PRODUCT MANUAL

Dynamic C®

An Introduction to ZigBee®

019-0162 • 080924-D

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

An Introduction to ZigBee®

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

Trademarks

Table of Contents

1. INTRODUCTION

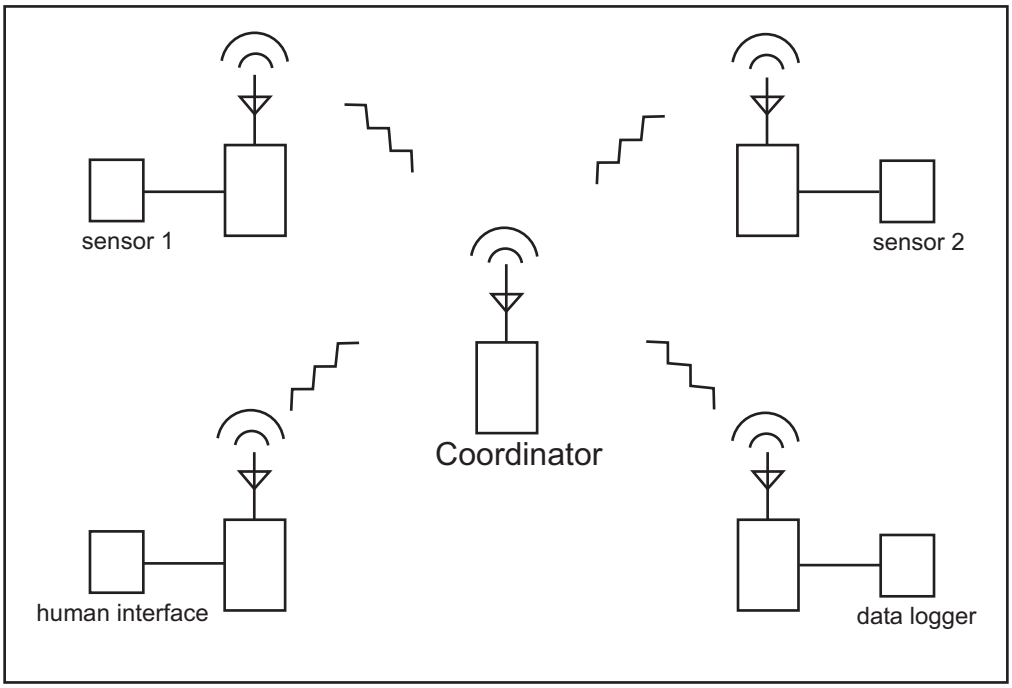
This manual provides an introduction to the various components of a ZigBee network. After a quick overview of ZigBee, we start with a description of high-level concepts used in wireless communication and move on to the specific protocols needed to implement the communication standards. This is followed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

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.

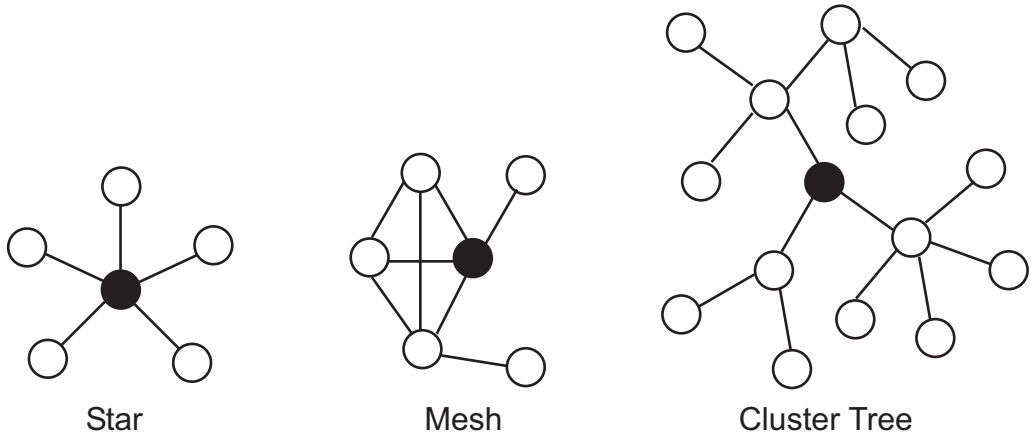
2.2.3 WWAN

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

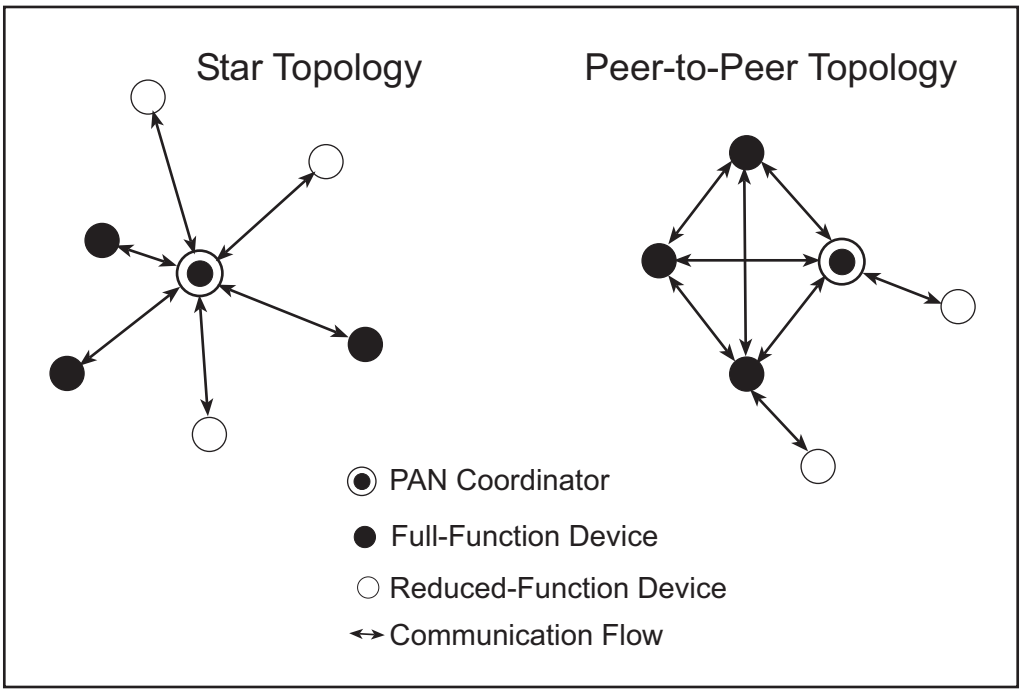
Of the three ISM frequency bands only the 2.4 GHz band operates world-wide. The 868 MHz band only operates in the EU and the 915 MHz band is only for North and South America. However, if global interoperability is not a requirement, the relative emptiness of the 915 MHz band in non-European countries 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 to join any network or it can limit itself to a network with a particular PAN ID.

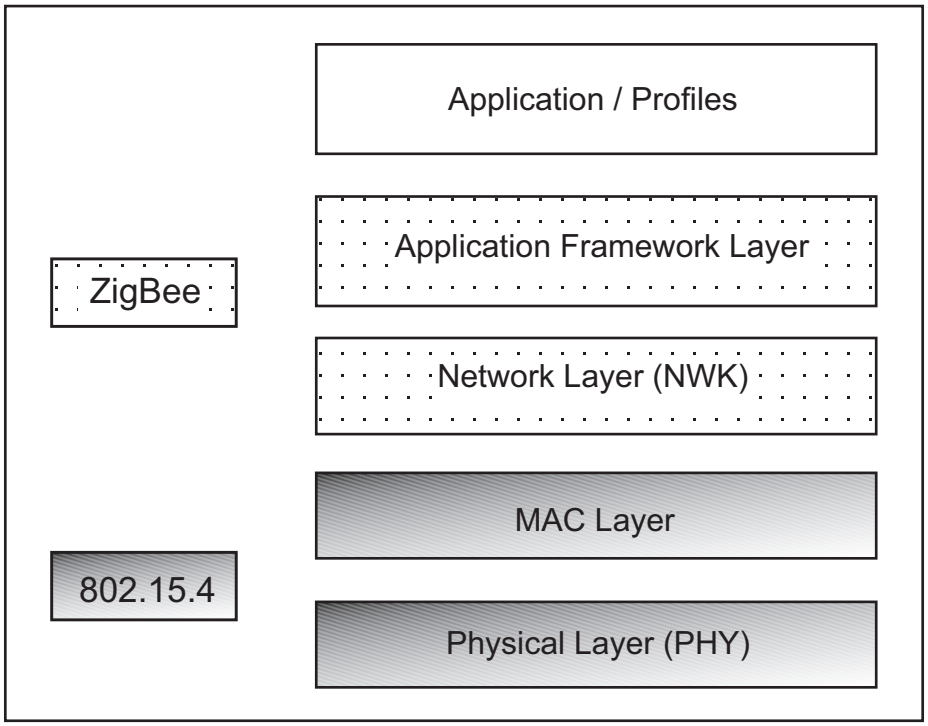
ZigBee PRO defines an extended PAN ID. It is a 64-bit number that is used as a network identifier in place of its 16-bit predecessor.

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.

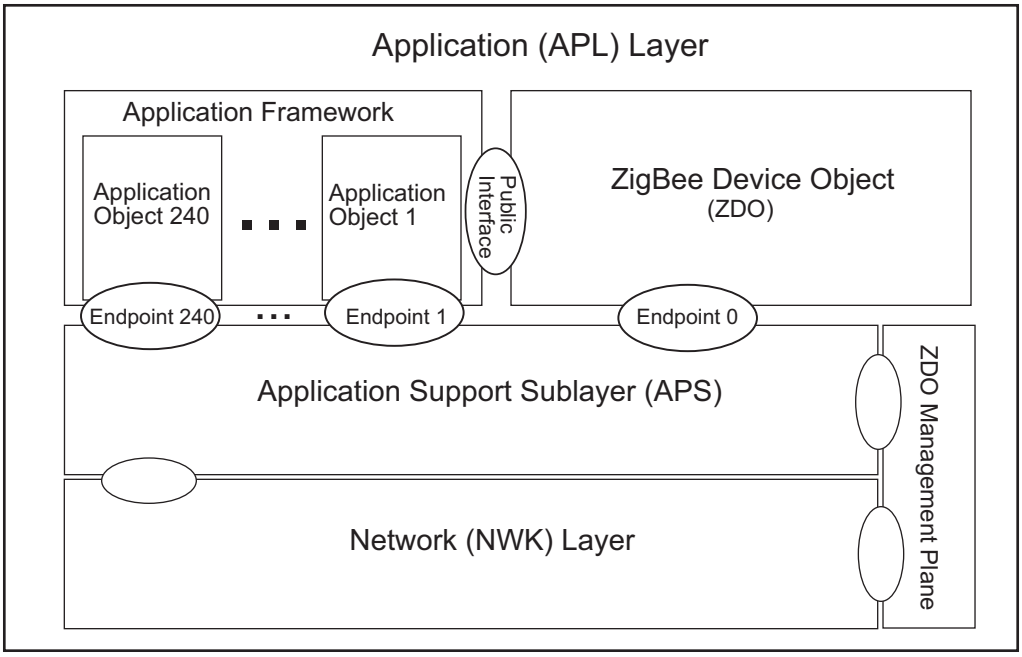
To join by association, a node sends out a beacon request on a channel, repeating the beacon request on other channels until it finds an acceptable network to join.

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 Profile (ZDP). 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 Profile (ZDP)

The ZDP 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.

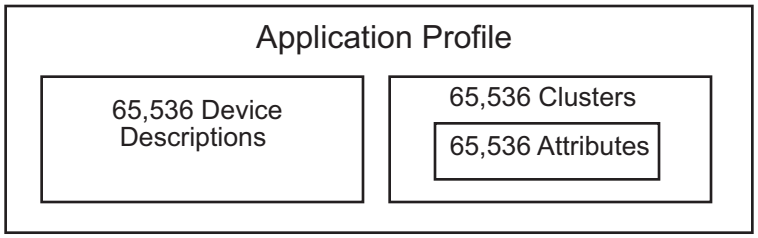
All profiles must have a unique profile identifier. You must contact the ZigBee Alliance if you have created a private profile in order to be allocated a unique profile identifier.

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 ZDP 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.

5. RABBIT AND ZIGBEE

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

RABBIT AND ZIGBEE

This chapter describes how to create a ZigBee application using Dynamic C and Rabbit-based ZigBee-capable boards. These libraries may also be used when adding a XBee module to any Rabbit-based design with some slight additional setup.

5.1 Implementation Overview

This group of libraries is not a full implementation of the ZigBee protocol, but rather an interface to the ZigBee API on the local XBee module. The XBee module has firmware that handles much of the real-time aspects of the ZigBee network. The module only delivers network traffic that is specific to the local node hosted by the XBee module. When the XBee module runs in API mode, it is dependent on the attached processor to handle the traffic it delivers and to initiate proper responses on to the network. Through the API, requests can also be made to other nodes on the network. So when using API mode, the tasks of the ZigBee node are split between the XBee module and the attached processor.

The XBee ZigBee driver is broken up into multiple layers, with well-defined interfaces between each layer. The layered interface is similar to an Ethernet NIC driver and a TCP/IP networking stack. The lowest layer is the Hardware Abstraction Layer (HAL) which allows a common interface for the upper layer regardless of the underlying hardware implementation. This layer provides a standard set of functions for working with the serial port used to communicate with the XBee module.

The next layer is the XBee Driver layer and it handles all serial communication with and configuration of the attached XBee device. This layer implements functions for working with the XBee module on a low level. It is comprised of several functional areas that all have their specific uses. These are the Device Interface, AT Commands, AT Mode, XBee ZCL Commissioning, Node Discovery, Firmware Updates, Digi Data Endpoint and the WPAN Layer Interface. The Device Interface works solely with the HAL and all other functions work through the Device Interface.

The third layer is the Wireless Personal Area Networking (WPAN) layer which provides generic 802.15.4 networking support. This layer introduces the concepts of endpoints and clusters. A single device can have multiple endpoints and each endpoint can have multiple clusters. A cluster in turn can have multiple attributes to describe or control specific functionality within the endpoint. Clusters are therefore the lowest level unit of addressability within the network. This layer also introduces the envelope structure which contains the network addresses of the sender and destination. Of course, on incoming envelopes the local XBee address is the destination and on outgoing envelopes the sender.

The top layer is the ZigBee Networking Stack layer which provides support for the ZigBee networking protocols. This layer is comprised of the ZigBee Device Object / Profile support and the ZigBee Cluster Library. Understand that the ZigBee Device Object (ZDO) and ZigBee Device Profile (ZDP) are in fact the same thing, the name was changed during the evolution of the ZigBee standard.

In addition to the ZigBee and driver stack layers there is the Simple XBee API that provides a simplified API for working with Digi XBee modules, either on their own or paired with a Rabbit-based controller. This particular API is XBee-specific both locally and on the network and is not compatible with ZigBee nodes from other manufacturers. This layer automates several aspects such as node discovery and simplifies access to remote I/O on other devices.

The diagram in Figure 5.1 shows the complete XBee/ZigBee API and illustrates the access between layers. The ZigBee Driver Layer and XBee Driver Layer show several of the specific components within the layer. The arrows illustrate that each upper layer can call all lower layers except the Hardware Abstraction Layer. This layer is typically accessed only through the XBee Driver Layer, with the exception of a few user support functions that are separate from actual XBee module communications and not involved in the direct operation of the ZigBee network.

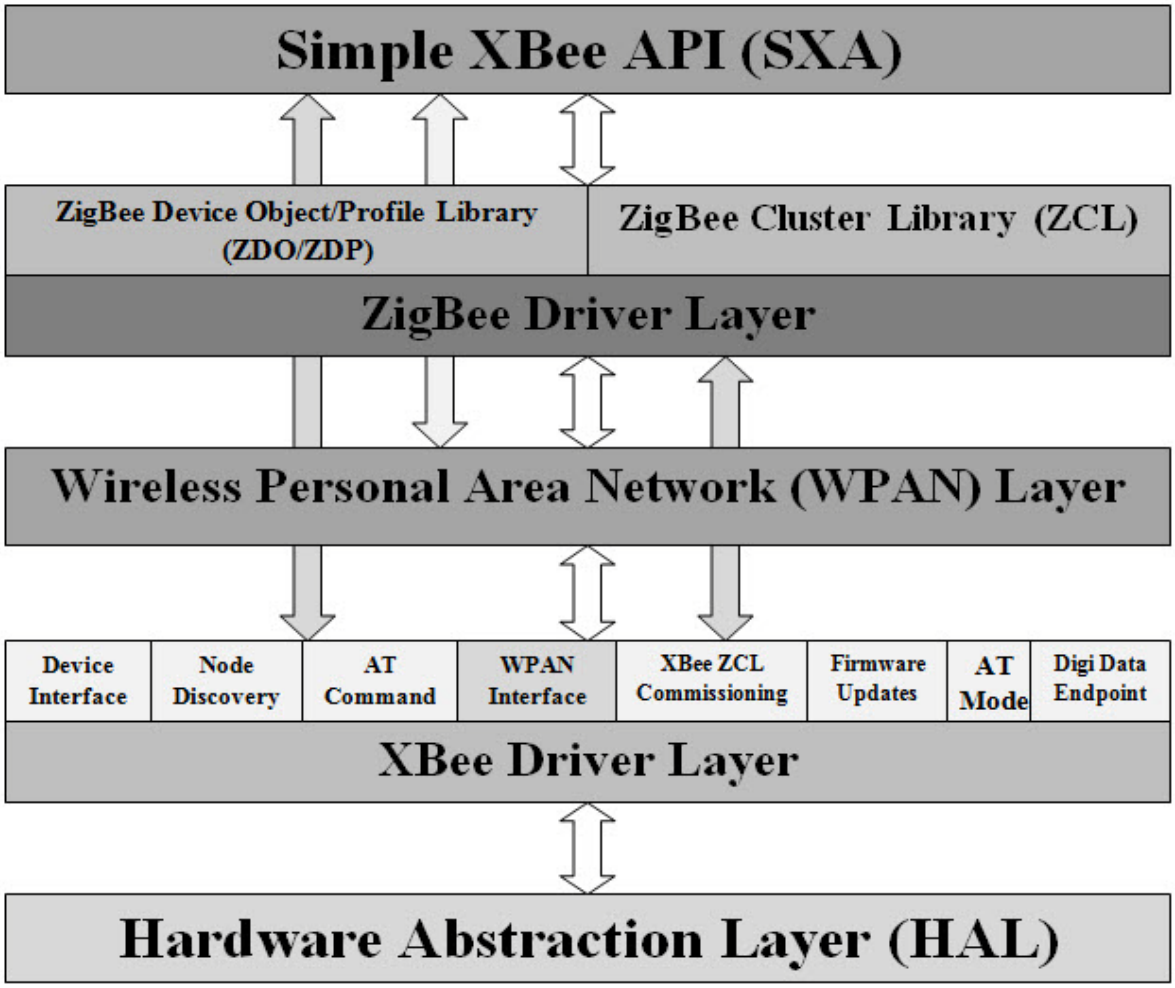


Figure 5.1 XBee/ZigBee Driver Layers

5.1.1 Configuration

The system is designed to simplify the interface to the XBee module. The sample file xbee_config.h in the `Samples/XBee` directory creates automatic configuration definitions for a variety of Rabbit-based core modules and single board computers. This allows a standardized call to the xbee_dev_init function to initialize the XBee device. All of the supplied samples make use of this header file to automatically configure the development environment.

When creating a customized core module to XBee interface, there is XBEE_STD_CONFIG which can be set to create a couple of standard port settings. The table below shows the possible settings available by setting XBEE_STD_CONFIG. If other port settings are desired, then the xbee_config.h file can be used as a guide to setup the parameters needed by the xbee_dev_init function.

XBEE_STD_CONFIG Value	Serial Port	TX Port Pin	RX Port Pin	RTS Port Pin	CTS Port Pin
1	- - - - - - - - - - Settings Based on Core / SBC Board In Use - - - - - - - - - -				
2	D	PC0	PC1	PC2	PC3
3	F	PE2	PE3	PA0	PD0

5.1.2 Simple XBee API Initialization

The Simple XBee API (herein SXA) makes initialization of a XBee-only network very easy. The sxa_init_or_exit function is an example of an all-in-one initialization sequence for the XBee device This function will initialize not only the device, but the network and AT command layers. The SXA layer will automatically explore the device and scan the network to discover each XBee module's control and I/O settings and save them to an internal node list used by the SXA layer. The SXA layer has its own dedicated tick function named `sxa_tick` which must be regularly called to allow processing of incoming responses or requests. The sample programs that use the SXA layer all have the sxa prefix at the start of their file name.

5.1.3 ZigBee Initialization

Initializing the system when not using the SXA layer is a bit more involved, but allows interaction with most ZigBee-compatible devices on the network. After device initialization, the next step is to initialize the WPAN layer by calling the xbee_wpan_init function. The next step is to initialize the AT command layer by calling the xbee_cmd_init_device function. This must be followed by repeated calls to the wpan_tick function and status checks to the xbee_cmd_query_status function.

The wpan_tick function drives both the WPAN layer and XBee Driver layer. Most functions in the system are non-blocking, which means they return before the XBee device or network devices have actually completed the requested operations. There are several 'tick' functions at different layers of the overall system which drive processing, as well as associated status functions to see when operations complete. This allows your embedded system to work on other time-sensitive operations while the XBee and network are processing your requests.

5.1.4 Network Overhead Requirements

As mentioned in the prior sections, the calling of 'tick' functions and appropriate status functions are fundamental to the overall ZigBee libraries and driver operation. These functions must be called within certain times to prevent internal buffers from overflowing. There are several tick functions that invoke processing at different layers and with different call timing. The lowest level tick is xbee_dev_tick and this drives the XBee Driver Layer. The primary higher-level tick functions are wpan_tick and sxa_tick. The wpan_tick function drives all ZigBee layers and should be used unless the SXA layer is desired. The sxa_tick function drives SXA and lower layers.

All of the tick functions (xbee_dev_tick, wpan_tick and sxa_tick) have the same timing requirements. These are based on the speed and packet/buffer size that is setup for the serial port connected to the XBee device. By default the baudrate is 115.2kbps and the receive buffer size is 255 bytes. Since this buffer size is larger than the typical largest packet size of 128 bytes, the timing should be based on the packet size. If the buffering of the port is reduced then this calculation would need to based on the buffer size.

The formula for minimum calling rate time is **time = size / (baud / 10)** where size is the lower of either packet or buffer size. For the default settings this comes out to 11ms. Therefore the tick function should be called at least every 11 ms to ensure no data loss. This timing requirement holds true for a simple tick function that handles the return code from a single buffered packet.

If longer times are needed between calls to the tick function, then this requires either slowing the baud rate or increasing the buffer size and creating a more complex tick function that processes all complete received packets from the buffer in a single tick call. This would guarantee that after the tick call the buffer would have less than one packet worth of data. So then the formula for minimum calling rate would change to **time = (buffer size - packet size) / (baud / 10)**. Therefore if the buffer size was increased to 511 bytes and maximum packet size was 128 bytes with the default baud rate the minimum call time would be 33ms. Larger serial buffer sizes can be used, but note that serial buffer sizes must be of a value equal to 2 to the power of N minus one for any positive value of N. The callback time should always be kept to less than 200ms to avoid network time out issues.

The final tick function is xbee_cmd_tick which is used to age and expire old AT command requests. This function should be called at least once every 2 or 3 seconds.

5.1.5 End Device Sleep Mode

This system has functions to check when the local XBee is asleep and will not allow transmissions to be sent to the device if there are hardware handshake ports included in the interface. If the hardware implementation does not include hardware handshaking, then it is the program's responsibility to check if the device is awake before sending a command to the device. The XBee Driver does not automatically check the awake status, only Clear To Send (CTS) signals when available.

5.2 Sample Programs

This section describes the Dynamic C sample programs that exercise the XBee and ZigBee network functionality.

Dynamic C sample programs for the XBee are in the folder `Samples/XBee` relative to the Dynamic C installation folder. Some of the sample programs can be run with one Rabbit-based board by itself, but can also be run as part of a network. Others require either a secondary Rabbit-based board, XBee interface board (XBIB) or a Digi XBee USB device to form an actual network. The Digi XBee USB device is a simple USB dongle. The XBee interface board includes a USB interface to an XBee module, as well as some switches, LEDs and connectors to allow further exploration or prototyping of a standalone XBee module. Either XBIB or dongle hardware can 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.

5.2.1 Sample Program Usage Requirements

All of the samples make use of the "xbee_config.h" header file. This file provides automatic configuration of the hardware interface between the Rabbit processor and the XBee module on standard XBee enabled Rabbit devices. This includes both core modules and single board computers.

There are samples that use the Simple XBee API as well as samples that use the ZigBee library system directly. All samples that use the Simple XBee API start with SXA.

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 Alone (No Networking)

The sample programs listed here require just one Rabbit-based board to run the sample. The network is not initialized in these samples, they only demonstrate direct communications to the XBee module.

- Basic_XBee_Init.c - This sample program illustrates the basic initialization of an XBee device. It does not go any further into configuration or network establishment. Running this successfully verifies that the communication link between the RF module and its Rabbit-based board is working properly.

- Basic_XBee_Query.c - This sample program shows how to set up and send a list of AT commands. It sends local AT commands to the XBee module one at a time. It reads the parameters that are returned and stores them in a local structure. If the command list processes to the end, it then displays them in the Stdio window. Otherwise it reports the error. It does not create a network, it is local only.
- serial_bypass.c - This sample program bridges the programming port of the Rabbit (Serial A) with the serial port connected to the XBee module. This allows the Rabbit programming port to talk directly with the XBee module. By running this sample code, the PC-based X-CTU program connects to an XBee module through the Rabbit programming port, allowing the X-CTU program to configure and update the firmware on the XBee module. To get the latest firmware files, perform a web update in the X-CTU program. See section 5.8.1 for more information on the X-CTU program.
- xbee_update_ebl.c - This sample program reads XBee firmware from an ximported '.ebl' file and updates the firmware on the attached XBee module. This demonstrates a method of remote update and could be expanded to allow the firmware file to be downloaded from a network or retrieved from a FAT file system file. X-CTU stores '.ebl' files in `Program Files\Digi\XCTU\update\ebl_files`. See section 5.8.1 for more information on X-CTU and a table of firmware files to use.

5.2.2.2 Sample Programs for One Rabbit-Based Board and XBIB or USB Dongle

The sample programs listed here require a minimum of one Rabbit-based board to run the sample and at least one additional XBee node which could be Rabbit-based, on an XBIB or inside a USB Dongle.

- 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 the user to enter one. Running this program successfully verifies that the communication link between the XBee module and its Rabbit-based board is working properly. This sample can communicate with remote nodes if the MAC address is known and the node has joined the network.
- AT_remote.c - Similar to the AT_interactive sample but with node discovery which will then display a list of remote nodes to communicate with. This sample makes it much easier to find and work with remote nodes through interactive AT commands.
- SXA-command.c - This sample program shows the use of the Simple XBee API to discover remote nodes and issue AT commands to them, both interactively and programmatically.
- SXA-stream.c - This demonstrates using the Simple XBee API library to discover remote nodes and allows sending simple streams of data between XBee nodes on a network. It uses the Digi Transparent Serial cluster ; the same cluster used by XBee modules running "AT firmware" instead of "API firmware". This sample will work with stand-alone XBee modules tied to a serial device.
- transparent_client.c - This sample demonstrates the manual setup required for sending simple streams of data between XBee nodes on a network when not using the Simple XBee API library. It uses the Digi Transparent Serial cluster; the same cluster used by XBee modules running "AT firmware" instead of "API firmware". This sample will work with stand-alone XBee modules tied to a serial device.

5.2.2.3 Sample Programs for Two Rabbit-Based Boards

The sample programs listed here require a minimum of two Rabbit-based boards to run the sample. Both boards must be programmed with sample code to perform the sample functionality.

- SXAsocket.c - This sample program shows the use of the Simple XBee API library to discover remote XBee noeds and send data streams back and forth, using a reliable TCP socket-like protocol. This sam-

ple uses a custom ZigBee endpoint to implement a protocol similar to TCP/IP. This requires the sample program to be run on both ends to process the custom protocol.

5.3 Hardware Abstraction Layer (HAL)

This section contains information about the Hardware Abstraction Layer for XBee devices.

5.3.1 General Overview

Much of the Hardware Abstraction Layer is actually designed to be accessed through the serial functions wrappers in the XBee driver layer. This section lists the helper functions that can be directly called.

5.3.2 API Functions and Macros

This section contains some useful helper function descriptions for the Hardware Abstraction Layer. These functions are not network related, but rather assist with stdio, data display, data conversion and timing. To use any of these functions you must #include either "xbee/serial.h" or the xbee_config.h header file from the `Samples/XBee` directory.

---

xbee_readline

```
int xbee_readline ( char * s )
```

DESCRIPTION

This function is a non-blocking version of gets().

It waits for a string from stdin terminated by a return. It should be called repeatedly, until it returns 1. The input string, stored at `s` is null-terminated without the return.

The caller is responsible to make sure the location pointed to by `s` is big enough for the string.

The user should make sure only one process calls this function at a time.

PARAMETERS

[in,out]s Buffer to store string from user.

RETURN VALUE

1 User ended the input with a newline.

0 User has not completed a line.

---

hex_dump

```
void hex_dump (const void FAR * address, uint16_t length, uint16_t
```

flags)

DESCRIPTION

Helper function for printing a hex dump of memory to stdout. A reference implementation is provided in as util/hexdump.c. Dumps data in hex/printable format, 16 bytes to a line, to stdout.

PARAMETERS

[in] address	Address of data to dump.
[in] length	Number of bytes to dump.
[in] flags	One of: • HEX_DUMP_FLAG_NONE • HEX_DUMP_FLAG_OFFSET • HEX_DUMP_FLAG_ADDRESS • HEX_DUMP_FLAG_TAB

hexstrtobyte

```
int hexstrtobyte (const char FAR * p)
```

DESCRIPTION

Converts two hex characters (0-9A-Fa-f) to an equivalent byte value.

PARAMETERS

[in] p	String of hex characters to convert.

RETURN VALUE

-1	Error (invalid character or string less than 2 bytes).
0-255	The byte represented by the first two characters of p.

Examples

hexstrtobyte("FF") returns 255

hexstrtobyte("0") returns -1 (error because < 2 characters)

hexstrtobyte("ABCDEF") returns 0xAB (ignores additional chars)

xbee_millisecond_timer

```
uint32_t xbee_millisecond_timer ( void )
```

DESCRIPTION

Macro which returns the number of elapsed milliseconds.

This counter does rollover and has 1ms resolution.

RETURN VALUE

0 Number of elapsed milliseconds.

xbee_seconds_timer

```
uint32_t xbee_seconds_timer ( void )
```

DESCRIPTION

Macro which returns the number of elapsed seconds. This counter does rollover and has 1 second resolution. In addition to determining timeouts, the ZCL Time Cluster makes use of it to report current time.

RETURN VALUE

Number of elapsed seconds.

5.4.2 XBee Driver Layer API Functions and Macros

This section contains descriptions for the XBee Driver Layer functions and macros separated into layer components.

5.4.2.1 XBee Driver Layer General Usage

Most of the functions in this layer are not used directly in applications.

5.4.2.2 Serial API Functions and Macros

This section contains API descriptions for the serial component of the XBee Driver Layer. To use any of these functions the program must **#include** either **"XBee/serial.h"** or the **xbee_config.h** header file from the DC_Root/Samples/XBee directory.

xbee_ser_baudrate

```
int xbee_ser_baudrate ( xbee_serial_t *  serial, uint32_t  baudrate )
```

DESCRIPTION

Change the baud rate of XBee serial port `serial` to `baudrate` bits/second.

PARAMETERS

[in] serial	XBee serial port
[in] baudrate	Bits per second of serial data transfer speed.

RETURN VALUE

0	Opened serial port within 5% of requested baudrate.
-EINVAL	Parameter `serial` is not a valid XBee serial port.
-EIO	Can't open serial port within 5% of requested baudrate.

SEE ALSO

xbee_ser_open(), xbee_ser_close(), xbee_ser_break()

xbee_ser_break

```
int xbee_ser_break ( xbee_serial_t *  serial, bool_t  enabled )
```

DESCRIPTION

Disable the serial transmit pin and pull it low to send a break to the XBee serial port.

PARAMETERS

[in] serial	XBee serial port
[in] enabled	Set to 1 to start the break or 0 to end the break (and resume transmitting).

RETURN VALUE

0	Success
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_open(), xbee_ser_close()

Referenced by xbee_fw_install_ebl_tick(), and xbee_fw_install_oem_tick().

xbee_ser_close

```
int xbee_ser_close ( xbee_serial_t *  serial )
```

DESCRIPTION

Close the serial port attached to XBee serial port `serial`.

PARAMETERS

[in] serial	XBee serial port

RETURN VALUE

0	Closed serial port
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_open(), xbee_ser_baudrate(), xbee_ser_break()

xbee_ser_flowcontrol

```
int xbee_ser_flowcontrol ( xbee_serial_t *  serial, bool_t  enabled )
```

DESCRIPTION

Enable or disable hardware flow control (CTS/RTS) on the serial port for XBee serial port `serial`.

PARAMETERS

[in] serial	XBee serial port
[in] enabled	Set to 0 to disable flow control or non-zero to enable flow control.

RETURN VALUE

0	Success.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_set_rts(), xbee_ser_get_cts()

xbee_ser_get_cts

```
int xbee_ser_get_cts ( xbee_serial_t *  serial)
```

DESCRIPTION

Read the status of the /CTS (clear to send) pin on the serial port connected to XBee serial port `serial`.

Note: This function doesn't return the value of the pin -- it returns whether it's asserted (i.e., clear to send to the XBee serial port) or not.

PARAMETER

[in] serial	XBee serial port

RETURN VALUE

1	It's clear to send.
0	It's not clear to send.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_flowcontrol(), xbee_ser_set_rts()

xbee_ser_getchar

```
int xbee_ser_getchar ( xbee_serial_t *  serial)
```

DESCRIPTION

Reads a single character from the XBee serial port `serial`.

PARAMETER

[in] serial	XBee serial port

RETURN VALUE

0-255	Character read from XBee serial port.
-ENODATA	There aren't any characters in the read buffer.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_read(), xbee_ser_write(), xbee_ser_getchar()

xbee_ser_invalid

```
bool_t xbee_ser_invalid ( xbee_serial_t *  serial)
```

DESCRIPTION

Helper function used by other xbee_serial functions to validate the serial parameter.

Confirms that it is non-NULL and is set to a valid port.

PARAMETER

[in] serial	XBee serial port

RETURN VALUE

1	`serial` is not a valid XBee serial port.
0	`serial` is a valid XBee serial port.

xbee_ser_open

```
int xbee_ser_open ( xbee_serial_t *  serial, uint32_t baudrate )
```

DESCRIPTION

Opens the serial port connected to XBee serial port `serial` at `baudrate` bits/second.

PARAMETERS

[in] serial	XBee serial port
[in] baudrate	Bits per second of serial data transfer speed.

RETURN VALUE

0	Opened serial port within 5% of requested baudrate.
-EINVAL	`serial` is not a valid XBee serial port.
-EIO	Can't open serial port within 5% of requested baudrate.

SEE ALSO

xbee_ser_baudrate(), xbee_ser_close(), xbee_ser_break()

xbee_ser_portname

```
const char* xbee_ser_portname ( xbee_serial_t *  serial)
```

DESCRIPTION

Returns a human-readable string describing the serial port attached (‘A’ thru ‘F’). If `serial` parameter is invalid or the port is not initialized, it will return the string “(invalid)”.

PARAMETERS

[in] serial	XBee serial port

RETURN VALUE

0	Null-terminated string describing the serial port.
“(invalid)”	If `serial` is invalid or not configured.

xbee_ser_putchar

```
int xbee_ser_putchar ( xbee_serial_t *  serial, uint8_t ch )
```

DESCRIPTION

Transmits a single character, `ch`, to the XBee serial port `serial`.

PARAMETERS

[in] serial	XBee serial port
[in] ch	Character to send

RETURN VALUE

0	Successfully sent (queued) character.
-ENOSPC	The write buffer is full and the character wasn't sent.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_read(), xbee_ser_write(), xbee_ser_getchar()

xbee_ser_read

```
int xbee_ser_read ( xbee_serial_t *  serial,void FAR * buffer, int
   bufsize )
```

DESCRIPTION

Reads up to `bufsize` bytes from XBee serial port `serial` and into `buffer`.

If there is no data available when the function is called, it will return immediately.

PARAMETERS

[in] serial	XBee serial port
[out] buffer	Buffer to hold bytes read from XBee serial port
[in] bufsize	Maximum number of bytes to read

RETURN VALUE

>=0	The number of bytes read from XBee serial port.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

`xbee_ser_write()`, `xbee_ser_putchar()`, `xbee_ser_getchar()`

xbee_ser_rx_flush

```
int xbee_ser_rx_flush ( xbee_serial_t *  serial)
```

DESCRIPTION

Deletes all characters in the serial receive buffer for XBee serial port `serial`.

PARAMETERS

[in] serial	XBee serial port

RETURN VALUE

0	Success.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

`xbee_ser_tx_free()`, `xbee_ser_tx_used()`, `xbee_ser_tx_flush()`, `xbee_ser_rx_free()`, `xbee_ser_rx_used()`

xbee_ser_rx_free

`int xbee_ser_rx_free ( xbee_serial_t * serial)`

DESCRIPTION

Returns the number of unused bytes in the serial receive buffer for XBee serial port `serial`.

PARAMETERS

`[in] serial` XBee serial port

RETURN VALUE

INT_MAX	The buffer size is unlimited (or unknown).
>=0	The number of bytes it would take to fill the XBee serial port's serial receive buffer.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

`xbee_ser_tx_free()`, `xbee_ser_tx_used()`, `xbee_ser_tx_flush()`, `xbee_ser_rx_used()`, `xbee_ser_rx_flush()`

xbee_ser_rx_used

`int xbee_ser_rx_used ( xbee_serial_t * serial)`

DESCRIPTION

Returns the number of queued bytes in the serial receive buffer for XBee serial port `serial`.

PARAMETERS

`[in] serial` XBee serial port

RETURN VALUE

>=0	The number of bytes queued in the XBee serial port's serial transmit buffer.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_tx_free(), xbee_ser_tx_used(), xbee_ser_tx_flush(), xbee_ser_rx_free(), xbee_ser_rx_flush()

NOTE: Unlike xbee_ser_tx_used(), this function MUST return the number of bytes available. Some layers of the library wait until enough bytes are ready before continuing.

xbee_ser_set_rts

```
int xbee_ser_set_rts ( xbee_serial_t *  serial, bool_t asserted )
```

DESCRIPTION

Disable hardware flow control and manually set the RTS (ready to send) pin on the XBee device's serial port.

Typically used to enter the XBee device's boot loader and initiate a firmware update.

PARAMETERS

[in] serial	XBee serial port
[in] asserted	Set to 1 to assert RTS (ok for XBee to send to us) or 0 to deassert RTS (tell XBee not to send to us).

RETURN VALUE

0	Success.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_flowcontrol(), xbee_ser_get_cts()

xbee_ser_tx_flush

```
int xbee_ser_tx_flush ( xbee_serial_t *  serial)
```

DESCRIPTION

Flushes (i.e., deletes and does not transmit) characters in the serial transmit buffer for XBee serial port `serial`.

PARAMETERS

[in] serial	XBee serial port

RETURN VALUE

0	Success
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_rx_free(), xbee_ser_rx_used(), xbee_ser_rx_flush(), xbee_ser_tx_free(), xbee_ser_tx_used()

xbee_ser_tx_free

```
int xbee_ser_tx_free ( xbee_serial_t *  serial)
```

DESCRIPTION

Returns the number of bytes of unused space in the serial transmit buffer for XBee serial port `serial`.

PARAMETERS

[in] serial	XBee serial port

RETURN VALUE

INT_MAX	The buffer size is unlimited (or unknown).
>=0	The number of bytes it would take to fill the XBee serial port's serial transmit buffer.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_rx_free(), xbee_ser_rx_used(), xbee_ser_rx_flush(), xbee_ser_tx_used(), xbee_ser_tx_flush()

xbee_ser_tx_used

```
int xbee_ser_tx_used ( xbee_serial_t *  serial)
```

DESCRIPTION

Returns the number of queued bytes in the serial transmit buffer for XBee serial port `serial`.

PARAMETERS

[in] serial	XBee serial port

RETURN VALUE

0	The buffer size is unlimited (or space used is unknown).
>0	The number of bytes queued in the XBee serial port's serial transmit buffer.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_rx_free(), xbee_ser_rx_used(), xbee_ser_rx_flush(), xbee_ser_tx_free(), xbee_ser_tx_flush()

xbee_ser_write

```
int xbee_ser_write ( xbee_serial_t *  serial, const void FAR * buffer,
   int length)
```

DESCRIPTION

Transmits `length` bytes from `buffer` to the XBee serial port `serial`.

PARAMETERS

[in] serial	XBee serial port
[in] buffer	Source of bytes to send
[in] length	Number of bytes to write

RETURN VALUE

>=0	The number of bytes successfully written to XBee serial port.
-EINVAL	`serial` is not a valid XBee serial port.

SEE ALSO

xbee_ser_read(), xbee_ser_putchar(), xbee_ser_getchar()

5.4.2.3 Device Interface API Functions and Macros

This section contains function descriptions for the device interface component of the XBee Driver Layer. To use these functions the program must have **#include "XBee/device.h"** within the code.

xbee_dev_dump

```
void xbee_dev_dump ( xbee_dev_t * xbee, uint16_t flags )
```

DESCRIPTION

Print information to stdout about the XBee device.

Default behavior is to print the name of the serial port, the XBee module's hardware version (ATHV), firmware version (ATVR), IEEE address (ATSH/ATSL) and network address (ATMY).

Assumes the user has already called xbee_cmd_init_device() and waited for xbee_cmd_query_status() to finish.

PARAMETERS

[in] xbee	Device to get information about
[in] flags	XBEE_DEV_DUMP_FLAG_NONE: default settings

SEE ALSO

xbee_cmd_init_device(), xbee_cmd_query_status(), xbee_ser_portname()

xbee_dev_init

```
int xbee_dev_init ( xbee_dev_t * xbee, const xbee_serial_t * serport,
   xbee_is_awake_fn is_awake, xbee_reset_fn reset )
```

DESCRIPTION

Initialize the XBee device structure and open a serial connection to a local, serially-attached XBee module.

This function does not actually initiate communications with the XBee module. See xbee_cmd_init_device() for information on initializing the "AT Command" layer of the driver, which will read basic information from the XBee module.

PARAMETERS

[in] xbee	XBee device to initialize.
[in] serport	Pointer to an xbee_serial_t structure used to initialize xbee->serport.

[in] is_awake Pointer to function that reads the XBee module's "ON" pin. The function should return 1 if XBee is on and 0 if it is off.

ie:int xbee_is_awake (xbee_dev_t * xbee)

[in] reset Pointer to function that asserts the XBee module's "/RESET" pin. If `asserted` is TRUE, puts the XBee into reset. If `asserted` is FALSE, takes it out of reset. No return value.

ie:void xbee_reset (xbee_dev_t * xbee, bool_t asserted)

RETURN VALUE

0	Success
-EINVAL	Invalid parameter (`xbee` is NULL, `serport` is not valid, etc.)
-EIO	Couldn't set serial port baudrate within 5% of `serport->baudrate`

xbee_dev_reset

```
int xbee_dev_reset ( xbee_dev_t * xbee, uint16_t flags )
```

DESCRIPTION

Toggles the reset line of the XBee device.

PARAMETERS

[in] xbee XBee device to reset.

RETURN VALUE

0	Successfully toggled reset.
-EINVAL	Invalid xbee_dev_t passed to function.
-EIO	This XBee device doesn't have an interface to the module's reset pin.

xbee_dev_tick

```
int xbee_dev_tick ( xbee_dev_t * xbee )
```

DESCRIPTION

Check for newly received frames on an XBee device and dispatch them to registered frame handlers. See section 5.1.4 for a discussion of how frequently this function needs to be called for a

given configuration.

Execution time depends greatly on how long each frame handler takes to process its frame.

Warning!

This function is NOT re-entrant and will return -EBUSY if it is called when already running.

PARAMETERS

[in] xbee	XBee device to check for, and then dispatch, new frames.

RETURN VALUE

>=0	Number of frames received and dispatched.
-EINVAL	If `xbee` isn't a valid device structure.
-EBUSY	If xbee_dev_tick() was called when it's already running for this device.

SEE ALSO

xbee_frame_load()

xbee_frame_dump_modem_status

```
int xbee_frame_dump_modem_status ( xbee_dev_t * xbee, const void FAR
   * frame, uint16_t length, void FAR * context )
```

DESCRIPTION

Frame handler for 0x8A (XBEE_FRAME_MODEM_STATUS) frames -- dumps modem status to STDOUT for debugging purposes.

PARAMETERS

[in] xbee	Dummy parameter, here to match standard frame handler callback API
[in] frame	Pointer to the frame to get the modem status from.
[in] length	Dummy parameter, here to match standard frame handler callback API
[in] context	Dummy parameter, here to match standard frame handler callback API

SEE ALSO

xbee_frame_handler_fn()

xbee_frame_write

```
int xbee_frame_write ( xbee_dev_t * xbee, const void FAR * header,
   uint16_t headerlen, const void FAR * data, uint16_t datalen,
   uint16_t flags )
```

DESCRIPTION

Copies a frame into the transmit serial buffer to send to an XBee module.

Header should include bytes as they will be sent to the XBee. Function accepts separate header and data to limit the amount of copying necessary to send requests.

This function should only be called after `xbee` has been initialized by calling xbee_serial_init().

PARAMETERS

[in] xbee	XBee device to send to.
[in] header	Pointer to the header to send. Header starts with the frame type (this function will pre-pend the 0x7E start-of-frame and 16-bit length). Pass NULL if there isn't a header and the entire frame is in the payload (data and).
[in] headerlen	Number of header bytes to send (starting with address passed in header). Ignored if header is NULL.
[in] data	Address of frame payload or NULL if the entire frame content is stored in the header bytes (header and header-len).
[in] datalen	Number of payload bytes to send (starting with address passed in data). Ignored if data is NULL.
[in] flags	Optional flags XBEE_WRITE_FLAG_NONE

RETURN VALUE

0	Successfully queued frame in transmit serial buffer.
-EINVAL	`xbee` is NULL or invalid flags passed
-ENODATA	No data to send (`headerlen` + `datalen` == 0).
-EBUSY	Transmit serial buffer is full, or XBee is not accepting serial data (deasserting /CTS signal).

SEE ALSO

xbee_serial_init(), xbee_serial_write(), xbee_frame_load()

5.4.2.4 AT Commands API Functions and Macros

This section contains API descriptions for the AT commands component of the XBee Driver Layer. To use these functions, the program must have **#include "xbee/atcmd.h"** within the code.

xbee_cmd_clear_flags

```
int xbee_cmd_clear_flags ( int16_t handle, uint16_t flags )
```

DESCRIPTION

Clear the flags for a given AT Command request.

PARAMETERS

[in] handle	Handle to the request, as returned by xbee_cmd_create()
[in] flags	Optionally one or more of the following: •XBEE_CMD_FLAG_QUEUE_CHANGE don't apply changes until ATAC or another command without this flag is sent.

RETURN VALUE

0	Flags cleared.
-EINVAL	Handle is not valid.

SEE ALSO

xbee_cmd_set_flags()

xbee_cmd_create

```
int xbee_cmd_create ( xbee_dev_t * xbee, const char FAR command[3] )
```

DESCRIPTION

Allocate an AT Command request.

PARAMETERS

[in] xbee	XBee device to use as the target (local) or to send through (remote). This function will automatically call xbee_cmd_init() if it hasn't already been called for this device.
[in] command	Two-letter AT Command to send (e.g., "VR", "NI", etc.).

RETURN VALUE

>0	A "handle" to a request that can be built up and sent to a local (serially-attached) or remote XBee module.
-ENOSPC	The AT command table is full (increase the compile-time macro XBEE_CMD_REQUEST_TABLESIZE).
-EINVAL	An invalid parameter was passed to the function.

SEE ALSO

xbee_cmd_init_device(), xbee_cmd_query_device(), xbee_cmd_set_command(), xbee_cmd_set_callback(), xbee_cmd_set_target(), xbee_cmd_set_param(), xbee_cmd_set_param_bytes(), xbee_cmd_set_param_str(), xbee_cmd_send()

xbee_cmd_execute

```
int xbee_cmd_execute ( xbee_dev_t * xbee, const char FAR command[3],
   const void FAR * data, uint8_t length )
```

DESCRIPTION

Simple interface for sending a command with an optional parameter to the local XBee without checking for a response.

For an asynchronous method of sending AT commands and getting the response, see xbee_cmd_create, the xbee_cmd_set_* functions and xbee_cmd_send.

PARAMETERS

[in] xbee	XBee device to use as the target.
[in] command	Two-letter AT Command to send (e.g., "ID", "CH", etc.).
[in] data	Optional big-endian (MSB-first) value to assign to command. Use NULL if command doesn't take a parameter.
[in] length	Number of bytes in `data`; ignored if data is NULL.

RETURN VALUE

0	Command sent
-EINVAL	An invalid parameter was passed to the function
-EBUSY	Transmit serial buffer is full, or XBee is not accepting serial data (deasserting /CTS signal).

SEE ALSO

xbee_cmd_create(), xbee_cmd_set_command(), xbee_cmd_set_callback(), xbee_cmd_set_target(), xbee_cmd_set_param(), xbee_cmd_set_param_bytes(), xbee_cmd_set_param_str(), xbee_cmd_simple()

xbee_cmd_init_device

```
int xbee_cmd_init_device ( xbee_dev_t * xbee )
```

DESCRIPTION

Initialize the AT Command layer for an XBee device. You need to call this function before any of the other xbee_cmd_* functions.

PARAMETERS

[in] xbee	XBee device on which to enable the AT Command layer. This function will automatically call xbee_cmd_query_device if it hasn't already been called for this device.

RETURN VALUE

0	The XBee device was successfully configured to send and receive AT commands.
-EINVAL	An invalid parameter was passed to the function

SEE ALSO

xbee_cmd_query_device(), xbee_cmd_create(), xbee_cmd_set_command(), xbee_cmd_set_callback(), xbee_cmd_set_target(), xbee_cmd_set_param(), xbee_cmd_set_param_bytes(), xbee_cmd_set_param_str(), xbee_cmd_send()

xbee_cmd_list_execute

```
int xbee_cmd_list_execute ( xbee_dev_t * xbee,
```

```
xbee_command_list_context_t FAR * clc,
 const xbee_atcmd_reg_t FAR * list,
 void FAR * base, const wpan_address_t FAR * address )
```

DESCRIPTION

Execute a list of AT commands.

PARAMETERS

[in, out] xbee	Device to execute commands
[out] clc	List head to set up. This must be static since callback functions access it asynchronously. Used as handle to check status of list execution.
[in] list	First entry of list of commands to execute. The list must be terminated by either XBEE_ATCMD_REG_END or XBEE_ATCMD_REG_END_CB. List entries are created using XBEE_ATCMD_REG macros etc.
[in] base	Base address of a structure to fill in with command results.
[in] address	Remote address, or NULL if local device.

RETURN VALUE

0	The XBee device was successfully configured to send and receive AT commands
-ENOSPC	The AT command table is full (increase the compile-time macro XBEE_CMD_REQUEST_TABLESIZE)
-EINVAL	An invalid parameter was passed to the function

SEE ALSO

xbee_cmd_list_status()

xbee_command_list_status

```
enum xbee_command_list_status xbee_cmd_list_status(
   xbee_command_list_context_t FAR *clc)
```

DESCRIPTION

Determine status of command list execution.

PARAMETERS

[in] clc	List head passed to xbee_cmd_list_execute().

RETURN VALUE

XBEE_COMMAND_LIST_RUNNING	Currently executing commands
XBEE_COMMAND_LIST_DONE	Successfully completed.
XBEE_COMMAND_LIST_TIMEOUT	Timed out.
XBEE_COMMAND_LIST_ERROR	Completed with error(s).

SEE ALSO

xbee_cmd_list_execute()

xbee_cmd_query_device

```
int xbee_cmd_query_device ( xbee_dev_t * xbee, uint_fast8_t refresh )
```

DESCRIPTION

Learn about the underlying device by sending a series of commands and storing the results in the xbee_dev_t.

This function will likely get called by the XBee stack at some point in the startup/initialization phase.

Use xbee_cmd_query_status() to check on the progress of querying the device.

PARAMETERS

[in,out] xbee	XBee device to query.
[in] refresh	If non-zero, just refresh the volatile values (e.g., network settings, as opposed to device serial number)

RETURN VALUE

0	Started querying device.
-EBUSY	Transmit serial buffer is full, or XBee is not accepting serial data (deasserting /CTS signal).

SEE ALSO

xbee_cmd_init_device(), xbee_cmd_query_status()

xbee_cmd_query_status

```
int xbee_cmd_query_status ( xbee_dev_t * xbee )
```

DESCRIPTION

Check the status of querying an XBee device, as initiated by xbee_cmd_query_device().

PARAMETERS

[in] xbee	Device to check

RETURN VALUE

0	Query completed.
-EINVAL	`xbee` is NULL.
-EBUSY	Query underway.
-ETIMEDOUT	Query timed out.
-EIO	Halted, but query may not have completed (unexpected response).

xbee_cmd_release_handle

```
int xbee_cmd_release_handle ( int16_t handle, uint16_t flags )
```

DESCRIPTION

Release an entry from the request table by marking it as available.

PARAMETERS

[in] handle	Handle to the request (as returned by xbee_cmd_create)

RETURN VALUE

0	Request freed
-EINVAL	`handle` is not valid

SEE ALSO

xbee_cmd_create(), xbee_cmd_release_handle()

xbee_cmd_send

```
int xbee_cmd_send ( int16_t handle )
```

DESCRIPTION

Send an AT Command to a local or remote XBee device.

PARAMETERS

[in] handle	Handle to the request, as returned by xbee_cmd_create().

RETURN VALUE

0	Frame sent
-EINVAL	`handle` is not valid
-EBUSY	Transmit serial buffer is full, or XBee is not accepting serial data (deasserting /CTS signal).

NOTE: If the request does not have a callback set, it will be automatically released if xbee_cmd_send() returns 0.

SEE ALSO

xbee_cmd_create(), xbee_cmd_set_command(), xbee_cmd_set_callback(), xbee_cmd_set_target(), xbee_cmd_set_param(), xbee_cmd_set_param_bytes(), xbee_cmd_set_param_str()

xbee_cmd_set_callback

```
int xbee_cmd_set_callback ( int16_t handle,
    xbee_cmd_callback_fn callback, void FAR * context )
```

DESCRIPTION

Associate a callback with a given AT Command request.

PARAMETERS

[in] handle	Handle to the request, as returned by xbee_cmd_create().

[in] callback	Callback function to receive the AT Command response. This function should take a single parameter (pointer to an xbee_cmd_response_t) and return either XBEE_ATCMD_DONE (if done with the request handle) XBEE_ATCMD_REUSE (if more responses are expected, or the request handle is going to be reused). ie: `int atcmd_callback( const xbee_cmd_response_t FAR *response)`
[in] context	Context (or "userdata") value to pass to the callback along with the AT Command response when it arrives. Should be set to NULL if not used.

RETURN VALUE

0	Callback set.
-EINVAL	`handle` is not valid.

SEE ALSO

xbee_cmd_create(), xbee_cmd_set_command(), xbee_cmd_set_target(), xbee_cmd_set_param(), xbee_cmd_set_param_bytes(), xbee_cmd_set_param_str(), xbee_cmd_send()

xbee_cmd_set_command

```
int xbee_cmd_set_command (int16_t handle, const char FAR command[3])
```

DESCRIPTION

Change the command associated with a request.

PARAMETERS

[in] handle	Handle to the request, as returned by xbee_cmd_create()
[in] command	Two-letter AT Command to send (e.g., "VR", "NI", etc.)

RETURN VALUE

0	Command changed.
-EINVAL	`handle` is not valid.

SEE ALSO

xbee_cmd_create(), xbee_cmd_set_callback(), xbee_cmd_set_target(), xbee_cmd_set_param(), xbee_cmd_set_param_bytes(), xbee_cmd_set_param_str(), xbee_cmd_send()

xbee_cmd_set_flags

```
int xbee_cmd_set_flags ( int16_t handle, uint16_t flags )
```

DESCRIPTION

Set the flags for a given AT Command request.

PARAMETERS

[in] handle	Handle to the request, as returned by xbee_cmd_create().
[in] flags	One or more of the following: XBEE_CMD_FLAG_QUEUE_CHANGE don't apply changes until ATAC or another command without this flag is sent

RETURN VALUE

0	Flags set.
-EINVAL	`handle` is not valid.

SEE ALSO

xbee_cmd_clear_flags()

xbee_cmd_set_param

```
int xbee_cmd_set_param ( int16_t handle, uint32_t value )
```

DESCRIPTION

Set the parameter (up to 32-bits) for a given AT Command request.

PARAMETERS

[in] handle	Handle to the request, as returned by xbee_cmd_create().

[in] value	Value to use as the parameter to the AT Command. For negative values, or values > 0xFFFFFFFF, use xbee_cmd_set_param_bytes(). For string parameters (e.g., for the "NI" command), use xbee_cmd_set_param_str().

RETURN VALUE

0	Parameter set.
-EINVAL	`handle` is not valid.

SEE ALSO

xbee_cmd_create(), xbee_cmd_set_command(), xbee_cmd_set_callback(), xbee_cmd_set_target(), xbee_cmd_set_param_bytes(), xbee_cmd_set_param_str(), xbee_cmd_send()

xbee_cmd_set_param_bytes

```
int xbee_cmd_set_param_bytes ( int16_t handle, const void FAR * data,
   uint8_t length )
```

DESCRIPTION

Set the parameter for a given AT Command request to a sequence of bytes.

PARAMETERS

[in] handle	Handle to the request, as returned by xbee_cmd_create().
[in] data	Pointer to bytes (MSB-first) to copy into request.
[in] length	Number of bytes to copy. 0 < length <= XBEE_CMD_MAX_PARAM_LENGTH

RETURN VALUE

0	Parameter set.
-EINVAL	`handle` or `length` is not valid.
-EMSGSIZE	`length` is greater than XBEE_CMD_MAX_PARAM_LENGTH.

SEE ALSO

xbee_cmd_create(), xbee_cmd_set_command(), xbee_cmd_set_callback(), xbee_cmd_set_target(), xbee_cmd_set_param(), xbee_cmd_set_param_str(), xbee_cmd_send()

xbee_cmd_set_param_str

```
int xbee_cmd_set_param_str ( int16_t handle, const char FAR * str )
```

DESCRIPTION

Set a string parameter for a given AT Command request (e.g., the "NI" node identifier command).

PARAMETERS

[in] handle	Handle to the request, as returned by xbee_cmd_create().
[in] str	String to use as the parameter. Must be less than XBEE_CMD_MAX_PARAM_LENGTH characters long.

RETURN VALUE

0	Parameter set.
-EINVAL	`handle` is not valid.
-EMSGSIZE	String is more than XBEE_CMD_MAX_PARAM_LENGTH characters.

SEE ALSO

xbee_cmd_create(), xbee_cmd_set_command(), xbee_cmd_set_callback(), xbee_cmd_set_target(), xbee_cmd_set_param(), xbee_cmd_set_param_bytes(), xbee_cmd_send()

xbee_cmd_set_target

```
int xbee_cmd_set_target ( int16_t handle, const addr64 FAR * ieee,
   uint16_t network)
```

DESCRIPTION

Associate a remote XBee device with a given AT Command request. By default, xbee_cmd_create() configures the request as a local AT Command for the serially-attached XBee module. Use this function to send the command to a remote device.

PARAMETERS

[in] handle	Handle to the request, as returned by xbee_cmd_create().

[in] ieee	Pointer to 64-bit IEEE hardware address of target, or NULL to switch back to the local XBee device. WPAN_IEEE_ADDR_BROADCAST for broadcast. WPAN_IEEE_ADDR_COORDINATOR for the coordinator (use WPAN_NET_ADDR_UNDEFINED for the network address). WPAN_IEEE_ADDR_UNDEFINED to use the 16-bit network address.
[in] network	16-bit network address of target. WPAN_NET_ADDR_UNDEFINED if the node's network address isn't known. WPAN_NET_ADDR_COORDINATOR for the coordinator (use the coordinator's actual IEEE address for the `ieee` parameter). WPAN_NET_ADDR_BCAST_NOT_ASLEEP to broadcast to all nodes that aren't sleeping. WPAN_NET_ADDR_BCAST_ROUTERS to broadcast to the coordinator and all routers

RETURN VALUE

0	Target set.
-EINVAL	`handle` is not valid.

SEE ALSO

xbee_cmd_create(), xbee_cmd_set_command(), xbee_cmd_set_callback(), xbee_cmd_set_param(), xbee_cmd_set_param_bytes(), xbee_cmd_set_param_str(), xbee_cmd_send()

xbee_cmd_simple

```
int xbee_cmd_simple ( xbee_dev_t * xbee, const char FAR command[3],
   uint32_t value )
```

DESCRIPTION

Simple interface for sending a command with a parameter to the local XBee without checking for a response.

PARAMETERS

[in] xbee	XBee device to use as the target.
[in] command	Two-letter AT Command to send (e.g., "ID", "CH", etc.).
[in] value	Value to use as the parameter to the AT Command.

RETURN VALUE

0	Command sent.
-EINVAL	An invalid parameter was passed to the function.
-EBUSY	Transmit serial buffer is full, or XBee is not accepting serial data (deasserting /CTS signal).

SEE ALSO

xbee_cmd_create(), xbee_cmd_set_command(), xbee_cmd_set_callback(), xbee_cmd_set_target(), xbee_cmd_set_param(), xbee_cmd_set_param_bytes(), xbee_cmd_set_param_str(), xbee_cmd_execute()

xbee_cmd_tick

```
int xbee_cmd_tick ( void )
```

DESCRIPTION

This function should be called periodically (at least every few seconds) to expire old entries from the AT Command Request table.

RETURN VALUE

>0	Number of requests expired.
0	None of the requests in the table expired.

xbee_identify

```
void xbee_identify ( xbee_dev_t * xbee, bool_t identify )
```

DESCRIPTION

Programs with the ZCL Identify Server Cluster can call this function in their main loop to have the XBee module's association LED flash fast (100ms cycle) when in Identify Mode.

PARAMETERS

[in] xbee	Device to identify
[in] identify	TRUE if XBee should be in identify mode

SEE ALSO

xbee_zcl_identify, zcl_identify_isactive, ZCL_CLUST_ENTRY_IDENTIFY_SERVER

5.4.2.5 Node Discovery API Functions and Macros

This section contains API descriptions for the node discovery component of the XBee Driver Layer. To use these functions the program must have **#include "XBee/discovery.h"** within the code.

xbee_disc_device_type_str

```
const char * xbee_disc_device_type_str ( uint8_t device_type )
```

DESCRIPTION

Return a string ("Coord", "Router", "EndDev", or "???") description for the "Device Type" field of 0x95 frames and ATND responses.

PARAMETERS

[in] device_type	The device type field from an 0x95 frame or ATND response

RETURN VALUE

0	Pointer to a string describing the device type, or string "???" if `device_type` is invalid.

xbee_disc_nd_parse

```
int xbee_disc_nd_parse ( xbee_node_id_t FAR * parsed, const void FAR
   * source )
```

DESCRIPTION

Parse a Node Discovery response and store it in an xbee_node_id_t structure.

PARAMETERS

[in,out] parsed	Pointer to an xbee_node_id_t structure to store parsed response into
[in] source	Pointer to an xbee_node_id_t structure

RETURN VALUE

0	Response was parsed correctly and stored.
-EINVAL	Invalid parameter or parsing error.

xbee_disc_node_id_dump

```
void xbee_disc_node_id_dump ( const xbee_node_id_t FAR * ni )
```

DESCRIPTION

Debugging function used to dump an xbee_node_id_t structure to stdout.

PARAMETERS

[in] ni	Pointer to an xbee_node_id_t structure

5.4.2.6 Firmware Update API Functions and Macros

This section contains API descriptions for the firmware update component of the XBee Driver Layer. To use these functions the program must have **#include "XBee/firmware.h"** within the code.

xbee_fw_buffer_init

```
int xbee_fw_buffer_init ( xbee_fw_buffer_t * fw, uint32_t length,
   const char FAR * address )
```

DESCRIPTION

Helper function for setting up an `xbee_fw_buffer_t` for use with a source firmware image held entirely in a buffer.

PARAMETERS

[out] fw	Structure to configure for reading firmware from a buffer
[in] length	Number of bytes in firmware image

[in] address	Address of buffer containing firmware

RETURN VALUE

0	Success
-EINVAL	Invalid parameter.

xbee_fw_install_ebl_tick

```
int xbee_fw_install_ebl_tick ( xbee_fw_source_t * source )
```

DESCRIPTION

Drive the firmware update process for boards that use .EBL files to store their firmware.

PARAMETERS

[in,out] source	Object used to track state of transfer

RETURN VALUE

0	Update is in progress.
1	Update completed successfully.
-EINVAL	`source` is invalid.
-EIO	General failure.
-ETIMEDOUT	Connection timed out waiting for data from target.

SEE ALSO

xbee_fw_install_init

xbee_fw_install_init

```
int xbee_fw_install_init ( xbee_dev_t * xbee, const wpan_address_t
   FAR * target, xbee_fw_source_t * source )
```

DESCRIPTION

Prepare to install new firmware on an attached XBee module.

The host must be able to control the reset pin of the XBee module.

PARAMETERS

[in] xbee — XBee device to install firmware on. Must have been set up with xbee_dev_init().

[in] target — The current version of this library can only update the local XBee module, so this parameter should always be NULL. When over-the-air (OTA) updating is supported, this paramter will be the address of a remote module to update.

[in] source — Structure with function pointers for seeking and reading from the new firmware image.

```
// Function prototypes for functions
that will provide firmware

// bytes when called by
xbee_fw_install_tick().

int my_firmware_seek( uint32_t off-
set);

int my_firmware_read( void FAR *buf-
fer, int16_t bytes);

xbee_fw_source_t        fw;

xbee_dev_t                      xbee;



fw.seek = &my_firmware_seek;

fw.read = &my_firmware_read;



xbee_dev_init( &xbee, ...);

xbee_fw_install_init( &xbee, &fw);
```

RETURN VALUE

0	Success.
-EINVAL	NULL parameter passed to function.
-EIO	XBee device passed to function doesn't have a callback for controlling the reset

pin on the XBee module.

xbee_fw_install_oem_tick

```
int xbee_fw_install_oem_tick ( xbee_fw_source_t * source )
```

DESCRIPTION

Install the firmware image stored in `source`.

You must call xbee_fw_install_init() on the source before calling this tick function.

If successful, XBee will be running the new firmware at 115,200 baud.

PARAMETERS

[in] source	Firmware source initialized with seek and read functions.

RETURN VALUE

1	Successfully installed new firmware.
0	Firmware installation in progress (incomplete).
-EILSEQ	Firmware does not contain a valid firmware image.
-EBADMSG	Firmware checksum failed, image is bad.
-EIO	Couldn't establish communications with XBee module.
-EFTYPE	Firmware is not compatible with this hardware.

xbee_fw_status_ebl

```
char FAR * xbee_fw_status_ebl( xbee_fw_source_t * source, char FAR *
   buffer )
```

DESCRIPTION

Update buffer with the current install status of `source`.

PARAMETERS

[in] source	State variable to generate status string for.

[out] buffer	Buffer (at least 80 bytes) to receive dynamic status string.

RETURN VALUE

0 Returns pointer to the buffer given as parameter 2 or a fixed status string.

xbee_fw_status_oem

```
char FAR * xbee_fw_status_oem( xbee_fw_source_t * source, char FAR *
   buffer )
```

DESCRIPTION

Update buffer with the current install status of `source`.

PARAMETERS

[in] source	State variable to generate status string for.
[out] buffer	Buffer (at least 80 bytes) to receive dynamic status string.

RETURN VALUE

0 Returns pointer to the buffer given as parameter 2 or a fixed status string.

5.5 Wireless Personal Area Network Layer (WPAN)

This section contains information about the Wireless Personal Area Network Layer for ZigBee-capable devices. This layer adds the concepts of endpoints and clusters.

5.5.1 General Overview

The Wireless Personal Area Network Layer handles endpoint processing through an endpoint table and an endpoint dispatcher. In simplistic terms, it could be looked at as a mail delivery system. It introduces the envelope structure which contains the network addresses of the sender and receiver. The local node address will always be the receiver address on received envelopes and the sender address on outgoing envelopes.

The endpoint table is a complex data structure that describes all endpoints, clusters, ZCL attributes and manufacturer-specific command handlers for a given device. Note that this structure includes storage for ZigBee driver layer information such as attributes and manufacturer-specific command handlers. Although included in this structure, the ZigBee driver layer processes this information. The WPAN layer passes endpoint and cluster frames off to the specified handlers, which are typically in the ZigBee driver layer. The structure of the table is as follows (note that not all members of each object are listed):

- Each DEVICE (wpan_dev_t) corresponds to a local, serially-connected XBee module. This DEVICE has multiple ENDPOINTs.
- Each ENDPOINT (wpan_endpoint_table_entry_t) has:

- An ENDPOINT ID (0 to 254), PROFILE ID, DEVICE ID and DEVICE VERSION.
- Multiple CLUSTERs.
- A HANDLER to process frames for CLUSTERs without their own HANDLER, or CLUSTERs that aren't in the table.

- Each CLUSTER (wpan_cluster_table_entry_t) has:
  - FLAGs indicating whether it is an input or output cluster (or both), and whether packets sent to/from the cluster require APS-layer encryption.
  - A HANDLER to process frames for that cluster.
  - A CONTEXT pointer that is passed to the HANDLER. For a ZCL endpoint the CONTEXT points to an ATTRIBUTE TREE.
- On ZCL endpoints, each entry in the ATTRIBUTE TREE (zcl_attribute_tree_t) has:
  - A manufacturer ID with ZCL_MFG_NONE (0) representing the general attributes for the cluster.
  - Pointers to a list of SERVER and CLIENT cluster ATTRIBUTEs.
- Each ATTRIBUTE is either a BASE ATTRIBUTE (zcl_attribute_base_t) with:
  - A 16-bit ID.
  - FLAGS indicating whether the attribute is read-only, is a full attribute (see below), has a minimum or maximum limit and is reportable.
  - A ZCL TYPE.
  - A pointer to the ATTRIBUTE's VALUE.
- ... or a FULL ATTRIBUTE (zcl_attribute_full_t) which has
  - A BASE ATTRIBUTE structure (so both BASE and FULL attributes start with the same structure elements).
  - Optional MINIMUM and MAXIMUM (zcl_attribute_minmax_t) values.
  - A READ function (zcl_attribute_update_fn) to refresh the ATTRIBUTE's VALUE.
  - A WRITE function (zcl_attribute_write_fn) to process a ZCL Write Attributes request if the attribute requires additional processing over what the standard function, zcl_decode_attribute, does.

Endpoint Dispatching:

The endpoint dispatcher searches the endpoint table to find a handler for a given frame, based on its destination endpoint and cluster ID.

5.5.2 API Functions and Macros

This section contains API descriptions for the WPAN Layer functions and macros separated into layer components.

5.5.2.1 Cluster/Endpoint API Functions and Macros

This section contains API descriptions for the firmware update component of the XBee Driver Layer. To use these functions the program must have **#include "wpan/aps.h"** within the code.

wpan_cluster_match

```
const wpan_cluster_table_entry_t *wpan_cluster_match ( uint16_t
   match, uint8_t mask, const wpan_cluster_table_entry_t * entry )
```

DESCRIPTION

Search a cluster table for a matching cluster ID.

PARAMETERS

[in] match	ID to match
[in] mask	Flags to match against the flags member of the wpan_cluster_table_entry_t structure. If any flags match, the entry is returned. Typically one of: • WPAN_CLUST_FLAG_INPUT (or WPAN_CLUST_FLAG_SERVER) • WPAN_CLUST_FLAG_OUTPUT (or WPAN_CLUST_FLAG_CLIENT)
[in] entry	Pointer to list of entries to search

RETURN VALUE

NULL	`entry` is invalid or search reached WPAN_CLUSTER_END_OF_LIST with out finding a match.
!NULL	Pointer to matching entry from table.

SEE ALSO

wpan_endpoint_get_next(), wpan_endpoint_match()

wpan_conversation_delete

```
void wpan_conversation_delete (wpan_conversation_t FAR *
   conversation)
```

DESCRIPTION

Delete a conversation from an endpoint's conversation table.

PARAMETERS

[in,out] conversation	Conversation to delete

wpan_endpoint_get_next

```
const wpan_endpoint_table_entry_t *wpan_endpoint_get_next (
   wpan_dev_t * dev, const wpan_endpoint_table_entry_t * ep )
```

DESCRIPTION

Function used to walk a device's endpoint table. For most devices, this will just walk the entries in `dev->endpoint_table`. For custom applications a function may dynamically return entries.

Use of this function is the only way to walk the endpoint table.

PARAMETERS

[in] dev	device with endpoint table to walk
[in] ep	NULL to return first entry in table, or a pointer previously returned by this function to get the next entry

RETURN VALUE

NULL	`dev` is invalid or reached end of table.
!NULL	Next entry from table.

SEE ALSO

wpan_endpoint_match(), wpan_cluster_match()

wpan_conversation_register

```
int wpan_conversation_register (wpan_ep_state_t FAR * state,
   wpan_response_fn handler,  const void FAR * context, uint16_t
   timeout )
```

DESCRIPTION

Add a conversation to the table of tracked conversations.

PARAMETERS

[in,out] state	Endpoint state associated with sending endpoint
[in] handler	Handler to call when responses come back, or NULL to increment and return the endpoint's transaction ID

[in] context	Pointer stored in conversation table and passed to call-back handler
[in] timeout	Number of seconds before generating timeout, or 0 for none

RETURN VALUE

0-255	Transaction ID to use in sent frame
-EINVAL	State is invalid (NULL)
-ENOSPC	Table is full

SEE ALSO

wpan_endpoint_next_trans

wpan_conversation_timeout

```
void wpan_conversation_timeout ( wpan_ep_state_t FAR * state )
```

DESCRIPTION

Walk an endpoint's conversation table and expire any conversations that have timed out.

PARAMETERS

[in] state	Endpoint state (from endpoint table)

wpan_endpoint_dispatch

```
int wpan_endpoint_dispatch (wpan_envelope_t FAR * envelope )
```

DESCRIPTION

Find the matching endpoint for the provided `envelope` and have it process the frame. In the case of the broadcast endpoint (255), dispatches the frame to all endpoints matching the envelope's profile ID.

Looks up the destination endpoint and cluster ID in the device's endpoint table and passes `envelope` off to the cluster handler (if a matching cluster was found) or the endpoint handler.

PARAMETERS

[in] envelope	Structure containing all necessary information about message (endpoints, cluster, profile, etc.)

RETURN VALUE

0	Successfully dispatched message.
-ENOENT	No handler for this endpoint/cluster.
!0	Error dispatching messages.

wpan_endpoint_match

```
const wpan_endpoint_table_entry_t wpan_endpoint_match (
   wpan_dev_t * dev, uint8_t endpoint, uint16_t profile_id )
```

DESCRIPTION

Walk a device's endpoint table looking for a matching endpoint ID and profile ID.

Used by the endpoint dispatcher and ZDO layer to describe available endpoints on this device.

PARAMETERS

[in] dev	Device with endpoint table to search.
[in] endpoint	Endpoint number to search for.
[in] profile_id	Profile to match or WPAN_APS_PROFILE_ANY to search on endpoint number only.

RETURN VALUE

NULL	`dev` is invalid or reached end of table without finding a match.
!NULL	Next entry from table.

SEE ALSO

wpan_endpoint_of_cluster(), wpan_endpoint_get_next(), wpan_cluster_match(), wpan_endpoint_of_envelope()

wpan_endpoint_next_trans

```
uint8_t wpan_endpoint_next_trans (const wpan_endpoint_table_entry_t
   FAR * ep )
```

DESCRIPTION

Increment and return the endpoint's transaction ID counter.

PARAMETERS

[in] ep Entry from endpoint table

RETURN VALUE

0-255 Current transaction ID for endpoint.

wpan_endpoint_of_cluster

```
const wpan_endpoint_table_entry_t *wpan_endpoint_of_cluster (
   wpan_dev_t * dev, uint16_t profile_id,uint16_t cluster_id, uint8_t
   mask )
```

DESCRIPTION

Walk a device's endpoint table looking for a matching profile ID and cluster ID.

Used to find the correct endpoint to use for sending ZCL client requests.

PARAMETERS

[in] dev Device with endpoint table to search.

[in] profile_id Profile to match or WPAN_APS_PROFILE_ANY to search on cluster ID only.

[in] cluster_id Cluster ID to search for

[in] mask Flags to match against the flags member of the wpan_cluster_table_entry_t

structure. If any flags match, the entry is returned. Typically one of

- WPAN_CLUST_FLAG_INPUT (or WPAN_CLUST_FLAG_SERVER)
- WPAN_CLUST_FLAG_OUTPUT (or WPAN_CLUST_FLAG_CLIENT)

RETURN VALUE

NULL dev is invalid or reached end of table without finding a match.

!NULL Matching entry from table.

SEE ALSO

wpan_endpoint_match(), wpan_endpoint_get_next(), wpan_cluster_match()

wpan_endpoint_of_envelope

```
const wpan_endpoint_table_entry_t *wpan_endpoint_of_envelope (
   const wpan_envelope_t * env )
```

DESCRIPTION

Look up the endpoint table entry for the source endpoint of an envelope.

PARAMETERS

[in] env	Envelope for lookup. Uses env->source_endpoint and env->profile_id to find the endpoint table entry for env->dev.

RETURN VALUE

NULL	Reached end of table without finding a match.
!NULL	Entry from table.

SEE ALSO

wpan_endpoint_of_cluster(), wpan_endpoint_get_next(), wpan_cluster_match(), wpan_endpoint_match()

wpan_envelope_create

```
void wpan_envelope_create (wpan_envelope_t * envelope,
   wpan_dev_t * dev, const addr64 * ieee, uint16_t network_addr )
```

DESCRIPTION

Starting with a blank envelope, fill in the device, 64-bit IEEE address and 16-bit network address of the destination.

PARAMETERS

[out] envelope	Envelope to populate

[in] dev — Device that will send this envelope

[in] ieee — 64-bit IEEE/MAC address of recipient or one of:

- WPAN_IEEE_ADDR_COORDINATOR (send to coordinator)
- WPAN_IEEE_ADDR_BROADCAST (broadcast packet)
- WPAN_IEEE_ADDR_UNDEFINED (use network address only)

[in] network_addr — 16-bit network address of recipient or one of:

- WPAN_NET_ADDR_COORDINATOR for the coordinator (use the coordinator's actual IEEE address for the `ieee` parameter).
- WPAN_NET_ADDR_BCAST_ALL_NODES (broadcast to all nodes)
- WPAN_NET_ADDR_BCAST_NOT_ASLEEP (broadcast to awake nodes)
- WPAN_NET_ADDR_BCAST_ROUTERS (broadcast to routers/coordinator)
- WPAN_NET_ADDR_UNDEFINED (use 64-bit address only)

When sending broadcast packets, use WPAN_IEEE_ADDR_BROADCAST and WPAN_NET_ADDR_UNDEFINED. Don't set both addresses to broadcast.

SEE ALSO

wpan_envelope_reply()

wpan_envelope_dump

```
void wpan_envelope_dump (const wpan_envelope_t FAR * envelope )
```

DESCRIPTION

Debugging function to dump the contents of an envelope to stdout. Displays all fields from the envelope, plus the contents of the payload.

PARAMETERS

[in] envelope — Envelope to dump

wpan_envelope_reply

```
int wpan_envelope_reply (wpan_conversation_t FAR * conversation )
```

DESCRIPTION

Create a reply envelope based on the envelope received from a remote node.

Copies the interface, addresses, profile and cluster from the original envelope, and then swaps the source and destination endpoints.

Note: It may be necessary for the caller to change the cluster_id as well, after building the reply envelope. For example, ZDO responses set the high bit of the request's cluster ID.

PARAMETERS

[out] reply	Buffer for storing the reply envelope.
[in] original	Original envelope, received from a remote node, to base the reply on.

RETURN VALUE

0	Addressed reply envelope.
-EINVAL	Either parameter is NULL or they point to the same address.

SEE ALSO

wpan_envelope_create()

wpan_envelope_send

```
int wpan_envelope_send (const wpan_envelope_t FAR * envelope )
```

DESCRIPTION

Send a message to an endpoint using address and payload information stored in a wpan_envelope_t structure.

PARAMETERS

[in] envelope	Envelope of request to send

RETURN VALUE

0	Request sent
!0	Error trying to send request

wpan_tick

```
int wpan_tick (wpan_dev_t * dev )
```

DESCRIPTION

Calls the underlying hardware to process received frames and times out expired conversations. See section 5.1.4 for a discussion of how frequently this function needs to be called for a given configuration.

PARAMETERS

[in] dev	WPAN device to tick

RETURN VALUE

>=0	Number of frames processed during the tick.
-EINVAL	Device does not have a tick function assigned to it.
<0	Some other error encountered during the tick.

xbee_wpan_init

```
int xbee_wpan_init ( xbee_dev_t * xbee, const
   wpan_endpoint_table_entry_t * ep_table )
```

DESCRIPTION

Configure XBee device for APS-layer (endpoint/cluster) networking. If using this layer, be sure to call wpan_tick (instead of xbee_dev_tick) so it can manage the APS layers of the network stack.

PARAMETERS

[in] xbee	Pointer to an XBee device structure
[in] index	Pointer to a WPAN endpoint table to use with this device

RETURN VALUE

0	Success
!0	Error

5.5.2.2 Datatypes and Support API Functions and Macros

This section contains API descriptions for the firmware update component of the XBee Driver Layer. To use these functions the program must have **#include "wpan/types.h"** within the code.

addr64_equal

```
bool_t addr64_equal ( const addr64 FAR * addr1, const addr64 FAR *
   addr2 )
```

DESCRIPTION

Compare two 64-bit addresses for equality.

PARAMETERS

[in] addr1	Address to compare
[in] addr2	Address to compare

RETURN VALUE

TRUE	`addr1` and `addr2` are not NULL and point to identical addresses.
FALSE	NULL parameter passed in, or addresses differ.

addr64_format

```
char FAR * addr64_format ( char FAR * buffer, const addr64 FAR *
   address )
```

DESCRIPTION

Format a 64-bit address as a null-terminated, printable string (e.g., "00-13-A2-01-23-45-67").

To change the default separator ('-'), define ADDR64_FORMAT_SEPARATOR to any character. For example:

```
#define ADDR64_FORMAT_SEPARATOR ':'
```

PARAMETERS

[out] buffer	Pointer to a buffer of at least ADDR64_STRING_LENGTH (8 2-character bytes + 7 separators + 1 null = 24) bytes.
[in] address	64-bit address to format.

RETURN VALUE

`address`	As a printable string (stored in `buffer`).

addr64_is_zero

```
bool_t addr64_is_zero (const addr64 FAR * addr)
```

DESCRIPTION

Test a 64-bit address for zero.

PARAMETERS

[in] addr	Address to test

RETURN VALUE

TRUE	`addr` is NULL or points to an all-zero address
FALSE	`addr` points to a non-zero address

SEE ALSO

WPAN_IEEE_ADDR_ALL_ZEROS

addr64_parse

```
int addr64_parse ( addr64 * address_be, const char FAR * str )
```

DESCRIPTION

Parse a text string into a 64-bit IEEE address.

Converts a text string with eight 2-character hex values, with an optional separator between any two values. For example, the following formats are all valid:

- 01-23-45-67-89-ab-cd-ef
- 01234567-89ABCDEF
- 01:23:45:67:89:aB:Cd:EF
- 0123 4567 89AB cdef

PARAMETERS

[out] address_be	Converted address (stored big-endian)
[in] str	String to convert, starting with first hex character

RETURN VALUE

0	String converted
-EINVAL	Invalid parameters passed to function; if `address_be` is not NULL,

it will be set to all zeros.

5.6 ZigBee Driver Layer

This section contains information about the ZigBee Driver Layer for ZigBee-capable devices.

5.6.1 General Overview

The ZigBee Driver Layer is comprised of two components, the ZigBee Device Object / Profile component and the ZigBee Cluster Library component. This layer adds cluster attributes and handles complete endpoint and cluster discovery. This includes discovering endpoints, clusters and attributes of remote devices, as well as reporting this data on for local endpoints and clusters on incoming discovery requests. It also has commands to allow reading and writing of attribute values.

5.6.2 ZigBee Device Object/ Profile Component (ZDO / ZDP)

This section describes the functionality and usage of the ZigBee Device Object / Profile Library and contains descriptions for library functions and macros.

5.6.2.1 General Overview

The ZDO endpoint handler (registered to endpoint 0) walks the endpoint table to respond to requests. It needs to know about all endpoints and their input clusters and output clusters. It walks the endpoint table, but stops at the context elements. Therefore, all information required by the ZDO layer must exist outside of the context elements of the endpoint table.

5.6.2.2 ZigBee Device Object / Profile API Functions and Macros

This section contains API descriptions for the ZigBee Device Object / Profile Layer functions and macros. To use these functions the program must have **#include "zigbee/zdo.h"** within the code.

---

zdo_ endpoint_state

```
wpan_ep_state_t FAR * zdo_endpoint_state ( wpan_dev_t * dev )
```

DESCRIPTION

Returns the ZDO endpoint's state if a device has a ZDO endpoint.

PARAMETERS

[in] dev Device to query

RETURN VALUE

NULL `dev` does not have a ZDO endpoint.

!NULL Address of wpan_ep_state_t variable used for state tracking.

zdo_handler

```
int zdo_handler ( const wpan_envelope_t FAR * envelope,
   wpan_ep_state_t FAR * ep_state )
```

DESCRIPTION

Process ZDO frames (received on endpoint 0 with Profile ID 0).

PARAMETERS

[in] envelope	Envelope of received ZDO frame, contains address, endpoint, profile, and cluster info.
[in] ep_state	Pointer to endpoint's state structure (used for tracking transactions)

RETURN VALUE

0	Successfully processed.
!0	Error trying to process frame.

zdo_match_desc_request

```
int zdo_match_desc_request ( void * buffer, int16_t buflen,
   uint16_t addr_of_interest, uint16_t profile_id, const uint16_t *
   inClust, const uint16_t * outClust )
```

DESCRIPTION

Generate a Match_Desc (Match Descriptor) request (ZigBee spec 2.4.3.1.7) to send on the network.

PARAMETERS

[out] buffer	Buffer to hold generated request.
[in] buflen	Size of buffer used to hold generated request.
[in] addr_of_interest	See ZDO documentation for NWKAddrOfInterest.
[in] profile_id	Profile ID to match, must be an actual profile ID (cannot be WPAN_APS_PROFILE_ANY).

[in] inClust	List of input clusters, ending with WPAN_CLUSTER_END_OF_LIST. Can use NULL if there aren't any input clusters.
[in] outClust	List of output clusters, ending with WPAN_CLUSTER_END_OF_LIST. Can use NULL if there aren't any output clusters.

RETURN VALUE

-ENOSPC	Buffer isn't large enough to hold request; need 7 bytes plus (2 * the number of clusters).
>0	Number of bytes written to buffer.

zdo_mgmt_leave_req

```
int zdo_mgmt_leave_req( wpan_dev_t * dev, const addr16 * address,
   uint16_t flags )
```

DESCRIPTION

Send a ZDO Management Leave Request.

PARAMETERS

[in] dev	Device to send request on
[in] address	Address to send request to, or NULL for self-addressed
[in] flags	One or more of the following flags: • ZDO_MGMT_LEAVE_REQ_FLAG_NONE • ZDO_MGMT_LEAVE_REQ_FLAG_REMOVE_CHILDREN - sets the Remove Children flag in the ZDO request • ZDO_MGMT_LEAVE_REQ_FLAG_REJOIN - sets the Rejoin flag in the ZDO request • ZDO_MGMT_LEAVE_REQ_ENCRYPTED - sends the request with APS encryption

RETURN VALUE

0	Successfully sent request.
-EINVAL	Bad parameter passed to function.
!0	Error sending request.

zdo_send_bind_req

```
int zdo_send_bind_req ( wpan_envelope_t * envelope, uint16_t type,
   uint16_t flags )
```

DESCRIPTION

Send a ZDO Bind (or Unbind) Request to the destination address in the envelope.

Binds `.dest_endpoint` on `.ieee_address` to `.source_endpoint` on `.dev` using `.cluster_id`.

Ignores the `.options`, `.payload`, and `.length` members of the envelope.

PARAMETERS

[in] envelope	Addressing information used for the Bind Request.
[in] type	ZDO_BIND_REQ for a Bind Request or ZDO_UNBIND_REQ for an Unbind Request; all other values are invalid.
[in] callback	Callback to receive Bind/Unbind (or Default) Response; NULL if you don't care about the response
[in] context	Context passed to callback

RETURN VALUE

0	Successfully sent request.
-EINVAL	Bad parameter passed to function.
!0	Error sending request.

zdo_send_descriptor_req

```
int zdo_send_descriptor_req ( wpan_envelope_t * envelope,
   uint16_t cluster, uint16_t addr_of_interest,wpan_response_fn
   callback, const void FAR * context )
```

DESCRIPTION

Send a ZDO Node, Power, Complex or User Descriptor request, or an Active Endpoint request.

PARAMETERS

[in,out] envelope	Envelope created by wpan_envelope_create; this function will fill in the cluster and reset the payload and length.
[in] cluster	Any ZDO request with a transaction and 16-bit network address as its only fields, including: • ZDO_NODE_DESC_REQ • ZDO_POWER_DESC_REQ • ZDO_ACTIVE_EP_REQ • ZDO_COMPLEX_DESC_REQ • ZDO_USER_DESC_REQ
[in] addr_of_interest	Address to use in ZDO request
[in] callback	Function to receive response
[in] context	Context to pass to callback with response

RETURN VALUE

!0	Error sending request.
0	Request sent.

zdo_send_nwk_addr_req

```
int zdo_send_nwk_addr_req ( wpan_dev_t * dev, const addr64 * ieee,
   uint16_t FAR * net )
```

DESCRIPTION

Given a device's IEEE (64-bit) address, get its 16-bit network address by unicasting a ZDO NWK_addr request to it.

After calling this function, *net is set to WPAN_NET_ADDR_UNDEFINED. When a response comes back, *net is set to the 16-bit network address in the response. If a timeout occurs waiting for a response, *net is set to WPAN_NET_ADDR_BCAST_ALL_NODES. (So the caller needs to wait until (*net != WPAN_NET_ADDR_UNDEFINED)).

PARAMETERS

[in] dev	wpan_dev_t to send request
[in] ieee	IEEE address of device we're seeking a network address for

[out] net	Location to store the 16-bit network address when the NWD_addr response comes back.

RETURN VALUE

-EINVAL	Invalid parameter passed to function.
-ENOSPC	Conversation table is full, wait and try sending later.
0	Request sent.
!0	Error trying to send request.

zdo_send_response

```
int zdo_send_response ( const wpan_envelope_t * request,     const
   void FAR * response, uint16_t length )
```

DESCRIPTION

Send a response to a ZDO request.

Automatically builds the response envelope and sets its cluster ID (to the request's cluster ID with the high bit set) before sending.

PARAMETERS

[in] request	Envelope of original request
[in] response	Frame to send in response
[in] length	Length of response

RETURN VALUE

0	Sent response.
!0	Error sending response.

5.6.3 ZigBee Cluster Library

This section describes the functionality and usage of the ZigBee Cluster Library and contains descriptions for library functions and macros.

5.6.3.1 General Overview

The ZigBee Cluster Library can support a cluster with manufacturer-specific attributes and commands for more than one manufacturer ID. The handler registered to that cluster would check the frame type and manufacturer-specific bits and hand any GENERAL/PROFILE or MANUFACTURER-SPECIFIC commands off to the ZCL General Command Handler (zcl_general_command).

The ZCL General Command Handler will see frames from the endpoint dispatcher for:

- Clusters that aren't in the cluster table for the endpoint (invalid clusters).
- Clusters that don't have their own handler (no cluster commands).

It also receives frames from clusters with handlers for cluster-specific commands that are passing on a general or manufacturer-specific command frame.

The ZCL General Command Handler finds the correct attribute list form the attribute tree (general vs. manufacturer-specific, server vs. client cluster), and then:

- If the frame is marked CLUSTER & MANUFACTURER-SPECIFIC, it passes the frame on to the handler for that MFG ID (stored in the attribute tree).
- If it is marked CLUSTER-SPECIFIC but GENERAL, it generates an error.
- If it is marked PROFILE-SPECIFIC and GENERAL, it processes the frame as a general command, using the attribute list it retrieved from the tree.

Summary of ZCL Handlers:

- ZCL General Command Handler registered to the endpoints to process frames for invalid clusters or clusters without cluster-specific commands (ie., clusters with a NULL command handler).
- Cluster-specific, general command handler registered to each cluster.
- Cluster-specific, manufacturer-specific command handler(s) listed in the attribute tree (stored in the cluster structure's context member) under the appropriate manufacturer ID.

5.6.3.2 ZigBee Cluster Library API Functions and Macros

This section contains API descriptions for the primary ZigBee Cluster Library functions and macros.

To use these functions the program must have **#include "zigbee/zcl.h"** within the code.

zcl_build_header

```
int zcl_build_header ( zcl_header_response_t * rsp, zcl_command_t *
   cmd )
```

DESCRIPTION

Support function to fill in a ZCL response header.

Set the `frame_control`, `sequence` and optional `mfg_code` fields of the ZCL response header, and return the offset to the actual start of the header (non-mfg specific skips the first two bytes).

PARAMETERS

[out] rsp	Buffer to store header of response.
[in] cmd	Command we're responding to.

RETURN VALUE

0	Responding to manufacturer-specific command.
2	Responding to non-manufacturer-specific command.
-EINVAL	Invalid parameter passed to function.

zcl_check_minmax

```
int zcl_check_minmax ( const zcl_attribute_base_t * entry, const
   uint8_t FAR * buffer_le )
```

DESCRIPTION

Checks whether a new value from a Write Attributes command is within the limits specified for a given attribute.

PARAMETERS

[in] entry	Entry to use for min/max check.
[in] buffer_le	Buffer with new value (in little-endian byte order), from write attributes command.

RETURN VALUE

0	New value is within range (or attribute doesn't have a min/max).
!0	Value is out of range.
-EINVAL	Invalid parameter passed to function.

ZCL_CMD_IS_CLUSTER

```
bool ZCL_CMD_IS_CLUSTER ( const void FAR * p )
```

DESCRIPTION

Macro for checking the frame control field of a ZCL command. Used to identify cluster commands that are not manufacturer specific.

VALUE

```
(ZCL_FRAME_TYPE_CLUSTER ==                \
   ((const uint8_t FAR *)(p) & (ZCL_FRAME_TYPE_MASK |
ZCL_FRAME_MFG_SPECIFIC)))
```

PARAMETERS

[in] p Pointer to the frame control field of a ZCL command structure:

(zcl_command_t.frame_control)

RETURN VALUE

TRUE ZCL command is cluster specific but not manufacturer specific.

FALSE ZCL command doesn't have the manufacturer-specific bit set, or is a profile-wide command.

ZCL_CMD_IS_MFG_CLUSTER

```
bool ZCL_CMD_IS_MFG_CLUSTER ( const void FAR * p )
```

DESCRIPTION

Macro for checking the frame control field of a ZCL command. Used to identify cluster commands that are manufacturer specific.

VALUE

```
((ZCL_FRAME_TYPE_CLUSTER | ZCL_FRAME_MFG_SPECIFIC) ==          \
((const uint8_t FAR *)(p) & (ZCL_FRAME_TYPE_MASK |
ZCL_FRAME_MFG_SPECIFIC)))
```

PARAMETERS

[in] p Pointer to the frame control field of a ZCL command structure:

(zcl_command_t.frame_control)

RETURN VALUE

TRUE ZCL command is both manufacturer and cluster specific.

FALSE ZCL command doesn't have the manufacturer-specific bit set, or is a profile-wide Command.

ZCL_CMD_IS_PROFILE

```
bool ZCL_CMD_IS_PROFILE ( const void FAR * p )
```

DESCRIPTION

Macro for checking the frame control field of a ZCL command. Used to identify commands that act across the entire profile. Ignores the manufacturer-specific bit (which specifies use of manufacturer-specific attributes).

VALUE

```
(ZCL_FRAME_TYPE_PROFILE ==            \
((const uint8_t FAR *)(p) & ZCL_FRAME_TYPE_MASK ))
```

PARAMETERS

[in] p	Pointer to the frame control field of a ZCL command structure: (`zcl_command_t.frame_control`)

RETURN VALUE

TRUE	ZCL command is not cluster-specific (profile command).
FALSE	ZCL command is cluster-specific.

ZCL_CMD_MATCH

`bool ZCL_CMD_MATCH ( const void FAR * p )`

DESCRIPTION

Macro for checking the frame control field of a ZCL command. Used to filter out commands that are (or are not) handled by the current function.

VALUE

```
((*(const uint8_t FAR *)(p) & (ZCL_FRAME_MFG_SPECIFIC |      \
ZCL_FRAME_DIRECTION | ZCL_FRAME_TYPE_MASK)) ==         \
(ZCL_FRAME_ ## mfg | ZCL_FRAME_ ## dir | ZCL_FRAME_TYPE_ ##
type))
```

PARAMETERS

[in] p	Pointer to the frame control field of a ZCL command structure: (`zcl_command_t.frame_control`)
[in] mfg	Non-delimited string values of either GENERAL or MFG_SPECIFIC to filter the manufacturer specific bit being either clear or set.

[in] dir	Non-delimited string values of either CLIENT_TO_SERVER or SERVER_TO_CLIENT to filter on the desired direction
[in] type	Non-delimited string values of either PROFILE or CLUSTER to filter for the desired frame type, profile wide or cluster specific.

RETURN VALUE

TRUE	ZCL command matches all three filters.
FALSE	ZCL command does not match.

zcl_command_build

```
int zcl_command_build ( zcl_command_t * cmd, const wpan_envelope_t
   FAR * envelope, zcl_attribute_tree_t FAR * tree )
```

DESCRIPTION

Parse a ZCL request and store in a `zcl_command_t` structure with fields in fixed locations (therefore easier to use than the variable-length frame header).

Also uses the direction bit and manufacturer ID to search the attribute tree for the correct list of attributes.

PARAMETERS

[out] cmd	Buffer to store parsed ZCL command
[in] envelope	Envelope from received message
[in] tree	Pointer to attribute tree for cluster (typically passed in via endpoint dispatcher)

RETURN VALUE

0	Parsed payload from envelope into `cmd`.
-EINVAL	Invalid parameter passed to function.
-EBADMSG	Frame is too small for ZCL header's frame_control value.

zcl_command_dump

```
int zcl_command_dump ( zcl_command_t * cmd )
```

DESCRIPTION

Debugging routine that dumps contents of a parsed ZCL command structure to STDOUT.

PARAMETERS

[in] cmd	zcl_command_t structure populated by zcl_command_build()

zcl_convert_24bit

```
uint32_t zcl_convert_24bit ( const void FAR * value_le, bool_t
   extend_sign )
```

DESCRIPTION

Convert a 24-bit (3-byte) little-endian value to a 32-bit value in host byte order.

PARAMETERS

[in] value_le	Pointer to 3 bytes to convert
[in] extend_sign	If TRUE, set high byte of result to 0xFF if top bit of 24-bit value is set

RETURN VALUE

A 32-bit, host-byte-order version of the 24-bit, little-endian value passed in.

zcl_decode_attribute

```
int zcl_decode_attribute ( const zcl_attribute_base_t FAR * entry,
   zcl_attribute_write_rec_t * rec )
```

DESCRIPTION

Decode attribute value from a Write Attribute Request or Read Attribute Response record and optionally write to entry.

PARAMETERS

[in] entry	Entry from attribute table
[in,out] rec	State information for parsing write request

RETURN VALUE

>=0	Number of bytes consumed from `rec->buffer`.
-EINVAL	Invalid parameter passed to function.

zcl_default_response

```
int zcl_default_response ( zcl_command_t * request, uint8_t status )
```

DESCRIPTION

Send a Default Response (ZCL_CMD_DEFAULT_RESP) to a given command.

PARAMETERS

[in] cmd	Command we're responding to
[in] status	Status to use in the response (see ZCL_STATUS_* macros under "ZCL Status Enumerations" in zigbee/zcl.h)

RETURN VALUE

0	Successfully sent response, or response not required (message was broadcast, sent to the broadcast endpoint, or sender set the disable default response bit).
!0	Error on send.
-EINVAL	Invalid parameter passed to function.

zcl_encode_attribute_value

```
int zcl_encode_attribute_value ( uint8_t FAR * buffer,int16_t
   bufsize, const zcl_attribute_base_t FAR * entry )
```

DESCRIPTION

Format a ZCL attribute for a Read Attributes Response.

Copies the value of attribute entry to `buffer` in little-endian byte order. Will not write more than `bufsize` bytes. Used to build ZCL frames.

PARAMETERS

[out] buffer	Buffer to store encoded attribute

[in] bufsize	Number of bytes available in buffer
[in] entry	Attribute to encode into buffer

RETURN VALUE

-ZCL_STATUS_SOFTWARE_FAILURE	Invalid parameter passed in.
-ZCL_STATUS_DEFINED_OUT_OF_BAND	Ghost attribute with NULL `value`.
-ZCL_STATUS_INSUFFICIENT_SPACE	Buffer too small to encode value.
-ZCL_STATUS_FAILURE	Unknown/unsupported attribute type.
-ZCL_STATUS_HARDWARE_FAILURE	Failure updating attribute value.
-ZCL_STATUS_SOFTWARE_FAILURE	Failure updating attribute value >=0 number of bytes written.

zcl_find_attribute

```
zcl_attribute_base_t FAR * zcl_find_attribute ( const
   zcl_attribute_base_t FAR * entry,uint16_t search_id )
```

DESCRIPTION

Search the attribute table starting at entry, for attribute ID search_id.

PARAMETERS

[in] entry	Starting entry for search
[in] search_id	Attribute ID to look for

RETURN VALUE

NULL	Attribute id `search_id` not in list.
!NULL	Pointer to attribute record.

zcl_general_command

```
int zcl_general_command ( const wpan_envelope_t FAR * envelope,
   void FAR * context )
```

DESCRIPTION

Handler for ZCL General Commands.

Used as the handler for attribute-only clusters (e.g., Basic Cluster), or called from a cluster's handler for commands it doesn't handle.

Will send a Default Response for commands it can't handle.

Currently does not support attribute reporting.

PARAMETERS

[in] envelope	Envelope from received message
[in] context	Pointer to attribute tree for cluster (typically passed in via endpoint dispatcher)

RETURN VALUE

0	Command was processed, including sending a possible response.
!0	Error sending response or processing command.

zcl_invalid_cluster

```
int zcl_invalid_cluster ( const wpan_envelope_t FAR * envelope,
   wpan_ep_state_t FAR * ep_state )
```

DESCRIPTION

Called if a request comes in for an invalid endpoint/cluster combination.

Sends a Default Response (ZCL_CMD_DEFAULT_RESP) of ZCL_STATUS_FAILURE unless the received command was a Default Response or was broadcast.

PARAMETERS

[in] envelope	Envelope from received message
[in] ep_state	Pointer to endpoint's wpan_ep_state_t

RETURN VALUE

0	command was a default response (and therefore ignored), or a default response was successfully sent.
!0	error sending response.

zcl_invalid_command

```
int zcl_invalid_command ( const wpan_envelope_t FAR * envelope )
```

DESCRIPTION

Called if a request comes in for an invalid command on a valid endpoint/cluster combination.

Sends a Default Response (ZCL_CMD_DEFAULT_RESP) unless the received command was a Default Response or was broadcast. Depending on the command received, the response will be one of:

- ZCL_STATUS_UNSUP_CLUSTER_COMMAND
- ZCL_STATUS_UNSUP_GENERAL_COMMAND
- ZCL_STATUS_UNSUP_MANUF_CLUSTER_COMMAND
- ZCL_STATUS_UNSUP_MANUF_GENERAL_COMMAND

PARAMETERS

[in] envelope	Envelope from received message

RETURN VALUE

0	Default response was successfully sent, or command did not require a default response.
!0	Error sending response.

zcl_parse_attribute_record

```
int zcl_parse_attribute_record ( const zcl_attribute_base_t FAR *
   entry, zcl_attribute_write_rec_t * write_rec )
```

DESCRIPTION

Parse an attribute record from a Write Attributes Request or a Read Attributes Response and possibly store the new attribute value.

If ZCL_ATTR_WRITE_FLAG_READ_RESP is set in `write_rec->flags`,

`write_rec->buffer` will be parsed as a Read Attributes Response record. Otherwise, it's parsed as a Write Attributes Request record.

PARAMETERS

[in] entry	Attribute table to search
[in,out] write_rec	State information for parsing write request

RETURN VALUE

>=0	Number of bytes consumed from buffer.

zcl_send_response

```
int zcl_send_response ( zcl_command_t * request, const void FAR *
   response,uint16_t length )
```

DESCRIPTION

Send a response to a ZCL command.

Uses information from the command received to populate fields in the response.

PARAMETERS

[in] request	Command to respond to
[in] response	Payload to send as response
[in] length	Number of bytes in response

RETURN VALUE

0	Successfully sent response.
!0	Error on send.

zcl_status_text

```
const char * zcl_status_text ( uint_fast8_t status )
```

DESCRIPTION

Converts a ZCL status byte (one of the ZCL_STATUS_* macros) into a string.

PARAMETERS

[in] status	Status byte to convert

RETURN VALUE

pointer to an unmodifiable string with a description of the status

5.6.3.3 Basic Cluster API Functions and Macros

This section contains descriptions for the Basic Cluster library functions and macros. To use these functions the program must have **#include "zigbee/zcl_basic.h"** within the code.

zcl_basic_server

```
int zcl_basic_server ( const wpan_envelope_t FAR * envelope, void FAR
   * context )
```

DESCRIPTION

Handles commands for the Basic Server Cluster.

Currently supports the only command ID in the spec, 0x00 - Reset to Factory Defaults.

Note: You must define the macro ZCL_FACTORY_RESET_FN in your program, and have it point to a function to be called when a factory reset command is sent.

PARAMETERS

[in] envelope	Envelope from received message
[in] context	Pointer to attribute tree for cluster

RETURN VALUE

0	Command was processed, including sending a possible response
!0	Error sending response or processing command

5.6.3.4 Commissioning Cluster API Functions and Macros

This section contains API descriptions for the Commissioning Cluster library functions and macros. To use these functions the program must have **#include "zigbee/zcl_commissioning.h"** within the code.

zcl_comm_reset_parameters

```
int zcl_comm_reset_parameters ( wpan_envelope_t FAR * envelope,
   const zcl_comm_reset_startup_param_t * parameters )
```

DESCRIPTION

Send a "Reset Startup Parameters" command to the ZCL Commissioning Cluster.

App needs to set dev, `ieee_address`, `network_address`, `profile_id`, `source_endpoint`, `dest_endpoint` and (optionally) `options` in the envelope. This function will set the cluster ID and erase the payload and length of the envelope

PARAMETERS

[in,out] envelope	Partial envelope used to send the request. Caller must set `dev`, `ieee_address`, `network_address`, `profile_id`, `source_endpoint`, `dest_endpoint` and (optionally) `options`. On return, `payload` and `length` are cleared, and `cluster_id` is set to ZCL_CLUST_COMMISSIONING. If `source_endpoint` is 0, function will search the endpoint table for a Commissioning Client Cluster and set the envelope's `source_endpoint` and `profile_id`.
[in] parameters	Parameters for the Reset Startup Parameters command or NULL for default settings (reset current parameters only).

RETURN VALUE

0	Request sent.
-EINVAL	Couldn't find source endpoint in endpoint table, or some other error in parameter passed to function.
!0	Error trying to send request.

zcl_comm_restart_device

```
int zcl_comm_restart_device ( wpan_envelope_t FAR * envelope,
   const zcl_comm_restart_device_cmd_t * parameters )
```

DESCRIPTION

Send a "Restart Device" command to the ZCL Commissioning Cluster.

App needs to set `dev`, `ieee_address`, `network_address`, `profile_id`, `source_endpoint`, `dest_endpoint` and (optionally) `options` in the envelope. This function will set the cluster ID and erase the payload and length of the envelope

PARAMETERS

[in,out] envelope	Partial envelope used to send the request. Caller must set `dev`, `ieee_address`, `network_address`, `profile_id`, `source_endpoint`, `dest_endpoint` and (optionally) `options`. On return, `payload` and `length` are cleared, and `cluster_id` is set to ZCL_CLUST_COMMISSIONING. If `source_endpoint` is 0, function will search the endpoint table for a Commissioning Client Cluster and set the envelope's `source_endpoint` and `profile_id`.
[in] parameters	Parameters for the Restart Device command or NULL for default settings (save changes and restart without delay/jitter).

RETURN VALUE

0	Request sent.
-EINVAL	Couldn't find source endpoint in endpoint table, or some other error in parameter passed to function.
!0	Error trying to send request.

5.6.3.5 Identify Cluster API Functions and Macros

This section contains descriptions for the Identify Cluster library functions and macros. To use these functions the program must have **#include "zigbee/zcl_identify.h"** within the code.

zcl_identify_command

```
int zcl_identify_command( const wpan_envelope_t FAR * envelope,
   void FAR * context )
```

DESCRIPTION

Handler for ZCL Identify Server Commands (Identify, IdentifyQuery).

Used in the ZCL_CLUST_IDENTIFY cluster entry for an endpoint.

PARAMETERS

[in] envelope	Envelope from received message
[in] context	Pointer to attribute list for cluster (typically passed in via endpoint dispatcher)

RETURN VALUE

0 Command was processed, including sending a possible Identify Query Response.

!0 Error sending response or processing command.

zcl_identify_isactive

```
uint16_t zcl_identify_isactive( void )
```

DESCRIPTION

Used by main program to see if a device is in "Identification mode".

RETURN VALUE

>0 Number of seconds of "Identification mode" left.

0 Device is not in "Identification mode".

5.6.3.6 Time Cluster API Functions and Macros

This section contains API descriptions for the Cluster Time library functions and macros. To use these functions the program must have **#include "zigbee/zcl_time.h"** within the code.

zcl_gmtime

```
struct tm * zcl_gmtime(struct tm * dest_tm, zcl_utctime_t timestamp)
```

DESCRIPTION

Converts a ZCL UTCTime value into a "broken-down time" (a `struct tm`) for directly accessing month, day, year, hour, minute and second, or for use with other functions from <time.h>.

PARAMETERS

[in] timestamp	Timestamp to convert. Number of seconds since 1/1/2000 UTC.
[out] dest_tm	Destination `struct tm` structure to hold the broken-down time.

See time.h for details on that structure.

RETURN VALUE

Returns pointer to the `dest_tm` parameter.

zcl_mktime

```
zcl_utctime_t zcl_mktime( struct tm * time_rec )
```

DESCRIPTION

Convert a `struct tm` (from the Standard C Library's time.h) to a `zcl_utctime_t` type (number of seconds since Midnight January 1, 2000 UTC).

Does NOT properly handle DST and Timezones in the struct tm. Assumes the struct is in UTC. Keep this in mind if passing a `struct tm` generated by the host's gmtime() to this function.

PARAMETERS

[in] time_rec	Broken-down (componentized) calendar representation of time.

RETURN VALUE

0	Number of seconds since 01/01/2000 00:00:00 UTC or ZCL_UTCTIME_INVALID if `time_rec` is before 1/1/2000.

zcl_time_client

```
int zcl_time_client( const wpan_envelope_t FAR * envelope, void FAR
   * context )
```

DESCRIPTION

Handle Read Attribute Responses to requests sent as part of the zcl_time_find_servers() process.

This function expects to receive "read attribute response" packets ONLY for reads of Time and TimeStatus.

If responding device is a master or is synchronized with one, use it's Time value to update a "skew" global used to track the offset between system time (which may just be "seconds of uptime") and calendar time.

PARAMETERS

[in] envelope	Envelope from received message
[in] context	Pointer to attribute list for cluster (typically passed in via endpoint dispatcher)

RETURN VALUE

0	Command was processed and default response (with either success or error status) was sent.
!0	Error sending default response; time may or may not have been set.

zcl_time_find_servers

```
int zcl_time_find_servers( wpan_dev_t * dev, uint16_t profile_id )
```

DESCRIPTION

Find Time Servers on the network, query them for the current time and then synchronize this device's clock to that time.

Note: This function uses a static buffer to hold context information for the ZDO responder that generates the ZCL Read Attributes request. Wait at least 60 seconds between calls to allow for earlier requests to time out.

This function will only work correctly if the Time Cluster Client in your endpoint table is using the zcl_time_client() function as its callback handler. If you use the ZCL_CLUST_ENTRY_TIME_CLIENT or ZCL_CLUST_ENTRY_TIME_BOTH macro, the client cluster is set up correctly.

PARAMETERS

[in] dev	Device to send query on
[in] profile_id	Profile Id to match in endpoint table or #WPAN_APS_PROFILE_ANY to use the first endpoint with a Time Cluster Client

RETURN VALUE

0	Successfully issued ZDO Match Descriptor Request to find Time Cluster Servers on the network. No guarantee that we'll get a response.
!0	Some sort of error occurred in generating or sending the ZDO Match Descriptor Request.
-EINVAL	Couldn't find a Time Cluster Client with profile_id in the endpoint table of dev.

zcl_time_now

```
zcl_utctime_t zcl_time_now( void )
```

DESCRIPTION

Returns the current date/time, using the ZCL epoch of January 1, 2000.

Assumes that device has connected to a time server and updated its clock accordingly. Returns ZCL_UTCTIME_INVALID if the device has not synchronized its clock.

Do not use this value for tracking elapsed time -- use xbee_seconds_timer() or xbee_millisecond_timer() instead. The value may jump forward (or even backward) when the device decides to synchronize with a time server.

RETURN VALUE

ZCL_UTCTIME_INVALID	Clock not synchronized to a time source.
0-0xFFFFFFFE	Number of elapsed seconds since midnight UTC on January 1, 2000.

SEE ALSO

xbee_seconds_timer, xbee_millisecond_timer

5.6.3.7.1 64-bit Integer Support API Functions and Macros

This section contains descriptions for the 64-bit integer support functions and macros. To use these functions the program must have **#include "zigbee/zcl64.h"** within the code.

5.6.3.7.2 Cluster Support Datatypes Functions and Macros

This section contains descriptions for the Cluster Time library functions and macros. To use these functions the program must have **#include "zigbee/zcl_types.h"** within the code.

zcl_sizeof_type

```
int zcl_sizeof_type ( uint8_t type )
```

DESCRIPTION

Return the number of octets used by a given ZCL datatype.

PARAMETERS

[in] type	Type to look up. Typically one of the ZCL_TYPE_* macros, or the type element of an attribute record.

RETURN VALUE

ZCL_T_SIZE_INVALID	Unknown or invalid type.
-1	1-octet size prefix.
-2	2-octet size prefix.
0 to 8, 16	Number of octets used by type.

SEE ALSO

zigbee/zcl_types.h

ZCL_TYPE_IS_ANALOG

```
bool_t ZCL_TYPE_IS_ANALOG ( uint8_t type )
```

DESCRIPTION

Macro which returns a logical true/false if the type given is analog (signed, unsigned, float, etc.). Basically a numerical value that can be added or subtracted, not a discrete value.

RETURN VALUE

Logical true / false value indicating if type is analog.

ZCL_TYPE_IS_DISCRETE

```
bool_t ZCL_TYPE_IS_DISCRETE ( uint8_t type )
```

DESCRIPTION

Macro which returns a logical true/false if the type given is discrete (bitmap, enum, etc.). Basically a value that can be added or subtracted.

RETURN VALUE

Logical true / false value indicating if `type` is discrete.

ZCL_TYPE_IS_REPORTABLE

```
bool_t ZCL_TYPE_IS_REPORTABLE ( uint8_t type )
```

DESCRIPTION

Macro which returns a logical true/false if the `type` is marked as reportable.

RETURN VALUE

Logical true / false value indicating if `type` is reportable.

ZCL_TYPE_IS_SIGNED

```
bool_t ZCL_TYPE_IS_SIGNED ( uint8_t type )
```

DESCRIPTION

Macro which returns a logical true/false if the type given is signed or has the ability to be positive or negative.

RETURN VALUE

Logical true / false value indicating if `type` is signed.

5.6.3.7.3 ZCL Client Support API Functions and Macros

This section contains descriptions for the ZCL Client support library functions and macros. To use these functions the program must have **#include "zigbee/zcl_client.h"** within the code.

zcl_find_and_read_attributes

```
int zcl_find_and_read_attributes ( wpan_dev_t * dev, const uint16_t
   * clusters, const zcl_client_read_t * cr )
```

DESCRIPTION

Use ZDO Match Descriptor Requests to find devices with a given profile/cluster and then automatically send a ZCL Read Attributes request for some of that cluster's attributes.

PARAMETERS

[in] dev	Device to use for time request
[in] clusters	Pointer to list of server clusters to search for, must end with: WPAN_CLUSTER_END_OF_LIST
[in] cr	zcl_client_read record containing information on the request (endpoint, attributes, etc.); must be a static object (not an auto variable) since the ZDO responder will need to access it

RETURN VALUE

0	Request sent.
!0	Error sending request.

zdo_send_match_desc

```
int zdo_send_match_desc ( wpan_dev_t * dev, const uint16_t *
   clusters,uint16_t profile_id,wpan_response_fn callback, const
   void FAR * context )
```

DESCRIPTION

Send a ZDO Match Descriptor request for a list of cluster IDs. Commonly used with _zcl_send_read_from_zdo_match as the callback and a zcl_client_read_t structure as the context to automatically generate a ZCL Read Attributes Request in response to the ZDO Match Descriptor response.

PARAMETERS

[in] dev	Device to use for time request
[in] clusters	Pointer to list of server clusters to search for, must end with WPAN_CLUSTER_END_OF_LIST
[in] profile_id	Profile ID associated with the cluster IDs (cannot be WPAN_APS_PROFILE_ANY)
[in] callback	Function that will process the ZDO Match Descriptor responses; see wpan_response_fn for the callback's API
[in] context	Context to pass to callback in the wpan_conversation_t structure

RETURN VALUE

0	Request sent.
!0	Error sending request.

SEE ALSO

zcl_find_and_read_attributes()

5.7 Simple XBee API

This section contains information about the Simple XBee API. It is a library designed to simplify the setup and operation of a ZigBee network. This API works on ZigBee networks comprised of Digi XBee modules. This API does not work with other manufacturers ZigBee devices as it takes advantage of some XBee functionality not found on other devices.

5.7.1 General Overview

The Simple XBee API requires just the initialization of the XBee Driver layer before calling the SXA initialization function to bring up the ZigBee network, including node discovery. The API includes simplified commands

for configuring inputs and outputs, reading analog and digital input values and output states, and writing new output states. All samples that use the Simple XBee API start with the letters ‘sxa’.

5.7.2 API Functions and Macros

This section contains descriptions for the Simple XBee API functions and macros separated into layer components.

5.7.2.1 XBee I/O API Functions and Macros

This section contains API descriptions for the XBee I/O library functions and macros. To use these functions the program must have **#include “xbee/io.h”** within the code.

xbee_io_configure

```
int xbee_io_configure ( xbee_dev_t * xbee, xbee_io_t FAR * io,
   uint_fast8_t index, enum xbee_io_type type, const wpan_address_t
   FAR * address )
```

DESCRIPTION

Configure XBee digital and analog I/Os.

This modifies the shadow state in the `io` parameter, as well as sending the appropriate configuration command to the target device. If the new state is the same as the shadow state, then the function returns without doing anything, unless XBEE_IO_FORCE is specified in the `type` parameter.

PARAMETERS

[in, out] xbee	Local device through which to action the request
[in, out] io	Pointer to an xbee_io_t structure as set up by xbee_io_query(). Shadow state of I/O may be modified.
[in] index	Digital or analog I/O number e.g. 0 for DIO0 or AD0.
[in] type	Type of I/O to configure. If the XBEE_IO_FORCE flag is ORed in, force a configuration update to the device. Otherwise, a configuration change will only be sent to the device if the shadow (i.e. last known) configuration is different.
[in] address	NULL for local XBee, or destination IEEE (64-bit) or network (16 bit) address of a remote device.

RETURN VALUE

>=0	The number of bytes queued in the XBee serial port's serial transmit buffer.

-EINVAL	Serial is not a valid XBee serial port.
0	Success.
-EINVAL	Specified I/O index does not exist, or bad parameter.
-EPERM	Specified I/O cannot be configured as requested because the hardware or firmware does not support it.
<0	Other negative value indicates problem transmitting the configuraion request.

SEE ALSO

xbee_ser_tx_free(), xbee_ser_tx_used(), xbee_ser_tx_flush(), xbee_ser_rx_free()

xbee_io_get_analog_input

```
int xbee_io_get_analog_input ( const xbee_io_t FAR * io,
   uint_fast8_t index )
```

DESCRIPTION

Return reading of an analog input.

PARAMETERS

[in] io	Pointer to an xbee_io_t structure as set up by xbee_io_response_parse()
[in] index	Analog input number e.g. 0 for AD0, 2 for AD2, 7 for Vcc reading (where available)

RETURN VALUE

0..32767	Raw single-ended analog reading, normalized to the range 0..32767. Current hardware supports 10-bit ADCs, hence the 5 LSBs will be zero. Future hardware with higher resolution ADCs will add precision to the LSBs.
-16384..16383	Raw differential analog reading, normalized to the range -16384..16383. This is reserved for future hardware.
XBEE_IO_ANALOG_INVALID	Output unknown because not configured as an analog input, or is a non-existent input index.

xbee_io_get_digital_input

```
int xbee_io_get_digital_input ( const xbee_io_t FAR * io,
   uint_fast8_t index )
```

DESCRIPTION

Return state of a digital input.

PARAMETERS

[in] io	Pointer to an xbee_io_t structure as set up by xbee_io_response_parse()
[in] index	Digital input number e.g. 0 for DIO0, 12 for DIO12.

RETURN VALUE

0	Input low
1	Input high
-EINVAL	Input unknown because not configured as a digital input, or is a non-existent input index.

xbee_io_get_digital_output

```
int xbee_io_get_digital_output ( const xbee_io_t FAR * io,
   uint_fast8_t index )
```

DESCRIPTION

Return state of a digital output. This is a shadow state i.e. the last known state setting.

PARAMETERS

[in] io	Pointer to an xbee_io_t structure as set up by xbee_io_query()
[in] index	Digital output number e.g. 0 for DIO0, 2 for DIO2

RETURN VALUE

0	Output low
1	Output high
-EINVAL	Output unknown because not configured as a digital output, or is a non-existent

output index

xbee_io_get_query_status

```
int xbee_io_get_query_status (xbee_io_t FAR * io )
```

DESCRIPTION

Check the status of querying an XBee device, as initiated by xbee_io_query().

PARAMETERS

[in] io	I/O query to check (pointer as passed to xbee_io_query()).

RETURN VALUE

0	Query completed
-EINVAL	io is NULL
-EBUSY	Query underway
-ETIMEDOUT	Query timed out
-EIO	Halted, but query may not have completed (unexpected response)

xbee_io_query

```
int xbee_io_query ( xbee_dev_t * xbee, xbee_io_t FAR * io, const
   wpan_address_t FAR * address )
```

DESCRIPTION

Read current configuration of XBee digital and analog I/Os.

This sets the shadow state in the io parameter by querying the device configuration with a sequence of AT commands (D0,D0,...P0,...,PR). Unless the application has prior knowledge of the I/O configuration, this function should be used when a new node is discovered.

Since several commands must be executed, the results are not available immediately on return from this function. Instead, the application must call xbee_io_query_status() in oprder to poll for command completion.

PARAMETERS

[in, out] xbee	Local device through which to action the request

[in, out] io	Pointer to an xbee_io_t structure, which will be modified by this call. Since the results are returned asynchronously, the data pointed to must be static
[in] address	NULL for local XBee, or destination IEEE (64-bit) or network (16 bit) address of a remote device

RETURN VALUE

0	Success
-EINVAL	Bad parameter.
-EBUSY	Device is currently busy with another request for this device. Try again later (after calling xbee_cmd_tick()). In general, several get configuration requests can run simultaneously, however only one per remote device.
<0	Other negative value indicates problem transmitting the configuration query commands.

SEE ALSO

xbee_io_query_status().

xbee_io_response_dump

```
void xbee_io_get_response_parse (xbee_io_t FAR * parsed, const void
   FAR * source )
```

DESCRIPTION

Debugging function used to dump an xbee_io_t structure to stdout.

PARAMETERS

[in] io	Pointer to an xbee_io_t structure as set up by xbee_io_response_parse()

xbee_io_response_parse

```
int xbee_io_response_parse (xbee_io_t FAR * parsed, const void FAR *
   source )
```

DESCRIPTION

Parse an I/O response and store it in an xbee_io_t structure.

PARAMETERS

[out] parsed	Result structure #param
[in] source	Pointer to the start of the data i.e. the num_samples field of the I/O response

RETURN VALUE

0	Response parse completed
-EINVAL	Invalid parameter (NULL for either pointer or invalid source)

xbee_io_set_digital_output

```
int xbee_io_set_digital_output ( xbee_dev_t * xbee, xbee_io_t FAR *
   io, uint_fast8_t index, enum xbee_io_digital_output_state state,
   const wpan_address_t FAR * address )
```

DESCRIPTION

Set state of a digital output.

This modifies the shadow state in the io parameter, as well as sending the appropriate configuration command to the target device. If the new state is the same as the shadow state, then the function returns without doing anything, unless XBEE_IO_FORCE is specified in the state parameter.

PARAMETERS

[in, out] xbee	Local device through which to action the request
[in, out] io	Pointer to an xbee_io_t structure as set up by xbee_io_query(). Shadow state of I/O may be modified.
[in] index	Digital output number e.g. 0 for DIO0, 2 for DIO2
[in] state	New state of I/O. If the XBEE_IO_FORCE flag is ORed in, then force a state update to the device, whether or not configured as an input or with shadow state unchanged. This can be used to initially configure and set a digital output. Otherwise, a state change will only be sent to the device if the shadow (i.e. last known) state is opposite AND the I/O is configured as a digital output.

[in] address	NULL for local XBee, or destination IEEE (64-bit) or network (16 bit) address of a remote device. See wpan_envelope_create() for use of IEEE and network addressing.

RETURN VALUE

0	Output low
1	Output high
-EINVAL	Output unknown because not configured as a digital output (and force was not specified), or is a non-existent output index, or bad parameter passed.
<0	Other negative value indicates problem transmitting the configuration request.

xbee_io_set_options

```
int xbee_io_set_options ( xbee_dev_t * xbee, xbee_io_t FAR * io,
   uint_16_t sample_rate, uint_16_t change_mask, const
   wpan_address_t FAR * address )
```

DESCRIPTION

Configure XBee automatic I/O sampling options.

This basically controls the ATIR and ATIC settings. IR specifies an automatic sampling interval, and IC specifies sampling on digital I/O change.

PARAMETERS

[in, out] xbee	Local device through which to action the request
[in, out] io	Pointer to an xbee_io_t structure as set up by xbee_io_query(). Shadow state of I/O may be modified
[in] sample_rate	Rate sample period in ms. 0 to turn off sampling, otherwise must be a value greater than XBEE_IO_MIN_SAMPLE_PERIOD (50 ms with current hardware).
[in] change_mask	Bitmask of I/Os which are to generate samples when their state changes. Construct from ORed combination of XBEE_IO_DIO0, XBEE_IO_DIO1 etc.

[in] address	NULL for local XBee, or destination IEEE (64-bit) or network (16 bit) address of a remote device.

RETURN VALUE

0	Success
-EINVAL	Bad parameter.
-EPERM	Specified sampling rate or I/O bitmask not supported.
<0	Other negative value indicates problem transmitting the configuration request.

5.7.2.2 Simple XBee API Functions and Macros

This section contains API descriptions for the Simple XBee library functions and macros. To use these functions the program must have **#include "xbee/sxa.h"** within the code.

sxa_get_analog_input

```
int sxa_get_analog_input ( const sxa_node_t FAR * sxa,uint_fast8_t
   index )
```

DESCRIPTION

Return reading of an analog input.

PARAMETERS

[in] sxa	Pointer to a sxa_node_t structure
[in] index	Analog input number e.g. 0 for AD0, 2 for AD2, 7 for Vcc reading (where available)

RETURN VALUE

0..32767	Raw single-ended analog reading, normalized to the range 0..32767. Current hardware supports 10-bit ADCs, hence the 5 LSBs will be zero. Future hardware with higher resolution ADCs will add precision to the LSBs.
-16384..16383	Raw differential analog reading, normalized to the range -16384..16383. This is reserved for future hardware.
XBEE_IO_ANALOG_INVALID	Output unknown because not configured as an analog in put, or is a non-existent input index.

sxa_get_digital_input

```
int sxa_get_digital_input ( const sxa_node_t FAR * sxa,
   uint_fast8_t index )
```

DESCRIPTION

Return state of a digital input.

PARAMETERS

[in] sxa	Pointer to a sxa_node_t structure
[in] index	Digital input number e.g. 0 for DIO0, 12 for DIO12

RETURN VALUE

0	Input low.
1	Input high.
-EINVAL	Input unknown because not configured as a digital input, or is a non-existent input index.

sxa_get_digital_output

```
int sxa_get_digital_output ( const sxa_node_t FAR * sxa,
   uint_fast8_t index )
```

DESCRIPTION

Return state of a digital output. This is a shadow state i.e. the last known state setting.

PARAMETERS

[in] sxa	Pointer to a sxa_node_t structure
[in] index	Digital output number e.g. 0 for DIO0, 2 for DIO2

RETURN VALUE

0	Output low
1	Output high
-EINVAL	Output unknown because not configured as a digital output, or is a non-existent output index.

sxa_init_or_exit

```
int sxa_init_or_exit ( xbee_dev_t * xbee, const
   wpan_endpoint_table_entry_t * ep_table, int verbose)
```

DESCRIPTION

Initializes the XBee module for use with the Simple XBee API or fails and calls exit().

PARAMETERS

[in] xbee	Pointer to a XBee device structure
[in] index	Pointer to a WPAN endpoint table entry structure
[in] verbose	Verbose stdio output flag (non-zero value selects verbose)

RETURN VALUE

0	Input low
1	Input high
-EINVAL	Input unknown because not configured as a digital input, or is a non-existent input index.

sxa_io_configure

```
int sxa_io_configure ( sxa_node_t * sxa, uint_fast8_t index, enum
   xbee_io_type type )
```

DESCRIPTION

Configure XBee digital and analog I/Os.

This modifies the shadow state in the io parameter, as well as sending the appropriate configuration command to the target device. If the new state is the same as the shadow state, then the function returns without doing anything, unless XBEE_IO_FORCE is specified in the type parameter.

PARAMETERS

[in] sxa	Pointer to a sxa_node_t structure
[in] index	Digital or analog I/O number e.g. 0 for DIO0 or AD0

[in] type	Type of I/O to configure. If the XBEE_IO_FORCE flag is ORed in, force a configuration update to the device. Otherwise, a configuration change will only be sent to the device if the shadow (i.e. last known) configuration is different.

RETURN VALUE

>=0	The number of bytes queued in the XBee serial port's serial transmit buffer.
-EINVAL	Serial is not a valid XBee serial port.
0	Success
-EINVAL	Specified I/O index does not exist, or bad parameter.
-EPERM	Specified I/O cannot be configured as requested because the hardware or firmware does not support it.
<0	Other negative value indicates problem transmitting the configuration request.

SEE ALSO

xbee_ser_tx_free(), xbee_ser_tx_used(), xbee_ser_tx_flush(), xbee_ser_rx_free()

sxa_io_dump

```
int sxa_io_dump ( sxa_node_t FAR * sxa )
```

DESCRIPTION

Debug function to dump the sxa.io.config[] parameters to stdio.

PARAMETERS

[in] sxa	Pointer to an sxa_node_t structure

sxa_io_set_options

```
int sxa_io_set_options ( sxa_node_t * sxa, uint_16_t sample_rate,
   uint_16_t change_mask )
```

DESCRIPTION

Configure XBee automatic I/O sampling options.

This basically controls the ATIR and ATIC settings. IR specifies an automatic sampling interval,

and IC specifies sampling on digital I/O change.

PARAMETERS

[in] sxa	Pointer to a sxa_node_t structure
[in] sample_rate	Rate sample period in ms. 0 to turn off sampling, otherwise must be a value greater than XBEE_IO_MIN_SAMPLE_PERIOD (50 ms with current hardware)
[in] change_mask	Bitmask of I/Os which are to generate samples when their state changes. Construct from ORed combination of XBEE_IO_DIO0, XBEE_IO_DIO1 etc

RETURN VALUE

0	Success
-EINVAL	Bad parameter.
-EPERM	Specified sampling rate or I/O bitmask not supported.
<0	Other negative value indicates problem transmitting the configuration request.

sxa_io_query

```
int sxa_io_query ( sxa_node_t * sxa )
```

DESCRIPTION

Read current configuration of XBee digital and analog I/Os.

This sets the shadow state in the sxa parameter by querying the device configuration with a sequence of AT commands (D0,D0,...P0,...,PR). Unless the application has prior knowledge of the I/O configuration, this function should be used when a new node is discovered.

Since several commands must be executed, the results are not available immediately on return from this function. Instead, the application must call sxa_io_query_status() in order to poll for command completion.

PARAMETERS

[in] sxa	Pointer to a sxa_node_t structure

RETURN VALUE

0	Success

-EINVAL	Bad parameter.
-EBUSY	Device is currently busy with another request for this device. Try again later (after calling xbee_cmd_tick()). In general, several get configuration requests can run simultaneously, however only one per remote device.
<0	Other negative value indicates problem transmitting the configuration query commands.

SEE ALSO

sxa_io_query_status().

sxa_io_query_status

```
int sxa_io_query_status ( sxa_node_t FAR * sxa )
```

DESCRIPTION

Check the status of querying an XBee device, as initiated by xbee_io_query().

PARAMETERS

[in] sxa	Pointer to a sxa_node_t structure

RETURN VALUE

0	Query completed
-EINVAL	io is NULL
-EBUSY	Query underway
-ETIMEDOUT	Query timed out
-EIO	Halted, but query may not have completed (unexpected response)

sxa_set_digital_output

```
int sxa_set_digital_output ( sxa_node_t * sxa, uint_fast8_t index,
   enum xbee_io_digital_output_state state )
```

DESCRIPTION

Set state of a digital output.

This modifies the shadow state in the sxa parameter, as well as sending the appropriate configu-

ration command to the target device. If the new state is the same as the shadow state, then the function returns without doing anything, unless XBEE_IO_FORCE is specified in the state parameter.

PARAMETERS

[in] sxa	Pointer to a sxa_node_t structure
[in] index	Digital output number e.g. 0 for DIO0, 2 for DIO2
[in] state	New state of I/O. If the XBEE_IO_FORCE flag is ORed in, then force a state update to the device, whether or not configured as an input or with shadow state unchanged. This can be used to initially configure and set a digital output. Otherwise, a state change will only be sent to the device if the shadow (i.e. last known) state is opposite AND the I/O is configured as a digital output.

RETURN VALUE

0	Output low.
1	Output high.
-EINVAL	Output unknown because not configured as a digital output (and force was not specified), or is a non-existent output index, or bad parameter passed.
<0	Other negative value indicates problem transmitting the configuration request.

sxa_tick

```
void sxa_tick ( void )
```

DESCRIPTION

Tick function to be called periodically to drive the Simple XBee API library and all lower layers in the ZigBee stack and drivers. See section 5.1.4 for a discussion of how frequently this function needs to be called for a given configuration.

****

5.8 Protocol Firmware

ZigBee-capable Rabbit-based boards must be programmed with the appropriate RF module firmware. This firmware determines the role the module plays within the network: Coordinator, Router or End Device.

5.8.1 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\xbee_update_ebl.c`, relative to the Dynamic C installation

folder. The sample uses .ebl firmware files which cover several network protocols and node types. See the table below to find the file that should be ximported into the sample for the desired XBee functionality. The firmware files require the installation of X-CTU utility, as well as a web update of available firmware files. The files can then be found in the `\Digi\XCTU\update\ebl_files` directory. See section 5.8.2 for more information on the X-CTU utility.

Node Type	Firmware File Name
Coordinator	XB24-ZB_21xx.ebl
Router	XB24-ZB_23xx.ebl
End Device	XB24-ZB_29xx.ebl
On Firmware File Names, 'xx' is the version designator and the highest value should be used. It is preferrable that all nodes in the system be at the same version level, but not required.	

5.8.2 X-CTU: Updating RF Module Settings on a DIGI XBee USB Device

A utility program, X-CTU, is provided for reading and writing module settings and firmware on the Digi XBee devices. This utility is not able to update firmware on XBee devices that are connected to a Rabbit module or board as firmware updates require hardware handshaking and this is not available through the serial bypass utility on Rabbit/XBee modules or boards. It may still be used for viewing or changing the module settings on these designs. On Digi XBee USB dongles or XBee modules mounted to the XBIB board, the X-CTU program can update firmware provided the hardware handshake is enabled within the X-CTU program. The X-CTU utility is described in the "Users Guide: XCTU Configuration & Test Utility Software" document available at http://www.digi.com/support/ under the XCTU keyword.

5.9 Summary

This is the ground floor of a very useful low power wireless 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.

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.

ZDP

This is a specialized application object called the ZigBee Device Profile. It is addressed as endpoint 0.ZDP was referred to as ZigBee Device Object (ZDO) in the earlier releases.

Index