voltage for I/O	In normal running mode		3.3 V		V
VI	Voltage on any pin		-0.3		VDD_IO + 0.3	V
VIL	Input low voltage				0.3*VDD_IO	V
VIH	Input high voltage		0.7*VDD_IO			V
VOL	Voltage output low	Sinking 6 mA VDD_IO = 3.3 V			0.2*VDD_IO	V
VOH	Voltage output high	Sourcing 6 mA VDD_IO = 3.3 V	0.75*VDD_IO			V

Symbol	Parameter	Condition	Min	Typical	Max	Units
I_IN	Input leakage current	High Z state I/O connected to Ground or VDD_IO		0.1	100	nA
RPU	Internal pull-up resistor	Enabled		40		kΩ
RPD	Internal pull-down resistor	Enabled		40		kΩ

Regulatory approvals

The following table provides the regulatory and carrier approvals for the device.

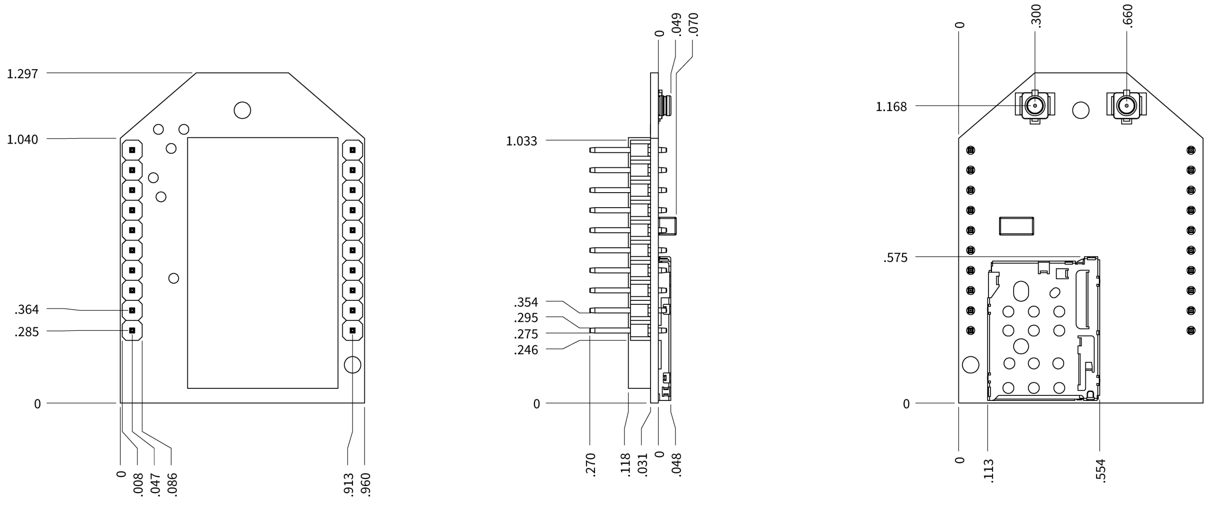
Note The contains statement of FCC and IC IDs listed on the customer labels must match the ID visible on the XBee device that is installed.

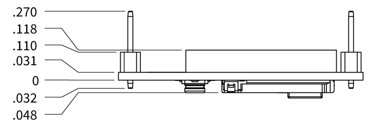
Specification	Value	Value
Model	XBCEL	XBCEL
Revision	XBC-V1-UT-001 version M and prior XBC-V1-UT-102 version F and prior	XBC-V1-UT-001 version N and later XBC-V1-UT-102 version G and later
United States	Contains FCC ID: RI7LE866SV1	Contains FCC ID: RI7LE866SV1A
Innovation, Science and Economic Development Canada (ISED)	Contains IC: 5131A-LE866SV1	Contains IC: 5131A-LE866SV1A
Europe (CE)	N/A	N/A
RoHS	Lead-free and RoHS compliant	Lead-free and RoHS compliant
Australia	N/A	N/A
Verizon end-device certified	Yes	Yes

Hardware

Mechanical drawings

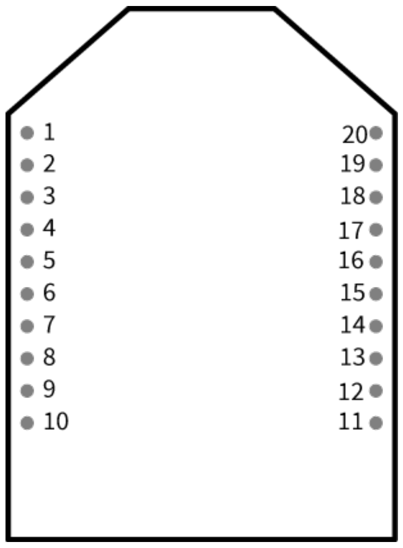
The following figures show the mechanical drawings for the XBee Cellular Modem. All dimensions are in inches.





Pin signals

The pin locations are:



The following table shows the pin assignments for the through-hole device. In the table, low-asserted signals have a horizontal line above signal name.

Pin	Name	Direction	Default	Description
Pin	Name	Direction	Default	Description
1	V_{CC}			Power supply
2	DOUT	Output	Output	UART Data Out
3	DIN / $\overline{\text{CONFIG}}$	Input	Input	UART Data In
4	DIO12 / SPI_MISO	Either	Disabled	Digital I/O 12 or SPI Slave Output line
5	$\overline{\text{RESET}}$	Input		
6	PWM0 / RSSI / DIO10	Either	Output	PWM Output 0 / RX Signal Strength Indicator / Digital I/O 10
7	DIO11	Either	Disabled	Digital I/O 11
8	[reserved]			Do not connect
9	$\overline{\text{DTR}}$ / SLEEP_RQ/ DIO8	Either	Disabled	Pin Sleep Control Line or Digital I/O 8
10	GND			Ground
11	DIO4 / SPI_MOSI	Either	Disabled	Digital I/O 4 or SPI Slave Input Line
12	$\overline{\text{CTS}}$ / DIO7	Either	Output	Output Clear-to-Send Flow Control or Digital I/O 7
13	ON /$\overline{\text{SLEEP}}$/DIO9	Output	Output	Module Status Indicator or Digital I/O 9
14	VREF	-		Feature not supported on this device. Used on other XBee devices for analog voltage reference.
15	Associate / DIO5	Either	Output	Associated Indicator, Digital I/O 5
16	$\overline{\text{RTS}}$ / DIO6	Either	Disabled	Input Request-to-Send Flow Control, Digital I/O 6
17	AD3 / DIO3 / SPI_$\overline{\text{SS}}$	Either	Disabled	Analog Input 3 or Digital I/O 3, SPI low enabled select line
18	AD2 / DIO2 / SPI_CLK	Either	Disabled	Analog Input 2 or Digital I/O 2, SPI Clock line
19	AD1 / DIO1 / SPI_$\overline{\text{ATTN}}$	Either	Disabled	Analog Input 1 or Digital I/O 1, SPI Attention line output
20	AD0 / DIO0	Either	Input	Analog Input 0, Digital I/O 0

Pin connection recommendations

The recommended minimum pin connections are VCC, GND, DIN, DOUT, $\overline{\text{RTS}}$, $\overline{\text{DTR}}$ and $\overline{\text{RESET}}$. Firmware updates require access to these pins.

RSSI PWM

The XBee Cellular Modem features an RSSI/PWM pin (pin 6) that, if enabled, adjusts the PWM output to indicate the signal strength of the cellular connection. Use P0 (DIO10/PWM0 Configuration) to enable the RSSI pulse width modulation (PWM) output on the pin. If **P0** is set to 1, the RSSI/PWM pin outputs a PWM signal where the frequency is adjusted based on the received signal strength of the cellular connection.

The RSSI/PWM output is enabled continuously unlike other XBee products where the output is enabled for a short period of time after each received transmission. If running on the XBIB development board, DIO10 is connected to the RSSI LEDs, which may be interpreted as follows:

PWM duty cycle	Number of LEDs turned on	Received signal strength (dBm)
79.39% or more	3	-83 dBm or higher
62.42% to 79.39%	2	-93 to -83 dBm
45.45% to 62.42%	1	-103 to -93 dBm
Less than 45.45%	0	Less than -103 dBm, or no cellular network connection

SIM card

The XBee Cellular Modem uses a 4FF (Nano) size SIM card.

CAUTION! Never insert or remove SIM card while the power is on!

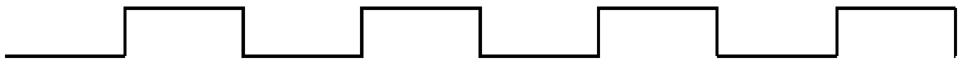
Associate LED functionality

The following table describes the Associate LED functionality. For the location of the Associate LED on the XBIB-U development board, see number 6 on the XBIB-U-DEV reference.

LED status	Blink timing	Meaning
On, solid		Not joined to a mobile network.

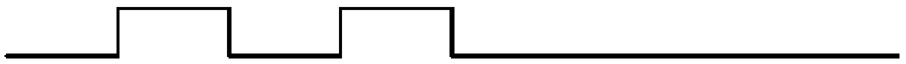
LED status	Blink timing	Meaning
Double blink	½ second	The last TCP/UDP/SMS attempt failed. If the LED has this pattern, you may need to check DI (Remote Manager Indicator) or CI (Protocol/Connection Indication) for the cause of the error.
Standard single blink	1 second	Normal operation.

The normal association LED signal alternates evenly between high and low as shown below:



Where the low signal means LED off and the high signal means LED on.

When **CI** is not **0** or **0xFF**, the Associate LED has a different blink pattern that looks like this:



Antenna recommendations

Antenna specifications

This equipment complies with FCC and IC radiation exposure limits set forth for an uncontrolled environment. The antenna should be installed and operated with minimum distance of 20 cm between the radiator and your body. Antenna gain must be below:

Frequency band	Gain
Band 4 (1700 MHz)	12.9 dBi
Band 13 (700 MHz)	6.0 dBi

This transmitter must not be co-located or operating in conjunction with any other antenna or transmitter.

Cet appareil est conforme aux limites d'exposition aux rayonnements de la IC pour un environnement non contrôlé. L'antenne doit être installé de façon à garder une distance minimale de 20 centimètres entre la source de rayonnements et votre corps. Gain de l'antenne doit être ci-dessous:

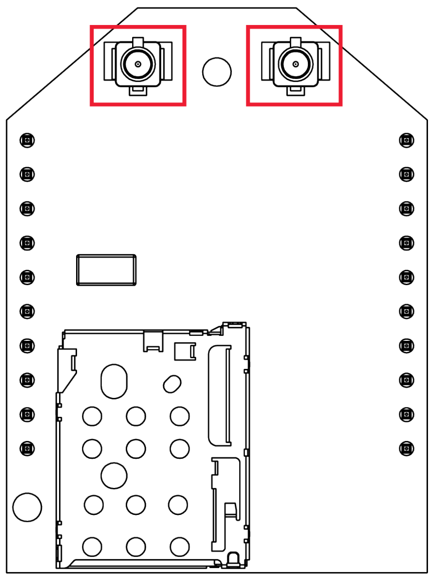
Bande de fréquence	Gain
Band 4 (1700 MHz)	12.9 dBi
Band 13 (700 MHz)	6.0 dBi

L'émetteurs ne doit pas être colocalisé ni fonctionner conjointement avec à autre antenne ou autre émetteur.

Antenna connections

CAUTION! The XBee Cellular Modem will not function properly with only the secondary antenna port connected!

The XBee Cellular Modem has two U.FL antenna ports; a primary on the upper left of the board and a secondary port on the upper right, see the drawing below. You must connect the primary port and the secondary port is optional. The secondary antenna improves receive performance in certain situations, so we recommend it for best results.



Antenna placement

It is important to keep the antenna as far away from the XBee Cellular Modem and other metal objects as possible. Often, small antennas are desirable, but at the cost of increasing size of dead zones because of reduced range and efficiency.

We recommend that antennas do not touch each other, but the XBee Cellular Modem works if they do. To optimize receive performance, orient the two antennas at right angles to each other.

RF exposure

If you are an integrating the into another product, you must include the following Caution statement in OEM product manuals to alert users of FCC RF exposure compliance:

CAUTION! To satisfy FCC RF exposure requirements for mobile transmitting devices, a separation distance of 20 cm or more should be maintained between the antenna of this device and persons during device operation. To ensure compliance, operations at closer than this distance are not recommended. The antenna used for this transmitter must not be co-located in conjunction with any other antenna or transmitter.

Design recommendations

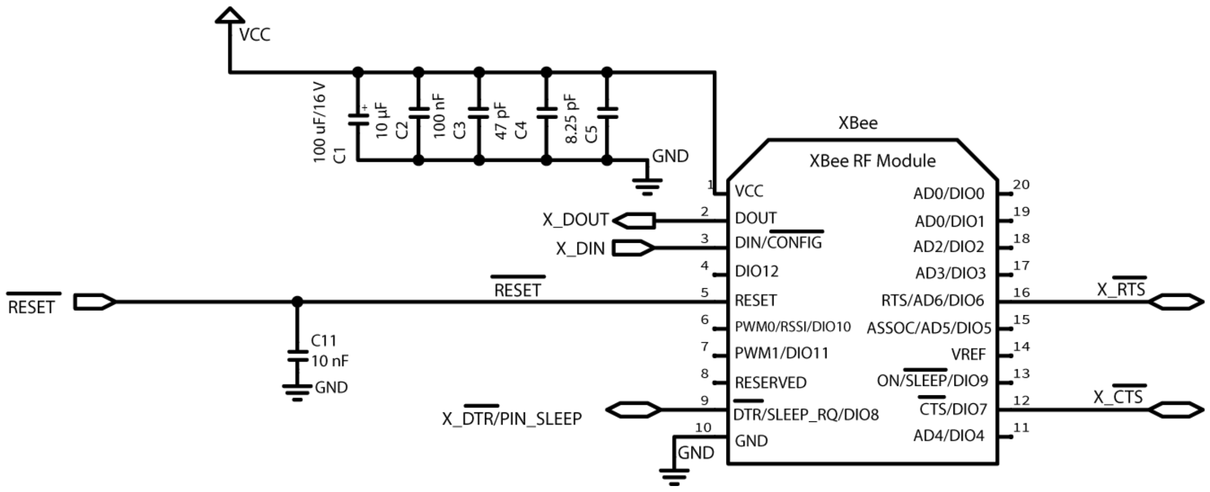
Power supply considerations

When considering a power supply, use the following design practices.

1. Power supply ripple should be less than 75 mV peak to peak.
2. The power supply should be capable of providing a minimum of 1.5 A at 3.3 V (5 W). Keep in mind that operating at a lower voltage requires higher current capability from the power supply to achieve the 5 W requirement.
3. Place sufficient bulk capacitance on the XBee VCC pin to maintain voltage above the minimum specification during inrush current. Inrush current is about 2 A during initial power up of cellular communications and wakeup from sleep mode.
4. Place smaller high frequency ceramic capacitors very close to the XBee Cellular Modem VCC pin to decrease high frequency noise.
5. Use a wide power supply trace or power plane to ensure it can handle the peak current requirements with minimal voltage drop. We recommend that the power supply and trace be designed such that the voltage at the XBee VCC pin does not vary by more than 0.1 V between light load (~0.5 W) and heavy load (~3 W).

Add a capacitor to the RESET line

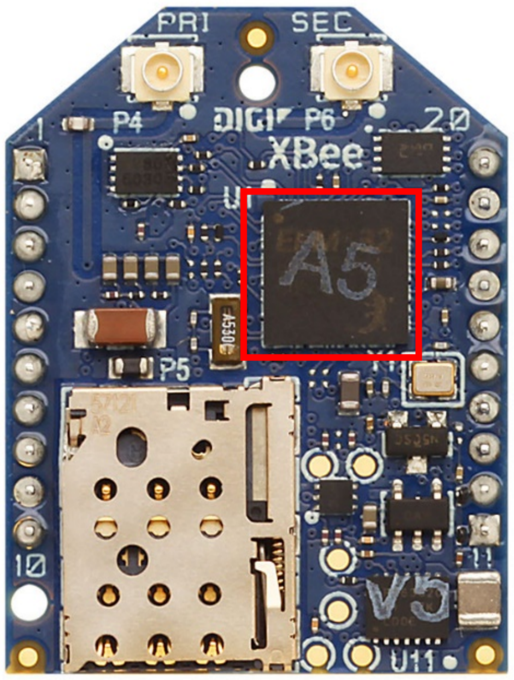
In high EMI noise environments, we recommend adding a 10 nF ceramic capacitor very close to pin 5.



Heat considerations and testing

The XBee Cellular Modem may generate significant heat during sustained operation. In addition to heavy data transfer, other factors that can contribute to heating include ambient temperature, air flow around the device, and proximity to the nearest cellular tower (the XBee Cellular Modem must transmit at a higher power level when communicating over long distances). Overheating can cause device malfunction and potential damage. In order to avoid this it is important to consider the application the XBee Cellular Modem is going into and mitigate heat issues if necessary.

We recommend that you perform thermal testing in your application to determine the resulting steady state temperature of the XBee Cellular Modem. Use TP (Temperature) to estimate the device temperature[1]. Convert the **TP** reading from hex format to decimal. We recommend that you confirm the **TP** readings by attaching a thermocouple directly to the onboard microcontroller (if using a heat sink place the thermocouple under the thermal gasket), and reading the temperature from the thermocouple. The location of the microcontroller is shown below.



You also need to know the ambient temperature and the average current consumption during your test. If you do not have a way to measure current consumption you can estimate it from the table in the next section.

Use those results to approximate the maximum safe ambient temperature for the XBee Cellular Modem, $T_{MAX,amb}$, with the following equation:

$$T_{MAX,amb} = 80°\text{C} - (T_{XBee} - T_{amb,test})\left(\frac{I_{MAX}}{I_{AVG,test}}\right)$$

Where:

T_{XBee} is the steady state temperature of the XBee Cellular Modem that you measured during your test (if using the **TP** command, be sure to convert from hex format to decimal).

$T_{amb,test}$ is the ambient air temperature during your test.

$I_{AVG,test}$ is the average current measured during your test.

I_{MAX} is the maximum current draw expected for your application during transmission (we recommend you use 950 mA unless you have verified it will be lower).

[1]The **TP** reading may not be calibrated. To compensate for this you can determine an offset to use in the equations above as follows: With the XBee Cellular Modem not powered, allow it to sit at room temperature for 15 - 20 minutes. Power the device and immediately read the **TP** command. Convert the **TP** reading from hex format to decimal and subtract the result from the actual room temperature. Add this offset to to T_{XBee} in your numbers above.

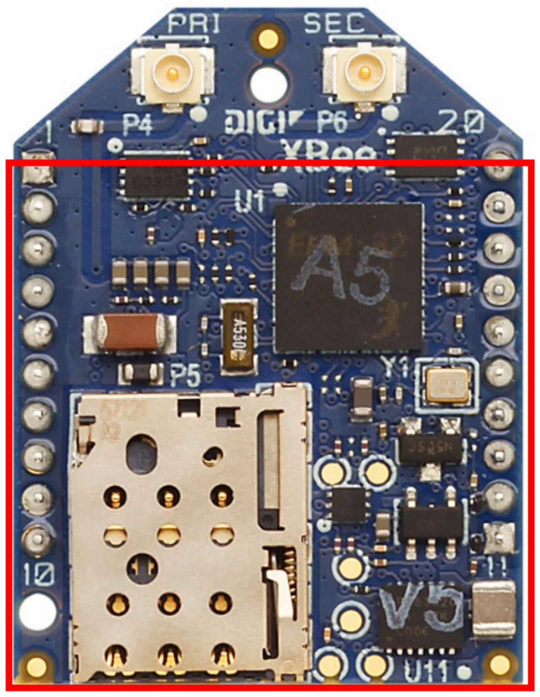
Heat sink guidelines

Based on the results of your thermal testing you may find it is advisable or required to implement a method of dissipating excess heat. This section explains how to employ a heat sink on top of the XBee Cellular Modem.

Bolt-down style

A bolt-down style heat sink on top of the XBee Cellular Modem provides the best performance. An example part number is Advanced Thermal Solutions ATS-PCBT1084/ATS-PCB1084. You must use an electrically non-conductive thermal gasket on top of the XBee device under the area that will be covered by the heat sink. A thermal gasket such as Gap Pad® 2500S20 is suitable for this purpose. We recommend using a gasket with thickness of 0.080 in to ensure that components on top of the XBee device do not tear through the material when pressure is applied to the heat sink.

Install the SIM card prior to placement of the heat sink. Position the thermal gasket and heat sink assembly on the top of the device so that it covers the microcontroller and surrounding components. You may cover the section shown inside the red box below; do not cover the U.FL connectors. When attaching to the host PCB, tighten the mounting hardware until the thermal gasket is compressed about 25%. Avoid overtightening. To prevent shorting, check that the surface of the heat sink does not directly contact the XBee device.



Adhesive style heat sink

For applications where the size or mounting requirement of the bolt-down heat sink is undesirable, you may alternatively employ an adhesive style heat sink. The heat sink should be no more than 8x8 mm in size (one option is the Assman WSW Components V2016B). Use a thermally conductive epoxy to attach the heat sink directly to the microcontroller package, and to prevent shorting ensure that the heat sink does not touch any other components.

The following table provides a list of typical scenarios and the maximum ambient temperature at which the XBee Cellular Modem can be safely operated under that condition. These are provided only as guidelines as your results will vary based on application. We recommend that you perform sufficient testing, as explained in Heat considerations and testing, to ensure that the XBee Cellular Modem does not exceed temperature specifications.

Scenario	Average current consumption (VCC = 3.3 V)	Example application	Peak data consumed (MB/hr)	Maximum ambient temperature			
				No heat sink	Adhesive heat sink	Bolt-down heat sink	Bolt-down heat sink and fan
Maximum transmission duty cycle	950 mA	Running video camera	500 to 2000	N/A	25 °C	49 °C	70 °C
50% duty cycle	475 mA	Running low resolution video camera	200 to 400	42 °C	55 °C	65 °C	75 °C
20% duty cycle	200 mA	Sending high resolution photo less than once per minute	50 to 150	64 °C	72 °C	74 °C	78 °C
Device awake, limited transmissions	170 mA	Updating traffic sign	1 to 10	66 °C	74 °C	75 °C	80 °C
Device primarily asleep, very limited transmissions	20 mA	Small data transmission/ receptions which occur once per hour	Less than 0.1	80 °C	80 °C	80 °C	80 °C

Add a fan to provide active cooling

Another option for heat mitigation is to add a fan to your system to provide active cooling. You can use a fan instead of or in addition to a heat sink. The XBee Cellular Modem offers a fan control feature on I/O pin DIO11 (pin 7). When the functionality is enabled, that line is pulled high to indicate when the fan should be turned on. The line is pulled high when the device gets above 70 °C and the cellular component is running, and turns off when the device drops below 65 °C.

To enable the functionality set P1 (DIO11/PWM1 Configuration) to **1**. Note that the I/O pin is not capable of driving a fan directly; you must implement a circuit to power the fan from a suitable power source.

Custom configuration: Create a new factory default

You can create a custom configuration that is used as a new factory default. This feature is useful if you need, for example, to maintain certain settings for manufacturing or want to ensure a feature is always enabled. When you perform a factory reset on the device using the RE command, the custom configuration is set on the device rather than the original factory default settings.

For example, by default the baud rate is set to 9600. You can create a custom configuration where the baud rate is set to 115200 by default. When you use the **RE** command to reset the device to the factory defaults, the baud rate is set to the custom configuration (115200) rather than the original factory default (9600).

The custom configuration is stored in non-volatile memory. You can continue to create and save custom configurations until the XBee3 memory runs out of space. If there is no space left to save a configuration, XBee returns an error.

You can use the **!C** command to clear or overwrite a custom configuration at any time.

Set a custom configuration

1. Open XCTU on the device.
2. Enter Command mode.
3. Perform the following process for each configuration that you want to set as a factory default.
    a. Issue an **AT%F** command. This command enables you to enter a custom configuration.
    b. Issue the custom configuration command. For example: **ATBD 7**. This command sets the default for the baud rate to 115200.

Clear all custom configurations on a device

After you have set configurations using the AT%F command, you can return all configurations to the original factory defaults.

1. Open XCTU on the device.
2. Enter Command mode.
3. Issue **AT!C**.

Clean shutdown

Digi strongly recommends performing a clean shutdown procedure on your XBee cellular devices before removing power from the devices. Performing a shutdown allows the module to unregister from the cellular network and safely store operating parameters. Failure to shutdown properly has the potential to result in delays resuming network operation and in some rare instances may result in an unrecoverable module failure.

You can use any of the following methods to perform a clean shutdown.

SD (Shutdown) command

You should use the SD command to safely shut down a device before removing power. This is the recommended method.

Issue the **SD** command. When the shut down process is complete, the device returns **OK**. After the device responds **OK**, you can safely remove power from the device.

The device will return **ERROR** if any of the following actions are in progress:

- Over-the-air update of the cellular component
- Local update of the cellular component
- Over-the-air update of the XBee firmware.

In addition, if the radio can't be fully shut down within two minutes, the device returns **ERROR**.

You can verify the state of the device using the AI command. After you issue the **SD** command and a response has been returned (either **OK** or **ERROR**), issue the **AI** command. If the shutdown was successful, **2D** is returned.

Sleep feature

The recommended approach is to use one of the available sleep configurations such as the Pin Sleep feature (**SM**=1) or sleeping through MicroPython (xbee.sleep_now()). When the module has gone to sleep and the SLEEP pin (pin 13) is low, power may safely be removed.

1. Initiate sleep: Assert SLEEP_RQ
2. Wait for sleep state to be entered: SLEEP pin (pin 13) low.
3. Power off the device.

Airplane mode

Change the XBee configuration to use Airplane mode (**AM**=1). This puts the XBee into a safe state for shutdown.

1. Set **AM**=1.
2. Apply configuration change.
3. Wait 30 seconds to allow time for shutdown to occur.
4. Power off the device.

Cellular connection process

Connecting

In normal operations, the XBee Cellular Modem automatically attempts both a cellular network connection and a data network connection on power-up. The sequence of these connections is as follows:

Cellular network

1. The device powers on.
2. It looks for cellular towers.
3. It chooses a candidate tower with the strongest signal.
4. It negotiates a connection.
5. It completes cellular registration; the phone number and SMS are available.

Data network connection

1. The network enables the evolved packet system (EPS) bearer with an access point name (APN). See AN (Access Point Name) if you have APN issues. You can use OA (Operating APN) to query the APN value currently configured in the cellular component.
2. The device negotiates a data connection with the access point.
3. The device receives its IP configuration and address.
4. The AI (Association Indication) command now returns a **0** and the sockets become available.

Data communication with remote servers (TCP/UDP)

Once the data network connection is established, communication with remote servers can be initiated in several ways.

- Transparent mode data sent to the serial port (see TD (Text Delimiter) and RO (Packetization Timeout) for timing).
- API mode: Transmit (TX) Request: IPv4 - 0x20 received over the serial connection.
- Digi Remote Manager connectivity begins.

Data communication begins when:

1. A socket opens to the remote server.
2. Data is sent.

Data connectivity ends when:

1. The server closes the connection.
2. The **TM** timeout expires (see TM (IP Client Connection Timeout)).
3. The cellular network may also close the connection after a timeout set by the network operator.

Disconnecting

When the XBee Cellular Modem is put into Airplane mode or deep sleep is requested:

1. Sockets are closed, cleanly if possible.
2. The cellular connection is shut down.
3. The cellular component is powered off.

Note We recommend entering Airplane mode before resetting or rebooting the device to allow the cellular module to detach from the network.

SMS encoding

The XBee Cellular Modem transmits SMS messages using the standard GSM 03.38 character set.[1] Because this character set only provides 7 bits of space per character, the XBee Cellular Modem ignores the most significant bit of each octet in an SMS transmission payload.

The device converts incoming SMS messages to ASCII. Characters that cannot be represented in ASCII are replaced with a space (' ', or 0x20 in hex). This includes emoji and other special characters.

[1]Also referred to as the GSM 7-bit alphabet.

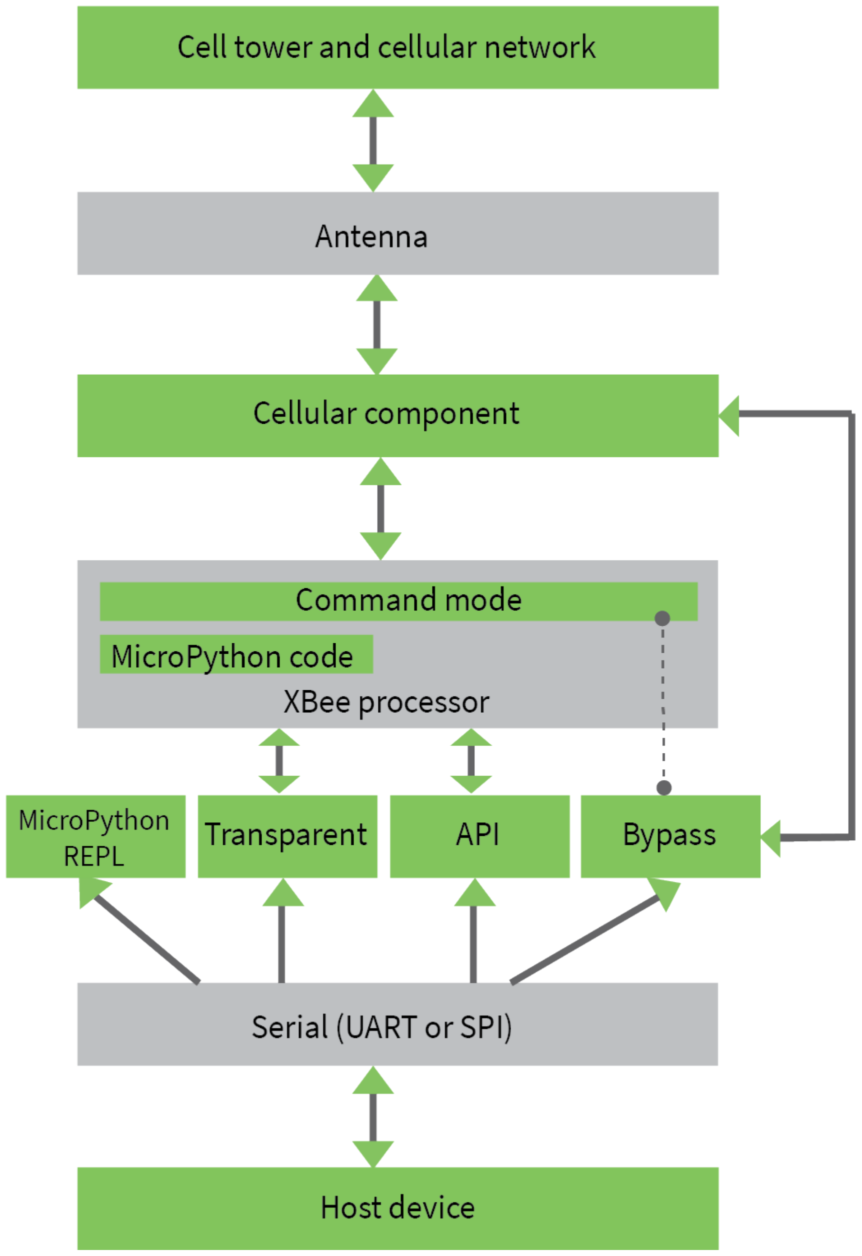
Modes

Select an operating mode

The XBee Cellular Modem interfaces to a host device such as a microcontroller or computer through a logic-level asynchronous serial port. It uses a UART for serial communication with those devices.

The XBee Cellular Modem supports three operating modes: Transparent operating mode, API operating mode, and Bypass operating mode. The default mode is Transparent operating mode. Use the AP (API Enable) command to select a different operating mode.

The following flowchart illustrates how the modes relate to each other.



Transparent operating mode

Devices operate in this mode by default. The device acts as a serial line replacement when it is in Transparent operating mode. The device queues all serial data it receives through the DIN pin for RF transmission. When a device receives RF data, it sends the data out through the DOUT pin. You can set the configuration parameters using Command mode.

The IP (IP Protocol) command setting controls how Transparent operating mode works for the XBee Cellular Modem.

Note Transparent operation is not available when using SPI.

API operating mode

API operating mode is an alternative to Transparent operating mode. API mode is a frame-based protocol that allows you to direct data on a packet basis. The device communicates UART or SPI data in packets, also known as API frames. This mode allows for structured communications with computers and microcontrollers.

The advantages of API operating mode include:

- It is easier to send information to multiple destinations
- The host receives the source address for each received data frame
- You can change parameters without entering Command mode

Bypass operating mode (DEPRECATED)

WARNING! Bypass mode is now deprecated and is not recommended for new designs. Direct access to the cellular module is not recommended or supported on the XBee Cellular CAT 1 Verizon model. XBee3 Cellular products support USB direct mode.

CAUTION! Bypass operating mode is an alternative to Transparent and API modes for advanced users with special configuration needs. Changes made in this mode might change or disable the device and we do not recommended it for most users.

In Bypass mode, the device acts as a serial line replacement to the cellular component. In this mode, the XBee Cellular Modem exposes all control of the cellular component's AT port through the UART. If you use this mode, you must setup the cellular modem directly to establish connectivity. The modem does not automatically connect to the network.

Note The cellular component can become unresponsive in Bypass mode. See Unresponsive cellular component in Bypass mode for help in this situation.

When Bypass mode is active, most of the XBee Cellular Modem's AT commands do not work. For example, **IM** (IMEI) may never return a value, and **DB** does not update. In this configuration, the firmware does not test communication with the cellular component (which it does by sending AT commands). This is useful in case you have reconfigured the cellular component in a way that makes it incompatible with the firmware. Bypass operating mode exists for users who wish to communicate directly with the cellular component settings and do not intend to use XBee Cellular Modem software features such as API mode.

Command mode is available while in Bypass mode; see Enter Command mode for instructions.

Enter Bypass operating mode

To configure a device for Bypass operating mode:

1. Set the AP (API Enable) parameter value to **5**.
2. Send WR (Write) to write the changes.
3. Send FR (Force Reset) to reboot the device.
4. After rebooting, enter Command mode and verify that Bypass operating mode is active by querying AI (Association Indication) and confirming that it returns a value of **0x2F**.

It may take a moment for Bypass operating mode to become active.

Leave Bypass operating mode

To configure a device to leave Bypass operating mode:

1. Set AP (API Enable) to something other than 5.
2. Send WR (Write) to write the changes.
3. Send FR (Force Reset) to reboot the device.
4. After rebooting, enter Command mode and verify that Bypass operating mode is not active by querying AI (Association Indication) and confirming that it returns a value other than **0x2F**.

Restore cellular settings to default in Bypass operating mode

Send **AT&F1** to reset the cellular component to its factory profile.

Command mode

Command mode is a state in which the firmware interprets incoming characters as commands. It allows you to modify the device's configuration using parameters you can set using AT commands. When you want to read or set any parameter of the XBee Cellular Modem using this mode, you have to send an AT command. Every AT command starts with the letters **AT** followed by the two characters that identify the command and then by some optional configuration values.

The operating modes of the XBee Cellular Modem are controlled by the AP (API Enable) setting, but Command mode is always available as a mode the device can enter while configured for any of the operating modes.

Command mode is available on the UART interface for all operating modes. You cannot use the SPI interface to enter Command mode.

Enter Command mode

To get a device to switch into Command mode, you must issue the following sequence: **+++** within one second. There must be at least one second preceding and following the **+++** sequence. Both the command character (**CC**) and the silence before and after the sequence (**GT**) are configurable. When the entrance criteria are met the device responds with **OK\r** on UART signifying that it has entered Command mode successfully and is ready to start processing AT commands.

If configured to operate in Transparent operating mode, when entering Command mode the XBee Cellular Modem knows to stop sending data and start accepting commands locally.

Note Do not press **Return** or **Enter** after typing **+++** because it interrupts the guard time silence and prevents you from entering Command mode.

When the device is in Command mode, it listens for user input and is able to receive AT commands on the UART. If **CT** time (default is 10 seconds) passes without any user input, the device drops out of Command mode and returns to the previous operating mode. You can force the device to leave Command mode by sending CN (Exit Command mode).

You can customize the command character, the guard times and the timeout in the device's configuration settings. For more information, see CC (Command Sequence Character), CT (Command Mode Timeout) and GT (Guard Times).

Troubleshooting

Failure to enter Command mode is often due to baud rate mismatch. Ensure that the baud rate of the connection matches the baud rate of the device. By default, BD (Baud Rate) = **3** (9600 b/s).

There are two alternative ways to enter Command mode:

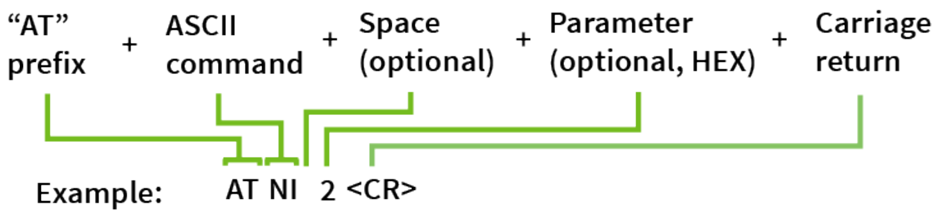
- A serial break for six seconds enters Command mode. You can issue the "break" command from a serial console, it is often a button or menu item.
- Asserting DIN (serial break) upon power up or reset enters Command mode. XCTU guides you through a reset and automatically issues the break when needed.

Both of these methods temporarily set the device's baud rate to 9600 and return an **OK** on the UART to indicate that Command mode is active. When Command mode exits, the device returns to normal operation at the baud rate that **BD** is set to.

Send AT commands

Once the device enters Command mode, use the syntax in the following figure to send AT commands. Every AT command starts with the letters **AT**, which stands for "attention." The AT is followed by two characters that indicate which command is being issued, then by some optional configuration values.

To read a parameter value stored in the device's register, omit the parameter field.



Multiple AT commands

You can send multiple AT commands at a time when they are separated by a comma in Command mode; for example, **ATNIMy XBee,AC<cr>**.

The preceding example changes the **NI (Node Identifier)** to **My XBee** and makes the setting active through AC (Apply Changes).

Parameter format

Refer to the list of AT commands for the format of individual AT command parameters. Valid formats for hexidecimal values include with or without a leading **0x** for example **FFFF** or **0xFFFF**.

Response to AT commands

When using AT commands to set parameters the XBee Cellular Modem responds with **OK<cr>** if successful and **ERROR<cr>** if not.

For devices with a file system:

ATAP1<cr>

OK<cr>

When reading parameters, the device returns the current parameter value instead of an **OK** message.

ATAP<cr>

1<cr>

Apply command changes

Any changes you make to the configuration command registers using AT commands do not take effect until you apply the changes. For example, if you send the **BD** command to change the baud rate, the actual baud rate does not change until you apply the changes. To apply changes:

1. Send AC (Apply Changes).
2. Send WR (Write).

   or:
3. Exit Command mode.

Make command changes permanent

Send a WR (Write) command to save the changes. **WR** writes parameter values to non-volatile memory so that parameter modifications persist through subsequent resets.

Send as RE command to wipe settings saved using **WR** back to their factory defaults.

Note You still have to use **WR** to save the changes enacted with **RE**.

Exit Command mode

1. Send CN (Exit Command mode) followed by a carriage return.

   or:
2. If the device does not receive any valid AT commands within the time specified by CT (Command Mode Timeout), it returns to Transparent or API mode. The default Command mode timeout is 10 seconds.

For an example of programming the device using AT Commands and descriptions of each configurable parameter, see AT commands.

MicroPython mode

MicroPython mode (**AP** = **4**) allows you to communicate with the XBee Cellular Modem using the MicroPython programming language. You can use the MicroPython Terminal tool in XCTU to communicate with the MicroPython stack of the XBee Cellular Modem through the serial interface.

MicroPython mode connects the primary serial port to the stdin/stdout interface on MicroPython, which is either the REPL or code launched at startup.

When code runs in MicroPython with **AP** set to a value other than **4**, stdout goes to the bit bucket and there is no input to read on stdin.

Sleep modes

About sleep modes

A number of low-power modes exist to enable devices to operate for extended periods of time on battery power. Use SM (Sleep Mode) to enable these sleep modes.

Normal mode

Set **SM** to 0 to enter Normal mode.

Normal mode is the default sleep mode. If a device is in this mode, it does not sleep and is always awake.

Devices in Normal mode are typically mains powered.

Pin sleep mode

Set **SM** to 1 to enter pin sleep mode.

Pin sleep allows the device to sleep and wake according to the state of the SLEEP_RQ pin (pin 9).

When you assert SLEEP_RQ (high), the device finishes any transmit or receive operations, closes any active connection, and enters a low-power state.

When you de-assert SLEEP_RQ (low), the device wakes from pin sleep.

Cyclic sleep mode

Set **SM** to 4 to enter Cyclic sleep mode.

Cyclic sleep allows the device to sleep for a specific time and wake for a short time to poll.

If you use the **D7** command to enable hardware flow control, the $\overline{\text{CTS}}$ pin asserts (low) when the device wakes and can receive serial data, and de-asserts (high) when the device sleeps.

Cyclic sleep with pin wake up mode

Set **SM** to 5 to enter Cyclic sleep with pin wake up mode.

This mode is a slight variation on Cyclic sleep mode (**SM** = 4) that allows you to wake a device prematurely by de-asserting the SLEEP_RQ pin (pin 9).

In this mode, you can wake the device after the sleep period expires, or if a high-to-low transition occurs on the SLEEP_RQ pin.

Airplane mode

While not technically a sleep mode, Airplane mode is another way of saving power. When set, the cellular component of the XBee Cellular Modem is fully turned off and no access to the cellular network is performed or possible. Use AM (Airplane Mode) to configure this mode.

Connected sleep mode

XBee Cellular Modem hardware part number XBC-V1-UT-xxx can enter Connected sleep mode.

Only some hardware versions are compatible with this mode. To see if the module is capable of using connected sleep mode, read the HI (Hardware Identity) command. If the value returned is 3 then the hardware is compatible and connected sleep mode is available.

Set bit 0 of SO (Sleep Options) for connected sleep. When bit 0 is set and the XBee Cellular Modem goes to sleep, instead of the cellular component shutting down, it enters a lower power consumption mode that maintains registration with the cellular network. This allows significantly faster resumption of communications when coming out of sleep at the cost of additional power used.

Connected sleep mode draws 9 mA during sleep and 11 mA average over time, which includes periodically waking up to maintain connection.

The sleep timer

The sleep timer starts when the device wakes and resets on re-configuration. When the sleep timer expires the device returns to sleep.

MicroPython sleep behavior

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.

Upon entering deep 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 will:

- **socket.send** raises **OSError: ENOTCONN**
- **socket.sendto** raises **OSError: ENOTCONN**
- **socket.recv** returns the empty string, the traditional end-of-file return value
- **socket.recvfrom** returns an empty message, for example:

  (b'', (<address from connect()>, <port from connect()>))

  The underlying UDP socket resources have been released at this point.

Serial communication

Serial interface

The XBee Cellular Modem interfaces to a host device through a serial port. The device's serial port can communicate:

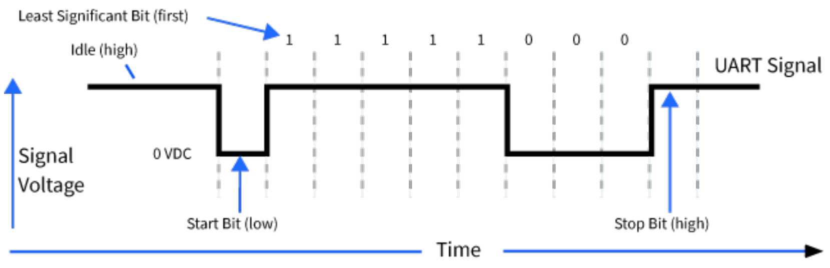
- Through a logic and voltage compatible universal asynchronous receiver/transmitter (UART).
- Through a level translator to any serial device, for example, through an RS-232 or USB interface board.
- Through a serial peripheral interface (SPI) port.

Serial data

A device sends data to the XBee Cellular Modem's UART through pin 3 DIN as an asynchronous serial signal. When the device is not transmitting data, the signals should idle high.

For serial communication to occur, you must configure the UART of both devices (the microcontroller and the XBee Cellular Modem) with compatible settings for the baud rate, parity, start bits, stop bits, and data bits.

Each data byte consists of a start bit (low), 8 data bits (least significant bit first) and a stop bit (high). The following diagram illustrates the serial bit pattern of data passing through the device. The diagram shows UART data packet 0x1F (decimal number 31) as transmitted through the device.

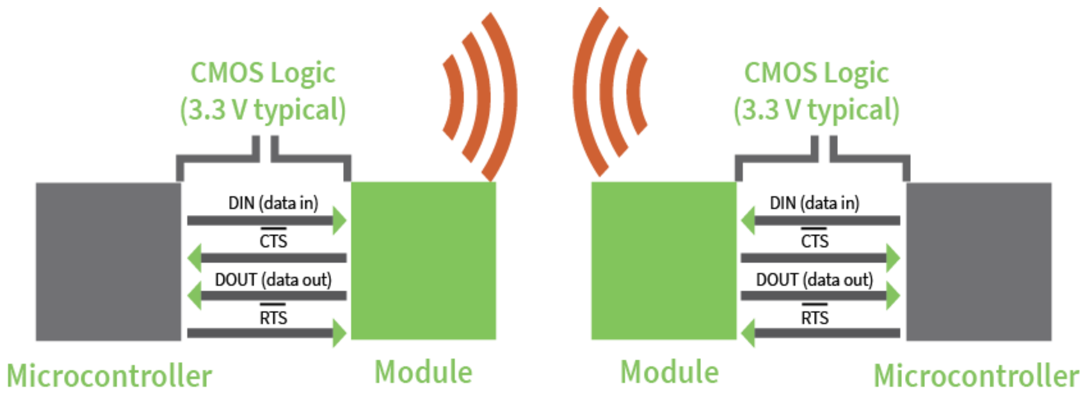


You can configure the UART baud rate, parity, and stop bits settings on the device with the **BD**, **NB**, and **SB** commands respectively. For more information, see Serial interfacing commands.

In the rare case that a device has been configured with the UART disabled, you can recover the device to UART operation by holding DIN low at reset time. DIN forces a default configuration on the UART at 9600 baud and it brings the device up in Command mode on the UART port. You can then send the appropriate commands to the device to configure it for UART operation. If those parameters are written, the device comes up with the UART enabled on the next reset.

UART data flow

Devices that have a UART interface connect directly to the pins of the XBee Cellular Modem as shown in the following figure. The figure shows system data flow in a UART-interfaced environment. Low-asserted signals have a horizontal line over the signal name.



Serial buffers

The XBee Cellular Modem maintains internal buffers to collect serial and RF data that it receives. The serial receive buffer collects incoming serial characters and holds them until the device can process them. The serial transmit buffer collects the data it receives via the RF link until it transmits that data out the serial or SPI port.

$\overline{\text{CTS}}$ flow control

We strongly encourage you to use flow control with the XBee Cellular Modem to prevent buffer overruns.

$\overline{\text{CTS}}$ flow control is enabled by default; you can disable it with D7 (DIO7/CTS). When the serial receive buffer fills with the number of bytes specified by FT (Flow Control Threshold), the device de-asserts CTS (sets it high) to signal the host device to stop sending serial data. The device re-asserts CTS when less than FT-16 bytes are in the UART receive buffer.

Note Serial flow control is not possible when using the SPI port.

$\overline{\text{RTS}}$ flow control

If you set D6 (DIO6/RTS) to enable $\overline{\text{RTS}}$ flow control, the device does not send data in the serial transmit buffer out the DOUT pin as long as RTS is de-asserted (set high). Do not de-assert RTS for long periods of time or the serial transmit buffer will fill.

SPI operation

SPI communications

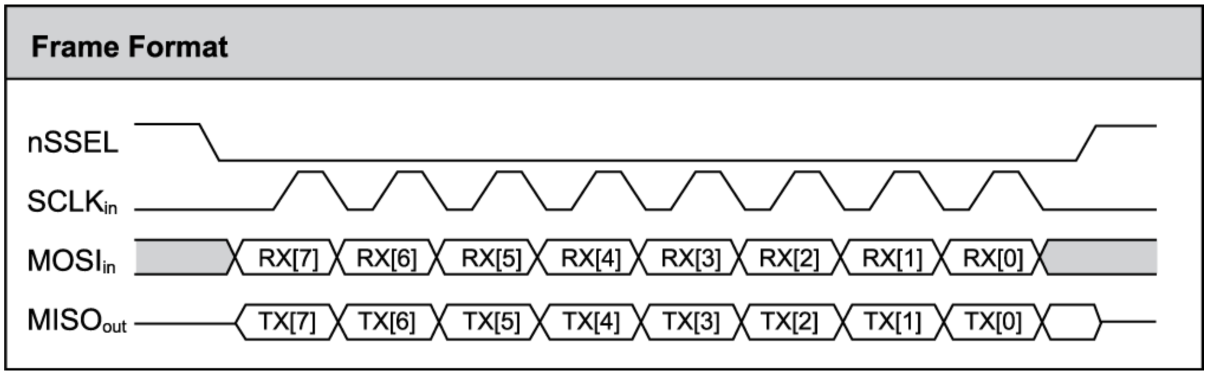
The XBee Cellular Modem supports SPI communications in slave mode. Slave mode receives the clock signal and data from the master and returns data to the master. The following table shows the signals that the SPI port uses on the device.

Signal	Function
SPI_MOSI (Master Out, Slave In)	Inputs serial data from the master
SPI_MISO (Master In, Slave Out)	Outputs serial data to the master
SPI_SCLK (Serial Clock)	Clocks data transfers on MOSI and MISO
SPI_$\overline{\text{SSEL}}$ (Slave Select)	Enables serial communication with the slave
SPI_$\overline{\text{ATTN}}$ (Attention)	Alerts the master that slave has data queued to send. The XBee Cellular Modem asserts this pin as soon as data is available to send to the SPI master and it remains asserted until the SPI master has clocked out all available data.

In this mode:

- SPI clock rates up to 6 MHz are possible.
- Data is most significant bit (MSB) first; bit 7 is the first bit of a byte sent over the interface.
- Frame Format mode 0 is used. This means CPOL= 0 (idle clock is low) and CPHA = 0 (data is sampled on the clock's leading edge).
- The SPI port only supports API Mode (**AP** = **1**).

The following diagram shows the frame format mode 0 for SPI communications.


Frame Format
nSSEL
SCLKin
MOSIin
RX[7] RX[6] RX[5] RX[4] RX[3] RX[2] RX[1] RX[0]
MISOout
TX[7] TX[6] TX[5] TX[4] TX[3] TX[2] TX[1] TX[0]

SPI mode is chip to chip communication. We do not supply a SPI communication option on the device development evaluation boards.

Full duplex operation

The specification for SPI includes the four signals SPI_MISO, SPI_MOSI, SPI_CLK, and SPI_SSEL. Using these four signals, the SPI master cannot know when the slave needs to send and the SPI slave cannot transmit unless enabled by the master. For this reason, the SPI_ATTN signal is available in the design. This allows the SPI slave to alert the SPI master that it has data to send. In turn, the SPI master is expected to assert SPI_SSEL and start SPI_CLK, unless these signals are already asserted and active respectively. This, in turn, allows the XBee Cellular Modem SPI slave to send data to the master.

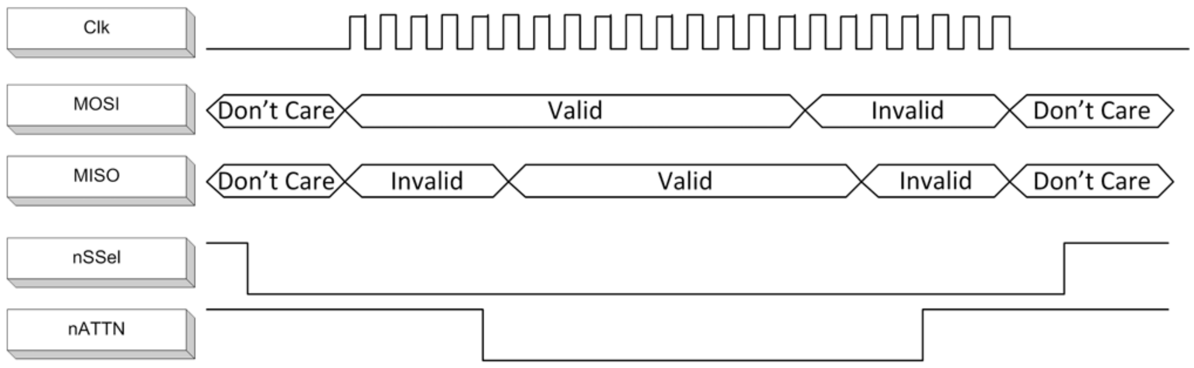
SPI data is latched by the master and slave using the SPI_CLK signal. When data is being transferred the MISO and MOSI signals change between each clock. If data is not available then these signals will not change and will be either 0 or 1. This results in receiving either a repetitive 0 or 0xFF. The means of determining whether or not received data is valid is by packetizing the data with API packets, without escaping. Valid data to and from the XBee Cellular Modem is delimited by 0x7E, a length, the payload, and finally a checksum byte. Everything else in both directions should be ignored. The bytes received between frames will be either 0xff or 0x00. This allows the SPI master to scan for a 0x7E delimiter between frames.

SPI allows for valid data from the slave to begin before, at the same time, or after valid data begins from the master. When the master is sending data to the slave and the slave has valid data to send in the middle of receiving data from the master, it allows a true full duplex operation where data is valid in both directions for a period of time. During this time, the master and slave must simultaneously transmit valid data at the clock speed so that no invalid bytes appear within an API frame, causing the whole frame to be discarded.

An example follows to more fully illustrate the SPI interface during the time valid data is being sent in both directions. First, the master asserts SPI_SSEL and starts SPI_CLK to send a frame to the slave.

Initially, the slave does not have valid data to send the master. However, while it is still receiving data from the master, it has its own data to send. Therefore, it asserts SPI_ATTN low. Seeing that SPI_ SSEL is already asserted and that SPI_CLK is active, it immediately begins sending valid data, even while it is receiving valid data from the master. In this example, the master finishes its valid data before the slave does. The master will have two indications of valid data: The SPI_ATTN line is asserted and the API frame length is not yet expired. For both of these reasons, the master should keep SPI_SSEL asserted and should keep SPI_CLK toggling in order to receive the end of the frame from the slave, even though these signals were originally turned on by the master to send data. During the time that the SPI master is sending invalid data to the SPI slave, it is important no 0x7E is included in that invalid data because that would trigger the SPI slave to start receiving another valid frame.

The following figure illustrates the SPI interface while valid data is being sent in both directions.



Low power operation

Sleep modes generally work the same on SPI as they do on UART. However, due to the addition of SPI mode, there is an option of another sleep pin, as described below.

By default, Digi configures DIO8 (SLEEP_REQUEST) as a peripheral and during pin sleep it wakes the device and puts it to sleep. This applies to both the UART and SPI serial interfaces.

If SLEEP_REQUEST is not configured as a peripheral and SPI_SSEL is configured as a peripheral, then pin sleep is controlled by SPI_SSEL rather than by SLEEP_REQUEST. Asserting SPI_SSEL (pin 17) by driving it low either wakes the device or keeps it awake. Negating SPI_SSEL by driving it high puts the device to sleep.

Using SPI_SSEL to control sleep and to indicate that the SPI master has selected a particular slave device has the advantage of requiring one less physical pin connection to implement pin sleep on SPI. It has the disadvantage of putting the device to sleep whenever the SPI master negates SPI_SSEL (meaning time is lost waiting for the device to wake), even if that was not the intent.

If the user has full control of SPI_SSEL so that it can control pin sleep, whether or not data needs to be transmitted, then sharing the pin may be a good option in order to make the SLEEP_REQUEST pin available for another purpose.

If the device is one of multiple slaves on the SPI, then the device sleeps while the SPI master talks to the other slave, but this is acceptable in most cases.

If you do not configure either pin as a peripheral, then the device stays awake, being unable to sleep in **SM**1 mode.

Select the SPI port

To force SPI mode, hold DOUT/DIO13 pin 2 low while resetting the device until SPI_ATTN asserts. This causes the device to disable the UART and go straight into SPI communication mode. Once configuration is complete, the device queues a modem status frame to the SPI port, which causes the SPI_ATTN line to assert. The host can use this to determine that the SPI port is configured properly. This method forces the configuration to provide full SPI support for the following parameters:

- **D1** (This parameter will only be changed if it is at a default of zero when the method is invoked.)
- **D2**
- **D3**
- **D4**
- **P2**

As long as the host does not issue a **WR** command, these configuration values revert to previous values after a power-on reset. If the host issues a **WR** command while in SPI mode, these same parameters are written to flash. After a reset, parameters that were forced and then written to flash become the mode of operation.

If the UART is disabled and the SPI is enabled in the written configuration, then the device comes up in SPI mode without forcing it by holding DOUT low. If both the UART and the SPI are enabled at the time of reset, then output goes to the UART until the host sends the first input. If that first input comes on the SPI port, then all subsequent output goes to the SPI port and the UART is disabled. If the first input comes on the UART, then all subsequent output goes to the UART and the SPI is disabled.

Once you select a serial port (UART or SPI), all subsequent output goes to that port, even if you apply a new configuration. The only way to switch the selected serial port is to reset the device. On surface-mount devices, forcing DOUT low at the time of reset has no effect. To use SPI mode on the SMT devices, assert the SPI_SSEL (pin 17) low after reset and before any UART data is input.

When the master asserts the slave select (SPI_SSEL) signal, SPI transmit data is driven to the output pin SPI_MISO, and SPI data is received from the input pin SPI_MOSI. The SPI_SSEL pin has to be asserted to enable the transmit serializer to drive data to the output signal SPI_MISO. A rising edge on SPI_SSEL causes the SPI_MISO line to be tri-stated such that another slave device can drive it, if so desired.

If the output buffer is empty, the SPI serializer transmits the last valid bit repeatedly, which may be either high or low. Otherwise, the device formats all output in API mode 1 format, as described in Operate in API mode. The attached host is expected to ignore all data that is not part of a formatted API frame.

Force UART operation

If you configure a device with only the SPI enabled and no SPI master is available to access the SPI slave port, you can recover the device to UART operation by holding DIN / CONFIG low at reset time. DIN/CONFIG forces a default configuration on the UART at 9600 baud and brings up the device in Command mode on the UART port. You can then send the appropriate commands to the device to configure it for UART operation. If you write those parameters, the device comes up with the UART enabled on the next reset.

Data format

SPI only operates in API mode 1. The XBee Cellular Modem does not support Transparent mode or API mode 2 (which escapes control characters). This means that the AP configuration only applies to the UART, and the device ignores it while using SPI. The reason for this operation choice is that SPI is full duplex. If data flows in one direction, it flows in the other. Since it is not always possible to have valid data flowing in both directions at the same time, the receiver must have a way to parse out the valid data and to ignore the invalid data.

The XBee Cellular Modem sends **0XFF** when there is no data to send to the host.

File system

For detailed information about using MicroPython on the XBee Cellular Modem refer to the *Digi MicroPython Programming Guide*.

Overview of the file system

XBee Cellular Modem firmware versions ending in **0B** (for example, 1130B, 100B, 3100B) and later include support for storing files on an internal 1 MB SPI flash.

CAUTION! You need to format the file system if upgrading a device that originally shipped with older firmware. You can use XCTU, AT commands or MicroPython for that initial format or to erase existing content at any time.

Note To use XCTU with file system, you need XCTU 6.4.0 or newer.

See ATFS FORMAT confirm and ensure that the format is complete.

Directory structure

The SPI flash appears in the file system as **/flash**, the only entry at the root level of the file system. It has a **lib** directory intended for MicroPython modules and a **cert** directory for files used for TLS sockets.

Paths

The XBee Cellular Modem stores all of its files in the top-level directory **/flash**. On startup, the **ATFS** commands and MicroPython each use that as their current working directory. When specifying the path to a file or directory, it is interpreted as follows:

- Paths starting with a forward slash are "absolute" and must start with **/flash** to be valid.
- All other paths are relative to the current working directory.
- The directory **..** refers to the parent directory, so an operation on **../filename.txt** that takes place in the directory **/flash/test** accesses the file **/flash/filename.txt**.
- The directory **.** refers to the current directory, so the command **ATFS ls .** lists files in the current directory.
- Names are case-insensitive, so **FILE.TXT**, **file.txt** and **FiLe.TxT** all refer to the same file.
- File and directory names are limited to 64 characters, and can only contain letters, numbers, periods, dashes and underscores. A period at the end of the name is ignored.
- The full, absolute path to a file or directory is limited to 255 characters.

Secure files

The file system includes support for secure files with the following properties:

- Created via the **ATFS XPUT** command or in MicroPython using a mode of ***** with the **open()** method.
- Unable to download via the **ATFS GET** command or MicroPython's **open()** method.
- SHA256 hash of file contents available from **ATFS HASH** command (to compare with a local copy of a file).
- Encrypted on the SPI flash.

- MicroPython can execute code in secure files.
- Sockets can use secure files when creating TLS connections.

XCTU interface

XCTU releases starting with 6.4.0 include a **File System Manager** in the **Tools** menu. You can upload files to and download files from the device, in addition to renaming and deleting existing files and directories. See the File System manager tool section of the *XCTU User Guide* for details of its functionality.

Encrypt files

You can encrypt files on the file system. This provides two things:

1. Protection of the client private key for TLS authentication while it is stored on the XBee Cellular Modem.
2. Protection for user's MicroPython applications.

Use ATFS XPUT filename to place encrypted files on the file system. The XPUT operation is otherwise identical to the PUT operation. Files placed in this way are indicated with a **pound sign** (**#**) following the filename. The XBee Cellular Modem does not allow an encrypted file to be read by normal use so it:

1. Cannot be retrieved with the GET operation.
2. Cannot be opened and read in MicroPython applications.
3. Cannot be created by a MicroPython application.

When ATFS HASH filename is run with the filename of an encrypted file, it reports the SHA256 hash of the file contents. In this way you can validate that the correct file has been placed on the XBee Cellular Modem.

Socket behavior

Supported sockets

The XBee Cellular Modem supports the following number of sockets:

- 10 maximum: some combination of 6 TCP, 6 UDP, 6 TLS.[1]

Best practices when using sockets

Sockets and Remote Manager

If you use Remote Manager to remotely communicate with and configure your XBee Cellular device, you must leave at least two sockets available in the system: one UDP socket (for periodic low-data-usage check-ins), and one TCP/TLS socket (to be used when a full connection is needed).

If your application allocates so many sockets that Remote Manager functionality in the firmware cannot get the sockets that it requires, Remote Manager functionality will be prevented from working until sockets become available.

For example, each call to `socket.socket()` in MicroPython will allocate a socket, and this socket will remain allocated to MicroPython until the socket's close method is called, or the MicroPython REPL is restarted using Ctrl-D.

See Supported sockets for more information on the total number of sockets supported by the device.

Sockets and API mode

When using API mode to transmit TCP/TLS data to a remote destination (using the 0x20 or 0x23 API frames), sending a large amount of data as a single API frame is preferable to multiple smaller API frames. Using a single large API frame allows the XBee to transmit the data using fewer operations than transmitting multiple pieces of data in sequence, which improves overall throughput.

Additionally, one API frame consumes less dynamic memory in the system than multiple smaller API frames, which means there will be more memory available to process incoming IP data as well as subsequent API frames sent into the XBee Cellular device.

Socket timeouts

The XBee Cellular Modem implicitly opens the socket any time there is data to be sent, and closes it according to the timeout settings. The TM (IP Client Connection Timeout) command controls the timeout settings.

Socket limits in API mode

In API mode there are a fixed number of sockets available; see Supported sockets. When a Transmit (TX) Request: IPv4 - 0x20 frame is sent to the XBee Smart Modem for a new destination, it creates a new socket. The exception to this is when using the UDP protocol with the C0 source port, which allows unlimited destinations on the socket created by C0 (Source Port). If no more sockets are available, the device sends back a Transmit (TX) Status - 0x89 frame with a Resource Error. The Resource Error resolves when an existing socket is closed. An existing socket may be closed when the socket times out (see TM (IP Client Connection Timeout) and TS (IP Server Connection Timeout)) or when the socket is closed via a TX request with the CLOSE flag set.

[1] 1 UDP socket is always reserved for DNS, so subtract 1 socket from the values above.

In API mode each socket has a maximum number of pending Transmit (TX) Requests allowed. When a Transmit (TX) Request: IPv4 - 0x20 frame is sent to the XBee Smart Modem for an existing destination, it sends that request using the socket for that destination. If the number of pending Transmit (TX) Requests would be exceeded for the socket, the device sends back a Transmit (TX) Status - 0x89 frame with a Resource Error indicating that the device is not able to send the request and should retry again later. The Resource Error resolves when a Transmit (TX) Request that is pending on the socket is transmitted; this is indicated by the Transmit (TX) Status frame for the request.

Enable incoming TCP connections

TCP establishes virtual connections between the XBee Cellular Modem and other devices. You can enable the XBee Cellular Modem to listen for incoming TCP connections. Listen means waiting for a connection request from any remote TCP and port.[1]

The following devices support incoming TCP connections:

- Part number: XBC-V1-UT-001 (Digi XBee Cellular Verizon LTE Cat 1)
- Part number: XBC-M1-UT-001 (Digi XBee Cellular AT&T LTE Cat 1)
- Part number: XB3-C-A1-UT-xxx (Digi XBee3 Cellular AT&T LTE Cat 1)
- Part number: XB3-C-V1-UT-xxx (Digi XBee3 Cellular Verizon LTE Cat 1)

The XBee Cellular Modem only supports incoming TCP and UDP connections as configured in IP (IP Protocol), TLS is not supported.

To enable incoming connections in XCTU:

1. Set AP (API Enable) to **Transparent Mode [0]** or **API Mode**. You can use either API mode with escapes or without escapes.
2. Set **IP** to **TCP [1]** or **UDP [0]**.
3. Set C0 (Source Port) to the value of the TCP port that the device listens on.
4. Click the **Write** button .

API mode behavior for outgoing TCP and TLS connections

To initiate an outgoing TCP or TLS connection to a remote host, send a Transmit (TX) Request: IPv4 - 0x20 frame to the XBee Cellular Modem's serial port specifying the destination address and destination port for the remote host; the data is optional and the source port is **0**.

If the connection is disconnected at any time, send a Transmit TX Request frame to trigger a new connection attempt.

To send data over this connection use the Transmit (TX) Request: IPv4 - 0x20.

The device sends a Transmit (TX) Status - 0x89 frame in reply to the Transmit TX Request indicating the status of the request. A status of **0** indicates the connection and/or data was successful, a value of 0x32 indicates a temporary Resource Error (see Socket limits in API mode), and other values indicates a failure.

Any data received on the connection is sent out the XBee Cellular Modem's serial port as a Receive RX frame.

A connection is closed when:

[1]See https://tools.ietf.org/html/rfc793.

- The remote end closes the connection.
- No data is sent or received for longer than the socket timeout set by TM (IP Client Connection Timeout).
- A Transmit TX Request is sent with the CLOSE flag set.

API mode behavior for outgoing UDP data

To send a UDP datagram to a remote host, send a Transmit (TX) Request: IPv4 - 0x20 frame to the XBee Cellular Modem's serial port specifying the destination address and destination port of the remote host. If you use a source port of **0**, the device creates a new socket for the purpose of sending to the remote host. The XBee Cellular Modem supports a finite number of sockets, so if you need to send to many destinations:

1. The socket must be closed after use.

   or
2. You must use the socket specified by the C0 (Source Port) setting.

To use the socket specified by the **C0** setting, in the Transmit TX request frame use a source port that matches the value configured for the **C0** setting.

The device sends a Transmit (TX) Status - 0x89 frame in reply to the Transmit TX Request to indicate the status of the request. A status of **0** indicates the connection and/or data was successful, a value of 0x32 indicates a temporary Resource Error (see Socket limits in API mode), and other values indicates a failure.

Any data received on the UDP socket is sent out the XBee Cellular Modem's serial port as a Receive (RX) Packet: IPv4 - 0xB0 frame.

A UDP socket is closed when:

- No data has been sent or received for longer than the socket timeout set by TM (IP Client Connection Timeout).
- A transmit TX Request is sent with the CLOSE flag set.

API mode behavior for incoming TCP connections

For incoming connections and data in API mode, the XBee Cellular Modem uses the C0 (Source Port) and IP (IP Protocol) settings to specify the listening port and protocol used. The XBee Cellular Modem does not currently support the TLS protocol for incoming connections.

When the **IP** setting is TCP the XBee Cellular Modem allows multiple incoming TCP connections on the port specified by the **C0** setting. Any data received on the connection is sent out the XBee Cellular Modem's serial port as a Receive (RX) Packet: IPv4 - 0xB0 frame.

To send data from the device over the connection, use the Transmit (TX) Request: IPv4 - 0x20 frame with the corresponding address fields received from the Receive RX frame. In other words:

- Take the source address, source port, and destination port fields from the Receive (RX) frame and use those respectively as:
- The destination address, destination port, and source port fields for the Transmit (TX) Request frame.

A connection is closed when:

- The remote end closes the connection.
- No data has been sent or received for longer than the socket timeout set by TS (IP Server Connection Timeout).
- A Transmit (TX) Request frame is sent with the CLOSE flag set.

API mode behavior for incoming UDP data

When the IP (IP Protocol) setting is UDP, any data sent from a remote host to the XBee Cellular Modem's network port specified by the C0 (Source Port) setting is sent out the XBee Cellular Modem's serial port as a Receive (RX) Packet: IPv4 - 0xB0 frame.

To send data from the XBee Cellular Modem to the remote destination, use the Transmit (TX) Request: IPv4 - 0x20 frame with the corresponding address fields received from the Receive RX frame. In other words take the source address, source port, and destination port fields from the Receive (RX) frame and use those respectively as the destination address, destination port, and source port fields for the Transmit (TX) Request frame.

Transparent mode behavior for outgoing TCP and TLS connections

For Transparent mode, the IP (IP Protocol) setting specifies the protocol and the DL (Destination Address) and DE (Destination port) settings specify the destination address used for outgoing data (UDP) and outgoing connections (TCP and TLS).

To initiate an outgoing TCP or TLS connection to a remote host, send data to the XBee Cellular Modem's serial port. If CI (Protocol/Connection Indication) reports a value of **0**, then the connection was successfully established, otherwise the value of **CI** indicates why the connection attempt failed. Any data received over the connection is sent out the XBee Cellular Modem's serial port.

A connection is closed when:

- The remote end closes the connection.
- No data has been sent or received for longer than the socket timeout set by TM (IP Client Connection Timeout).
- You make and apply a change to the **IP**, **DL**, or **DE**.

Transparent mode behavior for outgoing UDP data

To send outgoing UDP data to a remote host, send data to the XBee Cellular Modem's serial port. If CI (Protocol/Connection Indication) reports a value of **0**, the data was successfully sent; otherwise, the value of **CI** indicates why the data failed to be sent.

The RO (Packetization Timeout) setting provides some control in how the serial data gets packetized before being sent to the remote host. The first send opens up a UDP socket used to send and receive data. Any data received by this socket is sent out the XBee Cellular Modem's serial port.

Note Set **RO** to **FF** for realtime typing by humans. Also, see TD (Text Delimiter).

Transparent mode behavior for incoming TCP connections

The C0 (Source Port) and IP (IP Protocol) settings specify the listening port and protocol used for incoming connections (TCP) and incoming data (UDP) in Transparent mode. TLS is not currently supported for incoming connections.

When the **IP** setting is TCP and there is no existing connection to or from the XBee Cellular Modem, the device accepts one incoming connection. Any data received on the connection is sent out the XBee Cellular Modem's serial port. Any data sent to the XBee Cellular Modem's serial port is sent over the connection. If the connection is disconnected, it discards pending data.

Transparent mode behavior for incoming UDP connections

When the IP (IP Protocol) setting is UDP any data sent from a remote host to the XBee Cellular Modem's network port specified by C0 (Source Port) is sent out the XBee Cellular Modem's serial port. Any data sent to the XBee Cellular Modem's serial port is sent to the network destination specified by the DL (Destination Address) and DE (Destination port) settings. If the **DL** and **DE** settings are unspecified or invalid, the XBee Cellular Modem discards data sent to the serial port.

Extended Socket frames

The XBee Cellular product line includes a set of Extended Socket frames. You can use these frames in applications where the existing frames (Transmit Request (0x20), TLS Transmit (0x23) and Receive (0xB0)) limit the possibilities for an application.

You can use Extended Socket frames to do the following:

- Multiple simultaneous connections can be made to the same port on the same host. For example, you can overlap simultaneous HTTP requests.
- Immediate unsolicited notification of changes in socket status. This allows an application to react to a server-side socket closure rather than relying on an implicit connection to be re-established for continuing communication.
- A generalized mechanism for per-socket option selection. Currently used for TLS profile selection. Previously this required a unique frame, as options are added, this allows combinations of choices.
- Allow DNS look up during the connection process rather than a separate step.

In addition, for diagnostic purposes, you can use the Socket Info (SI) AT command to retrieve information regarding all open sockets currently active in the system. This can be queried during development or used by an application to confirm or refresh information during execution.

Note Sockets opened with the Extended Socket frames cannot be used with the legacy frames (Transmit Request (0x20), TLS Transmit (0x23) and Receive (0xB0)), nor vice versa.

For a list of the socket frames, see Available Extended Socket frames.

Examples

In the examples below the Frame IDs in all frames are set to 1 for simplicity. Socket IDs in all frames after the Socket Create are hard-coded to 0 as well. If you wish to use the example repeatedly the XBee should be rebooted between attempts.

We recommend the use of the XCTU frame generator for experimentation with frames during development. Paste the provided frame content directly into the **Add API frame to list** window in XCTU to follow along manually.

Extended Socket example: Single HTTP Connection

Extended Socket example: UDP

Extended Socket example: TCP Listener

Available Extended Socket frames

Note For information about all frames, see API frames.

Extended Socket example: Single HTTP Connection

This example demonstrates a complete request with an HTTP server. It fetches a random fact about a number from a web services API offered by the website http://numbersapi.com.

Note Digi is not affiliated with numbersapi.com and the example is for education only.

Send a Socket Create frame

Note To adapt this example for an HTTPS server, change **Protocol** below to 0x04 (TLS) and optionally use the Socket Option frame to specify a TLS profile.

Field	Value
Frame type	0x40 (Socket Create)
Frame ID	0x01
Protocol	0x01 (TCP)

Socket Create frame data:

```
7E 00 03 40 01 01 BD
```

Receive a Socket Create response

The XBee responds to the Socket Create request with a response. The response contains the socket ID assigned. In this example, the socket ID is 0.

Field	Value
Frame type	0xC0 (Socket Create Response)
Frame ID	0x01
Socket ID	0x00
Status	0x00 (Success)

Socket Create Response received from XBee:

```
7E 00 04 C0 01 00 00 3E
```

Send Socket Connect

This examples uses the "string" destination address type to have the XBee perform DNS look-up during the connection process.

Note To adapt this example for TLS, use destination port 0x01 0xbb (decimal 443). Be aware that many HTTPS servers use SNI (Server Name Identification) which is not currently supported.

Field	Value
Frame type	0x42 (Socket Create Response)
Frame ID	0x01
Socket ID	0x00
Destination Port	0x00 0x50 (80 decimal, HTTP)
Destination Address Type	0x01 (String)
Destination Address	numbersapi.com

Socket Connect frame data:

```
7E 00 14 42 01 00 00 50 01 6E 75 6D 62 65 72 73 61 70 69 2E 63 6F 6D C8
```

Receive a Socket Connect Response

The request to connect is immediately acknowledged with a response. However, it is not permitted to proceed transmitting data until the next stage, after a Socket Status frame has been received indicating success.

Field	Value
Frame type	0xC2 (Socket Connect Response)
Frame ID	0x01
Socket ID	0x00
Status	0x00 (Success)

Socket Connect Response received from XBee:

```
7E 00 04 C2 01 00 00 3C
```

Receive a Socket Status

The socket has been fully established when a Socket Status frame is received with the connected status after the socket has connected.

Field	Value
Frame type	0xCF (Socket Status)
Socket ID	0x00
Status	0x00 (Connected)

Socket Status received from XBee with connected status:

```
7E 00 03 CF 00 00 30
```

Send HTTP Request using Socket Send frame

The request uses the "Connection: close" header to have the server close the connection on request completion. This allows the example to demonstrate the Socket Status reporting of a close by the peer.

Field	Value
Frame type	0x44 (Socket Status)
Frame ID	0x01
Socket ID	0x00
Transmit Options	0x00
Data	GET /random/trivia HTTP/1.1 Host: numbersapi.com Connection: close

Socket Send frame data:

```
7E 00 4C 44 01 00 00 47 45 54 20 2F 72 61 6E 64 6F 6D 2F 74 72 69 76 69 61 20 48
54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 6E 75 6D 62 65 72 73 61 70 69 2E 63
6F 6D 0D 0A 43 6F 6E 6E 65 63 74 69 6F 6E 3A 20 63 6C 6F 73 65 0D 0A 0D 0A B6
```

Receive TX Status

Extended sockets use the existing TX Status frame (0x89) to report acceptance of the data for transmit.

Field	Value
Frame type	0x89 (TX Status)
Frame ID	0x01
Status	0x00 (Success)

TX Status received from XBee data:

```
7E 00 03 89 01 00 75
```

Receive one or more Receive Data frames

The server will respond with an interesting fact about a number. The following information is a sample response. Multiple frames may be needed to contain the full response content depending on size and network conditions.

Field	Value
Frame type	0xCD (Socket Receive)
Frame ID	0x00
Socket ID	0x00
Status	0x00
Payload	HTTP/1.1 200 OK Server: nginx/1.4.6 (Ubuntu) Date: Thu, 18 Jul 2019 16:13:47 GMT Content-Type: text/plain; charset="UTF-8"; charset=utf-8 Content-Length: 53 Connection: close X-Powered-By: Express Access-Control-Allow-Origin: * Access-Control-Allow-Headers: X-Requested-With X-Numbers-API-Number: 270 X-Numbers-API-Type: trivia Pragma: no-cache Cache-Control: no-cache Expires: 0 270 is the average number of days in human pregnancy.

Receive Data received from XBee containing web service response:

```
7E 01 C5 CD 00 00 00 48 54 54 50 2F 31 2E 31 20 32 30 30 20 4F 4B 0D 0A 53 65 72
76 65 72 3A 20 6E 67 69 6E 78 2F 31 2E 34 2E 36 20 28 55 62 75 6E 74 75 29 0D 0A
44 61 74 65 3A 20 54 68 75 2C 20 31 38 20 4A 75 6C 20 32 30 31 39 20 31 36 3A 31
33 3A 34 37 20 47 4D 54 0D 0A 43 6F 6E 74 65 6E 74 2D 54 79 70 65 3A 20 74 65 78
74 2F 70 6C 61 69 6E 3B 20 63 68 61 72 73 65 74 3D 22 55 54 46 2D 38 22 3B 20 63
68 61 72 73 65 74 3D 75 74 66 2D 38 0D 0A 43 6F 6E 74 65 6E 74 2D 4C 65 6E 67 74
68 3A 20 35 33 0D 0A 43 6F 6E 6E 65 63 74 69 6F 6E 3A 20 63 6C 6F 73 65 0D 0A 58
2D 50 6F 77 65 72 65 64 2D 42 79 3A 20 45 78 70 72 65 73 73 0D 0A 41 63 63 65 73
73 2D 43 6F 6E 74 72 6F 6C 2D 41 6C 6C 6F 77 2D 4F 72 69 67 69 6E 3A 20 2A 0D 0A
41 63 63 65 73 73 2D 43 6F 6E 74 72 6F 6C 2D 41 6C 6C 6F 77 2D 48 65 61 64 65 72
73 3A 20 58 2D 52 65 71 75 65 73 74 65 64 2D 57 69 74 68 0D 0A 58 2D 4E 75 6D 62
65 72 73 2D 41 50 49 2D 4E 75 6D 62 65 72 3A 20 32 37 30 0D 0A 58 2D 4E 75 6D 62
65 72 73 2D 41 50 49 2D 54 79 70 65 3A 20 74 72 69 76 69 61 0D 0A 50 72 61 67 6D
61 3A 20 6E 6F 2D 63 61 63 68 65 0D 0A 43 61 63 68 65 2D 43 6F 6E 74 72 6F 6C 3A
20 6E 6F 2D 63 61 63 68 65 0D 0A 45 78 70 69 72 65 73 3A 20 30 0D 0A 0D 0A 32 37
30 20 69 73 20 74 68 65 20 61 76 65 72 61 67 65 20 6E 75 6D 62 65 72 20 6F 66 20
64 61 79 73 20 69 6E 20 68 75 6D 61 6E 20 70 72 65 67 6E 61 6E 63 79 2E 8B
```

Receive Socket Status indicating closed connection

Finally, due to the "Connection" header in the request, the server should remotely close the connection.

Field	Value
Frame type	0xCF (TX Status)
Socket ID	0x00
Status	0x07 (Connection lost)

Example Socket Status received from XBee indicating connection lost:

```
7E 00 03 CF 00 07 29
```

When Socket Status indicating a connection close is received, the socket ID will have been de-allocated by the XBee and no further operations are possible or necessary using that ID.

Extended Socket example: UDP

UDP is connection-less, so this example demonstrates that a Socket Connect frame is not required to begin communication and that multiple peers can be used with a single socket.

Send a Socket Create frame

Field	Value
Frame type	0x40 (Socket Create)
Frame ID	0x01
Protocol	0x00 (UDP)

UDP Socket Create frame data:

```
7E 00 03 40 01 00 BE
```

Receive a Socket Create response

Field	Value
Frame type	0xC0 (Socket Create Response)
Frame ID	0x01
Socket ID	0x00
Status	0x00 (Success)

Socket Create Response received from XBee:

```
7E 00 04 C0 01 00 00 3E
```

Bind local source addres

The bind/listen operation is necessary prior to transmit in order to assign a known source address to all data sent from this socket.

Field	Value
Frame type	0x46 (Socket Bind/Listen)
Frame ID	0x01
Socket ID	0x00
Source Port	0x12 0x34

Socket Bind/Listen frame data:

```
7E 00 05 46 01 00 12 34 72
```

Receive Bind/Listen Response

The XBee generates a response indicating the status of the request to bind the requested port.

Field	Value
Frame type	0xC6 (Socket Bind/Listen Response)
Frame ID	0x01
Socket ID	0x00
Status	0x00 (Success)

Socket Bind/Listen Response received from XBee:

```
7E 00 04 C6 01 00 00 38
```

Send to Digi echo server

Digi hosts a server at 52.43.121.77 port 10001 which echos all UDP traffic sent to it.

Field	Value
Frame type	0x45 (Socket SendTo)
Frame ID	0x01
Socket ID	0x00
Destination Address	0x34 0x2B 0x79 0x4D (52.43.121.77)
Destination Port	0x27 0x11 (decimal 10001)
Transmit Options	0x00
Payload	echo this

Socket SendTo frame data:

```
7E 00 13 45 01 00 34 2B 79 4D 27 11 00 65 63 68 6F 20 74 68 69 73 E5
```

Receive TX Status

Extended sockets use the existing TX Status frame (0x89) to report acceptance of the data for transmit.

Field	Value
Frame type	0x89 (TX Status)
Frame ID	0x01
Status	0x00 (Success)

TX Status received from XBee:

```
7E 00 03 89 01 00 75
```

Receive echoed data

When the response from the server is sent back, the XBee provides it using a Socket Receive From frame.

Field	Value
Frame type	0xCE (Socket Receive From)

Field	Value
Frame ID	0x00
Socket ID	0x00
Source address	0x34 0x2B 0x79 0x4D (52.43.121.77)
Source Port	0x27 0x11 (decimal 10001)
Status	0x00 (Success)
Payload	echo this

Socket ReceiveFrom received from XBee, containing echoed data:

```
7E 00 13 CE 00 00 34 2B 79 4D 27 11 00 65 63 68 6F 20 74 68 69 73 5D
```

Send to Digi time server

Digi hosts a server at 54.43.121.77 port 10002 which will reply with the time when it receives a packet.

Field	Value
Frame type	0x45 (Socket SendTo)
Frame ID	0x01
Socket ID	0x00
Destination Address	0x34 0x2B 0x79 0x4D (52.43.121.77)
Destination Port	0x27 0x12 (decimal 10002)
Transmit Options	0x00
Payload	0x20 (ASCII space, any value should do)

Socket SendTo time server frame data:

```
7E 00 0B 45 01 00 34 2B 79 4D 27 12 00 20 3B
```

Receive TX Status

This is exactly the same as the previous transmission to the echo server on success.

Receive daytime value

When the response from the server is sent back, the XBee will provide it using a Socket Receive From frame.

Field	Value
Frame type	0xCE (Socket Receive From)
Frame ID	0x00
Socket ID	0x00
Source address	0x34 0x2B 0x79 0x4D (52.43.121.77)
Source Port	0x27 0x12 (decimal 10002)
Status	0x00 (Success)
Payload	<current UTC time>

Socket Receive From frame received from XBee containing time data:

```
7E 00 1E CE 00 00 34 2B 79 4D 27 12 00 32 30 31 39 2D 30 37 2D 31 38 20 31 38 3A
35 32 3A 34 33 0A 08
```

Close the socket

When the socket is no longer needed it should be closed to return resources to the system.

Field	Value
Frame type	0x43 (Socket Close)
Frame ID	0x01
Status	0x00

Socket Close frame data:

```
7E 00 03 43 01 00 BB
```

Receive close response

Finally, the XBee indicates the socket has been closed with a Socket Close Response frame.

Field	Value
Frame type	0xC3 (Socket CloseResponse)
Frame ID	0x01
Socket ID	0x00
Status	0x00 (Success)

Socket Close Response received from XBee:

```
7E 00 04 C3 01 00 00 3B
```

Extended Socket example: TCP Listener

The following example demonstrates setting up a TCP listener on the XBee Cellular and interacting with incoming connections. It will open up a listener socket on a given port and then receive data from a client.

Note The module must either have a public IP or a be on a private network in order to be accessible as a server (listener).

Send a Socket Create frame

Note The XBee Cellular does not support incoming TLS sockets.

Field	Value
Frame type	0x40 (Socket Create)
Frame ID	0x01
Protocol	0x01 (TCP)

Socket Create frame data:

```
7E 00 03 40 01 01 BD
```

Receive a Socket Create response

The response contains the socket ID assigned. This example assumes zero.

Field	Value
Frame type	0xC0 (Socket Create Response)
Frame ID	0x01
Socket ID	0x00
Status	0x00 (Success)

Socket Create Response received from XBee:

```
7E 00 04 C0 01 00 00 3E
```

Designate the socket as a listener

The Socket Bind/Listen Frame takes the socket ID from the socket create response and a source port that the socket will then listen on. In this example port 10001 is used.

Field	Value
Frame type	0x46 (Socket Listen)
Frame ID	0x01
Socket ID	0x00
Source Port	0x2711 (10001)

Socket Bind/Listen frame data:

```
7E 00 05 46 01 00 27 11 80
```

Receive a Socket Bind/Listen Response

The Socket Bind/Listen Response contains a Status. A Status of zero is a success and any other value is an error.

Field	Value
Frame type	0xC6 (Socket Listen)
Frame ID	0x01
Socket ID	0x00
Status	0x00 (Success)

Socket Bind/Listen frame received from XBee:

```
7E 00 04 C6 01 00 00 38
```

Making a connection to the listener socket

The IP of the XBee can be acquired through the MY at command.

```
ATMY
172.20.1.235
```

Using an external tool like netcat, a connection can be made to the given address.

```
nc -p 10001 172.20.1.235 10001
Hello XBee!
```

After the connection has been made, the XBee outputs a Socket New IPv4 Client frame indicating the presence of a new client connection. It contains the listener's socket ID and the new Client Socket ID along with the connection's remote address information.

Field	Value
Frame type	0xCC (Socket New IPv4 Client)

Field	Value
Socket ID	0x00
Client Socket ID	0x01
Remote Address	0x0A 0x0A 4A 9D
Remote Port	0x27 0x11

Socket New IPv4 Client frame:

```
7E 00 09 CC 00 01 0A 0A 4A 9D 27 11 FF
```

Note XBee Cellular Cat-1 variants require data to be sent before the connection is presented. Other variants present the connection as soon as it is made.

Receiving Data from the new socket

After the connection is established, data received from the new socket is contained in a Socket Receive frame just like any other TCP socket.

Field	Value
Frame type	0xCD (Socket Status)
Frame ID	0x01
Socket ID	0x01
Status	0x00
Payload	Hello XBee!

Receive Data indicating data from remote TCP peer:

```
7E 00 10 CD 00 01 00 48 65 6C 6C 6F 20 58 42 65 65 21 0A 8E
```

Receive a Socket Status indicating closed connection

You may close the client socket remotely which elicits a Socket Status with a Status of 0x07.

Field	Value
Frame type	0xCF (Socket Status)
Socket ID	0x01
Status	0x07 (Connection lost)

Socket Status received from XBee indicating connection lost:

```
7E 00 03 CF 01 07 28
```

When a Socket Status indicating a connection close is received, the socket ID will have been de-allocated by the XBee and no further operations are possible or necessary using that ID.

Transport Layer Security (TLS)

For detailed information about using MicroPython on the XBee Cellular Modem refer to the *Digi MicroPython Programming Guide*.

TLS AT commands

These AT commands, when used together, let you interact with TLS features: ATFS (File System), TL (TLS Protocol Version), IP (IP Protocol), $0 (TLS Profile 0), $1 (TLS Profile 1), and $2 (TLS Profile 2). The format of the **$** commands is:

AT$<num>[<ca_cert>];[<client_cert>];[<client_key>]

Where:

- **num**: Profile index. Index zero is used for Transparent mode connections and TLS connections using Transmit (TX) Request: IPv4 - 0x20.
- **ca_cert**: (optional) Filename of a file in the **certs/** directory. Indicates the certificate identifying a trusted root certificate authority (CA) to use in validating servers. If **ca_cert** is empty the server certificate will not be authenticated. This must be a single root CA certificate. The modules do not allow a non-self signed certificate to work, so intermediate CAs are not enough.
- **client_cert**: (optional) Filename of a file in the **certs/** directory. Indicates the certificate presented to servers when requested for client authentication. If **client_cert** is empty no certificate is presented to the server should it request one. This may result in mutual authentication failure.
- **client_key**: (optional) Filename of a file in the **certs/** directory. Indicates the private key matching the public key contained in **client_cert**. This should be a secure file uploaded with ATFS XPUT filename. This should always be provided if **client_cert** is provided and match the certificate or client authentication will fail.

The default value is "**;;**". This default value preserves the legacy behavior by allowing the creation of encrypted connections that are confidential but not authenticated.

To specify a key stored outside of **certs/**, you can either use a relative path, for example **../server.pem** or an absolute path starting with **/flash**, for example **/flash/server.pem**. Both examples refer to the same file.

It is not an error at configuration time to name a file that does not yet exist. An error is generated if an attempt to create a TLS connection is made with improper settings.

- Files specified should all be in PEM format, not DER.
- Upload private keys securely with ATFS XPUT filename.
- Certificates can be uploaded with ATFS PUT filename as they are not sensitive. It is not possible to use ATFS GET filename to **GET** them if they have been securely uploaded.

To authenticate a server not participating in a public key infrastructure (PKI) using CAs, the server must present a self-signed certificate. That certificate can be used in the **ca_cert** field to authenticate that single server.

There are effectively three levels of authentication provided depending on the parameters provided

1. No authentication: None of the parameters are provided, this is the default value. With this configuration identity is not validated and a man in the middle (MITM) attack is possible.
2. Server authentication: Only **ca_cert** is provided. Only the servers identity is checked
3. Mutual authentication: All items are provided and both sides are assured of the identity of their peer

It is not possible to only have client authentication.

Transparent mode and TLS

Transparent mode connections made when IP (IP Protocol) = **4** (TLS) are made using the configuration specified by $0 (TLS Profile 0).

API mode and TLS

On the Transmit (TX) Request: IPv4 - 0x20 frame, when you specify protocol **4** (TLS), the profile configuration specified by $0 (TLS Profile 0) is used to form the TLS connection. Tx Request with TLS Profile - 0x23 lets you choose the IP setting for the serial data.

Key formats

The RSA PKCS#1 format is the only common format across XBee Cellular device variants. You can identify a PKCS#1 key file by the presence of **BEGIN RSA PRIVATE KEY** in the file header.

Digi's implementation does not support encrypted keys, we use file system encryption to protect the keys at rest in the system.

Certificate limitations

The XBee Cellular Modem only supports certificate files that contain a single certificate in them.

The implications of this are:

- For client certificate files (for example when client authentication is required):
  - Self-signed certificates will work.
  - Certificates signed by the root CA will work, because the root CA can be omitted per RFC 5246. The root certificate authority may be omitted from the chain, under the assumption that the remote end must already possess it in order to validate it in any case.
  - Certificate chains that include a intermediate CA are problematic. To work around this the client's certificate chain has to be supplied to the server outside of the connection.
- For server certificate files (when server authentication is required) this is not a problem unless the client is expected to connect to multiple servers that are using different self signed certificates or are using certificate chains that are signed by different root CA certificates. To work around this you have to change the certificates before making the connection, or in the case of API mode specify a different authentication profile.

Cipher suites

The only documented shared suites between the XBee3 Cellular LTE Cat 1 Smart Modem and the XBee3 Cellular LTE-M Global Smart Modem are:

- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA

For the Telit LE866 cellular component:

- TLS_RSA_WITH_RC4_128_MD5
- TLS_RSA_WITH_RC4_128_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_NULL_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA

This list may be incomplete.

Server Name Indication (SNI)

We do not currently support SNI. Therefore servers which use SNI to present certificates based on client provided host data may be unable to establish the expected connections.

Secure the connection between an XBee and Remote Manager with server authentication

The XBee devices that have the x11 or later version of the firmware installed are by default able to secure the TLS connection to Digi Remote Manager. The default configuration provides confidentiality of the communication but is not able to authenticate the server without a certificate being provided.

If you have devices that have been upgraded in the field or manufactured prior to being pre-populated with the Remote Manager certificate, you should follow the procedure below to add the necessary certificate if server authentication is needed.

Step 1: Get the certificate

1. Navigate to the **Firmware Updates** section of the Digi XBee Cellular LTE CAT 1 Verizon support page.
2. Click **Remote Manager TLS Public Certificate** to download the certificate .zip file.
3. Unzip the .zip file.
4. Calculate the SHA-256 hash to verify that the file is correct. The correct file will have an SHA-256 hash of:

   33d91e18668b0d8a9ec59c5f9f312c53ca2884adaa62337839e5495c26d2d64c

Step 2: Configure device

You should confirm that the default settings are correct. You can use either Remote Manager or XCTU to verify these settings and place the certificate file in the correct location.

1. Verify the following settings:

Setting	Value
DO	Bit 0 (mask 0x1) must be set. This enables the use of Digi Remote Manager within the firmware.

Setting	Value
MO	Bit 1 (mask 0x2) must be set. When this value is set the Remote Manager TCP connection will be secured with TLS.
$D	By default will contain the value */flash/cert/digi-remote-mgr.pem*. This is the file system location where the firmware will look for the certificate to use.

2. Use XCTU or Remote Manager to place the downloaded and unzipped certificate file in the location specified in the **$D** command.

Step 3: Verify that authentication is being performed

The next TCP connection to Remote Manager should only succeed if the server can be authenticated using the provided certificate. You can confirm that the server has been authenticated.

1. Cause an active connection to Remote Manager. For example, you could set bit 0 for the **MO** command. Make sure that you do not clear bit 1.
2. After a short wait you should be able to see the device as connected in Remote Manager.
    a. Log in to Remote Manager.
    b. Click **Device Management**.
    c. Locate the device in the device list and verify that the connection icon in the left column is blue and the hover tool tip says "Connected".
3. When the device is connected to Remote Manager, the **DI** command can take on any of the three values shown below, based on the security level of the connection. Verify the that the **DI** command is set to **6** to verify that the server was correctly authenticated.
    - **0**: Connected without TLS
    - **5**: Connected with TLS but without authentication
    - **6**: Connected with TLS and with authentication

AT commands

Special commands

The following commands are special commands.

AC (Apply Changes)

Immediately applies new settings without exiting Command mode.

Applying changes means that the device re-initializes based on changes made to its parameter values. Once changes are applied, the device immediately operates according to the new parameter values.

This behavior is in contrast to issuing the **WR** (Write) command. The **WR** command saves parameter values to non-volatile memory, but the device still operates according to previously saved values until the device is rebooted or you issue the **CN** (Exit AT Command Mode) or **AC** commands.

Parameter range

N/A

Default

N/A

FR (Force Reset)

Resets the device. The device responds immediately with an **OK** and performs a reset 100 ms later.

If you issue **FR** while the device is in Command Mode, the reset effectively exits Command mode.

Note We recommend entering Airplane mode before resetting or rebooting the device to allow the cellular module to detach from the network.

Parameter range

N/A

Default

N/A

RE command

Restore device parameters to factory defaults.

The **RE** command does not write restored values to non-volatile (persistent) memory. Issue the **WR** (Write) command after issuing the **RE** command to save restored parameter values to non-volatile memory.

Parameter range

N/A

Default

N/A

SD (Shutdown)

Shuts down the device. When the shut down process is complete, the device returns **OK**. After the device responds **OK**, you can safely remove power from the device.

If the radio can't be fully shut down within two minutes, the device returns **ERROR**.

You can verify the state of the device using the AI command. After you issue the **SD** command and a response has been returned (either **OK** or **ERROR**), issue the **AI** command. If the shutdown was successful, **2D** is returned.

Parameter range

N/A

Default

N/A

WR (Write)

Writes parameter values to non-volatile memory so that parameter modifications persist through subsequent resets.

Note Once you issue a **WR** command, do not send any additional characters to the device until after you receive the **OK** response.

Parameter range

N/A

Default

N/A

HI (Hardware Identity)

Returns a hexadecimal value that indicates the hardware identity of the module. You can use this command to determine the feature availability on the specific hardware.

Parameter range

0 - 3

Value	Description
3	If the value returned is 3 then the hardware is compatible with the connected sleep feature. If any other value is returned, the connected sleep feature cannot be used on the device.

Default

N/A

Cellular commands

The following AT commands are cellular configuration and data commands.

PH (Phone Number)

Reads the SIM card phone number.

If **PH** is blank, the XBee Cellular Modem is not registered to the network.

Parameter range

N/A

Default

Set by the cellular carrier via the SIM card

S# (ICCID)

Reads the Integrated Circuit Card Identifier (ICCID) of the inserted SIM.

Parameter range

N/A

Default

Set by the SIM card

IM (IMEI)

Reads the device's International Mobile Equipment Identity (IMEI).

Parameter range

N/A

Default

Set in the factory

II (Subscriber identity)

Reads the IMSI (International Mobile Subscriber Identity) from the SIM inserted into the module.

Parameter range

N/A

Default

N/A

MN (Operator)

Reads the network operator on which the device is registered.

Parameter range

N/A

Default

Verizon

MV (Modem Firmware Version)

Read the firmware version string for cellular component communications. See the related VR (Firmware Version) command.

Parameter range

N/A

Default

Set in the currently loaded firmware

MU (Modem firmware revision number)

Read the firmware revision number of the cellular component. See the related MV (Modem Firmware Version) command.

Parameter range

N/A

Default

Set in the currently loaded firmware

DB (Cellular Signal Strength)

Reads the absolute value of the current signal strength to the cell tower in dB. If **DB** is blank, the XBee Cellular Modem has not received a signal strength from the cellular component.

DB only updates when the modem is registered with the celllular tower. It is updated periodically, and not when read.

Parameter range

0x71 - 0x33 (-113 dBm to -51 dBm) [read-only]

Default

N/A

AN (Access Point Name)

Specifies the packet data network that the modem uses for Internet connectivity. This information is provided by your cellular network operator. After you set this value, applying changes with AC (Apply Changes) or CN (Exit Command mode) triggers a network reset.

In order to meet network requirements, on Verizon 4G, the APN value in the cellular component is only changed when **AN** has been run (with the same or a different value) and changes are applied.

When you change APN and after you send **AC**, wait for **AI** to return **0**, and for OA (Operating APN) to return the APN that you set.

Hyphen (**-**) means no APN is being specified. On Verizon 4G, this leaves the APN in the cellular component alone. On 3G Global, this configures the cellular component to use an APN supplied by the network. This depends on your service plan.

Some common APN values are:

Value	Description
WYLESLTE.GW7.VZWENTP	KORE SIMS in the evaluation kit
VZWINTERNET	Standard Verizon SIMS

Parameter range

1 - 100 ASCII characters

Default

-

AM (Airplane Mode)

When set, the cellular component of the XBee Cellular Modem is fully turned off and no access to the cellular network is performed or possible.

Parameter range

0 - 1

0 = Normal operation

1 = Airplane mode

Default

0

OA (Operating APN)

Reads the APN value currently configured in the cellular component.

Parameter range

ASCII characters

Default

N/A

DV (Secondary Antenna Function Switch)

Set and read the secondary antenna function setting of the cellular component. When enabled, the cellular component uses both antennas to improve receive sensitivity.

This setting is applied only while the XBee Cellular Modem is initializing the cellular component. After changing this setting, you must:

1. Use WR (Write) to write all values to flash.
2. Use FR (Force Reset) to reset the device.
3. Wait for the cellular component to be initialized: AI (Association Indication) reaches **0x00**.
4. Use **FR** to reset the device a second time.
5. Wait again for the cellular component to initialize: **AI** reaches **0x00**.

Parameter range

0 - 1

Bit	Description
0	The secondary antenna is unused.
1	The cellular component uses the secondary antenna to improve received sensitivity. This is the default setting.

Default

1

Network commands

The following commands are network commands.

IP (IP Protocol)

Sets or displays the IP protocol used for client and server socket connections in IP socket mode.

Parameter range

0 - 4

Value	Description
0x00	UDP
0x01	TCP
0x02	SMS
0x03	Reserved
0x04	TLS over TCP communication

Default

0x01

TL (TLS Protocol Version)

Sets the TLS protocol version used for the TLS socket. If you change the **TL** value, it does not affect any currently open sockets. The value only applies to subsequently opened sockets.

Note Due to known vulnerabilities in prior protocol versions, we strongly recommend that you use the latest TLS version whenever possible.

Range

Value	Description
0x00	SSL v3
0x01	TLS v1.0
0x02	TLS v1.1
0x03	TLS v1.2

Default

0x03

$0 (TLS Profile 0)

Specifies the TLS certificate(s) to use in Transparent mode (when IP (IP Protocol) = **4**) or API mode (Transmit (TX) Request: IPv4 - 0x20 or Tx Request with TLS Profile - 0x23 with profile set to **0**).

Format

server_cert;client_cert;client_key

Parameter range

From 1 through 127 ASCII characters.

Default

N/A

$1 (TLS Profile 1)

Specifies the TLS certificate(s) to use for Tx Request with TLS Profile - 0x23 transmissions with profile set to **1**.

Format

server_cert;client_cert;client_key

Parameter range

From 1 through 127 ASCII characters.

Default

N/A

$2 (TLS Profile 2)

Specifies the SSL/TLS certificate(s) to use in Transparent mode (when IP (IP Protocol) = **4**) or API mode (Transmit (TX) Request: IPv4 - 0x20 or Tx Request with TLS Profile - 0x23 with profile set to **0**).

Format

server_cert;client_cert;client_key

Parameter range

From 1 through 127 ASCII characters.

Default

N/A

TM (IP Client Connection Timeout)

The IP client connection timeout. If there is no activity for this timeout then the connection is closed. If **TM** is **0**, the connection is closed immediately after the device sends data.

If you change the **TM** value while in Transparent Mode, the current connection is immediately closed. Upon the next transmission, the **TM** value applies to the newly created socket.

If you change the **TM** value while in API Mode, the value only applies to subsequently opened sockets.

Parameter range

0 - 0xFFFF [x 100 ms]

Default

0xBB8 (5 minutes)

TS (IP Server Connection Timeout)

The IP server connection timeout. If no activity for this timeout then the connection is closed. When set to **0** the connection is closed immediately after data is sent.]

Parameter Range

10 - 0xFFFF; (x 100 ms)

Default

3000

DO (Device Options)

Enables and disables special features on the XBee Cellular Modem.

Bit 0 - Remote Manager support

If the XBee Cellular Modem cannot establish a connection with Remote Manager, it waits 30 seconds before trying again. On each successive connection failure, the wait time doubles (60 seconds, 120, 240, and so on) up to a maximum of 1 hour. This time resets to 30 seconds once the connection to Remote Manager succeeds or if the device is reset.

Bits 1 - 7

Reserved

Range

0x00 - 0x03

Bitfield

Bit	Description
0x00	Enable Remote Manager support
0x01 - 0x07	Reserved for future use

Default

0x01 (Bit 0 enabled)

DT (Cellular Network Time)

Reads the current network-provided local time of the XBee device, as reported by the cellular tower.

If the time is not known, the response is empty. This value is synchronized with the network approximately once per hour.

Parameter range

0 - 1

Value	Description
0	The response is the number of seconds since 2000-01-01 00:00:00, as a 32-bit number. This is the default.
1	The response is the current date and time in ISO 8601 format. For example, "2018-12-25T22:00:05".

Note You can also send **DT**, which acts like **DT=0**.

Default

0

Addressing commands

The following AT commands are addressing commands.

SH (Serial Number High)

The upper digits of the unique International Mobile Equipment Identity (IMEI) assigned to this device.

Parameter range

0 - 0xFFFFFFFF [read-only]

Default

N/A

SL (Serial Number Low)

The lower digits of the unique International Mobile Equipment Identity (IMEI) assigned to this device.

Parameter range

0 - 0xFFFFFFFF [read-only]

Default

N/A

MY (Module IP Address)

Reads the device's IP address. This command is read-only because the IP address is assigned by the mobile network.

In API mode, the address is represented as the binary four byte big-endian numeric value representing the IPv4 address.

In Transparent or Command mode, the address is represented as a dotted-quad string notation.

Parameter range

0- 15 IPv4 characters

Default

0.0.0.0

P# (Destination Phone Number)

Sets or displays the destination phone number used for SMS when IP (IP Protocol) = **2**. Phone numbers must be fully numeric, 7 to 20 ASCII digits, for example: 8889991234.

P# allows international numbers with or without the **+** prefix. If you omit **+** and are dialing internationally, you need to include the proper International Dialing Prefix for your calling region, for example, 011 for the United States.

Range

7 - 20 ASCII digits including an optional **+** prefix

Default

N/A

N1 (DNS Address)

Displays the IPv4 address of the primary domain name server.

Parameter Range

Default

0.0.0.0 (waiting on cellular connection)

N2 (DNS Address)

Displays the IPv4 address of the secondary domain name server.

Parameter Range

Default

0.0.0.0 (waiting on cellular connection)

DL (Destination Address)

The destination IPv4 address or fully qualified domain name.

To set the destination address to an IP address, the value must be a dotted quad, for example **XXX.XXX.XXX.XXX**.

To set the destination address to a domain name, the value must be a legal Internet host name, for example **remotemanager.digi.com**

Parameter range

0 - 128 ASCII characters

Default

0.0.0.0

OD (Operating Destination Address)

Read the destination IPv4 address currently in use by Transparent mode. The value is **0.0.0.0** if no Transparent IP connection is active.

In API mode, the address is represented as the binary four byte big-endian numeric value representing the IPv4 address.

In Transparent or Command mode, the address is represented as a dotted-quad string notation.

Parameter range

-

Default

0.0.0.0

DE (Destination port)

Sets or displays the destination IP port number.

This command reads all input as hexadecimal. All values must be entered in hexadecimal with no leading 0x. For example, the destination port 9001 has the hexadecimal value of 0x2329. The command would be entered as **ATDE 2329**.

Parameter range

0x0 - 0xFFFF

Default

0x2616

C0 (Source Port)

Set or get the port number used to provide the serial communication service. Data received by this port on the network is transmitted on the XBee Cellular Modem's serial port.

As long as a network connection is established to this port (for TCP) data received on the serial port is transmitted on the established network connection.

IP (IP Protocol) sets the protocol used when UART is in Transparent or API mode.

For more information on using incoming connections, see Socket behavior.

Parameter range

0 - 0xFFFF

Value	Description
0	Disabled
Non-0	Enabled on that port

Default

0

LA (Lookup IP Address of FQDN)

Performs a DNS lookup of the given fully qualified domain name (FQDN) and outputs its IP address.

When you issue **LA** in API mode, the IP address is formatted in binary four byte big-endian numeric value. In all other cases (for example, Command mode) the format is dotted decimal notation.

Range

Valid FQDN

Default

-

Serial interfacing commands

The following AT commands are serial interfacing commands.

BD (Baud Rate)

Sets or displays the serial interface baud rate for communication between the device's serial port and the host.

Modified interface baud rates do not take effect until the XBee Cellular Modem exits Command mode or you issue AC (Apply Changes). The baud rate resets to default unless you save it with WR (Write) or by clicking the **Write module settings** button in XCTU.

Parameter range

Standard baud rates: 0x1 - 0xA

Non-standard baud rates: 0x5B9 to 0x5B8D80 (up to 6 Mb/s)

Parameter	Description
0x1	2400 b/s
0x2	4800 b/s
0x3	9600 b/s
0x4	19200 b/s
0x5	38400 b/s
0x6	57600 b/s
0x7	115200 b/s
0x8	230400 b/s
0x9	460800 b/s
0xA	921600 b/s

Default

0x3 (9600 b/s)

NB (Parity)

Set or read the serial parity settings for UART communications.

Parameter range

0x00 - 0x02

Parameter	Description
0x00	No parity
0x01	Even parity
0x02	Odd parity

Default

0x00

SB (Stop Bits)

Sets or displays the number of stop bits for UART communications.

Parameter range

0 - 1

Parameter	Configuration
0	One stop bit
1	Two stop bits

Default

0

RO (Packetization Timeout)

Set or read the number of character times of inter-character silence required before transmission begins when operating in Transparent mode.

RF transmission also starts after the maximum packet size for the selected protocol is received in the UART receive buffer.

Set **RO** to **0** to transmit characters as they arrive instead of buffering them into one RF packet.

Parameter range

0 - 0xFF (x character times)

Default

3

TD (Text Delimiter)

The ASCII character used as a text delimiter for Transparent mode. When you select a character, information received over the serial port in Transparent mode is not transmitted until that character is received. To use a carriage return, set to **0xD**. Set to zero to disable text delimiter checking.

Parameter range

0 - 0xFF

Default

0x0

FT (Flow Control Threshold)

Set or display the flow control threshold.

The device de-asserts $\overline{\text{CTS}}$ when **FT** bytes are in the UART receive buffer.

Parameter range

0x9D - 0x82D

Default

0x681

AP (API Enable)

Enables the frame-based application programming interface (API) mode.

The API mode setting. The device can format the RF packets it receives into API frames and send them out the UART. When API is enabled the UART data must be formatted as API frames because Transparent mode is disabled. See Modes for more information.

Parameter range

0x00 - 0x05

Parameter	Description
0x00	API disabled (operate in Transparent mode)
0x01	API enabled
0x02	API enabled (with escaped control characters)
0x03	N/A
0x04	MicroPython REPL
0x05	Bypass mode

Default

0

I/O settings commands

The following AT commands are I/O settings commands.

D0 (DIO0/AD0)

Sets or displays the DIO0/AD0 configuration (pin 20).

Parameter range

0, 2 - 5

Parameter	Description
0	Disabled
1	N/A
2	Analog input
3	Digital input
4	Digital output, default low
5	Digital output, default high

Default

0

D1 (DIO1/AD1)

Sets or displays the DIO1/AD1 configuration (pin 19).

Parameter range

0 - 6

Parameter	Description
0	Disabled
1	SPI_$\overline{\text{ATTN}}$
2	ADC
3	Digital input
4	Digital output, low
5	Digital output, high
6	I2C SCL

Default

0

D2 (DIO2/AD2)

Sets or displays the DIO2/AD2 configuration (pin 18).

Parameter range

0 - 5

	Description
0	Disabled
1	SPI_CLK
2	Analog input
3	Digital input
4	Digital output, default low
5	Digital output, default high

Default

0

D3 (DIO3/AD3)

Sets or displays the DIO3/AD3 configuration (pin 17).

Parameter range

0 - 5

Parameter	Description
0	Disabled
1	SPI_$\overline{\text{SSEL}}$
2	Analog input
3	Digital input
4	Digital output, default low
5	Digital output, default high

Default

0

D4 (DIO4)

Sets or displays the DIO4 configuration (pin 11).

Parameter range

0, 1, 3 - 5

Parameter	Description
0	Disabled
1	SPI_MOSI
2	N/A
3	Digital input
4	Digital output, default low
5	Digital output, default high

Default

0

D5 (DIO5/ASSOCIATED_INDICATOR)

Sets or displays the DIO5/ASSOCIATED_INDICATOR configuration (pin 15).

Parameter range

0, 1, 3 - 5

Parameter	Description
0	Disabled
1	Associated LED
2	N/A
3	Digital input
4	Digital output, default low
5	Digital output, default high

Default

1

D6 (DIO6/RTS)

Sets or displays the DIO6/$\overline{\text{RTS}}$ configuration (pin 16).

Parameter range

0, 1, 3 - 5

Parameter	Description
0	Disabled
1	$\overline{\text{RTS}}$ flow control

Parameter	Description
2	N/A
3	Digital input
4	Digital output, default low
5	Digital output, default high

Default

0

D7 (DIO7/CTS)

Sets or displays the DIO7/$\overline{\text{CTS}}$ configuration (pin 12).

Parameter range

0, 1, 3 - 5

Parameter	Description
0	Disabled
1	$\overline{\text{CTS}}$ flow control
2	N/A
3	Digital input
4	Digital output, default low
5	Digital output, default high

Default

0x1

D8 (DIO8/SLEEP_REQUEST)

Sets or displays the DIO8/$\overline{\text{DTR}}$/SLP_RQ configuration (pin 9).

Parameter range

0, 1, 3 - 5

Parameter	Description
0	Disabled
1	SLEEP_REQUEST input
3	Digital input
4	Digital output, default low
5	Digital output, default high

Default

1

D9 (DIO9/ON_SLEEP)

Sets or displays the DIO9/ON_$\overline{\text{SLEEP}}$ configuration (pin 13).

Parameter range

0, 1, 3 - 5

Parameter	Description
0	Disabled
1	ON/$\overline{\text{SLEEP}}$ output
3	Digital input
4	Digital output, default low
5	Digital output, default high

Default

1

P0 (DIO10/PWM0 Configuration)

Sets or displays the PWM/DIO10 configuration (pin 6).

This command enables the option of translating incoming data to a PWM so that the output can be translated back into analog form.

Parameter range

0 - 5

Parameter	Description
0	Disabled
1	RSSI PWM0 output
2	PWM0 output
3	Digital input
4	Digital output, low
5	Digital output, high

Default

0

P1 (DIO11/PWM1 Configuration)

Sets or displays the DIO11 configuration (pin 7).

Parameter range

0, 1, 3 - 6

Parameter	Description
0	Disabled
1	Fan enable. Output is low when the XBee Cellular Modem is sleeping, turning an attached fan off when the cellular component is in a power saving mode, and also during Airplane Mode
3	Digital input
4	Digital output, default low
5	Digital output, default high
6	I2C SDA

Default

0

P2 (DIO12 Configuration)

Sets or displays the DIO12 configuration (pin 4).

Parameter range

0, 1, 3 - 5

Parameter	Description
0	Disabled
1	SPI_MISO
2	N/A
3	Digital input
4	Digital output, default low
5	Digital output, default high

Default

0

PD (Pull Direction)

The resistor pull direction bit field (**1** = pull-up, **0** = pull-down) for corresponding I/O lines that are set by PR (Pull-up/down Resistor Enable).

If the bit is not set in **PR**, the device uses **PD**.

Note Resistors are not applied to disabled lines.

See PR (Pull-up/down Resistor Enable) for bit mappings, which are the same.

Parameter range

0x0 – 0x7FFF

Default

0 – 0x7FFF

PR (Pull-up/down Resistor Enable)

Sets or displays the bit field that configures the internal resistor status for the digital input lines. Internal pull-up/down resistors are not available for digital output pins, analog input pins, or for disabled pins.

Use the **PD** command to specify whether the resistor is pull-up or pull-down.

- If you set a **PR** bit to 1, it enables the pull-up/down resistor.
- If you set a **PR** bit to 0, it specifies no internal pull-up/down resistor.

The following table defines the bit-field map for both the **PR** and **PD** commands.

Bit	I/O line	Module pin
0	DIO4	pin 11
1	DIO3/AD3	pin 17
2	DIO2/AD2	pin 18
3	DIO1/AD1	pin 19
4	DIO0/AD0	pin 20
5	DIO6/$\overline{\text{RTS}}$	pin 16
6	DIO8/SLEEP_REQUEST	pin 9
7	DIO14/DIN	pin 3
8	DIO5/ASSOCIATE	pin 15
9	DIO9/On/$\overline{\text{SLEEP}}$	pin 13
10	DIO12	pin 4
11	DIO10	pin 6
12	DIO11	pin 7
13	DIO7/$\overline{\text{CTS}}$	pin 12
14	DIO13/DOUT	pin 2

Parameter range

0 - 0x7FFF (bit field)

Default

0x7FFF

M0 (PWM0 Duty Cycle)

Sets the duty cycle of PWM0 (pin 6) for **P0** = **2**, where a value of 0x200 is a 50% duty cycle.

Before setting the line as an output:

1. Enable PWM0 output (P0 (DIO10/PWM0 Configuration) = **2**).
2. Apply the settings (use CN (Exit Command mode) or AC (Apply Changes)).

The PWM period is 42.62 µs and there are 0x03FF (1023 decimal) steps within this period. When **M0** = **0** (0% PWM), **0x01FF** (50% PWM), **0x03FF** (100% PWM), and so forth.

Parameter range

0 - 0x3FF

Default

0

I/O sampling commands

The following AT commands configure I/O sampling parameters.

TP (Temperature)

Displays the temperature of the XBee Cellular Modem in degrees Celsius. The temperature value is displayed in 8-bit two’s complement format. For example, **0x1A** = 26 °C, and **0xF6** = -10 °C.

Parameter range

0 - 0xFF which indicates degrees Celsius displayed in 8-bit two's complement format.

Default

N/A

IS (Force Sample)

When run, **IS** reports the values of all of the enabled digital and analog input lines. If no lines are enabled for digital or analog input, the command returns an error.

Command mode

In Command mode, the response value is a multi-line format, individual lines are delimited with carriage returns, and the entire response terminates with two carriage returns. Each line is a series of ASCII characters representing a single number in hexadecimal notation. The interpretation of the lines is:

- Number of samples. For legacy reasons this field always returns 1.
- Digital channel mask. A bit-mask of all I/O capable pins in the system. The bits set to **1** are configured for digital I/O and are included in the digital data value below. Pins D0 - D9 are bits 0 - 9, and P0 - P2 are bits 10 - 12.
- Analog channel mask. The bits set to **1** are configured for analog I/O and have individual readings following the digital data field.
- Digital data. The current digital value of all the pins set in the digital channel mask, only present if at least one bit is set in the digital channel mask.
- Analog data. Additional lines, one for each set pin in the analog channel mask. Each reading is a 10-bit ADC value for a 2.5 V voltage reference.

API operating mode

In API operating mode, **IS** immediately returns an **OK** response.

The API response is ordered identical to the Command mode response with the same fields present. Each field is a binary number of the size listed in the following table. Multi-byte fields are in big-endian byte order.

Field	Size
Number of samples	1 byte

Field	Size
Digital channel mask	2 bytes
Analog chanel mask	1 byte
Samples	2 bytes each

Parameter range

N/A

Default

N/A

Sleep commands

The following AT commands are sleep commands.

SM (Sleep Mode)

Sets or displays the sleep mode of the device.

The sleep mode determines how the device enters and exits a power saving sleep.

Sleep mode is also affected by the **SO** command, option bit 6. See Sleep modes for more information about sleep modes.

Parameter range

0, 1, 4, 5

Parameter	Description
0	Normal. In this mode the device never sleeps.
1	Pin Sleep. In this mode the device honors the SLEEP_RQ pin. Set D8 (DIO8/SLEEP_ REQUEST) to the sleep request function: **1**.
4	Cyclic Sleep. In this mode the device repeatedly sleeps for the value specified by **SP** and spends **ST** time awake.
5	Cyclic Sleep with Pin Wake. In this mode the device acts as in Cyclic Sleep but does not sleep if the SLEEP_RQ pin is inactive, allowing the device to be kept awake or woken by the connected system.

Default

0

SP (Sleep Period)

Sets or displays the time to spend asleep in cyclic sleep modes. In Cyclic sleep mode, the node sleeps with CTS disabled for the sleep time interval, then wakes for the wake time interval.

Parameter range

0x1 - 0x83D600 (x 10 ms)

Default

0x7530 (5 minutes)

ST (Wake Time)

Sets or displays the time to spend awake in cyclic sleep modes.

Parameter range

0x1 - 0x36EE80 (x 1 ms)

Default

0xEA60 (60 seconds)

SO (Sleep Options)

Set or read the sleep options bit field of a device. This command is a bitmask.

Parameter range

0x0 - 0xFFFF

Bit field:

Bit	Setting	Meaning	Description
0x00	0	Connected sleep	On compatible hardware, enters a lower power consumption mode that maintains registration with the cellular network. Read the HI (Hardware Identity) command to determine if the hardware is compatible with the connected sleep feature. If the HI command returns a value of 3, then the module is able to use the connected sleep feature. Otherwise the hardware is not compatible.
Set all other option bits to 0.			

Default

0

Command mode options

The following commands are Command mode option commands.

CC (Command Sequence Character)

The character value the device uses to enter Command mode.

The default value (**0x2B**) is the ASCII code for the plus (**+**) character. You must enter it three times within the guard time to enter Command mode. To enter Command mode, there is also a required period of silence before and after the command sequence characters of the Command mode sequence (**GT** + **CC** + **GT**). The period of silence prevents inadvertently entering Command mode.

Parameter range

Recommended: 0x20 - 0x7F (ASCII)

Default

0x2B (the ASCII plus character: **+**)

CT (Command Mode Timeout)

Sets or displays the Command mode timeout parameter. If a device does not receive any valid commands within this time period, it returns to Idle mode from Command mode.

Parameter range

2 - 0x1770 (x 100 ms)

Default

0x64 (10 seconds)

CN (Exit Command mode)

Immediately exits Command Mode and applies pending changes.

Note Whether Command mode is exited using the **CN** command or by **CT** timing out, changes are applied upon exit.

Parameter range

N/A

Default

N/A

GT (Guard Times)

Set the required period of silence before and after the command sequence characters of the Command mode sequence (**GT** + **CC** + **GT**). The period of silence prevents inadvertently entering Command mode.

Parameter range

0x2 - 0x576 (x 1 ms)

Default

0x3E8 (one second)

MicroPython commands

The following commands relate to using MicroPython on the XBee Cellular Modem.

PS (Python Startup)

Sets whether or not the XBee Cellular Modem runs the stored Python code at startup.

Range

0 - 1

Parameter	Description
0	Do not run stored Python code at startup.
1	Run stored Python code at startup.

Default

0

PY (MicroPython Command)

Interact with the XBee Cellular Modem using MicroPython. **PY** is a command with sub-commands. These sub-commands are arguments to **PY**.

PYC(Code Report)

You can store compiled code in flash using the **Ctrl-F** command from the MicroPython REPL; refer to the *Digi MicroPython Programming Guide*. The **PYC** sub-command reports details of the stored code. In Command mode, it returns three lines of text, for example:

```
source: 1662 bytes (hash=0xC3B3A813)
bytecode: 619 bytes (hash=0x0900DBCE)
compiled: 2017-05-09T15:49:44
```

The messages are:

- **source**: the size of the source code used to generate the bytecode and its 32-bit hash.
- **bytecode**: the size of bytecode stored in flash and its 32-bit hash. A size of **0** indicates that there is no stored code.
- **compiled**: a compilation timestamp. A timestamp of **2000-01-01T00:00:00** indicates that the clock was not set during compilation.

In API mode, **PYC** returns five 32-bit big-endian values:

- source size
- source hash
- bytecode size
- bytecode hash
- timestamp as seconds since 2000-01-01T00:00:00

PYD (Delete Code)

PYD interrupts any running code, erases any stored code and then does a soft-reboot on the MicroPython subsystem.

PYV (Version Report)

Report the MicroPython version.

PY^ (Interrupt Program)

Sends **KeyboardInterrupt** to MicroPython. This is useful if there is a runaway MicroPython program and you have filled the stdin buffer. You can enter Command mode (**+++**) and send **ATPY^** to interrupt the program.

Default

N/A

Firmware version/information commands

The following AT commands are firmware version/information commands.

VR (Firmware Version)

Reads the firmware version on a device.

The firmware version returns four hexadecimal values (2 bytes) **ABCD**. Digits **ABC** are the main release number and **D** is the revision number from the main release. **B** is a variant designator where **0** means standard release.

Parameter range

0 - 0xFFFF [read-only]

Default

Set in firmware

VL (Verbose Firmware Version)

Shows detailed version information including the application build date and time.

Parameter range

N/A

Default

Set in firmware

HV (Hardware Version)

Display the hardware version number of the device.

Read the device's hardware version. Use this command to distinguish between different hardware platforms. The upper byte returns a value that is unique to each device type. The lower byte indicates the hardware revision.

Parameter range

0 - 0xFFFF [read-only]

Default

Set in firmware

AI (Association Indication)

Reads the Association status code to monitor association progress. The following table provides the status codes and their meanings.

Status code	Meaning
0x00	Connected to the Internet.

Status code	Meaning
0x22	Registering to cellular network.
0x23	Connecting to the Internet.
0x24	The cellular component is missing, corrupt, or otherwise in error. The cellular component requires a new firmware image.
0x25	Cellular network registration denied.
0x2A	Airplane mode.
0x2F	Bypass mode active.
0xFF	Initializing.

Parameter range

0 - 0xFF [read-only]

Default

N/A

HS (Hardware Series)

Read the device's hardware series number.

Parameter range

N/A

Default

Set in the firmware

CK (Configuration CRC)

Displays the cyclic redundancy check (CRC) of the current AT command configuration settings.

Parameter range

0 - 0xFFFFFFFF

Default

N/A

Diagnostic interface commands

The following AT commands are diagnostic interface commands.

DI (Remote Manager Indicator)

Displays the current Remote Manager status for the XBee.

Range

Value	Description
0x00	Connected, but without TLS or authentication.
0x01	Before connection to the Internet.
0x02	Remote Manager connection in progress.
0x03	Disconnecting from Remote Manager.
0x04	Not configured for Remote Manager.
0x05	Connected over TLS.
0x06	Connected over TLS with authenticated server.

Default

N/A

CI (Protocol/Connection Indication)

Displays information regarding the last IP connection when using Transparent mode (**AP** = **0**), and when **IP** = **0**, **1** or **4** or when **IP** = **2** for an SMS transmission.

The value for this parameter resets to **0xFF** when the device switches between IP (IP Protocol) modes.

When **IP** is set to **0**, **1**, or **4** (UDP, TCP, over TLS over TCP), **CI** resets to **0xFF** when you apply changes to any of the following settings:

- DL (Destination Address)
- DE (Destination port)
- TM (IP Client Connection Timeout)

When **IP** is set to **2** (SMS), **CI** resets to **0xFF** when P# (Destination Phone Number) is changed.

The following table provides the parameter's meaning when **IP** = **0** for UDP connections.

Parameter	Description
0x00	The socket is open.
0x01	Tried to send but could not.
0x02	Invalid parameters (bad IP/host).

Parameter	Description
0x03	TCP not supported on this cellular component.
0x10	Not registered to the cell network.
0x11	Cellular component not identified yet.
0x12	DNS query lookup failure.
0x13	Socket leak
0x20	Bad handle.
0x21	User closed.
0x22	Unknown server - DNS lookup failed.
0x23	Connection lost.
0x24	Unknown.
0xFF	No known status.

The following table provides the parameter's meaning when **IP** = **1** or **4** for TCP connections.

Parameter	Description
0x00	The socket is open.
0x01	Tried to send but could not.
0x02	Invalid parameters (bad IP/host).
0x03	TCP not supported on this cellular component.
0x10	Not registered to the cell network.
0x11	Cellular component not identified yet.
0x12	DNS query lookup failure.
0x13	Socket leak
0x20	Bad handle.
0x21	User closed.
0x22	No network registration.
0x23	No internet connection.
0x24	No server - timed out on connection.
0x25	Unknown server - DNS lookup failed.
0x26	Connection refused.
0x27	Connection lost.

Parameter	Description
0x28	Unknown.
0xFF	No known status.

The following table provides the parameter's meaning when **IP** = **2** for SMS connections.

Parameter	Description
0x00	SMS successfully sent.
0x01	SMS failed to send.
0x02	Invalid SMS parameters - check P# (Destination Phone Number).
0x03	SMS not supported.
0x10	No network registration.
0x11	Cellular component stack error.
0x13	Socket leak
0xFF	No SMS state to report (no SMS messages have been sent).

Parameter range

0 - 0xFF (read-only)

Default

-

AS (Active scan for network environment data)

Scans for mobile cells in the vicinity and returns information about the cells in the service area of the device. When you run the command, the cell module waits until all other communication is idle and then performs the scan.

The information that can be reported by this command varies based on the network technology of the module that you are using.

In both AT and API mode the command returns line-based records mapping key-value pairs. The record for the serving cell begins with the capital letter S, and keys for the fields are MCC, MNC, Area, CID, and Signal. Each line describes a particular cell and only those values determined during a single scan are reported.

Example

```
atas

S MCC:311 MNC:480 Area:48707
CID:48825632 Signal:-88
CID:48825612 Signal:-95
CID:48825603 Signal:-68
CID:48825601 Signal:-71
```

Parameter range

0-1

Value	Description
0 or no value	Scans for mobile cells in the vicinity and returns information about the cells in the service area of the module. When you run the command, the cell module waits until all other communication is idle and then performs the scan.
1	Attempts a full scan, which requires dropping network registration. Any outstanding sockets or other activity will be lost. Since registration is lost, no "serving cell" information is provided, as the "serving cell" that the device will re-join cannot be reported, and there is no guarantee that the "serving cell" the device was on before network registration was dropped will still be used. A full scan can return more complete information for all cells seen, which includes cells offered by other carriers. The duration of the scan is approximately 25 seconds. **Note** This action should be used only on CAT 1 modules. If this value is set on an LTE-M module, the result will be as if the value was set to 0.

Parameter range

N/A

Default

N/A

Execution commands

The location where most AT commands set or query register values, execution commands execute an action on the device. Execution commands are executed immediately and do not require changes to be applied.

NR (Network Reset)

NR resets the network layer parameters.

The XBee Cellular Modem responds immediately with an **OK** on the UART and then causes a network restart.

If **NR** = **0**, the XBee Cellular Modem tears down any TCP/UDP sockets and resets Internet connectivity.

You can also send **NR**, which acts like **NR** = **0**.

Parameter range

0

Default

N/A

!R (Modem Reset)

Forces the cellular component to reboot.

CAUTION! This command is for advanced users, and you should only use it if the cellular component becomes completely stuck while in Bypass mode. Normal users should never need to run this command. See the FR (Force Reset) command instead.

Range

N/A

Default

N/A

File system commands

To access the file system, Enter Command mode and use the following commands. All commands block the AT command processor until completed and only work from Command mode; they are not valid for API mode or MicroPython's **xbee.atcmd()** method. Commands are case-insensitive as are file and directory names. Optional parameters are shown in square brackets (**[]**).

FS is a command with sub-commands. These sub-commands are arguments to **FS**.

For **FS** commands, you have to type **AT** before the command, for example **ATFS PWD**, **ATFS LS** and so forth.

Error responses

If a command succeeds it returns information such as the name of the current working directory or a list of files, or **OK** if there is no information to report. If it fails, you see a detailed error message instead of the typical **ERROR** response for a failing AT command. The response is a named error code and a textual description of the error.

Note The exact content of error messages may change in the future. All errors start with a capital **E**, followed by one or more uppercase letters and digits, a space, and an description of the error.
If writing your own AT command parsing code, you can determine if an **FS** command response is an error by checking if the first letter of the response is capital **E**.

ATFS (File System)

When sent without any parameters, **FS** prints a list of supported commands.

ATFS PWD

Prints the current working directory, which always starts with **/** and defaults to **/flash** at startup.

ATFS CD *directory*

Changes the current working directory to **directory**. Prints the current working directory or an error if unable to change to **directory**.

ATFS MD *directory*

Creates the directory **directory**. Prints **OK** if successful or an error if unable to create the requested directory.

ATFS LS [*directory*]

Lists files and directories in the specified directory. The **directory** parameter is optional and defaults to a period (**.**), which represents the current directory. The list ends with a blank line.

Entries start with zero or more spaces, followed by filesize or the string **<DIR>** for directories, then a single space character and the name of the entry. Directory names end with a forward slash (**/**) to differentiate them from files. Secure files end with a hash mark (**#**) and you cannot download them.

```
<DIR> ./
<DIR> ../
<DIR> cert/
```

```
<DIR> lib/
   32 test.txt
 1234 secure.bin#
```

ATFS PUT *filename*

Starts a YMODEM receive on the XBee Cellular Modem, storing the received file to filename and ignoring the filename that appears in block 0 of the YMODEM transfer. The XBee Cellular Modem sends a prompt (**Receiving file with YMODEM...**) when it is ready to receive, at which point you should initiate a YMODEM send in your terminal emulator.

If the command is incorrect, the reply will be an error as described in Error responses.

ATFS XPUT filename

Similar to the **PUT** command, but stores the file securely on the XBee Cellular Modem. See Secure files for details on what this means.

If the command is incorrect, the reply will be an error as described in Error responses.

ATFS HASH *filename*

Print a SHA-256 hash of a file to allow for verification against a local copy of the file.

- On Windows, you can generate a SHA-256 hash of a file with the command **certutil -hashfile test.txt SHA256**.
- On Mac and Linux use **shasum -b -a 256 test.txt**.

ATFS GET *filename*

Starts a YMODEM send of **filename** on the XBee device. When it is ready to send, the XBee Cellular Modem sends a prompt: (**Sending file with YMODEM...**). When the prompt is sent, you should initiate a YMODEM receive in your terminal emulator.

If the command is incorrect, the reply will be an error as described in Error responses.

ATFS MV *source_path dest_path*

Moves or renames the selected file or directory **source_path** to the new name or location **dest_path**. This command fails with an error if **source_path** does not exist, or **dest_path** already exists.

Note Unlike a computer's command prompt which moves a file into the **dest_path** if it is an existing directory, you must specify the full name for **dest_path**.

ATFS RM *file_or_directory*

Removes the file or empty directory specified by **file_or_directory**. This command fails with an error if **file_or_directory** does not exist, is not empty, refers to the current working directory or one of its parents.

ATFS INFO

Report on the size of the filesystem, showing bytes in use, available, marked bad and total. The report ends with a blank line, as with most multi-line AT command output. Example output:

```
    204800 used
     695296 free
          0 bad
     900096 total
```

ATFS FORMAT confirm

Reformats the file system, leaving it with a default directory structure. Pass the word **confirm** as the first parameter to confirm the format. The XBee Cellular Modem responds with **Formatting...**, adds a period every second until the format is complete and ends the response with a carriage return.

Remote Manager commands

The following commands are used with Remote Manager.

MO (Remote Manager Options)

Configures the connection to Remote Manager.

Note When the bit 0 is set to 0, you should manage the Remote Manager keepalive interval, which may otherwise result in excessive data usage. See Configure Remote Manager keepalive interval.

Parameter range

0 - 7

Bit	Description
0	Maintains a persistent TCP connection to Remote Manager.
1	TCP connection uses TLS. This is the default.
2	Reserved for future use.

Default

6 (Bits 1 and 2 are enabled by default.)

DF (Remote Manager Status Check Interval)

Defines the number of minutes between polls for Remote Manager activity.

Parameter range

N/A

Default

1440

EQ (Remote Manager FQDN)

Sets or display the fully qualified domain name of the Remote Manager server.

Range

From 0 through 63 ASCII characters.

Default

my.devicecloud.com

K1 (Remote Manager Server Send Keepalive)

Specify the Remote Manager Server Send Transmit Keepalive Interval value in seconds. The XBee device considers a Remote Manager connection to have failed after 3 missed keepalives.

This command works with the K2 command to limit data usage. See Configure Remote Manager keepalive interval.

Note Changing this value causes any currently active Remote Manager connections to be closed and recreated.

Parameter range

10 - 7200 (x 1 s)

Default

75

K2 (Remote Manager Device Send Keepalive)

Specify the Remote Manager Device Send Transmit Keepalive Interval value in seconds. The Remote Manager considers a connection to have failed after 3 missed keepalives.

This command works with the K1 command to limit data usage. See Configure Remote Manager keepalive interval.

Note Changing this value causes any currently active Remote Manager connections to be closed and recreated.

Parameter range

10 - 7200 (x 1 s)

Default

60

$D (Remote Manager certificate)

Defines the TLS Remote Manager certificate.

Parameter range

N/A

Default

/flash/cert/digi-remote-mgr.pem

System commands

The following commands are used to assign descriptors to the XBee Cellular Modem, which distinguish the devices from each other in Remote Manager.

KL (Device Location)

Sets or displays a user-defined physical location for the XBee displayed in Remote Manager.

Range

Up to 20 ASCII characters

Default

One ASCII space character (0x20).

KC (Contact Information)

Sets or displays user-defined contact information for the XBee displayed in Remote Manager.

Range

Up to 20 ASCII characters

Default

One ASCII space character (0x20).

KP (Device Description)

Sets or displays a user-defined description for the XBee displayed in Remote Manager.

Range

Up to 20 ASCII characters

Default

One ASCII space character (0x20)

Socket commands

The following AT commands are socket commands.

SI (Socket Info)

Lists either information about a given socket or lists the socket IDs of all active (open) sockets on the modem in a human-readable format.

When the **SI** command is issued without a parameter, the XBee outputs a list of socket IDS in hex, separated by carriage returns (<CR>). After the last socket ID has been printed the list is terminated with an additional carriage return.

In both API and command mode the payload (output) will have the following format:

```
ID<CR>
ID<CR>
. . .
ID<CR>
<CR>
```

In the list of socket IDs, an asterisk (*) displays after the socket ID for non-Extended API Sockets (which are sockets created implicitly when using IPv4 TX API frames). In the example below, the 0x00 socket is an IPv4 TX/RX socket, and the 0x01 and 0x02 sockets are both Extended API sockets. The socket IDs are displayed in ascending order, from smallest socket value to the largest.

```
0x00*
0x01
0x02
```

Note When sending AT commands for API frames it is standard to send the command as ASCII text and the parameters for that command as binary.

When the **SI** command is issued with a socket ID, specified in hex, the response is a list of information about the socket. The list is separated by carriage returns (<CR>) and terminated with an additional carriage return.

In both API and command mode the payload/output will have the following format:

```
ID<CR>
STATE<CR>
PROTOCOL<CR>
LOCAL_PORT<CR>
REMOTE_PORT<CR>
REMOTE_ADDRESS<CR>
<CR>
```

Field	Description
ID	The socket ID.

Field	Description
STATE	The state of the socket: ■ ALLOCATED ■ CONNECTING ■ CONNECTED ■ LISTENING ■ BOUND ■ CLOSING
PROTOCOL	The protocol of the socket: ■ UDP ■ TCP ■ TLS
LOCAL_PORT	The local port of the socket. This is 0 unless the socket is explicitly bound to a port.
REMOTE_PORT	The remote port of the socket.
REMOTE_ADDRESS	The remote IPv4 address for the given socket. This is 0.0.0.0 for an unconnected socket.

Parameter range

0x00 - 0xFE

Default

-

Operate in API mode

API mode overview

As an alternative to Transparent operating mode, you can use API operating mode. API mode provides a structured interface where data is communicated through the serial interface in organized packets and in a determined order. This enables you to establish complex communication between devices without having to define your own protocol. The API specifies how commands, command responses and device status messages are sent and received from the device using the serial interface or the SPI interface.

We may add new frame types to future versions of firmware, so build the ability to filter out additional API frames with unknown frame types into your software interface.

Use the AP command to set the operation mode

Use AP (API Enable) to specify the operation mode:

AP command setting	Description
AP = 0	Transparent operating mode, UART serial line replacement with API modes disabled. This is the default option.
AP = 1	API operation.
AP = 2	API operation with escaped characters (only possible on UART).
AP = 3	N/A
AP = 4	MicroPython REPL
AP = 5	Bypass mode. This mode is for direct communication with the underlying chip and is only for advanced users.

The API data frame structure differs depending on what mode you choose.

API frame format

An API frame consists of the following:

- Start delimeter
- Length
- Frame data
- Checksum

API operation (AP parameter = 1)

This is the recommended API mode for most applications. The following table shows the data frame structure when you enable this mode:

Frame fields	Byte	Description
Start delimiter	1	0x7E
Length	2 - 3	Most Significant Byte, Least Significant Byte
Frame data	4 - number (n)	API-specific structure
Checksum	n + 1	1 byte

Any data received prior to the start delimiter is silently discarded. If the frame is not received correctly or if the checksum fails, the XBee replies with a radio status frame indicating the reason for the failure.

API operation with escaped characters (AP parameter = 2)

Setting API to 2 allows escaped control characters in the API frame. Due to its increased complexity, we only recommend this API mode in specific circumstances. API 2 may help improve reliability if the serial interface to the device is unstable or malformed frames are frequently being generated.

When operating in API 2, if an unescaped 0x7E byte is observed, it is treated as the start of a new API frame and all data received prior to this delimiter is silently discarded. For more information on using this API mode, see the Escaped Characters and API Mode 2 in the Digi Knowledge base.

API escaped operating mode works similarly to API mode. The only difference is that when working in API escaped mode, the software must escape any payload bytes that match API frame specific data, such as the start-of-frame byte (0x7E). The following table shows the structure of an API frame with escaped characters:

Frame fields	Byte	Description	
Start delimiter	1	0x7E	
Length	2 - 3	Most Significant Byte, Least Significant Byte	Characters escaped if needed
Frame data	4 - n	API-specific structure	
Checksum	n + 1	1 byte	

Start delimiter field

This field indicates the beginning of a frame. It is always 0x7E. This allows the device to easily detect a new incoming frame.

Escaped characters in API frames

If operating in API mode with escaped characters (**AP** parameter = 2), when sending or receiving a serial data frame, specific data values must be escaped (flagged) so they do not interfere with the data frame sequencing. To escape an interfering data byte, insert 0x7D and follow it with the byte to be escaped (XORed with 0x20).

The following data bytes need to be escaped:

- 0x7E: start delimiter
- 0x7D: escape character
- 0x11: XON
- 0x13: XOFF

To escape a character:

1. Insert 0x7D (escape character).
2. Append it with the byte you want to escape, XORed with 0x20.

In API mode with escaped characters, the length field does not include any escape characters in the frame and the firmware calculates the checksum with non-escaped data.

Example: escape an API frame

To express the following API non-escaped frame in API operating mode with escaped characters:

Start delimiter	Length	Frame type	Frame Data	Checksum
			Data	
7E	00 0F	17	01 00 13 A2 00 40 AD 14 2E FF FE 02 4E 49	6D

You must escape the 0x13 byte:

1. Insert a 0x7D.
2. XOR byte 0x13 with 0x20: 13 ⊕ 20 = 33

The following figure shows the resulting frame. Note that the length and checksum are the same as the non-escaped frame.

Start delimiter	Length	Frame type	Frame Data	Checksum
			Data	
7E	00 0F	17	01 00 7D 33 A2 00 40 AD 14 2E FF FE 02 4E 49	6D

The length field has a two-byte value that specifies the number of bytes in the frame data field. It does not include the checksum field.

Length field

The length field is a two-byte value that specifies the number of bytes contained in the frame data field. It does not include the checksum field.

Frame data

This field contains the information that a device receives or will transmit. The structure of frame data depends on the purpose of the API frame:

Start delimiter	Length		Frame data								Checksum
			Frame type	Data							
1	2	3	4	5	6	7	8	9	...	n	n+1
0x7E	MSB	LSB	API frame type	Data							Single byte

- **Frame type** is the API frame type identifier. It determines the type of API frame and indicates how the Data field organizes the information.
- **Data** contains the data itself. This information and its order depend on the what type of frame that the Frame type field defines.

Multi-byte values are sent big-endian.

Calculate and verify checksums

To calculate the checksum of an API frame:

1. Add all bytes of the packet, except the start delimiter 0x7E and the length (the second and third bytes).
2. Keep only the lowest 8 bits from the result.
3. Subtract this quantity from 0xFF.

To verify the checksum of an API frame:

1. Add all bytes including the checksum; do not include the delimiter and length.
2. If the checksum is correct, the last two digits on the far right of the sum equal 0xFF.

Example

Consider the following sample data packet: **7E 00 0A 01 01 50 01 00 48 65 6C 6C 6F B8**+

Byte(s)	Description
7E	Start delimiter
00 0A	Length bytes
01	API identifier
01	API frame ID
50 01	Destination address low
00	Option byte
48 65 6C 6C 6F	Data packet
B8	Checksum

To calculate the check sum you add all bytes of the packet, excluding the frame delimiter **7E** and the length (the second and third bytes):

7E 00 0A 01 01 50 01 00 48 65 6C 6C 6F B8

Add these hex bytes:

01 + 01 + 50 + 01 + 00 + 48 + 65 + 6C + 6C + 6F = 247

Now take the result of 0x247 and keep only the lowest 8 bits which in this example is 0xC4 (the two far right digits). Subtract 0x47 from 0xFF and you get 0x3B (0xFF - 0xC4 = 0x3B). 0x3B is the checksum for this data packet.

If an API data packet is composed with an incorrect checksum, the XBee Cellular Modem will consider the packet invalid and will ignore the data.

To verify the check sum of an API packet add all bytes including the checksum (do not include the delimiter and length) and if correct, the last two far right digits of the sum will equal FF.

01 + 01 + 50 + 01 + 00 + 48 + 65 + 6C + 6C + 6F + B8 = 2FF

API frames

The following sections describe the API frames.

AT Command - 0x08

Description

Use this frame to query or set parameters on the local device. Changes this frame makes to device parameters take effect after executing the AT command.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x08	Byte	
Frame ID		Byte	Identifies the data frame for the host to correlate with a subsequent ACK. If set to **0**, the device does not send a response.
AT command		Byte	Command name: two ASCII characters that identify the AT command.
Parameter value		Byte	If present, indicates the requested parameter value to set the given register. If no characters are present, it queries the register.

AT Command: Queue Parameter Value - 0x09

Description

This frame allows you to query or set device parameters. In contrast to AT Command - 0x08, this frame queues new parameter values and does not apply them until you issue either:

- The AT Command (0x08) frame
- The **AC** command

When querying parameter values, the 0x09 frame behaves identically to the 0x08 frame. The device returns register queries immediately and not does not queue them. The response for this command is also an AT Command Response frame (0x88).

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x09	Byte	
Frame ID		Byte	Identifies the data frame for the host to correlate with a subsequent ACK. If set to **0**, the device does not send a response.
AT command		Byte	Command name: two ASCII characters that identify the AT command.
Parameter value		Byte	If present, indicates the requested parameter value to set the given register. If no characters are present, it queries the register.

Transmit (TX) SMS - 0x1F

Description

Transmit an SMS message. The frame allows international numbers with or without the **+** prefix. If you omit **+** and are dialing internationally, you need to include the proper International Dialing Prefix for your calling region, for example, 011 for the United States.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x1F	Byte	
Frame ID		Byte	Reference identifier used to match status responses. **0** disables the TX Status frame.
Options		Byte	Reserved for future use.
Phone number		20 byte string	String representation of phone number terminated with a null (0x0) byte. Use numbers and the **+** symbol only, no other symbols or letters.
Payload		Variable (160 characters maximum)	Data to send as the body of the SMS message.

Transmit (TX) Request: IPv4 - 0x20

Description

A TX Request message causes the device to transmit data in IPv4 format. A TX request frame for a new destination creates a network socket. After the network socket is established, data from the network that is received on the socket is sent out the device's serial port in the form of a Receive (RX) Packet frame.

When you specify protocol **4** (TLS), the profile configuration specified by $0 (TLS Profile 0) is used to form the TLS connection.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x20	Byte	
Frame ID		Byte	Reference identifier used to match status responses. **0** disables the TX Status frame.
Destination address		32-bit big endian	
Destination port		16-bit big endian	
Source port		16-bit big endian	If the source port is **0**, the device attempts to send the frame data using an existing open socket with a destination that matches the destination address and destination port fields of this frame. If there is no matching socket, then the device attempts to open a new socket. If the source port is non-zero, the device attempts to send the frame data using an existing open socket with a source and destination that matches the source port, destination address, and destination port fields of this frame. If there is no matching socket, it returns an error.
Protocol		Byte	0 = UDP 1 = TCP 4 = SSL over TCP

Field name	Field value	Data type	Description
Transmit options		Byte bitfield	Bit fields are offset 0 Bit field 0 - 7. Bits 0, and 2-7 are reserved, bit 1 is not. BIT 1 = **1** - Terminate the TCP socket after transmission is complete **0** - Leave the socket open. Closed by timeout, see TM (IP Client Connection Timeout). Ignore this bit for UDP packets. All other bits are reserved and should be **0**.
Payload		Variable	Data to be transferred to the destination, may be up to 1500 bytes.

Tx Request with TLS Profile - 0x23

Description

The frame gives greater control to the application over the TLS settings used for a connection.

A TX Request with TLS Profile frame implies the use of TLS and behaves similar to the TX Request (0x20) frame, with the protocol field replaced with a TLS Profile field to choose from the profiles configured with the $0, $1, and $2 configuration commands.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x23	Byte	
Frame ID		Byte	Reference identifier used to match status responses. **0** disables the TX Status frame.
Destination address		32-bit big endian	
Destination port		16-bit big endian	
Source port		16-bit big endian	If the source port is **0**, the device attempts to send the frame data using an existing open socket with a destination that matches the destination address and destination port fields of this frame. If there is no matching socket, then the device attempts to open a new socket. If the source port is non-zero, the device attempts to send the frame data using an existing open socket with a source and destination that matches the source port, destination address, and destination port fields of this frame. If there is no matching socket, the TX Status frame returns an error.
TLS profile		Byte	Zero-indexed number that indicates the profile as specified by the corresponding **$*<num>*** command.
Transmit options		Byte bitfield	Bit fields are offset 0 Bit field 0 - 7. Bits 0, and 2-7 are reserved, bit 1 is not. BIT 1 = **1** - Terminate the TCP socket after transmission is complete **0** - Leave the socket open. Closed by timeout, see TM (IP Client Connection Timeout). Ignore this bit for UDP packets. All other bits are reserved and should be **0**.

Field name	Field value	Data type	Description
Payload		Variable	Data to be transferred to the destination, may be up to 1500 bytes.

AT Command Response - 0x88

Description

A device sends this frame in response to an AT Command (0x08) frame. Some commands send back multiple frames.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x88	Byte	
Frame ID		Byte	Identifies the data frame for the host to correlate with a subsequent ACK. If set to **0**, the device does not send a response.
AT command		Byte	Command name: two ASCII characters that identify the AT command.
Status	##	Byte	0 = OK 1 = ERROR 2 = Invalid command 3 = Invalid parameter
Parameter value		Byte	Register data in binary format. If the register was set, then this field is not returned.

Transmit (TX) Status - 0x89

Description

Indicates the success or failure of a transmit operation.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x89	Byte	
Frame ID		Byte	Refers to the frame ID specified in a previous transmit frame
Status		Byte	Status code (see the table below)

The following table shows the status codes.

Code	Description
0x0	Successful transmit
0x21	Failure to transmit to cell network
0x22	Not registered to cell network
0x2c	Invalid frame values (check the phone number)
0x31	Internal error
0x32	Resource error (retry operation later). See Socket limits in API mode for more information.
0x74	Message too long
0x76	Socket closed unexpectedly
0x78	Invalid UDP port
0x79	Invalid TCP port
0x7A	Invalid host address
0x7B	Invalid data mode
0x7C	Invalid interface. See User Data Relay - 0x2D.
0x7D	Interface not accepting frames. See User Data Relay - 0x2D.

Code	Description
0x80	Connection refused
0x81	Socket connection lost
0x82	No server
0x83	Socket closed
0x84	Unknown server
0x85	Unknown error
0x86	Invalid TLS configuration (missing file, and so forth)

Modem Status - 0x8A

Description

Cellular component status messages are sent from the device in response to specific conditions.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x8A	Byte	
Status	##	Byte	0 = Hardware reset or power up 1 = Watchdog timer reset 2 = Registered with cellular network 3 = Unregistered with cellular network 0x0E = Remote Manager connected 0x0F = Remote Manager disconnected 0x35 = Cellular component update started 0x36 = Cellular component update failed 0x37 = Cellular component update completed 0x38 = XBee firmware update started 0x39 = XBee firmware update failed 0x3A = XBee firmware update applying

Receive (RX) Packet: SMS - 0x9F

Description

This XBee Cellular Modem uses this frame when it receives an SMS message.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame Type	0x9F	Byte	
Phone number		20 byte string	String representation of the phone number, padded out with null bytes (0x0).
Payload		Variable	Body of the received SMS message.

Receive (RX) Packet: IPv4 - 0xB0

Description

The XBee Cellular Modem uses this frame when it receives RF data on a network socket that is created by a TX request frame or configuring C0 (Source Port).

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Frame data fields	Offset	Description
Frame type	3	0xB0
IPv4 32-bit source address	MSB 4	The address in the example below is for a source address of **192.168.0.104**. 32-bit big endian.
	5	
	6	
	7	
16-bit destination port	MSB 8	The port that the packet was received on. 16-bit big endian.
	LSB 9	
16-bit source port	MSB 10	The port that the packet was sent from. 16-bit big endian.
	LSB 11	
Protocol	MSB 12	0 = UDP 1 = TCP 4 = SSL over TCP
Status	13	Reserved
Payload	14	Data received from the source. The maximum size is 1500 bytes.
	15	
	16	
	17	
	18	

User Data Relay - 0x2D

Description

Allows for data to be sent to an interface with a designation of a target interface for the data to be output on. The frame can be sent or received from either of these interfaces: MicroPython (internal interface) or UART. This frame is used in conjunction with User Data Relay Output - 0xAD.

You can send and receive User Data Relay Frames from MicroPython. See Send and receive User Data Relay frames in the *MicroPython Programming Guide*.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x2D	Byte	
Frame ID			Reference identifier used to match TX Status frames (type 0x89) sent for errors. A value of **0** disables the TX Status frame.
Destination interface		Byte	0 = Serial port (SPI, or UART when in API mode) 2 = MicroPython
Data		Variable	

Error cases

The Frame ID is used to report error conditions in a method consistent with existing transmit frames. The error codes are mapped to statuses. The following conditions result in an error that is reported in a TX Status frame, referencing the frame ID from the 0x2d request.

- **Invalid interface** (0x7c) : The user specified a destination interface that does not exist.

Example use cases

An external processor outputs the Frame over the UART with the Micropython interface as a target. Micropython operates over the data and publishes the data to mqtt topic.

User Data Relay Output - 0xAD

Description

Allows for data to be received on an interface with a designation of the target interface for the data to be output on. The frame can be sent or received from any of the following interfaces: MicroPython (internal interface) or UART. This frame is used in conjunction with User Data Relay - 0x2D.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0xAD	Byte	
Source interface		Byte	0 = Serial port (SPI, or UART when in API mode) 2 = MicroPython
Data		Variable	

Socket Create - 0x40

Description

Use this frame to create a new socket with the following protocols: TCP, UDP, or TLS.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x40	Byte	
Frame ID		Byte	Reference identifier used to match status responses. A response is required and will be sent regardless of the frame ID.
Protocol		Byte	0 = UDP 1 = TCP 4 = SSL over TCP

Socket Create Response - 0xC0

Description

The device sends this frame in response to a Socket Create (0x40) frame. It contains a socket ID that should be used for future transactions with the socket and a status field.

If the status field is non-zero, which indicates an error, the socket ID will be set to 0xFF and the socket will not be opened.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0xC0	Byte	
Frame ID		Byte	A reference identifier used to match status responses.
Socket ID		Byte	A unique socket ID to address the socket. This field is 0xFF if the value in the status field is non-zero.
Status		Byte	Status code. See table below.

The following table shows the status codes.

Code	Description
0x0	Successful open
0x22	Not registered to cell network
0x31	Internal error
0x32	Resource error: retry the operation later See Socket limits in API mode.
0x7B	Invalid protocol

Socket Option Request - 0x41

Description

Use this frame to modify the behavior of sockets to change their behavior to be different than the normal default behavior. If the Option Data field is zero-length the request acts as a query, and the Socket Option Response frame (0xC1) reports the current effective value.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x41	Byte	
Frame ID		Byte	A reference identifier used to match status responses. Requests made with Frame ID 0 will not send a response.
Socket ID		Byte	The socket ID to modify.
Option ID		Byte	Identifier of the parameter to change.
Option Data		Variable	Variable length field based on option type. If zero length, the current effective value will be returned in the response frame.

Options

Option ID	Option Name	Data Type	Default Value	Description
0x00	TLS Profile	Byte	0x00	Determines the TLS profile to be used: $0 - $2. This is valid only for TLS sockets.

Socket Option Response - 0xC1

Description

Reports the status of requests made with the Socket Option Request (0x41) frame.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0xC1	Byte	
Frame ID		Byte	Identifier provided in request.
Socket ID		Byte	The socket ID for which modification was requested.
Option ID		Byte	Identifier of the parameter requested.
Status		Byte	0x00: Success 0x01: Invalid parameters 0x02: Failed to retrieve option value 0x20: Bad socket ID
Option Data		Variable	Current effective value of the option. This field is only present if the corresponding request was a query (empty value).

Socket Connect - 0x42

Description

Use this frame to connect a socket to the given address and port.

For a UDP socket, this filters out any received responses that are not from the specified remote address and port.

Two frames occur in response:

1. Socket Connect Response frame: Arrives immediately and confirms the request.
2. Socket Status frame: Indicates if the connection was successful.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x42	Byte	
Frame ID		Byte	A reference identifier used to match status responses. If set to **0**, the device does not send a response.
Socket ID		Byte	ID of the socket to connect.
Destination port		16-bit big endian	
Destination address type		Byte	0: Indicates the destination address field is a **binary** IPv4 address in network byte order. 1: Indicates the destination address field is a string containing either a dotted quad value or a domain name to be resolved.
Destination address		Variable	

Socket Connect Response - 0xC2

Description

The device sends this frame in response to a Socket Connect (0x42) frame. The frame contains a status regarding the initiation of the connect.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0xC2	Byte	
Frame ID		Byte	A reference identifier used to match status responses.
Socket ID		Byte	ID of the socket that will be connected.
Status		Byte	Status code. See the table below.

The following table shows the status codes.

Code	Description
0x00	Successfully started the connection process
0x01	Invalid destination address type
0x02	Invalid parameter: address or port
0x03	Connection already in progress
0x04	Already connected
0x05	Unknown error
0x20	Invalid socket ID

Socket Close - 0x43

Description

Use this frame to close a socket when given an identifier.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x43	Byte	
Frame ID		Byte	A reference identifier used to match status responses. If set to **0**, the device does not send a response.
Socket ID		Byte	ID of the socket to be closed.

Socket Close Response - 0xC3

Description

The device sends this frame in response to a Socket Connect (0x43) frame. Since a close will always succeed for a socket that exists, the status can be only one of two values: Success or Bad socket ID.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0xC3	Byte	
Frame ID		Byte	A reference identifier used to match status responses.
Socket ID		Byte	ID of the socket that has been closed.
Status		Byte	0x00 = Success 0x20 = Bad socket ID

Socket Send (Transmit) - 0x44

Description

A Socket Send message causes the device to transmit data using the current connection. For a non-zero frame ID, this will elicit a Transmit (TX) Status - 0x89 frame.

This frame requires a successful Socket Connect - 0x42 frame first. For a socket that is not connected, the device responds with a Transmit (TX) Status - 0x89 frame with an error. To send data from a UDP socket that is not connect, use a Socket SendTo - 0x45 frame.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x44	Byte	
Frame ID		Byte	A reference identifier used to match status responses. If set to **0**, the Transmit (TX) Status - 0x89 frame is disabled.
Socket ID		Byte	ID of the socket to send on.
Transmit options		Byte bit-field	Reserved
Payload		Variable	Data to be transferred to the destination, up to 1500 bytes.

Socket SendTo (Transmit Explicit Data): IPv4 - 0x45

Description

A Socket SendTo (Transmit Explicit Data) message causes the device to transmit data using an IPv4 address and port. For a non-zero frame ID, this will elicit a Transmit (TX) Status - 0x89 frame.

If this frame is used with a TCP, SSL, or a connected UDP socket, the address and port fields are ignored.

You must perform a Socket Bind/Listen - 0x46 frame for a UDP connection before you attempt a SendTo in order to assign a source port.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x45	Byte	
Frame ID		Byte	A reference identifier used to match status responses. If set to **0**, the Transmit (TX) Status - 0x89 frame is disabled.
Socket ID		Byte	ID of the socket to send on.
Destination address		32-bit big endian	
Destination port		16-bit big endian	
Transmit options		Byte bit-field	Reserved
Payload		Variable	Data to be transferred to the destination, up to 1500 bytes.

Socket Bind/Listen - 0x46

Description

Opens a listener socket that listens for incoming connections.

When there is an incoming connection on the listener socket, a Socket New IPv4 Client - 0xCC frame is sent, indicating the socket ID for the new connection along with the remote address information.

For a UDP socket, this frame binds the socket to a given port. A bound UDP socket can receive data with a Socket Receive From: IPv4 - 0xCE frame.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0x46	Byte	
Frame ID		Byte	A reference identifier used to match status responses. If set to **0**, the device does not send a response.
Socket ID		Byte	The socket ID to listen on.
Source port		16-bit big endian	The port to listen on.

Socket Listen Response - 0xC6

Description

The device sends this frame in response to a Socket Bind/Listen (0x46) frame.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0xC6	Byte	
Frame ID		Byte	Resource identifier used to match status responses.
Socket ID		Byte	The socket ID of the socket that has started listening.
Status		Byte	Status code. See table below.

The following table shows the status codes.

Code	Description
0x00	Success
0x01	Invalid port
0x02	Error
0x03	Already bound or listening
0x20	Invalid socket ID

Socket New IPv4 Client - 0xCC

Description

The XBee Cellular modem generates this frame when an incoming connection is accepted on a listener socket.

This frame contains the original listener's socket ID and a new socket ID of the incoming connection, along with the connection's remote address information.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0xCC	Byte	
Socket ID		Byte	The socket ID of the listener socket.
Client Socket ID		Byte	The socket ID of the new connection.
Remote address		32-bit big endian	
Remote port		16-bit big endian	

Socket Receive - 0xCD

Description

The XBee Cellular modem uses this frame when it receives RF data on the specified socket.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0xCD	Byte	
Frame ID		Byte	(Optional) This field allows for solicited reads to be in the future.
Socket ID		Byte	ID of the socket that the data has been received on.
Status		Byte bit-field	Reserved
Payload		Variable	Data received from the destination. It may be up to 1500 bytes.

Socket Receive From: IPv4 - 0xCE

Description

The XBee cellular modem uses this frame when it receives RF data on the specified socket. This frame is sent only for UDP sockets that have not used a Socket Connect - 0x42 frame to connect, providing addressing information about the source.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Field value	Data type	Description
Frame type	0xCE	Byte	
Frame ID		Byte	Optional: This field allows for solicited reads to be in the future.
Socket ID		Byte	ID of the socket that the data has been received on.
Source address		32-bit big endian	
Source port		16-bit big endian	
Status		Byte bit-field	Reserved
Payload		Variable	Data to be transferred to the destination, up to 1500 bytes.

Socket Status - 0xCF

Description

This frame is sent out the device's serial port to indicate the state related to the socket.

Format

The following table provides the contents of the frame. For details on frame structure, see API frame format.

Field name	Size	Description
Frame type	1	Socket Status frame type (0xCF)
Socket ID	1	Socket ID for status reported
Status	1	0x00 = Connected All values other than **0x00 = Connected** are fatal and the Socket ID is closed and invalid after receipt. 0x01 = Failed DNS lookup 0x02 = Connection refused 0x03 = Transport closed 0x04 = Timed out 0x05 = Internal error 0x06 = Host unreachable 0x07 = Connection lost 0x08 = Unknown error 0x09 = Unknown server 0x0A = Resource error

Packaged firmware updates

A packaged firmware update is the process of updating the firmware on the module (microcontroller) and on the cellular component if and only if it is needed. The primary reason for not always updating the cellular component firmware is that it takes 15-20 minutes to complete.

A host program that performs packaged firmware updates over the XBee Cellular Modem's serial interface is required to know of the association between module firmware and cellular component firmware. See About packaged firmware updates.

In addition to knowing which cellular component firmware is required for a given release of the module firmware, the host program needs to know which firmware versions for the module support a cellular component firmware update. For example, if Release 3 is the first version of the module firmware that supports cellular component firmware updates, you must update it before updating the cellular component firmware. But to downgrade from Release 3 or greater to Release 2 or less, you must downgrade the cellular component firmware before downgrading the module firmware. Otherwise, the older firmware would not be able to downgrade the cellular component firmware.

Important notes

Consider the following before performing a cellular component firmware update:

- We recommend using XCTU to update the firmware rather than using the packaged firmware update.

CAUTION! Avoid interrupting the process if possible. An interruption requires starting over. If the interruption occurs while the bootloader is being updated (part number 82004156) the device may not be recoverable.

- When downgrading the module firmware to version 1009 or earlier, Perform a cellular component firmware update before the module firmware is updated.
- When updating to module firmware version 100A or later, Perform a cellular component firmware update after the module firmware is updated.
- With the cellular component firmware updated, the APN is lost from the cellular component configuration, even though it remains on the module configuration. To resolve this, re-enter AN (Access Point Name) and re-apply it for the cellular component to connect to the cellular network.

Perform a cellular component firmware update

The typical method that we recommend for performing a cellular component firmware update is using XCTU. This topic specifies how a host program can perform a cellular component firmware update without XCTU.

The cellular component firmware consists of two entities:

- Part number 82004156, which is the code for the bootloader on the Telit module
- Part number 82004015, which is the cellular code for the Telit module

Just as there is an association between module firmware releases and cellular component firmware releases, there is also an association between bootloader and cellular code for the cellular component. Once it is determined that a cellular component update is needed, the bootloader (part number 82004156) should be updated followed by the cellular code (part number 82004015).

1. Configure the module at a high baud rate. 460,800 (**BD** = **9**) or 921,600 (**BD** = **0xA**) is best to optimize speed.
2. Configure the module in API mode (**AP** = **1**).
3. Set up the host program to a matching baud rate and API mode.

4. Update the bootloader file (part number 82004156)
   a. Send the first block of the file with **ID** set to **0** and bit 0 of the flags byte set to indicate the first frame. The size of the block does not matter as long as it is less than maximum buffer size (1500 bytes).
   b. Wait for an ACK before proceeding. An ACK comes in a FW Update Response - 0xAB with a status of **0**. Under normal conditions, the ACK occurs within 100 ms. However, some responses have been measured to take 80 seconds. To be safe it is best not to timeout on the response for 90 seconds.
   c. Send all but the last frame of the file with incrementing values for the ID and all bits in the Flags field cleared. Wait for an ACK between each frame sent.
   d. Send the last block of the file with the next ID and with bit 1 set to indicate last frame. Wait for an ACK on the final case.
5. Update the cellular code file (part number 82004015) using the same steps as the bootloader file.

After the final ACK is received for both the bootloader file and the cellular code file, the cellular component firmware update is complete.

WARNING! With the cellular component firmware updated, the APN is lost from the cellular component configuration, even though it remains on the module configuration. To resolve this, re-enter AN (Access Point Name) and re-apply it for the cellular component to connect to the cellular network.

As a verification, enter MV (Modem Firmware Version) to reveal the version of the cellular component firmware.

Note The **AI** status must be **0x23** or **0** for **MV** to give a valid response.

About packaged firmware updates

An XBee Cellular Modem contains two processors: a microcontroller that controls most operations of the module, and a cellular component. Both processors contain firmware that you can update. For any given release of the microcontroller firmware (after this referred to as the module firmware), there is an associated release of the cellular component firmware. One or more releases of the module firmware is associated with a given cellular component firmware. However, for a given module firmware, there is only one associated release of the cellular component firmware. The following table depicts an example of this with arbitrary release numbers:

Module firmware	Cellular component firmware
Release 1	Release A
Release 2	Release A
Release 3	Release B
Release 4	Release C

Module firmware	Cellular component firmware
Release 5	Release C
Release 6	Release C
Release 7	Release D

Note The module version number keeps incrementing whether or not the cellular component firmware version increases.

FW Update - 0x2B

Description

Use this frame to send Cellular component firmware updates.

Format

The following table provides the contents of the frame.

Frame data fields	Offset	Type	Description
ID	1	uint8	Will be matched in response. Typically starts at 0, but may start at any number and it must increment with each successive frame (modulo 256).
Component identifier	2	uint8	Set to zero, may be used in the future to identify the target component.
Flags	3	uint8	Bit mask of values indicating various status: bit 0 (0x01) - Initial request. bit 1 (0x02) - Final request (File fully transferred). bit 2 (0x04) - Cancel request (Used to abort an update in progress).
Payload	4	multi-byte	Next section of file being transferred.

FW Update Response - 0xAB

Description

This frame is read from the module and it provides the status for each 0x2B frame sent.

Format

The following table provides the contents of the frame.

Frame data fields	Offset	Type	Description
ID	1	uint8	Value from request payload.
Status	2	uint8	Enumeration of status values: 0 - Success >0 - errors ■ 1 - Operation cancelled ■ 2 - Update in progress ■ 3 - Update not started ■ 4 - Sequence error ■ 5 - Internal error ■ 6 - Resource error

Error recovery

Several different types of errors can occur:

Corrupted firmware on the cellular component

If something goes wrong during a firmware update, (such as a loss of power), there is a good chance that the firmware on the cellular component will be corrupted. This is indicated by an **AI** status of **0x24**. If you see this status, reset the module (using **FR** for example) and then follow the steps in Perform a cellular component firmware update to redo the cellular component firmware update.

Error

An error occurs when FW Update Response - 0xAB returns a non-zero status code. This can be caused by a programming error on the host side (for example out of order sequence numbers), a software error on the module side (for example too short of a timeout waiting for responses from the cellular component), or an invalid image of the cellular component firmware. In any of these cases, the firmware update is aborted such that it cannot be picked up from where it left off. The only reliable recovery is to reset the module and then immediately Perform a cellular component firmware update.

Host initiated cancellation

If the host sets bit 2 of the flags byte in FW Update - 0x2B, the update in progress is aborted. Recovery is then equivalent to the recovery for negative acknowledgments, described above.

General case

Regardless of the reason for the error, a cellular component firmware update should always work within ten seconds of a reset and after **AI** is **0x23** or **0**.

Troubleshooting

This section contains troubleshooting steps for the XBee Cellular Modem.

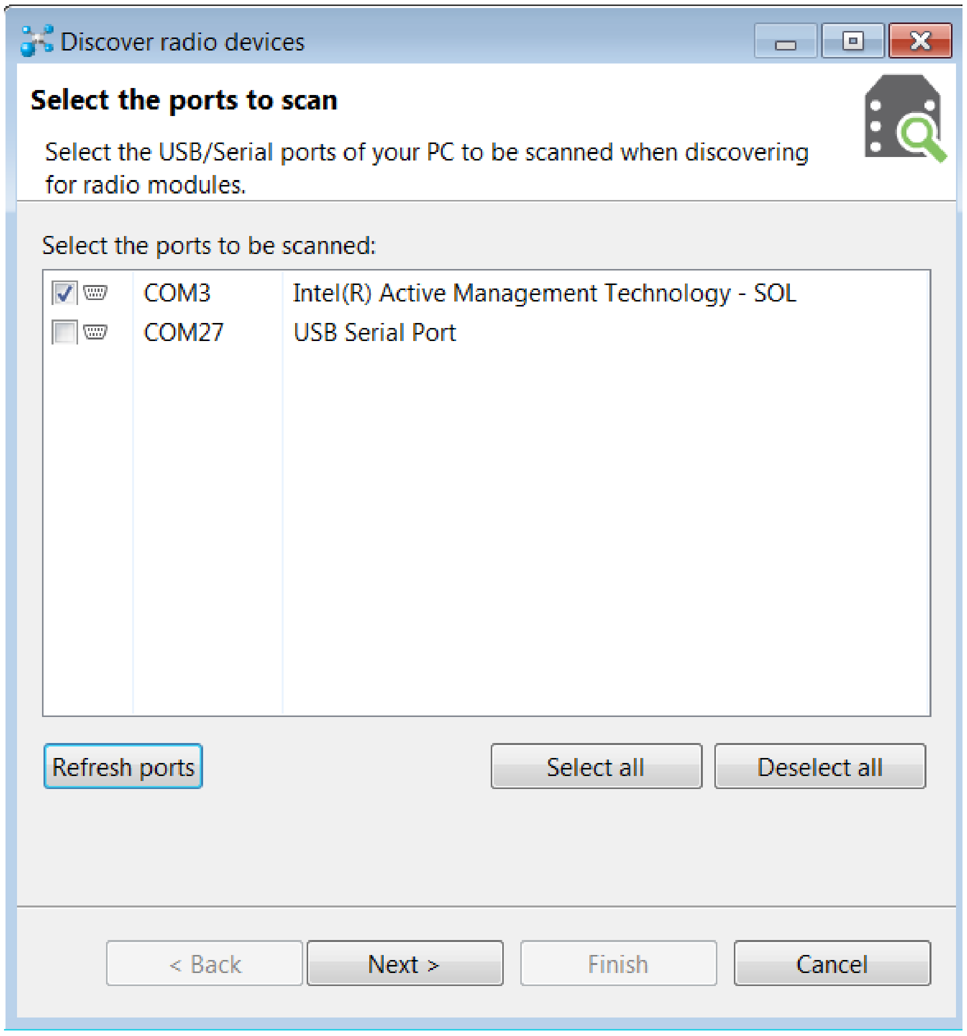
Cannot find the serial port for the device

Condition

In XCTU, the serial port that your device is connected to does not appear.

Solution

1. Click the **Discover radio modules** button.
2. Select all of the ports to be scanned.
3. Click **Next** and then **Finish**. A dialog notifies you of the devices discovered and their details.


Discover radio devices

Select the ports to scan

Select the USB/Serial ports of your PC to be scanned when discovering for radio modules.

Select the ports to be scanned:

✓	COM3	Intel(R) Active Management Technology - SOL
☐	COM27	USB Serial Port

4. Remove the development board from the USB port and view which port name no longer appears in the **Discover radio devices** list of ports. The port name that no longer appears is the correct port for the development board.

Other possible issues

Other reasons that the XBee Cellular Modem is not discoverable include:

1. If you accidentally have the loopback pins jumpered.
2. You may not have a driver installed. If you do not have a driver installed, the item will have an exclamation point icon next to it in the Windows Device Manager.
3. You may not be using an updated FTDI driver.
    a. Click here to download the drivers for your operating system.
    b. This may require you to reboot your computer.
    c. Disconnect the power and USB from the XBIB-U-DEV board and reconnect it.
4. If you have a driver installed and updated but still have issues, on Windows 10 you may have to enable VCP on the driver; see Enable Virtual COM port (VCP) on the driver.

Enable Virtual COM port (VCP) on the driver

On Windows 10 computers, if XCTU does not see the devices you have attached to a PC, you may need to enable VCP on the USB driver.

To enable VCP:

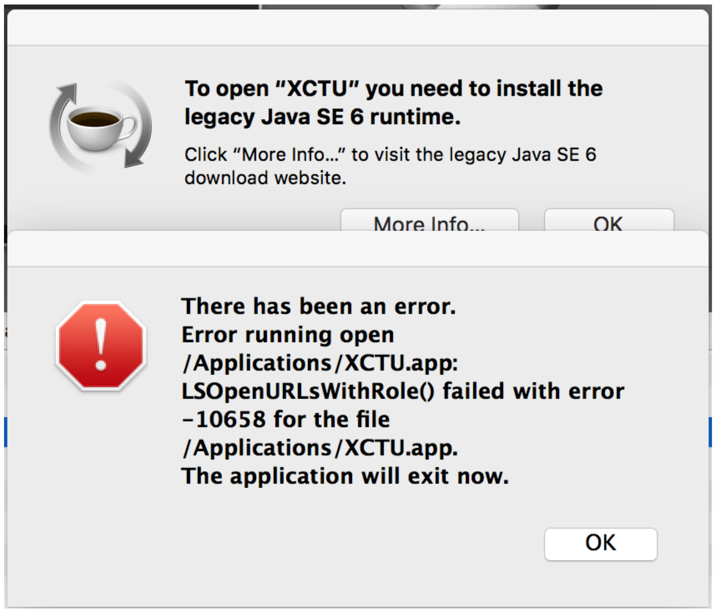
1. Click the **Search** button.
2. Type **Device Manager** to search for it.
3. Click **Universal Serial Bus controllers**.
4. If it displays more than one USB controller, unplug the XBee Cellular Modem and plug it back in to make sure you choose the correct one.
5. Right-click the USB controller and select **Properties**; a dialog displays.
6. Select the **Advanced** tab.
7. Check **Load VCP**.
8. Click **OK**.
9. Unplug the board and plug it back in.

Correct a macOS Java error

When you use XCTU on macOS computer, you may encounter a Java error.

Condition

When opening XCTU for the first time on a macOS computer, you may see the following error:



To open "XCTU" you need to install the legacy Java SE 6 runtime.

Click "More Info..." to visit the legacy Java SE 6 download website.

**There has been an error.
Error running open
/Applications/XCTU.app:
LSOpenURLsWithRole() failed with error
-10658 for the file
/Applications/XCTU.app.
The application will exit now.**

Solution

1. Click **More info** to open a browser window.
2. Click **Download** to get the file javaforosx.dmg.
3. Double-click on the downloaded javaforosx.dmg.
4. In the dialog, double-click the JavaForOSX.pkg and follow the instructions to install Java.

Unresponsive cellular component in Bypass mode

When in Bypass mode, the XBee Cellular Modem does not automatically reset or reboot the cellular component if it becomes unresponsive.

Condition

In Bypass mode, the XBee Cellular Modem does not respond to commands.

Solution

1. Query the AI (Association Indication) parameter to determine whether the cellular component is connected to the XBee Cellular Modem software. If **AI** is **0x2F**, Bypass mode should work. If not, look at the status codes in AI (Association Indication) for guidance.
2. You can send the !R (Modem Reset) command to reset only the cellular component.

Not on expected network after APN change

Condition

The XBee Cellular Modem is not on the expected network after a change to the AN (Access Point Name) command.

Solution

Send **ATNR0** to reset Internet connectivity. See NR (Network Reset) for more information.

Syntax error at line 1

You may get a **syntax error at line 1** error after pasting example MicroPython code and pressing **Ctrl**+**D**.

Solution

This commonly happens when you accidentally type a character at the beginning of line 1 before pasting the code.

Error Failed to send SMS

In MicroPython, you consistently get **Error Failed to send SMS** messages.

Solution

Your device cannot connect to the cell network. The reason may be:

1. The antenna is improperly or loosely connected.
2. The device is at a location where cellular service cannot reach. If the device is connected to the network, the red LED blinks about twice in a second. If it is not connected it does not blink; see Associate LED functionality.
3. You SIM card is out of SMS text quota.
4. The device is not getting enough current, for example if power is being supplied only by USB to the XBIB development board, rather than using an additional external power supply.

Regulatory information

Modification statement

Digi International has not approved any changes or modifications to this device by the user. Any changes or modifications could void the user's authority to operate the equipment.

Digi International n'approuve aucune modification apportée à l'appareil par l'utilisateur, quelle qu'en soit la nature. Tout changement ou modification peuvent annuler le droit d'utilisation de l'appareil par l'utilisateur.

Interference statement

This device complies with Part 15 of the FCC Rules and Industry Canada license-exempt RSS standard (s). Operation is subject to the following two conditions: (1) this device may not cause interference, and (2) this device must accept any interference, including interference that may cause undesired operation of the device.

Le présent appareil est conforme aux CNR d'Industrie Canada applicables aux appareils radio exempts de licence. L'exploitation est autorisée aux deux conditions suivantes : (1) l'appareil ne doit pas produire de brouillage, et (2) l'utilisateur de l'appareil doit accepter tout brouillage radioélectrique subi, même si le brouillage est susceptible d'en compromettre le fonctionnement.

FCC notices

IMPORTANT: XBee modules have been certified by the FCC for use with other products without any further certification (as per FCC section 2.1091). Modifications not expressly approved by Digi could void the user's authority to operate the equipment.

IMPORTANT: OEMs must test final product to comply with unintentional radiators (FCC section 15.107 & 15.109) before declaring compliance of their final product to Part 15 of the FCC Rules.

IMPORTANT: The RF module has been certified for remote and base radio applications. If the module will be used for portable applications, the device must undergo SAR testing.

This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation.

If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures: Re-orient or relocate the receiving antenna, Increase the separation between the equipment and receiver, Connect equipment and receiver to outlets on different circuits, or Consult the dealer or an experienced radio/TV technician for help.

FCC Class B digital device notice

This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- Reorient or relocate the receiving antenna.
- Increase the separation between the equipment and receiver.
- Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.
- Consult the dealer or an experienced radio/TV technician for help.

Labeling requirements for the host device

The device shall be properly labeled to identify the product within the host device. For more information, see the Regulatory Approvals table.

The certification labels of the module shall be clearly visible at all times when installed in the host device, otherwise the host device must be labeled to display the FCC ID and IC of the module, preceded by the words "Contains transmitter module", or the word "Contains", or similar wording expressing the same meaning, as follows:

Contains FCC ID: RI7LE866SV1

Contains IC: 5131A-LE866SV1

or

Contains FCC ID: RI7LE866SV1A

Contains IC: 5131A-LE866SV1A

L'appareil hôte doit être étiqueté comme il faut pour permettre l'identification des modules qui s'y trouvent. Pour plus d'informations, reportez-vous au tableau des approbations réglementaires.

L'étiquettes de certification du module donné doit être posée sur l'appareil hôte à un endroit bien en vue en tout temps. En l'absence d'étiquette, l'appareil hôte doit porter une étiquette donnant le FCC ID et le IC du module, précédé des mots « Contient un module d'émission », du mot « Contient » ou d'une formulation similaire exprimant le même sens, comme suit:

Contains FCC ID: RI7LE866SV1

Contains IC: 5131A-LE866SV1

ou

Contains FCC ID: RI7LE866SV1A

Contains IC: 5131A-LE866SV1A

CAN ICES-3 (B) / NMB-3 (B)

This Class B digital apparatus complies with Canadian ICES-003.

Cet appareil numérique de classe B est conforme à la norme canadienne ICES-003.

FCC publication 996369 related information

In publication 996369 section D03, the FCC requires information concerning a module to be presented by OEM manufacturers. This section assists in answering or fulfilling these requirements.

2.1 General

No requirements are associated with this section.

2.2 List of applicable FCC rules

This module conforms to FCC Parts 27(cellular).

2.3 Summarize the specific operational use conditions

Certain approved antennas require attenuation for operation. For the XBee Cellular Modem, see Antenna specifications.

Host product user guides should include the antenna table if end customers are permitted to select antennas.

2.4 Limited module procedures

Not applicable.

2.5 Trace antenna designs

While it is possible to build a trace antenna into the host PCB, this requires at least a Class II permissive change to the FCC grant which includes significant extra testing and cost. If an embedded trace or chip antenna is desired contact a Digi sales representative for information on how to engage with a lab to get the modified FCC grant.

2.6 RF exposure considerations

For RF exposure considerations see RF exposure.

Host product manufacturers need to provide end-users a copy of the “RF Exposure” section of the manual: RF exposure.

2.7 Antennas

A list of approved antennas is provided for the XBee Cellular Modem. See Antenna specifications.

2.8 Label and compliance information

Host product manufacturers need to follow the sticker guidelines outlined in Labeling requirements for the host device .

2.9 Information on test modes and additional testing requirements

Contact a sales representative for information on how to configure test modes for the XBee Cellular Modem.

2.10 Additional testing, Part 15 Subpart B disclaimer

All final host products must be tested to be compliant to FCC Part 15 Subpart B standards. While the XBee Cellular Modem was tested to be complaint to FCC unintentional radiator standards, FCC Part 15 Subpart B compliance testing is still required for the final host product. This testing is required for all end products, and XBee Cellular Modem Part 15 Subpart B compliance does not affirm the end product’s compliance.

See FCC notices.