6: Sleep, Reset & API Mode

In this chapter we will explore using Sleep and Reset with the XBee, as well as digging into API (Application Programming Interface) mode where data is sent as frames between the controller and the transceiver. While allowing greater data and abilities, API mode can be difficult due to the requirements of data framing.

Interfacing 5 V Input to the XBee Module

With the XBee being a 3.3 V device, it is important to limit the voltage input to that voltage or damage may occur to the device. As discussed in Chapter 2, many of the communications and control lines to the XBee are buffered on the XBee 5V/3.3V Adapter and the XBee SIP Adapter. While many of the control lines are buffered, the general I/O lines are not. Whether working with the BASIC Stamp or 5 V devices with the Propeller chip, it may be necessary to reduce the input voltage to around the 3.3 V level. With digital I/O, as long as the voltage is near or lower than 3.3 but high enough to be seen as a digital high, a precise voltage is not important.

The voltage divider in Figure 6-1 may be used to reduce a 5 V digital output to a safe level for an XBee input based on the formula of **Vout = Vin x (R2)/(R1+R2).** While this provides a voltage output of 3.4 V, it is within the tolerance of the XBee. The reason for choosing these values is that the 10 kΩ (R2) and 4.7 kΩ (R1) resistors are fairly popular sizes. To lower the voltage more, the size of R1 may be increased slightly or R2 decreased.
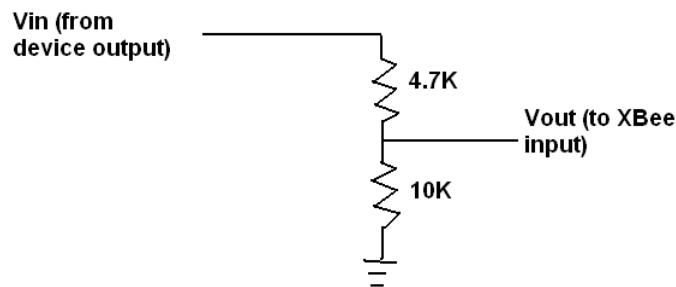

Figure 6-1: Voltage Divider for Digital 5 V to 3.3 V interfacing

The XBee has analog inputs as well which we will explore. With the analog-to-digital converters (ADC) on the XBee, a reference voltage is used to set the upper limit of the analog resolution. This reference voltage should not exceed the supply voltage of the XBee and is typically set to the supply voltage of 3.3 V. Analog devices that have an output which exceeds the 3.3 V should be limited to prevent potential damage to the XBee.

Limiting or dividing analog voltages can be a little tricky, and sensors within the operating limits are preferable. When a suitable sensor is not available, a simple method is to employ the voltage divider in Figure 6-1, but it may lead to issues of loading down the sensor's output—that is based on the output capability of the device and its output impedance (similar to resistance); the voltage measured at the ADC may be lower than the sensor's actual output. Larger resistors can aid in limiting the effect of loading, such as using 47 kΩ and 100 kΩ instead, but a point can be reached where the interface between the voltage divider and the ADC's input can cause loading effects.

A simple test to determine if loading is causing issues is to connect the sensor through the voltage divider, and then measure the sensor's output voltage and the voltage into the XBee's analog input. Using the voltage divider formula, calculate and compare the expected voltage to the actual. With

devices that have variable resistance, such as a potentiometer, this should be checked at low and high values.

Using Sleep Modes

With a current draw of around 50 mA while idle, the XBee is a major draw of current reducing life in a battery-powered system. The XBee has available two major sleep modes to reduce the current consumption, but the XBee cannot receive RF or serial data while sleeping. Planning an appropriate network scheme is important, such as using scheduled events controlled by battery powered remote nodes with a continually powered base.

The major sleep modes consist of using the SLEEP_RQ input pin to control the sleep status and cyclic sleep mode where the XBee automatically cycles between being awake and asleep. Table 6-1 is from the XBee manual providing an overview of the sleep modes (SM command). By default, the XBee SM parameter is 0 for "No Sleep" and the XBee is always awake and ready to receive data. By changing the SM parameter, the sleep mode can be changed.

Table 6-1: XBee Sleep Modes from XBee Manual

Mode Setting	Transition into Sleep Mode	Transition Out of Sleep Mode (Wake)	Characteristics	Related Commands	Power Consumption
Pin Hibernate (SM = 1)	Assert (high) Sleep_RQ (pin 9)	De-assert (low) Sleep_RQ	Pin/Host-controlled / NonBeacon systems only / Lowest Power	(SM)	<10 µA (@3.0 VCC)
Pin Doze (SM = 2)	Assert (High) Sleep_RQ (pin 9)	De-assert (low) Sleep_RQ	Pin/Host-controlled / NonBeacon systems only / Fastest wake-up	(SM)	< 50 µA
Cyclic Sleep (SM = 4)	Automatic transistion to Sleep Mode as defined by the SM (Sleep Mode) and ST (Time before Sleep) Parameters	Transition occurs after the cyclic sleep time interval elapses. The time interval is defined by the SP (Cyclic Sleep Period) parameter.	RF module wakes in pre-determined time intervals to detect if RF data is present / When SM = 5	(SM), SP, ST	< 50 µA when sleeping
Cyclic Sleep (SM = 5)	Automatic transition to Sleep Mode as defined by the SM (Sleep Mode) and ST (Time before Sleep) parameters or on a falling edge transition of the SLEEP_RQ pin	Transition occurs after the cyclic sleep time elapses. The time interval is defined by the SP (Cyclic Sleep Period) parameter.	RF module wakes in pre-determined time intervals to detect if RF data is present. Module also wakes on a falling edge of SLEEP_RQ.	(SM), SP, ST	< 50 µA when sleeping

Typically, pin hibernate mode is used to reduce power consumption. This allows the controller to control the sleep status of the XBee, waking it when needs to communicate and sleeping when not. Once in mode 1 or 2 (SM = 1 or SM = 2), a high on the SLEEP_RQ pin will cause the XBee to enter sleep once all data has been processed by the XBee. This is important in that the XBee will not enter any sleep mode unless its buffers are empty.

> **Sleep modes and RTS**
>
> An XBee will not enter any sleep mode unless the serial buffer is empty. If using RTS this could require extra steps, but in testing it appears that using sleep and RTS are not compatible—once having slept and awoken, RTS appears to be disabled. We don't recommend using sleep while requiring RTS for flow control at this time.

On the BASIC Stamp, use Command Mode to enable a pin sleep mode (SM = 1 or SM = 2):

```
SEROUT Tx, Baud, ["ATSM 1",CR]
```

Similarly performed with the Propeller chip, here is a Spin example using Command Mode to place the module in a pin sleep mode:

```
XB.AT_Config("ATSM 1")
```

Use an output to bring the pin SLEEP_RQ pin high for placing the XBee into sleep mode, and set the pin low to wake the XBee module. Allow a short pause or delay before sending any data of around 10 ms. It is a good idea to bring this pin LOW early in your code. Upon controller reset, the XBee module will still be in sleep mode and may miss critical configuration changes. Placing the pin low will ensure the module is awoken early in your code.

> **(i)** **Adapter Pin Labeling & Signal Conditioning**
>
> The SLEEP_RQ pin is multifunction as DTR/SLEEP_RQ/DI8. On the XBee USB Adapter, XBee Adapter and XBee 5V/3.3V Adapter it is labeled as DTR. On the XBee SIP Adapter, it is labeled as SLP on the 5-pin female header. For the BASIC Stamp, it is conditioned for 5 volts on the XBee SIP Adapter and the XBee 5V/3.3V Adapter Rev B.

Using Reset

Since the XBee module does not reset when the controller does, the XBee may not be in the correct state to receive configuration changes in your code. For example, you may perform configuration settings before changing baud rate or switching to API mode. When your controller resets manually (not due to power cycling) or from a PC due to a program download, the XBee would still be in the last configuration. By using the reset pin, you can perform a reset of the XBee so that it is in the default configuration or the last saved change to the configuration.

The XBee resets with a LOW on the reset pin. By bringing this pin LOW momentarily at the start of your code ensures the XBee is in configuration expected. Another option is to tie this pin to the reset pin of your controller so that both reset at the same time.

> **(i)** **Adapter Signal Conditioning**
>
> The XBee SIP Adapter and XBee 5V/3.3V Adapter Rev B use a diode to the reset pin, cathode to controller I/O, to allow this pin to be brought low while not preventing the XBee from bringing the pin low itself..

API Programming and Uses

In API mode (Application Programming Interface), the data between the XBee and the controller is framed with essential information to allow communications, configuration changes, and special XBee functions. Once in API mode data is no longer simply sent and received transparently, but must be framed for reception by the XBee and parsed from a frame sent from the XBee. Added to the message, or data sent, area header fields containing information needed for delivery and use of the data. The controller is encompassed, or packaged, with these headers. The entire package is a single frame of data. On transmission the headers are added to the data sent to form the frame. On reception, the headers are read and used by the controller.

Benefits and features of using API mode include:

- Ensuring data sent from the controller to the XBee is free of errors.

- Allowing controller code to verify data from XBee to controller is error free (error checking not currently implemented in XBee_Object.spin).

- Building a frame for transmission with necessary information such as destination node address.

- Ability to configure the XBee without flipping into Command Mode.

- Reception of data from a remote node in a frame containing source node's address and RSSI level of received data.

- Acknowledgement frames to ensure remote nodes received data and configuration information.

- Special XBee functions such as receiving analog and digital data from a remote node, remotely configuring a remote node, and line-passing (inputs to one XBee control outputs on another XBee).

The XBee manual illustrates how frames are constructed for transmission to and from the XBee. Each frame type has a unique identifier called the API Identifier. For example, to transmit data to another XBee by its 16-bit address (the DL address we normally use), the API identifier is: 0x01 (hexadecimal) or 1. The frame format is shown in Figure 6-2.
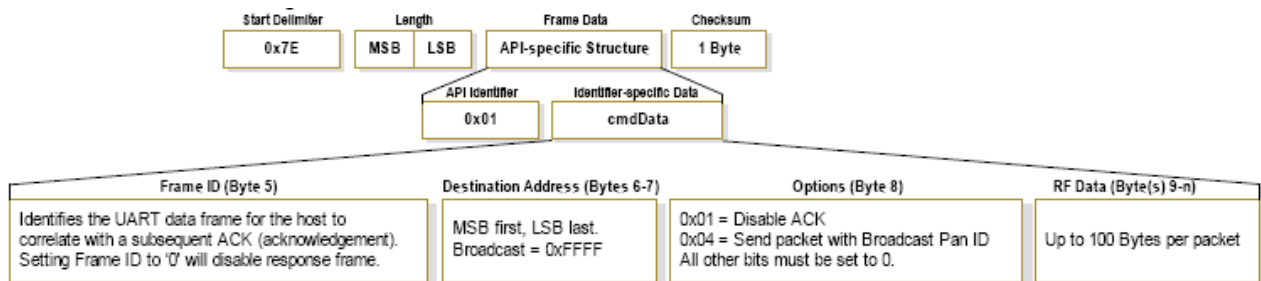

Figure 6-2: API Frame Format for Data Transmission to 16-bit Address

Breaking down the frame:

- **Start Delimiter**: All frames start with 0x7E ($7E) to mark the start of the frame.

- **Length** (2-bytes) is the number of byte from then next byte up to but not including the checksum. It is broken up over 2-byte (MSB and LSB) for lengths over 255 bytes.

- **Frame Data** includes:
  - **API Identifier**: The unique value identifying the function of the frame.
  - **cmdData**: Data to be transmitted made up of:
    - **Frame ID**: Numbers the frame to allow acknowledgements to be correctly identified as belonging to a specific transmission. A value of 0 will return no acknowledgement.
    - **Destination Address** (2-bytes): The 16-bit address to which to send data to. Set to $FFFF for broadcast address.
    - **Options**: Allows special functions and indications on reception such as to not send an acknowledgement.
    - **RF Data** (up to 100 bytes): The actual data to be transmitted.

- **Checksum**: All byte values summed together from after the length byte up to the checksum byte and subtracted from $FF.

Let's say we wanted to transmit the decimal value of 100 as ASCII values, that is a string of "1","0"and "0", followed by a carriage return to be sent to a node at address 2. The bytes for the frame would be as shown in Table 6-2.

Table 6-2: API Frame Values to Transmit Decimal 100

Frame Element	Value	Function
Start Delimiter	$7E	
Length MSB	$00	
Length LSB	$09	Number of bytes from API Identifier up to Checksum
API Identifier	$01	
Frame ID:	$01	
Dest. Addr. MSB	$00	Destination node address ($0002)
Dest. Addr. LSB	$02	
Options	$00	Send an acknowledgement
Data	$31	ASCII character "1"
Data	$30	ASCII character "0"
Data	$30	ASCII character "0"
Data	$0D	ASCII Carriage Return (decimal 13)
Checksum	$5D	($FF – ($01+$01+$00+$02+$00+$31+$30+$30+$0D))

If any byte in the frame sent to the XBee is incorrect, it will not be accepted. If correct, the XBee will repackage the data for the 802.15.4 protocol and transmitted to node $2. Figure 6-3 shows manually framing the packet for transmission using the Assemble Packet window of the X-CTU terminal.
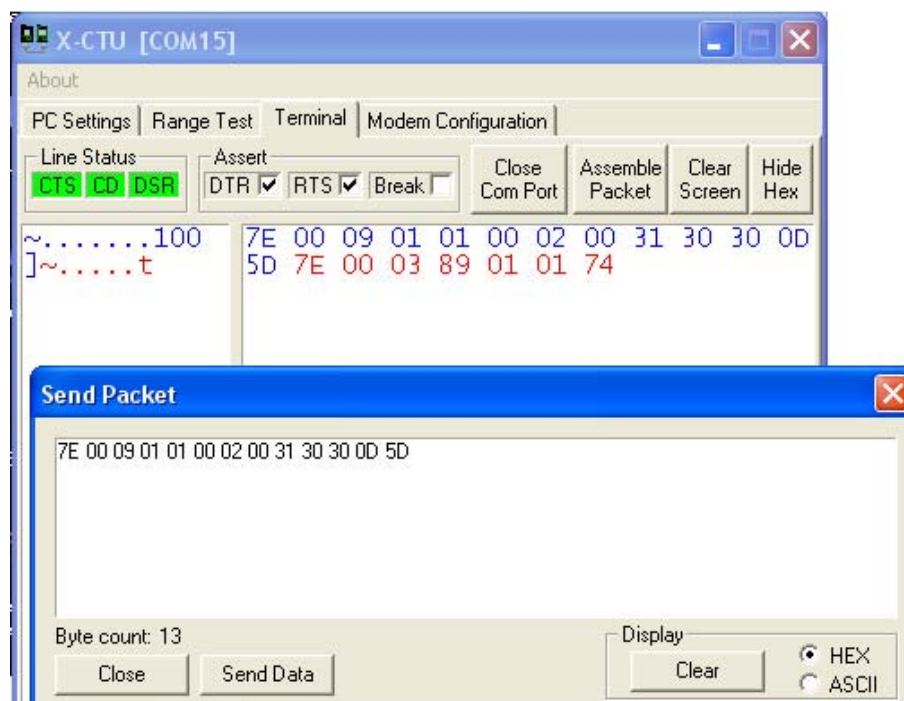

Figure 6-3: Manually Constructing an API Frame

Notice that after the last byte, more data was returned by the XBee. This is a transmit delivery acknowledgement frame (API Identifier $89). The last 01 indicates that delivery failed—we had no node listening at address $2. If you wish to test this yourself, try to send some data manually in API mode:

- ✓ With an XBee connected to USB, place the XBee in API mode (ATAP 1) using the **Modem Configuration** tab or using Command Mode.
- ✓ On the PC Settings window, select **Enable API** to ensure you can change configurations when you need to.
- ✓ In the **Terminal** tab, select **Show Hex** and open the **Assemble Packet** window.
- ✓ Select the **HEX** button.
- ✓ Enter the hexadecimal values and press **Send Data**.
- ✓ You should get an acknowledgement—if you do have another XBee at address 2, it should receive the data of 100.
- ✓ Modify the values to send data to node 0 instead.
- ✓ Try bad values—you should get no acknowledgement frame back because the packet was not accepted and not transmitted.
- ✓ Return the XBee to AT mode by setting **ATAP** to 0 and deselecting **Enable API** on **PC Settings**.

> **AT & API Communications**
>
> Note the XBee modules DO NOT need to be in the same mode to communicate. An XBee at address $2 in AT mode will simply show the text of 100 was received. API mode is for communications between the controller and the XBee, not between two XBee modules. XBee to XBee communications are also framed, using the IEEE-802.15.4 protocol, but we never see it or work directly in it.

When in API mode, data received is similarly framed for reception by the controller to error check (if desired), parse the fields, and pull out data. Figure 6-4 is the API frame for Receive Packet frame showing that the source's address, RSSI level and other fields that may be used by the receiving unit.
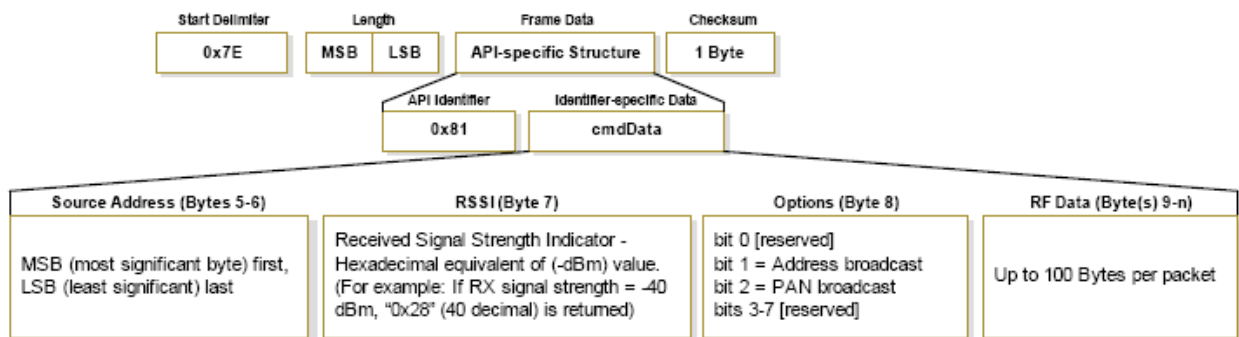

Figure 6-4: API Frame Format for Data Reception by 16-bit Address

Please see the XBee manual for other frame types discussed in this chapter.

Programming for API Mode

For the Propeller, the XBee_Object.spin file provides the ability to send framed data of various types and to accept and parse framed data of various types. The following code demonstrates the method to send a string, such as "100", from the Propeller using API mode. The array of `dataset[]` is constructed of the required bytes and data, the checksum calculated, and the entire frame is transmitted.

```
Pub API_Str (addy16,stringptr)| Length, chars, csum,ptr
{{
  Transmit a string to a unit using API mode – 16 bit addressing
  XB.API_Str(2,string("Hello number 2"))     ' Send data to address 2
  TX response of acknowledgement will be returned if FrameID not 0
  XB.API_RX
  If XB.Status == 0 '0 = Acc, 1 = No Ack

}}
  ptr := 0
  dataSet[ptr++] := $7E
  Length := strsize(stringptr) + 5  ' API Ident + FrameID + API TX cmd +
                                    ' AddrHigh + AddrLow + Options
  dataSet[ptr++] := Length >> 8     ' Length MSB
  dataSet[ptr++] := Length          ' Length LSB
  dataSet[ptr++] := $01             ' API Ident for 16-bit TX
  dataSet[ptr++] := _FrameID        ' Frame ID
  dataSet[ptr++] := addy16 >>8      ' Dest Address MSB
  dataSet[ptr++] := addy16          ' Dest Address LSB
  dataSet[ptr++] := $00             ' Options '$01 = disable ack,
                                    ' $04 = Broadcast PAN ID
  Repeat strsize(stringptr)         ' Add string to packet
     dataSet[ptr++] := byte[stringptr++]
  csum := $FF                       ' Calculate checksum
  Repeat chars from 3 to ptr-1
    csum := csum - dataSet[chars]
  dataSet[ptr] := csum

  Repeat chars from 0 to ptr
    tx(dataSet[chars])              ' Send bytes to XBee
```

On reception of data for the Propeller, the API Identification number of the received data is checked and based on it the fields and data for the received frame are parsed.

```
Pri RxPacketNow | char, ptr, chan
{{
  Process incoming frame based on Identifier
  See individual cases for data returned.
  Check ident with :
  IF XB.rxIdent == value
     and process data accordingly as shown below
}}

    ptr := 0
    Repeat
      Char := rxTime(1)              ' accept remainder of data
      dataSet[ptr++] := Char
    while Char <> -1
    ptr := 0
    _RxFlag := 1
    _RxLen := dataSet[ptr++] << 8 + dataSet[ptr++]
    _RxIdent := dataSet[ptr++]
        case _RxIdent
            $81:   '' ********* Rx data from another unit packet
                   '' Returns:
                   '' XB.scrAddr            16bit addr of sender
```

```
'' XB.RxRSSI              RSSI level of reception
'' XB.RXopt               See XB manual
'' XB.RxData              Pointer to data string
'' XB.RXLen               Length of actual data
_srcAddr16 := dataSet[ptr++] << 8 + dataSet[ptr++]
_RSSI := dataSet[ptr++]
_RXopt := dataSet[ptr++]
bytefill(@_RxData,0,105)
bytemove(@_RxData,@dataSet+ptr,_RxLen-5)
_RxLen := _RxLen - 5
```

The BASIC Stamp doesn't have the large amount of RAM or processing power of the Propeller chip. We will show some sample code that can be used for some API testing, but it requires manually doing much of the work and filling in some values for use. In general though, for basic transmission and reception of data, AT mode where the destination address is set and the RSSI level polled using ATDB, as prior examples have used, is perfectly fine for use.

Propeller XBee Object API Interfacing

In this section we will explore API interfacing using XBee_Object.spin (version 1.6 or higher). The top object, API_Base.spin, will be used to communicate to a local XBee using API mode for a variety of tasks. The initial configuration shown in Figure 6-5 has a Propeller-interfaced XBee as the base, and a USB-connected XBee as the remote assigned to address 2. USB is used to both configure and monitor the remote XBee and to power the unit. The program API_Base.spin will illustrate how the XBee_Object is sending, receiving and using data from the XBee.

Figure 6-5: Propeller API Testing Configuration

✓ Connect the hardware as shown (the I/O devices will be used later).
✓ Set the USB-Adapter connected XBee to a MY of 2 with an open instance of X-CTU.
✓ Have an instance of the X-CTU interface open for monitoring data communications from the Propeller or use the Propeller Serial Terminal.
✓ Using F11, download API_Base.spin to your Propeller, then open the terminal for display.
✓ A menu should be shown in the terminal. If not, reset your Propeller.

Transmitting a Byte

The first task we want to explore is simply sending and receiving characters that could represent individual byte values from the base to the remote for reception. Keep in mind, that when transmitting individual bytes, each byte is sent as an individual packet for reception. Sending a collection of bytes in a single transmission takes a little more work that we will explore later.

- ✓ Choice 1 in the API_Base.spin menu is to **Send Character**. Press **1** to select.
- ✓ Set the number of the destination node for the character —Enter **2**.
- ✓ Type a character to be transmitted.

If all goes well, the results should look similar to Figure 6-6. In the remote terminal window (right), we can see the character displayed. In the base terminal, we have received notification that the transmission was successful. A transmission acknowledgement frame was received that our data was accepted with a status of acknowledged.
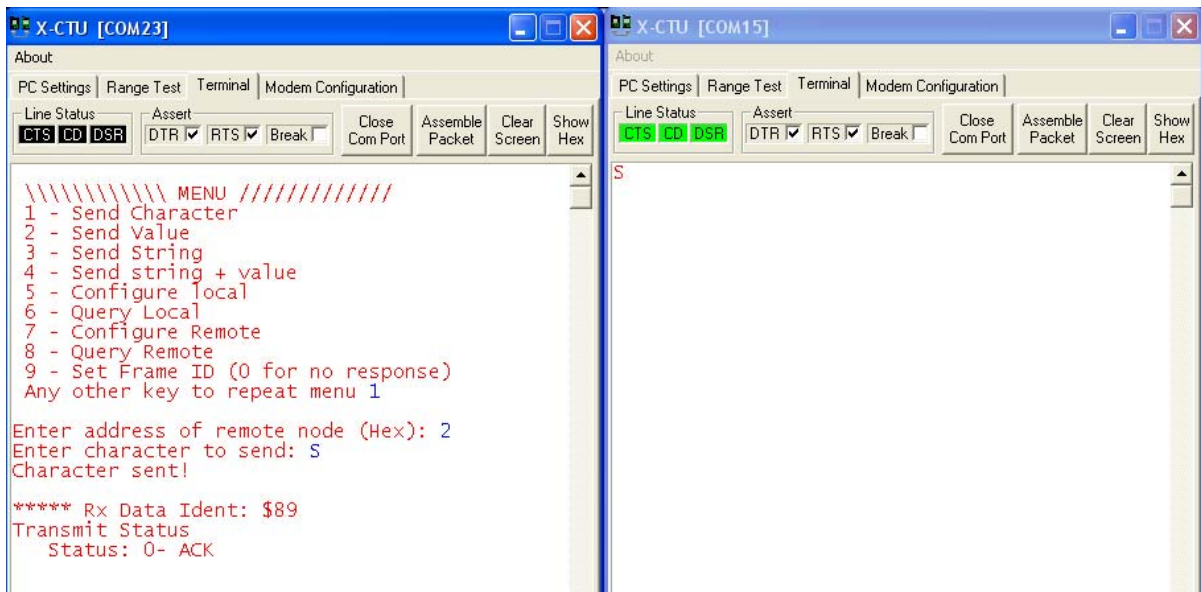

Figure 6-6: Using API_Base.spin to Transmit a character

- ✓ Send another character to a node that is not present and see that you receive notification it was not acknowledged.

The code to accept your menu choice and data for transmission is as follows:

```
Case Choice
    "1":
        PC.str(string(13,13,"Enter address of remote node (Hex): "))
        Addr := PC.RxHex
        PC.str(string("Enter character to send: "))
        Value := PC.Rx
        XB.API_Tx(Addr, Value)
        PC.str(string(13,"Character sent!"))
        XB.Delay(2000)
```

The method call XB.API_Tx(addr,value) sends the address and byte value to the XBee_Object, so it can create a transmit packet properly framed for reception and use by the base XBee, which will then repackage and send the data to the remote XBee.

In a separate cog, the Propeller using the XBee object monitors for incoming data frames. Based on any incoming frames, the API Identification value (`XB.RxIdent`) defines what data type of frame was received.

```
repeat
    XB.API_Rx
    PC.Str(string(13,13,"***** Rx Data Ident: $"))
    PC.Hex(XB.RxIdent,2)
    case XB.RxIdent
```

When the remote XBee acknowledges reception to the base XBee, the base XBee sends a transmit status frame to the Propeller. The XBee object accepts the frame, parses it, and the API_Base code display the information.

```
$89:                         ' Tx Status
    PC.str(string(13,"Transmit Status"))
    PC.str(string(13,"   Status: "))
    PC.Dec(XB.Status)
    Case XB.Status
        0: PC.str(string("- ACK"))
        1: PC.str(string("- NO ACK"))
        2: PC.str(string("- CCA Failure"))
        0: PC.str(string("- Purged"))
```

The status of a transmission can be: an acknowledgement; no acknowledgement; CCA failure when the RF levels are too high to transmit; or it could be purged in certain cases.

- ✓ Use **Set Frame ID** to change the frame to **0** and send another character. Note that using a frame of 0 disables acknowledgement frames from being use which can minimize the data we need to process.
- ✓ Enter a Frame value of a higher value to turn acknowledgements back on.

Transmitting a Decimal Value

In order to send a decimal value such as "1234" it must be converted to a string using the number.spin object. This allows the XBee to accept the string using the **XB.API_Str** method.

- ✓ Use Choice 2 of the menu to **Send Value**.
- ✓ Enter the address of the remote node: **2**.
- ✓ Enter a decimal value and press **Enter**.
- ✓ Verify the value is shown on the remote's terminal and an "Ack" was received.

In the code, the address and address are accepted. The value is converted to a string and `API_Str` is used to send the values for framing and transmission.

```
"2":
        PC.str(string(13,13,"Enter address of remote node (Hex): "))
        Addr := PC.RxHex
        PC.str(string("Enter Value to send (Decimal): "))
        Value := PC.RxDec
        XB.API_str(Addr, num.toStr(Value,num#dec))
        PC.str(string("Value sent!"))
        XB.Delay(2000)
```

Sending String Data

To send a string of text (Choice 3), the code builds an array to transmitted consisting of up to 100 characters, or terminating with a carriage return (**Enter**) and sent by calling the XB.API_Str method as well.

```
"3":
        PC.str(string(13,13,"Enter address of remote node (Hex): "))
        Addr := PC.RxHex
        PC.str(string("Enter string to send: "))
        ptr := 0
        repeat
            strIn[ptr++] := PC.Rx
        while (strIn[ptr-1] <> 13) and (ptr < 100)
        strIn[ptr-1] := 0
        XB.API_str(Addr, @strIn)
        PC.str(string("String Sent!"))
        XB.Delay(2000)
```

Of course, a constant string can be sent using the remote address and string function:

```
XB.API_Str(2, string("Hello World!",13))
```

Transmitting Multiple Data

Choice 4 allows us to combine text and decimal values. You may test by entering the necessary information. Remember, if we want this data to be sent in a single transmission, the individual data must be assembled before the packet is ready to be framed and transmitted. This required adding individual data that will be used to construct an array to eventually be transmitted. The code for Choice 4 demonstrates assembling multiple data for one transmission.

```
"4":
        PC.str(string(13,13,"Enter address of remote node (Hex): "))
        Addr := PC.RxHex
        PC.str(string("Enter string to send: "))
        ptr := 0
        repeat
            strIn[ptr++] := PC.Rx
        while (strIn[ptr-1] <> 13) and (ptr < 100)
        strIn[ptr-1] := 0
        PC.RxFlush
        PC.str(string("Enter Value to Send (Decimal): "))
        Value := PC.RxDec
        XB.API_NewPacket
        XB.API_AddStr(@strIn)
        XB.API_AddStr(num.toStr(Value,num#dec))
        XB.API_AddByte(13)
        XB.API_str(Addr,XB.API_Packet)
        PC.str(string("String & Value Sent!"))
        XB.Delay(2000)
```

After the data is accepted, the .API_NewPacket method is called to create a new packet of data. Using both .API_AddStr and .API_AddByte an array of data is assembled. Once the packet is constructed, the method of .API_Str is called to transmit the data using API_Packet as a pointer to the data.

Note that API_Str expects valid string data ending with a terminating 0 (done by filling the API_Packet with all 0's before use when XB.NewPacket is called). If data containing a 0 is required to be sent, the .API_txPacket method must be used instead:

```
XB.API_TxPacket(addr, pointer, number of bytes in packet)
```

Receiving Data

As data is received by the base XBee, it is placed into a frame for delivery to the Propeller. The API_Base will demonstrate ways of pulling out the incoming data for use as well as use of other features in the receive frame.

- ✓ In the remote's terminal, select **Assemble Packet** to open the Send Packet window, and enter "Hello World!" and send data. The Send Packet window is used to ensure the entire string is sent in one transmission instead of a transmission for each character.
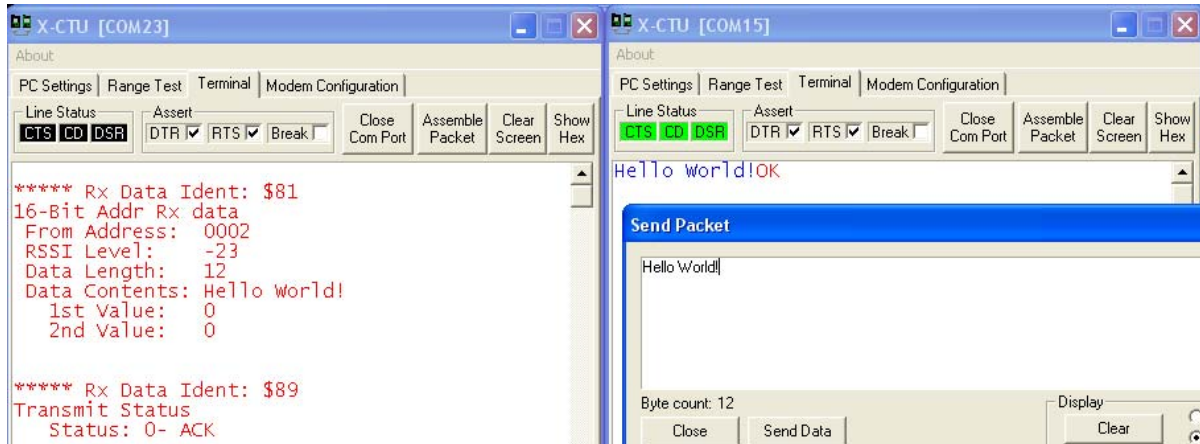- ✓ Your results should look similar to Figure 6-7.

Figure 6-7: Receiving String of Data by API_Base

On receiving a data packet, the frame is broken down by the XBee_Object. API_Base shows pertinent data received, including the API identification, the source address, the RSSI level of the received packet, the length of the actual data, the data itself, and any values sent (shown next example). In this example, the most important piece of data is XB.RxData which points to an array holding the incoming data.

```
$81:                    ' 16-Bit Rx Data
    PC.str(string(13,"16-Bit Addr Rx data"))
    PC.str(string(13," From Address:  "))
    PC.hex(XB.srcAddr,4)
    PC.str(string(13," RSSI Level:    "))
    PC.Dec(-XB.RxRSSI)
    PC.str(string(13," Data Length:   "))
    PC.dec(XB.RxLen)
    PC.str(string(13," Data Contents: "))
    PC.Str(XB.RxData)
    PC.Str(string(13,"   1st Value:   "))
    Value := XB.ParseDec(XB.RxData,1)
    PC.Dec(Value)
    PC.Str(string(13,"   2nd Value:   "))
    Value := XB.ParseDec(Xb.RxData,2)
    PC.Dec(Value)
    XB.API_str(XB.srcAddr,string("OK"))
```

Also note that an "OK" is shown on the remote's node terminal and the base terminal shows the status of transmission of the "OK". The address parsed from the received data is used to reply back to the remote. Now let's send some numeric values as data:

- ✓ Erase the old data and enter 100,200 in the remote's Send Packet window and send data.

✓ On the base node, you should see 100 and 200 for 1st value and 2nd value respectively along with the other data.

✓ The XBee object can take a string, and using commas and carriage returns as delimiters, parse decimal values from strings. For example, this line pulls the 2nd decimal value from a string of text held in `XB.RxData`:

```
XB.ParseDec(XB.RxData,2)
```

Local Configuration

While in API mode, configuration information is also sent for use by the XBee and it will be rejected if not properly framed. While AT mode may still be used, using API mode eliminates the need of popping in and out of Command Mode. Choice 5, **Configure Local**, allows entering the parameter name and value for configuring the local XBee.

✓ Select Choice **5**.

✓ Enter D5 for configuration parameter (association indication output).

✓ Enter 0 for value (disable indicator).

✓ If you have the LED connected to IO4, use D4 and values of 5 and 4 to turn on and off.

If available, the association LED should stop blinking on the base's XBee adapter. The code of `XB.API_Config(str, value)` is used to have the XBee object package a frame for delivery to the local XBee.

```
"5":
        PC.str(string(13,13,"Enter 2-letter Command Code: "))
        strIn[0] := PC.rx
        strIn[1] := PC.rx
        strIn[2] := 0
        PC.str(string(13,"Enter Value (Hex): "))
        Value := PC.RxHex
        XB.API_Config(@StrIn, value)
        PC.str(string("Configuration Sent!"))
        XB.Delay(2000)
```

Notice in the terminal window an acknowledgement is received from the XBee configuration. The API Identification code of $88 shows the frame was received with data of the command sent (D5). The value returned is valid on a query only, not a configuration change. In normal code (non-interactive), the following code may be used to send configuration information:

```
XB.API_Config(string("D5"),0)
```

Local Query of Configuration Value

You may also have the XBee Object query a setting on the local XBee by using choice 6 and entering a parameter to query, such as ID for PAN ID. The API_Base object will send the query and display the returned data parsed by the XBee Object using the API frame identification of $88 once again. Code to send data:

```
"6":
        PC.str(string(13,13,"Enter 2-letter Command Code: "))
        strIn[0] := PC.rx
        strIn[1] := PC.rx
        strIn[2] := 0
        XB.API_Query(@StrIn)
        PC.str(string(13,"Query Sent!"))
        XB.Delay(2000)
```

Returned frame data from XBee Object:

```
$88:                          ' Command Response
    PC.str(string(13,"Local Config/Query Response"))
    PC.str(string(13," Status:         "))
    PC.Dec(XB.Status)
    Case XB.Status
        0:  PC.str(string(" - OK"))
        1:  PC.str(string(" - Error"))
        2:  PC.str(string(" - Invalid Command"))
        3:  PC.Str(string(" - Invalid Parameter"))
    PC.str(string(13," Command Response"))
    PC.str(string(13,"   Command Sent:        "))
    PC.str(XB.RxCmd)
    PC.str(string(13,"   Data (valid on Query):"))
    PC.hex(XB.RxValue,4)
```

Remote XBee Configuration

If you have firmware 10CD or higher installed on the XBee module, you can also configure the parameters on a remote node! This can be used to change normal parameters, or to control output on a remote node – such as D5, the association LED (or any other digital output you desire along with the PWM outputs).

- ✓ Monitor the Association LED on the remote node.
- ✓ Enter Choice **7**.
- ✓ Enter **2** for the remote's address.
- ✓ Enter **D5** for the association LED.
- ✓ Enter **4** to turn it off and verify.
- ✓ Repeat, using a value of 5 to turn it on.

```
"7":
        PC.str(string(13,13,"Enter address of remote node (Hex): "))
        Addr := PC.RxHex
        PC.str(string(13,"Enter 2-letter Command Code: "))
        strIn[0] := PC.rx
        strIn[1] := PC.rx
        PC.str(string(13,"Enter Value (Hex): "))
        Value := PC.RxHex
        XB.API_RemConfig(Addr,@StrIn, value)
        PC.str(string(13,"Remote Configuration Sent!"))
        XB.Delay(2000)
```

Notice that confirmation of configuration information is returned and parsed including the address, status of configuration change and parameter set.

Remote XBee Query

Using choice 8, a remote XBee may be queried for current settings and values. Note that you cannot read the value of a digital or analog input, only its configuration setting. Use choice 8 to enter a remote address (2) and a value to query.

Code to send query:

```
"8":
        PC.str(string(13,13,"Enter address of remote node (Hex): "))
        Addr := PC.RxHex
        PC.str(string(13,"Enter 2-letter Command Code: "))
        strIn[0] := PC.rx
        strIn[1] := PC.rx
        XB.API_RemQuery(Addr, @StrIn)
        PC.str(string(13,"Remote Query Sent!"))
        XB.Delay(2000)
```

Code to show parsed data:

```
$97:
    PC.str(string(13,"Remote Query/Configuration"))
    PC.str(string(13," From Address:     "))
    PC.hex(XB.srcAddr,4)
    PC.str(string(13," Status:          "))
    PC.Dec(XB.Status)
    Case XB.Status
        0:  PC.str(string(" - OK"))
        1:  PC.str(string(" - Error"))
        2:  PC.str(string(" - Invalid Command"))
        3:  PC.Str(string(" - Invalid Parameter"))
        4:  PC.str(string(" - No Response"))
    PC.str(string(13," Command Response"))
    PC.str(string(13,"   Command Sent:          "))
    PC.str(XB.RxCmd)
    PC.str(string(13,"   Data (valid on Query): "))
    PC.hex(XB.RxValue,4)
```

Analog and Digital Data

Analog and digital data may be sent as framed data from a remote XBee to one reading the API frames if you are using firmware version 10A1 or higher. On the remote unit, the destination address (DL) is set, the various I/O's are configured for digital inputs or ADC, and IR is set for the sample rate. The remote unit will send data of the configured I/O at the set interval as a framed packet with bits and data identifying the transmitted data. The number of samples per transmission may be set using the IT setting (samples before transmit). XBee_Object.spin only accepts one sample per transmission at this time.

The XBee needs to be configured with appropriate I/O devices connected. Figure 6-5 shows the XBee USB Adapter being used on a breadboard, and powered for convenience of configuration changes from USB. It is using the photo-diode as an analog input and a pushbutton as digital input. Note that the inputs on the XBee have internal pull-up resistors eliminating the need for pull-up or pull-down resistors in the configuration.

- ✓ Through the X-CTU **Modem Configuration** tab, configure the remote USB XBee for the following:
    - o MY to 1 (remote XBee to address 1)
    - o I/O Setting D0 to ADC (2) (to read analog input)
    - o I/O Setting D1 to ADC (2) (to read analog input)
    - o I/O Setting D3 to DI (3) (to read digital input)
    - o I/O Setting D4 to DI (3) (to read digital input)
    - o IR Setting to $7D0 (2000 decimal to sample once every two second)

✓ Monitor the Propeller-connected Base XBee in X-CTU.

✓ Information similar to Figure 6-8 should be displayed.

> **ⓘ Using Remote Configuration for non-USB connected XBee**
>
> The Base XBee may also be used for configuration using menu choice 7, Remote configuration to address 1. Use WR to store data in non-volatile memory of the remote unit.
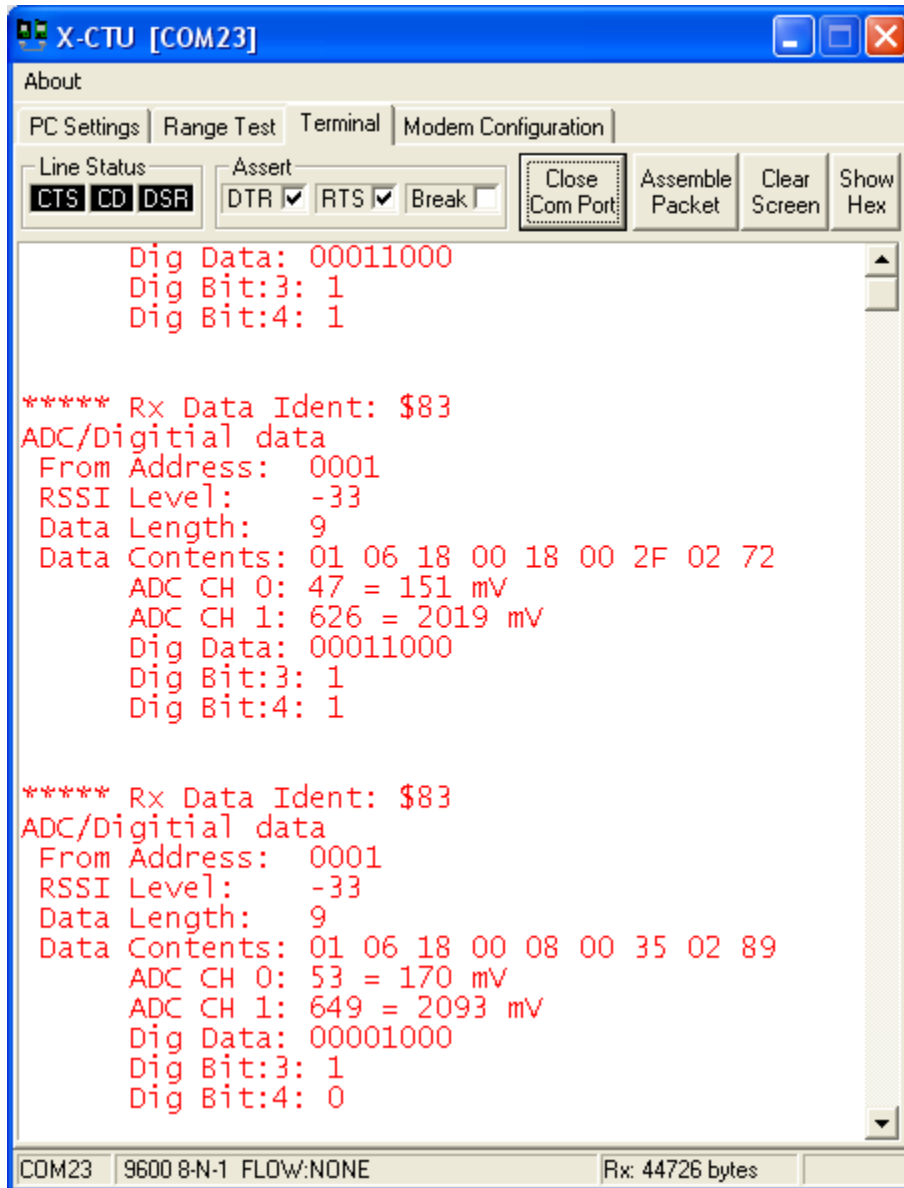

Figure 6-8: Analog and Digital Testing Results

Displayed is the data sent from the remote XBee module. The frame identification indicates it is analog and digital frame type ($83) from address 0001. The RSSI level is displayed as well as the raw data contents in hexadecimal. Any active channel data is shown: the analog inputs on D0 and D1 as both decimal and converted to millivolts based on the 3.3 V reference. The full digital data is shown and the active digital inputs on D3 and D4. Note that per our hardware, we only have an analog input

on D1 and a digital input on D4. The other data is sent simply for testing reading the data. From the XBee manual, the data sent for the frame is constructed as shown as in Figure 6-9.
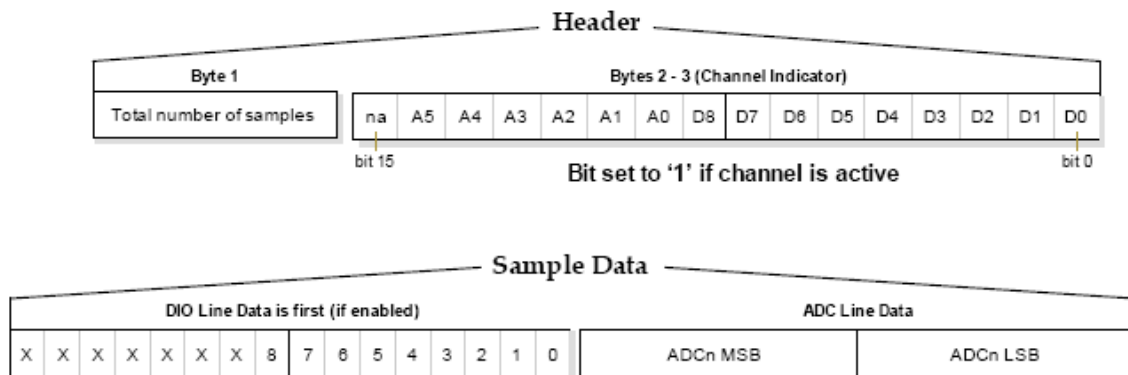

Figure 6-9: ADC and Digital Data Format

The Header defines the number of samples in the transmission and sets any bit high for which analog or digital or data is sent. The Sample Data contains the actual digital data as two bytes, and the MSB and LSB bytes (for 10-bit data, maximum value 1023) of each analog channel sent.
Looking at our actual data contents received for the last transmission:

- 01 = Number of samples: 1

- 06 = Analog channels A1 and A2 are active (Binary 0000_0110)

- 18 = Digital channels D4 and D3 are active (Binary 0001_1000)

- 00 = DIO line data is all 0's for upper byte (Only D8 and rest unused)

- 08 = DIO line data for D7 to D0, only D3 is high (Binary 0000_1000)

- 00 & 35 = Analog channel 0 data (0035 converted to decimal = 53)

- 02 & 89 = Analog channel 1 data (0289 converted to decimal = 649)

The analog channels are converted to a voltage with the following equation to multiply received value by 3300 mV (3.3V = reference) and divide by 1023 (the maximum analog value):

Data value x 3300 / 1023.

The XBee object reads this fairly complex data, analyzes it, and allows the top object to access the data in a manageable format for use. Should any channel read not contain enabled data, a value of -1 is returned.

```
$83:                                    ' ADC / Digital I/O Data
    PC.str(string(13,"ADC/Digitial data"))
    PC.str(string(13," From Address:  "))
    PC.hex(XB.srcAddr,4)
    PC.str(string(13," RSSI Level:    "))
    PC.Dec(-XB.RxRSSI)
    PC.str(string(13," Data Length:   "))
    PC.dec(XB.RxLen)
    PC.str(string(13," Data Contents: "))
    repeat ch from 0 to XB.RxLen - 1     ' Loop through each data byte
      PC.Hex(byte[XB.RxData][ch],2)      ' Show hex value of each data byte
      PC.Tx(" ")
```

```
repeat ch from 0 to 5                       ' loop through ADC Channels
  If XB.rxADC(ch) <> -1                      ' If not -1, data on channel
    PC.str(string(13,"       ADC CH "))
    PC.Dec(ch)                               ' display channel
    PC.str(string(": "))
    PC.Dec(XB.rxADC(Ch))                     ' display channel data
    PC.str(string(" = "))
    PC.Dec(XB.rxADC(Ch) * 3300 / 1023)  ' display in millivolts
    PC.str(string(" mV"))
```

> **?** **Transmit LED?**
>
> The Tx LED on the USB remote will not light when sending the analog/digital data. This LED is actually indicating data sent to the XBee for transmission (DIN), not that it is transmitting.

Line-Passing

In line passing, the analog and digital inputs are passed to another XBee to be directly used to control devices or have the outputs read by controller instead of reading the packet. The same packet is sent from the remote, but the values are used to control the base I/O instead, D0 to D7 on the remote will control D0 to D7 on the base respectively. If analog values are read instead, an analog value on D0 or D1 can control PWM outputs on PWM0 (normally used for RSSI) and PWM1 respectively.

To configure the base for accepting Line Passing, use menu choice **5** for the following setting where specified on the base.

- ✓ Unplug the remote from USB to stop data from being sent.
- ✓ Set IA to 1: Sets the address from which data will be used, for our example, remote node 1 (FFFF may be used to accept data from any device).
- ✓ Set D4 to 4: Sets the direction of the IO to be controlled to be an output low.
- ✓ Set P1 to 2: Sets PWM1 to output PWM.
- ✓ Set IU to 0: Disables the received data from being sent to the Propeller Chip (data is used for I/O control only). This value is normally 1.
- ✓ Connect remote XBee to USB. Every 2 seconds data should be sent updating the hardware on the base. The button should control the state of one LED, and the brightness at the remote should control the brightness of the other LED using PWM.

All I/O controlled can use timeout values. If data is not received in a set amount of time, the output will turn off. This value is in 100 ms increments. T0 to T7 (timeout for D0 to D1 outputs) is by default $FF, 255 or 25.5 Seconds. PT controls the PWM timeout. Setting these to 0 will disable timeouts.

For more rapid updates, set the sample rate, IR, of the remote XBee to a lower value. Also, IC may be set to send data should digital data change. The binary value describes which I/O to use; FF means any, D0–D7. A value of $10 (0001_0000) would send data only if D4 changes. Only digital data is sent for a state change.

- ✓ Set the remote XBee's IR value using menu choice 7 and address 1 to configure IR to 50.

The digital output may be read directly by the Propeller to determine remote states. The PWM may be read directly by measuring the pulse width, but it is very fast, from around 3 to 61 µs (high to low PWM) and read by a cog counter (such as using PulsIn_us or using PULSIN_Clk for direct counter

value from BS2_Functions.Spin), but note that a constant high or low value will cause the cog to lockup waiting for a pin change.

- ✓ `PULSIN_Clk` has been included in your code for menu choice 0. Monitor the clock counts on P7 as you adjust light levels. Note that maximum or minimum values may cause the cog to lock up waiting for pin change.
- ✓ Press any key while reading data to exit.

The PWM output may also be filtered to create an analog voltage using a low-pass filter shown in Figure 6-10.
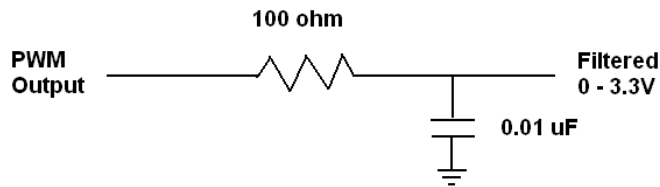

Figure 6-10: PWM Filtering for Analog Voltage

BASIC Stamp and API Mode

While using API mode has clear advantages, the amount of RAM memory (26 bytes) on the BASIC Stamp can make working with frames cumbersome and memory intensive. We'll demonstrate sending and receiving frames, but if you simply need to send and receive data, we don't recommend API mode. Each unique use would require work to make it function for that use. Remember, both sides of a transmission DO NOT need to be in API mode. It is simply a means of packaging data to and from the controller and using higher-level functions.

The code, API_Base.BS2, will be used to send and receive framed data in API mode to highlight some of the nice features of API mode and the XBee module in this mode. The code is very long and memory intensive. Further on we will discuss some tricks to use less memory, but generally these tricks reduce the amount of data received and nearly eliminate the need to use API mode in the first place. The configuration in Figure 6-11 is used to test API_Base.bs2. The base XBee is controlled and monitored via the BASIC Stamp Debug Terminal and the remote XBee on USB for power using X-CTU for a terminal. An array of 18 bytes called **DataSet** is used to hold the incoming and outgoing frame.

Figure 6-11: BASIC Stamp Testing Hardware Configuration

Sending a Decimal Value

- ✓ Configure the hardware as showing in Figure 6-11.
- ✓ Configure the Remote XBee on USB for an address of 2 (MY=2) and monitor the terminal window.
- ✓ Download API_Base.bs2 to the base.
- ✓ Use menu option 1.
- ✓ Enter a remote address of 2 and a value.
- ✓ Note the value should be shown in the remote's terminal window.
- ✓ On the Base, you should see the framed data that was sent, and a reply frame similar to Figure 6-12.

Figure 6-12: API_Base Sending Value with Response
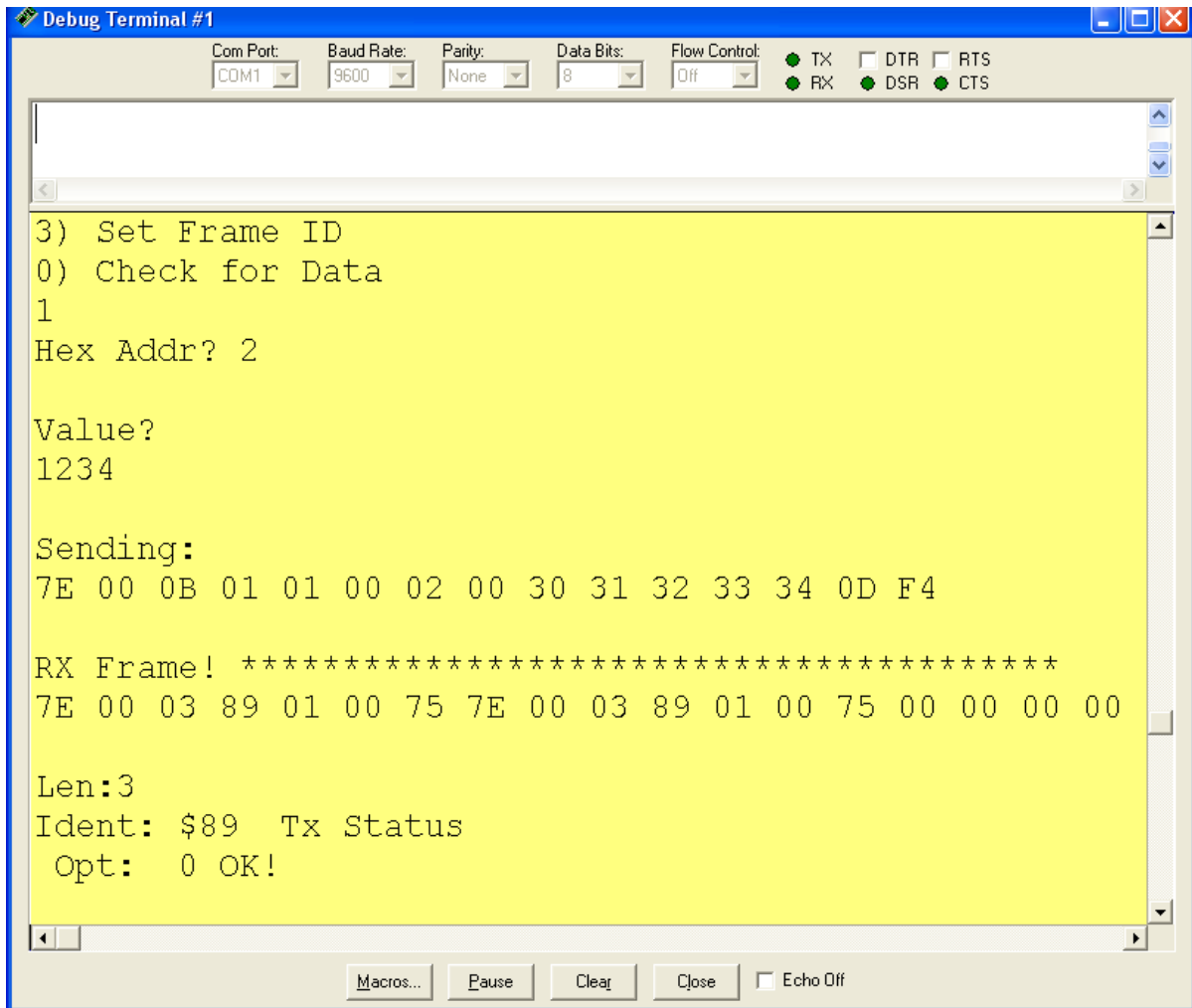
You can compare the data sent to API frame format shown in Figure 6-2 and identify individual bytes of the frame. The Rx Frame data is a response from the XBee on the status of the transmission. The option field identifies the result of the transmission.

✓ Send a value to a node that is NOT present and view the status results.

Looking at the code to send our value, each field is assigned, the value is converted to decimal, the checksum is calculated by subtracting all bytes from $FF , the data is transmitted, and the response frame is checked.

```
        SELECT DataSet(0)
          CASE "1"                      ' Send value to address
              DataSet(0) = $7E          ' Start Delimiter
              DataSet(1) = 0            ' Length MSB
              DataSet(2) = 11           ' Length LSB
              DataSet(3) = $01          ' API Ident
              DataSet(4) = FrameID      ' FrameID
              DEBUG CR,"Hex Addr? "     ' Enter address
              DEBUGIN HEX Value
              DataSet(5) = Value.HIGHBYTE ' Parse address into frame
```

```
              DataSet(6) = Value.LOWBYTE
              DataSet(7) = 0                ' Options - request Ack
              DEBUG CR,"Value? ",CR        ' Enter value to send
              DEBUGIN DEC Value
              FOR Idx = 0 TO 4              ' Save value in frame as ASCII
                DataSet(8+Idx) = Value DIG (4-Idx) + $30
              NEXT
              DataSet(13) = CR             ' End with CR
              DataSet(14) = $FF            ' Checksum start value
              FOR idx = 3 TO 13            ' Calculate checksum
                DataSet(14) = DataSet(14) - DataSet(idx)
              NEXT
              DEBUG CR
              DEBUG "Sending:",CR
              FOR idx = 0 TO 14            ' Show data frame
                DEBUG HEX2 DataSet(Idx), " "
              NEXT
              FOR idx = 0 TO 14            ' Send data frame
                SEROUT Tx,Baud,[DataSet(Idx)]
              NEXT
              GOSUB GetFrame               ' accept returning frame
```

Upon calling **GetFrame**, incoming data is checked for and, based on the API Identifier, pertinent data is displayed (more could be shown for data, but memory limitations of the BS2 required a tightening of the code). Once data is received, the first byte is checked for the value of $7E (Start Delimiter), and then based on the value of API Identifier, it is parsed pulling out appropriate data.

```
        GetFrame:
          GOSUB ClearDataSet                 ' Clear frame
                                             ' accept frame data with RTS
          SERIN Rx\RTS,Baud,50,Timeout,[STR DataSet\18]
          Timeout:
          IF DataSet(0) = $7E THEN           ' Check for start delimiter
            DEBUG CR,CR,"RX Frame! ",REP "*"\40,CR
            FOR IDX = 0 TO 17                ' Show frame
              DEBUG HEX2 DataSet(IDX), " "
            NEXT
            GOSUB ParseData                  ' Break down frame
          ENDIF
        RETURN


        ParseData:
          DEBUG CR,                          ' Display length & API Ident
                "Len:",DEC DataSet(1)<<8 + DataSet(2),CR,
                "Ident: ",IHEX2 DataSet(3)
          SELECT DataSet(3)
```

Based upon an API Identifier value of $89, it is identified as a Transmit Status Frame, and the field values are used as needed:

```
            CASE $89                        ' Transmit status frame
                DEBUG "  Tx Status",CR,
                      " Opt:  ",DEC DataSet(5)
                IF DataSet(5) = 0 THEN DEBUG " OK!"
                IF DataSet(5) = 1 THEN DEBUG " No Ack"
```

```
IF DataSet(5) = 2 THEN DEBUG " CCA Fail"
IF DataSet(5) = 3 THEN DEBUG " Purged"
DEBUG CR
```
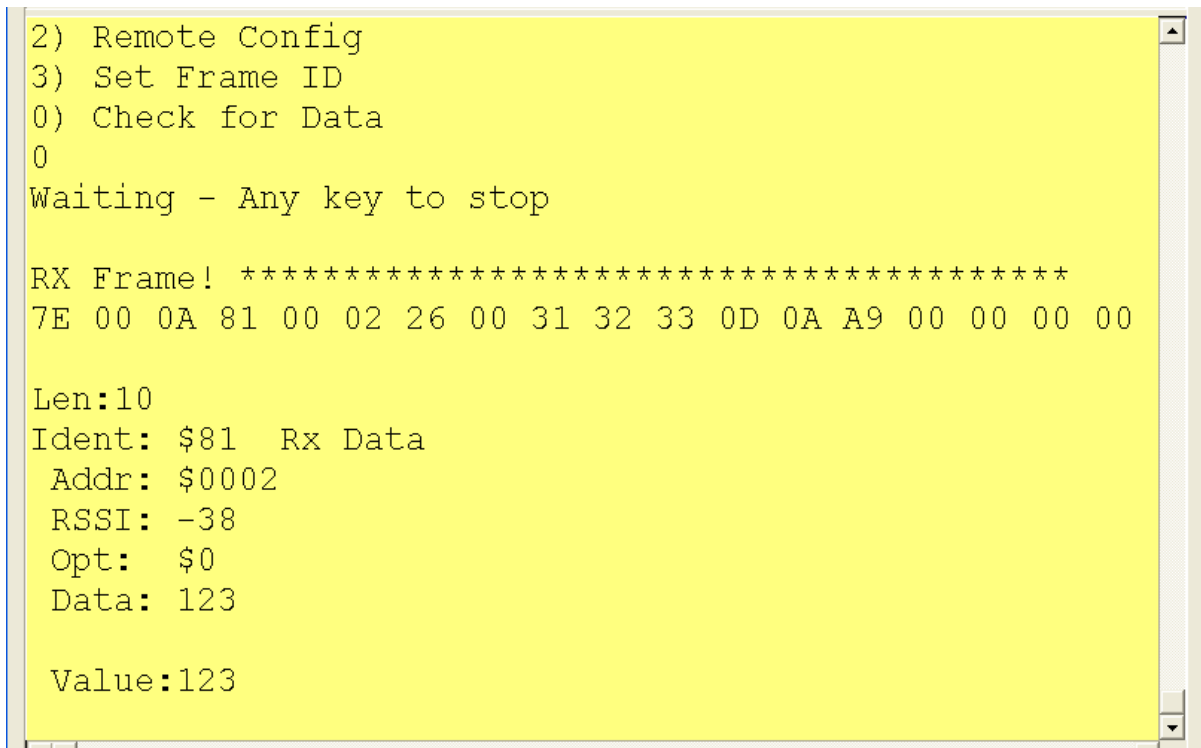
Setting Frame ID

If the frame ID is set to 0, a status frame will not be sent to the controller. This may be useful at times to limit the amount of data coming in. Use menu option 3 to set the frame ID to 0, test sending a value, and returning frame ID to a higher value.

Receiving a Decimal Value

API_Base.bs2 can receive a decimal value as well, and from the data packet pull additional information—this is the true power of using API mode.

- ✓ Select Menu Option 0 to continually monitor for incoming frames.
- ✓ On the remote node, use Assemble Packet to send a value to the base.
- ✓ Monitor the Debug Terminal.
- ✓ Press any key to return to the menu (may take several presses).

A shown in Figure 6-13, a frame with an API Identifier of $81 is accepted and parsed showing the sender's address, the RSSI level of the received data, and the content of the data.

```
2) Remote Config
3) Set Frame ID
0) Check for Data
0
Waiting - Any key to stop

RX Frame! ***********************************
7E 00 0A 81 00 02 26 00 31 32 33 0D 0A A9 00 00 00 00

Len:10
Ident: $81  Rx Data
 Addr: $0002
 RSSI: -38
 Opt:  $0
 Data: 123

 Value:123
```

Figure 6-13: API Base Parsing Received Frame of Data

In the code, the frame bytes are parsed, displayed, and the data bytes are converted to a decimal value for use.

```
    CASE $81                          ' Received data frame
      DEBUG "  Rx Data",CR,
            " Addr: ",IHEX4 DataSet(4)<<8 + DataSet(5),CR,
            " RSSI: -",DEC DataSet(6),CR,
            " Opt:  ",IHEX DataSet(7),CR,
            " Data: "
      Value = 0                       ' show value & convert to decimal
      FOR IDX = 8 TO (8 + DataSet(2)-6)
        DEBUG DataSet(IDX)
        IF (DataSet(Idx) > $29) AND (DataSet(Idx) < $40) THEN
          Value = Value * 10 + (DataSet(Idx)- $30)
        ENDIF
      NEXT
      DEBUG " Value:", DEC Value,CR ' Display data
```

Receiving ADC and Digital Data

One of the great features of API mode is to receive a frame of data from a remote XBee containing digital and ADC data from signals applied to the remote XBee. Inputs D0 to D8 may be configured for ADC input (not all inputs allow ADC data) or for digital inputs. Based upon the sampling rate, a frame of data with the values is sent for use in API mode. Please see the Propeller section for a fuller discussion of this operation.

- ✓ On the remote USB XBee, ensure DL is set to 0 (address of base).
- ✓ In I/O Settings, set:
  - o D0 to ADC
  - o D1 to ADC
  - o D3 to DI (Digital Input)
  - o D4 to DI (Digital Input)
- ✓ Set sampling rate (IR) to 7D0 (2000 decimal, every 2 seconds).
- ✓ On the Base's DEBUG Terminal, monitor frames using option 0.

You should see framed data arrive as shown in Figure 6-14.

```
  DIO Ch3=1
  DIO Ch4=0



RX Frame! ***********************************
7E 00 0E 83 00 02 26 00 01 06 18 00 08 00 5A 00 4D 86

Len:14
Ident: $83  Rx ADC/DIO
 Addr: $0002
 RSSI: -38
 Opt:  $0
 Data:
  ADC Ch0=90
  ADC Ch1=77
  DIO Ch3=1
  DIO Ch4=0
```

Figure 6-14: API Base Receiving Analog and Digital Data

Pressing the button on the remote should change the value of DIO Ch4, and changing the light level to the photo-transistor should change the ADC Ch1 value. Ch0 and Ch3 are not used in our hardware, but shown as additional data. The ADC is 10-bit, having a value of 0 to 1023. Please see the Propeller section for a full discussion on how the data is arranged and what analysis is required.

   ✓ Set IR to 0 on the Remote node to stop transmissions.

On the remote, you may have it send only digital data when there is a state change on the digital inputs by setting IC to FF (monitor all 8 bits, DI0 to DI7). Note that the base code can only accept up to 2 channels of ADC Data.

Remote Configuration

Another great use of API mode is configuring a remote XBee (Firmware version 10CD or higher required). In this mode, the 2-letter command configuration and values are sent to a remote XBee providing the address. This allows one XBee to manually control digital or PWM outputs on another XBee or to change its configuration.

   ✓ Using Base option 2, enter the address for the remote base (2).
   ✓ Control the Association LED on the remote by enter D5 for the command code.
   ✓ Enter 4 to turn off the Association LED.
   ✓ Repeat using 5 to turn the Association LED on.
   ✓ Connect an LED to IO2 and control it using command D2.

Line Passing

In Line Passing, the outputs on one XBee are controlled by the digital and analog inputs on another XBee. Please see the Propeller section on this topic for controlling the LEDs on the base XBee. Base configuration needs to be done in code locally, such as:

```
SEROUT Tx,Baud,["ATAP1,D61,IA1, D44, P12,UI0,CN",CR]
```

- ✓ Set the remote USB XBee to send data every 50 ms (IR to 32).
- ✓ Press the remote's pushbutton and cover the phototransistor while monitoring the LEDs on the base.

Shortcuts for API Mode

There are several good examples on the Parallax Forums for API mode on the BASIC Stamp. These generally demonstrate very specific use while not gobbling up the majority of the free memory. The basic method is to do much of the frame assembly manually, parse in the needed data and calculate the final checksum value. If, for example, you wanted to send 2 bytes manually, the length and other fields can be manually configured and the last few bytes calculated to be transmitted. For our illustration, the following code is shortcut method for remotely configuring an XBee using very little RAM and ROM:

```
DO
        DEBUG CR,"Hex Address of Remote? "        ' Enter address
        DEBUGIN HEX Addr
        DEBUG "2-letter Command? "                ' Enter config code
        DEBUGIN Code(0)
        DEBUGIN Code(1)
        DEBUG CR,"Hex Value? "                     ' Enter value for config
        DEBUGIN HEX Value
        Checksum = $FF-$17-2-Addr-code(0)-code(1)-value ' Calc checksum
        DEBUG "Sending:",CR
        SEROUT Tx,Baud,[$7E,0,17,$17,              ' start, length, ident
                       0,REP 0\8,                  ' Frame ID 0, 0 64-bit
                       Addr.HIGHBYTE, Addr.LOWBYTE, ' address
                       2,                          ' option - apply now
                       code(0),code(1),            ' Config code
                       Value.HIGHBYTE,value.LOWBYTE,' Value to send
                       checksum]                   ' Checksum value
LOOP
```

On receiving data, you may take advantage of special formatters within the **SERIN** *Inputdata* argument. **WAIT** can be used to wait for the start delimiter, and **SKIP** can be used to skip over data to the pertinent data:

```
SERIN Rx,Baud,[WAIT($7E), SKIP 7, DEC Value]
```

Summary

When working with the XBee individual I/O, it's important to ensure that proper voltages are applied to the 3.3 V device to prevent damage. Sleep mode allows placing the XBee module into a low-current state. Using API, data can be framed for transmission to a particular node and received data provides additional information and uses. The sending node's address and the RSSI level can often be pulled from the frame, and options such as remote configuration and transmitting ADC and digital data are available. While the coding for frames can be complex, the XBee object can assist when

working in API mode for the Propeller. Use of API mode on the BASIC Stamp should be weighed against the need for the additional features it provides.