CHAPTER 5

ZigBee, ZDO, and ZDP

It's all well and good to know how to transmit data to another node through an APSDE-DATA.request, and what endpoints and groups are all about, but how does a node in a ZigBee network decide which other node(s) in the network to talk to? How is the network set up and maintained?

ZigBee contains two sets of services for network commissioning and maintenance:

- The ZigBee Device Object (together with the ZigBee Device Profile)
- The ZigBee Cluster Library

This chapter describes the ZigBee Device Object (ZDO) and the ZigBee Device Profile (ZDP). The next chapter (Chapter 6) describes the ZigBee Cluster Library (ZCL).

But first, before delving into ZDO, the *real* story behind the ZigBee name.

> *"Hey, Big Z! Come look at this!" Ford Prefect shouted, staring down at his computer console.*
>
> *Zaphod Beeblebrox swiveled one of his two heads toward Ford, saying, "Is it about me?"*
>
> *"Nah. More interesting than that. Take a look at the new Heart of Gold Mark II! Remember the last one with that annoying personality that was always asking you to say 'please' before it would open a door, or giving you extra tidbits of information you didn't ask for every time you queried the computer? Well, in the Mark II they got rid of it. They replaced it with some new wireless technology that automatically handles, well—everything! It opens doors automatically, it makes the lights follow you around the ship, and quiets the music down when you start talking, it says here, almost like it reads your mind."*
>
> *"Yeah, baby, but I'm of two minds, and I can't seem to get them to agree. For example, my second head is sleeping right now, you see." In fact, Zaphod's other head was snoring, loudly.*
>
> *"Well, Big Z, I'm going to steal it," said Ford, matter-of-factly.*
>
> *"What, my head?" asked Zaphod.*

"The HoG Mark II."

"Not cool," quipped Zaphod. "Already been done. I stole the first Heart of Gold, remember? Anyway, how would you do it?"

"Toss me another Pan Galactic Gargle Blaster, while you toss yours down. We're hitching a ride." Ford fingered his electronic thumb.

"You can't hitch a ride on the most expensive ship in the Galaxy with just an electronic thumb. It will never work. Impossible!"

"That's exactly why it is going to work," said Ford calmly. "It's just so amazingly improbable that it's nearly impossible. Probability drive. Remember?"

It's a well-known fact to anyone who has ever read The Hitchhiker's Guide to the Galaxy that the only way to handle hitching a ride on a passing space ship and still keep your mind was to be out-of-your-mind drunk when it happened. This fact was almost as well-known as the use of the electronic thumb, the interstellar equivalent of extending your thumb on the side of the road on 1960s Earth (a planet somewhere in the unpopular arm of a small spiral galaxy). It was significantly less well-known that the Heart Of Gold, and subsequently, the Heart Of Gold Mark II, achieved interstellar travel through the use of a probability drive, a drive which ignored very likely and very constant things such as the speed of light, and instead landed you, quite improbably, exactly where you didn't even know you wanted to go, and did it in almost no time at all.

After a few more Pan Galactic Gargle Blasters (Ford stopped counting after three), Ford Prefect said "Big Z, are you ready?" It actually sounded more like, "BigZeeuready," what with the slurring and all.

"And why are we stealing it, exactly?" asked Zaphod, talking mostly to the floor, which wasn't talking back. Ford answered instead.

"We're stealing it to get Trillian back."

"Ah. Trillian?" queried Zaphod.

"Yes, Big Z, Trillian. Remember her? You picked her up from Earth many years ago. She was with us on the last Heart of Gold."

"Ah. And what's the name of that there ol' thingy in the Heart of Gold Mark II that automatically handles, doors an' lights and well, everything?" asked Zaphod.

Ford, who was now also staring mostly at the floor, slurred "Hmm, Zig B? What?"

"Ah. ZigBee. Strange name for a technology, ZigBee."

At that moment, the electronic thumb started beeping and blinking madly. For some reason, engineers love to make gadgets beep and blink. In addition to beeping and blinking, the electronic thumb did what it was actually designed to do and winked them out of existence, to reappear right in the cargo hold somewhere inside the Heart of Gold Mark II.

The ZigBee Device Object (ZDO, shown in Figure 5.1) is simply the application running on endpoint 0 in every ZigBee device. (Remember, application endpoints are numbered 1 through 240.)

This application, ZDO, keeps track of the state of the ZigBee device on and off the network, and provides an interface to the ZigBee Device Profile (ZDP), a specialized Application Profile (with profile ID 0x0000) for discovering, configuring, and maintaining ZigBee devices and services on the network.

As you can see from the figure, ZDO not only interacts with APS, but also interacts directly with the network layer. ZDO controls the network layer, telling it when to form or join a network, and when to leave, and provides the application interface to network layer management services. For example, ZDO can be configured to continue attempting to join a network until it is successful, or until a user-specified number-of-retries has occurred before giving up, and informing the application of the join failure.

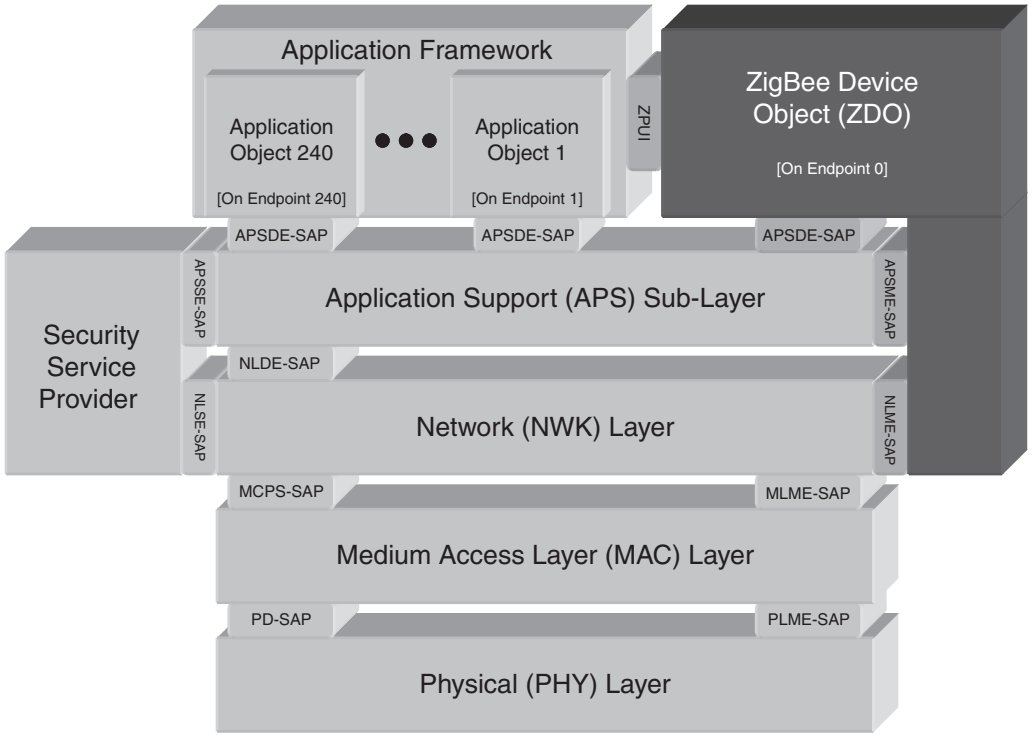

Figure 5.1: ZDO Is a ZigBee Application Object

The over-the-air Application Profile supported by ZDO, called the ZigBee Device Profile (ZDP), is no different than any other, and in most stacks is handled just like any other application object on an endpoint. ZDP services are separated into client and server. Client side services (also called requests), are always optional in ZigBee, but many of the server side ZDP services (also called responses), are mandatory.

Nearly every service follows the same pattern when used. A client device (the node which is doing the asking) first makes a request. The server device then sends the response back to the client device. The cluster number for the response is exactly the same as the cluster number for the request, but with the high bit set. For example, the ZDP command **IEEE_addr_req** is cluster 0x0001, and **IEEE_addr_rsp** is cluster 0x8001.

It doesn't matter how many hops the nodes are from each other. The nodes A and B could be 10 hops away from each other, and the ZDP request/response mechanism will work in exactly the same way, just as it does for applications sending data on an application endpoint (see Figure 5.2).

Many ZDP requests must be either explicitly unicast or broadcast. Others can unicast or broadcast at the client node's discretion (typically with different responses). If a ZDP request is broadcast, only the node that has the requested information returns any data. For example, **NWK_Addr_req** is broadcast, but only the node that matches the IEEE address, provided in the request, responds.

Every ZDP response starts with a status byte. If the particular optional service is not supported by the receiving node, the status returned will be **gZdoNotSupported_c** (0x84).

For sleeping devices, the parents of the device keep track of the IEEE and short address of the child, and will respond for them. However, all other information about the sleeping device, such as the list of active endpoints, are not recorded by the parent and must be retrieved directly from the devices themselves. In Chapter 8, "Commissioning ZigBee Networks," I'll discuss the means of commissioning sleeping devices.

In this chapter, I've organized the ZDP services slightly differently than in the ZigBee specification. For one thing, I've put the request and responses in the same section. The

Figure 5.2: ZDP Request and Response

ZigBee specification organizes the services numerically, so the request and responses are many pages apart. Also, I've organized the ZDP services by usage; so, for example, all the node-wide services are together.

ZDP services include the following categories:

- Device discovery services
- Service discovery services
- Binding services
- Management services

After discussing ZDP, I'll discuss how applications interact with ZDO, including:

- Starting and stopping the network through ZDO
- ZDO and low power nodes

5.1 Device Discovery

The ZigBee Device Profile (ZDP) contains a set of commands for discovering various aspects about nodes in the network. The ZigBee specification calls these "device discovery services," which can be confusing because endpoints contain device IDs which really describe individual ZigBee applications running in that node. So, when you see ZDP Device Discovery, think node-wide (not application/endpoint specific) services.

Device discovery services have a few things in common:

- They provide additional information about a node.
- They are all optional from the client side, but some server side processing is mandatory (a common subset among all ZigBee devices).
- They are node-wide, and do not represent any particular application, or Application Profile residing on an endpoint in the node.

The ZDP device discovery services are listed below in Table 5.1. Notice that all the ZDP services on the client side are optional. ZigBee does not require that a node be able to send **NWK_addr_req,** for example. But on the server side of this equation (a node receiving a **NWK_addr_req** and responding to it), the ZDP service is mandatory.

Table 5.1: ZigBee Device Profile Device Discovery Services

Device Discovery Services	Unicast (U), Broadcast (B) or Either (U,B)	Client Transmission (Request)	Server Processing (Response)
NWK_addr_req	U,B	O	M
IEEE_addr_req	U	O	M
Node_Desc_req	U	O	M
Power_Desc_req	U	O	M
Complex_Desc_req	U	O	O
User_Desc_req	U	O	O
User_Desc_set	U	O	O
Device_annce	B	O	M

This makes sense if you think about how the service is used. A tool may want to collect the IEEE (aka MAC) address of every node in the network (using **IEEE_addr_req**, for example) so all nodes in the network must support the server side (**IEEE_addr_rsp**). But only the tool needs to support the client side.

What happens if a given client issues two ZDP requests in a row? How does the client application know which response belongs to which request? Some stack vendors have solved this problem by only allowing a single request to be issued at any one time. Other stack vendors, such as Freescale, provide a *transaction ID* which correlates the request with the response. This rolling 8-bit transaction ID is sent with each request, meaning, in theory, that a single application could have up to 256 requests in flight at once. Normally, however, an application makes one or two requests, and then waits for the response.

In the Freescale ZigBee solution, all ZDP requests begin with the prefix **ASL_** (for example, **ASL_NWK_addr_req()**). Simply look up the particular ZDP request in the table, or the ZigBee specification, and prefix it with **ASL_**. Why ASL, and not ZDP? ASL stands for Application Support Library, which is the prefix used for all optional application-level commands in Freescale BeeStack.

The response to a ZDP request may take some time to come back, because, perhaps, the responding node may be many hops away. In a BeeStack application, this occurs through a C callback function registered with **Zdp_AppRegisterCallBack()**.

Each ZDP request in BeeStack requires a destination address, which may be unicast or broadcast, as the ZigBee specification allows.

```
void ASL_NWK_addr_req
(
  zbCounter_t *pSequenceNumber,
  zbNwkAddr_t aDestAddress,
  zbIeeeAddr_t aIeeeAddr,
  uint8_t requestType,
  index_t startIndex
);
```

One thing that is not always obvious with Freescale BeeStack (and this is true of other stack vendors as well) is that optional ZDP services are not enabled by default. In fact, they are compiled-out by default. Often ZigBee stacks run in systems that are very limited by RAM and Flash (ROM), which means every byte can be precious. Services that might not be used by the application are turned off to conserve space.

To enable the optional ZDP services, enable either the client-side service, server-side service, or both. For example, to enable both the server and client for **NWK_addr_req**, enable both **gNWK_addr_req_d** and **gNWK_addr_rsp_d** in BeeStack. All the ZDP services, even the mandatory ones, can be enabled or disabled through Freescale BeeKit, the graphical BeeStack configuration tool.

Although Freescale BeeKit allows it, I don't recommend disabling the mandatory ZDP services unless your company controls all the nodes in the ZigBee network, and you are willing to live with a (slightly) incompatible ZigBee stack. Certainly, the product cannot be certified by ZigBee if the mandatory ZDP services are disabled.

For some application profiles, such as Home Automation, some of the ZDP services listed as optional by the ZigBee specification are mandatory for certain devices in that profile. ZDP binding is a good example of this.

> Use ZDP to discover which nodes to talk to in a ZigBee network.
>
> Optional ZDP services may be mandatory in the application profile.
>
> Remember to enable the optional services if they are needed by a BeeStack application.

5.1.1 NWK_addr_req and IEEE_addr_req

Use ZDP network address request (**NWK_addr_req**) when you already know the MAC address of a node (also called its IEEE or long address), but want to find its short, 16-bit address on the network. This service request can be broadcast or unicast.

Table 5.2: NWK_addr_req/rsp

NWK_addr_req	NWK_addr_rsp
typedef struct zbNwkAddrRequest_tag { zbIeeeAddr_t aIeeeAddr; uint8_t requestType; zbIndex_t startIndex; } zbNwkAddrRequest_t;	typedef struct zbExtendedDevResp_tag { zbStatus_t iStatus; zbIeeeAddr_t aIeeeAddrRemoteDev; zbNwkAddr_t aNwkAddrRemoteDev; zbCounter_t numAssocDev; zbIndex_t startIndex; zbNwkAddr_t aNwkAddrAssocDevList[1]; } zbExtendedDevResp_t;

For example, say the gateway in a particular ZigBee network (which may or may not be on the ZigBee Coordinator) is known to be IEEE address **0x0050c237b0041234**. Issue a **NWK_addr_req** and the gateway will respond with its short address on the network.

Unfortunately, there is no ZigBee-standard way to find nodes within a range of IEEE addresses.

IEEE_addr_req is the converse of **NWK_addr_req** (see Tables 5.2 and 5.3). It returns the IEEE address of a node, given a 16-bit short address. This command is unicast to the destination. The responses are exactly the same for the two commands, and the requests are quite similar.

Table 5.3: IEEE_addr_req/rsp

IEEE_addr_req	IEEE_addr_rsp
typedef struct zbIeeeAddrRequest_tag { zbNwkAddr_t aNwkAddrOfInterest; uint8_t requestType; zbIndex_t startIndex; } zbIeeeAddrRequest_t;	typedef struct zbExtendedDevResp_tag { zbStatus_t iStatus; zbIeeeAddr_t aIeeeAddrRemoteDev; zbNwkAddr_t aNwkAddrRemoteDev; zbCounter_t numAssocDev; zbIndex_t startIndex; zbNwkAddr_t aNwkAddrAssocDevList[1]; } zbExtendedDevResp_t;

Notice that the first byte of the response is a status byte. This will be 0x00 (success) if the response is valid. If this contains an error code, then the rest of the information will not be included in the response. Every ZDP response begins with a status code, so be sure to check it in your applications before assuming that the rest of the information is valid.

Both **NWK_addr_req** and **IEEE_addr_req** contain a **requestType** field. The request type field affects whether the extended information is included in the response. Use **requestType** 0x00 to get only the IEEE and NWK address for one node. Use the extended **requestType** 0x01 to get the information for the node and for all its children as well. Remember, only routers will have children.

This particular request is generally broadcast across the network. If the request is broadcast, and the targeted NWK or IEEE address does not exist on the network, then no over-the-air response is issued. The client application should set up a time-out to let itself know that the node couldn't be found, perhaps to try again at another time.

Why unicast this command? It's a good way to see if a particular device is the child of a given parent. For example, say that you want to ensure that node XYZ is a child of the room controller in a hotel room. Issue a unicast to that room controller (a ZigBee Router) and it will respond either with an error code, or with the short address of the child.

A start index is normally used if the response can't fit in a single over-the-air packet (a payload of about 80 bytes). This field isn't actually needed in **NWK_addr_req** or **IEEE_addr_req** because the response will *always* fit, so always set it to 0.

The example in this section, *Example 5-1—ZDP NWK_addr_req*, uses **NWK_addr_req** to find the short address of a particular node, in this case the node with the IEEE address of 0x0050c237b0040002 (see Figure 5.3). The application on the ZC sends a broadcast across the network, and the node with proper IEEE address responds with its short address.

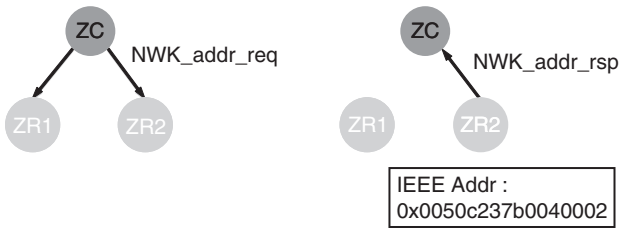

Figure 5.3: Example 5-1—ZDP NWK_addr_req

The BeeKit solution file for this example, found in the directory "Chapter05\Example 5-1-ZDP NWK_addr_req," contains three projects: one for an NCB ZigBee Coordinator (ZC)—the node making the request, and two for ZigBee Routers (ZR)—one of which is the node we're looking for.

To run the example, program the three boards (ZcNcb, Zr1Srb, Zr2Srb, respectively). Next, form the network with the ZcNcb (ZigBee Coordinator) board by pressing SW1. Join the other two nodes in any order, pressing SW1 on each of them. When the LEDs have finished chasing each other, the nodes are on the network. Then, press SW2 on the NCB board. The NCB will send out the **NWK_addr_req** and should display the short address of the node we're looking for. In the figure, this would be **0x143e**.

Try booting all the nodes, joining the routers in the opposite order (so that ZR2 boots first). Notice the NwkAddr returned is now **0x0001**.

> Use NWK_addr_req and IEEE_addr_req to find nodes based on short or long address.
>
> NWK_addr_rsp and IEEE_addr_rsp populate the address map.

5.1.2 ZigBee Descriptors

ZigBee uses *descriptors* to describe a node and its properties, allowing other applications running in the network to discover these properties over-the-air. Node-wide descriptors include the node descriptor, the power descriptor, the complex descriptor, and the user descriptor.

Of these descriptors, I find the **Node_Desc_req** the most useful (see Table 5.4). The results of this include the ZigBee node type (ZR, ZC, or ZED) and the manufacturer ID (a 16-bit ZigBee assigned number that uniquely identifies the manufacturer of the device).

The node descriptor contains a variety of fields, including the node type of the device (whether the node is a ZigBee Coordinator, Router, or End-Device), the manufacturer's code, whether the optional user and complex descriptors are present, and whether the node supports fragmentation.

Use this command when the application needs to know the manufacturer ID, whether the destination node can support the optional fragmentation, or if any other optional service

Table 5.4: Node Descriptor Request and Response

Node_Desc_req	Node_Desc_rsp

typedef struct
zbNodeDescriptorRequest_tag
{
  zbNwkAddr_t aNwkAddrOfInterest;
} zbNodeDescriptorRequest_t;
	typedef struct
	zbNodeDescriptorResponse_tag
	{
	  zbStatus_t status;
	  zbNwkAddr_t
	aNwkAddrOfInterest;
	  zbNodeDescriptor_t nodeDescriptor;
	} zbNodeDescriptorResponse_t;
	
	typedef struct zbNodeDescriptor_tag
	{
	  uint8_t logicalType;
	  uint8_t apsFlagsAndFreqBand;
	  uint8_t macCapFlags;
	  uint8_t aManfCodeFlags[2];
	  uint8_t maxBufferSize;
	  uint8_t aMaxTransferSize[2];
	  zbServerMask_t aServerMask;
	} zbNodeDescriptor_t;

is present. I rarely use this command in actual applications, except perhaps to find the manufacturer ID. That can be useful if a particular application wants to use extended commands only available from a particular manufacturer.

The other descriptors include the power descriptor, which defines which power modes this node supports, and the user descriptor, which contains a user definable string to identify the location (such as living room or office). These descriptors are all optional in the ZigBee spec. The user descriptor is settable over-the-air, the rest are only gettable.

Tables 5.5, 5.6, and 5.7 describe each of the other descriptors. By and large, these descriptors (with the exception of the Node descriptor) have been supplanted by the ZigBee Cluster Library (ZCL) Basic Cluster. If you are using a profile such as Home Automation (HA), or Automatic Metering (AMI), which use the ZigBee Cluster Library, use the Basic Cluster mechanism instead.

Table 5.5: Power Descriptor Request and Response

Power_Desc_req	Power_Desc_rsp
void ASL_Power_Desc_req (zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress);	typedef struct zbPowerDescriptor_tag { uint8_t currModeAndAvailSources; uint8_t currPowerSourceAndLevel; } zbPowerDescriptor_t;

Table 5.6: User Descriptor Request and Response

User_Desc_req	User_Desc_rsp
void ASL_User_Desc_req (zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress);	typedef struct zbUserDescriptorResponse_tag { zbStatus_t status; zbNwkAddr_t aNwkAddrOfInterest; uint8_t aUserDescriptor[16]; } zbUserDescriptorResponse_t;

Table 5.7: Complex Descriptor Request and Response

Complex_Desc_req	Complex_Desc_rsp
void ASL_User_Desc_req (zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress);	typedef struct zbComplexDescriptor_tag { uint8_t fieldCount; uint8_t aLanguageAndCharSet[4]; uint8_t aManufacturerName[6]; uint8_t aModelName[6]; uint8_t aSerialNumber[6]; uint8_t aDeviceUrl[17]; uint8_t aIcon[4]; uint8_t aIconUrl[9]; } zbComplexDescriptor_t;

Descriptors describe the node.

Use the ZigBee Cluster Library (ZCL) basic cluster rather than the power, complex and user descriptors.

Table 5.8: Device Announce Fields

Device_annce
typedef struct zbEndDeviceAnnounce_tag { zbNwkAddr_t aNwkAddress; zbIeeeAddr_t aIeeeAddress; macCapabilityInfo_t capability; } zbEndDeviceAnnounce_t;

5.1.3 Device Announce

The ZDP **Device_annce** command is issued by the ZigBee stack, not by the applications. Occasionally in a network, a device must change its short address while still on the network. In Stack profile 0x01, this occurs when an end-device loses track of its parent and needs to find a new one. In Stack profile 0x02, this occurs when an address conflict is detected.

Device_annce can also occur if an end-device wants to tell its parent to start buffer packets for it while it sleeps (called an RxOnIdle = FALSE device), or wants its parent to quit buffering packets because the device won't be sleeping anymore. (Perhaps it was plugged into mains power.)

All the **Device_annce** command accomplishes is to instruct any node in the network that cares about this node to update its internal tables, such as the neighbor table, address map, and binding table (see Table 5.8). The over-the-air device announce structure is fairly simple: a short address, IEEE address, and MAC capabilities flags.

The example in this section, *Example 5-2 Device_annce*, demonstrates device announce occurring when a child changes to a new parent (see Figure 5.4). A node is set up to look

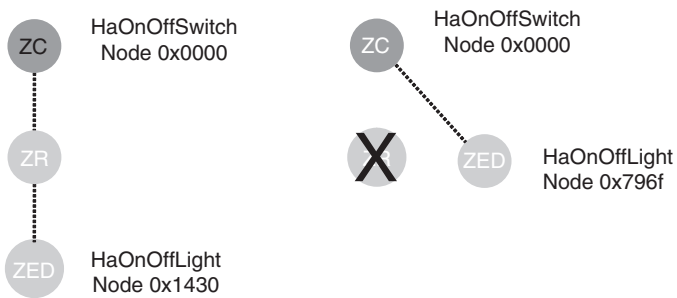

Figure 5.4: Example 5-2—Device_annce

for a new parent after it has lost contact with its original parent for three polling periods in a row. In the left portion of the figure, the ZED has a ZigBee router for a parent. But for some reason (Okay, because I turned it off), the ZED loses contact with its parent. The node then looks for a new parent, and finds the ZigBee Coordinator, shown on the right.

This same thing would happen if the ZED were, perhaps, a roaming remote control device. The same thing could occur if something happened to the link, such as if a large wall of metal or body of water was placed between the ZED and its parent.

To run the example shown above, use the BeeKit solution found in the folder "Chapter05\Example 5-2 Device_annce." This BeeKit solution contains three projects: a ZcNcbSwitch, a ZrSrbRangeExtender, and a ZedSrbLight. Export the solution, and import, compile, and download each project into their respective boards.

The steps to see the demo (and produce a capture) are:

1. Turn on Daintree to record on channel 25.
2. Boot and form a network with the ZcNcbSwitch and ZrSrbRangeExtender boards, by pressing SW1.
3. Turn off joining the ZcNcbSwitch by pressing SW2.
4. Join the network with ZedSrbLight by pressing SW1.
5. Bind the switch and light, by pressing SW3 (in any order) on both ZcNcbSwitch and ZedSrbLight.
6. Go to Application (as opposed to Configuration) Mode on both light and switch, by pressing and holding switch 1 (LSW1).
7. Toggle the light, by pressing SW1 on the ZcNcbSwitch.
8. Force the light to move to a new parent, by turning off ZcSrbRangeExtender.
9. Toggle the light again, by pressing SW1.

Notice the ZcNcbSwitch knows where to find the light (at 0x796f), even though it has moved (from 0x1430).

To see this action over-the-air, take a look at the following excerpts from the Daintree capture. First of all, notice that the switch (node 0x0000) is sending to the light (node 0x1430), which is a child of the range extender (node 0x0001):

```
87  +00:00:00.571  0x0000  0x0001  0x0000  0x1430  0x50  Zigbee APS Data
HA:On/off
88  +00:00:00.001                                      IEEE 802.15.4
Acknowledgment
89  +00:00:00.452  0x1430  0x0001                      IEEE 802.15.4
Command: Data Request
90  +00:00:00.001                                      IEEE 802.15.4
Acknowledgment
91  +00:00:00.003  0x0001  0x1430  0x0000  0x1430  0x75  Zigbee APS Data
HA:On/off
```

Now, the child has lost track of its parent. So, it issues a rejoin request to join a new parent. And then it announces via a broadcast its new short address to the network with **Device_annce**, called ZDP:EndDeviceAnnce:

```
138  +00:00:00.421  0x1430  0x0000  0x1430  0x0000  0x1a  Zigbee NWK
NWK Command: Rejoin Request
139  +00:00:00.001                                      IEEE 802.15.4
Acknowledgment
140  +00:00:00.409  0x1430  0x0000                      IEEE 802.15.4
Command: Data Request
141  +00:00:00.001                                      IEEE 802.15.4
Acknowledgment
142  +00:00:00.005  0x0000  0x1430  0x0000  0x1430  0x51  Zigbee NWK
NWK Command: Rejoin Response
143  +00:00:00.002                                      IEEE 802.15.4
Acknowledgment
144  +00:00:00.005  0x796f  0x0000  0x796f  0xffff  0x1b  Zigbee APS Data
ZDP:EndDeviceAnnce
145  +00:00:00.002                                      IEEE 802.15.4
Acknowledgment
146  +00:00:00.018  0x0000  0xffff  0x796f  0xffff  0x1b  Zigbee APS Data
ZDP:EndDeviceAnnce
```

Finally, notice that the ZcNcbSwitch still knows where to find the light. Instead of sending to address 0x1430, it sends to address 0x796f, the light's new short address:

```
164  +00:00:00.002  0x0000  0x796f  0x0000  0x796f  0x52  Zigbee APS Data
HA:On/off
```

One thing to be aware of: **Device_annce** is a broadcast, and every ZigBee network is limited by the number of broadcasts it can sustain at any given time. Don't design a network where children need to move constantly or the network may be overloaded.

5.2 Service Discovery

In addition to the services related to devices, or nodes, ZDP also contains a variety of standard services for querying the applications within those nodes (see Table 5.9). As with the device discovery services, most of the ZDP service discovery services are optional. Only a few service side responses are required.

5.2.1 Discovering and Matching Endpoints

Discovering application endpoints and the services they support is a common commissioning step in ZigBee. Different manufacturers may choose different endpoints for their applications. For example, a manufacturer of a switch (Leviton, perhaps) may choose endpoint 3 for their switch. Philips may choose endpoint 8 for their light. So how does an application which needs to bind this switch to the light find these endpoints?

Table 5.9: ZDP Service Discovery Services

Service Discovery Services	Client Transmission (Request)	Server Processing (Response)
Simple_Desc_req (unicast)	O	M
Extended_Simple_Desc_req (unicast)	O	O
Active_EP_req (unicast)	O	M
Extended_Active_EP_req (unicast)	O	O
Match_Desc_req (broadcast)	O	M
System_Server_Discover_req	O	O
Find_node_cache_req (broadcast)	O	O
Discovery_Cache_req (unicast)	O	O
Discovery_store_req (unicast)	O	O
Node_Desc_store_req (unicast)	O	O
Power_Desc_store_req (unicast)	O	O
Active_EP_store_req (unicast)	O	O
Simple_Desc_store_req (unicast)	O	O
Remove_node_cache_req (unicast)	O	O

Table 5.10: Active Endpoint Request and Response

Active_EP_req	Active_EP_rsp
void ASL_Active_EP_req (zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress);	typedef struct zbActiveEpResponse_tag { zbStatus_t status; zbNwkAddr_t aNwkAddrOfInterest; zbCounter_t activeEpCount; zbEndPoint_t pActiveEpList[1]; } zbActiveEpResponse_t;

ZDP can locate active endpoints through **Active_EP_req** (see Table 5.10). This call returns a list of the active endpoints in a node. The application then calls **Simple_Desc_req**, which returns a description of the endpoint (see Table 5.11). The simple descriptor really should have been called the endpoint descriptor, as that is the object it describes.

Table 5.11: Simple Descriptor Request and Response

Simple_Desc_req	Simple_Desc_rsp
void ASL_Simple_Desc_req (zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, zbEndPoint_t endPoint);	typedef struct zbSimpleDescriptor_tag { zbEndPoint_t endPoint; zbProfileId_t aAppProfId; zbDeviceId_t aAppDeviceId; uint8_t appDevVerAndFlag; zbCounter_t cNumInClusters; zbClusterId_t *pInClusterList; zbCounter_t cNumOutClusters; zbClusterId_t *pOutClusterList; } zbSimpleDescriptor_t; typedef struct zbSimpleDescriptorResponse_tag { zbStatus_t status; zbNwkAddr_t aNwkAddrOfInterest; zbSize_t length; zbSimpleDescriptor_t sSimpleDescriptor; } zbSimpleDescriptorResponse_t;

The simple descriptor basically describes everything there is to know about the endpoint: its Application Profile ID, and its list of endpoints, both input and output.

The **Extended_Simple_Desc_req** and **Extended_Active_EP_req** were added in ZigBee 2007 in case the simple descriptor or active endpoint list were too large to fit into a single packet. For example, assume that a node supports all 240 endpoints. Each active endpoint returned in an **Active_EP_req** requires one byte. That's at least 240 bytes, far too large to fit into the 127 byte 802.1.54 PHY. Likewise, if the cluster list is too long, an **Extended_Simple_Desc_req** might be needed. Normally, however, the standard versions are sufficient. It's a rare ZigBee network that contains nodes with that many endpoints or clusters on an endpoint.

Match_Desc_req can be used to find a particular service anywhere across the network (see Table 5.12). As input, it takes a simple descriptor, and as output it provides a matching list of endpoints from any node that matches. It matches both profile ID and input/output cluster lists. The profile ID must be the same, and at least one input must match one output cluster, or vice versa. Any overlap will do. Think of it this way.

Table 5.12: Match Descriptor Request

Match_Desc_req	Match_Desc_rsp
typedef struct **zbSimpleDescriptor_tag** **{** **zbEndPoint_t endPoint;** **zbProfileId_t aAppProfId;** **zbDeviceId_t aAppDeviceId;** **uint8_t appDevVerAndFlag;** **zbCounter_t cNumInClusters;** **zbClusterId_t *pInClusterList;** **zbCounter_t cNumOutClusters;** **zbClusterId_t *pOutClusterList;** **} zbSimpleDescriptor_t;** void ASL_MatchDescriptor_req (zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, zbSimpleDescriptor_t *pSimpleDescriptor);	typedef struct zbMatchDescriptorResponse_tag { zbStatus_t status; zbNwkAddr_t aNwkAddrOfInterest; zbSize_t matchLength; zbEndPoint_t matchList[1]; } **zbMatchDescriptorResponse_t;**

A switch has an On/Off Cluster (0x0006) as an output. A light has an On/Off Cluster as an input. They match. Two lights would not match.

Match_Desc_req may be broadcast (with 0xfffd) or unicast. Here is a simple experiment to cause a flurry of route requests and unicasts in a ZigBee network. Send out a **Match_Desc_req** with the Basic Cluster (0x0000) listed as an output cluster, on profile ID 0x0104. Every node in the network on the Home Automation profile will respond.

5.2.2 Backing Up and Caching Discovery Information

ZigBee utilizes the concepts of backing up and also caching the discovery information. This includes the following ZDP commands:

- System_Server_Discover_req
- Find_node_cache_req
- Discovery_Cache_req
- Discovery_store_req
- Node_Desc_store_req
- Power_Desc_store_req
- Active_EP_store_req
- Simple_Desc_store_req
- Remove_node_cache_req

The concept is fairly simple. The **System_Server_Discovery_req** permits nodes in the network to find the primary cache for everything from endpoints, to simple descriptors, to node descriptors, assuming nodes in the network stored copies of their information on the cache. Then, a commissioning tool or other node can retrieve the information. The trouble with this is that no vendors actually implement the primary discovery cache in a network. In fact, at the time of this writing, Freescale is the only vendor that has actually implemented these optional ZDP commands in their stack.

My advice is not to use them. Get the information directly from the nodes themselves. Or do without.

If you would like to use these commands anyway, here's how to do it. Use a **Discovery_store_req** first to allocate the space on the discovery cache for the various

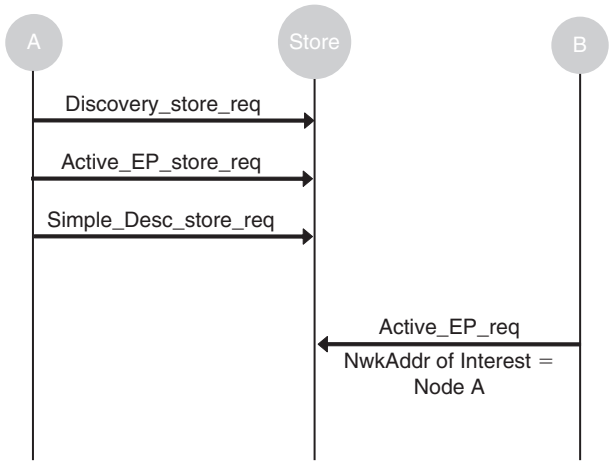

Figure 5.5: Discovery_store_req

items, including endpoints and simple descriptors, as seen in Figure 5.5. Then use the various store commands (e.g., **Simple_Desc_store_req**) to actually store the data on the cache. Using **System_Server_Discovery_req**, other nodes in the network can find the cache and request the sleeping node's information using commands such as **Active_EP_req**.

5.3 Binding

In Chapter 4, "ZigBee Applications," you learned all about APS (local) binding. I'll give a quick refresher here, and then talk about ZDP binding.

Binding provides a mechanism for attaching an endpoint on one node to one or more endpoints on another node. Binding can even be destined for groups of nodes. Then, when using APSDE-DATA.request, simply use the "indirect" addressing mode, and the request will be sent to each endpoint or group listed in the local binding table.

The binding table is smart, and keeps track of both the short (16-bit NwkAddr) and long (IEEE) address of a node. If a destination device has changed its short address (either due to a ZigBee End-Device moving from one parent to another in ZigBee stack profile 0x01, or due to a address conflict in ZigBee Pro), the binding table entry is updated automatically to point to that new address (see Figure 5.6).

As shown in Table 5.13, if the local application sent application data using indirect mode from endpoint 12, the packet would simply be dropped, as there is no source endpoint

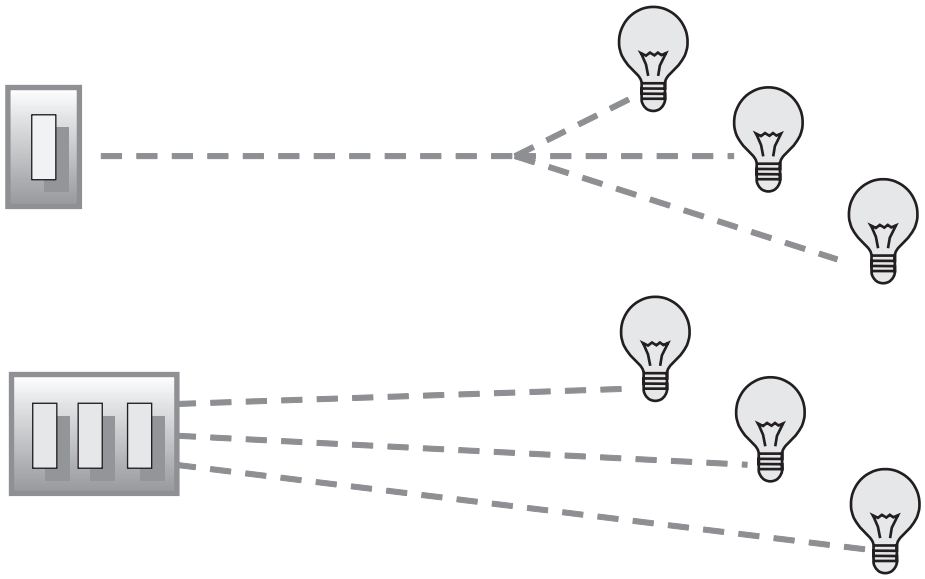

Figure 5.6: Binding Connects One Endpoint to One or More Other Endpoints

12 in the table. If the local application sent a APSDE-DATA.request using indirect mode from endpoint 5, it would go to three destinations: node 0x1234 endpoint 12, broadcast to group 0x9999, and to node 0x5678 endpoint 44.

ZDP provides over-the-air binding services to complement the local APS binding services. This allows a third-party tool (such as a remote control, or PC with a ZigBee dongle) to connect one application to another. It's easy to envision a drag-and-drop interface to bind switches to lights throughout a house, an office, or a hotel.

All ZDP binding services are optional. They are shown in Table 5.14.

Table 5.13: Sample Binding Table

Src EP	Destination Addr	Addr/Grp	Dst EP	Cluster ID
5	0x1234	A	12	0x0006
6	0x796F	A	240	0x0006
5	0x9999	G	--	0x0006
5	0x5678	A	44	0x0006

Table 5.14: ZDP Binding Services

ZDP Binding Services	Client Transmission (Req)	Server Processing (Rsp)
End_Device_Bind_req	O	O
Bind_req	O	O
Unbind_req	O	O

End_Device_Bind_req (see Figure 5.7) uses an optional state machine on the ZigBee Coordinator to bind or unbind two devices. This service can be useful in a "press-the-button-on-two-nodes-to-bind-them" operation, useful on some Home Automation products, but it's not generally useful in most ZigBee networks. One of the things I don't like about this command is that if it returns success, the caller has no idea if the targets were bound or unbound. It's a toggle!

The example in this section, *Section 5-3 Binding*, demonstrates a third-party node binding a switch to a light over-the-air. Granted, it's pretty simple, but it shows the concept of ZigBee commissioning with a third-party tool.

To run the example, simply compile and download the three targets (ZcNcbTool, ZedSrbSwitch, andZrSrbLight) from the BeeKit solution, and boot them all. Press SW1 on all of them to join each node to the network. Go to Application Mode on all three nodes by pressing and holding SW1 (long SW1). Press SW1 on the switch. Notice nothing happens. Then press SW2 in the tool to bind the switch to the light. Now press SW1 on the switch again and notice the light toggles.

There is one thing about over-the-air binding that is not obvious. The ZDP bind commands require an IEEE address, not a short address for the destination of the binding. If a node receives a ZDP bind command and it doesn't know about the destination address, it will issue a ZDP **NWK_Addr_req** to find the node, because it actually needs both long and short addresses to complete the operation.

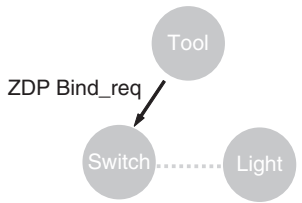

Figure 5.7: ZDP Bind Request

Table 5.15: ZDP Management Services

Network Management Services	Client Transmission (Req)	Server Processing (Rsp)
Mgmt_NWK_Disc_req (unicast)	O	O
Mgmt_Lqi_req (unicast)	O	O
Mgmt_Rtg_req (unicast)	O	O
Mgmt_Bind_req (unicast)	O	O
Mgmt_Leave_req (unicast)	O	O
Mgmt_Direct_Join_req (unicast)	O	O
Mgmt_Permit_Joining_req (unicast or broadcast)	O	M
Mgmt_Cache_req (unicast)	O	O
Mgmt_NWK_Update_req (unicast)	O	O

5.4 ZDP Management Services

The ZDP Management services are really handy optional services used for reading the various tables contained within ZigBee nodes, and to request certain common actions (see Table 5.15).

5.4.1 Network Discovery

The ZDP command **Mgmt_NWK_Disc_req** was implemented both to support frequency agility, which is the ability for the ZigBee network to change channels, and to help prevent PAN ID conflicts. A managing application can determine remotely what networks and nodes are in the vicinity of any node on the network.

PAN ID conflict happens when one network grows toward another. Perhaps they were both out of hearing range of each other when they started, and through chance happened to pick the same PAN ID, such as 0x1234. Now they've grown over time, and are beginning to overlap.

Changing channels in ZigBee is a fairly catastrophic event, and not one to be undertaken lightly. ZigBee is not a channel-hopping network, like Bluetooth™, for example. Instead, ZigBee relies on its robust CSMA-CA and O-QPSK technologies to continue to communicate even in noisy environments. But sometimes it's just necessary to change channels, and it would be a major hardship to tear down the network and rebuild it on another channel. This is the sort of thing that might happen at a hospital. The wireless

networks in a hospital are very carefully managed, and they do not want other wireless channels on the same frequency as their WiFi™ networks. If the WiFi network needed to change channels for some reason, it's possible the ZigBee network might have to as well.

The ZDP **Mgmt_NWK_Disc_req** command does exactly the same thing that ZDO does locally, when it determines what networks are nearby. It sends out a beacon request and reports the beacons that responded to a higher layer, that can then do something intelligent. In this case, the "higher layer" just happens to be on a remote managing node.

5.4.2 Table Management Services

ZDP contains services to read the various tables from remote ZigBee nodes. This can be useful in diagnostics during commissioning, or even at run-time. For example, the routing tables of various routers can be checked, and if one node in particular is always full while the other routers are not, perhaps a choke-point has been detected in the network. Another router may be needed in the vicinity.

ZigBee Table Management Services in ZDP:

- Mgmt_Lqi_req—the neighbor table
- Mgmt_Rtg_req—the routing table
- Mgmt_Bind_req—the (optional) binding table

Notice that there isn't any way to set these tables directly over ZigBee. Of course, an application specific cluster could be written to do this, but the proper way is to use the various other commands available that populate these tables. The binding table, for instance, is populated or cleared using the ZDP-Bind and ZDP-Unbind commands.

These tables can be quite large. To accommodate this, ZigBee allows them to be read from a starting index. For example, to read the entire neighbor table, use **Mgmt_Lqi_req** with a starting index of 0 to begin. Then, after it returns five or so entries, send another **Mgmt_Lqi_req** with a starting index of 5. That does mean that the operation is not always atomic, and can look strange if something has changed between the previous request and the next one.

The ZDP table requests and responses are listed in Tables 5.16, 5.17, and 5.18.

Table 5.16: Management Neighbor Table Request

Mgmt_Lqi_req	Mgmt_Lqi_rsp
void ASL_Mgmt_Lqi_req (zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, index_t index);	typedef struct zbNeighborTableList_tag { zbIeeeAddr_t aExtendedPanId; zbIeeeAddr_t aExtendedAddr; zbNwkAddr_t aNetworkAddr; uint8_t deviceProperty; bool_t permitJoining; uint8_t depth; uint8_t lqi; } zbNeighborTableList_t; typedef struct zbMgmtLqiResponse_tag { zbStatus_t status; zbCounter_t neighbourTableEntries; zbIndex_t startIndex; zbCounter_t neighbourTableListCount; zbNeighborTableList_t neighbourTableList[1]; } zbMgmtLqiResponse_t;

Table 5.17: Management Routing Table Request

Mgmt_Rtg_req	Mgmt_Rtg_rsp
void ASL_Mgmt_Rtg_req (zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, index_t index);	typedef struct routingTableList_tag { zbNwkAddr_t aDestinationAddress; uint8_t status; zbNwkAddr_t aNextHopAddress; } routingTableList_t; typedef struct zbMgmtRtgResponse_tag { zbStatus_t status; zbCounter_t routingTableEntries; index_t startIndex; zbCounter_t routingTableListCount; routingTableList_t routingTableList[1]; } zbMgmtRtgResponse_t;

Table 5.18: Management Binding Table Request

Mgmt_Bind_req	Mgmt_Bind_rsp

void ASL_Mgmt_Bind_req
(
zbCounter_t *pSequenceNumber,
zbNwkAddr_t aDestAddress,
index_t index
);
	typedef struct zbApsmeBindReq_tag {
	zbIeeeAddr_t aSrcAddr;
	zbEndPoint_t srcEndPoint;
	zbClusterId_t aClusterId;
	zbAddrMode_t dstAddrMode;
	zbIeeeAddr_t aDstAddr;
	zbEndPoint_t dstEndPoint;
	} zbApsmeBindReq_t;
	typedef struct zbMgmtBindResponse_tag {
	zbStatus_t status;
	zbCounter_t bindingTableEntries;
	zbIndex_t startIndex;
	zbCounter_t bindingTableListCount;
	zbApsmeBindEntry_t
	aBindingTableList[1];
	} zbMgmtBindResponse_t;

Don't confuse **Mgmt_Bind_req** (which retrieves a remote binding table) and **Bind_req** (which binds a remote node to another node).

5.4.3 Informing Other Nodes to Leave the Network

One of the other interesting things ZDP can do is to tell other nodes to leave the network. Why would you do this? Sometimes, such as when using the Commissioning Cluster from the ZigBee Cluster Library, a node might be commissioned with certain values on a commissioning network, and then told to go join a different network where it will do its work. Imagine a handheld device that an installer uses to make sure all of the lights, switches, thermostats, etc., are all functioning properly in each hotel room, before moving on to the next. Use **Mgmt_leave_req** for this purpose.

Mgmt_Direct_join_req is not used much. It's easier to simply use the network rejoin command, available through ZDO.

Mgmt_permit_joining_req can be very useful for disabling joining all throughout the network. Typically, this is the last step when commissioning a network. It closes it down to prevent other nodes getting on the network without permission.

5.5 Starting and Stopping ZigBee with ZDO

ZDO is the local-state machine that controls the state of the ZigBee node on and off the network. When a node boots up, it does not necessarily join a network right away. It may go into low-power mode, and wait for a button-press, or some other event that causes the node to decide it needs to network.

The Freescale platform uses a function called **ZDO_Start()** to join a node to the network. **ZDO_Start()** can start with any of the following options:

- gStartWithOutNvm_c
- gStartAssociationRejoinWithNvm_c
- gStartNwkRejoinWithNvm_c
- gStartSilentRejoinWithNvm_c

Starting without non-volatile memory (NVM) ensures that the node does not use anything it remembers from the last time it was booted and joined a network. Association join (or rejoin) uses the MAC association commands to join the network. Rejoin with NVM rejoins the network using the same PAN and channel selected previously. The node may get a new short address. The silent rejoin is very useful when nodes are reset after a battery change, or after a mains-powered network has reset after a power outage. The nodes do not actually *say* anything over the air, they simply start up and are capable of routing in a few tens of milliseconds.

To leave the network, Freescale uses one of two functions:

- ZDO_Stop()
- ZDO_Leave()

Stop leaves the network silently. Leave informs the node's parent so that the parent's internal tables can be cleaned up.

The example in this section, *Example 5-4 ZDO*, forces a node to leave one network and to rejoin the other. This operation is done fairly frequently in ZigBee network commissioning. A node doesn't know anything about the network it joins, other than the IEEE address of the parent it joined. Many times a ZigBee node needs to know more before deciding to remain on that network. It may, for example, query the network for a particular service.

In this example, the NCB board will attempt to join (randomly) one of the SRB boards. If it does, it will ask the SRB whether it supports the On/Off Cluster. If not, it will leave that network and attempt to join another. It continues doing this until it finds an On/Off Cluster that responds, and in this case, the light turns on.

Which of the networks issues the beacon response first is random, so the actual over-the-air capture may vary until the node finds the right parent.

5.6 ZDO, ZigBee, and Low Power

One of the most interesting aspects of ZigBee is the ability of nodes in a ZigBee network to last, not hours, not days, but for years on battery power. In fact, it's normal for a sleeping ZigBee device to last the shelf life of a couple of AA batteries (about five to seven years). Consider Figure 5.8. The ZigBee routers (in gray) and the ZigBee Coordinator (in black) are typically mains-powered. The ZigBee End Devices (in white) are ZigBee node types, which can sleep.

ZigBee End Devices can sleep, because they do not route. That is why they are called end devices: the route stops here. Notice the end devices in the figure below (for example, node 25) only have one connection to the ZigBee Network: the end-device's parent. The routers must have at least two connections. In reality, it's likely that all the routers in this house floor plan can all hear each other, but to simplify the figure, only some of the possible routes are shown.

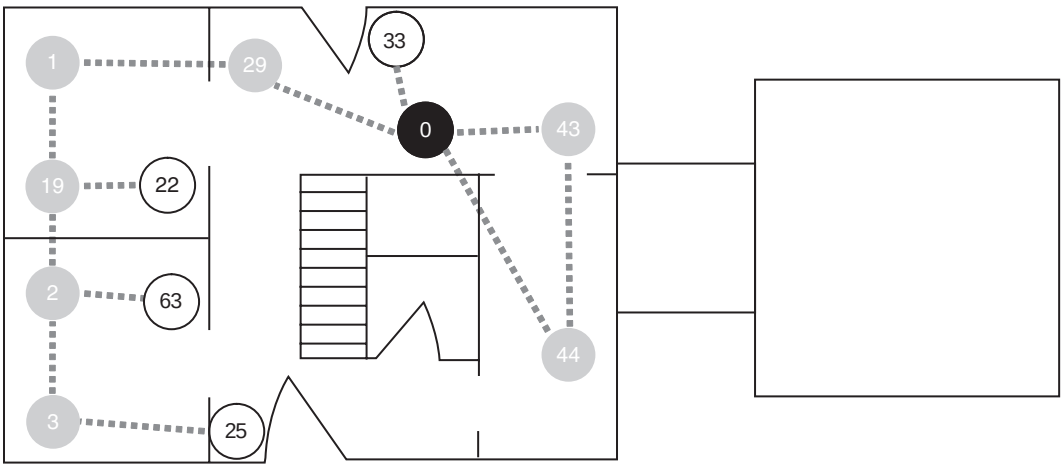

Figure 5.8: ZigBee End Devices Do Not Route

ZigBee is an asynchronous protocol. That is, a node may choose to transmit at any time. This makes sense when you think about how ZigBee is used. A light switch (let's use node 25 again, for example), can wake up and send a command to turn on the lights any time a user flips the switch. Or a factory automation system might need to send an alarm immediately. That is why routers must be awake all the time and ready to route a message.

Within the ZigBee Alliance, work is being done on an all-battery powered network for use in situations where latency doesn't matter, but eliminating mains-powered devices does, such as in a vineyard, or another agricultural setting. It doesn't matter if the temperature or moisture content is communicated now, or two minutes later. It would matter if the lights didn't turn on for two minutes! At the time of this writing, that work has not made it into any official ZigBee specifications.

So, let's go back to the example above. Someone flips a battery-powered ZigBee light switch. The switch causes an interrupt which wakes the CPU, which in turn wakes the radio. Once the system is fully powered (we're talking approximately a millisecond, here) the ZigBee End Device sends the command to turn the set of lights it controls on or off, and then it goes immediately back to sleep. Immediately is a relative term, so I'll go into the exact sequence of events with calculations, in a bit.

So what happens if you send the end-device a message while it is asleep? How does that end device receive it? That's where the special parent-child relationship comes in. In a ZigBee network, the parent will actually buffer messages for the sleeping child, delivering them when it wakes up.

However, the message is not buffered forever. The MAC generally buffers messages for about seven seconds. Some ZigBee stacks, like Freescale, are limited to this MAC timeout. Others are not. There is one other thing to note. If a given parent has many sleeping children, and many messages to deliver, the messages may time out before they are all delivered to the sleeping children. Generally, sleeping devices should wake up and communicate with some node in the network periodically, if the sleeping node can be configured or is to normally receive packets. Otherwise, just treat the end device as a low-power command, or data initiator. It wakes up when it wants, transmits data, then sleeps again.

One common question I get is this, "Can the radio wake the CPU upon receiving a valid ZigBee packet?" The answer is "Yes it can, but it doesn't make sense for a low-powered system." If the radio is awake enough to decode a signal, it is awake. That means it is consuming full power, somewhere in the neighborhood of 20–23 mA, which means the batteries won't last a very long time (days at best).

Table 5.19: ZigBee Battery Life Calculator

Battery capacity (mAh)	1900	1900 = 2 AA batteries
Supply efficiency (%)	100%	
System capacity (mAh)	1900	
Tx current (Radio) (mAh)	34	MC13193
Application payload size (bytes)	10	add 18–30 bytes for security
Packet frequency(s)	15	
Tx duration per packet (ms)	1.31	With security
Tx packets per day	5760	Calculated from packet frequency
PA current (or other Tx on) (mA)	0	
Rx current (Radio) (mA)	37	MC13193
Rx duration per packet (ms)	10	Waits for ACK (and msg) from parent
Rx packets per day	5760	
LNA current (or other Rx on) (mA)	0	
Radio sleep current (mA)	0.002	MC13193 sleep current
MCU active current (mA)	14	HCS08 Stop Mode 3
MCU sleep current (mA)	0.001	
MCU activity time in addition to radio (%)	20%	
MCU total activity time	120%	
MCU with AtoD on current	0	
MCU active time for AtoD per sample (ms)	0	
Number of samples per day	0	
Calculated radio duty cycle	0.08%	
Capacity used per day (mAh/day)	1.04	
Battery life in days	1828	
Battery life in years	5.0	

Table 5.19 is a battery calculator, and is included in Excel form on-line. As you can see from the calculations, it's very possible for a ZigBee node to last an entire five years on a pair of AA batteries.

Identifying all the power consumers in a system is not always easy. Some are obvious. A power regulator, consuming power to reduce the voltage from 9 volts down to 3, or that

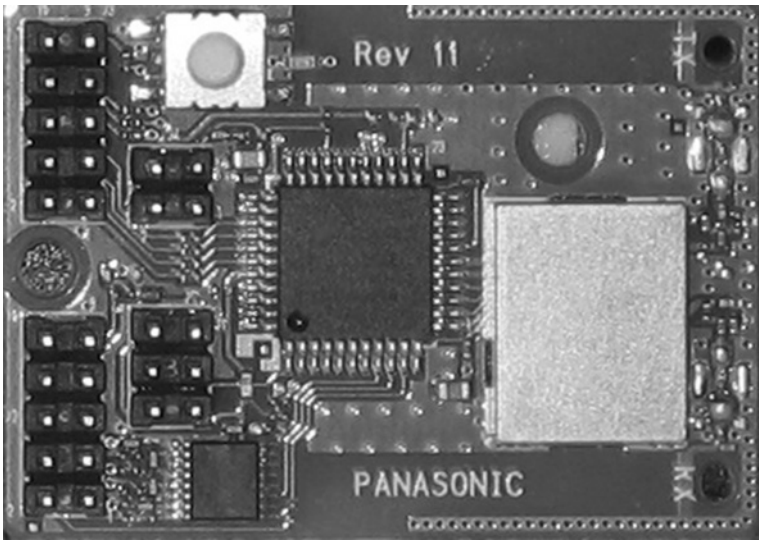

Figure 5.9: The Panasonic PAN802154HAR Low-Power ZigBee Board

TTL to RS2332 serial chip, or that blazing LED, is easy to figure out. But other power consumers are not so obvious.

For example, consider the Freescale HCS08GT60 microcontroller used both in the Freescale system-in-package MC13213, and in the two-chip solution with the MC13193 radio. This microcontroller uses the same core as the GB60, a part with significantly more GPIO pins brought out on the package. In the core on the GT60, the one used for the ZigBee nodes, those extra pins which aren't brought out on the smaller package are floating, and must be initialized to low output to prevent power consumption. If you don't turn them off, you'll wonder why your board is not achieving that 1.9 μA low-power sleep that the radio and MCU can.

The final example in this chapter is a low-power On/Off Switch. One thing that's very important to note in the Freescale solution is that it won't go into deep sleep unless all application timers have been stopped.

To run the demo, compile and download the ZcNcbOnOffLight and ZedPanOnOffSwitch. The "PAN" stands for the Panasonic PAN802154HAR. This board, pictured in Figure 5.9, can achieve 2 μA while asleep. Press the button, and the board wakes up, sends a toggle command to the light, and then goes back to sleep.

Alternately, use the Freescale SRB boards. The SRBs, while they are nice development boards, cannot achieve true low power under software control. This isn't due to the radio and MCU, but because other power consumers on the SRB board, such as the power regulator and USB chip, cannot be shut off.

When planning your project, always plan much more time than you think for low power. It seems so simple in concept, but there are always gotchas. One example is that the BDM debugger, used to debug programs in the Freescale environment, doesn't function once the MCU goes into low power. Low power is always more difficult than you think.

ZigBee provides no low power API. The API is always vendor-specific.

ZigBee End-Devices are the only nodes in a ZigBee network that achieve long battery life.