PRODUCT MANUAL

Dynamic C®

An Introduction to ZigBee®

019-0162 • 080924-C

The latest revision of this manual is available on the Rabbit Web site, www.rabbit.com, for free, unregistered download.

An Introduction to ZigBee®

Part Number 019-0162–C • 080924 • Printed in U.S.A.

Trademarks

PRODUCT MANUAL

Table of Contents

1. Introduction

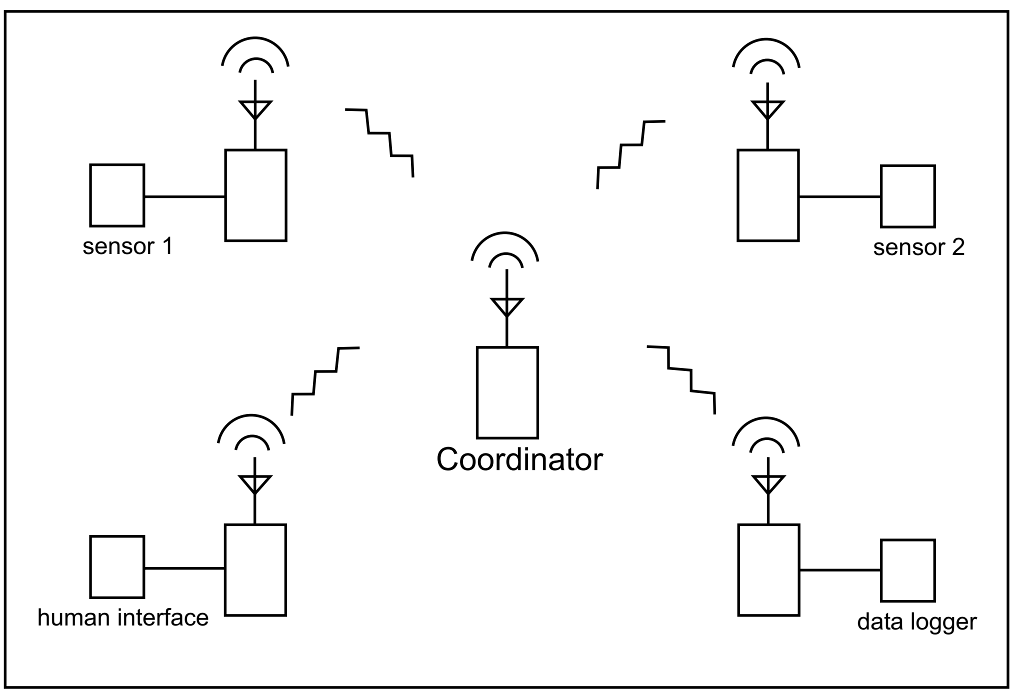
This manual provides an introduction to the various components of a ZigBee network. After a quick overview of ZigBee, we start with a description of high-level concepts used in wireless communication and move on to the specific protocols needed to implement the communication standards. This is followed by a description of using a Rabbit-based board and Dynamic C libraries to form a ZigBee network.

ZigBee, a specification for communication in a wireless personal area network (WPAN), has been called the "Internet of things." Theoretically, your ZigBee-enabled coffee maker can communicate with your ZigBee-enabled toaster. The benefits of this technology go far beyond the novelty of kitchen appliances coordinating your breakfast. ZigBee applications include:

- Home and office automation
- Industrial automation
- Medical monitoring
- Low-power sensors
- HVAC control
- Plus many other control and monitoring uses

ZigBee targets the application domain of low power, low duty cycle and low data rate requirement devices. Figure 1.1 shows a block diagram of a ZigBee network with five nodes.

Figure 1.1ZigBee Network



Before going further, note that there is a list of glossary terms in Appendix A.

This page intentionally left almost blank.

2. Wireless Communication

This chapter presents a select high-level overview of wireless communication.

2.1 Communication Systems

All wireless communication systems have the following components:

- Transmitter
- Receiver
- Antennas
- Path between the transmitter and the receiver

In short, the transmitter feeds a signal of encoded data modulated into RF waves into the antenna. The antenna radiates the signal through the air where it is picked up by the antenna of the receiver. The receiver demodulates the RF waves back into the encoded data stream sent by the transmitter.

2.2 Wireless Network Types

There are a number of different types of networks used in wireless communication. Network types are typically defined by size and location.

2.2.1 WPAN

A wireless personal area network (WPAN) is meant to span a small area such as a private home or an individual workspace. It is used to communicate over a relatively short distance. The specification does not preclude longer ranges being achieved with the trade-off of a lower data rate.

In contrast to other network types, there is little to no need for infrastructure with a WPAN.

Ad-hoc networking is one of the key concepts in WPANs. This allows devices to be part of the network temporarily; they can join and leave at will. This works well for mobile devices like PDAs, laptops and phones.

Some of the protocols employing WPAN include Bluetooth, ZigBee, Ultra-wideband (UWB) and IrDA. Each of these is optimized for particular applications or domains. ZigBee, with its sleepy, battery-powered end devices, is a perfect fit for wireless sensors. Typical ZigBee application domains include: agricultural, building and industrial automation, home control, medical monitoring, security and, lest we take ourselves too seriously, toys, toys and more toys.

2.2.2 WLAN

Wireless local area networks (WLANs) are meant to span a relatively small area, e.g., a house, a building, or a college campus. WLANs are becoming more prevalent as costs come down and standards improve.

A WLAN can be an extension of a wired local area network (LAN), its access point connected to a LAN technology such as Ethernet. A popular protocol for WLAN is 802.11, also known as Wi-Fi.

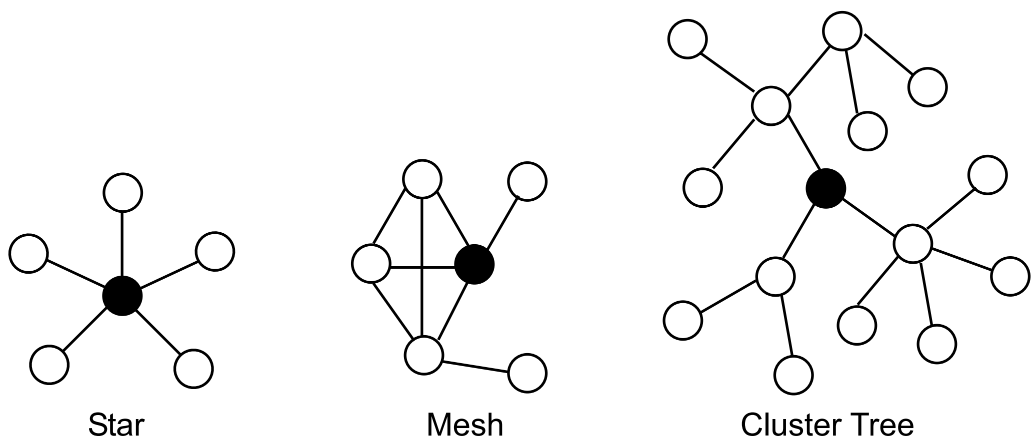
2.2.3 WWAN

A wireless wide area network (WAN) is meant to span a large area, such as a city, state or country. It makes use of telephone lines and satellite dishes as well as radio waves to transfer data. A good description of WWANs is found at: `http://en.wikipedia.org/wiki/WWAN`.

2.3 Wireless Network Topologies

This section discusses the network topologies supported by the IEEE 802.15.4 and ZigBee specifications. The topology of a network describes how the nodes are connected, either physically or logically. The physical topology is a geometrical shape resulting from the physical links from node to node, as shown in Figure 2.1. The logical topology maps the flow of data between the nodes.

Figure 2.1 Physical Network Topologies Supported by ZigBee



IEEE 802.15.4 supports star and peer-to-peer topologies. The ZigBee specification supports star and two kinds of peer-to-peer topologies, mesh and cluster tree.

ZigBee-compliant devices are sometimes specified as supporting point-to-point and point-to-multipoint topologies.

2.4 Wireless Standards

The demand for wireless solutions continues to grow and with it new standards have come forward and other existing standards have strengthened their position in the marketplace. This section compares three popular wireless standards being used today and lists some of the design considerations that differentiate them.

Table 2-1 Comparison of Wireless Standards

Wireless Parameter	Bluetooth	Wi-Fi	ZigBee
Frequency band	2.4 GHz	2.4 GHz	2.4 GHz
Physical/MAC layers	IEEE 802.15.1	IEEE 802.11b	IEEE 802.15.4
Range	9 m	75 to 90 m	Indoors: up to 30 m Outdoors (line of sight): up to 100 m
Current consumption	60 mA (Tx mode)	400 mA (Tx mode) 20 mA (Standby mode)	25-35 mA (Tx mode) 3 µA (Standby mode)
Raw data rate	1 Mbps	11 Mbps	250 Kbps
Protocol stack size	250 KB	1 MB	32 KB 4 KB (for limited function end devices)
Typical network join time	>3 sec	variable, 1 sec typically	30 ms typically
Interference avoidance method	FHSS (frequency-hopping spread spectrum)	DSSS (direct-sequence spread spectrum)	DSSS (direct-sequence spread spectrum)
Minimum quiet bandwidth required	15 MHz (dynamic)	22 MHz (static)	3 MHz (static)
Maximum number of nodes per network	7	32 per access point	64 K
Number of channels	19	13	16

Each wireless standard addresses the needs of a different market segment. Choosing the best-fit wireless standard is a crucial step in the successful deployment of any wireless application. The requirements of your application will determine the wireless standard to choose.

For more information on design considerations, see Technical Note 249, “Designing with Wireless Rabbits.”

2.5 Security in a Wireless Network

This section discusses the added security issues introduced by wireless networks. The salient fact that signals are traveling through the air means that the communication is less secure than if they were traveling through wires. Someone seeking access to your network need not overcome the obstacle of tapping into physical wires. Anyone in range of the transmission can potentially listen on the channel.

Wireless or not, a network needs a security plan. The first thing to do is to decide what level of security is appropriate for the applications running on your network. For instance, a financial institution, such as a bank or credit union offering online account access would have substantially different security concerns than would a business owner offering free Internet access at a coffee shop.

2.5.1 Security Risks

After you have decided the level of security you need for your network, assess the potential security risks that exist.

- Who is in range of the wireless transmissions?
- Can unauthorized users join the network?
- What would an unauthorized user be able to do if they did join?
- Is sensitive data traveling over the wireless channel?

Network security is analogous to home security: You do not want your house to be a target so you do things to minimize your risk, whether that be outside lighting, motion sensors, or even just keeping bushes pruned back close to the house so bad guys have fewer hiding places.

Deterrence is the goal because nothing is guaranteed to be 100% safe in the real world.

3. IEEE 802.15.4 Specification

This chapter is an overview of the IEEE 802.15.4 specification. 802.15.4 defines a standard for a low-rate WPAN (LR-WPAN).

3.1 Scope of 802.15.4

802.15.4 is a packet-based radio protocol. It addresses the communication needs of wireless applications that have low data rates and low power consumption requirements. It is the foundation on which ZigBee is built. Figure 4.1 shows a simplified ZigBee stack, which includes the two layers specified by 802.15.4: the physical (PHY) and MAC layers.

3.1.1 PHY Layers

The PHY layer defines the physical and electrical characteristics of the network. The basic task of the PHY layer is data transmission and reception. At the physical/electrical level, this involves modulation and spreading techniques that map bits of information in such a way as to allow them to travel through the air. Specifications for receiver sensitivity and transmit output power are in the PHY layer.

The PHY layer is also responsible for the following tasks:

- enable/disable the radio transceiver
- link quality indication (LQI) for received packets
- energy detection (ED) within the current channel
- clear channel assessment (CCA)

3.1.2 MAC Layer

The MAC layer defines how multiple 802.15.4 radios operating in the same area will share the airwaves. This includes coordinating transceiver access to the shared radio link and the scheduling and routing of data frames.

There are network association and disassociation functions embedded in the MAC layer. These functions support the self-configuration and peer-to-peer communication features of a ZigBee network.

The MAC layer is responsible for the following tasks:

- beacon generation if device is a coordinator
- implementing carrier sense multiple access with collision avoidance (CSMA-CA)
- handling guaranteed time slot (GTS) mechanism
- data transfer services for upper layers

3.2 Properties of 802.15.4

802.15.4 defines operation in three license-free industrial scientific medical (ISM) frequency bands. Below is a table that summarizes the properties of IEEE 802.15.4 in two of the ISM frequency bands: 915 MHz and 2.4 GHz.

Table 3-1. Comparison of IEEE 802.15.4 Frequency Bands

Property Description	Prescribed Values	
	915 MHz	**2.4 GHz**
Raw data bit rate	40 kbps	250 kbps
Transmitter output power	1 mW = 0 dBm	
Receiver sensitivity (<1% packet error rate)	-92 dBm	-85 dBm
Transmission range	Indoors: up to 30 m; Outdoors: up to 100 m	
Latency	15 ms	
Channels	10 channels	16 channels
Channel numbering	1 to 10	11 to 26
Channel access	CSMA-CA and slotted CSMA-CA	
Modulation scheme	BPSK	O-QPSK

3.2.1 Transmitter and Receiver

The power output of the transmitter and the sensitivity of the receiver are determining factors of the signal strength and its range. Other factors include any obstacles in the communication path that cause interference with the signal.

The higher the transmitter's output power, the longer the range of its signal. On the other side, the receiver's sensitivity determines the minimum power needed for the radio to reliably receive the signal. These values are described using dBm (deciBels below 1 milliwatt), a relative measurement that compares two signals with 1 milliwatt used as the reference signal. A large negative dBm number means higher receiver sensitivity.

3.2.2 Channels

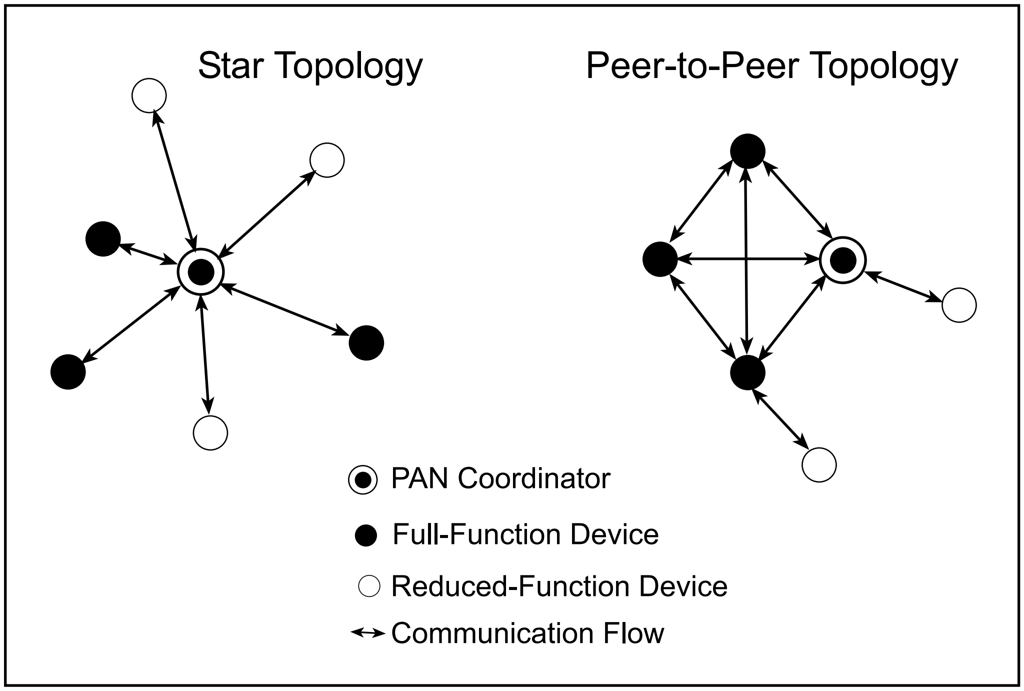
Of the three ISM frequency bands only the 2.4 GHz band operates world-wide. The 868 MHz band only operates in the EU and the 915 MHz band is only for North and South America. However, if global interoperability is not a requirement, the relative emptiness of the 915 MHz band in non-European countries might be an advantage for some applications.

For the 2.4 GHz band, 802.15.4 specifies communication should occur in 5 MHz channels ranging from 2.405 to 2.480 GHz.

3.3 Network Topologies

According to the IEEE 802.15.4 specification, the LR-WPAN may operate in one of two network topologies: star or peer-to-peer. 802.15.4 is designed for networks with low data rates, which is why the acronym "LR" (for "low rate") is prepended to "WPAN."

Figure 3.1 Network Topologies Supported by IEEE 802.15.4



As shown in Figure 3.1, the star topology has a central node with all other nodes communicating only with the central one. The peer-to-peer topology allows peers to communicate directly with one another. This feature is essential in supporting mesh networks.

3.4 Network Devices and their Operating Modes

Two types of devices can participate in a LR-WPAN: a full function device (FFD) and a reduced function device (RFD).

An RFD does not have routing capabilities. RFDs can be configured as end nodes only. They communicate with their parent, which is the node that allowed the RFD to join the network.

An FFD has routing capabilities and can be configured as the PAN coordinator. In a star network all nodes communicate with the PAN coordinator only so it does not matter if they are FFDs or RFDs. In a peer-to-peer network there is also one PAN coordinator, but there are other FFDs which can communicate with not only the PAN coordinator, but also with other FFDs and RFDs.

There are three operating modes supported by IEEE 802.15.4: PAN coordinator, coordinator, and end device. FFDs can be configured for any of the operating modes. In ZigBee terminology the PAN coordinator is referred to as simply "coordinator." The IEEE term "coordinator" is the ZigBee term for "router."

3.5 Addressing Modes Supported by 802.15.4

802.15.4 supports both short (16-bit) and extended (64-bit) addressing.

An extended address (also called EUI-64) is assigned to every RF module that complies to the 802.15.4 specification.

When a device associates with a WPAN it can receive a 16-bit address from its parent node that is unique in that network.

3.5.1 PAN ID

Each WPAN has a 16-bit number that is used as a network identifier. It is called the PAN ID. The PAN coordinator assigns the PAN ID when it creates the network. A device can try and join any network or it can limit itself to a network with a particular PAN ID.

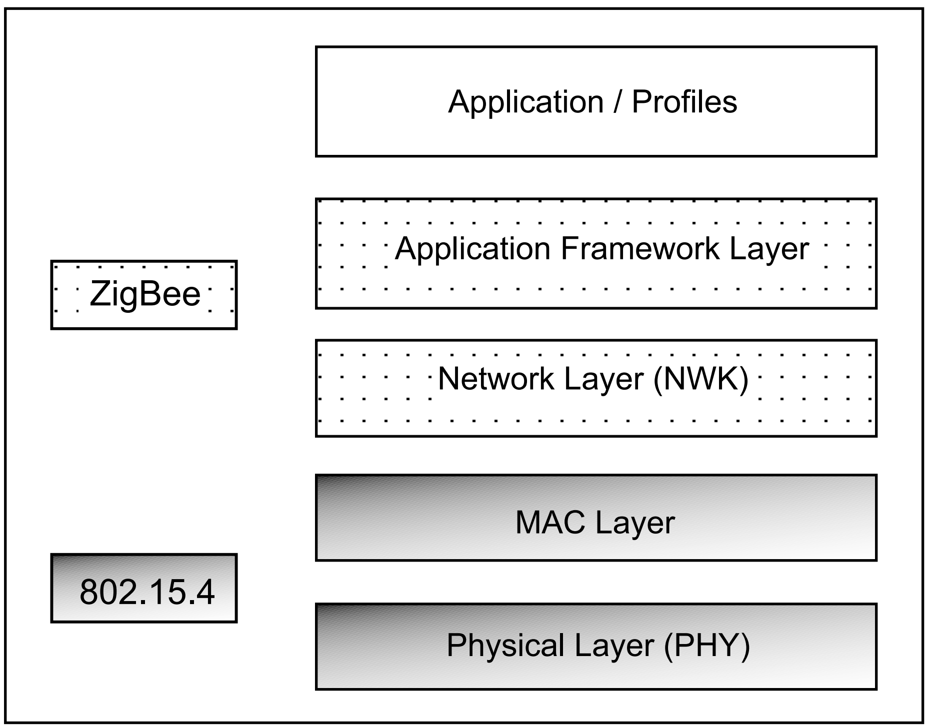
ZigBee PRO defines an extended PAN ID. It is a 64-bit number that is used as a network identifier in place of its 16-bit predecessor.

4. ZigBee Specification

This chapter gives an overview of the ZigBee specification. ZigBee, its specification and promotion, is a product of the ZigBee Alliance. The Alliance is an association of companies working together to ensure the success of this open global standard.

ZigBee is built on top of the IEEE 802.15.4 standard. ZigBee provides routing and multi-hop functions to the packet-based radio protocol.

Figure 4.1 ZigBee Stack



4.1 Logical Device Types

The ZigBee stack resides on a ZigBee logical device. There are three logical device types:

- coordinator
- router
- end device

It is at the network layer that the differences in functionality among the devices are determined. See Table 4-1 for more information. It is expected that in a ZigBee network the coordinator and the routers will be mains-powered and that the end devices can be battery-powered.

In a ZigBee network there is one and only one coordinator per network. The number of routers and/or end devices depends on the application requirements and the conditions of the physical site.

Within networks that support sleeping end devices, the coordinator or one of the routers must be designated as a Primary Discovery Cache Device. These cache devices provide server services to upload and store discovery information, as well as respond to discovery requests, on behalf of the sleeping end devices.

4.2 ZigBee Stack Layers

As shown in Figure 4.1, the stack layers defined by the ZigBee specification are the network and application framework layers. The ZigBee stack is loosely based on the OSI 7-layer model. It implements only the functionality that is required in the intended markets.

4.2.1 Network (NWK) Layer

The network layer ensures the proper operation of the underlying MAC layer and provides an interface to the application layer. The network layer supports star, tree and mesh topologies. Among other things, this is the layer where networks are started, joined, left and discovered.

Table 4-1. Comparison of ZigBee Devices at the Network Layer

ZigBee Network Layer Function	Coordinator	Router	End Device
Establish a ZigBee network	•		
Permit other devices to join or leave the network	•	•	
Assign 16-bit network addresses	•	•	
Discover and record paths for efficient message delivery	•	•	
Discover and record list of one-hop neighbors	•	•	
Route network packets	•	•	
Receive or send network packets	•	•	•
Join or leave the network	•	•	•
Enter sleep mode			•

When a coordinator attempts to establish a ZigBee network, it does an energy scan to find the best RF channel for its new network. When a channel has been chosen, the coordinator assigns the logical network identifier, also known as the PAN ID, which will be applied to all devices that join the network.

A node can join the network either directly or through association. To join directly, the system designer must somehow add a node's extended address into the neighbor table of a device. The direct joining device will issue an orphan scan, and the node with the matching extended address (in its neighbor table) will respond, allowing the device to join.

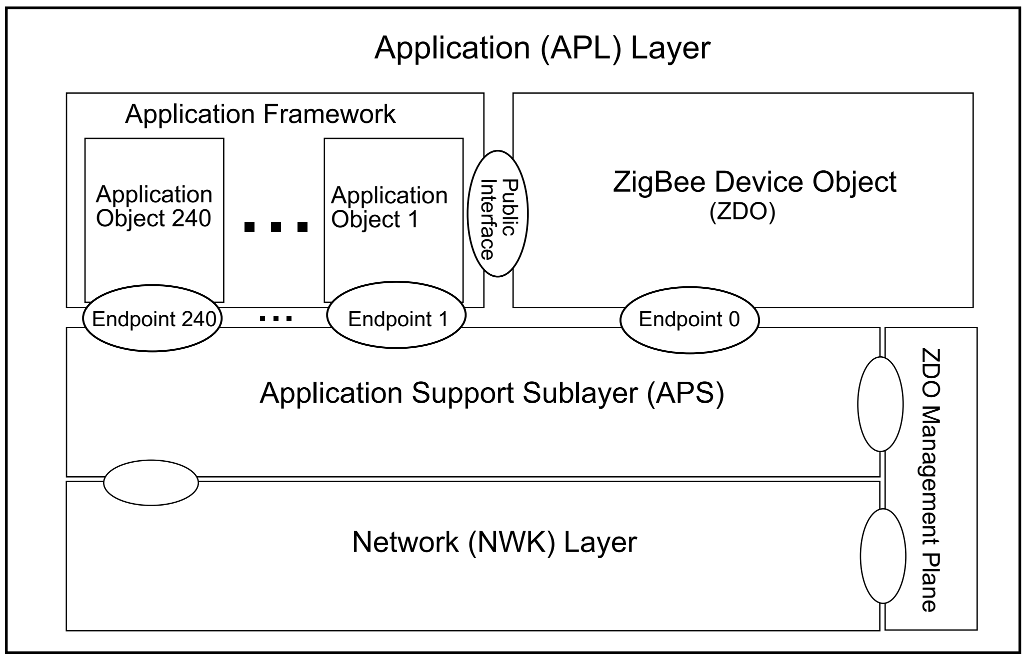
To join by association, a node sends out a beacon request on a channel, repeating the beacon request on other channels until it finds an acceptable network to join.

The network layer provides security for the network, ensuring both authenticity and confidentiality of a transmission.

4.2.2 Application (APL) Layer

The APL layer is made up of several sublayers. The components of the APL layer are shown in Figure 4.2. and discussed below. The ovals symbolize the interface, called service access points (SAP), between different sublayer entities.

Figure 4.2 ZigBee-Defined Part of Stack



4.2.2.1 Application Support Sublayer (APS)

The APS sublayer is responsible for:

- binding tables
- message forwarding between bound devices
- group address definition and management
- address mapping from 64-bit extended addresses to 16-bit NWK addresses
- fragmentation and reassembly of packets
- reliable data transport

The key to interfacing devices at the need/service level is the concept of binding. Binding tables are kept by the coordinator and all routers in the network. The binding table maps a source address and source endpoint to one or more destination addresses and endpoints. The cluster ID for a bound set of devices will be the same.

As an example, consider the common control problem of maintaining a certain temperature range. A device with temperature-sensing circuitry can advertise its service of providing the temperature as a

READ_TEMPERATURE cluster ID. A controller (for a furnace or a fan, perhaps) could discover the temperature sensor device. The binding table would identify the endpoint on the temp sensor that accepts the READ_TEMPERATURE cluster ID, for example. One temperature sensor manufacturer might have endpoint 0x11 support this cluster ID, while another manufacturer might use endpoint 0x72 to support this cluster ID. The controller would have to discover both devices and would then create two binding table entries, one for each device. When the controller wants to read the temperature of all sensors, the binding table tells it which address and endpoint the READ_TEMPERATURE packet should be sent to.

4.2.2.2 Application Framework

The application framework is an execution environment for application objects to send and receive data. Application objects are defined by the manufacturer of the ZigBee-enabled device. As defined by ZigBee, an application object is at the top of the application layer and is determined by the device manufacturer. An application object actually implements the application; it can be a light bulb, a light switch, an LED, an I/O line, etc. The application profile (discussed in Section 4.4) is run by the application objects.

Each application object is addressed through its corresponding endpoint. Endpoint numbers range from 1 to 240. Endpoint 0 is the address of the ZigBee Device Object (ZDO). Endpoint 255 is the broadcast address, i.e., message are sent to all of the endpoints on a particular node. Endpoints 241 through 254 are reserved for future use.

ZigBee defines function primitives, not an application programming interface (API).

4.2.2.3 ZigBee Device Object (ZDO)

The ZDO is responsible for overall device management, specifically it is responsible for:

- initializing the APS sublayer and the NWK layer
- defining the operating mode of the device (i.e., coordinator, router, or end device)
- device discovery and determination of which application services the device provides
- initiating and/or responding to binding requests
- security management

Device discovery can be initiated by any ZigBee device. In response to a device discovery inquiry end devices send their own IEEE or NWK address (depending on the request). A coordinator or router will send their own IEEE or NWK address plus all of the NWK addresses of the devices associated with it. (A device is associated with a coordinator or router if it is a child node of the coordinator or router.)

Device discovery allows for an ad-hoc network. It also allows for a self-healing network.

Service discovery is a process of finding out what application services are available on each node. This information is then used in binding tables to associate a device offering a service with a device that needs that service.

4.3 ZigBee Addressing

Before joining a ZigBee network (i.e., a LR-WPAN), a device with an IEEE 802.15.4-compliant radio has a 64-bit address. This is a globally unique number made up of an Organizationally Unique Identifier (OUI) plus 40 bits assigned by the manufacturer of the radio module. OUIs are obtained from IEEE to ensure global uniqueness.

When the device joins a Zigbee network, it receives a 16-bit address called the NWK address. Either of these addresses, the 64-bit extended address or the NWK address, can be used within the PAN to communicate with a device. The coordinator of a ZigBee network always has a NWK address of "0."

ZigBee provides a way to address the individual components on the device of a node through the use of endpoint addresses. During the process of service discovery the node makes available its endpoint numbers and the cluster IDs associated with the endpoint numbers. If a cluster ID has more than one attribute, the command is used to pass the attribute identifier.

4.3.1 ZigBee Messaging

After a device has joined the ZigBee network, it can send commands to other devices on the same network. There are two ways to address a device within the ZigBee network: direct addressing and indirect addressing.

Direct addressing requires the sending device to know three kinds of information regarding the receiving device:

1. Address
2. Endpoint Number
3. Cluster ID

Indirect addressing requires that the above three types of information be committed to a binding table. The sending device only needs to know its own address, endpoint number and cluster ID. The binding table entry supplies the destination address(es) based on the information about the source address.

The binding table can specify more than one destination address/endpoint for a given source address/endpoint combination. When an indirect transmission occurs, the entire binding table is searched for any entries where the source address/endpoint and cluster ID matches the values of the transmission. Once a matching entry is found, the packet is sent to the destination address/endpoint. This is repeated for each entry where the source endpoint/address and clusterID match the transmission values.

4.3.2 Broadcast Addressing

There are two distinct levels of broadcast addresses used in a ZigBee network. One is a broadcast packet with a MAC layer destination address of 0xFFFF. Any transceiver that is awake will receive the packet. The packet is re-transmitted three times by each device, thus these types of broadcasts should only be used when necessary.

The other broadcast address is the use of endpoint number 0xFF to send a message to all of the endpoints on the specified device.

4.3.3 Group Addressing

An application can assign multiple devices and specific endpoints on those devices to a single group address. The source node would need to provide the cluster ID, profile ID and source endpoint.

4.4 ZigBee Application Profiles

What is a ZigBee profile and why would you want one? Basically a profile is a message-handling agreement between applications on different devices. A profile describes the logical components and their interfaces. Typically, no code is associated with a profile.

The main reason for using a profile is to provide interoperability between different manufacturers. For example, with the use of the Home Lighting profile, a consumer could use a wireless switch from one manufacturer to control the lighting fixture from another manufacturer.

There are three types of profiles: public (standard), private and published. Public profiles are managed by the ZigBee Alliance. Private profiles are defined by ZigBee vendors for restricted use. A private profile can become a published profile if the owner of the profile decides to publish it.

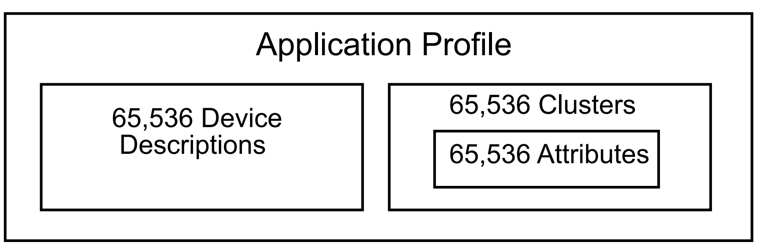
All profiles must have a unique profile identifier. You must contact the ZigBee Alliance if you have created a private profile in order to be allocated a unique profile identifier.

A profile uses a common language for data exchange and a defined set of processing actions. An application profile will specify the following:

- set of devices required in the application area
- functional description for each device
- set of clusters to implement the functionality
- which clusters are required by which devices

A device description specifies how a device must behave in a given environment. Each piece of data that can be transferred between devices is called an attribute. Attributes are grouped into clusters. Figure 4.3 illustrates the relative relationships of these entities and the maximum number that can exist theoretically per application profile.

Figure 4.3 Maximum Profile Implementation



All clusters and attributes are given unique identifiers. Interfaces are specified at the cluster level. There are input cluster identifiers and output cluster identifiers.

At time of this writing, the following public profiles are available:

- Commercial building automation
- Home automation
- Industrial plant monitoring
- Wireless sensor applications
- Smart energy

4.4.1 ZigBee Device Profile

The ZigBee Device Profile is a collection of device descriptions and clusters, just like an application profile. The device profile is run by the ZDO and applies to all ZigBee devices. The ZigBee Device Profile is defined in the ZigBee Application Level Specification. It serves as an example of how to write an application profile.

This page intentionally left almost blank.

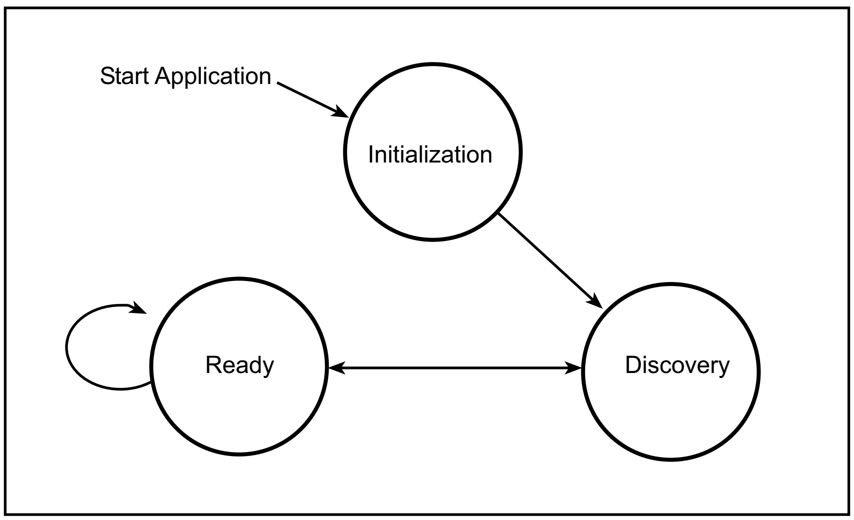
5. Rabbit and ZigBee

This chapter describes how to create a ZigBee application using Dynamic C and Rabbit-based ZigBee-capable boards.

5.1 Implementation Overview

The state machine that describes the underlying logic of a Dynamic C implementation of ZigBee at the application level is pictured in Figure 5.1. It is coded in the tick function `xbee_tick()` [1] which is defined in `xbee_api.lib`. The states are described in the subsections below.

Figure 5.1 ZigBee Application State Machine



5.1.1 Initialization State

This state configures the radio on the RF module, which includes setting the PAN ID and node string defaults. There are two types of firmware used with Rabbit-based hardware: ZNet 2.5 and ZB. They are similar, but are not compatible. That is, a device running ZNet 2.5 firmware cannot talk to one running ZB.

When using ZNet 2.5, the PAN ID defaults to 0x0234. When using ZB firmware, the PAN ID defaults to "0x0123456789abcdef". This longer PAN ID is called the extended PAN ID in the ZigBee specification. Since it is 64 bits in length it is represented as a string in Dynamic C. The node string defaults to "RabbitXBee" in both ZNet and ZB firmware.

The radio's firmware version and the ZigBee device type are checked against the application's expectation; they must match or the initialization will fail.

More information on the firmware is available in Section 5.4.

1. As of Dynamic C 10.44, xbee_tick() is used in place of the deprecated function zb_tick().

5.1.2 Discovery State

A Node Discover (ND) command is sent in the discovery state. The ND command discovers and reports on all RF modules found. This is useful for mapping out the network. An ND command is one of many commands that is used to communicate with the local XBee. For more information on the available AT commands, refer to Table 5-1 or the RF module manual, *XBee® / XBee-Pro® ZB OEM RF Module*, at `www.digi.com`; or run the sample program `\Samples\XBee\AT_interactive.c` to explore the AT commands yourself.

The following information is returned for each RF module found during discovery. The first four correspond to AT commands, each of which may be used separately to read or set a parameter on the XBee.

MY	16-bit network (NWK) address
SH	Serial number high; this is the high 32 bits of the RF module's unique IEEE address
SL	Serial number low; this is the low 32 bits of the RF module's unique IEEE address
NI	String identifier
MP	16-bit network address of the discovered node's parent.
Device Type	1 byte: 0 = coordinator; 1 = router; 2 = end device
Status	1 byte: Reserved
Profile Id	2 bytes: Application profile identifier
Manufacturer ID	2 bytes: Manufacturer specific identifier; the Digi International Inc. identifier is 0x101E.

By default, Node Discover (ND) takes approximately 6 seconds to terminate. During this time, it is essential to call `xbee_tick()` or `ZB_ND_RUNNING` (which calls the tick function) to correctly process the incoming ND packets. Prior to Dynamic C 10.44, other communication with the radio was prohibited during node discovery. But as of Dynamic C 10.44 this is no longer the case. Not being able to get up-to-date information from the node table, which is populated during node discovery, is the only restriction.

The ND timeout should be set larger than the maximum sleep time of any sleeping end devices that will be discovered. For example, if ND times out after 6 seconds and an end device sleeps for 20 seconds, the end device may not be discovered until after the ND timeout imposed by the library.

The ND timeout is controlled by the NT command. For more information on NT, refer to the description for xb_NT in Table 5-1.

5.1.3 Ready State

In this state the radio is queried for new packets. If a new packet is waiting, it is processed. The radio is also queried for status on its networking association. The response lets us know if a network has been joined successfully, or if not, what happened.

The function `xbee_tick()` must be called in the Dynamic C application in order to service new messages. A good rule of thumb is to call `xbee_tick()` whenever the Dynamic C application would otherwise be idle. Special care must be taken to prevent blocking operations and break lengthy processing into small segments. While dependant on expected network traffic, long delays between calls to `xbee_tick()` can result in unwelcome latency and even dropped messages.

Data message processing is performed in `xbee_tick()` using one of three methods.

1. Appropriately addressed messages are automatically routed to an endpoint and cluster function.
2. Messages not processed in a cluster function are passed to an application-defined general message handler.
3. Any message not already processed will cause `xbee_tick()` to return with `ZB_MESSAGE`. The Dynamic C application can then get the message and process it outside of `xbee_tick()`.

Most API functions only apply to the latest received message. `xbee_tick()` must not be called while directly processing a message or a new incoming message may overwrite the old. This could result in data corruption and a multitude of problems depending on the specific API function called. Prior to Dynamic C 10.21, sending AT commands was also forbidden during message processing. The library enforced this restriction for cluster functions; however, the general message handler and the `ZB_MESSAGE` processing were uncontrolled and the application was responsible for not calling `xbee_tick()` or sending AT commands while processing the message. The restriction on sending/receiving AT commands during message processing has been removed with Dynamic C 10.21. The restriction on calling `xbee_tick()` still applies.

NOTE: For more information on AT commands, please refer to Table 5-3.

5.1.4 End Device Sleep Mode

End devices, unlike coordinators and routers, can sleep in order to save power. This is controlled by the radio hardware via its /DTR|SLEEP_RQ|DIO8 pin (or “sleep pin” for short). The XBee’s sleep pin connection to the Rabbit-based target is design dependent.

On the RCM4510W, the sleep pin is connected to power, which means that the non-radio sections of the Rabbit core module are powered down when the XBee makes a sleep request. On the BL4S100, in contrast, the XBee’s sleep pin is not connected to power. So, instead of being powered down when a sleep request is made, an application running on the BL4S100 has options for its power saving strategy. The sample program `sleep.c` illustrates several ways to save power: It turns off the Ethernet interface and the ADC before changing the clock to the real-time clock (i.e., putting the Rabbit into sleepy mode).

Both hardware designs lead to significant savings in power use. In the case of the RCM4510W, the backup battery will maintain SRAM contents and will keep the real-time clock running.

Once put into sleep mode, the XBee module will periodically wake up and poll its parent to determine if there is an incoming message. If a message is found, it will be received and the Rabbit target board will either be restarted (in the case of the RCM4510W), which restarts the user application, or the application will unwind whatever power-saving methods it employed, such as being brought out of sleepy mode. In this latter case, the user application is not restarted since it was not stopped to begin with. The received message will be handled in the normal way by the application.

If no message is found, the XBee module will return to sleep. Depending on initialization of sleep mode, the radio may also restart the RCM4510W or bring the BL4S100 out of sleepy mode after a time-out has passed.

There are several limitations that must be accounted for:

- The RCM4510W and its user application will fully restart upon coming back from sleep.
- Variables that need to be retained between power cycles can be placed in battery-backed RAM if it is available. See the "bbram" keyword in the *Dynamic C User's Manual* for information on how to place variables in battery-backed RAM.
- User data should be initialized prior to calling `xbee_init()` to ensure that a cluster function or the default message handler correctly process an incoming wake message.
- Every sleep request will be preceded by at least 2 seconds of full power operation. This time is set when calling `xb_sleep()`. Any RF or serial traffic received during this time will reset the countdown. If a message is received during this time, it must be immediately processed as it will be unavailable after sleep has begun and then terminated. An application can call `xb_stayawake()` to cancel an imminent sleep if an incoming message must be processed.

There are several sample programs that illustrate how to control sleep mode. If you have a Dynamic C version prior to 10.42, refer to `/Samples/ZigBee/SleepMode.c`. Otherwise, see the board-specific folders (e.g., `Samples/BL4S1xx/XBee/` or `Samples/RCM4500W/XBee/`) for the sleep mode sample programs available for your hardware.

5.2 Sample Programs

This section discusses the Dynamic C sample programs that exercise ZigBee functionality.

Dynamic C sample programs that use ZigBee communication are in the folder `/Samples/XBee`[1] relative to the Dynamic C installation folder. ZigBee sample programs can also be found in folders specific to the hardware, such as: `/Samples/RCM4500W/XBee`, `/Samples/BL4S1xx/XBee` and `/Samples/BLxS2xx/XBee`.

Some of the sample programs can be run with one Rabbit-based board and a DIGI XBee USB device. This device is a simple USB dongle. Its purpose is to aid development by providing a ZigBee coordinator to create a network that the Rabbit-based target can then join as either a router or end device node.

Several sample programs require two Rabbit-based boards.

1. All folders named "XBee" were named "ZigBee" prior to Dynamic 10.44.

5.2.1 Sample Program Initialization Requirements

There are several tasks that must be done by all Dynamic C applications that use ZigBee. If you study the supplied samples, you will see similarities in the configuration code as well as some of the initialization code in `main()`.

All the sample programs define the configuration macros `XBEE_ROLE` and `NODEID_STR`. They both have library default values, but they are useful to put directly in the application code even if you are using the current defaults. Not only is it possible for library default values to change with a newer release of Dynamic C, but having them in the application code is more convenient when you are developing and debugging your software.

After `XBEE_ROLE` and `NODEID_STR` have been defined, as well as any other configuration macros from Section 5.3.2.1, the application must #use the ZigBee library. The next requirement is the creation of the endpoint table. At a minimum your application program will have the following code before `main()`:

```
#define XBEE_ROLE NODE_TYPE_ROUTER
#define NODEID_STR "My Descriptive String"
#use xbee_api.lib
```

Every ZigBee application must construct an endpoint table, even if it is empty:

```
// empty endpoint table
ENDPOINT_TABLE_BEGIN
ENDPOINT_TABLE_END
```

In `main()` there are two tasks that every ZigBee application must accomplish. If the tasks cannot be accomplished, the application should handle any error condition that arises from the attempt:

1. The radio portion of the board must be initialized by calling `xbee_init()`. This function will start the process of joining a network (router or end device) or creating one (coordinator) if the network is not already present.
2. Check the device's network join status and wait until the device is on an active network or has returned an error.

The sample programs illustrate how to accomplish these two tasks. They all have code similar to the following:

```
// Initialize the radio portion of the board
join_start = 0;
while ( (initres = xbee_init()) == -EBUSY){
   if (! join_start){
      join_start = SEC_TIMER;
      printf("Waiting for sleeping XBee to wake before continuing.");
   }
   if (join_start != SEC_TIMER){
      join_start = SEC_TIMER;
      printf( ".");
   }
}
printf( "\n");
```

```
if (initres){
   printf("xbee_init failed");
   exit();
}
```

It is possible for the Rabbit to start while its XBee module is sleeping. The XBee module will not respond while it is sleeping, thus `xbee_init()` will return `-EBUSY`[1] to indicate that fact. But by looping on the return code `-EBUSY`, an application can wait until `xbee_init()` returns something else (such as the return code `-ETIMEOUT` for failure or 0 for success if the XBee woke up). In this way, the initialization function will not return a failure simply because the XBee is asleep.

```
//  Check the join status.  For more information on ZB_JOINING_NETWORK,
//  perform a function lookup (ctrl-H) on ZB_LAST_STATUS().
printf("Waiting to join network...\n");
join_start = MS_TIMER;
while (ZB_JOINING_NETWORK()) {
   //  If unable to join a network, timeout after arbitrary time
   if (MS_TIMER - join_start > XBEE_JOIN_TIMEOUT) {
      printf("\n*** Error ***\n");
      printf("Timed out while trying to join a network.\n");
      exit(-ETIME)
   }
}
printf("Done (%s network)\n", xbee_protocol());
```

Please note that if the device is a ZigBee coordinator, the application may use the same macro (`ZB_JOINING_NETWORK`) that is used to check the join status of routers and end devices in order to determine whether or not a coordinator has finished creating the network.

5.2.2 Summary of ZigBee Sample Programs

The bulleted lists below are the available sample programs.

5.2.2.1 Sample Programs for One Rabbit-Based Board

- API_Test.c - This sample is only available prior to Dynamic C 10.44. Using the simple text interface you can bring up secondary menus that allow you to:
    - read analog and digital lines from the RF module
    - reset the radio
    - enter AT commands
    - put the Rabbit to sleep (end devices only)
    - for some Rabbit-based boards you can also update the RF module firmware
    - send strings to other nodes
    - ping other nodes (other nodes must have API_Test.c running also)

---

1. There are error conditions that will cause the XBee module to be unresponsive, for example, a bad firmware image. In such cases, xbee_init() will return -EBUSY before ultimately returning -ETIMEOUT.

- send_msg.c - This sample program shows how to send a message from a ZigBee end device to the Digi XBee USB ZigBee coordinator or to another end device. (This sample is only available prior to Dynamic C 10.44.)
- AT_interactive.c - This sample program illustrates how to set up and send an AT command. It's also useful for debugging purposes, and to configure some of the registers/commands on the XBee. It displays a menu of some of the more useful AT commands, then prompts user to select one. Running this program successfully verifies that the communication link between the RF module and its Rabbit-based board is working properly.
- AT_runOnce.c - This sample program illustrates how to set up and send an AT command. It sends some of the more useful AT commands to the RF module one time only. It reads the parameters that are returned and displays them in the Stdio window. Running this program successfully verifies that the communication link between the RF module and its Rabbit-based board is working properly.

5.2.2.2 Sample Programs for Two Rabbit-Based Boards

- sleep.c/SleepMode.c/SleepMode2.c - These sample programs are hardware specific. They demonstrate how to put a Rabbit device into sleep mode within a ZigBee environment. The Rabbit device must be an end device, as only end devices are allowed to sleep in a ZigBee network.
- EndPoint.c - This sample program requires two Rabbit-based targets. It shows how to set up and use endpoints.
- GeneralMessageHandler.c - This sample program requires two Rabbit-based targets. It demonstrates the use of the General Message Handler function, which sends a message between two Rabbit-based boards.

5.2.3 GPIO Server/Client Sample Programs

This collection of sample programs has both server and client applications. The GPIO protocol defined for these sample programs is described at the top of the server program files. The protocol defines a message handling agreement between the two sides, which is essentially the frame formats for the GPIO requests and their responses.

General purpose I/O includes both digital and analog I/O, making these applications a useful template for developing a wide variety of embedded systems software.

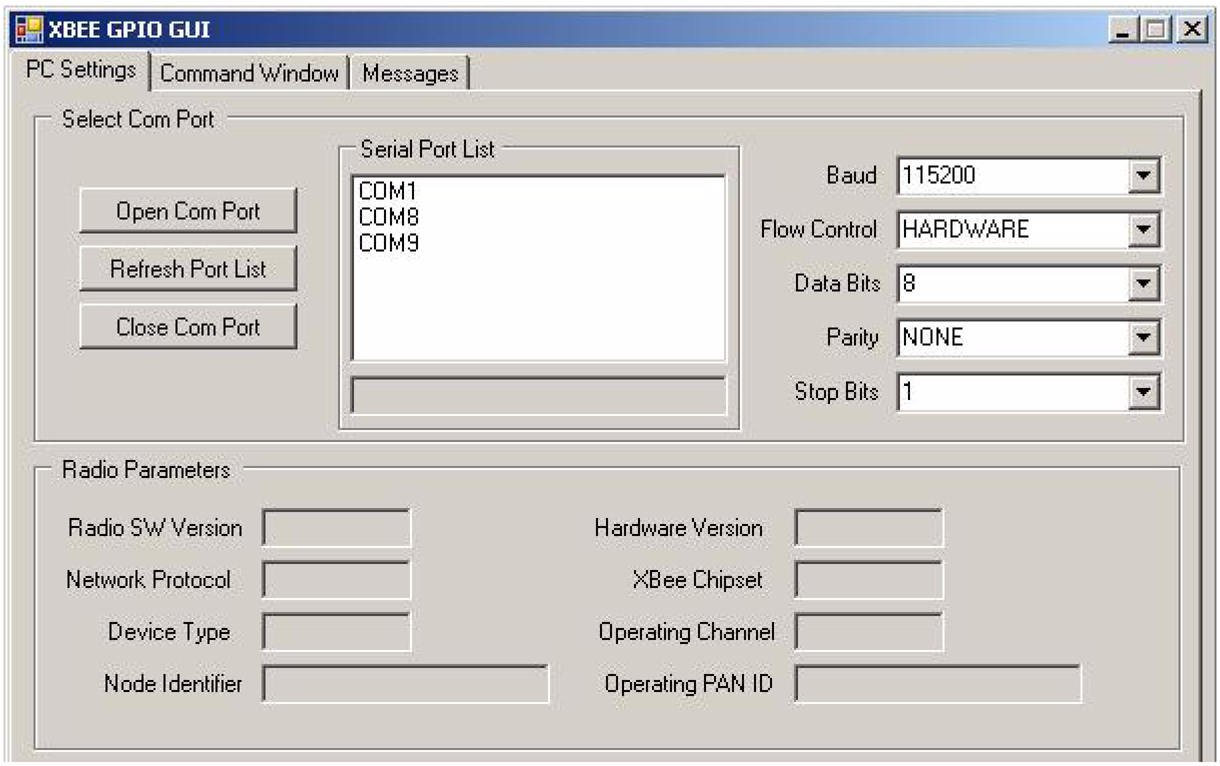
5.2.3.1 Running the GPIO Applications

The server application is hardware specific. The sample programs are in hardware specific folders (e.g., `\Samples\RCM4500W\XBee\XBee_GPIO_Server.c`). After the server application has been compiled and is running on the target board, it will attempt to join a network and if successful, will then wait for client requests.

The client application can be run in two different ways:

1. The first way uses the Digi XBee USB device. Plug the Digi XBee USB into your host PC. There is a Visual Basic application, `\Utilities\XBee GPIO GUI\XBEE_GPIO_GUI.exe`, that when run will display something very similar to the screen in Figure 5.2.

Figure 5.2 Opening Screen of VB GPIO Client

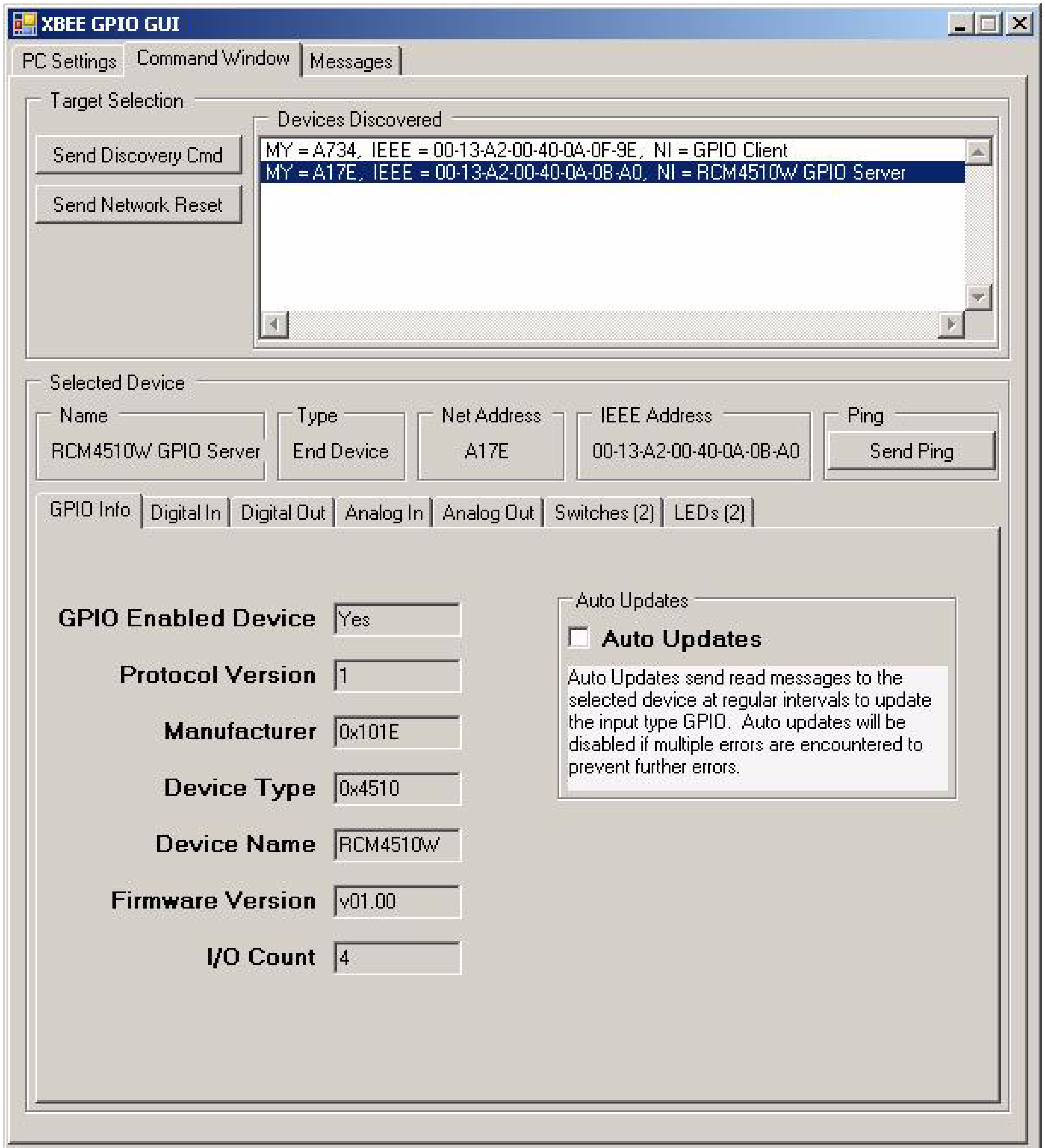


In the "Serial Port List" window select the COM port that is connected to the Digi XBee USB device. Make sure the baud rate matches. (The default is 115200 bps from the factory; however, 9600 is also very common.) Click on the button labeled "Open Com Port." This results in the radio parameters being filled in.

If an error occurs when you try to open the COM port, make sure you don't have something else open on that port. This is a common reason for getting an error.

After you have successfully connected the VB GPIO GUI client to the Digi XBee USB device, click on the tab labeled "Command Window." Select a device in the "Devices Discovered" area. This results in information being displayed for the selected device in the lower half of the GUI window, as shown below in Figure 5.3:

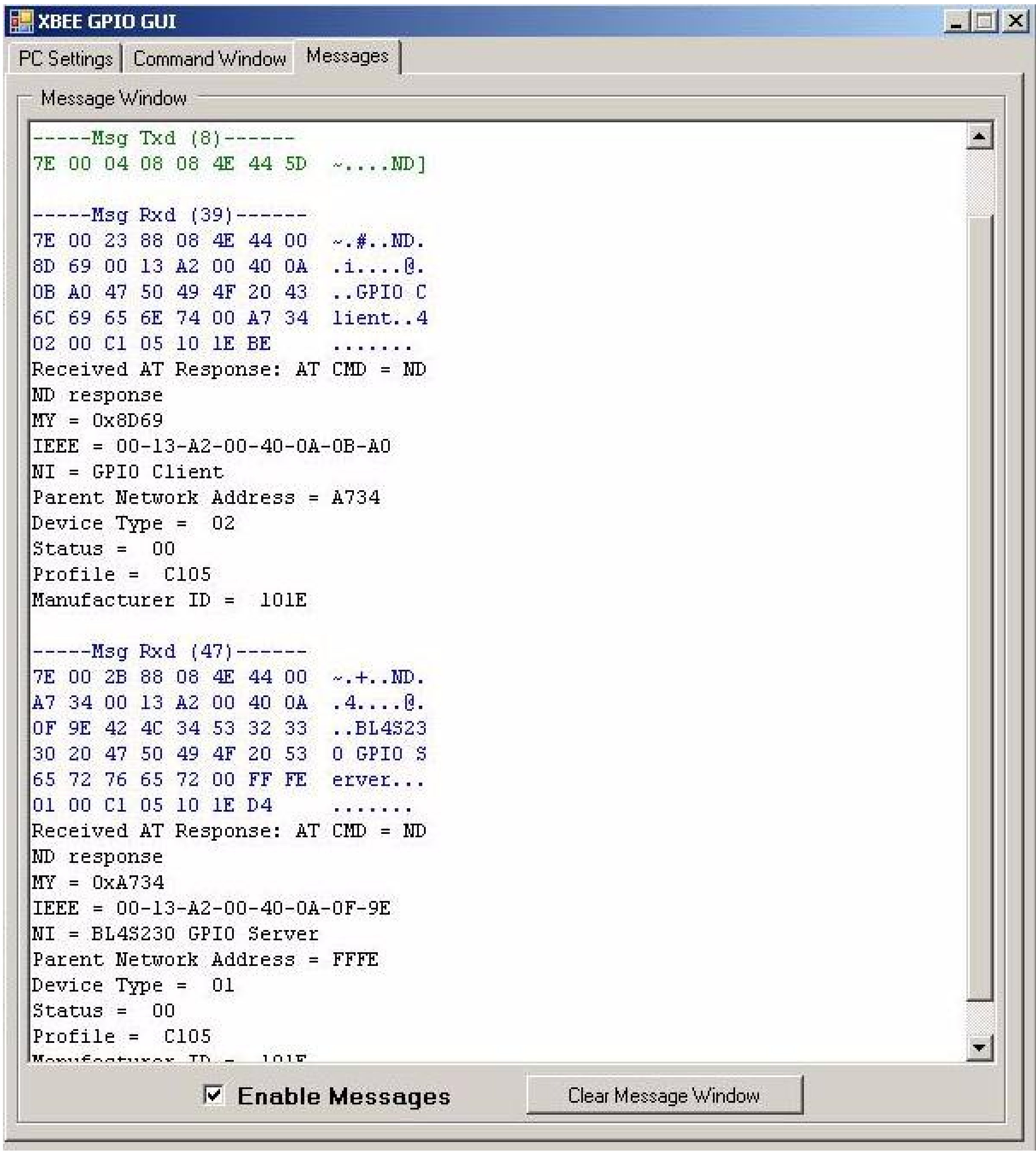
Figure 5.3 General Information for Selected Device



The tabs in the lower half of the “Command Window” screen let you view and modify values for the various I/O signals on the selected device. Which tabs contain information depends on the available I/O of the selected device.

The third main tab, “Messages,” displays the messages transmitted and received by the client. This message window is a valuable resource for understanding the communication between the server and client. Figure 5.4 shows the messages exchanged when the “Send Discovery Cmd” button is clicked.

Figure 5.4 GPIO GUI Messages



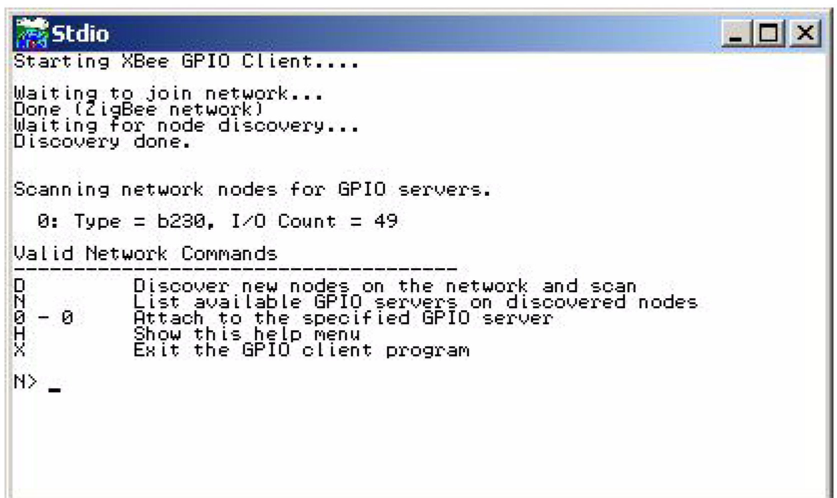
As you can see, a Node Discovery (ND) command was transmitted from the client in response to clicking the “Send Discovery Cmd” button. The actual byte values are followed by an English translation of their meaning.

In this example, two ZigBee devices responded with the following information:

- MY - 16-bit network address
- IEEE - 64-bit IEEE address
- NI - node identifier
- Parent Network Address - 16-bit parent network address; a value of 0xFFFE indicates no parent; only end devices have parents since they are the only devices that need messages buffered
- Device Type: 0=Coordinator, 1=Router, 2=End Device
- Status - reserved
- Profile - application profile
- Manufacturer ID - manufacturer's identifier

2. The second way to run the GPIO client uses a Rabbit-based board instead of the Digi XBee USB device. The Dynamic C sample program, `\Samples\XBee\XBee_GPIO_Client.c`, when compiled and run on a ZigBee-capable board performs the client side of the GPIO application. It is a command line version of the VB GUI application. The opening screen is shown in Figure 5.5.

Figure 5.5 GPIO Client Screen



In this example, there was one GPIO server found by the scan. From the "Network Commands" menu you can list the I/O signal names and types available on the GPIO server by entering the server's number at the prompt. This will also result in a new command menu being displayed that will allow for the reading and setting of the individual I/O signals.

5.2.3.2 Studying the Code

The GPIO server/client application illustrates the setting up and use of endpoints and clusters. The protocol defined for the GPIO application requires that the cluster ID values on both the server and client must match if the server is to recognize the client's request and likewise if the client is to recognize the server's response.

In both `XBee_GPIO_Server.c` and `XBee_GPIO_Client.c` the cluster IDs have the same values and are named:

- `XBEE_GPIO_CLUST_INFO`
- `XBEE_GPIO_CLUST_NAME`
- `XBEE_GPIO_CLUST_ANA_RANGE`
- `XBEE_GPIO_CLUST_READ`
- `XBEE_GPIO_CLUST_WRITE`

Although the cluster IDs are the same, the functions associated with them differ between the server and the client. The client makes a request that is then handled by the server. The client will then handle the response that is sent back from the server. This is shown in the code below.

From `XBee_GPIO_Server.c`:

```
RabbitClusterIDList_t const gpioEndPointReq = {
{XBEE_GPIO_CLUST_INFO, XBEE_GPIO_CLUST_NAME, XBEE_GPIO_CLUST_ANA_RANGE,
XBEE_GPIO_CLUST_READ, XBEE_GPIO_CLUST_WRITE },
{xbeeGpioDevInfoReq, xbeeGpioNameReq, xbeeGpioAnaRangeReq,
xbeeGpioReadReq, xbeeGpioWriteReq }
};

ENDPOINT_TABLE_BEGIN
ENDPOINT_TABLE_ENTRY(XBEE_ENDPOINT_GPIO,0,XB_PROFILE_DIGI,1,0,5,0,\
   &gpioEndPointReq, NULL)
ENDPOINT_TABLE_END
```

The code in `XBee_GPIO_Client.c` uses the same cluster ID values, but associates those cluster IDs with the functions that handle the various server responses.

```
RabbitClusterIDList_t const gpioEndPointResp = {
{XBEE_GPIO_CLUST_INFO, XBEE_GPIO_CLUST_NAME, XBEE_GPIO_CLUST_ANA_RANGE,
      XBEE_GPIO_CLUST_READ, XBEE_GPIO_CLUST_WRITE },
{xbeeGpioDevInfoResp, xbeeGpioNameResp, xbeeGpioAnaRangeResp,
      xbeeGpioReadResp, xbeeGpioWriteResp }
};

ENDPOINT_TABLE_BEGIN
ENDPOINT_TABLE_ENTRY(XBEE_ENDPOINT_RESPONSE, 0, XB_PROFILE_DIGI, 1, 0,
5, 0,\
        &gpioEndPointResp, NULL)
ENDPOINT_TABLE_END
```

What you will notice if you search through the server and client code is that the cluster functions are not explicity called within the code of either program. This is because the messages received by the ZigBee device are handled within the tick function of `xbee_api.lib`, as described in Section 5.1.3.

For more information about the parameters for `ENDPOINT_TABLE_ENTRY`, see the description for the `ENDPOINT_TABLE_*` macros in Section 5.3.2.1.

5.3 Dynamic C Library for ZigBee Applications

This section contains information about the library provided for ZigBee-capable devices. Data structures, error codes and configuration macros from the library are also discussed.

The Dynamic C library that supports ZigBee connectivity is `xbee_api.lib`. It is located in the folder `lib\..\XBee` relative to the Dynamic C installation folder. (Prior to Dynamic C 10.44, the folder was named "ZigBee.") The inclusion of the library in the application code must come after the configuration macro definitions. For example, if you are compiling and running your application on an end device, your program must order the lines of code as follows:

```
#define XBEE_ROLE NODE_TYPE_ENDDEV[1]
#use "xbee_api.lib"
```

Instead of defining the logical device type in the application, it can be defined in the application's project file. (See the "Defines" tab in the menu: Options | Program Options.)

5.3.1 Communication with an RF Module

Using Dynamic C, a Rabbit-based device on a ZigBee network may communicate with its XBee module to read or modify radio parameters, as well as communicate with other devices on its network. The API mode of operation is used as an alternative to the Transparent Operation (serial line replacement) of the RF module. The API mode requires a data frame structure be passed between the Rabbit-based device and the XBee.

The Dynamic C function `zb_sendAPICmd()` takes care of sending the data frame to the RF module. `zb_sendAPICmd()` is called by several other Dynamic C functions. Which one of these preliminary functions to use depends on what the application requires. The application will need to send a message to a remote device or will need to send an AT command to the RF module.

5.3.1.1 Sending Data to a Remote Device

If a message is being sent to a remote device on the network, the application must call the API function `zb_send()`.

(For information on handling messages received from a remote device, please see Section 5.1.3.)

1. Prior to Dynamic C 10.40, the #define statement would be: #define ZIGBEE_ENDEV

5.3.1.2 Radio Commands

If an AT command is being sent to the XBee, the application must call either the non-blocking function `zb_sendATCmd()` or the blocking function `zb_API_ATCmdResponse()`. Starting with Dynamic C 10.44, you can also use one of three wrapper functions for `zb_API_ATCmdResponse()`; they are: `xb_get_register()`, `xb_set_register()`, and `xb_send_command()`.

The non-blocking function does not wait for a reply, whereas the blocking functions do.

> **NOTE:** Prior to Dynamic C 10.21, a Dynamic C application cannot call the functions `zb_sendATCmd()` or `zb_API_ATCmdResponse()` when executing a Dynamic C function associated with a cluster ID. If it is attempted, the application will receive a `ZBERR_TX_LOCKED` return value.

5.3.2 Configuration Macros and Constants

This section lists the Dynamic C configuration macros and constants that are of interest to application developers.

5.3.2.1 Compile-Time Macros

All of the configuration macros listed here, except for the ones that build the endpoint table, must be defined either in the application program prior to the "#use xbee_api.lib" statement or in the "Defines" tab in the Options | Project Options dialog.

DEFAULT_CHANNELS

This is a bitmask of the channels that will be scanned when a device attempts to associate with a network. It is valid for both ZNet 2.5 and ZigBee (ZB) firmware.

Default = `ZB_DEFAULT_CHANNELS` prior to Dynamic C 10.44
Default = `XBEE_DEFAULT_CHANNELS` starting with Dynamic C 10.44

Both default values (0x1FFE) allow 12 channels to be scanned and used by the coordinator.

DEFAULT_EXTPANID

Default = "0x0123456789abcdef". Every ZigBee network requires a personal area network (PAN) ID. A coordinator with ZB firmware uses `DEFAULT_EXTPANID` as the 64-bit PAN ID for the network it is creating. Note that this macro is defined as a string in order to fulfill the requirement of it being a 64-bit value.

If you are using ZB firmware and have set `DEFAULT_PANID` in your program, but not `DEFAULT_EXTPANID`, `xbee_api.lib` will use `DEFAULT_PANID` padded to the left with zeros as the PAN ID for the network.

Setting `DEFAULT_EXTPANID` to "0x00" tells a coordinator to pick a random value for the PAN ID, and tells a router or end device to join any network.

DEFAULT_PANID

Default = 0x0234. Every network requires a personal area network (PAN) ID. A coordinator with ZNet firmware uses `DEFAULT_PANID` as the 16-bit PAN ID for the network it is creating.

The range for a valid PAN ID is 0x0 to 0x3FFF, inclusive. The value 0xFFFF is also valid. It is used to tell a coordinator to choose a random value for the PAN ID, and tells a routers or end device to join any network.

ENDPOINT_TABLE_*

This group of macros builds the endpoint table.

Before the endpoint table can be created, the Dynamic C application must initialize the structure that holds cluster ID and function information. Cluster IDs refer to the functions implemented on an endpoint. Endpoints may implement zero, one, or any number of the functions listed in a Cluster ID list.

Some variation of the following code is needed in a Dynamic C application that wants to run in a ZigBee network.

```
RabbitClusterIDList_t const StringInCluster = {
   { CLUSTER_ID },
   { recvString }
};
```

In the above code, `CLUSTER_ID` is associated with the function `recvString()`. Functions that are associated with a cluster ID are called cluster functions.

The application uses the cluster ID/function structure (`RabbitClusterIDlist_t`) to create entries in the endpoint table with the `ENDPOINT_TABLE_*` macros provided for this purpose. The code below is an example of building an endpoint table.

```
ENDPOINT_TABLE_BEGIN
ENDPOINT_TABLE_ENTRY(EP_NUM, 0, PROFILE_ID, 1, 0, 1, 0, &StringInClus-
ter, NULL)
ENDPOINT_TABLE_END
```

The parameters for `ENDPOINT_TABLE_ENTRY` are:

EP	Endpoint number, in the range 1 to 219, inclusive.
DSC	Reserved for future use. Currently set to 0 in sample programs.
PID	Application profile identifier; the macro `XB_PROFILE_DIGI` indicates Digi's private application profile.
DID	Device identifier, in the range 0 to 65,535, inclusive. This number can be used any way desired by the application.
flags	User-defined byte.
ICCOUNT	The number of input cluster functions in the RabbitClusterIDList_t structure referenced in the "ICL" field.
OCCOUNT	The number of output cluster functions in the RabbitClusterIDList_t structure referenced in the "OCL" field. This field is not currently used and should be set to 0.
ICL	Input cluster ID list. This is the address of a `RabbitClusterIDList_t` structure.
OCL	Output cluster ID list. This is the address `RabbitClusterIDList_t`, but not currently used and should be set to NULL.

Any device using `xbee_api.lib` must define an endpoint table, even if it is empty. An empty table is defined as:

```
ENDPOINT_TABLE_BEGIN
ENDPOINT_TABLE_END
```

NODEID_STR

This macro defines the NI value for the node and contains a maximum of 20 characters. It gives each node a unique identifier.

Default = "RabbitZigBee" prior to Dynamic C 10.40.
Default = "RabbitXBee" starting with Dynamic C 10.40.

XBEE_DEBUG

This configuration macro enables debugger functionality. This is necessary for things like setting breakpoints and stepping through the ZigBee library code. This macro replaced the deprecated macro `ZB_DEBUG` starting with Dynamic C 10.44.

XBEE_IN_BUF / XBEE_OUT_BUF

Default = 255 bytes for each buffer. This is the recommended minimum size for the serial buffers. They should be large enough to hold an entire frame.

XBEE_ROLE

This configuration macro defines the device type. It defaults to `NODE_TYPE_ROUTER`. It can be defined to one of the following:

- `NODE_TYPE_COORD` - the device will be a coordinator.
- `NODE_TYPE_ROUTER` - the device will be a router.
- `NODE_TYPE_ENDDEV` - the device will be an end device.

XBEE_VERBOSE

This configuration macro enables/disables the printing of extra information generated from the library code to the Stdio window. This macro replaced the deprecated macro `ZB_VERBOSE` starting with Dynamic C 10.44.

ZB_CONSTRUCT_NODE_ID

Define this if you want to construct your own node ID string at runtime.

```
#define ZB_CONSTRUCT_NODE_ID <function-name>
```

This will allow your software to construct an ID that could contain information about what the capabilities of the device connected to the radio are. A node ID string must be made up of printable ASCII characters, and be no more than `_MAX_NODE_ID_LEN` (20) characters long.

The function prototype must be:

```
char *<function-name>(void);
```

The function must return a pointer to static data.

ZB_FATAL_ERROR
The ZB_FATAL_ERROR macro handles the case where the library xbee_api.lib is reporting that it is experiencing an error beyond its capability to handle. This will usually occur during startup if the radio is not responding.

```
#define ZB_FATAL_ERROR <function-name>
```

The fatal error handler callback function prototype must be:

```
void <function-name>(int errorcode);
```

The error code is one of the defined errors in /Lib/../ERRNO.LIB, relative to the Dynamic C installation folder.

ZB_MULTI_PROFILE
The #define of this macro enables Profile ID checking in the message interpretation function for explicitly addressed messages. This check will require a received Profile ID to match an associated Endpoint Table Endpoint Descriptor Profile ID before calling the associated callback function.

5.3.2.2 Information Macros

The macros listed here are defined in xbee_api.lib and should not be modified by an application.

XBEE_IS_COORD
This macro will equal TRUE if the device is configured to be a coordinator, and FALSE otherwise.

XBEE_IS_ENDDEV
This macro will equal TRUE if the device is configured to be an end device, and FALSE otherwise.

XBEE_IS_ROUTER
This macro will equal TRUE if the device is configured to be a router, and FALSE otherwise.

5.3.2.3 Deprecated Device Type Macros

The macros listed here were deprecated starting with Dynamic C 10.40 in favor of the XBEE_ROLE macro.

ZIGBEE_COORDINATOR
This configuration macro defines the device as a coordinator.

ZIGBEE_ENDDEV
This configuration macro defines the device as an end device.

ZIGBEE_ROUTER
This configuration macro defines the device as a router.

5.3.3 Error Codes

Most of the error codes returned from the API functions in xbee_api.lib are defined in Lib\..\errno.lib. There are no true fatal errors; however, the I/O error -EIO is fatal in terms of not being able to recover from it without having specialized knowledge and making some low-level internal function calls.

Refer to ZB_FATAL_ERROR for handling error conditions that are beyond the capability of the library.

5.3.4 Data Structures

There are many data structures defined in `xbee_api.lib`. Most of them are related to the data packets that are received and transmitted via the RF module. There are also data structures for information about endpoints.

api_frame_t

This data structure holds data packet information. It is used by both send and receive functions. Many of the data structures defined at the beginning of the ZigBee library are used to construct `api_frame_t`.

A pointer to an instance of this data structure is returned by `zb_receive()`, which is a function that is called when `xbee_tick()` indicates that a message is waiting to be handled. A pointer to an instance of this data structure is passed by `zb_sendAPICmd()` to the RF module when the application wants to send a message to another device or wants to read or modify radio parameters.

xb_io_sample_t

This data structure holds information about the function and state of digital and analog pins on the XBee module.

_zb_NodeData_t

This data structure holds a node data entry. The information on a network node includes:

- 16-bit NWK address
- 64-bit IEEE address
- string identifier, 21 characters including NULL
- 16-bit NWK address of node's parent
- device type: coordinator, router, or end device.
- application profile ID
- whether the node is currently on the network

zb_sendAddress_t

This is the address data structure. It is used whenever a Rabbit ZigBee device transmits a message to another ZigBee device. The functions `zb_MakeEndpointClusterAddr()` and `zb_MakeIEEENetworkAddr()` build the data structure (passed as an argument) and then return the pointer passed to them.

The difference between `zb_MakeEndpointClusterAddr()` and `zb_MakeIEEENetworkAddr()` is that the former uses the endpoints and the cluster ID for addressing, while the latter function does not. Choosing which one to call allows the Dynamic C application to specify how a message is addressed.

The `zb_sendAddress_t` data structure is used by the functions that send messages to another ZigBee device: `zb_send()` and `zb_reply()`. The application will typically fill in the message and message length fields, either directly before calling `zb_send()`, or indirectly by passing the information to `zb_reply()`.

5.3.5 API Functions and Macros

This section contains function descriptions for ZigBee-specific functions and macros.

GET_NODE_DATA

```
_zb_NodeData_t * GET_NODE_DATA ( int index )
```

DESCRIPTION

This macro gets the node data located at the indexed spot in the array. The node array is typically populated by nodes that responded to a Network Discovery (ND) command. Information about the nodes are stored in the array in the order received.

The macro calls a function to determine where in memory the element is because the array could extend into xmem. Repeated calls to this macro do not incur a processing penalty since the last access is remembered. Note that the node structure is used internally by the library for addressing. The data stored in the node structure must be in network order.

This macro is non-reentrant.

PARAMETER

index	Index into the node array

RETURN VALUE

Address of the node in root memory or NULL if index is out of range.

LIBRARY

`xbee_api.lib`

resetRadio

```
void resetRadio ( void );
```

DESCRIPTION

Reset the XBee radio by toggling its reset line.

RETURN VALUE

None

LIBRARY

`xbee_api.lib`

xbee_awake

int xbee_awake( )

DESCRIPTION

This device-specific macro reads the XBee module's SLEEP_REQ pin (if available). Used by Rabbit hardware running XBee end device firmware to enter and exit low power mode.

This function was introduced in Dynamic C 10.46.

RETURN VALUE

1 = XBee is not asking the Rabbit to sleep (not asserting -SLEEP_REQ).
0 = XBee is asking the Rabbit to sleep (asserting -SLEEP_REQ).

LIBRARY

xbee_config.LIB

SEE ALSO

xbee_wait_for_wake()

xbee_init

int xbee_init(void);

DESCRIPTION

Initialize the Rabbit XBee driver and the XBee radio.

This function was introduced in Dynamic C 10.40, replacing the deprecated function zigbee_init().

RETURN VALUE

0: successful
-EBUSY: XBee end device is sleeping, try again. After 28 seconds of -EBUSY, xbee_init() will return -ETIME
-ETIME: XBee timed out
!=0: failure

See _zb_error for the specific error code. The values for _zb_error are defined in /Lib/../ERRNO.LIB relative to the Dynamic C installation folder.

LIBRARY

xbee_api.lib

xbee_protocol

```
char * xbee_protocol();
```

DESCRIPTION

Returns a string identifying the network protocol in use by the XBee connected to the Rabbit.

RETURN VALUE

One of the following strings:

- "ZNet 2.5"
- "ZigBee"
- "Unknown"

xbee_tick

`int xbee_tick ( api_frame_t * frame );`

DESCRIPTION

Drive the Rabbit XBee radio communications. Performs a Network Discovery once at initialization time.

This function was introduced in Dynamic C 10.44 and replaces the deprecated function `zb_tick()`.

PARAMETER

`frame`	Pointer to `api_frame_t` structure to receive a copy of the last frame. Note that the received frame may be a response to a packet sent by `xbee_api.lib`, or a non-response frame from another device on the network. Check the frame type (`frame->cmd.api_id`) and the frame id (`frame->cmd.u.frame_id`) to confirm that it is the expected response.

RETURN VALUE

`ZB_NOMESSAGE`: no messages received
`ZB_MESSAGE`: a message has arrived
`ZB_RADIO_STAT`: radio status change
`ZB_MSG_STAT`: message transmission status available
`ZB_ATRESP`: response to AT command
`ZB_REMOTE_RESP`: response to remote AT command
-`ENOMEM`: out of memory processing node discovery response
(various codes)<0: an error has occurred

NOTE: To retain backward compatibility with deprecated zb_tick (which is just a macro for `xbee_tick(NULL)`), return codes `ZB_ATRESP` and `ZB_REMOTE_RESP` are only returned if the frame parameter is not NULL.

LIBRARY

`xbee_api.lib`

xbee_wait_for_wake

`int xbee_wait_for_wake( void );`

DESCRIPTION

If the XBee is sleeping, this function will queue a null byte in the serial buffer so the XBee will stay awake as soon as it wakes up (and asserts CTS and receives the null byte).

Called from user program to wait for a sleeping end device to wake up. Can only be used after calling `xbee_init()`.

```
while (xbee_wait_for_wake()) {
   twiddle_thumbs();
   do_other_stuff();
}
```

On a router or coordinator, `xbee_wait_for_wake()` always returns 0.

This function was introduced in Dynamic C 10.46.

RETURN VALUE

`TRUE`: if waiting for the XBee to wake up
`FALSE`: if XBee is awake.

LIBRARY

`xbee_api.lib`

SEE ALSO

`xbee_awake`

xb_get_register

```
int xb_get_register( word reg, unsigned long * dest );
```

DESCRIPTION

Read an XBee register (up to 32 bits).

This function was introduced in Dynamic C 10.44.

PARAMETERS

reg	Register to read, as a 16-bit word (see `xb_XX` definitions in Table 5-1 for values to use). Also possible to use *(word *)"XX" for reading the ATXX register. Examples: `xb_VR` for firmware version, `xb_SH` for top 4 bytes of serial number, `xb_AI` for the association indicator.
dest	Address to store up to a 32-bit result. Can be NULL to send a command to the XBee and ignore the result.

RETURN VALUE

0: Success
`-ZBERR_AT_CMD_RESP_STATUS`: Radio returned failure
`-ETIME`: Timeout
`-EIO`: Serial I/O error
`-ENOSPC`: Output buffer full

LIBRARY

`xbee_api.lib`

xb_hexdump

```
void xb_hexdump( void * data, int len );
```

DESCRIPTION

Hex dump <len> bytes from <data>. Displays 8 hex bytes and their printable characters or a '.'.

This function was introduced in Dynamic C 10.44.

PARAMETERS

data	Pointer to buffer.
len	Number of bytes to print.

RETURN VALUE

None

LIBRARY

`xbee_api.lib`

xb_io_conf_desc

```
far char * xb_io_conf_desc ( int dio, int config );
```

DESCRIPTION

Returns a description of an XBee module's I/O pin configuration.

PARAMETERS

dio	DIO number, 0 <= dio < `XBEE_IO_COUNT` (13).
config	Configuration for pin (ATDx or ATPx setting), 0 to 5.

RETURN VALUE

`NULL`: Invalid parameters passed in (dio or config out of range).
`XB_IO_CONF_INVALID`: Invalid configuration for given dio (e.g., digital-only pin configured as analog input)
`!NULL`: String describing a pin's configuration.

LIBRARY

`xbee_api.lib`

xb_io_sample_clear

```
void xb_io_sample_clear (far xb_io_sample_t * sample );
```

DESCRIPTION

Reset the sample structure to default values (0x00 for first two values, -1 for the next five).

This function was introduced in Dynamic C 10.40.

PARAMETERS

xb_io_sample_t	Pointer to the `xb_io_sample_t` structure to clear.

RETURN VALUE

None

LIBRARY

`xbee_api.lib`

xb_IS_parse

```
int xb_IS_parse ( far xb_io_sample_t * sample, far char * IS_data );
```

DESCRIPTION

Parse the return data from an ATIS command or 0x92 (`_API_FRAME_IO_SAMPLE`) frame.

This function was introduced in Dynamic C 10.40.

PARAMETERS

sample	Pointer to the `xb_io_sample_t` buffer to receive the parsed data.
IS_data	Pointer to the data returned from ATIS or the 0x92 frame.

RETURN VALUE

0: if parsed successfully
`-EINVAL`: if there's a problem with parameter 2

LIBRARY

`xbee_api.lib`

xb_listNodes

```
void xb_listNodes ( void );
```

DESCRIPTION

Display list of nodes from the node table to Stdout. Useful when displaying information to the Stdio window and prompting the user to select a node.

Each node is listed with its index (for selecting a node), 16-bit network address, 16-bit parent address, 64-bit IEEE address, Active/Inactive state and its Node ID string.

This function was introduced in Dynamic C 10.44.

RETURN VALUE

none

LIBRARY

`xbee_api.lib`

xb_nd_nodetype_str

```
char * xb_nd_nodetype_str ( int nd_type );
```

DESCRIPTION

Used to convert from the node type field in an ND response to a string describing the type.

This function was introduced in Dynamic C 10.44.

PARAMETER

nd_type Valid parameters are:

- XB_ND_NODETYPE_COORD
- XB_ND_NODETYPE_ROUTER
- XB_ND_NODETYPE_ENDDEV

RETURN VALUE

If nd_type = XB_ND_NODETYPE_COORD, the return value is "coordinator"
If nd_type = XB_ND_NODETYPE_ROUTER, the return value is "router"
If nd_type = XB_ND_NODETYPE_ENDDEV, the return value is "end device"

If nd_type is anything else, the return value is "invalid"

LIBRARY

xbee_api.lib

xb_sendAPIremoteATcmd

```
int xb_sendAPIremoteATcmd ( far char dest_ieee[8], word nwk_addr, int
   options, int remote_cmd, far void * remote_data, int len );
```

DESCRIPTION

Send a command frame to a remote device using the XBee API format.

This function was introduced in Dynamic C 10.44.

PARAMETERS

dest_ieee	64-bit IEEE address of target device. Only used if nwk_addr (parameter 2) is set to `ZB_NETWORK_BROADCAST`..
nwk_addr	16-bit network address of target device, or `ZB_NETWORK_BROADCAST` to use IEEE addressing.
options	Options for the frame, valid choices are 0 (default) or `XB_REMOTE_REQ_OPT_APPLY` (0x02) (apply changes on remote).
remote_cmd	16-bit command to send (see the `xb_XX` macro definitions in Table 5-1 for values to use). Also possible to use *(word *)"XX" to send an ATXX command.
remote_data	Pointer to data to send.
len	Amount of data.

RETURN VALUE

0: success
`-ENOSPC`: not enough buffer space to send command
`-EIO`: XBee holding CTS low, or IO did not complete correctly

_zb_error will be set on error or successful message transmission.

LIBRARY

`xbee_api.lib`

MACRO DEFINITIONS

The `remote_cmd` parameter can be one of the macros in Table 5-1. Be careful when sending an AT command to the XBee, as some of the commands will cause a loss of communication (for example, sending xb_BD to change the baud rate). More information on AT commands is in the manual *XBee® / XBee-PRO® ZB OEM RF Modules* available at: `www.digi.com`.

Table 5-1. Commands to Send to XBee RF Modules

Special Commands, Write Only	
xb_FR	Respond with "OK" then XBee firmware reset in ~2 seconds
xb_NR	Reset network layer parameters
xb_RE	Restore module parameters to factory defaults
xb_WR	Write parameter values to non-volatile memory
Networking Commands	
xb_CH	Read RF operating channel. A value of 0 means the device has not joined a network.
xb_OP	Read operating PAN ID.
xb_JV	Read channel verification parameter. • JV=0: a router will continue operating on its current channel after a power cycle even if a coordinator is not detected. • JV=1: if the network is an open network (NJ=0xFF), a router will verify the coordinator on its operating channel when joining or coming up from a power cycle. If a coordinator is not detected, the router will leave its current channel and attempt to join a new PAN.
	Write channel verification parameter: • 0=channel verification disabled • 1=channel verification enabled
xb_ID	Read/write the PAN ID (ZB:64-bit, ZNET:16-bit)
xb_MY	Read 16-bit source address. (0xFFFF=disable)
xb_MP	Read 16-bit address of parent (for end nodes only)
xb_SH	Read upper 4 bytes of the 8 byte IEEE source address. The 64-bit source address is always enabled.
xb_SL	Read lower 4 bytes of the 8 byte IEEE source address. The 64-bit source address is always enabled.
xb_RN	Read/write minimum value of CSMA-CA back-off exponent
xb_NI	Read/write a string called the Node Identifier.

Table 5-1. Commands to Send to XBee RF Modules

xb_ND	Discovers and reports all RF modules found. The following information is reported for each module:
	`MY`
	`SH`
	`SL`
	`NI`
	`MP (2 Bytes)`
	`DEVICE_TYPE (1 Byte: 0=Coord, 1=Router, 2=End Device)`
	`STATUS`
	`PROFILE_ID (2 Bytes)`
	`MANUFACTURER_ID (2 Bytes)`
	Each response is returned as a separate AT_CMD_Response packet and handled internally by the library through `xbee_tick()`.

xb_NT	Read/write Node Discover Timeout. This is the amount of time a node will spend discovering other nodes when ND or DN is issued. The range is 0X20-0XFF; the equation NT*100 ms results in valid times of 3200 ms thru 25500 ms, inclusive.
xb_DN	Destination Node. Resolves an NI string to a physical address (case sensitive). After the destination node is discovered the 16-bit network and 64-bit extended addresses are returned in an API Command Response frame.
xb_SC	Read/set the list of channels to scan. • Coordinator - bit field of channels to choose from prior to starting network. • Router/End Device - bit field of channels that will be scanned to find a coordinator or router to join. Changes to SC do not take effect until the WR command is issued. Bit (Channel): 0 (0x0B) 4 (0x0F) 8 (0x13) 12 (0x17) 1 (0x0C) 5 (0x10) 9 (0x14) 13 (0x18) 2 (0x0D) 6 (0x11) 10 (0x15) 14 (0x19) 3 (0x0E) 7 (0x12) 11 (0x16) 15 (0x1A) Changing SC from its default value or setting it to include more than 12 continuous channels may cause communication problems. See the “XBee/XBee-PRO ZB OEM RF Modules” manual for details.
xb_SD	Read/set the scan duration exponent. Changes to SD do not take effect until the WR command is issued. • Coordinator - duration of the Active and Energy Scans (on each channel) that are used to determine an acceptable channel for the coordinator to start up on. • Router/end Device - duration of Active Scan (on each channel) used to locate an available coordinator/router to join. Scan time is measured as: (# channels to scan) * (2^SD) * 15.36 ms.

Table 5-1. Commands to Send to XBee RF Modules

xb_NJ	Read/write the time that a coordinator/router allows nodes to join. This value can be changed at run time without requiring a coordinator or router to restart. The time is reset on power cycles or when NJ changes. • Limit time for joining: 0x0 - 0xFE [x 1 sec.] • Always allow joining: 0xFF
xb_AI	Association Indication - Read information regarding last node join request. This status monitors the progress of the association process. The following return values may be seen: • 0x00=successful completion: coordinator started a network • 0xAB=attempted to join a device that did not respond • 0xAC=secure join error, network security key received unsecured • 0xAD=secure join error, network security key not received • 0xAF=secure join error, joining device does not have the correct preconfigured link key • 0x21=scan found no PANs • 0x22=scan found no valid PANs based on current SC and ID settings. • 0x23=valid coordinator or router found but they are not allowing joining: node join time (NJ) expired • 0x27=node's attempt to join a network failed; this is typically due to incompatible security settings • 0x2A=coordinator start attempt failed • 0x2B=checking for an existing coordinator • 0xFF=scanning for a parent
RF Interfacing Commands	
xb_PL	Power Level. Select/read the RF transmit power level • XBee (boost mode disabled) 0 = -8 dBm 1 = -4 dBm 2 = -2 dBm 3 = 0 dBm 4 = +2 dBm • XBee-PRO 4 = 17 dBm

Table 5-1. **Commands to Send to XBee RF Modules**

Serial Interfacing Commands	
xb_BD	Read/write the baud rate for communication between the RF module serial port and host. 0 = 1200 bps 4 = 19200 bps 2 = 4800 bps 5 = 38400 bps 3 = 9600 bps 6 = 57600 bps 4 = 19200 bps 7 = 115200 bps Any value above 0x07 will be interpreted as an actual baud rate.
xb_NB	Read/write the serial parity setting on the module. • 0 = no parity • 1 = even parity • 2 = odd parity • 3 = mark parity
xb_RO	Read/write the packetization timeout value. This is the number of character times of inter-character silence required before packetization. Set (RO=0) to transmit characters as they arrive instead of buffering them into one RF packet.
I/O Commands	
xb_CB	Simulate a commissioning button press.
xb_D7	Select/read options for the DIO7 line of the RF module: • 0 = disabled • 1 = CTS flow control • 3 = digital input • 4 = digital output, low • 5 = digital output, high • 6 = RS-485 transmit enable (low enable) • 7 = RS-485 transmit enable (high enable)
xb_D6	Select/read options for the DIO6 line of the RF module: • 0 = disabled • 1 = RTS flow control
xb_D5	Configure options for the DIO5 line of the RF module: • 0 = disabled • 1 = associated indication LED • 3 = digital input • 4 = digital output, low • 5 = digital output, high

Table 5-1. Commands to Send to XBee RF Modules

xb_D4	Select/read function for DIO4: • 0 = disabled • 3 = digital input • 4 = digital output, low • 5 = digital output, high
xb_D3	Select/read function for AD3/DIO3: • 0 = disabled • 2 = analog input, singled ended • 3 = digital input • 4 = digital output, low • 5 = digital output, high
xb_D2	Select/read function for AD2/DIO2: • 0 = disabled • 2 = analog input, singled ended • 3 = digital input • 4 = digital output, low • 5 = digital output, high
xb_D1	Select/read function for AD1/DIO1: • 0 = disabled • 2 = analog input, singled ended • 3 = digital input • 4 = digital output, low • 5 = digital output, high
xb_D0	Select/read function for AD0/DIO0: • 0 = disabled • 1 = node identification button enabled • 2 = analog input, singled ended • 3 = digital input • 4 = digital output, low • 5 = digital output, high
xb_P0	DIO_10 Configure Select/read function for PWM0: • 0 = disabled • 1 = RSSI PWM • 3 = digital input, monitored • 4 = digital output, low • 5 = digital output, high

Table 5-1. Commands to Send to XBee RF Modules

xb_P1	DIO_11 Configure Configure options for the DIO11 line of the RF module: • 0 = unmonitored digital input • 3 = digital input, monitored • 4 = digital output, low • 5 = digital output, high
xb_P2	DIO_12 Configure Configure options for the DIO12 line of the RF module: • 0 = unmonitored digital input • 3 = digital input, monitored • 4 = digital output, low • 5 = digital output, high
xb_IS	Forces a read of all enabled digital and analog input lines.
xb_PR	Pull-up Resistor Enabled Set/read the bit field that configures the internal pull-up resistor status for the RF module I/O lines. “1” specifies the pull-up resistor is enabled; “0” specifies no pull-up. The following list identifies bit positions (0-12) and their corresponding pin definitions: • 0 = DIO4 (pin 11) • 1 = AD3/DIO3 (pin 17) • 2 = AD2/DIO2 (pin 18) • 3 = AD1/DIO1 (pin 19) • 4 = AD0/DIO0 (pin 20) • 5 = RTS/DIO6 (pin 16) • 6 = DTR/Sleep Request/DIO8 (pin 9) • 7 = DIN/Config (pin 3) • 8 = Associate/DIO5 (pin 15) • 9 = On Sleep/DIO9 (pin13) • 10 = DIO12 (pin 4) • 11 = PWM0/RSSI/DIO10 (pin 6) • 12 = PWM1/DIO11 (pin7)

Table 5-1. Commands to Send to XBee RF Modules

Sleep Commands	
xb_SM	Sets the sleep mode on the RF module: • 0 = sleep disabled • 1 = pin sleep enabled • 4 = cyclic sleep enabled
xb_SP	This value determines how long the end device will sleep at one time, up to 28 seconds. The sleep time can be extended past 28 seconds using the SN command. On the parent, this value determines how long the parent will buffer a message for the sleeping end device. It should be set at least equal to the longest SP time of any child end device.
xb_ST	Sets the time-before-sleep timer on an end device. This timer is reset each time serial or RF data is received. Once the timer expires, an end device may enter low power operation. This timer is available for cyclic sleep end devices only.
xb_WH	Time before receiving first packet after wake up
xb_SN	Number of Sleep Periods. Sets the number of sleep periods to not assert the On/Sleep pin on wakeup if no RF data is waiting for the end device. This command allows a device to sleep for an extended time if no RF data is present.
xb_SO	Configure sleep options. Unused option bits should be set to 0. 0x02 = always wake for ST time after SN*SP time 0x04 = sleep entire SN * SP time
Diagnostics Commands	
xb_VR	Read firmware version of the RF module.
xb_HV	Read hardware version of the RF module.
xb_DD	Digi device type.
xb_RP	RSSI PWM Timer. This is the time an RSSI signal will be output after the last transmission. When RP=0xFF, output will always be on.
xb_DB	This command reports the received signal strength of the last received RF data packet. The DB command only indicates the signal strength of the last hop, thus it does not provide an accurate measurement for a multihop link. DB can be set to 0 to clear it.
xb_PCV	Supply voltage
AT Command Options Commands	
xb_AC	Apply changes
XBee Endpoint Definition Commands	
xb_AO	Receive message format options

xb_send_command

```
int xb_send_command( word cmd );
```

DESCRIPTION

Send an AT command to the XBee. Same behavior as reading an XBee register and ignoring the result.

This function was introduced in Dynamic C 10.44.

NOTE: `xb_send_command` is a macro for `xb_get_register()`.

PARAMETER

cmd	Command to send, as a 16-bit word (see `xb_XX` definitions in Table 5-1 for values to use). Also possible to use *(word *)"XX" for reading the ATXX register. Examples: • `xb_ND` for node discovery • `xb_WR` to write settings to non-volatile RAM • `xb_FR` to software reset the XBee module.

RETURN VALUE

0: Success
`-ZBERR_AT_CMD_RESP_STATUS`: Radio returned failure
`-ETIME`: Timeout
`-EIO`: Serial I/O error
`-ENOSPC`: Output buffer full

LIBRARY

`xbee_api.lib`

xb_set_register

```
int xb_set_register( word reg, unsigned long value );
```

DESCRIPTION

Set an XBee register (up to 32 bits).

This function was introduced in Dynamic C 10.44.

PARAMETERS

reg	Register to write, as a 16-bit word (see `xb_XX` definitions in Table 5-1 for values to use). Also possible to use *(word *)"XX" for reading the ATXX register. Examples: `xb_SC` to set list of channels to scan, `xb_D0` to set the I/O mode for DIO0.
value	Value to store in register (Parameter1).

RETURN VALUE

0: Success
`-ZBERR_AT_CMD_RESP_STATUS`: Radio returned failure, possibly due to setting an out-of-range value (e.g., 16-bit value in an 8-bit register).
`-ETIME`: Timeout
`-EIO`: Serial I/O error
`-ENOSPC`: Output buffer full

LIBRARY

`xbee_api.lib`

xb_sleep

```
int xb_sleep( word st, word sp, word sn, word so, word sm );
```

DESCRIPTION

Set the parameters that control the sleep mode, using the default XBee parameters. For more details refer to the manual *XBee® / XBee-PRO® ZB OEM RF Modules* available at: www.digi.com.

This function was introduced in Dynamic C 10.44 to replace the deprecated function `zb_Rabbit_Sleep()`.

PARAMETERS

st Time Before Sleep (in milliseconds).

st controls how long the Rabbit module will stay awake. The minimum value shown below exists because this is the time that the module takes to become fully operational. If st is set to a value smaller than 2000, the module will be go back to sleep before it has the chance to run its code. The minimum value for st is `ZB_MIN_ST_TIME` (0x07D0, 2.000 secs) The maximum value for st is `ZB_MAX_ST_TIME` (0xFFFE, 65.534 secs)

sp Cyclic Sleep Period (in 1/100 seconds).

sp controls how often the XBee module will wake up and check for new frames (end devices) or how long the XBee will buffer frames for sleeping end devices (routers and coordinators).
The minimum value for sp is `ZB_MIN_SP_TIME` (0x0020, 0.32 seconds)
The maximum value for sp is `ZB_MAX_SP_TIME` (0x0AF0, 28.00 seconds)

sn Sleep Time Extender. This parameter is used in cases where we want the Rabbit module to stay asleep for periods longer than the sp cyclic sleep period. The Rabbit module will remain asleep for a period of sn * sp. However, the radio will wake up briefly every time sp expires and it will poll its parent for messages. If there is a message waiting, the radio wakes up the Rabbit module and forwards the message. If there is no message, the radio goes to sleep again for another sp period. sn can be any non-zero value.

so Sleep Options. This parameter is a bitmask made up of the following options. Read the XBee Module manual for more details on these settings.

- `XB_SO_WAKE_ST`
- `XB_SO_SLEEP_SNxSP`

sm Sleep Mode. This parameter is a bitmask made up of the following options. On Rabbit products, the sleep pin is not connected so `XB_SM_CYCLIC` is the only useful option.

- `XB_SM_DISABLED`
- `XB_SM_PINWAKE`
- `XB_SM_CYCLIC`

Warning: On XBee end devices with ZNet firmware, `XB_SM_DISABLED` is considered an invalid parameter (and xb_sleep will return `-EINVAL`). With ZNet, setting the sm parameter to `XB_SM_DISABLED` will cause the end device to become a router, perform a network reset and rejoin the network.

ORETURN VALUE

0: Successful, power will turn off after the ST timer expires.
`-EINVAL`: Invalid parameters
`-EOPNOTSUPP`: Operation not supported (on routers and coordinators)
`-ZBERR_AT_CMD_RESP_STATUS`: Couldn't set one of the sleep parameters
`-ETIME`: Timeout
`-EIO`: Serial I/O error
`-ENOSPC`: Output buffer full

LIBRARY

`xbee_api.lib`

xb_stayawake

```
int xb_stayawake ();
```

DESCRIPTION

This function is for end devices only. It sets the idle timeout (ST) to maximum value, and keeps the XBee awake by sending serial data to it periodically.

This function was introduced in Dynamic C 10.44.

RETURN VALUE

0: Successful
`-EOPNOTSUPP`: Operation not supported (on Routers and Coordinators)
`-ZBERR_AT_CMD_RESP_STATUS`: Could not set ST register
`-ETIME`: Timeout
`-EIO`: Serial I/O error
`-ENOSPC`: Output buffer full.

LIBRARY

`xbee_api.lib`

zb_adc_in

```
int zb_adc_in( int dio );
```

DESCRIPTION

Read the analog input pin on the local XBee using AT commands. Return the 10-bit value. To convert the reading to millivolts perform the following calculation:

```
AD(mV) = (ADIO reading * 1200mV / 1023)
```

Note: The pin number is NOT the same as the `DIO_xx` macros, which are for configuring the function of the pin only.

PARAMETER

`dio`	Analog input pin to read. This value corresponds to the DIO number. Valid analog DIO values range from 0 to 3.

RETURN VALUE

0-1023: valid data
`-EINVAL`: invalid pin

LIBRARY

`xbee_api.lib`

zb_API_ATCmdResponse

```
int zb_API_ATCmdResponse ( char * _cmdstr, void * data, int dlen,
    _at_cmdresp_t * resp );
```

DESCRIPTION

Send an API AT command and get the response.

PARAMETERS

_cmdstr	pointer to command string: we expect "ATcc", so we skip the first two bytes for API format commands
data	address of data to send
dlen	length of data to send
resp	address of AT response buffer

RETURN VALUE

0: success
-ZBERR_AT_CMD_RESP_STATUS: Radio returned failure
-ETIME: Timeout
-EIO: Serial I/O error
-ENOSPC: Output buffer full

LIBRARY

xbee_api.lib

zb_check_sleep_params

```
int zb_check_sleep_params ( unsigned int st, unsigned int sp,
   unsigned int sn );
```

DESCRIPTION

This is an auxiliary function that checks if the main sleep mode parameters are within valid limits. For more details on the parameters, refer to the description for `xb_sleep()`.

PARAMETERS

st	Time Before Sleep
sp	Cyclic Sleep Period
sn	Sleep Time Extender

RETURN VALUE

0: successful, all parameters are valid
`-EINVAL`: invalid parameters

LIBRARY

`xbee_api.lib`

zb_dio_in

```
int zb_dio_in ( int dio );
```

DESCRIPTION

Read a digital input pin on the XBee module.

PARAMETER

dio	The pin number (0-`ZB_MAX_PIN`). This value corresponds to the DIO number. Valid digital DIO values are 0-5, 10, 11, and 12. **Note**: The pin number is NOT the same as the `DIO_xx` macros. Those are for configuring the function of the pin only.

RETURN VALUE

0 or 1: valid data
`-EINVAL`: invalid pin

LIBRARY

`xbee_api.lib`

zb_dio_out

```
int zb_dio_out ( int dio, int value );
```

DESCRIPTION

Set the digital output value on the XBee module pin.

PARAMETERS

dio	The pin number (0-`ZB_MAX_PIN`). This value corresponds to the DIO number. Valid digital DIO values are 0-5, 10, 11, and 12. Note: The pin number is NOT the same as the DIO_xx macros. Those are for configuring the function of the pin only.
value	The value of the pin (1 or 0).

RETURN VALUE

0: successful
`-EINVAL`: invalid pin number

LIBRARY

`xbee_api.lib`

ZB_ERROR

int ZB_ERROR()

DESCRIPTION

Returns the last error encountered. This is a macro to access the error code variable of the `XBee_API.lib` library.

LIBRARY

`xbee_api.lib`

ZB_GENERAL_MESSAGE_HANDLER

#define ZB_GENERAL_MESSAGE_HANDLER <functionName>

DESCRIPTION

Define the general message handler for messages that do not have endpoints or other addressing means specified. The general message handler callback function prototype must be:

```
int functionName (char *data);
```

"data" points to the message data. To get more data about the message call `zb_receive()`. This will give you access to any addressing information that was received.

To reply to this message directly use `zb_reply()` before another message arrives. To assure that no messages arrive before you have replied do not call `xbee_tick()` until your reply has been sent (i.e., `zb_reply()` or `zb_send()` has been called).

If the message cannot be handled by the general message handler it may return non-zero, and the `xbee_tick()` function will indicate that a message is available. You may access the message using `zb_receive()` and handle it then.

Return a zero to indicate that the message has been completely processed. `xbee_tick()` will then not indicate that a message is available.

LIBRARY

`xbee_api.lib`

zb_getATCmdResponse

```
int zb_getATCmdResponse ( _at_cmdresp_t * buffer, int blen );
```

DESCRIPTION

Wait (i.e., block) for a response to the current AT command.

PARAMETERS

buffer	Pointer to where to put the response
blen	Size of buffer (deprecated starting with Dynamic C 10.46)

RETURN VALUE

0: success
-ETIME: timeout

_zb_error will be set to the same value as the return value.

LIBRARY

xbee_api.lib

zb_io_init

```
int zb_io_init ( void );
```

DESCRIPTION

Initializes the I/O subsystem on the XBee module. The default behavior for each pin is predefined but can be overridden prior to #using xbee_api.lib by #defining the appropriate macro:

```
#define DIO_00    XBEE_IO_CONF_ANAIN
#define DIO_05    XBEE_IO_CONF_DIGOUT_HIGH
#define DIO_10    XBEE_IO_CONF_DIGOUT_LOW
#define DIO_12    XBEE_IO_CONF_DIGIN

#use xbee_api.lib
```

The DIO_xx defines automatically generate four additional macros for `zb_io_init()`: `DIO_INPUTS`, `DIO_OUTPUTS`, `AIO_INPUTS`, and `DIO_PULLEDUP`.

`DIO_PULLEDUP` sets which pins are pulled up. By default, all input pins are pulled up, but this can be overridden by #defining `DIO_PULLEDUP` prior to #using xbee_api.lib (note: the mask order for `DIO_PULLEDUP` is different than for `DIO_INPUTS`, see below).

While all pins can be configured as either inputs or outputs, only `DIO_00` - `DIO_03` can be configured as analog inputs.

The default configuration is:

Table 5-2.

Name	Function	State	Pull-up Mask Bit
DIO_00	XBEE_IO_CONF_ANAIN	N/A	4
DIO_01	XBEE_IO_CONF_ANAIN	N/A	3
DIO_02	XBEE_IO_CONF_ANAIN	N/A	2
DIO_03	XBEE_IO_CONF_ANAIN	N/A	1
DIO_04	XBEE_IO_CONF_DIGOUT_HIGH	N/A	0
DIO_05	XBEE_IO_CONF_DIGOUT_HIGH	N/A	8
DIO_10	XBEE_IO_CONF_DIGOUT_LOW	N/A	11
DIO_11	XBEE_IO_CONF_DIGIN	PULLED UP 30K	12
DIO_12	XBEE_IO_CONF_DIGIN	PULLED UP 30K	10

On the RCM45xxW, the DIO_xx signals map to AUX I/O header (J4 on the core module) as follows:

```
            AUX I/O (J4) Header Pins
            ----------------------------
                        1  2
          DIO_00 --- oo --- DIO_01
          DIO_02 --- oo --- DIO_03
             GND --- oo --- DIO_12
   No Connection --- oo --- +3.3V
          SysPwr --- oo --- DIO_05
          DIO_04 --- oo --- No Connection
          DIO_10 --- oo --- DIO_11
                       13  14
```

RETURN VALUE

0: success

!=0: the error code returned by a call to `zb_API_ATCmdResponse()`

LIBRARY

`xbee_api.lib`

SEE ALSO

`zb_API_ATCmdResponse()` for error codes

ZB_LATEST_MESSAGE

`api_frame_t *ZB_LATEST_MESSAGE()`

DESCRIPTION

This macro gives the address of the `api_frame_t` structure holding the last message received.

LIBRARY

`xbee_api.lib`

SEE ALSO

`ZB_LAST_MSG_DATA()`, `ZB_LAST_MSG_DATALEN()`

ZB_LAST_MSG_DATA

`char *ZB_LAST_MSG_DATA()`

DESCRIPTION

This macro points to the RF payload of the last frame received.

LIBRARY

`xbee_api.lib`

SEE ALSO

`ZB_LATEST_MESSAGE()`, `ZB_LAST_MSG_DATALEN()`

ZB_LAST_MSG_DATALEN

`int ZB_LAST_MSG_DATALEN()`

DESCRIPTION

This macro is set to the size of the RF payload of the last frame received.

LIBRARY

`xbee_api.lib`

SEE ALSO

`ZB_LATEST_MESSAGE()`, `ZB_LAST_MSG_DATA()`

ZB_LAST_STATUS

```
int ZB_LAST_STATUS();
```

DESCRIPTION

This macro returns the RF module status of this node. See the defined constants below for values and their meanings.

- `ZB_HARDWARE_RESET`, module is performing a hardware reset
- `ZB_WATCHDOG_RESET`, module is resetting from a watchdog timeout
- `ZB_JOINED`, module has joined a network
- `ZB_UNJOINED`, module has left (unjoined) a network
- `ZB_COORD_STARTED`, coordinator started

Additionally, the macro `ZB_JOINING_NETWORK` returns TRUE (1) when `ZB_LAST_STATUS() != ZB_JOINED`. You can use a statement like:

```
while (ZB_JOINING_NETWORK()) {}
```

to check if the arbitrary maximum time to join has expired. If it did, process the timeout error condition.

LIBRARY

`xbee_api.lib`

zb_MakeEndpointClusterAddr

```
zb_sendAddress_t * zb_MakeEndpointClusterAddr ( int node, int srcEP,
   int destEP, word clusterID, zb_sendAddress_t * addr );
```

DESCRIPTION

Fill in the address structure for sending to `zb_send()`.

PARAMETERS

node	Index into the node lookup table. Sending -1 indicates a broadcast address.
srcEP	Source (sending) endpoint.
destEP	Destination endpoint.
clusterID	Destination cluster ID.
addr	Buffer to put the address data into.

RETURN VALUE

A pointer to the address buffer.
A NULL return value means that either:
- the node value is out of range
- a valid node entry was not found

LIBRARY

`xbee_api.lib`

zb_MakeIEEENetworkAddr

```
zb_sendAddress_t * zb_MakeIEEENetworkAddr ( int node,
   zb_sendAddress_t * buffer );
```

DESCRIPTION

Create an address for `zb_send()` consisting of the 8-byte IEEE address and the 2-byte network address.

PARAMETERS

node	Index into the node lookup table. Sending -1 indicates a broadcast address.
buffer	Where to put the address data.

RETURN VALUE

A pointer to the address buffer.
A NULL return value means that either:
- the node value is out of range
- a valid node entry was not found

LIBRARY

`xbee_api.lib`

zb_missed_messages

int zb_missed_messages (void);

DESCRIPTION

Return the number of missed messages since the last time `zb_receive()` was called.

RETURN VALUE

Count of missed messages.

LIBRARY

xbee_api.lib

ZB_ND_RUNNING

DESCRIPTION

This macro tracks when a node is currently performing a Node Discovery (ND) command. Be aware that this macro calls `xbee_tick()`, which will then silently drop any received packets.

It is typically used in the following way:

```
printf("Waiting for node discovery...\n");
while (ZB_ND_RUNNING());
printf("Discovery done.\n\n");
```

Prior to Dynamic C 10.44, no other AT command could be sent to the radio while performing node discovery.

RETURN VALUE

TRUE: node discovery in progress
FALSE: node discovery not in progress

LIBRARY

xbee_api.lib

zb_Rabbit_poweroff

```
int zb_Rabbit_poweroff( _zb_power_control_t * zbp );
```

DESCRIPTION

Send the wake-up parameters to the Radio and instruct it to power-down the Rabbit.

This function is for use on the RCM45xxW only.

PARAMETER

zbp Wake-up parameter structure address. The `_zb_power_control_t` structure is defined as follows:

```
typedef struct {
   char wakeFlag;          // See below for valid flag values
   int digIOMask;          // bits = digital I/O pins on RF module
                           //    0 - ignore this I/O pin
                           //    1 - wake on change to this pin
   char anaChannelMask;    // bits = analog I/O channels on RF module
                           //    0: ignore this channel
                           //    1: wake on level reached for this channel
   int  anaChannelLevels[XBEE_ANA_CHANNELS];
                        // wake Rabbit when read value exceeds this level
                        // if the level is negative, the read value must
                        //       be <= the level.
                        //  if the level is positive, the read value must
                        //       be >= the level.
   long time_in_ms;     // Length of time (millisecs) for Rabbit to sleep
                        // The actual sleep time will be
                        // (int)(radio_duty * (time_in_ms / radio_duty))
   int  radio_duty;     // Sleep time in 1/100 seconds
                        // Please note that the sleep time will determine
                        // analog read interval if "wake on analog level"
                        // is active. The max value for the radio sleep
                        // cycle is 28 seconds (0x0Af0)
} _zb_power_control_t;
```

RETURN VALUE

0: successful, power will turn off after the ST timer[1] expires (`ZB_MIN_ST_TIME`).
`-EINVAL`: invalid parameters
`-EOPNOTSUPP`: operation not supported (on Routers or Coordinators) or function was called while in debug mode.

LIBRARY

`xbee_api.lib`

1. By default, the AT command ST (time before sleep) is set to 65.534 seconds

zb_Rabbit_Sleep

```
zb_Rabbit_Sleep ( unsigned st, unsigned sp, unsigned sn );
```

DESCRIPTION

Set the parameters that control the sleep mode using the default XBee parameters.

As of Dynamic C 10.44, this function has been deprecated in favor of `xb_sleep()`, which includes parametersfor setting the XBee SO and SM registers.

PARAMETERS

st Time Before Sleep (in milliseconds): this timer must be set in msecs.

st controls how long the Rabbit module will stay awake.The minimum value shown below exists because this is the time that the module takes to become fully operational. If st is set to a value smaller than 2000, the module will go back to sleep before it has the chance to run its code.

- Minimum value for st is `ZB_MIN_ST_TIME` (0x07D0, 2 seconds)
- Maximum value for st is `ZB_MAX_ST_TIME` (0xFFFE, 65.534 seconds)

sp Cyclic Sleep Period (in 1/100 seconds)

sp controls how often the XBee module will wake up and check for new frames (end devices) or how long the XBee will buffer frames for sleeping end devices (routers and coordinators).

- Minimum value for sp is `ZB_MIN_SP_TIME` (0x0020, 0.32 secs)
- Maximum value for sp is `ZB_MAX_SP_TIME` (0x0AF0, 28.00 secs)

sn Sleep Time Extender:

sn is used in cases where we want the Rabbit module to stay asleep for periods longer than the sp time. The Rabbit module will remain asleep for a period of sn * sp. However, the radio will wake up briefly every time sp expires and it will poll its parent for messages.

If there is a message waiting, the radio wakes up the Rabbit module and forwards the message. If there is no message the radio goes to sleep again for another sp period. sn can be any non-zero value.

RETURN VALUE

0: Successful, power will turn off after the st timer expires.
`-EINVAL`: invalid parameters.
`-EOPNOTSUPP`: operation not supported (on routers or coordinators) or function was called while in debug mode (on RCM45xxW).

LIBRARY

`xbee_api.lib`

zb_receive

```
api_frame_t * zb_receive ( char * data, int * len );
```

DESCRIPTION

This function should be called when the `xbee_tick()` function indicates that a message is waiting to be handled.

The parameter to this routine will be the address of the buffer which will accept the new message. The buffer should be `XBEE_MAX_RFPAYLOAD` bytes long to ensure there is sufficient space to receive the data.

The function returns the address of the beginning of the entire message (a pointer to an `api_frame_t`). If the function returns NULL there was no message. The current message will be held until the next message arrives.

Note that no new messages will arrive until `xbee_tick()` is called.

PARAMETERS

`data`	Buffer to receive data. Send NULL to get only the length of `api_frame_t`.
`len`	Buffer to receive data length. Send NULL to get only the address of `api_frame_t`.

RETURN VALUE

NULL: no message was received
!=NULL: address of current message.

LIBRARY

`xbee_api.lib`

zb_reply

```
int zb_reply ( char * reply, int len );
```

DESCRIPTION

Send a reply to the last message received. This function uses the address of the last message's sender as the addressee of the reply. The reply will be sent using explicit addressing.

PARAMETERS

reply	Pointer to message to send.
len	Length of message.

RETURN VALUE

0: Queued, the message has been sent to the Radio for transmission.
-EINVAL: bad parameters
-ENOSPC: cannot give message to serial port

LIBRARY

xbee_api.lib

zb_send

```
int zb_send ( zb_sendAddress_t * addr );
```

DESCRIPTION

Send a message to other XBee radios. The addressing modes may be combined to more completely direct messages. If endpoints and cluster ID are not specified, they will be set to zero.

PARAMETER

addr	Pointer to the address data structure.

RETURN VALUE

0: Queued - the message has been sent to the Radio for transmission.
-EINVAL: Bad parameters.
-ENOSPC: Cannot give msg to serial port.
-ENONET: Radio has not joined a network.
-EIO: XBee holding CTS low, or serial write did not complete correctly.

LIBRARY

xbee_api.lib

SEE ALSO

zb_MakeIEEENetworkAddr(), zb_MakeEndpointClusterAddr()

zb_sendAPICmd

```
int zb_sendAPICmd ( int cmd, void * data, int len );
```

DESCRIPTION

Send a raw command frame using the XBee API format.

PARAMETERS

cmd	API frame identifier.
data	Pointer to data to send.
len	Length of data.

RETURN VALUE

0: Success.
`-ENOSPC`: Not enough buffer space to send command.
`-EIO`: XBee is holding CTS low, or serial write did not complete correctly.

SIDE EFFECTS: _zb_error will be set on error or successful message transmission.

LIBRARY

`xbee_api.lib`

zb_sendATCmd

```
int zb_sendATCmd ( char * _cmdstr, char * data, int dlen );
```

DESCRIPTION

Send an AT command to the XBee module without waiting for a response. This is useful when a command is expected to take too long to respond (>`_XBEE_TIMEOUT` milliseconds).

PARAMETERS

`_cmdstr`	Pointer to command text ("ATxx", "AT" is ignored and "xx" used for command)
`data`	Pointer to command parameters.
`dlen`	Length of command parameters.

RETURN VALUE

0: success
`-EIO`: serial I/O error
`-ENOSPC`: output buffer full

SIDE EFFECTS: `_zb_error` will be set on error or successful message transmission.

LIBRARY

`xbee_api.lib`

zb_swapBytes

```
int zb_swapBytes ( int * value );
```

DESCRIPTION

Swap the bytes of a word-sized (two-byte) value.

PARAMETER

value	Address of word to change.

RETURN VALUE

New value of “value”

SEE ALSO

htons (host to network short)

LIBRARY

xbee_api.lib

zb_tick

```
int zb_tick ( void );
```

DESCRIPTION

Drive the Rabbit XBee radio communications. Performs a Network Discovery once at initialization time.

NOTE: This function has been deprecated, please use xbee_tick() (which includes more return values and can return a copy of the last frame processed) instead.

RETURN VALUE

ZB_NOMESSAGE: no messages received
ZB_MESSAGE: a message has arrived
ZB_RADIO_STAT: radio status change
ZB_MSG_STAT: message transmission status available
-ENOMEM: out of memory processing node discovery response
(various codes)<0: an error has occurred

LIBRARY

xbee_api.lib

ZB_XMIT_OVERHEAD

```
int ZB_XMIT_OVERHEAD( void );
```

DESCRIPTION

This macro returns the overhead required to send the last message.

RETURN VALUE

NO_DISC_OVERHEAD
ADDR_DISCOVERY
ROUTE_DISCOVERY
ADDR_AND_ROUTE

ZB_XMIT_STATUS

```
int ZB_XMIT_STATUS( void );
```

DESCRIPTION

This macro return the status of the last transmission sent by this software.

Under normal operation an application will be mostly concerned with ADDR_NOT_FOUND or ROUTE_NOT_FOUND. The first status would indicate that the device in question has shut down or moved out of range. The second status would indicate that some intermediate nodes that provided the route to the wanted device have shut down.

RETURN VALUE

DELIVERY_SUCCESS
CCA_FAILURE
BAD_DEST_ENDPOINT
NET_ACK_FAILURE
NOT_JOINED
SELF_ADDRESSED
ADDR_NOT_FOUND
ROUTE_NOT_FOUND

zigbee_init (deprecated)

```
int zigbee_init ( void );
```

DESCRIPTION

Initialize the Rabbit XBee driver and the XBee radio.

This function was deprecated in Dynamic C 10.40.

RETURN VALUE

0: successful
!=0: failure

See _zb_error for the specific error code. The values for _zb_error are defined in /Lib/../ERRNO.LIB relative to the Dynamic C installation folder.

LIBRARY

xbee_api.lib

5.4 Protocol Firmware

ZigBee-capable Rabbit-based boards must be programmed with the appropriate RF module firmware. There are two sets of firmware to consider: one for ZNet 2.5 and one for ZigBee PRO (ZB).

ZNet 2.5 firmware is only supported on RCM4510W core modules.

5.4.1 Updating RF Module FW on a Rabbit-Based Target

To update the protocol firmware on the XBee RF module housed on the Rabbit-based target board, use the Dynamic C sample program `\Samples\XBee\ModemFWLoad.c` located relative to the Dynamic C installlation folder.

The instructions at the top of `ModemFWLoad.c` explain the two macros you need to set in order to download the correct firmware. They are:

- `XBEE_PROTOCOL` - this macro must be #defined to `XBEE_PROTOCOL_ZB` or `ZBEE_PROTOCOL_ZNET`, depending on the protocol desired.
- `XBEE_ROLE`[1] - this macro must be #defined to `NODE_TYPE_COORD`, `NODE_TYPE_ROUTER` or `NODE_TYPE_ENDDEV`, depending on the ZigBee device type desired. The supported protocols (ZNet and ZB) have different versions of firmware for each ZigBee device type.

To change these macros from their default values (`NODE_TYPE_ROUTER` and `XBEE_PROTOCOL_ZB`) find their #define statements at the top of `ModemFWLoad.c` and modify the program code.

5.4.2 X-CTU: Updating RF Module FW on a DIGI XBee USB Device

A utility program, X-CTU, is provided for reading and writing the firmware on the Digi XBee USB device. The utility is described in the following subsections.

5.4.2.1 X-CTU Installation

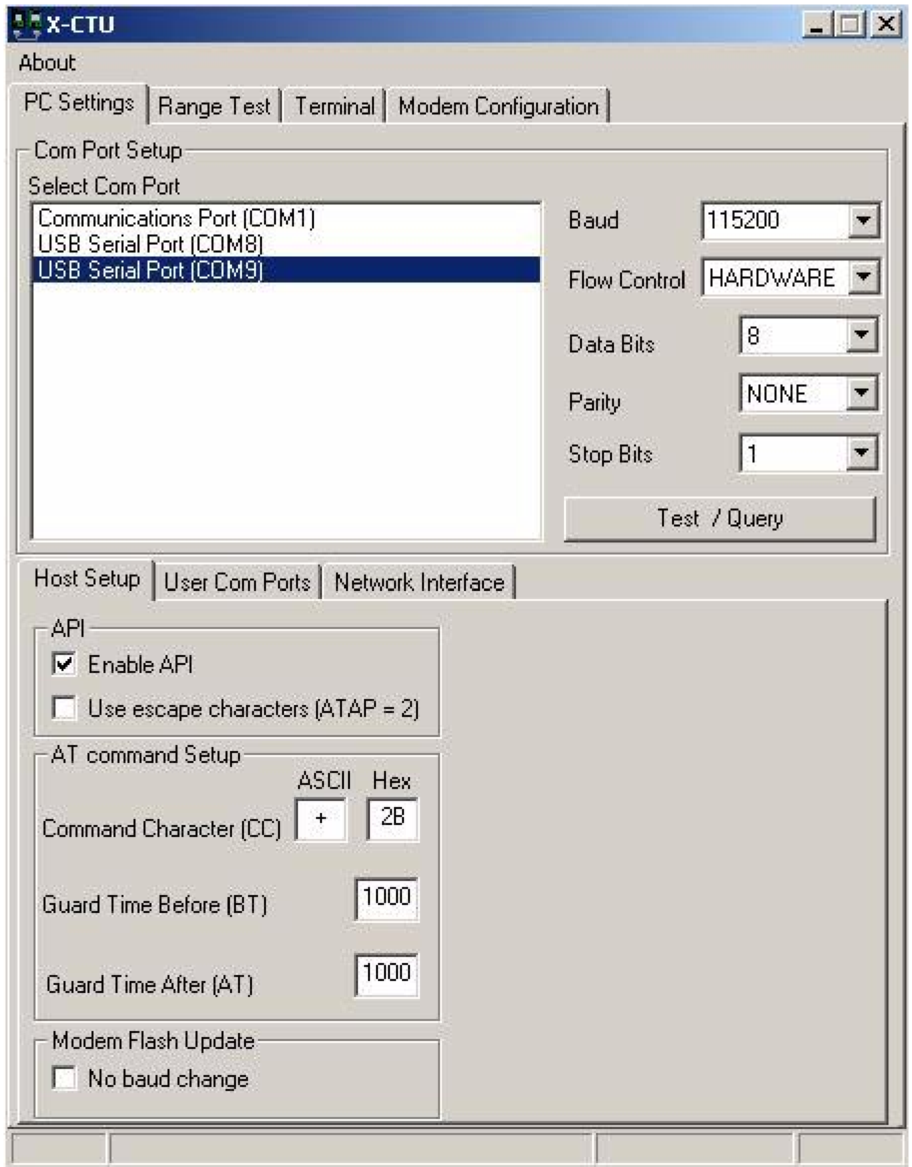
To install the X-CTU utility, run its setup program, `Setup_XCTU_XXXX.exe,` found in `\Utilities\X-CTU` relative to the Dynamic C installation folder. The installation process will create a desktop icon for the utility.

1. Prior to Dynamic C 10.40, the configuration macros ZIGBEE_COORDINATOR, ZIGBEE_ROUTER and ZIGBEE_ENDDEV were used instead of XBEE_ROLE.

5.4.2.2 PC Settings Tab

Click on the X-CTU icon to run the utility. The PC Settings tab is the first screen displayed when it starts. Before doing anything else, check the box labeled “Enable API”. This box may be unchecked by default, but it must be checked because only the API mode of communication is supported.

Figure 5.6 Opening Screen of X-CTU



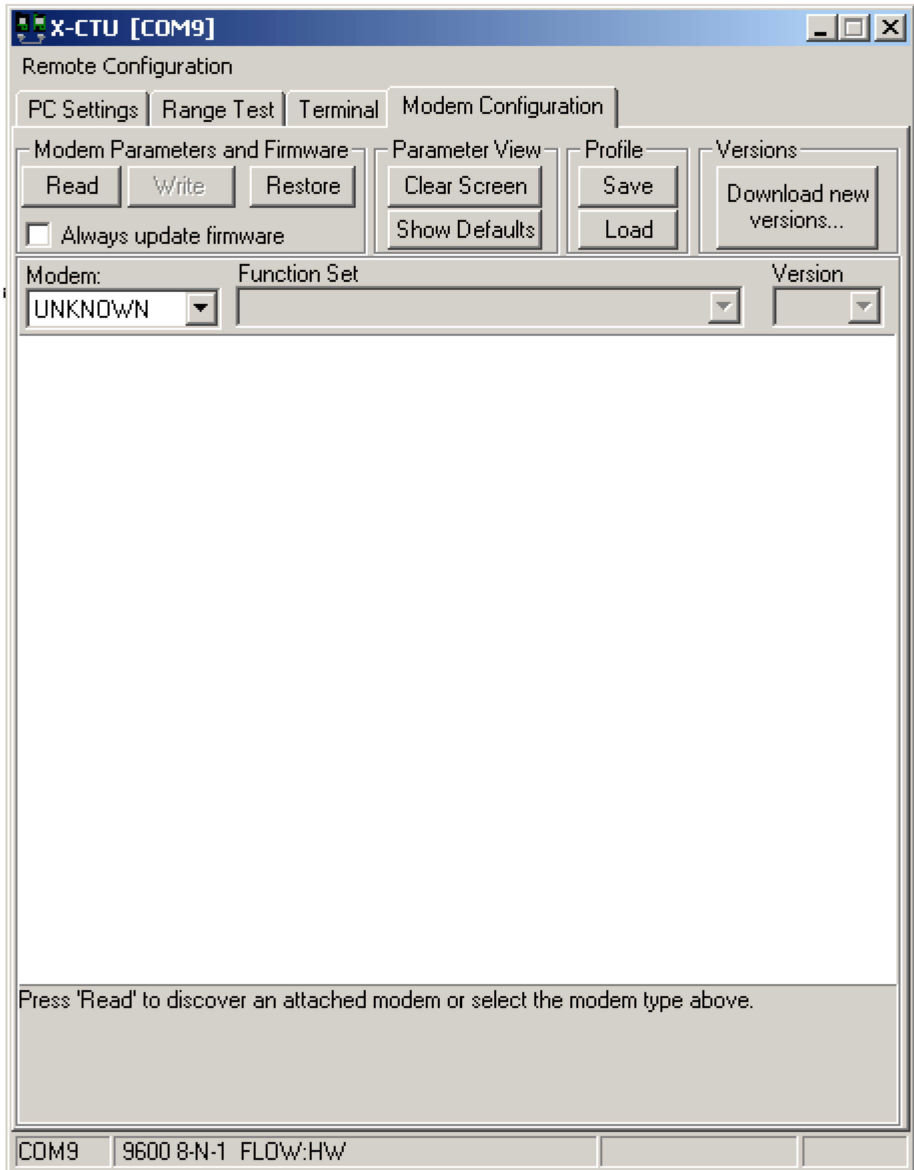
You can change serial parameters (baud, etc.) and test/query the COM port high-lighted in the “Select COM Port” window to test the connection between X-CTU and the selected COM port.

In Figure 5.6, the values shown on the “Host Setup” tab at the bottom half of the “PC Settings” tab are the default values. Leave them as is. To update the firmware on the DIGI XBee USB device, select the COM port that the device is connected to, in this case, COM9. Follow the instructions given in the next section.

5.4.2.3 Modem Configuration Tab

Click on the Modem Configuration tab. A screen similar to the one shown in Figure 5.7 will be displayed.

Figure 5.7 "Modem Configuration" Tab Default Screen



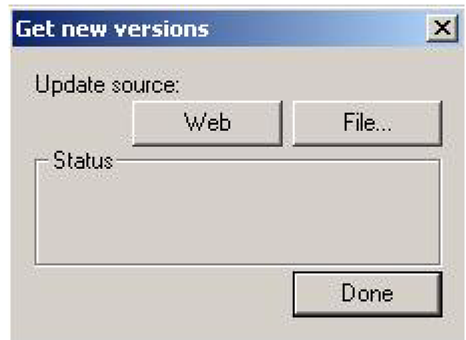
Click on the button labeled "Read". This will cause the firmware that is loaded onto the DIGI XBee USB device to be read and its parameters displayed, as shown in Figure 5.8.

Do not click on the button labeled "Restore" unless you want to write default parameter values for the current firmware version to the device; for example., the PAN ID and Node Identifier will be zeroed out. Also, do not check the box labeled "Always update firmware." If this box is checked, the entire firmware will be reloaded when you attempt to write network parameters. When the entire firmware is reloaded to the device, it takes significantly more time than just writing network parameters. In addition, reloading the firmware will cause a network reset.

5.4.2.3.1 Selecting the Firmware

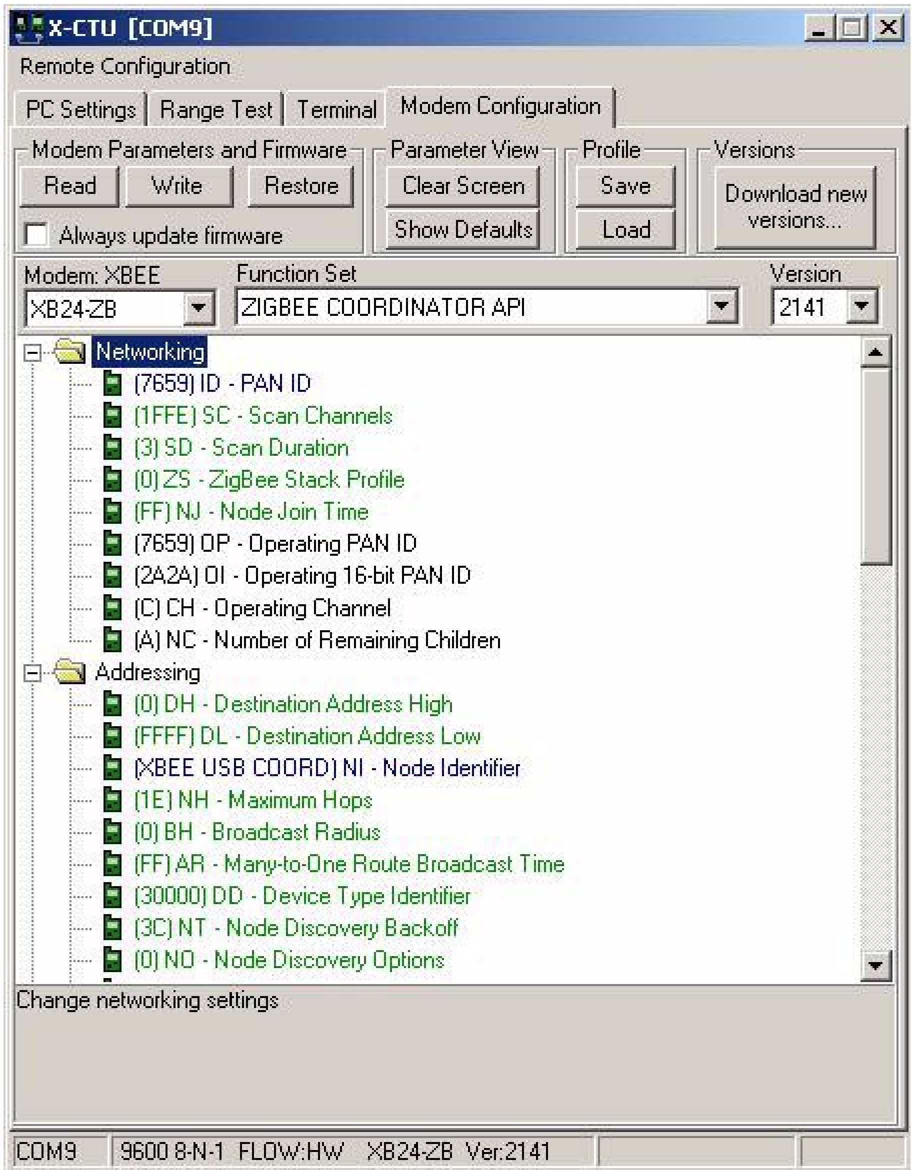
The buttons “Save” and “Load” under the label “Profile” allow use of files ending in .pro that contain firmware configuration settings. To load the default configuration values for the firmware, navigate to `Utilities/X-CTU` and select `XBee-USB ZB defaults.pro`. These default values will not take effect until they are written to the Digi XBee USB device by clicking on the “Write” button.

Three drop-down menus labeled “Modem:”, “Function Set” and “Version” are available for selecting the firmware to write to the DIGI XBee USB device. If the firmware version you want is not included in these drop-down menus, click on the button labeled “Download new versions...”. Click on “File...” in the resulting popup box:



Then browse to the Dynamic C installation folder. From there go to the folder /Utilities/X-CTU/MODEMFW/ to select the .zip file containing the desired firmware. After X-CTU’s firmware list has been updated, make your selection using the three drop-down menus. Then click on the button labeled “Write” in order to load the selected firmware onto the device. Writing new firmware always causes a network reset.

Figure 5.8 Loading FW onto the DIGI XBee USB Device



5.4.2.3.2 Modem Parameters

The parameters read from the device are described in Table 5-3. The available AT commands and their categories differ slightly between firmware versions. The following table reflects a read of firmware XB24-ZB version 2141, with the default configuration values from `Utilities/X-CTU/XBee-USB ZB defaults.pro`. The mnemonic for the AT command is given, followed by its default value in parenthesis.

Table 5-3. AT Commands

Networking Commands	
ID (0123456789ABCDEF)	PAN ID - Set/read the 64-bit extended PAN ID.If set to 0, the coordinator will select a random extended PAN ID; routers and end devices will join any PAN.
SC (0x1FFE)	Scan Channels - This is a bit field of channels to scan. ZigBee channels do not start at 0: bit 0 = channel 0xB; bit 15 = channel 0x1A. The default value of 0x0x1FFE allows all available channels to be scanned.
SD (3)	Scan Duration - For a coordinator, this exponent is used to determine an acceptable channel and PAN ID to start a network; for a router/end device, this exponent is used to locate an available coordinator or router to join.
ZS (0)	ZigBee Stack Profile -This value must be set to the same values on all devices in the same network.
NJ (0xFF)	Node Join Time - The time in seconds that this coordinator or router will allow other nodes to join it. If set to 0xFF, the coordinator or router will always allow joining. Changing this value does not cause a network reset. The time is reset whenever the device power cycles or when NJ changes.
OP (read only)	Operating Extended PAN ID - This command reads the 64-bit extended PAN ID that the module is running on. If ID>0, then OP=ID.
OI	Operating 16-bit PAN ID.
CH (read only)	Operating Channel - Read the channel number being used. A value of 0 indicates that the node has not joined a network, i.e., it is not operating on any channel. Valid channels are: • 0x0B - 0x1A, for XBee • 0x0B - 0x18, for XBee PRO
NC (read only)	Number of Remaining Children - Read the number of end device children than can join the device. The range is 0-10. If NC returns 0, then the device cannot allow any more end device children to join.
Addressing Commands	
DH (0)	Destination Address High - Not supported in API mode.
DL (0xFFFF)	Destination Address Low - Not supported in API mode.
NI (DIGI-XBEE-USB)	Node Identifier - String that identifies the node. This string is returned as part of the ND (Node Discover) command. NI is also used with the DN (Destination Node) command.

Table 5-3. AT Commands

NH (0x1E)	Maximum Unicast Hops - Set/read the max hops limit. This limit sets the maximum broadcast hops value (BH) and determines the unicast timeout. The timeout is computed as (50*NH) + 100 ms. The default unicast timeout of 1.6 seconds is enough time for data and the ack to traverse about 8 hops.
BH (0)	Broadcast Radius - Set/read the maximum number of hops for each broadcast data transmission. The default value of 0 uses the maximum number of hops.
AR (0xFF)	Many-to-One Route Broadcast Time (Aggregate Routing Notification) - Set/read time between consecutive aggregate route broadcast messages. AR should be set on only one device to enable many-to-one routing to the device. Setting AR to 0 sends only one broadcast.
DD (0x30000)	Device Type Identifier - Stores a device type value. This value can be used to differentiate multiple XBee-based products. Valid range is 0x0 - 0xFFFFFFFF.
NT (0x3C)	Node Discovery Timeout - Set/Read the amount of time a node will spend discovering other nodes when ND or DN is issued. Valid range is 0x20 - 0xFF [*100ms].
NO (0)	Node Discovery Options - Set/read the bitfield that defines options for the Network Discovery (ND) command. • 0x01= append DD value to ND responses or API node identification frames • 0x02=local device sends ND response when ND is issued
SH (factory set)	Serial Number High - Read high 32 bits of the RF module's unique IEEE 64-bit address.
SL (factory set)	Serial Number Low - Read low 32 bits of the RF module's unique IEEE 64-bit address.
MY (read-only)	16-bit Network Address - The address the device received when joining the network. For the coordinator this is 0.
NP (read-only)	Maximum Number of RF Payload Bytes - Returns the maximum number of RF payload bytes that can be sent in a unicast transmission.
RF Interfacing	
PL (4)	Power Level - Select/read the power level at which the RF module transmits conducted power.
PM (1)	Power Mode - Set/read the power mode of the device. Enabling boost mode improves receive sensitivity by 1dB and increases transmit power by 2dB. • 0=boost mode enabled • 1=boost mode disabled

Table 5-3. AT Commands

Security Commands	
EE (0)	Encryption Enable - Set/read the encryption enable setting • 0=encryption disabled • 1=encryption enabled
EO (0)	Encryption Options - Configure options for encryption. Unused bits should be set to 0. • 0x01=send the security key unsecured during join • 0x02=use trust center
KY	Encryption Key -Set the 128-bit AES encryption key. This is a write-only command.
Serial Interfacing	
BD (7)	Baud Rate - Set/read the serial interface data rate for communication between the DIGI XBEE USB device and the host running X-CTU. A drop-down menu lists valid values. The default value of 7 equals 115200 bps
NB (0)	Parity - Set/read the parity setting on the DIGI XBee USB device. A drop-down menu lists valid values.The default value of 0 specifies no parity.
D7 (1)	DIO7 Configuration - A drop-down menu lists valid values. The default value of 1 specifies CTS flow control.
D6 (1)	DIO6 Configuration - A drop-down menu lists valid values.
AP (1)	API Enable
AO (1)	API Output Mode
Sleep Modes	
SP (0xAF0)	Cyclic Sleep Period - This value determines how long the end device will sleep. On the parent, this value determines how long the parent will buffer a message for the sleeping end device. The valid range is 0x20-0xAF0 [* 10ms] (quarter second resolution).
SN (1)	Number of Cyclic Sleep Periods - Sets the number of sleep periods to not assert the On/Sleep pin on wakeup if no RF data is waiting for the end device. This command allows a host application to sleep for an extended time if no RF data is present. Range is 0x1 - 0xFFFF.
I/O Settings	
D0 (1)	AD0/DIO0 Configuration - A drop-down menu lists valid values for this pin. The default value of 1enables the commissioning button.
D1 (0)	AD1/DIO1 Configuration - A drop-down menu lists valid values for this pin. The default value of 0 disables it.
D2 (0)	AD2/DIO2 Configuration - A drop-down menu lists valid values for this pin. The default value of 0 disables it.

Table 5-3. AT Commands

D3 (0)	AD3/DIO3 Configuration - A drop-down menu lists valid values for this pin. The default value of 0 disables it.
D4 (0)	DIO4 Configuration - A drop-down menu lists valid values for this pin. The default value of 0 disables it.
D5 (1)	DIO5/Assoc Configuration - A drop-down menu lists valid values for this pin. The default value of 1 is what causes the LED labeled “Assoc” on the front of the DIGI XBee USB device to blink when the device has created a network.
P0 (1)	DIO10/PWM0 Configuration - A drop-down menu lists valid values for this pin. The default value of 1 indicates a received signal strength indicator (RSSI) value.
P1 (0)	DIO11 Configuration - A drop-down menu lists valid values for this pin. The default value of 0 disables it.
P2 (0)	DIO12 Configuration - A drop-down menu lists valid values for this pin. The default value of 0 disables it.
PR (0x1FFF)	Pull-up Resistor - Set/read the bit field that configures the internal pull-up resistor status for the I/O lines.
LT (0)	Associate LED Blink Time - Set/read the on/off blink times for the LED labeled “Assoc” on the front of the DIGI XBee USB device. The default value of 0 specifies the default blink rates: 500ms coordinator, 250ms router/end device. All other LT values are measured in 10 ms.
RP (0x28)	RSSI PWM Timer - Time RSSI signal will be output after last transmission. Valid range is 0-0xFF [* 100 ms]. A value of 0xFF means the output is always on.
Diagnostic Commands	
VR (factory set)	Firmware Version
HV (factory set)	Hardware Version

Table 5-3. AT Commands

AI (read only)	Association Indication - Read information regarding last node join request. This status monitors the progress of the association process. The following return values may be seen: • 0x00=successful completion: coordinator started a network • 0xAB=attempted to join a device that did not respond • 0xAC=secure join error, network security key received unsecured • 0xAD=secure join error, network security key not received • 0xAF=secure join error, joining device does not have the correct preconfigured link key • 0x21=scan found no PANs • 0x22=scan found no valid PANs based on current SC and ID settings. • 0x23=valid coordinator or router found but they are not allowing joining: node join time (NJ) expired • 0x27=node's attempt to join a network failed, typically due to incompatible security settings • 0x2A=coordinator start attempt failed • 0xFF=scanning for a parent • 0x2B=checking for an existing coordinator
DB	RSSI of Last Packet - reports the received signal strength of the last received RF data packet. Only the last hop is used, so this is not an accurate measurement for a multi-hop link.

5.4.2.3.3 Saving "Modem Parameters" to a File

The function of the button "Save" located under the label "Profile" is to save the parameter values in the display window to a file. The "Load" button lets you browse to a previously saved ".pro" file and open it to display its contents. To write the contents to the device attached to the selected COM port, you must click on "Write."

5.5 Summary

This is the ground floor of a very useful new standard. Dynamic C offers an easy-to-use implementation of ZigBee that works seamlessly with the Rabbit hardware as a solid foundation for a variety of embedded system projects that include wireless networking in their design.

This page intentionally left blank.

Appendix A. Glossary of Terms

This chapter defines a collection of terms that are commonly used when talking about networks in general or ZigBee in particular.

ad-hoc network

This term describes the mutable formation of small wireless networks. The peer-to-peer nature of mesh and cluster tree networks allows for this dynamic attribute by distributing the ability to join the network across the network.

application object

Code that implements the application. Each application object maps to one endpoint.

attribute

This term refers to a piece of data that can be passed between devices. A set of attributes is a cluster.

Bluetooth

Bluetooth is a set of standards that describes a short range (10 meter) frequency-hopping radio link between devices.

BPSK

This acronym stands for Binary Phase-Shift Keying. It is the keying of binary data by phase deviations of the carrier.

cluster

This is a ZigBee term that is defined as a container for attributes or as a command/response association. In the Dynamic C implementation of ZigBee, clusters are a collection of functions related to an endpoint.

cluster ID

This term refers to a unique 16-bit number that identifies a specific cluster within an application profile.

cluster tree

This term describes the physical topology of a network, its geometrical shape. For our purposes, a cluster tree network has as its root the coordinator for the WPAN. All routers that subsequently join the network form their own logical cluster.

coordinator

A ZigBee logical device type. There is one and only one coordinator per ZigBee network. This device has the unique responsibility of creating the WPAN.

CSMA-CA

This acronym stands for Carrier Sense Multiple Access/Collision Avoidance. It is a protocol used by a device that wants transmit on a network. The protocol seeks to avoid collisions by checking to see if the channel is clear before transmitting. If it is not clear, the device waits a radom amount of time and checks again.

device description

A device description is a document in a ZigBee profile. It describes the characteristics of a device that is required in the application area of the profile.

end device

This is a ZigBee term that indicates the device in question has no routing capability. It can only send and receive information for its own use. An end device functions as a leaf node in a cluster tree network. The nodes in a star network are all end devices except for the coordinator. A complete mesh network would not contain any end devices, but in practice a design may call for one or more of them.

endpoint

This is a ZigBee term that refers to an addressable unit on a device. For example, an LED or a digital input could be an endpoint on a Rabbit-based board.

FFD

This is an IEEE term that stands for full-function device. An FFD has routing capabilities, as opposed to an RFD (reduced-function device), which does not.

IEEE

Institute of Electrical and Electronics Engineers.

EUI-64

This acronym stands for Extended Unique Identifier 64 bits. It is an IEEE term used to describe the result of the concatenation of the 24-bit value assigned to an organization by the IEEE Registration Authority and a 40-bit extension assigned by that organization.

IrDA

This term stands for Infrared Data Association. It is a standard for transmitting data via infrared light waves. Look Ma! No cables!

LAN

This term stands for local area network. A LAN covers a relatively small area, though a larger area than a PAN. Corporations and academic institutions typically have their own LANs.

mesh

This term describes the physical topology of a network, its geometrical shape. A mesh network, with its dynamic arrangement of nodes, is ideally suited for the nimble world of wireless communication.

multi-hop

This term describes the ability of a message to be handled by intermediary nodes on its way to its destination node. Both mesh and cluster tree topologies are also known as multi-hop networks.

node

Generally, this term describes any device that is part of a network. For a ZigBee wireless network, the term applies to a device containing a single radio that has joined the network and therefore has a network ID.

O-QPSK

This acronym stands for Offset Quadrature Phase-Shift Keying. It is the keying of data by phase deviations of the carrier.

peer-to-peer

The term peer-to-peer refers to the relationship between two separate devices.

On a physical level it can mean the cables or the radio channel connecting the devices. In the physical sense of the term, peer-to-peer is the opposite of star where all devices in the network connect to one central device.

On a logical level, it means that the entities are equal in that they perform the same routing functions as their neighbor. In the logical sense, peer-to-peer is the opposite of the client/server model.

point-to-multipoint

This term refers to the communication path from a single location to multiple locations. Unlike a star topology which only has nodes one hop away from the coordinator node, in a point-to-multipoint ZigBee topology nodes can be several hops away from the coordinator node.

point-to-point

A circuit connecting two nodes only, creating a communication path from a single location to another single location.

profile

A profile (also known as an application profile) is a description of devices required in an application area and their interfaces.

router

A ZigBee logical device type that can route messages from one node to another.

RF

This term stand for radio frequency. The electromagnetic frequencies from 10 kHz to 300 GHz define the RF range. This is above audio range and below infrared light.

RFD

This is an IEEE term that stands for reduced-function device. An RFD does not have the routing capabilities of an FFD. A ZigBee end device and the IEEE reduced-function device both lack routing functions.

RSSI

Received Signal Strength Indicator.

self-healing network

This term describes the process of recovery in a mesh network. For example, if a node fails, the remaining nodes would find alternate routing paths to accomplish their tasks.

star

This term describes the physical topology of a network, its geometrical shape. For our purposes, a star network has as its root the coordinator for the WPAN. All devices that subsequently join the network can only communicate with the coordinator.

UWB

This terms stands for ultra-wideband. It refers to any radio technology that transmits information spread over a bandwidth larger than 500 MHz.

WPAN

This term stands for wireless personal area network. At bare minimum, it takes two devices operating a short distance from one another and communicating on the same physical channel to constitute a WPAN.

ZDO

This is a specialized application object called the ZigBee Device Object. It is addressed as endpoint 0.

Index

Z