

# **XBee Cookbook Issue 1.4 for Series 1 (Freescale) with 802.15.4 Firmware**

John Foster

April 26, 2011



# Contents

<b>1</b>	<b>Features</b>	<b>1-1</b>
<b>2</b>	<b>Electrical Specifications</b>	<b>2-1</b>
2.1	Pinout . . . . .	2-1
2.2	I/O Lines . . . . .	2-1
2.2.1	All I/O Lines . . . . .	2-1
2.2.2	All Inputs . . . . .	2-2
2.2.3	Digital Inputs . . . . .	2-3
2.2.4	Digital Outputs . . . . .	2-3
2.2.5	Analog Inputs and VREF . . . . .	2-3
2.2.6	Analog Outputs . . . . .	2-3
2.2.7	I/O Line Passing . . . . .	2-4
2.2.8	Serial Lines . . . . .	2-4
2.2.9	Pin 5 – RESET . . . . .	2-5
2.3	Voltage Level Conversions . . . . .	2-5
<b>3</b>	<b>Operation</b>	<b>3-1</b>
3.1	Version and Device Information . . . . .	3-1
3.2	Module Mode . . . . .	3-1
3.2.1	Transparent Mode . . . . .	3-1
3.2.2	API Mode . . . . .	3-1
3.3	Module State . . . . .	3-2
3.3.1	Idle State . . . . .	3-2
3.3.2	Sleep State . . . . .	3-2
3.3.3	Command State . . . . .	3-2
3.4	Flow Control . . . . .	3-3
3.4.1	Hardware Flow Control . . . . .	3-3
3.4.2	Software Flow Control . . . . .	3-3
<b>4</b>	<b>Networking</b>	<b>4-1</b>
4.1	Network Types . . . . .	4-1
4.2	NonBeacon . . . . .	4-1
4.3	NonBeacon w/Coordinator . . . . .	4-1
4.4	Transmission Modes . . . . .	4-1
4.5	Collision Avoidance . . . . .	4-1
4.6	MAC Mode . . . . .	4-2
4.7	Network Addressing . . . . .	4-2
4.8	Network Identification and Scanning . . . . .	4-2
4.9	Network Association . . . . .	4-3
4.10	Encryption . . . . .	4-3
4.11	Power Levels . . . . .	4-3
<b>5</b>	<b>API Packets</b>	<b>5-1</b>
5.1	All Packets . . . . .	5-1
5.2	Modem Status Packet . . . . .	5-1
5.3	Local AT Command Packets . . . . .	5-2

5.3.1	AT Command . . . . .	5-2
5.3.2	AT Response . . . . .	5-2
5.4	Remote AT Command Packets . . . . .	5-3
5.4.1	Remote AT Command . . . . .	5-3
5.4.2	Remote AT Response . . . . .	5-3
5.5	Input Line States . . . . .	5-4
5.6	Transmit Request . . . . .	5-4
5.6.1	TX Request . . . . .	5-4
5.6.2	RX . . . . .	5-5
5.6.3	TX Status . . . . .	5-5
<b>A</b>	<b>Baud Rates and their Accuracy</b>	<b>A-1</b>
A.1	System Clock Frequency and UART Clock . . . . .	A-1
A.2	Available Baud Rates . . . . .	A-1
A.3	Serial Line Receiver Operation . . . . .	A-2
A.4	Signal Quality . . . . .	A-2
A.5	Solution at 115,200 baud . . . . .	A-2
A.6	Discussion of 57,600 Baud . . . . .	A-3
A.7	Baud Rate Accuracy on Other XBees . . . . .	A-3
A.8	Historical Note . . . . .	A-3
<b>B</b>	<b>A Study of Custom Baud Rates on the Series 1 XBee</b>	<b>B-1</b>
B.1	Introduction . . . . .	B-1
B.2	The Calculation of Percentages . . . . .	B-1
B.3	Standard Baud Rates . . . . .	B-2
B.4	How Close do Baud Rates Need to Be? . . . . .	B-2
<b>C</b>	<b>Using X-CTU under Linux</b>	<b>C-1</b>
<b>D</b>	<b>Change Log</b>	<b>D-1</b>
D.1	Changes . . . . .	D-1
D.1.1	Changes in issue 1.4 (published 26 Apr 2011) . . . . .	D-1
D.1.2	Changes in issue 1.3 (published 9 Mar 2011) . . . . .	D-1
D.1.3	Changes in Issue 1.2 (published 7 Mar 2010) . . . . .	D-1
D.1.4	Changes in Issue 1.1 (published 26 Jan 2010) . . . . .	D-2
D.1.5	Issue 1.0 . . . . .	D-2
<b>Index</b>		<b>1</b>

# Chapter 1

## Features

The XBee and XBee Pro are plug-in replacements for each other. The difference between them is that the XBee Pro operates at higher power and has a more sensitive receiver. Table 1.1 shows the relevant figures. Note that in Europe the output power of the XBee Pro must be restricted to 10dBm. The current of 137mA corresponds to that level.

In each case, the RF data rate is 250Kbps and the serial I/O rate is selectable between 1200 and 115200 baud.

Item	XBee	XBee Pro
Range, indoor/urban	30m	100m
Range, outdoor/line of sight	100m	1 mile
Transmit power	1mW	100mW
Receiver sensitivity	-92dBm	-100dBm
TX current (@3.3V)	45mA	137mA
RX current (@3.3V)	50mA	55mA
Power-down current	< 10 $\mu$ A	< 10 $\mu$ A
Sequence channels	16	12

Table 1.1: Differences between XBee and XBee Pro.



## Chapter 2

# Electrical Specifications

### 2.1 Pinout

If the XBee is viewed from the top and with the bevelled end facing north, then pins 1 to 10 run down the west side from north to south, and pins 11 to 20 run up the east side from south to north. Table 2.1 shows the pinout. Pins that are not used should be left disconnected.

### 2.2 I/O Lines

#### 2.2.1 All I/O Lines

Eight of the I/O lines are fitted with pull-up resistors which can be enabled or disabled with the **PR parameter**. This is an 8-bit mask, where a 0 disables the pull-up and a 1 enables it. Table 2.1 gives the bit positions in the column labelled PR. The default value is 0xff.

I/O pins are configured using the **D<sub>n</sub> parameter**. The command is followed by the pin code from the AT column of Table 2.1, and the command assigns a mode code which is 0 to disable the line, 1 to enable the line's special function (see next para), 2 for analog input, 3 for digital input, 4 for digital output low and 5 for digital output high. For example, the command to set AD1/DIO1 (pin 19) as an analog input would be ATD1=2.

I/O lines with special functions have those functions enabled by setting their **D<sub>n</sub> parameter** to 1. These lines are line 7 (CTS), 6 (RTS) and 5 (Associated indicator). For lines 7 and 5, this is the default state. For line 6, the default state is disabled. Changes made via the D<sub>n</sub> parameter become effective the next time an AC command is issued.

Under firmware version 1084, any attempt to read the D<sub>n</sub> parameter for  $0 \leq n \leq 4$  just got an "ERROR" return. From version 10C8 the response is as it should be.

The Associated indicator output will blink a LED while the module is associated.

To disable any I/O line, set its D<sub>n</sub> parameter to 0.

#### I/O Data Packet

The **IS parameter** forces a read of all enabled inputs and outputs (analog or digital) with the data being returned through the UART. If no inputs are enabled, the command returns "ERROR". The IS command can take a parameter, with a value in the range 1 to 0xff and a default value of 1. The product manual is silent on the question of what the parameter is for, but when I asked Digi they replied that the parameter has no purpose and should be removed in a future firmware release. Under firmware version 10C8 an attempt to read the IS parameter will return "ERROR" unless at least one line is configured as a digital or analog input. In API mode, the data will be returned as a pair of packets. The first will be a status return, and presumably an error status means the second packet won't be present. The second packet is an I/O data packet.

An I/O data packet is also sent when the **IC parameter** has any bits set. This parameter enables change detection on DIO lines 7 to 0 when the corresponding bits are 1. Any change on any enabled line first causes transmission of any queued data, and then causes the DIO data to be transmitted. The default value for the IC parameter is 0x00. This packet will report only digital line states.

A module receiving a packet sent because of the IC parameter will respond in one of two ways:

Pin	Name	Direction	Description	AT	PR
1	VCC	–	Power supply		
2	DOUT	Output	UART data out		
3	DIN / CONFIG	Input	UART data in		7
4	DO8	Output	Digital output 8		
5	RESET	Input	Module reset (at least 200nS)		
6	PWM0 / RSSI	Output	PWM output 0 / RX signal strength indicator		
7	PWM1	Output	PWM output 1		
8	(reserved)		Do not connect		
9	DTR / SLEEP_RQ / DI8	Input	Pin sleep control line or digital input 8	D8	6
10	GND	–	Ground		
11	AD4 / DIO4	Either	Analog input 4 or digital I/O 4	D4	0
12	CTS / DIO7	Either	Clear to send flow control or digital I/O 7	D7	
13	ON / SLEEP	Output	Module status indicator		
14	VREF	Input	Voltage reference for AD inputs		
15	Associate / AD5 / DIO5	Either	Associated indicator, analog input 5 or digital I/O 5	D5	
16	RTS / AD6 / DIO6	Either	RTS flow control, analog input 6 or digital I/O 6	D6	5
17	AD3 / DIO3	Either	Analog input 3 or digital I/O 3	D3	1
18	AD2 / DIO2	Either	Analog input 2 or digital I/O 2	D2	2
19	AD1 / DIO1	Either	Analog input 1 or digital I/O 1	D1	3
20	AD0 / DIO0	Either	Analog input 0 or digital I/O 0	D0	4

Table 2.1: Pinout for XBee and XBee Pro. The AT column gives the tail of the  $Dn$  command for configuring the pin, and the PR column gives the bit number in the PR command for configuring the pull-up resistor (see section 2.2.1). A note in the 1xAx product manual says that pin 4 (DO8) is unsupported, and the 1xEx product manual says the same.

1. If the receiving module's **IA parameter** is set to enable control by the sending module by holding the sending module's address or by holding the "accept all" value, then it will update its own I/O wires from the data in the packet.
2. If the receiving module is not set to update its outputs, then the I/O packet is transmitted through the receiving module's UART.

When an IO Data Packet is sent in transparent mode, all bytes are first converted to ASCII. Then the values are reported, each terminated by a carriage return. The order is:

1. The number of sample sets to follow. Items 2 to 4 in this list constitute one sample set.
2. The channel indicator with a bit set for each enabled input or output. This is a 16-bit word, with the most significant byte sent first. In the high byte, bit 7 is not used and bits 6 to 1 are set if the data will include samples from ADC5 to ADC0 respectively. Bit 0 of the high byte is set if the data will include the state of line D8. In the low byte, bits 7 to 0 correspond to lines DIO7 to DIO0.
3. The DIO data (line state) for each enabled digital input or output, if there are any enabled digital inputs or outputs. This is a 16-bit quantity, with the D8 state in bit 0 of the msb and bits 7–0 in the lsb.
4. The ADC data for each enabled analog input, in ascending order of line number. Each is an unsigned 16-bit quantity, with the 10-bit value in the low-order 10 bits.
5. The data is terminated by an extra carriage return.

### 2.2.2 All Inputs

The **IR parameter** sets or reads the sample rate. When set to a value other than 0, the module will sample all enabled inputs at time intervals given by the parameter value. The time unit is mS, and the maximum value is 0xffff. The default is 0. A note in the Digi product manual says that sampling at a rate of more than one sample per 20mS is not recommended.

The **IT parameter** sets or reads how many samples (max) will be transmitted per packet. Its value is in the range 1 to 0xff, with 1 as the default. When sleep modes are enabled and a sample rate (IR) is set, the module will remain awake until IT samples have been collected.



### 2.2.3 Digital Inputs

Digital inputs will recognize a voltage lower than  $0.35 * VCC$  as low, or greater than  $0.7 * VCC$  as high. The input leakage current for any input voltage between 0 and VCC is typically  $0.025\mu A$ , max  $1\mu A$ . The input leakage current spec is the same, whether or not the input is configured for high impedance. To configure I/O line  $n$  as a digital input, use the **Dn parameter** with value 3.

### 2.2.4 Digital Outputs

When a digital output is in the low state, it can sink up to 2mA with the voltage level being no higher than 0.5V. In the high state, it can source 2mA with the voltage being no lower than  $VCC - 0.5$ . A Digi reply quoted in the 802.15.4 forum says that the outputs can source about 5mA each, with an overall limit of 20mA total.

To configure I/O line  $n$  as a digital output, use the **Dn parameter** with value 4 for low voltage, and 5 for high voltage. Line 8 cannot be used as a digital output in firmware versions earlier than 1xEx.

The **IO parameter** sets the output levels for lines configured as digital outputs. It's an 8-bit mask. The mask value overrides the **Dn parameter**, which remains as the default. Under versions 10C8 and 10E6 any attempt to read this parameter simply returns "OK" in either AT or API mode, so it must be treated as write-only. Setting the IO parameter has immediate effect.

The IO parameter has a curious side effect: if you give an IO command without a parameter value, it seems to behave as though you passed a value of zero, setting all digital outputs to their low state. Once in this state you can't restore an output whose **Dn parameter** was set to high just by re-issuing its **Dn command**. You have to set it low and then set it high again. I've observed this behaviour on a remote XBee using remote AT commands via API under firmware 10E6. Specifically, the DIO4 line was the one I observed. More extensive testing is left as an exercise for the reader.

### 2.2.5 Analog Inputs and VREF

The voltage on any analog input pin must be at least  $V_{SSAD} - 0.3$  and at most  $V_{DDAD} + 0.3$ . That's the electrical limit, not the range for the conversion to be valid. Any analog input must be filtered, with a capacitor in the range  $0.01\mu F$  to  $0.1\mu F$  connected between the input pin and VREF. Analog input pins must be driven from a source whose impedance has a real component of at most  $10k\Omega$ . An analog input pin whose voltage is VREF will be read as 0x3FF, as will any higher voltages.

To configure I/O line  $n$  as an analog input, set the **Dn parameter** to 2. Only lines 0 to 5 can be used in this way.

The voltage supplied to the VREF pin must be at least 2.08V and at most  $V_{DDAD}$ . The module will draw  $200\mu A$  from the source when A-D is enabled, or at most  $0.02\mu A$  when A-D is disabled or the module is in sleep mode.

Historical note: According to the 1xAx documentation, the XBee Pro has an internal voltage reference at VCC, and the **AV parameter**, if 1, causes the internal reference to be used instead of the VREF pin. The default is 0. If the AV command is issued on a non-Pro Xbee, it will return "ERROR". BUT a note in the hardware forum from Travish (a Digi support person) says that the internal reference was removed in the 1x9x beta versions, and the AV command should have been removed from the documentation. So all Series 1 XBees now have to have an external reference.

### 2.2.6 Analog Outputs

The PWM period is  $64\mu S$ , and there are 1023 (0x3ff) steps within it. To put that another way, the PWM outputs deliver a square wave at a frequency of about 15.6KHz with a duty cycle that can be varied in 1024 steps between 0% and 100%.

The **Pp parameter** sets the function of the PWM $p$  pin. A value of 0 disables it, 1 sets it for RSSI and 2 sets it as a PWM output. The default is 1 for P0 and 0 for P1. In RSSI mode, the PWM output indicates received signal strength.

The **Mp parameter** is the PWM output level for the PWM $p$  pin. The value range is 0 to 0x03ff and the default is 0. The parameter is write-only in firmware versions up to at least 10E6: any attempt to read it returns "OK" status but no value. The product manual, however, says you can set or read it.

The **RP parameter** gives the pulse duration for either PWM pin, when in RSSI mode. The duty cycle is updated with each received packet and is shut off when the timer expires. Timer values are in the range 0 to 0xff and the time unit is 100mS. The default value is 0x28. See the product manual for further details.

A cautionary note on analog outputs: with versions of firmware up to at least 1xE6, any change of setting to a DIO pin state or to sample rate, change detect etc will as a side effect reset the analog outputs to 0. Thus after any such change any active analog output values must be re-asserted. (Thanks are due to Brarb for posting this info on the forum.)

Further note: in Feb 2010 Caretjic reported in the forum that with firmware 10CD the analog outputs were not giving the advertised PWM signals. An upgrade to 10E6 led to success.

463f		464a	
IR	5 secs	IA	0xffff
IT	1		
DH	0		
DL	0x1234	MY	0x1234
D0	output		
D1	input	D1	output
IC	0x02	T1	2 secs

Table 2.2: I/O line passing between two XBees, using periodic sampling. The top line shows the last four hex digits of their serial numbers. The one on the left is passing its DIO1 input to the one on the right. On the left, output DIO0 is wired to input DIO1.

### 2.2.7 I/O Line Passing

I/O line passing creates “virtual wires” between two XBees, so that selected inputs on one (the source) are reflected as output states on the other (the destination).

To enable I/O line passing, the source must from time to time transmit the required input values to the destination. There are two possible ways to do this:

1. Use the **IR parameter** and the **IT parameter** to set a sampling rate. Table 2.2 shows the settings for a pair of XBees using this method.
2. Use the **IC parameter** to enable change detection on the input lines.

The **IA parameter** controls permissions for other modules to order this module to change the states of its output lines. The value is a 64-bit module address, and only that module is allowed to change the line states. The value may also be set to 0xffffffff (the default), which will prevent any other module from changing the lines, or to 0xffff which will allow any other module to change the lines. The X-CTU help box says this is for I/O line passing. It doesn’t mention remote AT commands.

The **Tn parameter** ( $n$  in 0–7) sets timeout values for output lines whose values are set through I/O line passing (*not* through the IO parameter). When any such line is set to its non-default value, its timer is started and the line will revert to the default value as set by the  $Dn$  parameter when the timer expires. The values are in the range 0 to 0xff and the time unit is 100mS. The default value is 0xff (25.5 seconds). If the timer value is 0, lines retain their levels indefinitely.

The **PT parameter** is a common timeout value for both analog outputs. When its value is non-zero, each time a PWM level is set through I/O line passing, the timer will be restarted. When it expires, the PWM lines will be reset to 0V. Timer values are in the range 0 to 0xff and the time unit is 100mS. The default value is 0xff (25.5 seconds).

The **IU parameter** controls what happens to line state packets. It defaults to 1, which means that any line state packets received by the XBee will be sent through the UART to the XBee’s controller and will not affect local output lines. If IU is set to 0, received line state packets will affect local output pins and will not be passed through the UART.

### 2.2.8 Serial Lines

The serial lines operate at logic levels, with the idle state being high voltage. Each data byte is transmitted with one start bit (low voltage), eight data bits (low voltage for 0, high for 1, lsb first), an optional parity bit (but none by default) and one stop bit (high voltage). The baud rate is controlled by the **BD parameter**, which defaults to 3 (9600 baud). Table 2.3 shows the possible values. For non-standard baud rates, set the BD value in the range 0x08 to 0x1c200 and then read back the BD value. The value read back will be the actual baud rate achieved. The baud rate setting will not change until AT mode is exited using the **CN command**, and the module has returned the “OK” response.

Operation at 115200 baud is flaky with the standard settings, as documented in a forum topic “XBee Series 1 not working well at 115200 baud”, around September 2008<sup>1</sup>. The main suggestion was to use two stop bits. Other suggestions were to operate the host at 112000 baud or 111111 baud, using one stop bit. The latter value can be achieved on a microprocessor host by dividing down the clock if the clock has a round value: for example, if the clock is 1MHz, divide by 9. See Appendix A for a lot more information on this topic.

<sup>1</sup>Currently (Jan 2010) a pinned topic in the 802.15.4 forum.

BD	Baud
0	1200
1	2400
2	4800
3	9600
4	19200
5	38400
6	57600
7	115200

Table 2.3: BD command values and standard baud rates.

Parity settings are controlled by the **NB parameter**. Values are in the range 0 to 4, where 0 means no parity, 1 means even, 2 means odd, 3 means mark and 4 means space. The default is 0. Note that setting NB to 3 effectively causes the XBee to send two stop bits.

For details of flow control, see Section 3.4 on page 3-3..

### 2.2.9 Pin 5 – $\overline{\text{RESET}}$

This pin has a 50k $\Omega$  pull-up resistor within the module.

## 2.3 Voltage Level Conversions

There are two main cases where voltage level conversions are needed. One is where the XBee serial lines are to be connected to an RS-232 port, and the other is where any lines including the serial lines are to be connected to a 5V device such as a PIC microcontroller.

For the RS-232 case, the MAX232 family of chips does the job just fine.

For the 5V case, a couple of resistors connected as a voltage divider is enough to connect a 5V output to an XBee input. An XBee output will normally need a couple of transistors for the conversion, or a single transistor if you don't mind the signal being inverted. There are also chips available for this.

When driving a 5V input from an XBee it's worth checking the spec for the 5V circuit – some may be compatible with 3.3V logic. At least one post in the forum has suggested that the PIC is an example of a circuit that does need level conversion.



# Chapter 3

## Operation

### 3.1 Version and Device Information

The **VR command** returns the firmware version of the RF module, as a value in the range 0 to 0xffff.

The **VL command** used to return a verbose description of versions, including the application build date, MAC, PHY and bootloader versions. Under firmware version 10C8, this command cannot be used as an API command. If you try it as a local command it returns error status, and as a remote command it gets no response at all. In fact, the 1xEx product manual says the command was removed from firmware version 1xC9 and all later versions.

The **HV command** returns the hardware version of the RF module, in the range 0 to 0xffff. On my ordinary XBees it returns 1742 (1744 on a few that I bought more recently), and on the Pro it returns 1842. A forum post by gworle of Digi says that plain series 1 XBees will return 17xx and Pros will return 18xx. For the series 2 modules, a plain one will return 19xx and a Pro will return 1axx.

The **DD command** appears to be new in firmware version 1xCx. According to X-CTU, it returns the “device type identifier value”. On my XBees the value is 0x10000, for both the normal and Pro versions. The DD command is not mentioned in the 1xCx product manual, nor in the 1xEx manual.

The **CK command** was introduced in version 10e4 according to the firmware revision history. It isn’t mentioned in the product manual for 10e6. It returns the checksum of the configuration in RAM, as an aid to determining whether the configuration has changed. The value is a single byte.

### 3.2 Module Mode

The module mode controls how the XBee responds to data received via its UART. There are two possible modes: transparent (the default) and API. In API mode there are two sub-modes, allowing data to be sent as straight binary or with certain characters escaped. The **AP command** selects the mode. A value of 0 (default) selects transparent mode, 1 enables API mode, and 2 enables API mode with escaped control characters.

#### 3.2.1 Transparent Mode

Transparent mode is enabled by default. Any RF data received by the module will be transmitted out through the UART, and any data received from the UART will be transmitted over RF. The module has a limited buffer size for this data (100 bytes), so with high transfer rates flow control must be implemented.

The **RO command** affects the packetization timeout. Its value lies in the range 0 to 0xff, and it gives the number of character times to wait after the last character was received, before transmitting the received data as a packet. When the value is 0, all characters received are transmitted immediately. The default value is 3.

#### 3.2.2 API Mode

In API mode, all data entering and leaving the module is contained in frames that cause operations or events within the module. Frames received through pin 3 (DI) are called Transmit Data Frames. Frames sent out through pin 2 (DO) are called Receive Data Frames.

From the command state, the **AP command** enables or disables API mode. A value of 0 disables it, 1 enables it, and 2 enables it with escaped control characters. The default value is 0. Operation with escaped control characters would be necessary if using XON/XOFF flow control.

### 3.3 Module State

At any given time, a module is in one of five states: idle, transmit, receive, command or sleep. From the idle state it can switch to any other state, and from any other state it can switch back to idle.

#### 3.3.1 Idle State

From the idle state:

- Serial data received in the DI buffer will switch the state to transmit.
- Valid data received through the RF antenna will switch to receive.
- If the sleep mode condition is met, the module will switch to sleep.
- If a command mode sequence is issued, the module will switch to command.

#### 3.3.2 Sleep State

Note that in my application so far I don't use sleep modes. The information given here is therefore sketchy compared to what the product manual contains.

Sleep is a state in which power consumption is low. For sleep mode to be entered, sleep must be enabled using the **SM parameter**, and either pin 9 (Sleep\_RQ) must be asserted or the module must have been idle for the length of time specified by the **ST parameter** (Time Before Sleep). The parameter gives the time in milliseconds. It takes values in the range 1 to 0xffff, and the default value is 0x1388 (5 seconds).

Sleep is disabled by setting SM=0. There are four other SM values: SM=1 gives Pin Hibernate, SM=2 gives Pin Doze, and SM=4 or SM=5 give Cyclic Sleep. If SM=5, the module will wake up after the timer expires or on a pin transition.

Pin Hibernate is the sleep mode with the lowest power consumption. When it is enabled, sleep is controlled solely by the state of the SLEEP\_RQ pin. The wake-up time is 13.2mS.

Pin Doze has higher power consumption than pin hibernate, but has a faster wake-up time of 2mS. It too is controlled solely by the state of the SLEEP\_RQ pin.

Cyclic Sleep has the same current consumption as pin doze. Sleep is entered after a no-activity timeout given by the **ST parameter**, and the module wakes after a period given by the **SP parameter**. While asleep, the RF module wakes periodically to detect whether RF data is present. The SP parameter takes values in the range 0 to 0x6880, with the time unit being 10mS. The default setting for SP is 0. The maximum value of 0x68b0 gives 268 seconds (4m28s). SP values for the coordinator and the end devices should be equal.

For end devices that are configured for association but not currently associated, the **DP command** controls the sleep time, with the same unit and range as for SP but with a default of 0x3e8 (10 seconds).

When a module awakes from sleep, it will sample all active I/O lines and transmit them. If the **IR parameter** is set, it will collect extra samples until the limit set by the **IT parameter** is reached. This automatic sampling on wakeup may be suppressed by setting bit 1 of the **SO parameter** (firmware 10cx).

The 1xAx product manual says that in all sleep modes, the current consumption rises dramatically if the supply voltage is more than 3.0V. In the 1xCx product manual, however, there is no mention of this.

#### 3.3.3 Command State

In the command state, incoming characters are interpreted as commands. There are two sub-states here: AT command state and API command state.

To enter the AT command state, send the three-character sequence “+++” with a guard interval before and after it. The guard interval is determined by the **GT parameter**: if it is set to 0x3e8 (the default), the guard interval is one second. The time unit is milliseconds, and the value range is 2 to 0x0ce4. The three + characters must also be sent within a time equal to the guard interval. The **CC parameter** specifies the ASCII code of the character to be recognized where ‘+’ is written above; ‘+’ is the default.

When in AT command state, commands take the form of AT followed by the command name (two letters, or a letter and a digit), followed by an optional space and then an optional hex parameter value. Each command is terminated by a carriage return character or a comma. If the hex parameter value is omitted, the current value will be displayed. After successful execution of a command, the module will display “OK”, or the result if the command was to display a parameter value. If an error occurred, it will instead display “ERROR”.

The **AC command** will explicitly apply changes to queued parameter values and then re-initialize the module. After this command is given, the module will be operating according to the parameter values.

Stored values will not be retained after a reset, unless the **WR command** is issued. The WR command copies parameter values to non-volatile memory. It takes a little time: don’t send any other command until the “OK” has been received.

To restore all parameters to their factory default values, use the **RE command**. This command does not copy the restored values to non-volatile memory, so use the WR command afterwards if that’s required.

To perform a software reset, use the **FR command**. It will respond immediately with OK, and then perform the reset about 100mS later. The FR command was introduced in firmware version 1x80, and is equivalent to powering off and then on again.

To exit from AT command state, either send the **CN command** or wait for a period controlled by the **CT parameter**. The CT parameter is a timeout value in the range 2 to 0x1770 (10 minutes), and the time unit is 100mS. The default value is 0x64 (10 seconds). Note: before firmware version 10e4 the maximum setting was 0xffff and the 10e6 product manual still has that figure. The change is noted in the firmware revision history.

## 3.4 Flow Control

Flow control is necessary on any serial line if the rate of transmission is such that the receiving buffer cannot always be guaranteed to be emptied at a rate which prevents it from overflowing. There are two flow control mechanisms: hardware flow control and software flow control.

- Hardware flow control is used between an XBee and its host.
- Software flow control is used between two hosts that are communicating via XBees. The XBees themselves do not take any notice of the software flow control characters.<sup>1</sup>

Whether you need to worry about flow control at all is entirely dependent on your application.

### 3.4.1 Hardware Flow Control

The hardware flow control mechanism (between host and XBee) uses the RTS and CTS serial lines. The RTS line is used by the host, to signal to the XBee that the host’s buffer is nearly full and the XBee should stop transmitting until the line is de-asserted. In the same way, the XBee uses the CTS line to pause transmissions from the host.

When either flow control line is activated, any character already being transmitted will still be sent, and in general more than one character may be received after the line activation. Software needs to be written in such a way as to allow for this. It’s the reason why the lines need to be activated *before* the respective buffers are full, instead of when they *are* full.

### 3.4.2 Software Flow Control

With software flow control (host to host), two ASCII characters are given special meanings. The character 0x13 (Control-S or DC3 or XOFF) when sent across the serial link tells the other side to stop sending. The character 0x11 (Control-Q or DC1 or XON) tells the other side to resume sending. When software flow control is in use, it follows that these characters cannot be sent as part of the data stream.

As with hardware flow control, the programmer cannot assume that characters will stop being received as soon as the XOFF character is sent.

---

<sup>1</sup>Special thanks to Digi support for digging me out when I got confused over this.

### Software Flow Control with API Mode

When sending API packets it isn't possible to avoid the need to include the software control characters. For instance, checksums and addresses can contain any values. Therefore the API escape mode (AP=2) exists. In this mode the character 0x7d is defined as the escape character. When a host wants to transmit 0x11 or 0x13 it transmits the escape character first, and then the desired character XORed with 0x20.

That's fine so far, but now the escape character has a special meaning so it too must be escaped in the same way when it appears in the data.

And as a final refinement, in API mode 2 the 0x7e character is also escaped whenever it appears within a packet. It is *not* escaped when it appears as the first byte of the packet. This allows for recovery if any buffers do overflow in spite of the flow control. When overflow does happen, some bytes will be lost and a receiver will usually see just part of a packet. It can then ignore all bytes received until it sees the next 0x7e which marks the start of the next packet. One or more packets will have been lost, but confusion is avoided.

### Software Flow Control with Transparent Mode

When two XBees are being used in transparent mode for host-to-host communication, software flow control is a matter for the hosts to arrange. If binary data is being sent (actually any data in which the flow control characters may appear), then each host must deal with the escaping of those data characters. Since the XBees themselves attach no special meaning to the characters sent, any escape mechanism could be used. It's probably easiest to use the same escape mechanism as for API mode unless there's a convincing reason to do otherwise.



# Chapter 4

## Networking

### 4.1 Network Types

The XBee modules support two of the IEEE 802.15.4 network types: NonBeacon, and NonBeacon w/Coordinator.

### 4.2 NonBeacon

By default, the NonBeacon mode is enabled. This is a peer-to-peer network. It is established by configuring each module to operate as an End Device (**CE parameter** set to 0), disabling End Device Association (**A1 parameter** set to 0), and setting the **ID parameter** and **CH parameter** to be identical for all nodes in the network.

### 4.3 NonBeacon w/Coordinator

For NonBeacon w/Coordinator mode, one module is set to act as the coordinator by setting its **CE parameter** to 1. Its power-up is controlled by the **A2 parameter**. This network type uses Association to determine which end devices are controlled by which coordinator.

### 4.4 Transmission Modes

There are two transmission modes: unicast and broadcast. Unicast is the default. It supports retries; broadcast doesn't.

In unicast mode, a module receiving a packet will send an ACK back to the transmitter. If the transmitter does not receive an ACK within 200mS, it will retry up to three times. These three retries are provided automatically, but the **RR command** will set or read a multiplier: the number of times the (up to) 3 retries will be repeated if necessary. The range of the RR parameter is 0 to 6, and the default is 0. The parameter was introduced in firmware version 1xa0.

The **EA parameter** gives the number of ACK failures since the counter was last reset. The counter is incremented when the module reaches its retransmission limit without an ACK having been received. When the counter reaches 0xffff it retains that value instead of overflowing. To reset the counter, set the EA parameter to 0.

In broadcast mode, all RF modules within range will accept the transmission. No ACKs are sent, and there is no automatic retransmission of packets. A packet is sent in broadcast mode if its destination address is DL=0xffff and DH=0.

### 4.5 Collision Avoidance

The modules use the CSMA-CA algorithm for collision avoidance. The **RN parameter** controls the back-off exponent in the algorithm, and has a range of 0 to 3. The 0 setting disables collision avoidance in the first iteration of the algorithm, and is the default.

The **CA parameter** sets the detection level for collision avoidance. The parameter range is 0 to 0x50, with a default of 0x2c. The unit is -dBm. If the energy on the channel is above this level, packet transmission will be delayed.

The **EC parameter** gives the number of times the module has avoided transmission because of collision detection since the counter was last reset. The range is 0 to 0xffff. When the counter reaches 0xffff it retains that value instead of overflowing. To reset the counter, set the EC parameter to 0.

## 4.6 MAC Mode

The **MM command**, introduced in firmware version 1x80, sets or reads the MAC Mode value, which is in the range 0 to 3. A value of 0 is the default, and causes the addition of a Digi header to the 802.15.4 packets. A value of 3 also adds the Digi header, but ACKs are suppressed. In either Digi mode, duplicate packets are detected and the **ND command**, **DN command** and **RR command** are enabled. The other two modes are strict 802.15.4 modes. Mode 3 appears to have been introduced in the 1xCx firmware.

## 4.7 Network Addressing

Every RF packet contains a source address and a destination address in its header. Either address may be short (16 bits) or long (64 bits). The long address is unique to each module, and is assigned when the module is manufactured. It can be read with the **SL parameter** (Serial number Low) and **SH parameter** (Serial number High) commands. A module will use its long source address in packets if its **MY parameter** is set to 0xffff or 0xffff. Otherwise, the MY parameter value is the short address that the module will use. The MY parameter defaults to 0. The 1xCx product manual says that setting MY to 0xffff disables the reception of packets with short addresses.

When sending data to a module, the **DL parameter** and **DH parameter** determine the destination address. To use a short address, set DL to the other module's MY value and set DH=0. To use long addressing, set DL and DH to the other module's SL and SH. These are 32-bit values. DH and DL default to 0. The module will use DH and DL when it sends line state packets, which are generated by the module itself. For all other packet types, addressing is either explicit in the packet, or implicit as a consequence of the packet type.

For setting the destination address, the **DN command** was introduced in firmware version 1x80. It takes a Node Identifier string as parameter (see Section 4.8), and sets DL and DH to the address of the node with that string. After the DN command, the module then exits AT command mode. If the parameter is not specified or if there is no response within 200ms from the destination module, the command displays "ERROR" as its result. The DN command cannot be used in MAC mode.

The **CH command** sets or reads the frequency channel number used for transmission and reception. For XBee use channels in the range 0x0b to 0x1a. For XBee Pro, use 0x0c to 0x17. The default is 0x0c. For a value  $c$ , the channel frequency is  $2.405\text{GHz} + ((c - 11) \times 5\text{MHz})$ . (The product manual, if I read it correctly, misses out the GHz in the equation.)

The **ID command** sets or reads the PAN (Personal Area Network) id. The default is 0x3332. Any 16-bit address may be used. To broadcast a message to all PANs, set ID to 0xffff.

## 4.8 Network Identification and Scanning

From version 1x80 of the firmware, each module can store an ASCII identifier string of up to 20 printable characters. The string must not start with a space. The **NI command** sets or reads the string, and is ended when a carriage return is seen, or when the 20th character has been entered. This string is known as the Node Identifier.

For the discovery of nodes in a network, the **ND command** was introduced in firmware version 1x80. The command reports, for each node, the MY, SH, SL, DB and NI values in that order. In AT command mode each value is followed by a carriage return, and an extra carriage return follows each node. A final carriage return indicates the end of the response. The ND command can also accept a Node Identifier as its parameter, in which case only that node will be reported. When reporting, a node that doesn't have an NI set will have a single space for the NI value. The ND command cannot be used in MAC mode. See the product manual for more details, including the details of the response when ND is sent as an API command. Note that ND cannot be sent via API as a remote AT command: it can only be sent from a host to its locally connected XBee.<sup>1</sup>

---

<sup>1</sup>This was posted as a forum response. I see though that page 61 of the 1xEx product manual mentions the ND command in the description of the remote AT response API packet, which would imply that maybe remote ND is now possible. But against that, the same comment appears in the 1xCx manual and in 1xCx remote ND definitely didn't work.

When performing node discovery, the **NT parameter** (introduced in firmware version 1xa0) sets or reads how long the node will wait for responses from other nodes. The parameter range is 0x01 to 0xfc and defaults to 0x19 (2.5 seconds). The time unit is 100mS.

Also when performing network discovery, the **NO command** controls the network discovery options. The parameter value is either 0 or 1, and defaults to 0. When set to 1 the local XBee will include itself in the list of discovered nodes.

The **ED command** causes an energy scan on all channels, returning the maximal energy on each. Each energy value is followed by a carriage return. Values of detected energy levels are in units of -dBm. Each channel's scan time in mS is  $(2^d * 15.36)$ , where  $d$  is the parameter value given with the ED command and is in the range 0 to 6. The total scan time is the channel scan time multiplied by the number of channels to be scanned.

The **SC command**, introduced in firmware version 1x80, sets or reads a list of channels to scan for all Active and Energy Scans. The list is held as a 16-bit mask. In the mask, bit 0 corresponds to channel 0x0b and bit 15 to channel 0x1a. On the XBee Pro, bits 0, 14 and 15 are not allowed. The default setting for the mask is 0x1ffe.

The **AS command** (Active Scan) sends a beacon request to the network and returns a set of descriptors for the beacons found. In firmware version 10c8, if we send the command as an API packet:

```
7e 00 04 08 01 41 53 62
```

all that comes back (given a 30-second timeout) is this incomplete packet:

```
7e 00 05 88 01 41 53
```

I haven't tried this with later firmware versions. I'd be surprised if it was still an issue.

## 4.9 Network Association

The **CE command**, introduced in firmware version 1x80, sets or reads the Coordinator Enable parameter. The value is 0 for an end device, or 1 for a coordinator. The default is 0.

The **SD command**, introduced in firmware version 1x80, sets or reads the scan duration exponent. The product manual gives the details.

The way in which an end device associates with a controller is set with the **A1 command**. See section 3.3 in the product manual for the details. For controlling the coordinator's association behaviour, see the **A2 command** in the same section. The **AI command** returns information about the status of the last association request.

The **DP parameter** sets the sleep period for cyclic sleeping remotes that are configured for association but not actually associated. This period dictates when they will wake up and try again to associate. The value range is 1 to 0x6880, and the time unit is 10mS. The default value is 0x3e8 (10 seconds). The maximum value corresponds to 268 seconds.

To force disassociation, use the **DA command**. This will disassociate the module and make it attempt another association.

When an associated end point wakes from sleep, it will use the **FP command** to request any messages the coordinator may be holding for it.

## 4.10 Encryption

The **EE command** enables or disables 128-bit AES encryption, with parameter values 1 or 0 respectively. When encryption is enabled, all modules will use the long form of their source addresses.

The **KY command** sets the encryption key. All modules in the network must use the same key. The key is a 128-bit (16-byte) value. The default key is 0. Once it has been set, there is no way to read it back from the module.

## 4.11 Power Levels

The **PL parameter** controls the power level for transmission. In Europe, the parameter value must be 0 for the regular XBee Pro, though there is also a so-called international version of the Pro available, with reduced power levels. The international version can be identified by a 'J' suffix after the XBP24 part number silk-screened on the chip. Table 4.1 gives the parameter values and corresponding power levels. The default setting is 4.

The **DB command** returns the signal level in dB of the last good packet received (RSSI). The absolute value is reported, so for example a returned value of 0x58 indicates a level of -88dBm. The reported value is accurate in a range between -40dB and the receiver sensitivity. For the XBee, values lie in the range 0x17 to 0x5c. For the XBee Pro, the

Value	XBee	XBee-Pro	XBee-Pro Intl
0	-10 dBm	10 dBm	-3 dBm
1	-6 dBm	12 dBm	-3 dBm
2	-4 dBm	14 dBm	2 dBm
3	-2 dBm	16 dBm	8 dBm
4	0 dBm	18 dBm	10 dBm

Table 4.1: Transmit power levels for the PL parameter.

range is 0x24 to 0x64. If no packets have been received since the last power cycle, reset or sleep event then the returned value will be 0. **Note:** A post by gworle in the XBee-PRO XSC forum suggests that the RSSI reading includes any noise component, so in the presence of noise the figure will be inflated. The suggestion from gworle is that packet percentage is a better measure of reliability.

# Chapter 5

## API Packets

When operating in API mode, all communication between the host and the XBee is by means of these binary packets. The product manuals for firmware versions 1xAx to 1xEx list thirteen packet types. There are nine main types, and four of those have two sub-types. This table, ordered by packet type id, lists them.

Id	Packet	Page
Packets you can send		
0x00	TX request with 64-bit destination address	5-4
0x01	TX request with 16-bit destination address	5-4
0x08	Local AT command, immediate action	5-2
0x09	Local AT command, queued action	5-2
0x17	Remote AT command with 64-bit destination address	5-3
Packets you may receive		
0x80	RX with 64-bit source address	5-5
0x81	RX with 16-bit source address	5-5
0x82	Input line states with 64-bit source address	5-4
0x83	Input line states with 16-bit source address	5-4
0x88	Local AT response	5-2
0x89	TX response	5-5
0x8a	Modem status packet	5-1
0x97	Remote AT response	5-3

### 5.1 All Packets

Every packet has a three-byte header, a payload and a single-byte checksum:

- The first byte of the header is always 0x7e.
- The second byte of the header is the most significant byte of the payload length. It is always zero, because currently there are no packets longer than 255 bytes.
- The third byte of the header is the least significant byte of the payload length.
- The payloads for the different packet types are described below. Every payload begins with a byte which identifies its packet type.
- The checksum is calculated from the payload bytes only. Its value is the sum of the payload bytes, subtracted from 0xff. When checking the checksum on an incoming packet, the sum of the payload bytes plus the checksum byte should be 0xff. All calculations are done modulo 0x100.

### 5.2 Modem Status Packet

1. Byte: packet type id 0x8a

## 2. Byte giving modem status

- 0: Hardware reset.
- 1: Watchdog timer reset.<sup>1</sup>
- 2: Associated.
- 3: Disassociated.
- 4: Synchronization lost (Beacon-enabled only).
- 5: Coordinator realignment, whatever that means.
- 6: Coordinator started.

Modem status packets are sent from the XBee to its host when status conditions change. They cannot be requested by the host. The modem status packet is identical for Series 1 and Series 2.5 XBees.

## 5.3 Local AT Command Packets

These packet types are useful only if the host wants to send commands to its local XBee.

### 5.3.1 AT Command

This comes in two varieties, depending on whether the command is to be queued, or acted on immediately. This packet is sent from the host to the XBee.

1. Byte: packet type id 0x08 for immediate action, or 0x09 for queued action.
2. Byte: frame id. This is an arbitrary value chosen by the host. The value will be returned in the response packet, except that a value of 0 will suppress the response packet.
3. Byte: first character of the AT command code.
4. Byte: second character of the AT command code.
5. Optional bytes: new value for a parameter, most significant first or (for text values) natural order. Numeric parameter values are specified as binary bytes or words.

The Series 2.5 product manual has the same packet format, but for the non-queued version adds that “A string parameter used with the NI (Node Identifier), ND (Node Discover) and DH (Destination Address High) command is terminated with a 0x00 character.” The Series 1 product manual makes no mention of this.

### 5.3.2 AT Response

This packet is returned to the host from the XBee for each AT Command packet.

1. Byte: packet type id 0x88.
2. Byte: frame id as given in the command packet.
3. Byte: first character of the AT command code.
4. Byte: second character of the AT command code.
5. Byte: status.
  - 0: OK
  - 1: ERROR
6. Optional bytes: value of parameter, most significant first. These bytes are present if the command was a command associated with a parameter value and if the command didn’t supply a new value for the parameter.

The Series 2.5 packet has the identical format, but adds two status codes: 2 for invalid command and 3 for illegal parameter.

---

<sup>1</sup>This is the first mention I’ve seen of a watchdog timer.

## 5.4 Remote AT Command Packets

### 5.4.1 Remote AT Command

The ability to send AT commands from a host to an XBee other than its immediately connected one was introduced in firmware version 1xCx.

1. Byte: packet type id 0x17 for 64-bit destination address.
2. Byte: frame id. This is an arbitrary value chosen by the host. The value will be returned in the response packet, except that a value of 0 will suppress the response packet.
3. Bytes: 64-bit destination address. The special value 0x000000000000ffff makes it into a broadcast packet.
4. Bytes: 16-bit destination address. If this address is set to 0xffff, then the 64-bit address will be used. Otherwise, this address is the 16-bit destination address and the 64-bit address will be ignored.
5. Byte: 02 to apply all queued changes immediately, 00 to queue them until the next AC command or the next remote AT command that has 02 in this field.
6. Byte: first character of the AT command code.
7. Byte: second character of the AT command code.
8. Optional bytes: new value for a parameter, most significant first or (for text values) natural order. Numeric parameter values are specified as binary bytes or words.

### 5.4.2 Remote AT Response

1. Byte: packet type id 0x97 for 64-bit source address.
2. Byte: frame id as given in the command packet.
3. Bytes: 64-bit source address (8 bytes).
4. Bytes: 16-bit address. It seems to be the source address of the module sending the response, but I suspect it's more subtle than that. The 1xCx product manual says it's the 16-bit network address of the remote.
5. Byte: first character of the AT command code.
6. Byte: second character of the AT command code.
7. Byte: status.
  - 0: OK
  - 1: ERROR
  - 2: Invalid command
  - 3: Invalid parameter
  - 4: No response.
  - 5: I've seen this, but I don't know what it means. It happened while I was experimenting with packet layouts, so maybe it's a general "that doesn't make sense" indicator.
8. Optional bytes: parameter value if the command was such as to return one.

## 5.5 Input Line States

This packet is used when a remote XBee and a base XBee have been configured so that the remote will sample its inputs at set intervals, and transmit the results to the base. If the base XBee is configured to pass such data out through the UART, the base's host will receive this packet.

1. Byte: packet type id 0x82 for 64-bit source address, or 0x83 for 16-bit source address.
2. Bytes: source address. Two bytes for 16-bit source addressing, or eight for 64-bit source addressing.
3. Byte: RSSI value.
4. Byte: options. If bit 1 is set, this is an address broadcast. If bit 2 is set, it is a PAN broadcast. All other bits are reserved.
5. Byte: sample quantity. This is the number of full sets of samples in what follows.
6. Word: 2-byte channel indicator msb first. Bits 14–9 are a 6-bit mask, with 1 for each ADC channel in AD5–AD0 respectively that will be reported. Bits 8–0 are for the digital lines D8–D0, showing which of them will be included in the values. Bit 15 is not used.
7. Word: optional 16-bit bitfield, with bits corresponding to lines D8–D0 as in the channel indicator. Where bits were set in the channel indicator, the corresponding bits here show the state of the input. These bytes are not present if there are no lines enabled as digital inputs.
8. Words: if any bits in the channel indicator were set among the A-D inputs, those readings now follow. Each is a 16-bit value with the A-D reading in the low-order 10 bits. A-D readings are given in order from AD0 to AD5.

## 5.6 Transmit Request

This is how a host can transmit data over the network to another host. There are three basic packet types involved, though as either 16 or 64-bit addressing may be used, there are two variants of two of them.

These packet types are useful only if there are multiple hosts in the network, and those hosts want to communicate with each other.

### 5.6.1 TX Request

The host initiates the transfer by sending this packet to its XBee.

1. Byte: packet type id. This is 0x00 for 64-bit addressing, or 0x01 for 16-bit.
2. Byte: frame id. This is an arbitrary value chosen by the host. The value will be returned in the response packet, except that a value of 0 will suppress the response packet.
3. Destination address. Either two or eight bytes, depending on the packet type id. For a broadcast packet, set this address to 0xffff. The broadcast address is the same for two or eight-byte addressing.
4. Byte: options. This is the bitwise OR of these flags. All other bits must be zero.
  - 0x01: Disable ACK. This tells the XBee not to return a response packet (see Section 5.6.3).
  - 0x04: Send the packet with broadcast PAN id.
5. Bytes: data. Between 0 and 100 bytes which will be transferred to the destination host.

A comparison between Series 1 and Series 2.5 XBees for this packet type is given in Table 5.1.



Field	Series 1		Series 2.5
type	0x00 (4)	0x01 (4)	0x10 (4)
frame id	(5)	(5)	(5)
64-bit dest	(6-13)		(6-13)
16-bit dest		(6-7)	(14-15)
broadcast radius			(16)
options	(14)	(8)	(17)
data	(15...)	(9...)	(18...)

Table 5.1: Comparison of API packets for TX Request. The numbers in brackets are the byte offsets for the fields.

### 5.6.2 RX

This is the packet that the destination host will receive, assuming the transmission was successful.

1. Byte: packet type id 0x80 for 64-bit addressing, or 0x81 for 16-bit addressing.
2. Source address. Either two or eight bytes, depending on the packet type id.
3. Byte: RSSI value. This is the absolute value of the (negative) dBm figure.
4. Byte: options. If bit 1 is set, this is an address broadcast. If bit 2 is set, it is a PAN broadcast. All other bits are reserved.
5. Bytes: data. These are the data bytes from the original TX Request packet.

### 5.6.3 TX Status

The originating host's XBee returns this packet to its host, after the packet has been successfully or unsuccessfully transmitted. This packet will *not* be returned if the ACK was disabled in the options byte of the TX Request.

1. Byte: packet type id 0x89.
2. Byte: frame id copied from the TX Request packet.
3. Byte: status.
  - 0: Success
  - 1: No ACK received from the destination, and all retries have been attempted. A broadcast transmission will never result in this value.
  - 2: CCA failure.
  - 3: Purged. This occurs when a coordinator times out of an indirect transmission. The timeout is 2.5 times the value of the SP parameter. I take that to mean that the coordinator was holding on to a message for a sleeping remote, and couldn't contact the remote within the timeout so the coordinator discarded the message. But I could well be wrong.



# Appendix A

## Baud Rates and their Accuracy

In serial communications, the baud rate should be about the simplest parameter you can get. But it isn't. The reason is that what you set is not always what you get. You may think you've specified a particular rate, but the actual rate may be different to a degree that affects the quality of communication.

### A.1 System Clock Frequency and UART Clock

In pretty much any digital device, there will be one oscillator (clock) that controls the timing of events. The clock is a signal source, running at a particular high frequency. Any circuitry that needs to use a lower frequency will use a sub-multiple of that frequency. The sub-multiple will be provided by a divider circuit, so any frequency which is an exact integer division of the master frequency is available.

From that, it follows that if you want a frequency that isn't a sub-multiple of the master clock frequency, you're going to have to settle for the nearest approximation. And it's here that the world tends to split into two camps. In one camp we have the PC serial port, which will normally use a crystal clock that does divide exactly to give the advertised baud rates. In the other camp we have microcontrollers, which often use clocks that are a fairly small multiple of 1MHz. These clocks do not divide down to give exact baud rates. The XBee is in the 1MHz-multiple camp: it has a 16MHz crystal clock.

The UART itself needs a clock signal with a frequency of 16 times the baud rate (see Receiver Operation below). It's this frequency that must be derived from the system clock by the divider circuit.

### A.2 Available Baud Rates

As an example, take a system with an 8MHz master clock. If this clock is to drive a UART at 9600 baud, it must be divided down to a frequency of  $16 \times 9600 = 153,600$  Hz. The dividing factor is therefore  $8,000,000/153,600 = 52.0833$ . That isn't an integer value, but the nearest integer is 52. The actual baud rate is then  $8,000,000/16/52 = 9615.38$  baud, which is a ratio of 1.0016 (0.16%) from the exact value. Pretty good.

Now suppose we want 115,200 baud from the same master clock. The UART clock frequency must be  $16 \times 115,200 = 1,843,200$  Hz. So the dividing factor is  $8,000,000/1,843,200 = 4.3402$ . The nearest integer is 4, so the actual baud rate will be  $8,000,000/16/4 = 125,000$  baud. That differs from what was wanted by a factor of 1.085 (8.5%), which is not at all good. This UART will not be able to communicate with one that runs accurately at 115,200 baud.

To fix this, suppose we double the master clock speed to 16MHz. The dividing factor is now  $16,000,000/1,843,200 = 8.68$ , or 9 when we take the nearest integer value. The actual baud rate is therefore  $16,000,000/16/9 = 111,111$  baud. The difference ratio is 1.0368 (3.68%), which is in theory just within acceptable tolerance (see below). The master clock of the XBee is 16MHz, so this scenario represents the situation where an XBee is connected to a PC.

Overall, lower baud rates mean higher factors by which the master clock is divided. In turn, that means there is finer control over the frequency. So if we're going to see problems, they're going to happen at the highest baud rates.

### A.3 Serial Line Receiver Operation

When a serial line is idle, it is in mark state (logic 1). Transmission of a byte begins with the start bit, which is a period of logic 0. Then come the 8 data bits, and finally the stop bit which is at logic 1 level (same as the following idle state).

The receiver must recognise the beginning of the start bit, and then sample the line halfway through each of the 10 bit periods. The receiver clock runs at 16 times the baud rate, and the receiver can sample the input line once per period of this clock. The sequence of operations is:

1. The start bit begins.
2. At its next clock cycle, the receiver detects that the start bit has begun. This may be up to  $1/16$  of a bit period after the actual start.
3. After another 8 cycles, the receiver samples the line again. If the line is still at logic 0, the start bit is confirmed. Otherwise the initial transition is dismissed as noise.
4. After another 16 cycles the receiver samples the line. This is repeated a further 7 times, to get the values of the eight data bits.
5. After another 16 cycles the receiver samples the line again, expecting to see the logic 1 level of the stop bit. If it doesn't see a logic 1 at that point, it discards the data and reports a framing error.
6. After one more cycle the receiver starts sampling the line at every cycle, waiting for the next start bit.

From that, it follows that a receiver can accept transmitted bytes if they arrive no faster than at periods of  $1 + 8 + 8 \times 16 + 16 + 1 = 154$  of its clock cycles. If the transmitter and receiver clocks are at the same frequency, the receiver will see a byte every 160 clock cycles. Therefore the transmitter can be faster than the receiver by a factor up to  $160/154 = 1.039$  (3.9%), and in theory all will still be well.

It also follows that if the transmitter is slower than the receiver, all will be well if the period between the beginning of the start bit and the beginning of the stop bit (144 clock cycles for the transmitter) corresponds to more than 153 of the receiver's clock cycles. So the transmitter baud rate must be greater than  $144/153 = 0.94$  of the receiver baud rate.

Any bidirectional serial channel, if there is a baud rate mismatch, will have the transmitter running faster in one direction and the receiver running faster in the other. The tighter condition for the speed difference is for the direction in which the transmitter is running faster, so the ratio of 1.039 is the limiting case for the connection as a whole.

### A.4 Signal Quality

We've seen that connecting an XBee to a PC at a nominal rate of 115,200 baud with one stop bit results in a connection which is close to the theoretical limit. In theory it should still work perfectly well, but in practice there are reports of problems with this setup. It seems to me that the likely reason is signal degradation, probably arising from the connecting cable. This cable will have capacitive and inductive qualities, so the signal will be slightly distorted when it emerges at the other end. And since the connection is operating so close to the limit, any distortion or other noise is likely to be fatal to the communication quality.

### A.5 Solution at 115,200 baud

Theory says that if you connect an XBee to a PC at 115,200 baud you should be able to get away with one stop bit. In practice, any element of noise or other signal degradation is likely to lead to data loss. The baud rates are already set to the best obtainable values, so to prevent the problem the only remaining option is to configure the PC to send two stop bits, thus slowing its transmission rate to one that the XBee can reliably accept.

Since at 115,200 baud nominal the XBee is running more slowly than the PC, there is no need to slow it further. So there is no need for the XBee to be configured with two stop bits. However, if non-standard baud rates are used, there will be some cases in which the XBee will be the faster transmitter. In such cases, it can be configured to send two stop bits with the command `ATNB=3`. The description of that command is to set mark parity, which is equivalent to an extra stop bit.

If instead of a PC the XBee is connected to a microcontroller, and if the microcontroller's clock is a multiple of 1MHz, then the whole problem goes away. It doesn't matter that neither end can support standard baud rates exactly, because they can still both be set to the identical non-standard rate.

Model	System Clock Hz	% Error at Baud Rate			Note
		57,600	115,200	230,400	
XTend	60M	0.16	1.73	1.73	
900/868	48M	0.16	0.16	0.16	
Series 2	24M	0.16	0.16	0.16	1
XSC	20M	3.34	-1.36		2
Series 1	16M	2.12	-3.55	8.51	3
XStream	8M	3.34	-1.36		4
Notes:					
1. Zigbee or ZNet 2.5.					
2. Special case in code for 115,200 (16 clocks per cell).					
3. 802.15.4 or Digimesh.					
4. Special case in code for 115,200 (8 clocks per cell).					

Table A.1: XBee models and baud rate accuracy.

For microcontrollers with clocks running at frequencies other than those discussed, it's probably a good idea to use the calculations given above to see what baud rate mismatch if any can be expected. If a mismatch is found which is a significant fraction of the limit, then an extra stop bit on the faster transmitter may be a wise precaution.

## A.6 Discussion of 57,600 Baud

At 57,600 baud there is still a mismatch between an XBee and its PC host, though to a lesser extent than at 115,200 baud. The nearest the XBee can get is 58,824 baud which means it is sending at a rate 2.12% faster than the host expects. To judge by postings in the forums, this mismatch generally doesn't cause problems. However, if problems do occur then the solution is again to slow down the faster transmitter. At 57,600 baud nominal the faster transmitter is now the XBee so it must be configured to send two stop bits with the `ATNB=3` command.

## A.7 Baud Rate Accuracy on Other XBees

Digi have very kindly supplied me with information on how other XBees perform at higher baud rates, and I reproduce their information here, in Table A.1.

## A.8 Historical Note

If you google for this sort of stuff, you'll find statements to the effect that serial line communications can tolerate as much as 10% in the mismatch of baud rates - a figure which appears to contradict the 3.9% figure derived above. The 10% figure in fact goes back to the days of mechanical teletypes, which used the stop bit purely as a delay mechanism between transmitted characters. Teletypes did not check the validity of the stop bit, but modern UARTs do. Teletypes, by their nature, also took an average of the line state during a bit period. And they didn't have the granularity issue of the 16x clock. These differences, and probably some others that I haven't thought of, contribute to the slightly surprising fact that a modern UART does in fact have a tighter speed match requirement than the old equipment did.



## Appendix B

# A Study of Custom Baud Rates on the Series 1 XBee

### B.1 Introduction

/large If you think this appendix ends rather abruptly, you're right. It's a work in progress, and at some point I may or may not finish it...

### B.2 The Calculation of Percentages

How hard can it be to calculate the percentage difference between two values? Well, it depends on what you want to do with the result.

Suppose we have three values  $a$ ,  $b$  and  $c$ . Suppose also that  $a < b < c$ . Which of  $a$  and  $c$  is closer to  $b$ ? To illustrate with a concrete example, assume that  $a = 95$ ,  $b = 100$  and  $c = 105$ . In pure arithmetic terms each of  $a$  and  $c$  differs from  $b$  by 5, so by that measure they are equally close and we would probably express this by saying that each of  $a$  and  $c$  differed from  $b$  by 5%.

That might be fine if  $a$ ,  $b$  and  $c$  are amounts of money, but it's less good if they are component values in an electronic circuit or (as here) baud rates. What is then of more concern (I would argue) is not the absolute difference but the ratios of the values. Taking the same example and calculating the ratios gives

$$\frac{b}{a} = 1.0526$$

and

$$\frac{c}{b} = 1.05$$

Not a great difference between the two ratios, but it does show that by this measure  $c$  is closer to  $b$  than  $a$  is. If we were trying to achieve a baud rate of 100 and our only choices were 95 or 105, we would probably prefer the latter.

Now a common way of expressing a ratio as a percentage is to use this formula:

$$\text{pct} = 100 \times \left( \frac{x}{y} - 1 \right)$$

Plug in  $b$  for  $x$  and  $c$  for  $y$ , and the answer comes out as  $-4.7619\%$ . Interchange the two values and the answer comes out at  $+5\%$ . Ignoring the difference in the sign, that's two different numeric values for the same ratio.

It would be preferable to modify the calculation of percentages so that numbers in the same ratio always gave the same numeric answer (again, apart from the sign). It turns out to be easy to do. Here's the algorithm for calculating the ratio as a percentage, given two values  $x$  and  $y$ :

1. If  $x > y$ , remember the sign as 1. Otherwise remember the sign as  $-1$  and exchange the values of  $x$  and  $y$ .
2. Calculate the ratio as  $x/y$ . This value will always be greater than or equal to 1.
3. Subtract 1 and then multiply by 100.

Rate	XBee	Error %	Rate	XBee	Error %
1200	1200	0	19200	19230	0.156
2400	2398	-0.083	38400	38461	0.159
4800	4807	0.146	57600	58823	2.123
9600	9615	0.156	115200	111111	3.680
14400	14492	0.639	230400	250000	8.507

Table B.1: Standard baud rates and the XBee's closest approximations.

4. Multiply the ratio by the sign.

Yes, it's more elaborate than the usual calculation. But it has the desirable property that a smaller absolute value indicates a closer ratio, which the usual calculation did not provide.

In what follows, all percentages are calculated by this method.

### B.3 Standard Baud Rates

Table B.1 shows a selection of standard baud rates, together with the closest baud rate that the Series 1 XBee can achieve for each, and with the percentage error (calculated as above).

We know from reports in the forums that communication at 115,200 baud is problematic, yet I've seen no reports of problems at 57,600. So it would appear that there is a threshold somewhere between 2.123% and 3.680% where trouble starts to kick in. That's a fairly wide gap, and it would be nice to have a better idea of where the threshold actually is.

### B.4 How Close do Baud Rates Need to Be?



## Appendix C

# Using X-CTU under Linux

Linux being my operating system of choice, I've always felt some frustration at having to dig out a Windows machine when I want to update XBee firmware. I now know that I was wrong in this – there's actually no need at all to run Windows in order to run X-CTU for firmware upgrades.

Google has shown me that I'm far from being the first to discover this, but I thought I'd publish it because I've seen a few questions about it in the XBee forums and I've not yet seen the solution posted.

The key is the wine program, and finding out how to make it work for X-CTU.

Wine is available for Linux and for Mac, so in principle this explanation should also apply for Mac users. I haven't been able to try it on a Mac though.

The wine home page is <http://www.winehq.org/>.

So if you use Linux or Mac and want to try this, here's the recipe.

Obviously, make sure you have wine installed. Typing

```
which wine
```

as a command is one way to find out. If it isn't installed, your distribution probably has it available as a package.

Now download the latest version of X-CTU. I got version 5.1.4.1 which came as a file called 40002637\_c.exe. If a later version has been released by the time you read this, use the later version instead.

Run the command

```
wine 40002637_c.exe
```

That installs X-CTU. If it asks questions, take the default answer every time.

The first time you run wine it creates a hidden directory called .wine in your home directory. This directory contains some configuration stuff and a subdirectory called drive\_c. The drive\_c directory corresponds to the C: drive (surprise surprise).

You'll find the X-CTU installation in ~/.wine/drive\_c/ProgramFiles/Digi/XCTU and if you navigate to there you can run X-CTU with

```
wine X-CTU.exe
```

Try it by all means, but there's more to do before it will run successfully.

Actually on my distribution (Fedora Core 11) I also got a desktop icon for launching X-CTU. Your mileage may vary.

In the .wine directory there's also a directory called dosdevices. The dosdevices directory contains symbolic links. Here's how mine looked originally:

```
[john@spike dosdevices]$ ls -l
total 0
lrwxrwxrwx. 1 john john 10 2010-02-13 18:30 c: -> ../drive_c
lrwxrwxrwx. 1 john john 1 2010-02-13 18:30 z: -> /
```

Now at this point the recipe forks. I'm using an RS-232 development board, so I needed to put into dosdevices a symbolic link called com1 to the serial port device file (/dev/ttyS0 in my case). To do that from within dosdevices:

```
[john@spike dosdevices]$ ln -s /dev/ttyS0 com1
```

Note: no colon after `com1`.

Next check the permissions on `/dev/ttyS0` (or whatever you're using). I found it was owned by root and in the dialout group, with permissions allowing access only by those users. There are two solutions to this: 1. Make it available to all users (you need to be root for this command):

```
chmod o+rw /dev/ttyS0
```

or 2. Add yourself to the dialout group (also as root):

```
usermod -a -G dialout john
```

and (as yourself) log out and in again for the change to take effect. Use the

```
groups
```

command to confirm that you've joined dialout.

If you're using a USB development board, the procedure will be similar but with a twist, because USB `/dev` files are typically created only when a device is plugged in. There will be a different group to join (perhaps `uucp`), or to change the permissions you have to dig in the `/etc/udev` directory. The device files have names like `/dev/ttyUSB0` so you'll need to set symbolic links accordingly. As you may gather from the slightly vague wording, I haven't actually tried this.

Now you're ready to run X-CTU. Have a development board or equivalent connected at this point, so that there's something for it to see.

Launch X-CTU and the next problem becomes clear: it won't detect any COM ports. Fear not. Click on the User Com Ports tab and enter the port by hand. So if you're using COM1 put a 1 in the Com Port Number box, then click on Add. Now COM1 appears in the Select Com Port box and you can click there to select it. Hit the Test/Query button and if all has gone well you'll get a dialog box with the modem type and its firmware version.

Unfortunately X-CTU won't remember the COM port next time it's run, so you have to do the last step every time. Still, that's not a huge imposition.

And at that point you're ready to go. Click on the Modem Configuration tab and upgrade the firmware on your module. A few caveats:

1. The display of parameter settings didn't look right to me, and I wouldn't like to use X-CTU in this way for viewing or changing parameters. That's ok by me – I've made my own software for that, and in this exercise I was only interested in firmware upgrades.
2. X-CTU didn't automatically discover the modem type and function set so I had to select those by hand from the menus.
3. It also didn't manage to read the parameter values when I asked it to.
4. The menu for selecting the version of the upgrade didn't appear in the proper order. The latest version was somewhere in the middle.
5. While X-CTU was running, wine was generating various "fixme" type messages to the console. That probably explains the other caveats, but hey – the firmware upgrades worked!

Maybe further work would solve the caveats, but for now I'm happy with what I have. If anyone else tries this, please use the forum to let us all know how you get on.

# Appendix D

## Change Log

I try to record changes to this cookbook here, to make things easier for readers when I post new issues. I don't record small changes such as the correction of typos or when I make minor rewordings to make things clearer.

### D.1 Changes

#### D.1.1 Changes in issue 1.4 (published 26 Apr 2011)

- Clarified the role of the IU parameter. My thanks to Pac for posting a pertinent question in the forum thus causing me to check the point.
- Added Table A.1, describing for XBees other than the Series 1 how accurate are their baud rates at the higher rates.
- Made a start on Appendix B. This is a work in progress, and not yet useful. In time I'll either finish or delete it.

#### D.1.2 Changes in issue 1.3 (published 9 Mar 2011)

- Added Section 2.3 on voltage level conversions (page 2-5).
- I learned how to make cross-references work as hyperlinks in PDF files. You can now, for instance, go to the index and click on a page number to go to that page. Depending on the setup of your PDF viewer, you should also see in the sidebar the table of contents instead of the page thumbnails.

#### D.1.3 Changes in Issue 1.2 (published 7 Mar 2010)

- Added an appendix on how to use X-CTU under Linux (page C-1). It's a lightly edited version of a posting I had previously made in the X-CTU forum.
- Added further note on analog outputs: reported not working with version 10CD (page 2-3).
- Added description of the NO (Node Discovery Options) command (page 4-3).
- Corrected the default values for the P0 and P1 commands. The default for P0 is 1, whereas previously the cookbook said the default for both was 0 (page 2-3).
- Added a description of the CK command (page 3-1).
- Added a note on transmission reliability at 57,600 baud (page A-3).
- Added a comment on a curious side effect of trying to read the IO parameter (page 2-3).
- Added a note in the description of the M0 and M1 parameters: under 10E6 their values cannot be read, even though the product manual says they can (page 2-3).
- Clarified the description of the IS command's parameter. The parameter does nothing and should be ignored (page 2-1).

**D.1.4 Changes in Issue 1.1 (published 26 Jan 2010)**

- Added Section 3.4 (page 3-3) to describe hardware and software flow control in more detail.
- Added cautionary note on analog outputs (page 2-3).
- Added this appendix.

**D.1.5 Issue 1.0**

Although it wasn't marked as such, issue 1.0 was the first issue I published on the website and to the XBee forums.

# Index

- A1 command, 4-3
- A1 parameter, 4-1
- A2 command, 4-3
- A2 parameter, 4-1
- AC command, 3-3
- ACK, 4-1
- AI command, 4-3
- analog inputs, 2-3
- analog outputs, 2-3
- AP command, 3-1, 3-2
- API mode, 3-1
- API packets, 5-1
- AS command, 4-3
- Associated indicator, 2-1
- association, 4-3
- AV parameter, 2-3
  
- baud rate, 2-4
- BD parameter, 2-4
- beacon request, 4-3
- broadcast mode, 4-1
  
- CA parameter, 4-1
- CC parameter, 3-2
- CE command, 4-3
- CE parameter, 4-1
- CH command, 4-2
- CH parameter, 4-1
- channel frequency, 4-2
- channel indicator, 2-2, 5-4
- channel number, 4-2
- CK command, 3-1
- CN command, 2-4, 3-3
- collision avoidance, 4-1
- command state, 3-2
- coordinator enable, 4-3
- CT parameter, 3-3
- custom baud rates, B-1
- cyclic sleep, 3-2
  
- $D_n$  parameter, 2-1, 2-3
- DA command, 4-3
- DB command, 4-3
- DD command, 3-1
- DH parameter, 4-2
- digital inputs, 2-3
- digital outputs, 2-3
- disassociation, 4-3
  
- DL parameter, 4-2
- DN command, 4-2
- DP command, 3-2
- DP parameter, 4-3
- duplicate packets, 4-2
  
- EA parameter, 4-1
- EC parameter, 4-2
- ED command, 4-3
- EE command, 4-3
- encryption, 4-3
- energy scan, 4-3
  
- factory default, 3-3
- features, 1-1
- flow control, 3-3
- FP command, 4-3
- FR command, 3-3
  
- GT parameter, 3-2
- guard interval, 3-2
  
- HV command, 3-1
  
- I/O data packet, 2-1
- I/O line passing, 2-4
- I/O lines, 2-1
- I/O pins, 2-1
- IA parameter, 2-2, 2-4
- IC parameter, 2-1, 2-4
- ID command, 4-2
- ID parameter, 4-1
- idle state, 3-2
- internal voltage reference, 2-3
- IO parameter, 2-3
- IR parameter, 2-2, 2-4, 3-2
- IS parameter, 2-1
- IT parameter, 2-2, 2-4, 3-2
- IU parameter, 2-4
  
- KY command, 4-3
  
- Linux, C-1
  
- $M_p$  parameter, 2-3
- MAC mode, 4-2
- MM command, 4-2
- module mode, 3-1
- module state, 3-2

MY parameter, 4-2

NB parameter, 2-5

ND command, 4-2

network addressing, 4-2

network association, 4-3

network identification, 4-2

network scanning, 4-2

network types, 4-1

NI command, 4-2

NO command, 4-3

Node Identifier, 4-2

node identifier, 4-2

non-volatile memory, 3-3

NT parameter, 4-3

*Pp* parameter, 2-3

PAN (Personal Area Network), 4-2

parity, 2-5

pin doze, 3-2

pin hibernate, 3-2

pinout, 2-1

PL parameter, 4-3

power levels, 4-3

PR parameter, 2-1

PT parameter, 2-4

pull-up resistors, 2-1

RE command, 3-3

receive data frame, 3-1

retries, 4-1

RN parameter, 4-1

RO command, 3-1

RP parameter, 2-3

RR command, 4-1, 4-2

RSSI, 4-3

SC command, 4-3

scan duration exponent, 4-3

SD command, 4-3

serial lines, 2-4

SH parameter, 4-2

SL parameter, 4-2

sleep, 3-2

Sleep\_RQ, 3-2

SM parameter, 3-2

SO parameter, 3-2

SP parameter, 3-2

ST parameter, 3-2

$T_n$  parameter, 2-4

transmit data frame, 3-1

transparent mode, 3-1

unicast mode, 4-1

version numbers, 3-1

virtual wires, 2-4

VL command, 3-1

VR command, 3-1

VREF, 2-3

WR command, 3-3

X-CTU, C-1