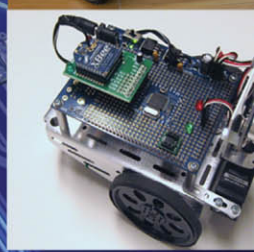
The Official Guide

PROGRAMMING AND CUSTOMIZING THE MULTICORE PROPELLER™ MICROCONTROLLER



PARALLAX INC.

Shane Avery, Chip Gracey, Vern Graner, Martin Hebel, Joshua Hintze,
André LaMothe, Andy Lindsay, Jeff Martin, and Hanno Sander

PROGRAMMING AND CUSTOMIZING THE MULTICORE PROPELLER™ MICROCONTROLLER

5

WIRELESSLY NETWORKING PROPELLER CHIPS

Martin Hebel

Introduction

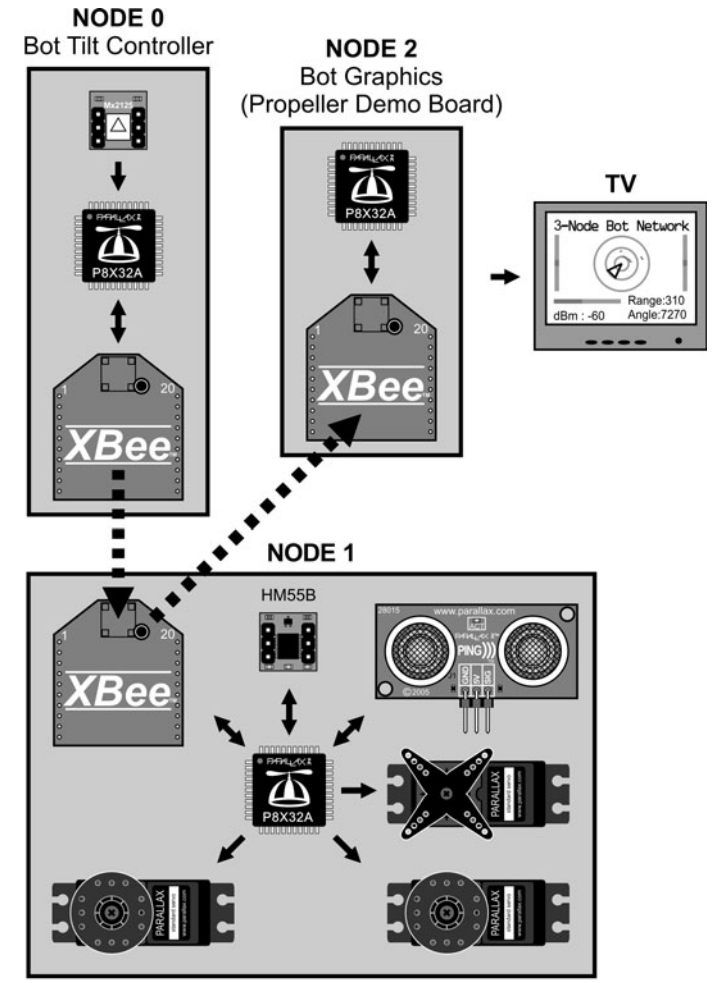
This chapter looks at how your Propeller can be part of a *wireless sensor network* (WSN) to share data through wireless communications. WSNs are not intended for large data transfers, such as files, but small amounts of data back and forth. The Propeller is an amazing controller, and its ability to perform parallel processing makes data communications fast and simple for use in a WSN. While the main task is being carried out, other cogs can be sending or receiving data on the network.

With a lot to discuss and learn along the way, the final completed project of this chapter, depicted in Fig. 5-1, will be a three-node network that has:

- A tilt-controller node transmitting drive and control data.
- A robot (bot) node that receives the data; has a compass and ultrasonic range finder; and is transmitting data on drive, range, and direction. It also has the ability to "map" what is in front of it for remote display.
- A node that accepts data from the bot and displays the information graphically on the TV.

This chapter highlights communications to, from, and between Propeller chips using XBee® transceivers from Digi International. Topics covered in this chapter include:

- Networking and XBee overview
- PC-to-XBee communications
- Configuring the XBee manually and with the Propeller



Networked Bot (Propeller Proto Board) on Boe-Bot Chassis

Figure 5-1 Three-node network for monitoring and control.

- PC-to-Propeller and Propeller-to-Propeller communications with the XBee
- Transparent and API data modes of the XBee
- Forming a multi-node Propeller network for robot control and monitoring

This chapter will work through several examples of communications, but really, the intent and focus is on *how* to perform the communications with the Propeller. It is left to you, the reader, to take the principles discussed, combine them with your imagination or needs, and develop a Propeller network of your own. Many other projects and information from this text can be combined with this chapter for truly amazing projects!

Resources: Demo code and other resources for this chapter are available for free download from ftp.propeller-chip.com/PCMProp/Chapter_05.

Overview of Networking and XBee Transceivers

The ability to communicate wirelessly has had such a significant impact on personal and data communications that many today cannot envision life without the use of cell phones, Wi-Fi networks and Bluetooth® features in personal devices. The ability of these devices to communicate on their respective networks (even your Bluetooth headset forms a network with the player) relies on key features:

- The use of addressing to send data to specific destination devices and to identify the source of the data
- The use of framing and packets to encompass the data itself in a "package" with necessary information (such as the destination address)
- The use of error checking to ensure the data arrives at the destination without errors
- The use of acknowledgements back to the source so that the sender knows the data arrived correctly at its destination

Simple two-device (or two-node) systems may not need all these features. It's really dependent on the needs of the network, but if ensuring data arrives correctly to an intended destination is vital, then these features are a must.

The XBee uses a fully implemented protocol and communicates on a *low-rate wireless personal area network* (LR-WPAN), sometimes referred to as a *wireless sensor network* (WSN) with RF data rates of 250 kbps between nodes. For the seasoned network readers, LR-PANs operate using IEEE 802.15.4, a standardized protocol similar to Wi-Fi (IEEE 802.11) and Bluetooth (IEEE 802.15.1). The XBee is currently available in the XBee 802.15.4 series and the XBee ZigBee/Mesh series. The 802.15.4 series (often referred to as Series 1) is the simplest and allows point-to-point communications on a network. The ZigBee/Mesh series (Series 2) uses the ZigBee® communications standard on top of 802.15.4 for WSNs to provide self-healing mesh networks with routing. This chapter will focus exclusively on the XBee 802.15.4 and its higher-power sibling the XBee-Pro 802.15.4. These will be referred to as simply the XBee.

Key benefits of using the XBee include the ability to perform addressing of individual nodes on the network, data is fully error-checked and delivery acknowledged, and data can be sent and received transparently—simply send and receive data as if the link between devices were directly wired. XBees operate in the 2.4 GHz frequency spectrum.

An image and a drawing of an XBee are shown in Fig. 5-2. The XBee is a 20-pin module with 2.0 mm pin spacing. This can cause some aggravation when working with breadboards and protoboards, which have 2.54 mm (0.1 in) pin spacing, but solutions to this will be addressed.

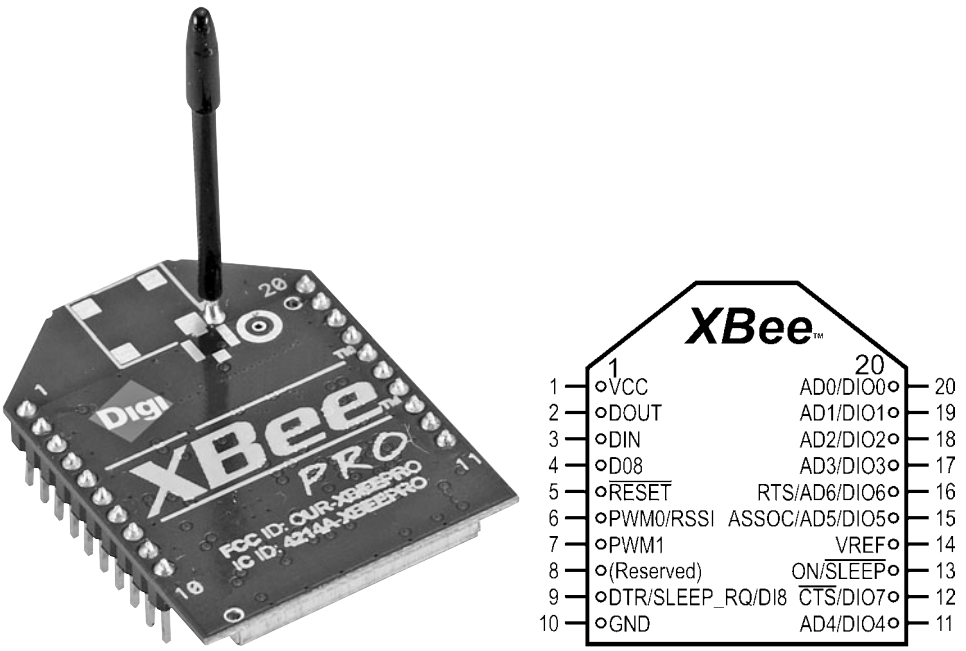


Figure 5-2 **XBee module and pins.**

Don't get scared! The XBee has a large number of pins, but for most of this chapter, we will use only four:

- Vcc, Pin 1: 2.8 V to 3.4 V (Propeller Vdd voltage)
- GND, Pin 10 (Propeller Vss)
- DOUT, Pin 2: Data out of the XBee (data received by Propeller)
- DIN, Pin 3: Data into the XBee (data to be transmitted by Propeller)

Other pins include a sleep pin (Sleep_RQ) for low power consumption, flow control pins (RTS/CTS), analog-to-digital (ADC) inputs, digital inputs and outputs (DIO), among others. This chapter will discuss some of these other pin functions, but the focus is on simply sending and receiving data between the Propeller and XBees using the DOUT and DIN pins.

Note: Please see the XBee manuals on Digi's web site for in-depth discussion and information: www.digi.com and included in the distribution files.

The XBee has a current draw of around 50 mA and a power output of 1 mW with a range of about 100 m (300 ft) outdoors. The XBee-Pro has a current draw of 55 mA when idle or receiving data and 250 mA when transmitting. With a power output of 100 mW, it has a range outdoors of 1600 m (1 mi) line sight. They both have sleep

modes, with current draws of less than 10 μA, but can't send or receive data while sleeping. There are different antenna styles as well, though the whip antenna is probably the most popular.

Tip: Don't get too excited about the distances. Line-of-sight communications rely on height as well as distance. Due to ground reflections and deconstructive interference (Fresnel losses), the heights of the antennas need to be taken into account. For good communications at 100 m, a height of 1.4 m (4.6 ft) is recommended.

Information: For more insight on distance, height issues, and calculations, search the web for "Fresnel clearance calculation."

Though the XBee is ready to go right out of the box, it is feature-rich and can be configured for specific applications.

Hardware Used in This Chapter

The following is a list of hardware used in this chapter and their sources, but as you read through, you'll find it's not written in stone. We recommend you read through the chapter to understand how the hardware is used before making an expensive investment.

- 2—Propeller Demo Boards (Parallax)
- 1—Propeller Proto Board (Parallax)
- 1—Prop Plug (Parallax)
- 3—XBee 802.15.4 (Series 1) modem/transceivers (www.digikey.com)
- 3—AppBee-SIP-LV XBee carrier boards (www.selmaware.com or other styles available on www.sparkfun.com)
- 1—PING))) ultrasonic sensor (Parallax)
- 1—HM55B compass module (Parallax)
- 1—Memsic 2125 accelerometer/inclinometer (Parallax)
- 1—Boe-Bot chassis (Parallax)
- 1—Ping Servo Mounting Bracket Kit (Parallax)
- 2—Additional Boe-Bot battery holders or other portable battery source
- Miscellaneous resistors

Testing and Configuring the XBee

An important step in constructing a complex project is to make sure the individual devices work properly and their use is understood. In this section, the XBees will be tested, configuration settings explored, and means of configuring these devices discussed.

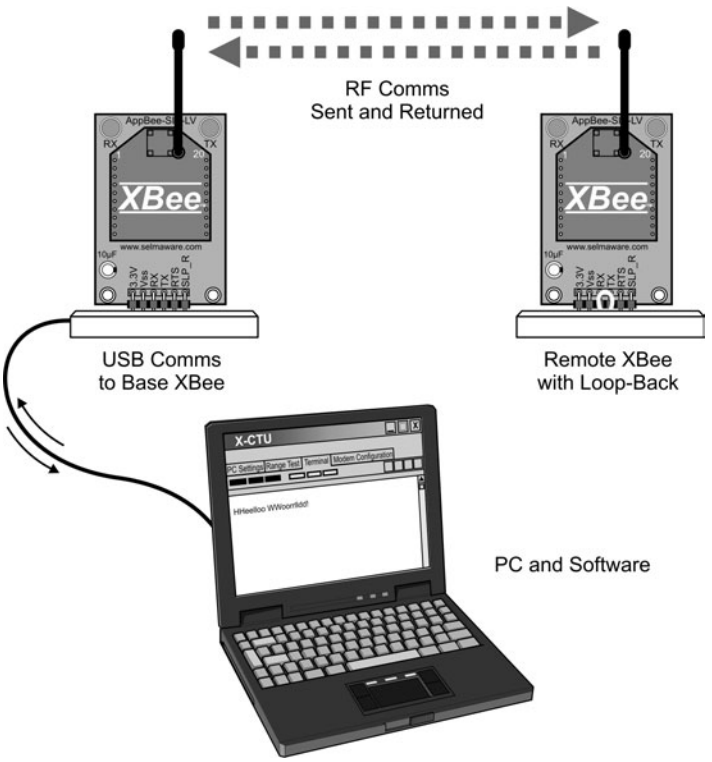


Figure 5-3 Configuration and testing diagram.

Figure 5-3 shows the diagram for this test. A PC will communicate directly to an XBee, and a remote XBee is set up with a loop-back jumper. In the loop-back, the DOUT line of the XBee is tied to its DIN so that any RF data it receives is looped back into the device to send it out again via RF.

The following is a list of the hardware and software used for this test, but there are many ways to achieve the same results. Essentially, a means is needed to communicate to an XBee serially from the PC and means to supply power to the base and remote XBees.

Equipment and other software:

- 2—Propeller Demo Boards (Parallax)
- 2—XBees (www.digikey.com)
- 1—Prop Plug (Parallax)
- 1—AppBee-SIP-LV from Selmaware Solutions (www.selmaware.com)
- X-CTU software from Digi International (www.digi.com)

The AppBee-SIP-LV is simply a carrier board for the XBee providing 3.3 V power from the Demo Board and access to I/O in a breadboard-compatible header. Figure 5-4 shows the AppBee-SIP-LV and a drawing of the physical connections to the XBee.

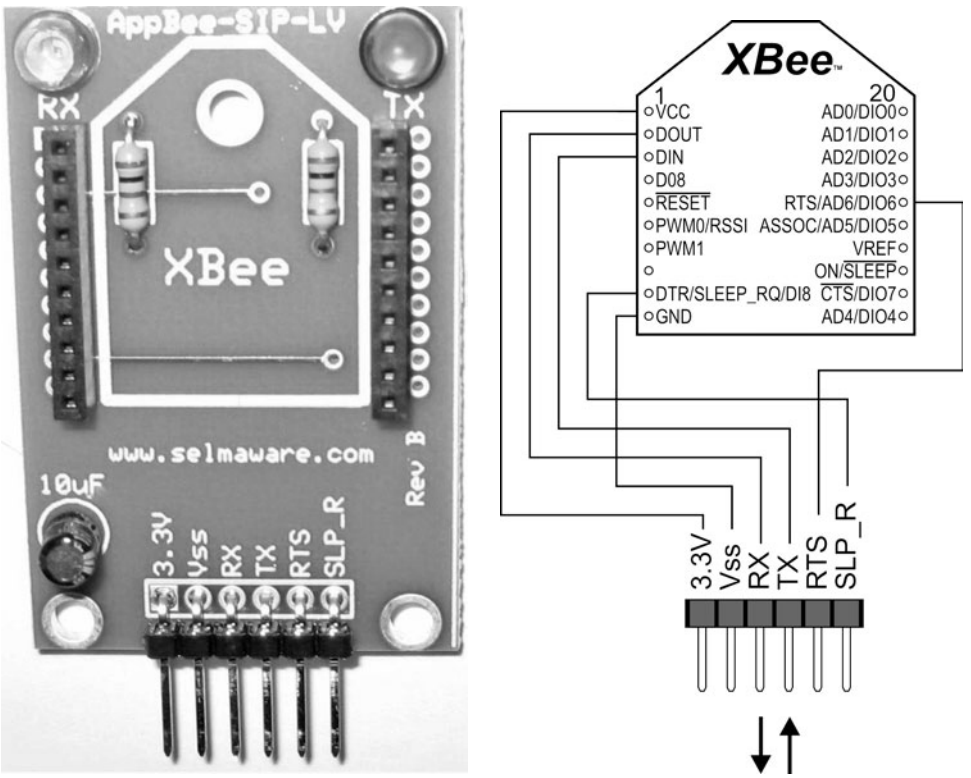


Figure 5-4 AppBee-SIP-LV carrier board and drawing with physical connections.

Tip: Another good source of carrier boards and other XBee accessories is www.sparkfun.com. Search their web site for XBee.

ESTABLISHING PC-TO-XBee COMMUNICATIONS

The first task is to communicate with the XBee directly from the PC for configuration changes and monitoring. Figure 5-5 shows two ways of establishing communications: using the Propeller as a serial pass-through device or communicating directly with the XBee using the Prop Plug as a serial interface. Either method allows the serial connection between the PC and the transceiver.

If you are using the Propeller to pass serial communications, the program Serial_Pass_Through.spin should be downloaded using F11. If the serial communications port is closed in the software, the Propeller may be cycled when the DTR is toggled, reloading the Propeller from EEPROM. Using F11 ensures a cycling of the Propeller will reload the correct program.

The program itself is simple but highlights the power of Propeller. Microcontrollers that provide multiserial communications are difficult to find. Two instances of the

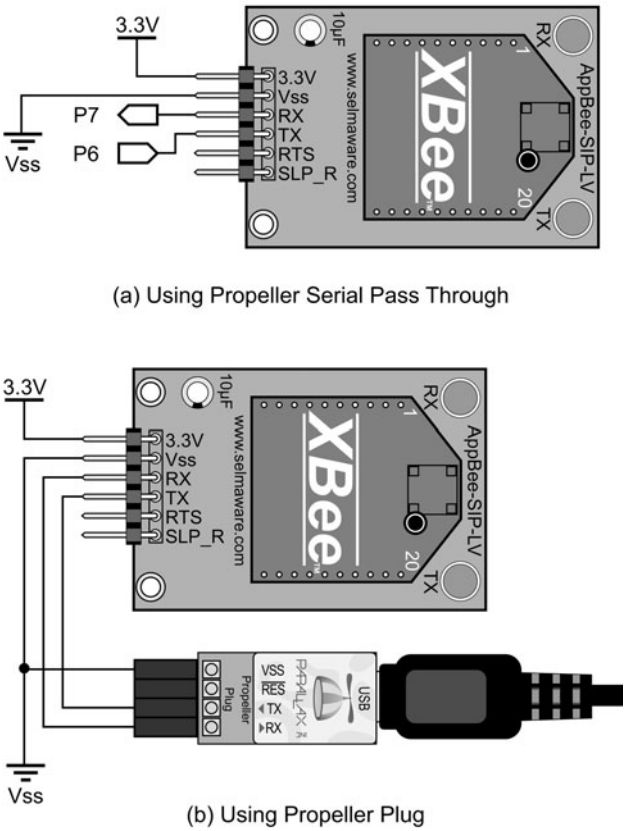


Figure 5-5 **Two methods of PC communications with XBee.**

FullDuplexSerial object establish the transparent link. Data from the PC is sent to the XBee, and data from the XBee is sent to the PC; with each method in separate cogs, it allows transfer speeds tested up to 115,200 bps. But for now we need to stick to 9600 bps since that is the default configuration on the XBee.

```
OBJ
  PC    : "FullDuplexSerial"
  XB    : "FullDuplexSerial"

Pub  Start

  PC.start(PC_Rx, PC_Tx, 0, PC_Baud) ' Initialize comms for PC
  XB.start(XB_Rx, XB_Tx, 0, XB_Baud) ' Initialize comms for XBee
  cognew(PC_Comms,@stack)        ' Start cog for XBee--> PC comms
```

```
  PC.rxFlush                    ' Empty buffer for data from PC
  repeat
    XB.tx(PC.rx)                ' Accept data from PC and send to XBee

Pub PC_Comms
  XB.rxFlush                    ' Empty buffer for data from XB
  repeat
    PC.tx(XB.rx)                ' Accept data from XBee and send to PC
```

Caution: Watch the I/O numbers! If another configuration is used, modify the pin numbers in the CON section of the code.

If you are using the Propeller for passing serial data:

✓ Connect the hardware as shown in Fig. 5-5*a*.
✓ Download the Serial_Pass_Through.spin program to the Propeller using F11.

If you are using the Prop Plug to communicate directly, connect it as shown in Fig. 5-5*b*.

✓ If you haven't yet, download and install the X-CTU software available in the distributed files or from Digi's web site. There is no need to check for updates—this can take a long time and the basic installation has all that is needed for now.
✓ Open the X-CTU software. It should look similar to Fig. 5-6. Select the COM port that your Propeller is communicating through.
✓ At this point, use the Test/Query pushbutton to test communications with the XBee.

Caution: As always, only one software package can access the same COM port at any time. You'll get used to slapping your head when you can't communicate as you go between the Propeller tool software and X-CTU!

Tip: If communications fail, recheck your hardware and pin numbers, reload the Propeller program, and verify no other software is using the COM port. If you continue to have problems and it is not a brand-new XBee, the serial baud rate may have been changed or the XBee may be in API mode—test various baud rates and check the API box to test.

If all went well, you may have seen the RX and TX lights blink on the board and received a message informing you communications were okay, along with the firmware version on the XBee.

✓ Select the Modem Configuration tab on the X-CTU software.
✓ If your XBee was reconfigured, this would be a good time to click the Restore button to return it to the default configuration.
✓ Click the Read button.

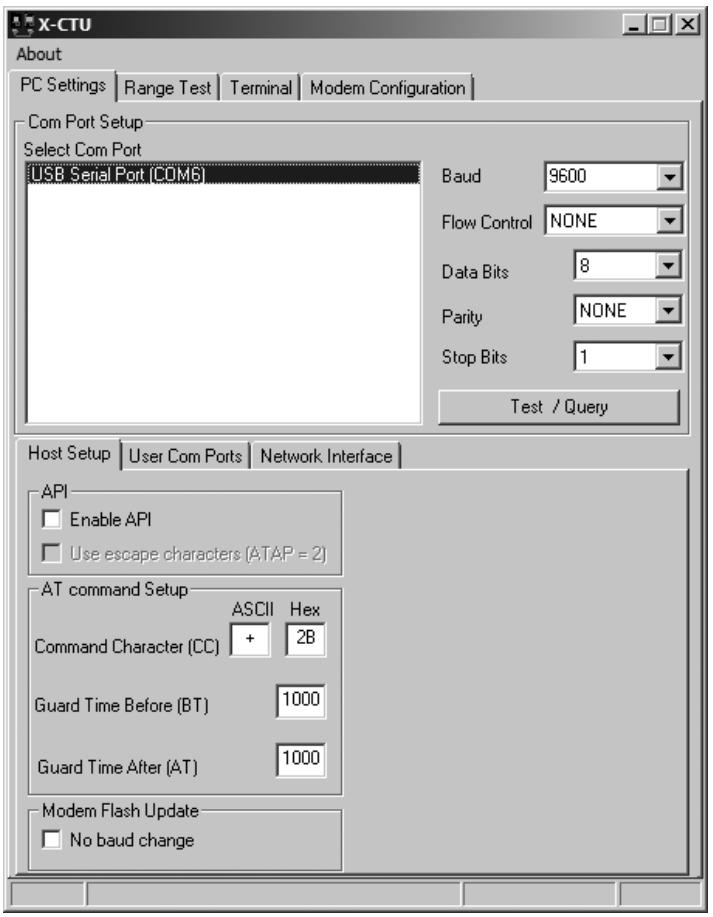


Figure 5-6 **X-CTU software showing COM port selection.**

The screen should have loaded with the configuration setting of the XBee as shown in Fig. 5-7. Many of them will be explained shortly—we're only going to use a handful of the settings available. But for now, let's test out some wireless communications.

TALKING XBee TO XBee USING LOOP-BACK

With a second XBee, supply power and connect a jumper between DOUT and DIN (or RX and TX on the carrier board), as illustrated in Fig. 5-8, using the AppBee-SIP-LV carrier board (or similar). Do not connect to any Propeller I/O at this time—we are simply using the board for power. We used a second Demo Board for this test.

✓ Power up the remote XBee with loop-back jumper in place.
✓ Click the X-CTU Terminal tab.
✓ Type "Hello World!"

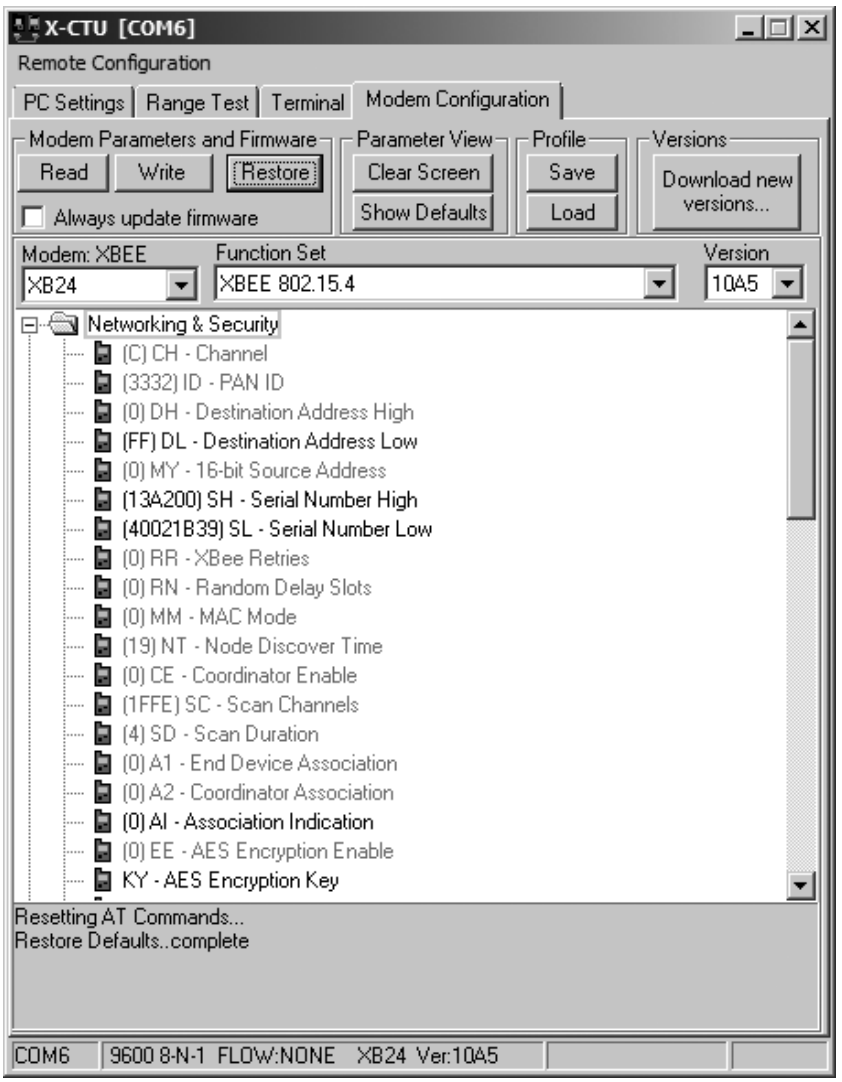


Figure 5-7 X-CTU software showing XBee configuration settings.

You should see the TX and RX lights flashing on both units (if using the AppBee carrier) and text in your Terminal window. You should see two of each character—what you typed in blue and what was echoed back and received in red—as shown in Fig. 5-9.

Tip: Having problems? If you don't see any data returning, be sure the remote XBee is connected properly. If it is not a new XBee, if may have been configured differently. Turn off both units and swap the XBees. After powering up, "Restore", the XBee to default configuration using the X-CTU button, read the second XBee using the X-CTU software, and test again.

Figure 5-8 **Remote XBee connections for loop-back.**

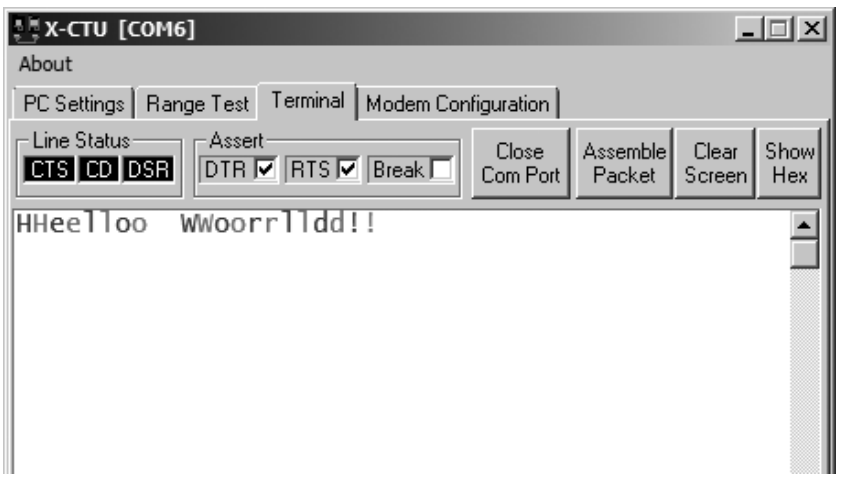


Figure 5-9 **X-CTU Terminal window.**

Tip: Beyond testing purposes, the X-CTU software is not essential, and any terminal program or other serial software package may be used, such as the PST Debug-LITE software used in previous chapters. Just ensure baud rates match between the software and the devices.

As noted, each character is transmitted as it is typed. The XBee can actually send a string of characters at once (up to 100), but it only waits so long before assembling a packet to be transmitted. We type too slowly to get multiple characters quickly enough with the default configuration, but we can assemble a packet of characters that will be kept together:

- ✓ On the X-CTU Terminal window, click Clear screen, and then click Assemble Packet.
- ✓ Type "Hello World!" in the packet box, and click Send Data.

You'll notice your text is returned as a single packet.

One last test is the range test. This allows you to monitor the signal strength from –40 dBm to the XBee's sensitivity limit of around –100 dBm by having the software repeatedly send out a packet to be echoed.

- ✓ Check the check box below the vertical RSSI (receiver signal strength indication).
- ✓ Click Start.
- ✓ Monitor the number of good packets received and signal strength.
- ✓ Block the area between the XBees or move the remote XBee to another room, and test the effect on RSSI level.

Note: In theory, you should never see a bad packet (malformed data) in the received data from the XBee, such as in the Terminal window. All data is error-checked and retried if there is no response or if the error check fails. You should receive either good data or no data at all. The serial-link issue with the XBee is a more probable cause than an RF issue with bad data.

Now that we have an RF link going, it's time to discuss and test some XBee configurations.

XBee CONFIGURATION SETTINGS

As seen, the XBee has numerous settings that can be configured. This configuration can be performed through the Configuration window, through the Terminal window, or through strings sent out from the Propeller. Let's first take a look at some of the more important settings shown in Table 5-1 for this chapter (we will use only a few) and others of interest should you delve deeper with your experiments. Click the Modem Configuration tab of the X-CTU software to view the settings. Clicking any setting will give a brief description and range of values at the bottom of the window.

OK, let's test out a few things:

- ✓ Test and verify your loop-back setup by sending a string.
- ✓ Under Modem Configuration, change DL to 1.
- ✓ Click Write.
- ✓ The XBee should be updated. Click Read and verify.
- ✓ Go to the Terminal window and type once again. You should get no response, and the remote RX light on the AppBee should not blink.

TABLE 5-1 SUMMARY OF PERTINENT XBee SETTINGS

COMMAND CODE	MEANING & USE
Networking & Security	
CH	Channel: Sets the operating frequency channel within the 2.4 GHz band. This may be modified to find to a clearer channel or to separate XBee networks.
ID	PAN ID: Essentially, the network ID. Different groups of XBee networks can be separated by setting up different PANs (personal area networks).
DL	Destination Low Address: The destination address where the transmitted packet is to be sent. We will use this often to define which node receives data. A hexadecimal value of FFFF performs a broadcast and sends data to all nodes on the PAN. The default value is 0.
MY	Source Address: Sets the address of the node itself. This will be used often in all our configurations. The default value is 0.
Sleep Modes	
SM	Sleep Mode: Allows the sleep mode to be selected for low power consumption (<10 µA). While we won't use it, a good choice is 1—Pin Hibernate. This would allow an output of the Propeller to put the XBee to sleep (using the Sleep Request pin) when it is not sending or expecting data.
Serial Interfacing	
BD	Interface Data Rate: Sets baud rate of the serial data into and out of the XBee.
AP	API Enable: Switches the XBee from transparent mode (AT) to a framed data version where the data must be manually framed with other information, such as address and checksum. This is a powerful mode and will be explored in this chapter.
RO	Packetization Timeout: In building a packet to be transmitted, the XBee waits a set length of time for another character. If not received in the set time, the packet is sent. This is why as we typed characters, each was sent and echoed back. This can be important to change if you have multiple units sending data to one node to ensure that all data sent is received as a single transmission from one unit; otherwise, you may get data from various nodes intermixed.
I/O Settings	
D0 – D8	Sets the function of the I/O pins on the XBee, such as digital output, input, ADC, RTS, CTS, and others.
IR	Sample Rate: The XBee can be configured to automatically send data from digital I/O or ADCs. It requires the receiving node to be in API mode and the data parsed for the I/O values.
Diagnostics	
DB	Received Signal Strength: The XBee can be polled to send back the RSSI level of the last packet received.
EC	CCA Failures: The protocol performs clear channel assessment (CCA)—that is, it listens to the RF levels before it transmits. If it cannot get an opening, the packet will fail and the CCA counter will be incremented.

EA	ACK Failures: If a packet is transmitted but receives no acknowledgement that data reached the destination, EA is incremented. The XBee performs two retries before failure. Additional retries can be added by using the RR setting.
AT Command Options	
CT	AT Command Timeout: Once in command mode, this sets how long of a delay before returning to normal operation.
GT	Guard Time: When switching into AT command mode, this defines how long the guard times should be (absence of data before the command line) so that accidental mode change is not performed.

By changing DL to 1, data is intended for an XBee at address 1. The default settings on XBees are a DL of 0 and an MY of 0. Previously, we were sending data to a node at address 0 from a node at address 0 and vice versa. Be aware, the XBee actually does receive data, sees it is not the intended node, and then dumps it instead of passing it to the DOUT pin (to which the RX LED is connected).

Let's now try configuring using the Terminal window. Due to timeouts, you may have to type a little fast, so you may need a few attempts. Enter the following lines—*do not* type what is in parentheses. `Press enter after each line except for +++.`

- ✓ (Wait three seconds since you typed anything last—this is guard time.)
- ✓ **+++** (*Do not* press ENTER.)
- ✓ (Wait a few more seconds and you should see that it is now in command mode.)
- ✓ `ATDL` (Requests the current DL value; it should return 1)
- ✓ `ATDL 0` (Sets the DL address to 0)
- ✓ `ATDL` (Again requests the DL address, which should be 0)
- ✓ `ATCN` (Exits AT command mode)
- ✓ `Hello World?`

If all went well, you should once again be getting echoes after changing the destination address back to 0. The waiting before and after the +++ is called the *guard time,* and it ensures that if a string containing +++ is sent, the unit won't flip into command mode inadvertently.

Tip: Permanent changes? Using the Modem Configuration feature of the X-CTU software, all changes are saved to nonvolatile memory and will still be in place after cycling power. Using the AT commands, the settings will revert to original values after cycling power, unless the ATWR (write) command is sent to write to nonvolatile memory.

The important aspect here is that just as we sent data strings to the Xbee for configuration changes, so can your Propeller configure the XBee through code. Multiple commands can be used in one line by separating them with commas. For example, the following sets DL to 0 and exits command mode: ATDL 0, CN.

TRY THESE!

- ✓ Try changing your MY address to 1 and sending data. You should see the remote unit receive and transmit, but you get nothing back. Why?
- ✓ Change your DL to FFFF. This is the broadcast address—any nodes on your network would receive it. Be sure to set MY back to 0 for the loop-back to work!
- ✓ Use the command ATND (Network Discovery). After a few seconds, you should see a list of other nodes in the network, including their MY address, two lines of the physical address (like a MAC address), and the RSSI level in hexadecimal.
- ✓ Use the command ATED (Energy Detect). You should see a list of about 11 hexadecimal values. This is the energy level seen on the various channels. Higher values are less noisy—a value such as 5A (hexadecimal), for example, converts to a level of –90 dBm.

✓ Use the Configuration tab to restore the XBee to its default values when done testing, or use the AT command ATRE, followed by ATWR, to save to memory.

UPDATING THE XBee VERSIONS

Just a note about the version of the XBees: In the Modem Configuration tab, you can see the version of firmware on your XBee, such as 1083, 10A5, or 10CD. Later versions are more capable. The majority of this chapter requires at least 1083. The firmware on the XBee can be updated by selecting a new version, checking Always update firmware, and clicking Write, but this requires more data lines than we have available with our configurations. A board such as the XBIB-U from Digi International or the WRL-08687, the XBee Explorer, from www.sparkfun.com (which can also double as a carrier board) is recommended. These boards can be used for direct USB access to the XBee as well as changing the firmware, and they supply power to the XBee.

Now that we can send and receive data and configure the XBee, we are ready to start using Spin and the Propeller to communicate via the XBee.

Sending Data from the Propeller to the PC

In this section we will equip a remote Propeller/XBee system with a couple of sensors and then transmit the data from the sensors back to the base XBee to send the data to

the PC for monitoring. The base can be the Propeller using serial pass-through, using the Prop Plug to the XBee, or using a dedicated XBee-to-PC board, as previously mentioned. The sensors used for testing are Parallax's HM55B compass module and the PING))) ultrasonic range finder. These devices will eventually assist in our robot project, but you are free to modify the code to use any of the sensors previously explored in this text.

Additional equipment:

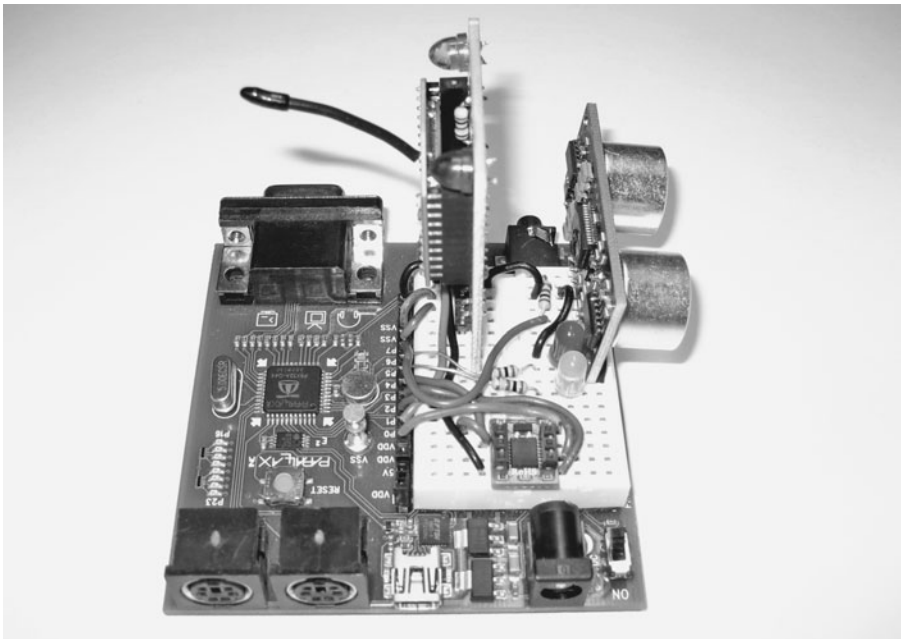
- HM55B Compass Module
- PING))) Ultrasonic Range Finder
- Or other sensors as desired, with appropriate code

Figure 5-10 is an image of the nodes. Even though we don't need to just yet, we will use this opportunity to set the DL address of the remote unit to 0 to ensure it is sending data to the base unit.

- ✓ Connect the PING))) sensor and HM55B compass on the remote unit as shown in Fig. 5-11. If a different I/O pin is used, update the pin numbers accordingly in the `CON` section of the code. Connect the LEDs as well; we will use them shortly.
- ✓ For the base unit XBee, open and clear the X-CTU Terminal window. Open the COM port if closed. Having that port in use will help ensure the correct Propeller is programmed.
- ✓ Download Simple_PC Monitoring_from_Remote.spin to the remote unit.
- ✓ Monitor the remote unit's LEDs—they should blink rapidly a few times after several seconds as the XBee is configured.
- ✓ Monitor the base unit's Terminal window. A "ready" message should be displayed, then the readings of the sensors should be reported every half-second.
- ✓ Test the compass bearing. It should read 0 to 8191 (roughly) as you rotate it, with 0 being approximately magnetic north.
- ✓ Test the range finder by placing an object in front and moving it in and out. The PING))) sensor will report distances from roughly 30 to 3000 mm (3 cm to 3 m).
- ✓ If either sensor fails to respond properly, check your connections and code.

Tip: The range finder has a fairly large angle of emission and detection. Test this by putting an object to the side of range finder and going in and out to determine how wide the angle is at different distances.

After initializing the XBee and compass, there is a three-second delay, +++ is sent followed by another three-second delay and the string of "ATDL 0, CN." Finally, a byte of 13 representing a CR or ENTER key is sent. The destination address is set to 0 and command mode is exited (CN) in exactly the same fashion as you did in the Terminal window.



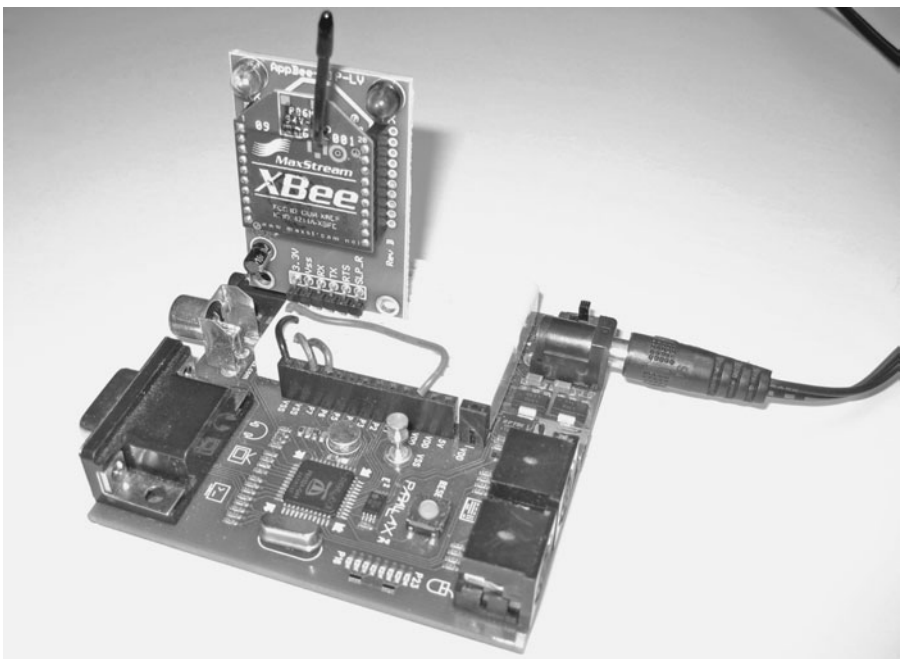


Figure 5-10 Base and remote nodes.

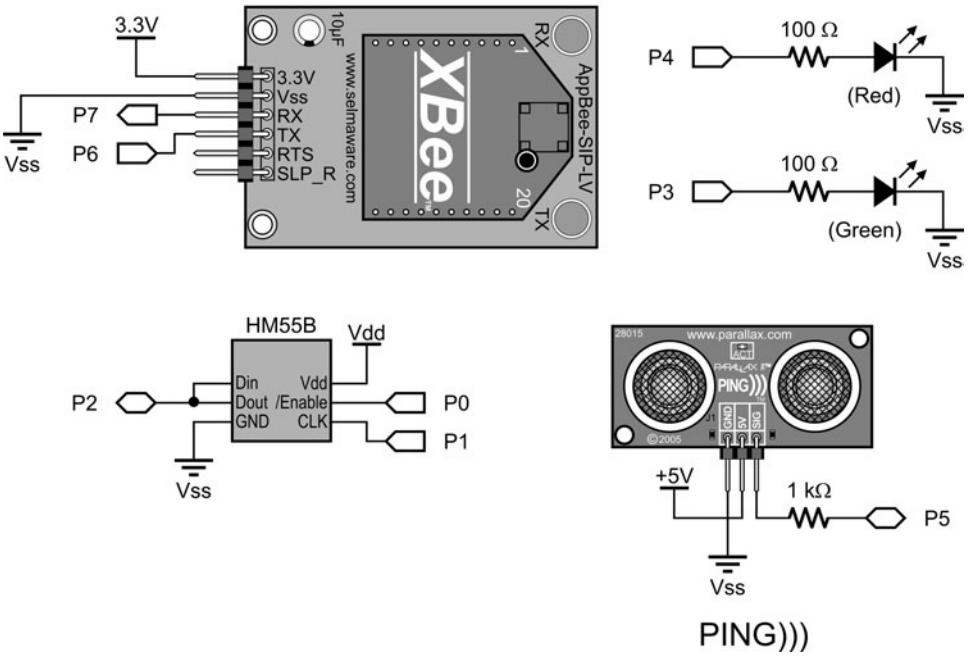


Figure 5-11 Remote unit with PING))) and compass.

```
delay(3000)                          ' Guard time for AT mode
XB.str(string("+++"))                ' Send AT command request
delay(3000)                          ' Guard time
XB.str(string("ATDL 0,CN"))          ' Send code to set DL = 0
XB.tx(13)                            ' Send carriage return
```

The command codes from the Propeller are passed to the XBee using the FullDuplexSerial object duplicating your serial terminal. Through this method any number of commands may be sent to the XBee for configuration changes on initialization or during operation. Those 3-second guard times can cause lag during operation, but we'll deal with that soon.

In the `SendData` method, you can see that the range and direction (`theta`) are read from the devices. Using a combination of text strings and `XB.dec` (decimal) methods, the data is sent to the base XBee, where it is passed through to the PC for monitoring in the Terminal window. Figure 5-12 is a sample output of received data.

```
repeat
  range := Ping.Millimeters(PING_Pin)      ' Get range in mm
  theta := HM55B.theta                     ' Get bearing (0-8191)
  XB.str(string(13,13,"Ping Range(mm): "))' Send string to base
  XB.dec(range)                            ' Send range as decimal
  XB.str(string(13,"Direction(0-8192): "))' Send string to base
  XB.dec(theta)                            ' Send bearing as decimal
  delay(500)                               ' Short delay before repeat
```

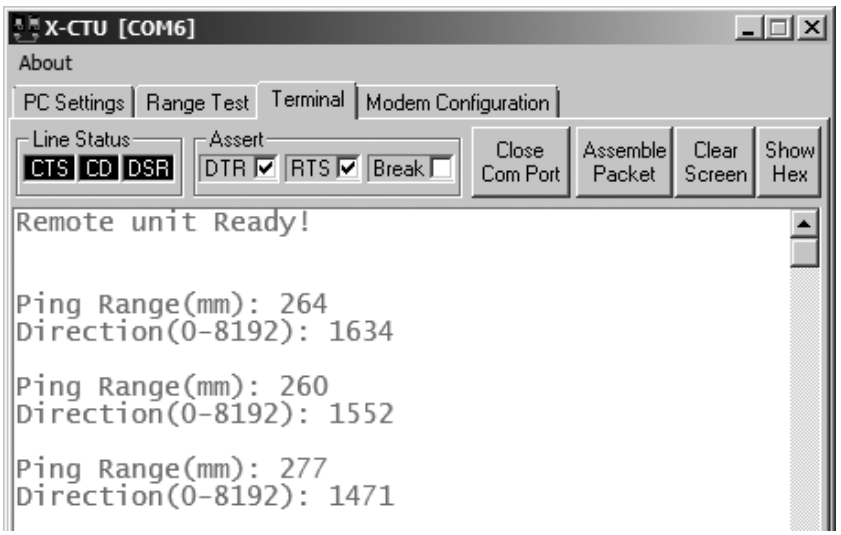


Figure 5-12 Sample output in Terminal window of range and bearing.

TRY IT!

✓ Try adding a simple device, such as a pushbutton, and reporting its state back to the PC. If you are out of I/O, you may remove the LEDs.

Polling Remote Nodes

In an LR-PAN, nodes typically come in one of three flavors:

- *Coordinators* help manage the network, from controlling communications to assigning information to devices.
- *End devices* are used to read and control devices on the network.
- *Routers* are used to pass data between nodes at distances too far to reach directly.

There is nothing prohibiting end devices from talking to one another, and once a network is established, the coordinator's job may come to an end. In this chapter we will refer to the base unit, the one at the PC, as a coordinator because it will help control communications and be a common collection point. Our remote nodes will be end devices that we will monitor and control.

Multinode communications can be tricky. Aspects to be dealt with include: Which node can send data when? When data arrives, who is it from? Do nodes need permission to talk or can they do so at any time? We need to ensure that nodes don't talk over one another

(causing collisions on the network) and that the receiving units know who the data is from in order to respond appropriately or take some other action. XBee, using IEEE 802.15.4, works similar to Wi-Fi. A node listens before it transmits to help ensure that no other node is transmitting at the time (this is *Clear Channel Assessment,* or CCA). Delivery of data is verified through acknowledgements. If the sender does not get a response, it tries again. This method is known as CSMA/CA or *Carrier Sense, Multiple Access/Collision Avoidance*. Unlike Ethernet, which uses collision detection (CSMA/CD), a node cannot listen once it starts transmitting so it cannot detect collisions.

So the data link layer of communications helps ensure data gets passed properly, but it still doesn't assist in higher-level functions controlling the who and when of communications. In the next section we will look at a method of using a Propeller acting as a coordinator to poll end devices for their data. USB works in much the same way—each device is polled one at a time to see if they need access or have data to send.

COORDINATOR MANUALLY POLLING REMOTE END DEVICES

A hardware configuration similar to the one from the previous section will be used, but this time, the Propeller needs to be in the communications chain at the base instead of simply using a Prop Plug for XBee communications. Also, to demonstrate control action, the two LEDs on the remote end device provide control action. You are welcome to have as many end points as you desire (well, up to 65,000), or just use one and change the end point's address to test. Figure 5-13 is a diagram of our network and hardware.

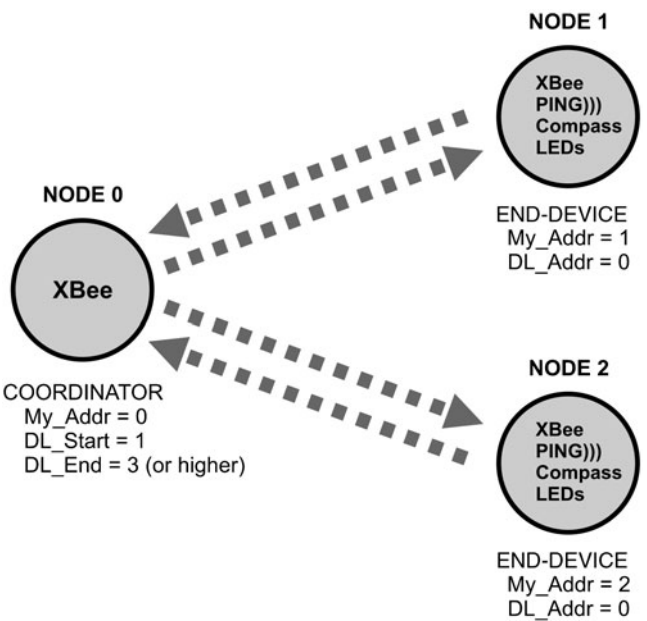


Figure 5-13 Hardware for coordinator polling.

In this example, the coordinator cycles through a range of end-point addresses by changing the DL value of the coordinator's XBee. It sends out codes and values to request data from each end point and to control the LEDs on each. Before allowing the coordinator to have control, we are going to manually test the control and responses.

- ✓ Add the LEDs to the remote end device.
- ✓ Open Acquisition_with_Control_End.spin.
- ✓ For each end device, number the constant `MY_Addr` in the `CON` section of the code sequentially from 1 up, skipping a few numbers to test "unresponsive nodes."
- ✓ Download Acquisition_with_Control_End.spin to each remote end device.
- ✓ Use the Propeller for serial pass-through or another PC-to-XBee configuration at the PC.
- ✓ Change the DL of the coordinator/base XBee to 1.
- ✓ In the Terminal window, type some p's and c's. If your end point at address 1 is awake, you should get values back for compass bearing and range finder distance.
- ✓ For this next test, use the "Assemble Packet" window. Type and send the following:
  - ○ Type i3 and then hit Enter.
  - ○ Type 1 and then hit Enter.
  - ○ Click Send.
- ✓ Change the 1 to a 0 and send again.
- ✓ Test again by using 4 instead of 3.
- ✓ What you should see is LEDs on P3 and P4 turning on with 1 and off with 0.

Figure 5-14 is an image of our communications test.

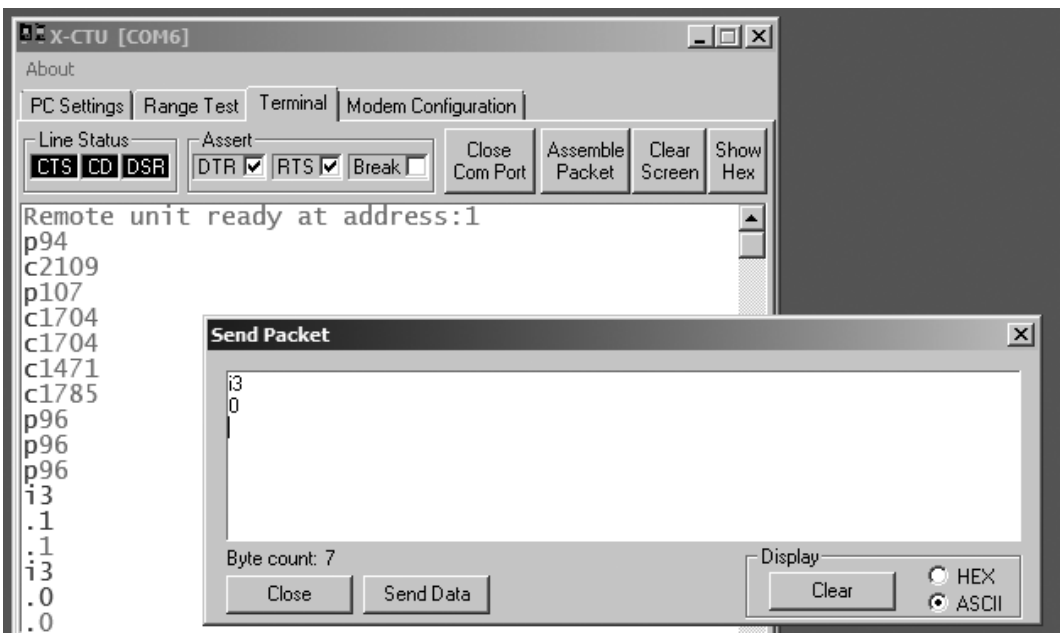


Figure 5-14 End-device responses to requests.

Looking at the end-device's code, data communications with the XBee is now through the XBee_Object. This is an object I wrote for easing some data communication and configuration issues. It uses FullDuplexSerial but greatly extends it.

Tip: The "XBee_Object" can be downloaded from Parallax's Object Exchange (http://obex.parallax.com). If you have previously downloaded it, be sure it is version 2 or higher. It is also included in the book's distributed files.

XB.AT_Init initialized the XBee to AT mode, allowing for short guard times (using ATGT), so instead of six seconds to modify a configuration, it can be done quickly in code. XB.AT_ConfigVal allows passing an AT command and a value to set configurations, such as the DL and MY addresses. The underlying code switches the XBee to command mode, sends data, and exits using the short guard times.

```
'' Enable XBee for fast configuration changes
XB.AT_Init

'' Set MY and DL (destination) address.
XB.AT_ConfigVal(string("ATMY"), MY_Addr)
XB.AT_ConfigVal(string("ATDL"), DL_Addr)
```

In the ProcessData method, XB.rx is used to tell the Propeller to wait for one character or byte of data. It then tests this character to determine what set of actions to take:

```
dataIn = XB.rx
Case dataIn
   "p":                                    ' p = PING distance
      range := Ping.Millimeters(PING_Pin)   ' Read PING in mm
      XB.dec(range)                 ' Send range as ASCII decimal value
      XB.cr                         ' End decimal string with CR

   "c":                             ' c = Compass
      theta := HM55B.theta          ' Read Compass
      XB.dec(theta)                 ' Send theta of bearing as decimal
      XB.cr                         ' End with carriage return

   "i":                               ' i = I/O control
       IO := XB.rxDecTime(timeout)    ' Accept IO number w/timeout
       state := XB.rxDecTime(timeout) ' Accept state (1/0) w/timeout
       if state <> -1
         dira[IO]~~                 ' Set direction of pin
         outa[IO] := state          ' Set state of pin
         XB.dec(outa[IO])           ' Send state back for verification
         XB.cr                      ' End decimal string with CR
```

If p, send back the decimal value of the range finder.
If c, send back the decimal value of the compass bearing.

If i, accept the next two decimal values and use them for I/O and State, which sets the I/O direction to be an output and the state of the I/O. RxDecTime is used to accept the decimal values with a timeout. This allows the program to continue to run if incorrect data is received following a timeout period. Should a timeout occur, a −1 is returned to the value. In accepting the data, note that each decimal value must end in an ASCII 13 or CR (or comma, see Sec. Data Acquisition and Control Using API Mode). Finally, the actual value of the I/O is sent back.

✓ Change the coordinator/base to a nonexisting end-device address (DL) and try again. You should get no data back.

What we are designing here can be considered a *protocol*—rules of communication. If you don't follow the rules set forth, nothing, or even incorrect things, may happen. When coding protocols, we attempt to cover all contingencies regarding what could go wrong and how they will be dealt with, such as i3 and no further data. What happens if you enter something other than a 1 or 0 for state? That contingency is not covered!

The data between the units is kept simple—byte codes and decimal strings. This allows short packets between the units and eases using the data in the code.

Caution: Be aware that currently the code can control *any* of the Propeller chip's I/O pins, so be careful of what you send for your IO values!

AUTOMATIC POLLING WITH THE PROPELLER

In this next exercise the Propeller will operate as the coordinator, polling each of the end devices in succession.

✓ Ensure you have downloaded Acquisition_with_Control_End.spin to your end device(s) using F11, with sequential MY_addr values while skipping a few values.
✓ In Acquisition_with_Control_Coor.spin, modify the values of DL_Start and DL_End in the CON section to match the range of your end-device addresses.
✓ Download Acquisition_with_Control_Coor.spin to the coordinator Propeller.
✓ Once downloaded, open the Terminal window.
✓ Wait and watch… you should see results similar to Fig. 5-15. Note that in this test only an end device with a MY_addr of 2 is responsive.

In Pub Start, once configured, the code loops through the range of defined end-device values, passing the address to the Poll method. Pub Poll accepts the address, sets the DL address, and informs the user. It then goes through a series of steps for acquisition and control.

Calling Control_IO, the I/O number and state are passed to turn on the LEDs. This method will send the correct i-instruction to control the end-device IO. The returning value with a timeout is accepted, passed back, and displayed.

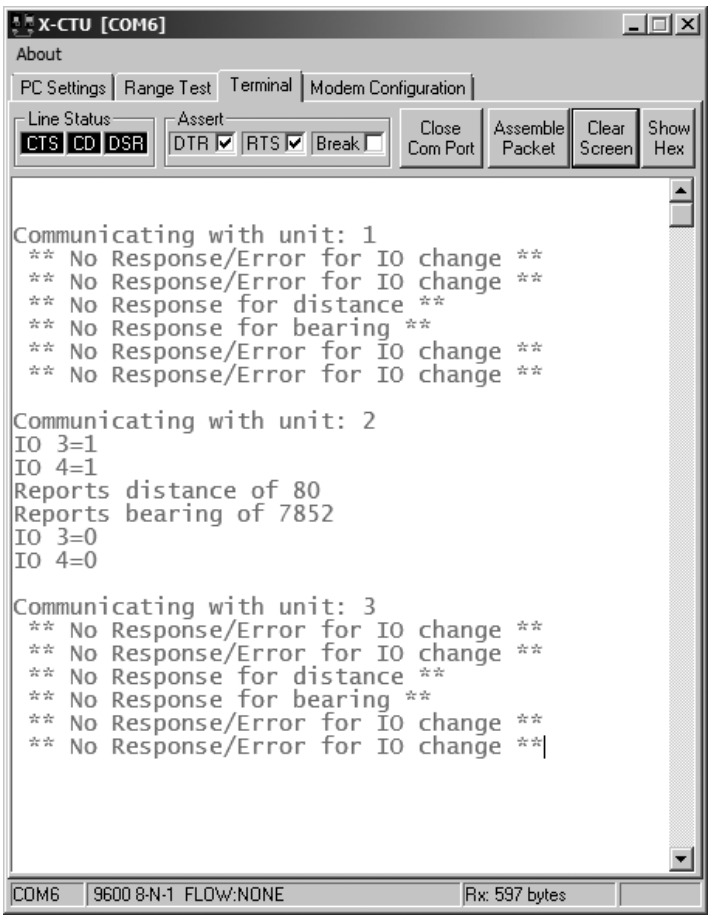


Figure 5-15 Coordinator responses from automated polling.

```
Pub Control_IO(pin, state) : Value
  XB.tx("i")                              ' Send i for IO control
  XB.dec(pin)                             ' Send pin as decimal value
  XB.cr                                   ' Send CR
  XB.dec(state)                           ' Send state as decimal value
  XB.cr                                   ' Send CR
  Value := XB.rxDecTime(200)              ' Accept value with timeout
```

Next the `GetDistance` method is called, which sends the p-instruction, accepts returning data, and passes it back for display. Then the `GetAngle` method is called; sends the

c-instruction; and accepts, returns, and displays data. Finally, the LEDs are again turned off using `Control_IO` sending 0s.

```
Pub GetDistance : mm

  XB.tx("p")                                  ' Send p to get range
  mm := XB.rxDecTime(50)                      ' Accept data with timeout
```

The cycle repeats each end-device value, pauses longer, and starts over. In each step of the way, timeouts are used to ensure nonresponsive end devices do not lock up the system and that they are reported as being nonresponsive.

In this example we are simply collecting data and controlling LEDs for testing purposes while displaying information for the user. The returned data could be used by the coordinator for some logical decisions or to control a local output or send data to another end device for action.

TRY THESE!

- ✓ Add another sensor and the code to request and respond with data.
- ✓ Use a returned value in some way at the coordinator, such as lighting an LED if the distance is within 100 mm (10 cm).
- ✓ Rapidly collect a remote value and plot it using ViewPort.

Though not used in our code, reading configuration values from the XBee can be done by sending the AT command and accepting returning data. The XBee uses hexadecimal for all values. The receiver is flushed to ensure that no data exists in the Propeller's object buffer. In this example, the dB level of a recent XBee reception is read and displayed.

```
XB.rxFlush
XB.AT_Config(string("ATDB"))
dataIn := XB.RxHex
PC.DEC(-dataIn)
```

Using the XBee API Mode

API MODE AND DATA FRAMING

Continual polling can take a lot of time and resources to check for data that may change infrequently. It is good to have the coordinator control the communications, but this requires the remote units to be awake. Another mode for the XBee is called API mode, for *application programming interface*. Instead of sending or receiving the data alone, the entire frame is manually constructed for transmission and manually parsed on reception. The frame consists of sender's address, RSSI level, options, frame IDs, and the data or message itself. Depending on the frame type, different types of data are carried. Some benefits to using API mode include:

- Pull sender's address directly from received frame.
- Pull RSSI level from certain received frame types.
- Place the destination address for the packet directly in the frame.
- Use frames for local XBee configuration as opposed to AT mode.
- Use frames for REMOTE XBee configuration (firmware version 10CD required).
- Pass analog and digital data from the XBee's I/O pins *without* a controller on the remote (firmware version 10A3 or higher required).
- Use frames that provide delivery notification to the sender.
- Data is received in a single frame (up to 100 bytes), ensuring it is from a single source.

As you can see, using API mode opens many doors to fast and powerful communications, but it can be a little complex. The XBee Object supports the means for constructing and retrieving data for many of the API frame types. Let's look at how a packet must be framed to be accepted for transmission, as shown in Fig. 5-16, taken from Digi International's XBee manual. This frame type is for sending strings between units, such as our data.

Note: Data is *always* sent in frames between XBees, but when in AT mode (transparent mode), the only thing we deal with is the data, or message, itself.

First, all frames start with a *start delimiter* so the receiving unit can locate where the start of a frame is as data pours in. Next is a 16-bit length (MSB and LSB), which has to match the number of bytes from after the length through to the checksum, but not including it. This is followed by the *API identifier,* a unique value telling the receiving unit what type of message it is.

Next is the *identifier-specific data,* consisting of the *frame ID* (if set to 0, it will suppress acknowledgement packets back to the controller; we will ignore these packets) and then the 16-bit destination address as 2 bytes. To send data to a unit at address 1, these would be values of 00 01. Options are set to disable acknowledgements or to send the data as a broadcast. Next is the actual data—up to 100 bytes. And finally, all the byte

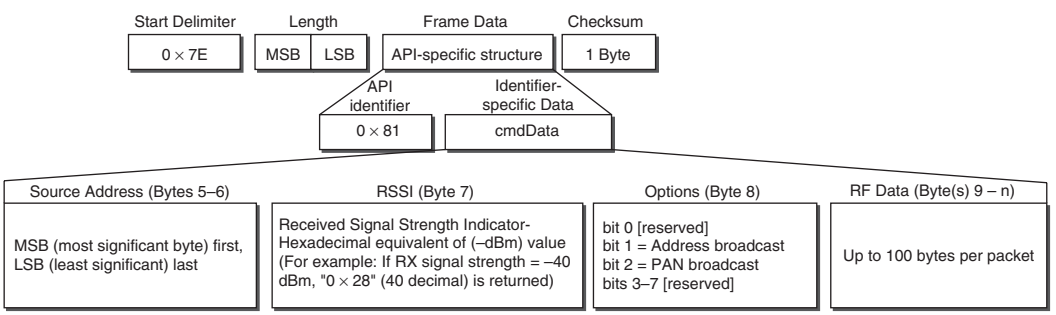


Figure 5-16 **API packet for transmitting string using 16-bit address.** (*Reprinted by permission of Digi International.*)

values up to that point are summed together to create a checksum value. The receiving unit will perform a summation itself, verifying against this value before using the data. If the packet is well formed, the XBee will accept this frame and transmit it. If not, it will be discarded.

Simple huh? Actually, it's not all that bad, but much more complex than just sending a string to be transmitted. Let's look at the Spin code that forms a packet when we send a string, such as `XB.API_Str(String("Hello!"))`.

From the XBee Object:

```
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 16
  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
```

As you look through the code, you can see how all the individual bytes that make up a well-formed frame for transmission are combined into an array of bytes, the bytes are summed (actually subtracted from $FF one at a time) for the checksum, and the array of bytes is transmitted.

When the data is received by the XBee, the frame is checked. If in API mode, the frame shown in Fig. 5-17 is sent to the Propeller for processing. Based on the API

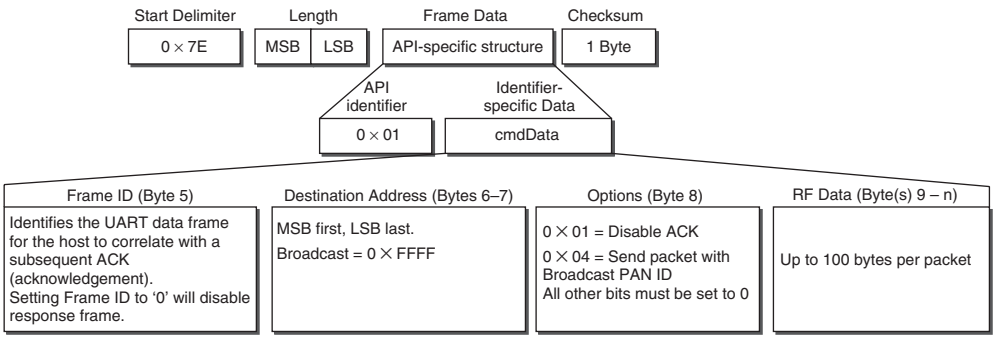


Figure 5-17 **API packet for received string using 16-bit address.** (*Reprinted by permission of Digi International.*)

identifier, the XBee Object can decide how to handle the frame data, and the top-level code can determine what to do with that type of frame data.

We won't go into the details, but again, specific bytes have specific meanings. In API mode, all this data is sent out to the Propeller. The XBee Object accepts the data and processes it accordingly using the `RxPacketNow` method. This method is actually private (`PRI`). The method called is `API_RX` or `API_RxTime(ms)`, which looks for the start delimiter ($7E). Once found, execution is passed to `RxDataNow` to accept remaining data. Once accepted, the identifier is checked to determine the type of packet, which in the case of our received string, would be $81. Next the packet is parsed, pulling out the data and placing it into variables that can be accessed from the top-level code, such as `XB.RxRSSI` to find out the value of RSSI for the packet, or `XB.srcAddr` to get the sources address. Note that the data is actually accessible through `XB.rxData`—it is not the actual data, but a pointer to where the data string resides in memory.

Other methods can help us pull decimal data out. After receiving data, calling `XB.ParseDEC(XB.rxData,2)` would pass the location of the string and pull out the second decimal value in the string (values can be separated by ASCII 13—CRs—or by commas).

In sending decimal values, an API_DEC does not exist. Numbers, unless sent as raw byte values, must be converted to a string and sent that way. The Numbers.spin object can aid in the conversion, such as sending the range in API mode:

```
XB.API_str(num.ToStr(range,num#DEC))
```

But that's the only thing that could be sent, since once called, the string is transmitted in a frame. To keep our data together, another method is used to assemble a string (packet) manually before sending it. This will be demonstrated in the example coming up. In API mode, all data to be sent in one transmission must be assembled first.

Note: Both transmitter and receiver *do not* need to be in API mode. One side can be using transparent transmission and the receiver using API reception and vice versa. This makes our job a little easier.

DATA ACQUISITION AND CONTROL USING API MODE

In this example, we will continue using the coordinator and end-device(s) hardware, but use API mode instead for data reception and transmission on the coordinator. The end devices have the ability to transmit at any time; while we have it on a delay, another option may be to use sleep mode for low power consumption and have it wake to transmit, or have it transmit only when some event takes place, such as a range being too close (someone is near!).

The end device will send a string for the values `range and theta` without being prompted. The receiver will accept the string in API mode and pull out the source address, RSSI level, and data. It will then send back strings to blink the LED on the end device.

- ✓ Open and modify the DL value in API_Mode_End.spin for each of your end devices. The value doesn't matter, as long as it is not more than 255 ($FF). We are sending only one byte to hold the address in our example. Note that the X-CTU software uses hexadecimal values when configuring as opposed to decimal.
- ✓ Download API_Mode_End.spin to your end device(s).
- ✓ Download API_Mode_Coor.spin to your coordinator.
- ✓ Open and monitor the coordinator's Terminal window.

The resulting data should be similar to that shown in Fig. 5-18.

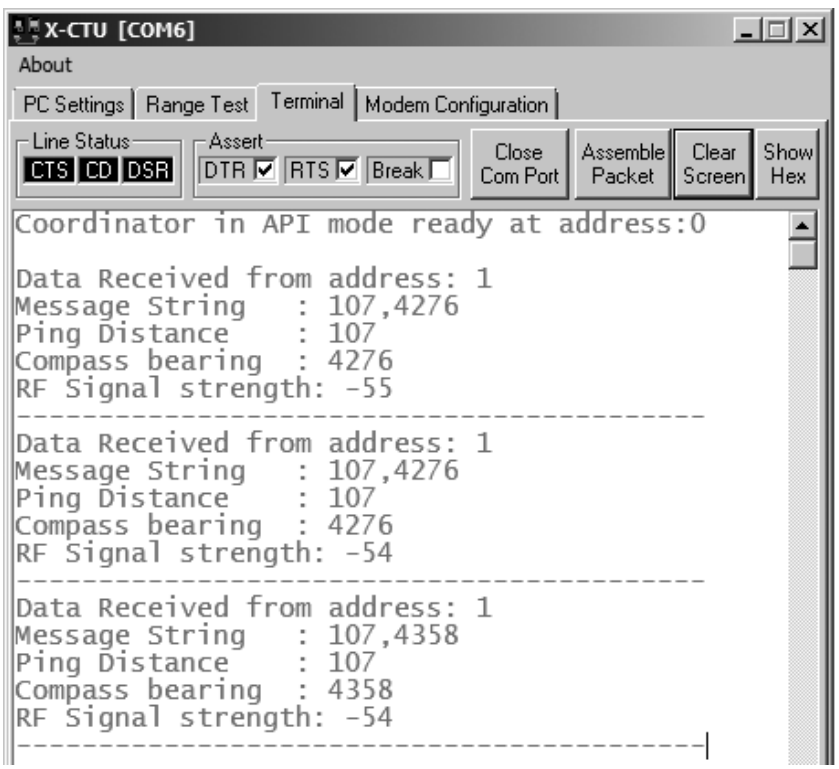


Figure 5-18 Example of terminal data from API data at coordinator.

In the end-device's code, the method SendUpdates is running in a separate cog to allow GetData to monitor for incoming data continuously. This allows data to be sent or received independent of the timing. The XBee is *not* in API mode, and SendUpdates sends the string values for range, theta every two seconds as decimal strings, such as the characters "1," "0," and "5" for the value of 105—three bytes' worth of data for one value. The unit does this endlessly.

```
Pub SendUpdates | range, theta
    HM55B.start(Enable,Clock,Data)
    XB.Delay(1000)
    repeat
      range := Ping.Millimeters(PING_Pin) ' Read range
      theta := HM55B.theta  ' Read Compass
      XB.dec(range)         ' Send range as decimal value
      XB.tx(",")            ' Send a comma to separate
      XB.dec(theta)         ' Send bearing
      XB.Delay(2000)        ' Wait 2 seconds and send again
```

Caution: The XBee only waits so long in assembling a packet for transmission. If the delay between data sent is too long (send some data, read a sensor and process new data, then send the new data), it may send it as two different frames. To increase the time it waits for more data, the RO (Packetization Timeout) value can be increased.

In the GetData method, the Propeller endlessly awaits data in one cog. Once received, if the byte is "i," it accepts the next two bytes and uses them for IO number and state to control an output pin. This is different from prior examples where we collected a decimal value.

```
   dataIn := XB.rx          ' Wait for incoming byte
   If dataIn == "i"         ' i = I/O control
      IO := XB.rx           ' Accept IO number as byte value
      value := XB.rx        ' Accept state (1/0) as byte value
      dira[IO]~~            ' Set direction of pin
      outa[IO] := value     ' Set state of pin
```

Using bytes instead, the packet is always three bytes long. To control P20 to turn on, the structure would be

| i | byte value 20 | byte value 1 |

Instead of

| i | string "20" (2 bytes or characters) | string "1" |

By using bytes as values instead of decimal string, the packet size can be compressed. For values greater than 255 (maximum byte value), two bytes can be used and combined:

```
Value = byte1 << 8 + byte2
```

. . . where byte1 (MSB) is shifted over by eight bits and then added to byte2 (LSB). We used a similar technique in the RxPacketNow methods in the XBee Object to assemble the 16-bit address from two received bytes.

In the coordinator's code, XB.AT_Config(string("ATAP 1")) shifts the XBee in API mode and the Propeller waits for an API packet to be received in ProcessFrame. If the Identifier (RxIdent) is $81, the packet is of the message variety, as opposed to a status or other type. The source address is accessed and displayed.

```
XB.API_Rx                       ' Wait for API data
if XB.RxIdent == $81            ' If data identifier is a msg string
                                ' Display source address
   PC.Str(string(13,"Data Received from address: "))
   PC.DEC(XB.srcAddr)
```

Since the actual message contained values separated by commas, the ParseDEC method is used to pull out and display the range and bearing. The signal strength, RSSI, is accessed and displayed.

```
   PC.str(string(13,"Ping Distance    : "))
   Range := XB.ParseDEC(XB.RxData,1)
   PC.DEC(Range)
   PC.str(string(13,"Compass bearing  : "))
   theta := XB.ParseDEC(XB.RxData,2)
   PC.DEC(theta)

   PC.str(string(13,"RF Signal strength: "))
   PC.DEC(-XB.rxRSSI)           ' Display RSSI level
```

The ControlPin method is used to send data back to the end device. It is passed the address to send the packet to (the source address of the incoming packet), the IO pin number, and the state (0 or 1). In order to packetize the data, a new packet is constructed and then passed to be transmitted.

```
Pub ControlPin(destAddr, pin, state)
    XB.API_NewPacket              ' Clean out packet of old data
    XB.API_AddStr(string("i"))    ' Add an i to packet
    XB.API_AddByte(pin)           ' Add a byte of pin number
    XB.API_AddByte(state)         ' Add a byte of pin state
                                  ' Send the packet
    XB.API_txPacket(destAddr,XB.API_Packet,3)
```

In ControlPin, the packet string in which the data will be sent is cleared out (API_NewPacket). All bytes in the packet are set to 0 when cleared. The string and byte values of i, pin number, and state are added to the packet (API_AddStr or AddByte). API_txPacket is used to send the data to the correct address, the pointer for the packet is given, and the number of bytes to be sent is provided.

The difference between the XBee Object's API_str and API_txPacket is that strings cannot have byte values of 0—a string ends with a byte value of 0. Our packet has a byte value of 0 for possibly either pin or state, so we needed to specify that it would be sent as a packet and then provide the number of bytes in it. Here are some examples of transmitting API data to address 5:

Sending a simple string:

```
XB.API_str(address, string)
XB.API_str(5, string("Hello!"))
```

To send a string with a value, such as "Range = range value" (in objects, declare num: "numbers"):

```
XB.API_NewPacket
XB.API_addStr(string("Range = "))
XB.API_addStr(num.ToStr(range,num#dec))
XB.API_str(5, XB.API_Packet)
```

To send just a byte in the packet:

```
XB.API_tx(5, 13)
```

Working in API mode can be intimidating, but its benefits are many. The XBee Object has multiple methods for interfacing with the XBee in both modes with example code. It would be of benefit to read through the object documentation as well as the XBee manual.

TRY THESE!

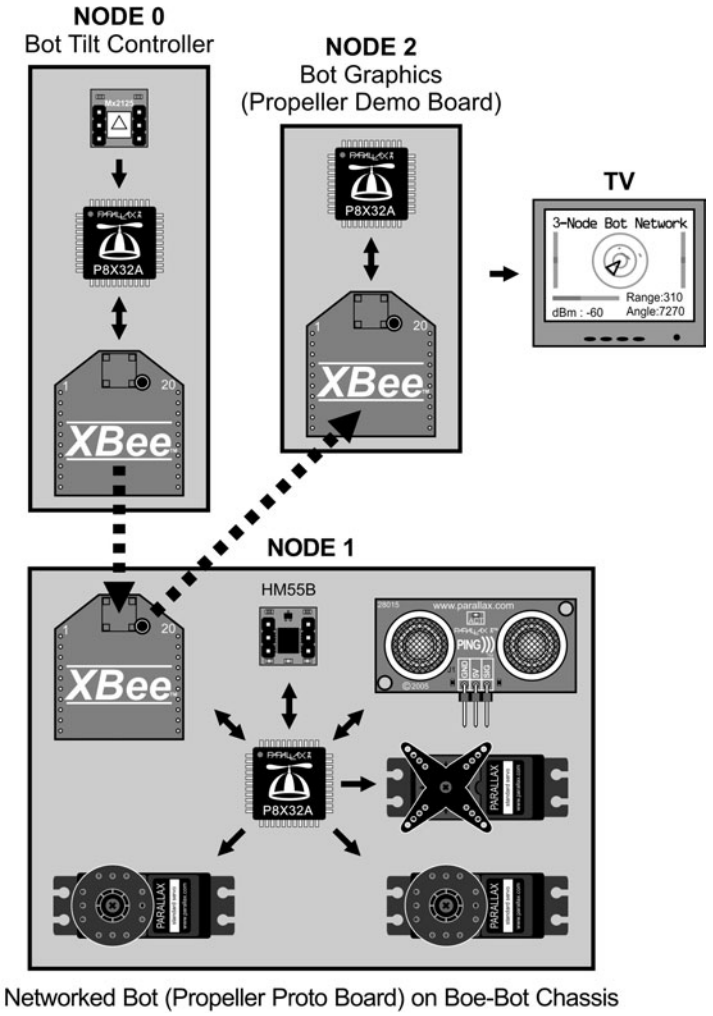
- ✓ Modify the end-device code so that it sends data only if range < 100 mm.
- ✓ Add a pushbutton to the coordinator. Have it control an LED on the end device (it's not a good idea to have two different cogs trying to send data; comment out the code to blink the LED on reception of data).

A Three-Node, Tilt-Controlled Robot with Graphical Display

OVERVIEW AND CONSTRUCTION

A three-node network for controlling and monitoring a robot will be explored for the last project in this chapter. The system shown in Fig. 5-19 has:

- A Propeller Demo Board network node (address 0) with an accelerometer to measure angle of inclination on two axes for the tilt controller



Networked Bot (Propeller Proto Board) on Boe-Bot Chassis

Figure 5-19 3-Node bot network diagram.

- A robot on the network (address 1) using a Propeller Proto Board on a Boe-Bot robot chassis with HM55B compass, PING))) range finder on a servo bracket, and LEDs
- A Propeller Demo Board on the network (address 2) driving a TV for video display of the graphical display

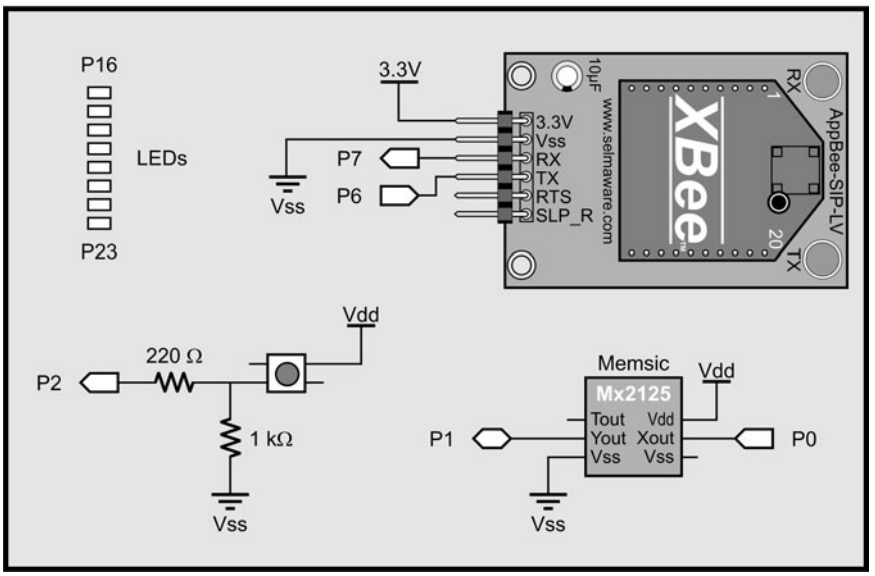
Figure 5-20 shows the wiring connection diagram for each of the nodes. Note that in switching to the Proto Board we will change the I/O pins used for the XBee.

Hardware construction tips for bot:

✓ If you are not familiar with the Boe-Bot robot, you may want to look through "Robotics with the Boe-Bot" by Andy Lindsay, available for download at www.parallax.com/education to familiarize yourself with the basic hardware and servo operation.

NODE 0

Bot Tilt Controller (Propeller Demo Board)



NODE 1

Bot (Propeller Proto Board on Boe-Bot Chassis)

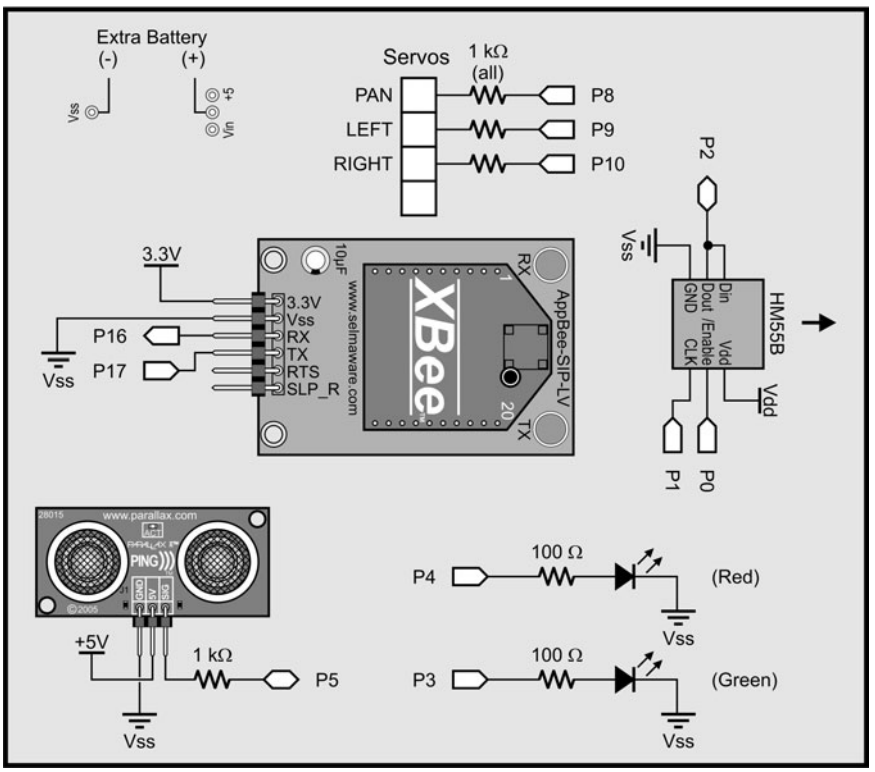


Figure 5-20 Hardware wiring diagram for bot system.

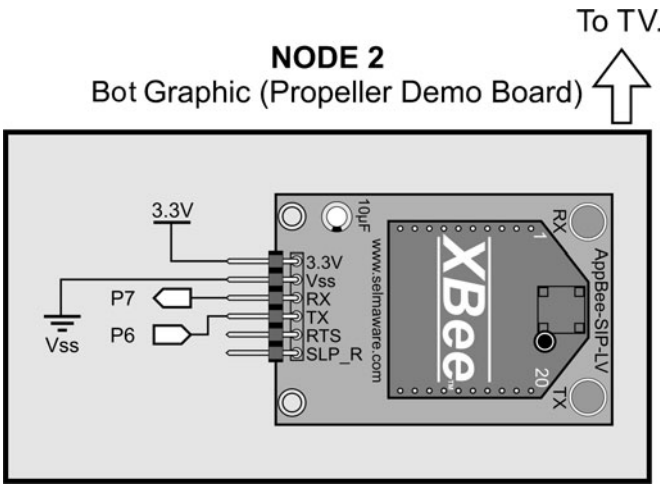


Figure 5-20 (*Continued*)

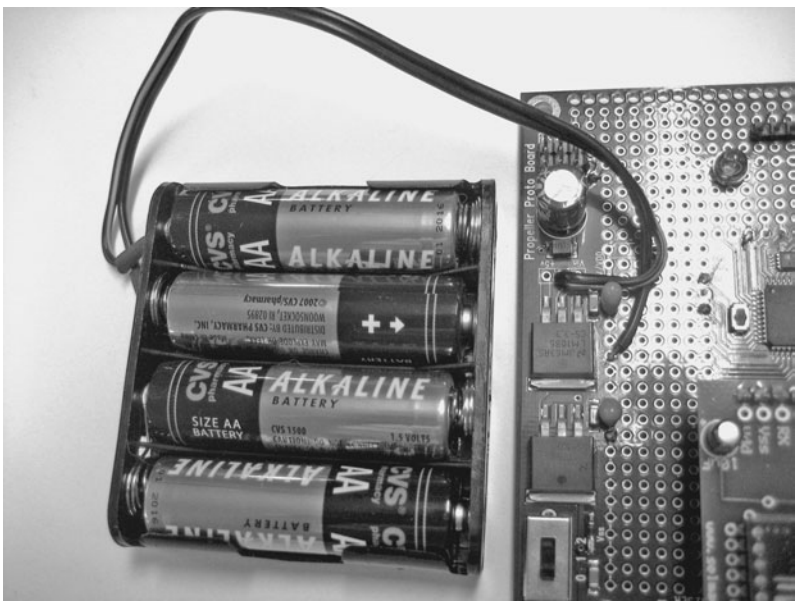


Figure 5-21 **Supplying separate power for bot's servo drive.**

✓ For the bot, two 4-AA packs were used with the spare tied under the normal battery pack. The second battery pack is used only for servo power. Attempts at using a single supply caused voltage and current spikes affecting the Propeller chip's operation. The connector of the battery pack was cut off and soldered to the servo header power and Vss (see Fig. 5-21). Other sources may be used, but supply voltage should not exceed 7.5 V or the servos can be damaged. Use coated wire to strap the batteries under the bot.

- Ensure the HM55B compass is mounted facing forward and that it is away from large current loads, such as batteries and servos. The magnetic fields will cause problems with proper compass bearing.
- The PING))) sensor is mounted using the PING))) Mounting Bracket Kit. Manually rotate the servo to find the center prior to mounting the bracket. Mount the cable header so that servo rotation does not hit it (the servo turns further manually than with the code—about 45 degrees each way).
- A battery supply was used on the tilt controller as well for unfettered operation.

OVERVIEW OF SYSTEM OPERATION

Tilt Controller: This is used to read the Memsic 2125 accelerometer module, calculate right and left motor drives based on inclination, and then send drive values to the bot. The tilt controller, shown in Fig. 5-22, also receives data from the bot and uses the PING))) range measurements to light the eight Demo Board LEDs, showing distances of 100 mm for local range indication. Pressing the pushbutton will send a panning map instruction to the bot to map the area in front of it for display by the TV graphics node. This node has a MY address of 0 and a DL address of 1 (the bot).
Bot: This controls all function of the networked bot shown in Fig. 5-23, including:

- Accepting drive values for motor drive. Should no data be received for 1.5 s, the bot will stop and blink the red LED.

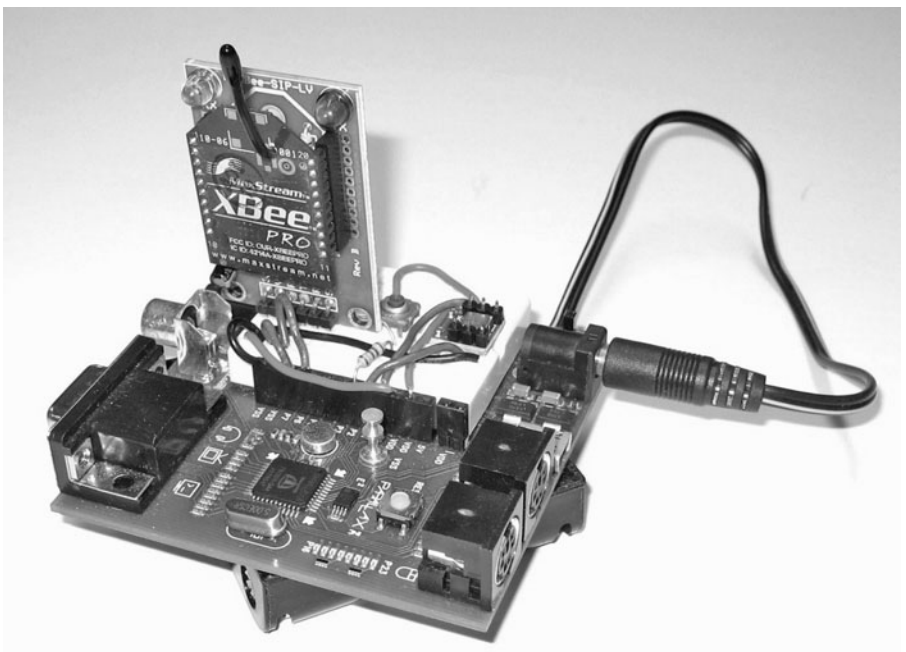


Figure 5-22 Tilt-controller board with Memsic 2125 Accelerometer.

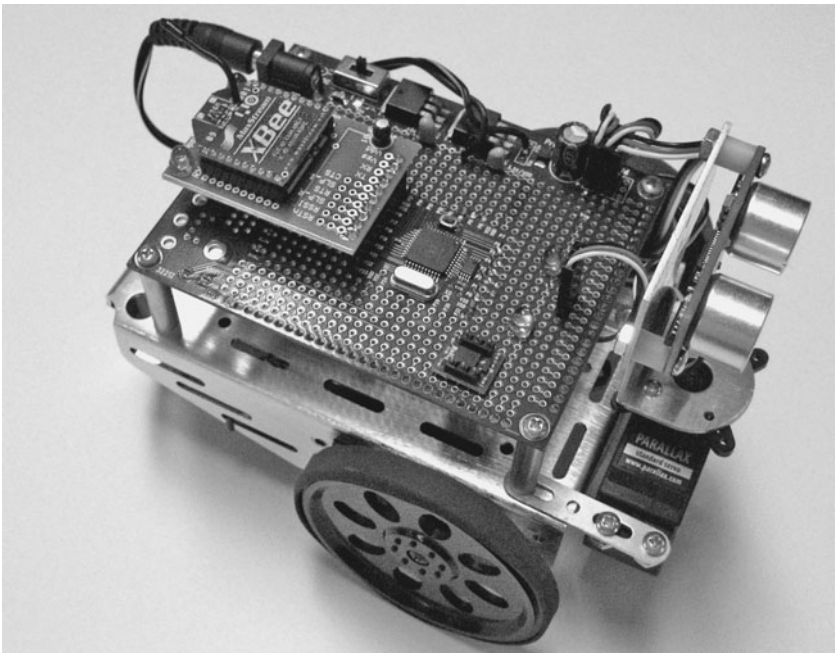


Figure 5-23 Networked bot with range finder, compass, and XBee.

- Accepting panning map instruction (p) to perform mapping operation. When this instruction is received, the bot will stop and pan the PING))) sensor from right to left, measuring distances and sending map data (m) to the TV graphics node and the tilt controller. When mapping is complete, the bot will remain steady and blink the green LED until the tilt controller releases it from mapping mode (user presses button again). The bot will send a clear (c) code to the video to clear the display when map mode exits.
- While driving, the bot will transmit updates (u) of right and left drive values, PING))) range, and direction of travel from the HM55B compass (0-8191).
- This node has a MY address of 1 and sends data to $ffff—all nodes on the network for a broadcast.

Bot Graphics: Shown in Fig. 5-24, drives the graphics TV display showing:

- The bot bearing as a rotating triangle and text
- Distance to object as a red point in front of the bot and text
- Yellow range marker circles at 0.25 m, 0.5 m, and 1 m
- Left and right drives as bar indicators
- Signal strength for RSSI (dBm) as bar and text

The update packets contain information for much of the display, and the RSSI is pulled from the received frame through API mode. When the bot is in mapping mode, mapping

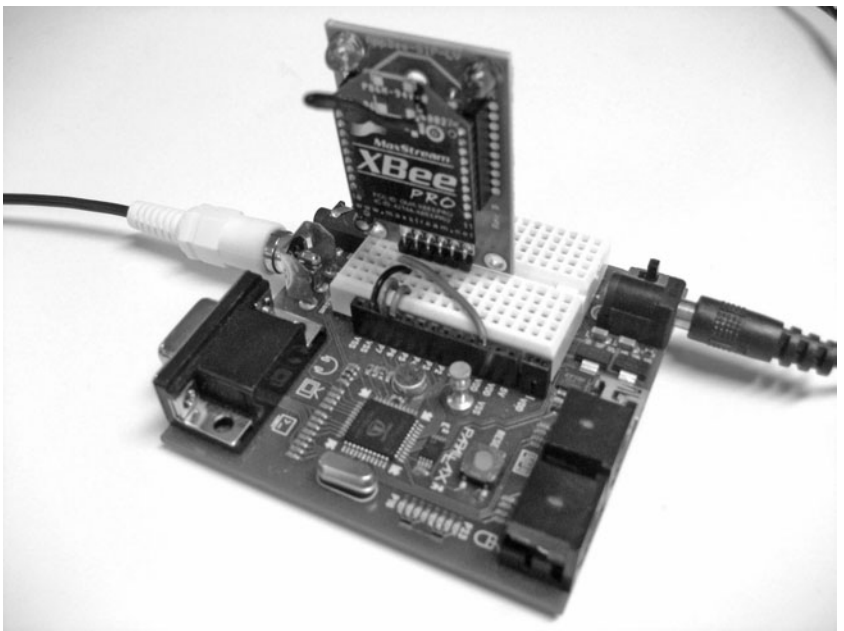


Figure 5-24 Bot TV graphics controller.

packets contain data to plot the range map. It also accepts clear codes from the bot to clear the mapped display. This node has a MY address of 2 and sends out no data.

The output display of the graphics controller is shown in Fig. 5-25 in both normal driving and with the bot performing a panning map operation.

Note: The PING))) range finder has a wide angle of emission and reception. Do not expect pinpoint accuracy when mapping.

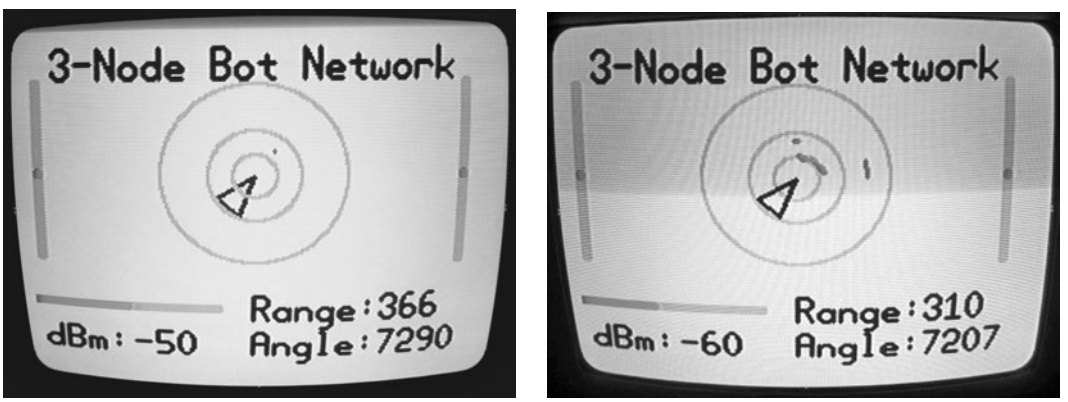


Figure 5-25 TV displays for normal and panning map data.

BOT NETWORK CODE

Bot Tilt Controller For the tilt controller, in the SendControl method, if the button is pressed, a series of p's is transmitted. With the amount of data flying, some missed bytes on reception are normal, and this helps ensure the bot gets a p-instruction for a panning map operation. If not pressed, the accelerometer is read for the X- and Y-axis (–90 to 90 degrees, 0 level) and the drive for each servo is calculated by mixing the two axes of tilt for a final servo value of 1000 to 2000 (the range of servo control) for each. The data is sent as a "d" packet for drive.

```
Forward := (accel.x*90-offset)/scale * -1

' Read and calculate -90 to 90 degree for turn
Turn := (accel.y*90-offset)/scale

' Scale and mix channels for drive, 1500 = stopped
Left_Dr := 1500 + Forward * 3 + Turn * 3
Right_Dr := 1500 - Forward * 3 + Turn * 3
```

In the AcceptData method (which is running in a separate cog), incoming packets are analyzed if update data (u) or map data (m) and the local LEDs are updated. Based on the range, eight 1s are shifted to the left eight positions, then shifted right again based on the range/100. This allows one LED to light for every 100 mm or 0.1 m, out to 800 mm or 0.8 m.

```
outa[16..23] := %11111111 << 8  >> (8 - range/100)
```

Bot Code For the bot controller, in Start, received bytes are analyzed with a timeout. If any data is not received for 1500 ms, the red LED will begin to blink. Received data is analyzed for either "d" for drive data or "p" to begin a mapping scan.

```
case DataIn                          ' Test accepted data

     "d":                              ' If drive data
          Right_dr    := XB.RxDEC      ' Get right and left drive
          Left_dr     := XB.RxDEC
          SERVO.Set(Right, Right_dr) ' Drive servos based on data
          SERVO.Set(Left, Left_Dr)

     "p":                              ' p = pan and map command
          mapping := true              ' Set flag for mapping
          outa[grnLED]~~               ' Turn on green LED
          Map                          ' Go map
```

The SendUpdate method is run in a separate cog to continually send out the status of the range, direction, and drive values led by "u." The value of theta is subtracted from

8191 to allow the direction of rotation to be correct in the graphics display. If a panning map is in progress, updates are suspended due to mapping being true.

```
Repeat
  if mapping == false                ' If not mapping
    XB.Delay(250)
    Range := Ping.Millimeters(PING_Pin)   ' Read range
    theta := HM55B.theta                  ' Read Compass
    XB.TX("u")                            ' Send "update" command
    XB.DEC(Range)              ' Send range as decimal string
    XB.CR
    XB.DEC(8191-theta)         ' Send theta of bearing (0-8191)
    XB.CR
    xb.DEC(Right_Dr)           ' Send right drive
    XB.CR
    XB.DEC(Left_Dr)            ' Send left drive
    XB.CR
```

When mapping, the value of `pan` is looped from 1000 to 2000, the range of allowable servo values. The range is measured, and the PanOffset is calculated. The value of the "`pan`" has 1500 subtracted (recall that 1500 is a centered servo). The result is multiplied by 2047 (90 degrees, with 8191 being a full 360 degrees) and divided by the full range of `pan`. Finally, an "m" is sent followed by range and angle of the servo plus the `pan` offset. This repeats for each value of `pan`, from 1000 to 2000, in increments of 15 steps or 1.35 degrees (15 · 90 degrees/1000 steps = 1.35 degrees). Once mapping is complete, the system will wait until another "p" is received to exit pan mapping mode while sending a "c" to clear the video display. The variable "mapping" is used as a flag to prevent the SendUpdates code running in a separate cog from sending updates while mapping.

```
Pub Map | panValue
 '' Method turns servo from -45 to + 45 degrees from center in increments
 '' and gets ping range and returns m value at each increment

    SERVO.Set(Right, 1500)       ' Stop servos
    SERVO.Set(Left, 1500)

    SERVO.Set(Pan, 1000)         ' Pan full right
    XB.Delay(1000)
                                 ' Pan right to left
    repeat pan from 1000 to 2000 step 15
      SERVO.Set(Pan,panValue)
      Range := Ping.Millimeters(PING_Pin)    ' Get range calculated
                                             ' based on compass
                                             ' and pan
      PanOffset := ((panValue-1500) * 2047/1000)
```

```
    XB.TX("m")                                  ' Send map data command
    XB.DEC(Range)                               ' Send range as decimal
    XB.CR
    XB.DEC((8191-Theta) + PanOffset)            ' Send theta of bearing
    XB.CR
    XB.delay(50)
  XB.delay(1000)

  SERVO.SET(Pan,1500)                           ' Re-center pan servo
```

TRY IT!

✓ Add another device, such as speaker, to your bot. Add a button on the tilt controller and modify code to control the device from the tilt controller.

Bot Graphics The bot graphics code is responsible for accepting the data and displaying it graphically on a TV screen. Note that this XBee is in API mode so that the RSSI level may be pulled out of the received frame. The code looks for one of three incoming byte instructions: "u," "m," and "c." Updates, "u," are messages with update data as the data moves, with range, bearing, and drive values (limited between 1000 and 2000), and it retrieves RSSI level for display creation.

```
Repeat
  XB.API_rx                             ' Accept data
  If XB.RxIdent == $81                  ' If msg packet...
    if byte[XB.RxData] == "u"           ' If updates, pull out data
        ' Get DEC data skipping 1st byte (u)
      range   := XB.ParseDEC(XB.RxData+1,1)
      bearing := XB.ParseDEC(XB.RxData+1,2)
      rDrive  := XB.ParseDEC(XB.RxData+1,3) <#2000 #>1000
      lDrive  := XB.ParseDEC(XB.RxData+1,4) <#2000 #>1000
      RSSI    := XB.RxRSSI
      Update
```

Mapping (m) strings are used to map what the bot "sees" without clearing off old data while a mapping pan is in progress. Clear, "c," is received once the bot switches back into drive mode after mapping.

We aren't going to delve too deeply into the graphics creation here, as it's not a major subject for this chapter. One point of interest is in that many graphic programs the video data is written into one part of memory (such as `bitmap_base`), and when the complete display change is ready, it is copied into the section of memory that the graphics driver uses to display the actual display. It is effectively double-buffered to prevent flicker on the screen. We do not have the luxury of the memory needed for that operation. Instead, to reduce flicker, values of the old data are saved. When updating, the graphics are redrawn in the background

color to "erase" them, then the new data is used to draw the graphics in the correct color, such as in this code:

```
' Draw bot vector image
gr.width(2)
gr.color(0)                                    ' White
gr.vec(120,120, 100, (bearing_1), @bot)        ' Erase last image
gr.color(1)                                    ' Black
gr.vec(120,120, 100, (bearing), @bot)          ' Draw new image
```

Many features of graphics.spin are used, including text, lines, arcs, and vector-based graphics. The code is fairly well commented for adaptation.

TRY IT!

✓ Add another sensor to your bot. Modify both the bot and graphics code to send and display the value.

Summary

In this chapter we looked at what the XBee is and how it can be configured and used in a wireless sensor network. Using AT codes sent from the controller, the XBee can be configured for specific applications, such as unique addresses used in polling operations. In API mode, frames are sent and received with specific data. Using the networking capabilities, a three-node bot system was developed for control and monitoring using the graphics ability of the Propeller chip.

The ability to configure the device and send data between Propeller chips efficiently and with addressing leads to a wide array of projects that can be implemented. Allowing different Propeller chips to perform their own processing and easily communicating with each other brings the excitement of parallel processing to a whole new level.

In my research with institutions, such as Southern Illinois University, University of Florida, USDA in Texas, and the University of Sassari, Italy, I have been involved in many XBee/Propeller (and some other controller) projects. These projects include monitoring corn irrigation needs, biological monitoring, and monitoring the vibration of citrus fruit as it's shaken from the tree.

Wireless sensor networks are a powerful and quickly expanding field for remote monitoring and control. They are finding use in research and in building, plant, and home automation. The Propeller, with its ability to perform parallel processing, is an outstanding choice for monitoring and control. As mentioned at the outset of this chapter, whether you build the projects in this chapter or simply gain an understanding of the material, I hope you can use the base code and principles in projects of your own invention.

Exercise

A FINAL PROJECT FOR YOU—DIRECT XBee ADC/DIGITAL DATA

For our final exploration into the Propeller/XBee combination, let's exercise the XBee's ability to measure and transmit analog and digital data without a controller. The received data has a packet identifier of $83 (the XBee needs to have firmware version 10A3 or higher to be able to this).

- ✓ Apply an analog voltage of up to 3.3 V (using a potentiometer or other device) to ADC 0 (pin 20) and ADC 2 (pin 19) and a pushbutton to DIO2 (pin 18).
- ✓ Using X-CTU software, starting from the default settings, configure for a MY of 6 and for I/O settings, such as:
  D0 = mode 2: ADC
  D1 = mode 2: ADC
  D3 = mode 3: DIN (Digital Input)
  IR = 3E8 (sample rate of 1 second. $3E8 = 1000 decimal or 1000 ms of time).
- ✓ Connect the Vcc (pin 1) and the Vref pin (pin 14) of the XBee to 3.3 V. Connect Vss to GND (pin 10). Do not connect anything else, including the Propeller.
- ✓ Download ADC-Dig Output Sample.spin to your coordinator board.
- ✓ Open the Terminal window and monitor. You should see something similar to Fig. 5-26.

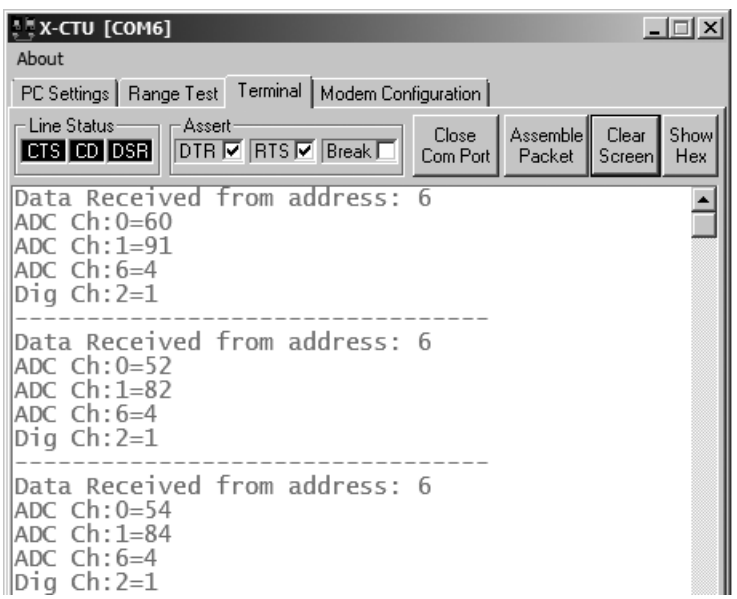


Figure 5-26 Displaying data from an XBee sending raw ADC/digital data.

The remote XBee is reading the ADC and digital channels specified and sending a packet containing the data. You will not see the LED blink on the sending XBee because no communication enters or exits it through the serial port.

On the coordinator, when a frame with an identifier of $83 (ADC/Digital data) arrives, valid data is pulled out and displayed (nonenabled channels are −1).

```
PUB Start | channel
  ' Configure XBee & PC Comms
  XB.start(XB_Rx, XB_Tx, 0,9600)
  PC.start(PC_Rx, PC_Tx, 0, 9600)

  XB.AT_Init                      ' Fast config
  XB.AT_ConfigVal(string("ATMY"), MY_Addr)
  XB.AT_Config(string("ATAP 1"))' Switch to API mode

  PC.str(string("Coordinator in API mode ready at address:"))
  PC.dec(MY_Addr)
  PC.Tx(13)

  Repeat
    XB.API_Rx                  ' Wait for API data
    if XB.RxIdent == $83       ' If data identifier is a ADC/Dig data
       PC.Str(string(13,"Data Received from address: "))
       PC.DEC(XB.srcAddr)
       repeat channel from 0 to 6        ' Cycle through ADC channels
         if XB.rxADC(Channel) <> -1      ' Display if not -1
           PC.str(string(13,"ADC Ch:"))
           PC.dec(channel)
           PC.tx("=")
           PC.DEC(XB.rxADC(Channel))

       repeat channel from 0 to 7      ' Cycle through Digital channels
         if XB.rxBit(Channel) <> -1    ' Display if not -1
           PC.str(string(13,"Dig Ch:"))
           PC.dec(channel)
           PC.tx("=")
           PC.DEC(XB.rxBit(Channel))

       PC.str(string(13,"---------------------------------"))
```

If you were to use an analog accelerometer, you could read the accelerometer and a digital pushbutton on the tilt controller and have the XBee send those values automatically. Then the Propeller could accept data at the bot and process it for control action. Or you may have an array of sensors in the area and collect data from them as they wake, sample, send, and go back to sleep.