AN965

Microchip Stack for the ZigBee™ Protocol

Author: *David Flowers, Kim Otten, Yifeng Yang and Nilesh Rajbharti*
Microchip Technology Inc.

INTRODUCTION

ZigBee™ is a wireless network protocol specifically designed for low data rate sensors and control networks. There are a number of applications that can benefit from the ZigBee protocol: building automation networks, home security systems, industrial control networks, remote metering and PC peripherals are some of the many possible applications.

Compared to other wireless protocols, the ZigBee wireless protocol offers low complexity, reduced resource requirements and most importantly, a standard set of specifications. It also offers three frequency bands of operation along with a number of network configurations and optional security capability.

If you are currently exploring alternatives to your existing control network technologies, such as RS-422, RS-485 or proprietary wireless protocol, the ZigBee protocol could be the solution you need.

This application note is specifically designed to assist you in adopting the ZigBee protocol for your application. You can use the Microchip Stack for the ZigBee protocol provided in this application note to quickly build your application. To illustrate the usage of the Stack, several working demo applications are included. You can use these demo applications as a reference or simply modify and adopt them to your requirements.

Note:	Please note that in order to distribute a product that utilizes the Microchip Stack for the ZigBee protocol, you must become a ZigBee Adopter through the ZigBee Alliance.

Commonly asked questions about the Microchip Stack and its usage, along with their answers, are provided at the end of this document in **"Answers to Common Questions"**.

ASSUMPTION

This document assumes that you are familiar with the C programming language. This document uses extensive terminology from the ZigBee protocol and IEEE 802.15.4™ specifications and provides a brief overview of the ZigBee protocol specification. You are advised to read the ZigBee protocol and IEEE 802.15.4 specifications in detail.

FEATURES

The Microchip Stack for the ZigBee protocol is designed to evolve with the ZigBee wireless protocol specification. At the time this document was published, the current ZigBee protocol specification version was v1.0. This document applies to Microchip Stack releases v1.0-3.8 and greater.

The Microchip Stack offers the following features:

- Certified ZigBee protocol v1.0 compliant platform
- Support for 2.4 GHz frequency band
- Support for all ZigBee protocol device types (Coordinators, Routers and End devices)
- Implements nonvolatile storage for neighbor and binding tables
- Portable across many of the PIC18 family of microcontrollers
- RTOS and application independent
- Out-of-box support for Microchip MPLAB® C18 compiler
- Modular design and standard nomenclature aligns with the nomenclature used in the ZigBee protocol and IEEE 802.15.4 specifications

For the latest features and enhancements over previous releases, refer to the Readme file distributed with the source code for the Microchip Stack for the ZigBee™ Protocol.

LIMITATIONS

Version 1.0-3.8 of the Microchip Stack for the ZigBee Protocol is the first version to be granted the status of ZigBee Compliant Platform (ZCP). For information on the ZCP status of the current revision, please refer to the Readme file distributed with the source code.

CONSIDERATIONS

The ZigBee protocol specification leaves many higher level decisions up to the developer. As such, the Microchip Stack provides no explicit support for some functions:

- Supports non-slotted networks only (no beacon network support)
- Network addresses of nodes that have left the network cannot be reassigned
- Automatic removal of nodes from the neighbor table is not performed
- PAN ID conflict resolution is not supported
- Automatic route repair is not performed
- Alternate PAN coordinator capability

ZigBee™ PROTOCOL OVERVIEW

ZigBee is a standard wireless network protocol designed for low data rate control networks. It is layered on top of the IEEE 802.15.4 specification and provides a standard methodology for functions, including network formation, messaging and device discovery.

IEEE 802.15.4

The ZigBee protocol uses the IEEE 802.15.4 specification as its Medium Access Layer (MAC) and Physical Layer (PHY). The IEEE 802.15.4 defines three frequency bands of operations: 2.4 GHz, 915 MHz and 868 MHz. Each frequency band offers a fixed number of channels. For example, the 2.4 GHz frequency band offers 16 channels (channels 11-26), 915 MHz offers 10 channels (channels 1-10) and 868 MHz offers 1 channel (channel 0).

The bit rate of the protocol depends on the operational frequency. The 2.4 GHz band provides 250 kbps, 915 MHz provides 40 kbps and 868 MHz provides a 20 kbps data rate. The actual data throughput will be less than the specified bit rate due to the packet overhead and processing delays.

The maximum length of an IEEE 802.15.4 MAC packet is 127 bytes, including a 16-bit CRC value. The 16-bit CRC value verifies the frame integrity. In addition, IEEE 802.15.4 optionally uses an Acknowledged data transfer mechanism. With this method, all frames with a special ACK flag set are Acknowledged by its receiver. This makes sure that a frame is in fact delivered. If the frame is transmitted with an ACK flag set and the Acknowledgement is not received within a certain time-out period, the transmitter will retry the transmission for a fixed number of times before declaring an error. It is important to note that the reception of an Acknowledgement simply indicates that a frame was properly received by the MAC layer. It does not, however, indicate that the frame was processed correctly. It is possible that the MAC layer of the receiving node received and Acknowledged a frame correctly, but due to the lack of processing resources, a frame might be discarded by upper layers. As a result, the upper layers may require additional Acknowledgement response.

DEVICE TYPES

IEEE 802.15.4 defines two types of devices. These devices types are shown in Table 1. Listed in Table 2 are the three types of ZigBee protocol devices as they relate to the IEEE device types.

TABLE 1: IEEE 802.15.4™ DEVICE TYPES

Device Type	Services Offered	Typical Power Source	Typical Receiver Configuration
Full Function Device (FFD)	Most or All	Mains	On when Idle
Reduced Function Device (RFD)	Limited	Battery	Off when Idle

TABLE 2: ZigBee™ PROTOCOL DEVICE TYPES

ZigBee Protocol Device	IEEE Device Type	Typical Function
Coordinator	FFD	One per network. Forms the network, allocates network addresses, holds binding table.
Router	FFD	Optional. Extends the physical range of the network. Allows more nodes to join the network. May also perform monitoring and/or control functions.
End	FFD or RFD	Performs monitoring and/or control functions.

NETWORK CONFIGURATIONS

A ZigBee protocol wireless network may assume many types of configurations. In all network configurations, there are at least two main components:

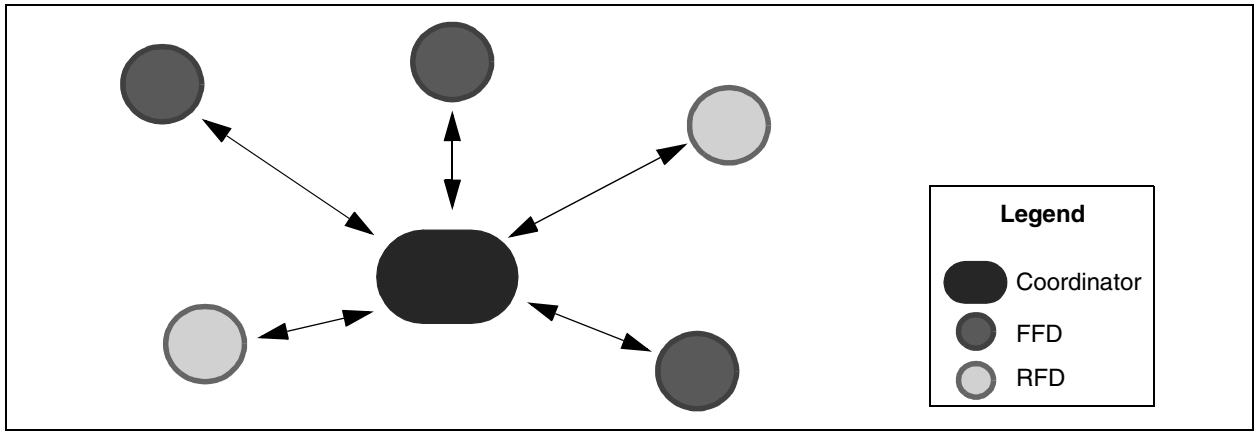
- Coordinator node
- End device

The ZigBee protocol coordinator is a special variant of a Full Function Device (FFD) that implements a larger set of ZigBee protocol services. An end device may be an FFD or a Reduced Function Device (RFD). An RFD is the smallest and simplest ZigBee protocol node. It implements only a minimal set of ZigBee protocol services. A third and optional component, the ZigBee protocol router, is present in some network configurations.

Star Network Configuration

A star network configuration consists of one ZigBee protocol coordinator node and one or more end devices. In a star network, all end devices communicate only with the coordinator. If an end device needs to transfer data to another end device, it sends its data to the coordinator. The coordinator, in turn, forwards the data to the intended recipient.

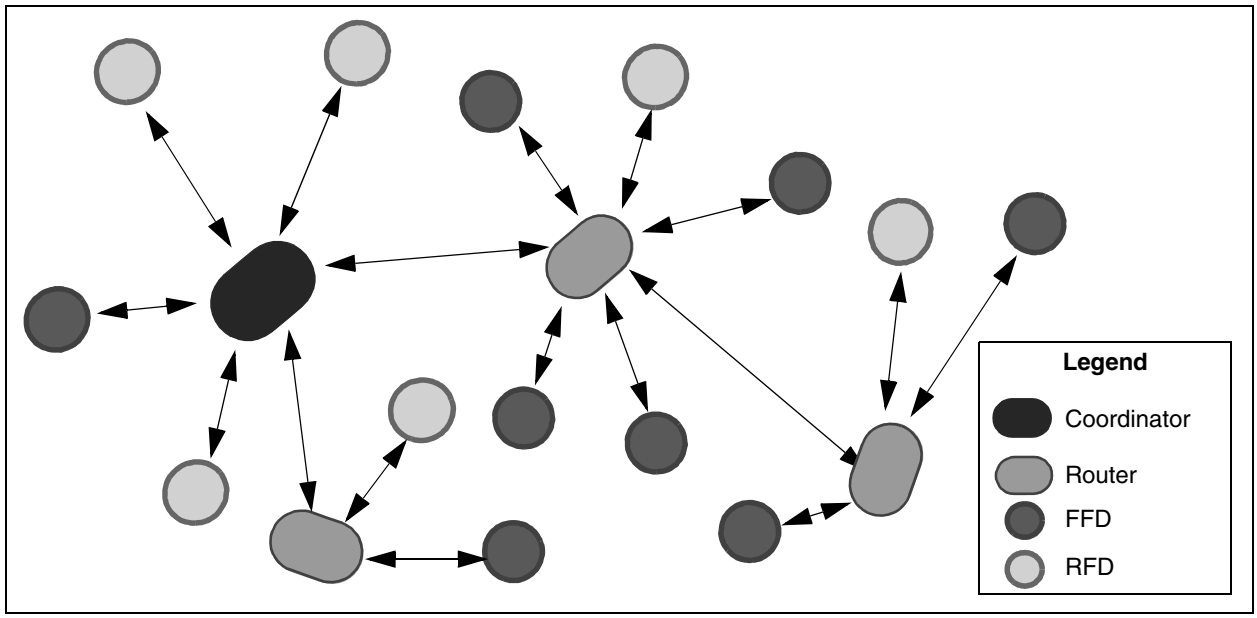
FIGURE 1: STAR NETWORK CONFIGURATION



Cluster Tree Topology

Another network configuration is a cluster tree topology. In this configuration, end devices may join either to the ZigBee protocol coordinator or to the ZigBee protocol routers. Routers serve two functions. One is to increase the number of nodes that can be on a network. The other is to extend the physical range of the network. With the addition of a router, an end device need not be in radio range of the coordinator. All messages in a cluster tree topology are routed along the tree.

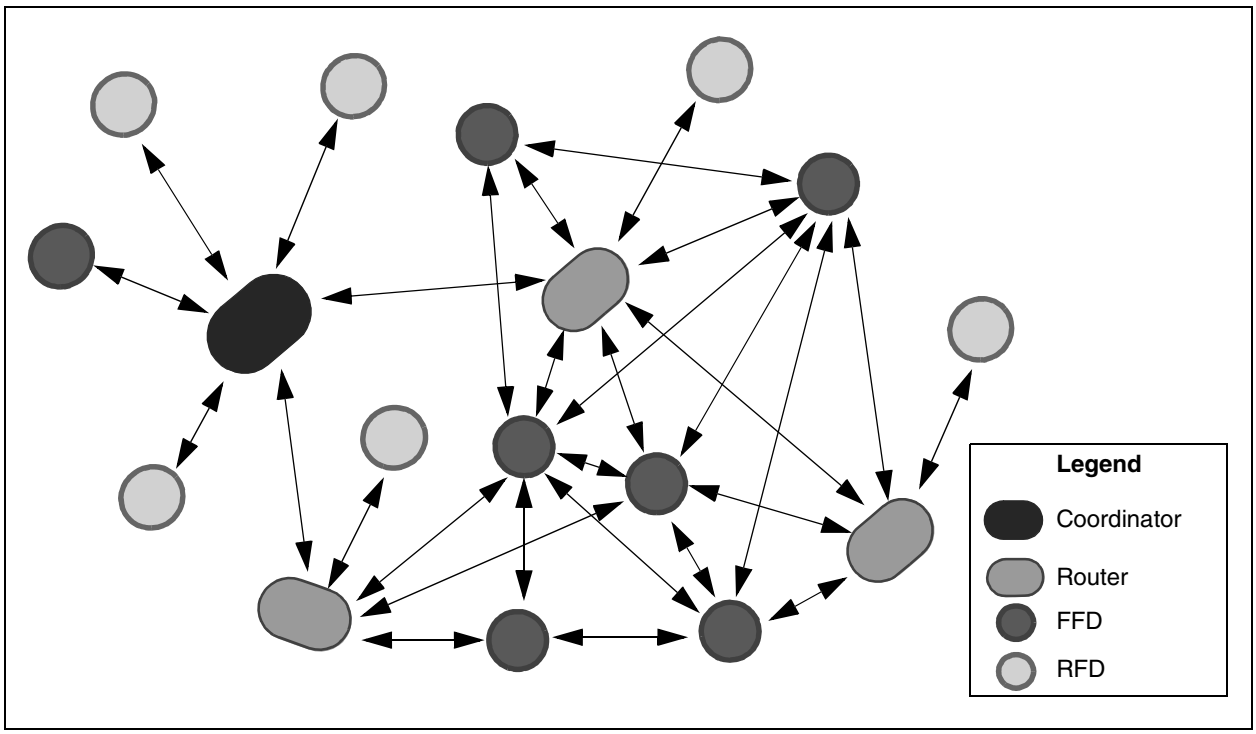
FIGURE 2: CLUSTER TREE TOPOLOGY



Mesh Network

A mesh network is similar to a cluster tree configuration, except that FFDs can route messages directly to other FFDs instead of following the tree structure. Messages to RFDs must still go through the RFD's parent. The advantages of this topology are that message latency can be reduced and reliability is increased.

FIGURE 3: MESH NETWORK



The cluster tree and mesh topologies are also known as multi-hop networks, due to their abilities to route packets through multiple devices, while the star topology is a single-hop network. A ZigBee protocol network is a multi-access network, meaning that all nodes in a network have equal access to the medium of communication.

There are two types of multi-access mechanisms, beacon and non-beacon. In a non-beacon enabled network, all nodes in a network are allowed to transmit at any time as long as the channel is Idle. In a beacon enabled network, nodes are allowed to transmit in predefined time slots only. The coordinator periodically begins with a superframe identified as a beacon frame, and all nodes in the network are expected to synchronize to this frame. Each node is assigned a specific slot in the superframe during which it is allowed to transmit and receive its data. A superframe may also contain a common slot during which all nodes compete to access the channel. The current version of the Microchip Stack supports only non-beacon networks.

ZigBee PROTOCOL TERMINOLOGY

A ZigBee protocol **profile** is simply a description of logical components (**devices**) and their interfaces. There is often no code associated with a profile. Each piece of data that can be passed between devices, such as a switch state or a potentiometer reading, is called an **attribute**. Each attribute is assigned to a unique identifier. These attributes are grouped in **clusters**. Each cluster is assigned to a unique identifier. Interfaces are specified at the cluster level, not at the attribute level, though attributes are transferred individually.

The profile defines the values of the Attribute IDs and the Cluster IDs, as well as the format of each attribute. For example, in the Home Control, Lighting profile, the cluster OnOffDRC of the Dimmer Remote Control (DRC) device contains one attribute, OnOff, which must be an unsigned 8-bit value, with the value 0xFF meaning “on”, the value 0x00 meaning “off” and the value 0xF0 meaning “toggle output”.

The profile also describes which clusters are mandatory and which are optional for each device. In addition, the profile may define some optional ZigBee protocol services as mandatory.

The user can take these definitions and write his code to use them. He can write the code any way he wants, grouping the functions any way he wants as long as he supports the mandatory clusters and services, and uses the attributes as they are defined in the profile. This way, one manufacturer’s switch will work with another manufacturer’s load controller.

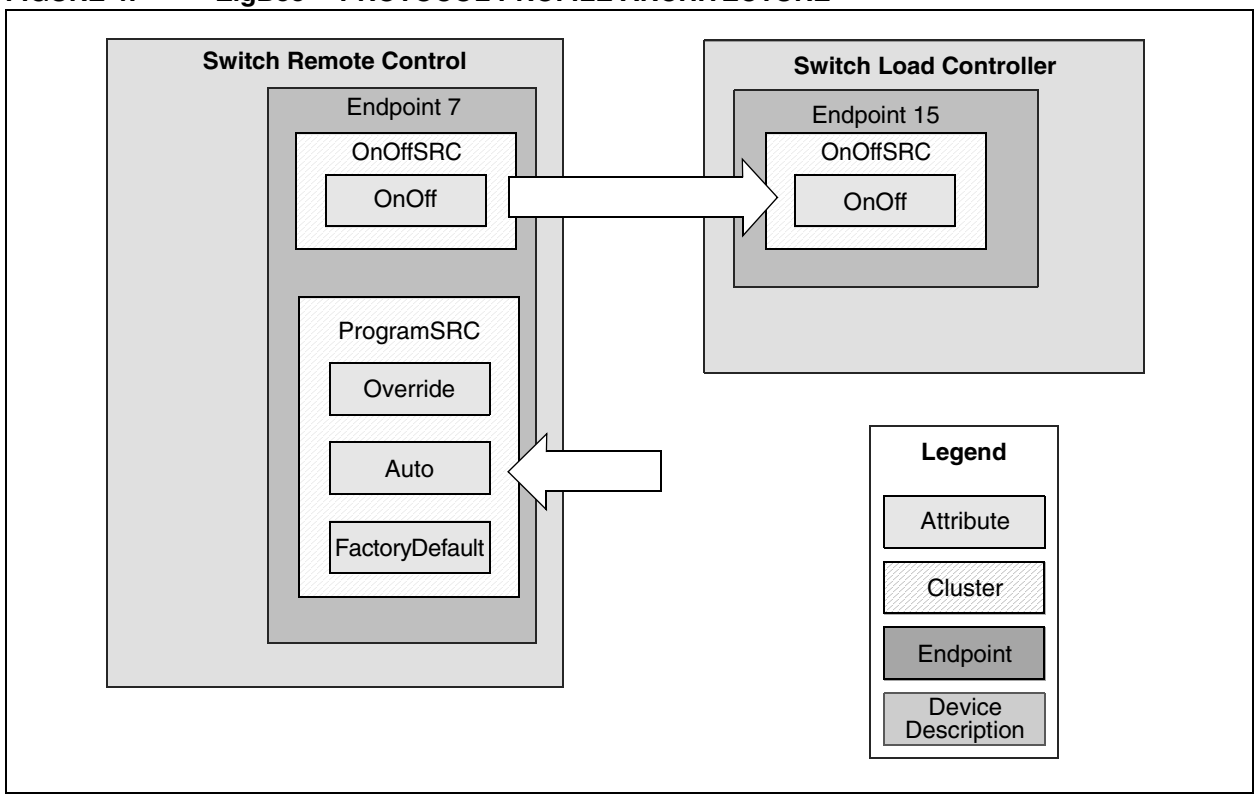
As an example, the Home Control, Lighting profile specifies six devices. The Microchip Stack for the ZigBee protocol provides support for this profile via a header file with the following information:

- Profile ID
- Device IDs and Versions
- Cluster IDs
- Attribute IDs
- Attribute Data Types

Each functional block of code that supports one or more clusters is called an **endpoint**. Different devices communicate via their endpoints and the clusters they support.

Figure 4 shows graphically how the various terms relate to each other. The figure shows two devices from the Home Control, Lighting profile. Each device has only one endpoint. The Switch Load Controller (e.g., a light) has one input cluster on that endpoint. The Switch Remote Control (e.g., a switch) has one output cluster and one input cluster on its endpoint. The switch could also be implemented such that the two clusters are on separate endpoints. Data flow is at the cluster level.

FIGURE 4: ZigBee™ PROTOCOL PROFILE ARCHITECTURE



Message Types and Binding

Devices can communicate with other devices on the network if they know the network addresses of those devices. These messages are called **direct** messages. However, there is a great deal of overhead involved in discovering and maintaining these destination addresses. ZigBee protocol offers a feature called “binding” to simplify messaging. The ZigBee protocol coordinator can create a table of matches at the cluster/endpoint level between the services and the needs of the devices in the network. Each of these pairs is called a “binding”. A binding can be requested by the devices themselves, or it can be created by the coordinator or another device. Once a binding is created, two devices can communicate through the coordinator. The source device sends its message to the coordinator, which then relays the message to one or more destination devices. These messages are called **indirect** messages.

ZigBee Protocol Message Format

A ZigBee protocol message consists of up to 127 bytes in the following fields:

MAC Header – The MAC header contains the MAC Frame Control fields, Beacon Sequence Number (BSN) and addressing information of the message as it is currently being transmitted. Note that it may not reflect the actual source or the final destination of the message if the message is being routed. The generation and use of this header is transparent to the application code.

- **Network Layer (NWK) Header** – This header contains, with other information, the actual source and final destination of the message. The generation and use of this header is transparent to the application code.
- **Application Support Sub-Layer (APS) Header**– This header includes the Profile ID, Cluster ID and destination endpoint of the current message. The generation and use of this header is transparent to the application code.
- **APS Payload** – This field contains the ZigBee protocol frame for the application to process. The application code is responsible for filling in the APS Payload.

ZigBee Protocol Frame Formats

ZigBee protocol defines two frame formats: the Key Value Pair (KVP) frame format and the Message (MSG) frame format. Both frame formats are associated with a Cluster ID, but KVP frames are designed to transfer one piece of information associated with an attribute using a strict structure, while MSG frames transfer information using a free form structure. The profile for the application will specify what frame formats should be used to transfer what information and the format of any MSG frames. Due to the difference in frame format, a cluster not may utilize both KVP frames and MSG frames.

KVP Frames

A KVP frame contains the following information, in order:

1. Transaction Count
2. Frame Type
3. Transactions
   - Transaction Sequence Number
   - Command Type and Attribute Data Type
   - Attribute ID
   - Error Code (optional)
   - Attribute Data (variable size)

The Command Type indicates what the application is supposed to do with the information. For example, the command, “Set”, requires the recipient to set the value of the attribute indicated by the Attribute ID to the value in Attribute Data, and the command, “Get with Acknowledge”, requires the recipient to send the value of the attribute indicated by the Attribute ID.

MSG Frames

An MSG frame contains the following information, in order:

1. Transaction Count
2. Frame Type
3. Transactions
   - Transaction Sequence Number
   - Transaction Length
   - Transaction Data

Both the sending and the receiving devices need to be aware of the transaction data format.

Addressing

Each node on a ZigBee protocol network will have two addresses: a 64-bit MAC address and a 16-bit network address. There are also two forms of message addressing available.

IEEE Extended Unique Identifiers – EUI-64

Each and every device that communicates using ZigBee protocol must have a globally unique, 64-bit MAC address. This address is made up of a 24-bit Organizationally Unique Identifier (OUI) plus 40 bits assigned by the manufacturer. OUIs must be purchased from IEEE to ensure global uniqueness. You may obtain your own OUI number by applying at the following web address:

https://standards.ieee.org/regauth/oui/forms/OUT-form.shtml

If your organization already has an OUI for Ethernet applications, you may use the same OUI for ZigBee protocol applications. You may not use the Microchip OUI for production devices.

Network Addresses

Devices use their extended addresses to communicate while they are in the process of joining a network. When a device successfully joins a ZigBee protocol network, it is assigned a 16-bit network address, which it then uses to communicate with other devices on the network.

Unicast

In a unicast message, the address of the destination node is provided in the MAC layer header of the packet. Only the device who has that address receives the message.

Broadcast

In a broadcast packet, the MAC layer destination address is 0xFFFF. Any transceiver that is RX enabled will receive the message. This form of addressing is used when joining a network, discovering routes in the network and performing other ZigBee protocol discovery functions. ZigBee protocol implements a "passive-acknowledge" of broadcast packets. What is meant by passive-acknowledge is that when a device originates or retransmits a broadcast packet, it will listen for all of its known neighbors to retransmit the packet. If all neighbors have not replicated the message within *nwkPassiveAckTimeout* seconds, it will retransmit the packet until it hears the retransmissions from all of its known neighbors or the packet times out after *nwkNetworkBroadcastDeliveryTime* seconds.

Data Transfer Mechanism

In a non-beacon network, when a device wants to send a data frame, it simply waits for the channel to become Idle. Upon detecting an Idle channel condition, the device may transmit the frame.

If the destination device is an FFD, then its transceiver is always on, and other devices may transmit to it at any time. This capability allows for mesh networking. However, if the device is an RFD, then it may power down its transceiver when it is Idle to conserve power. The RFD will not be able to receive messages while it is in this state. This condition is handled by requiring that all messages to and from an RFD go through the RFD's FFD parent. When the RFD powers up its transceiver, it requests messages from its parent. If the parent has buffered a message for the child, it then forwards that message to the child. This allows the RFD to conserve power, but requires that the FFD have enough RAM to buffer messages for all of its children. If the child does not request messages within a certain time period (*macTransactionPersistenceTime*), the message will time out, and the parent will discard it.

Routing

The Microchip Stack has the ability to route messages. Routing is done automatically by the Stack, without any intervention from the end application. Routing allows the range of the network to be extended by allowing end devices beyond radio distance of the ZigBee protocol coordinator to join the network through a ZigBee protocol router.

The type of routing desired for a message is indicated when the message is sent. There are three routing options available:

1. **SUPPRESS** – If a discovered mesh route exists, the message is routed along that route. Otherwise, the message is routed along the tree.
2. **ENABLE** – If a discovered mesh route exists, the message is routed along that route. If a mesh route has not been determined, the router can initiate route discovery. When discovery is complete, the message will be sent along the calculated route. If the router does not have route capacity, it will route the message along the tree.
3. **FORCE** – If the router has route capacity, it will initiate route discovery, even if a route already exists. When discovery is complete, the message will be sent along the calculated route. If the router does not have route capacity, it will route the message along the tree. This option should be used sparingly, as it generates a great deal of network traffic. Its primary use is to repair a broken route.

Network Association

A new ZigBee protocol network is first established by a ZigBee protocol coordinator. On start-up, a ZigBee protocol coordinator searches for other ZigBee protocol coordinators operating on its allowed channels. Based on the channel energy and number of networks found on each allowed channel, it establishes its own network and selects a unique 16-bit PAN ID. Once a new network is established, ZigBee protocol routers and end devices are allowed to join the network.

Once a network is formed, it is possible that due to the physical changes, more than one network may overlap and a PAN ID conflict may arise. In that situation, a coordinator may initiate a PAN ID conflict resolution procedure and one of the coordinators would change its PAN ID and/or channel. The affected coordinator would instruct all of its child devices to make the necessary changes. The current version of the Microchip Stack does not support PAN ID conflict resolution.

ZigBee protocol devices store information about other nodes in the network, including parent and child nodes, in an area of nonvolatile memory called a neighbor table. On power-up, if a child device determines through its neighbor table that it once was part of a network, it may execute an orphan notification procedure to locate its previously associated network. Devices that receive the orphan notification will check their neighbor tables and see if that device is one of their children. If so, the parent device will inform the child device of its place in the network. If orphan notification fails or the child device has no parent entry in its neighbor table, then it will try to join the network as a new device. It will generate a list of potential parents and try to join an existing network at the optimal depth.

Once on a network, a device can disassociate from the network either by being requested to leave the network by its parent or by requesting disassociation itself.

The amount of time that a device spends determining the channel energy and available networks on each channel is specified by the `ScanDuration` parameter. Refer to **"ZigBee Protocol Timing"** for details on how this parameter is derived. For the 2.4 GHz frequency band, the scanning time in seconds is calculated by the equation:

EQUATION 1:

$$0.01536 * (2^{\text{ScanDuration}} + 1)$$

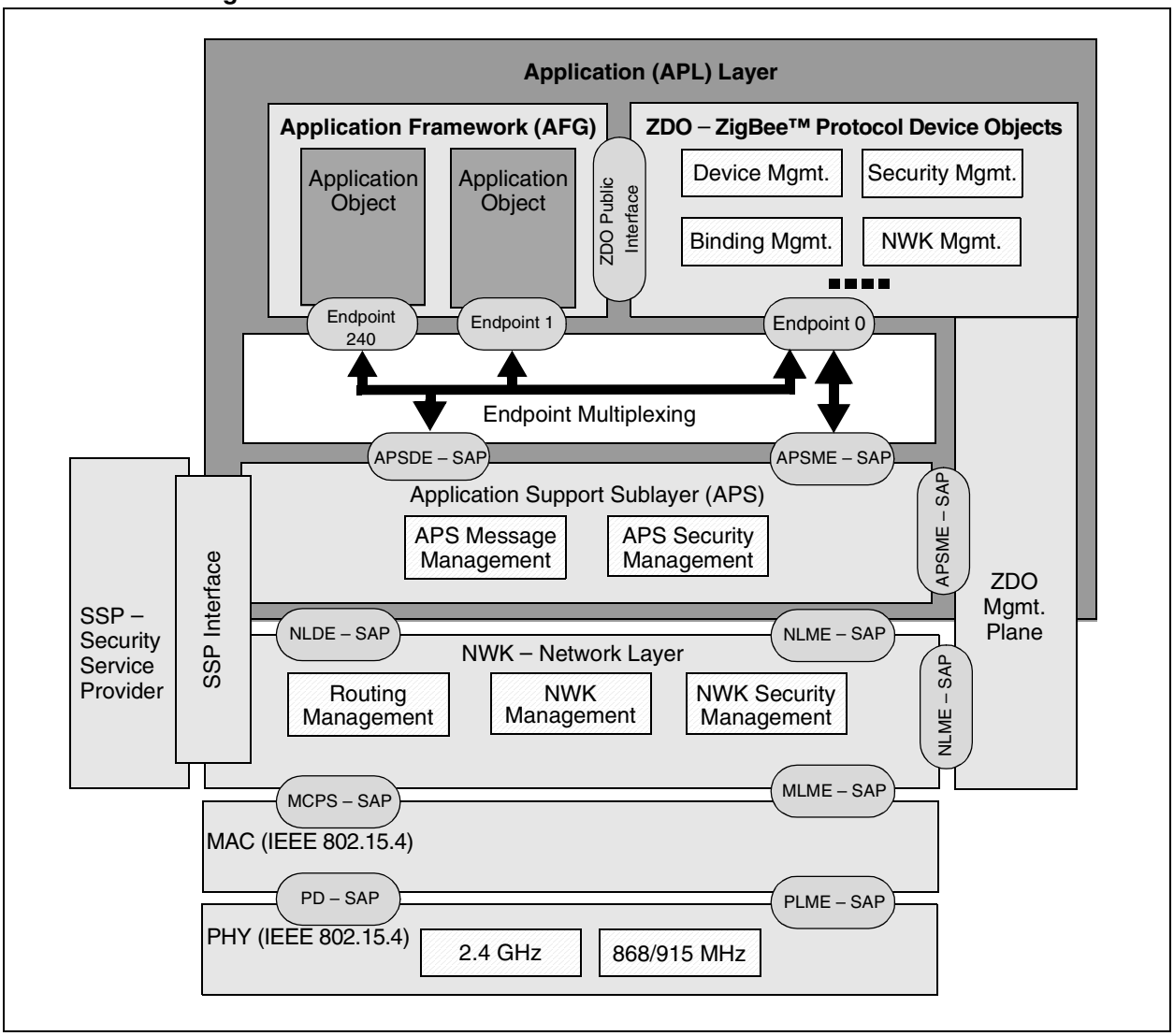
For the Microchip Stack, `ScanDuration` may be between 0 and 14, giving a scan time of 0.031 seconds to 4.2 minutes per channel. If `ScanDuration` is set to 8 and all 16 channels are specified, a device will spend over one minute performing each required scan. ZigBee protocol routers and end devices perform one scan to determine available networks, but ZigBee protocol coordinators perform two scans, one to sample channel energy and one to determine existing networks. The specified scan duration needs to balance the time needed to adequately perform each scan on the specified channels with the amount of time allocated for system start-up.

STACK ARCHITECTURE

The Microchip Stack is written in the C programming language, and is designed to run on Microchip's PIC® microcontrollers. The Microchip Stack uses internal Flash program memory to store many parameters, including MAC address, neighbor table and binding table. Consequently, you must use a self-programmable Flash memory microcontroller. If required, you may modify the Nonvolatile Memory (NVM) routines to support any other type of NVM and not use a self-programmable microcontroller. In addition, the Stack is targeted to run on the PICDEM™ Z Demonstration Board. However, it can be easily ported to any hardware equipped with a compatible PIC microcontroller.

The Microchip Stack was designed to follow the ZigBee protocol and IEEE 802.15.4 specifications, with each layer in its own source file. Terminology is copied as closely as possible from the specifications. The primitives defined in the two specifications are used to interface with the Stack through a single function call, using the parameter list defined for the primitives in the specifications. Refer to **"Interfacing with the Microchip Stack for the ZigBee Protocol"** for detailed descriptions of typical primitive flow. Refer to the ZigBee protocol and IEEE 802.15.4 specifications for detailed descriptions of the primitives and their parameter lists.

FIGURE 5: ZigBee™ PROTOCOL STACK ARCHITECTURE



TYPICAL ZigBee™ PROTOCOL NODE HARDWARE

To create a typical ZigBee protocol node using the Microchip Stack, you need, at a minimum, the following components:

- One Microchip microcontroller with an SPI interface
- Microchip MFR24J40 RF transceiver with required external components
- An antenna – may be PCB trace antenna or monopole antenna

As shown in Figure 6, the controller connects to the RF transceiver via the SPI bus and a few discrete control signals. The controller acts as an SPI master and the RF transceiver acts as a slave. The controller implements the IEEE 802.15.4 MAC layer and ZigBee protocol layers. It also contains application-specific logic. It uses the SPI bus to interact with the RF transceiver.

The Microchip Stack provides a fully integrated driver, which relieves the main application from managing RF transceiver functions. The hardware resources required by the PIC18F microcontroller family to drive the RF transceiver in the default implementation (provided in the PICDEM Z Demonstration Kit) are listed in Table 3. If you are using a Microchip reference schematic for a ZigBee protocol node, you may start using the Microchip Stack without any modifications. If required, you may relocate some of the non-SPI control signals to other port pins to suit your application hardware. In this case, you will have to modify the interface definitions to include the correct pin assignments.

TABLE 3: PIC18F MICROCONTROLLER RESOURCES REQUIRED BY THE ZigBee™ STACK

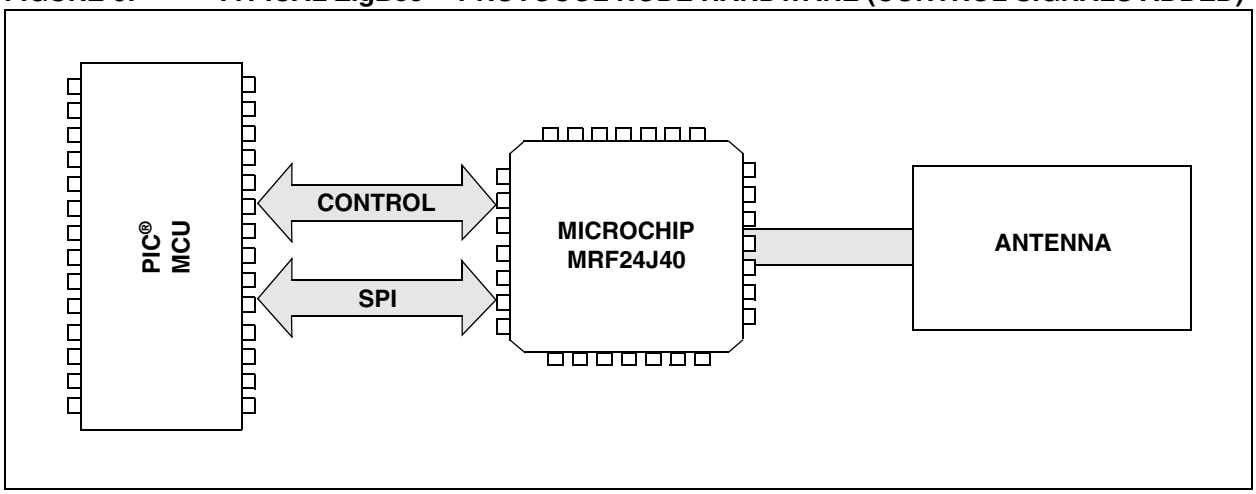
Resource	Description
INT0	Used to accept interrupts from MRF24J40 transceiver
TMR0	Used for symbol timer
RC0	Chip selection
RC1	Voltage regulator/wake-up pin
RC2	Transceiver Reset
RC3	SPI SCK
RC4	SPI SDI
RC5	SPI SDO

The Microchip reference design for the ZigBee protocol implements both a PCB trace antenna and a monopole antenna design. Depending on your choice of antenna, you will have to remove and solder a few components. Refer to the *"PICDEM™ Z Demonstration Kit User's Guide"* for more information (see **"References"**).

The Microchip reference design uses a 3.3V supply for both the controller and the RF transceiver. Depending on your requirements, you may either use mains or a battery power supply. Typically, ZigBee protocol coordinators and routers would operate on mains power supply and end devices would operate on a battery. When using a battery power supply, you must make sure that you operate the transceiver within the specified voltage range.

Refer to the *"PICDEM™ Z Demonstration Kit User's Guide"* for a Microchip reference design for a ZigBee protocol node. Refer to the Readme file for more information about supported transceivers, as well as addition PIC microcontroller and demo board support.

FIGURE 6: TYPICAL ZigBee™ PROTOCOL NODE HARDWARE (CONTROL SIGNALS ADDED)



ZENA™ ANALYZER – MICROCHIP'S ZigBee™ PROTOCOL STACK CONFIGURATION TOOL AND WIRELESS NETWORK ANALYZER

To assist in the development of ZigBee protocol applications, Microchip provides a low-cost network analyzer software, called ZENA. The ZENA PC software also contains a tool to create application-specific configuration files and linker scripts for ZigBee protocol applications. The ZENA demo software is provided free as part of the Microchip Stack for the ZigBee protocol installation and is located in the `MpZBee` directory. Refer to the "*ZENA™ Wireless Network Analyzer User's Guide"* for more information on using this tool.

The demo version of ZENA software provides the capability of creating application-specific source files to support the Microchip Stack and analyzing previously captured wireless network traffic. The full-featured version of ZENA software, which includes the ability to capture real-time wireless network activity, is available as a separate kit and includes an RF sniffer that can be connected to a PC through a USB port.

Note:	When ZENA software is used to configure a ZigBee protocol application, it will create three files for the application: `zigbee.def`, `myZigBee.c` and `zLink.lkr`. The `zigbee.def` and `myZigBee.c` files contain information critical to the configuration of the Stack. The `zLink.lkr` is the linker script for the application. It is highly recommended that you use ZENA software to generate these files, rather than editing the files manually, since the files are interdependent.

INSTALLING SOURCE FILES

The complete Microchip Stack source code is available for download from the Microchip web site. The source code is distributed in a single Windows® operating system installation file. Perform the following steps to complete the installation:

1. Execute the installation file. A Windows operating system installation wizard will guide you through the installation process.
2. Before the software is installed, you must accept the software license agreement by clicking **"I Accept"**.
3. If you wish to install the GERBER files for the PICDEM Z Demonstration Board, you must accept a second license agreement. If you choose not to accept this agreement, you can still install the Stack software.
4. After completion of the installation process, you should see the "Microchip Software Stack for ZigBee" protocol program group. The complete source code will be copied in the `MpZBee` directory in the root drive of your computer.
5. Refer to the Readme file distributed with the source code for the list of enhancements and limitations of the installed version.

SOURCE FILE ORGANIZATION

The Microchip Stack consists of multiple source files. For compatibility with other Microchip applications, files that are common to multiple application notes are stored in a single directory. ZigBee protocol Stack-specific files are stored in another directory. Each demo application is stored in its own directory. Table 4 shows the directory structure:

TABLE 4: SOURCE FILE DIRECTORY STRUCTURE

Directory Name	Contents
`Common`	Source files common to the Microchip Stack for the ZigBee™ protocol and other Microchip application notes.
`DemoCoordinator`	Source code for a demonstration ZigBee protocol coordinator application, plus a template for creating other ZigBee protocol coordinator applications.
`DemoRFD`	Source code for a demonstration ZigBee protocol RFD application, plus templates for creating other ZigBee protocol RFD and FFD end device applications.
`DemoRouter`	Source code for a demonstration ZigBee protocol router application, plus a template for creating other ZigBee protocol router applications.
`Documentation`	Microchip Stack for ZigBee protocol documentation.
`ZigBeeStack`	Microchip Stack for the ZigBee protocol source files.

The Stack files contain logic for all supported types of ZigBee protocol applications; however, only one set of logic will be enabled based on the preprocessor definitions in the `zigbee.def` file created by ZENA software. You may develop multiple ZigBee protocol node applications using the common set of Stack source files, but individual `zigbee.def` files. For example, each of the demonstration applications has its own `zigbee.def` file (and `myZigBee.c` file) in its respective directory.

This approach allows the development of multiple applications using common source files and generates unique hex files depending on application-specific options. This approach requires that when compiling an application project, you provide search paths to include files from the application, `Common`, and `ZigBeeStack` source directories. The demo application projects supplied with this application note include the necessary search path information.

Note: When working with multiple projects, take care when switching between projects. If the projects' `Intermediates` directories have not been altered, the object files for the Microchip Stack for the ZigBee protocol will be stored in the `ZigBeeStack` directory. These files may not be considered "out of date" when performing a project "Make", and erroneous capabilities may be linked in. Symptoms of this problem include unusual, unhandled primitives being returned to the application layer. To ensure that the Stack files have been compiled correctly for the current project, store the object files in a project unique directory by selecting ***Project>Build Options>Project*** from the main menu. Change the `Intermediates` directory to a unique directory for the project. The demo application projects supplied with this application note already specify unique `Intermediates` directories.

DEMO APPLICATIONS

Version 1.0-3.8 of the Microchip Stack includes three primary demonstration applications:

- `DemoCoordinator` – Demonstrates a typical ZigBee protocol coordinator device application.
- `DemoRFD` – Demonstrates a typical ZigBee protocol RFD device application.
- `DemoRouter` – Demonstrates a typical ZigBee protocol router device application.

Demo Application Features

The demo applications implement the following features:

- Targeted for use with the PICDEM Z demo board.
- Demonstrates low-power functionality using system Sleep and Watchdog Timer functionality.
- RS-232 terminal output to view device operation.
- Operates as a simple remote control switch and LED (“light”) application on one node.
- Configurable at compile time to demonstrate either binding or device discovery (direct or indirect messaging).

One PICDEM Z Demonstration Board must be programmed as a ZigBee protocol coordinator using the `DemoCoordinator` project. A second board must be programmed as an end device using the corresponding RFD project. If more PICDEM Z Demonstration Boards are available, they can be programmed either as more end devices, or as routers, using the `DemoRouter` project. Note that the router project has neither the “switch” nor “light” capability. Instead, its function is to extend the radio range of the network.

Demo Applications Project and Source Files

Table 5 through Table 9 list the source files required to implement the Microchip Stack for the ZigBee protocol and the demo applications. Note that additional files may be provided in the `ZigBeeStack` directory as additional transceivers are supported.

TABLE 5: MICROCHIP STACK SOURCE FILES IN `ZigBeeStack` SUBDIRECTORY

File Name	Description
`SymbolTime.c, .h`	Performs timing functions for the Microchip Stack for the ZigBee™ protocol.
`zAPL.h`	Application level interface header file for the Stack. This is the only file that the application code needs to include.
`zAPS.c, .h`	ZigBee protocol APS layer.
`zHCLighting.h`	ZigBee protocol’s Home Control, Lighting profile information.
`zigbee.h`	Generic ZigBee protocol constants.
`ZigBeeTasks.c, .h`	Directs program flow through the Stack layers.
`zMAC.h`	Generic IEEE 802.15.4™ MAC layer header file.
`zMAC_MRF24J40.c, .h`	IEEE 802.15.4 MAC layer for the Microchip MRF24J40 transceiver.
`zNVM.c, .h`	Performs nonvolatile memory storage functions.
`zNWK.c, .h`	ZigBee protocol NWK layer.
`zPHY.h`	Generic IEEE 802.15.4 PHY layer header file.
`zPHY_MRF24J40.c, .h`	IEEE 802.15.4 PHY layer for the Microchip MRF24J40 transceiver.
`zSecurity.h`	ZigBee protocol security layer header file.
`zSecurity_MRF24J40.c, .h`	ZigBee protocol security layer for the Microchip MRF24J40 transceiver.
`zZDO.c, .h`	ZigBee protocol’s ZDO (ZDP) layer.

TABLE 6: MICROCHIP COMMON SOURCE FILES IN `Common` SUBDIRECTORY

File Name	Description
Compiler.h	Compiler-specific definitions.
Console.c, .h	USART interface code (optional).
Generic.h	Generic constants and type definitions.
MSPI.c, .h	SPI interface code
sralloc.c, .h	Dynamic memory allocation (heap) code.

TABLE 7: ZigBee™ PROTOCOL COORDINATOR DEMO IN `DemoCoordinator` SUBDIRECTORY

File Name	Description
Coordinator Template.c	Template for creating ZigBee protocol coordinator applications.
Coordinator.c	Main application source file.
DemoCoordinator.mcp	Project file.
myZigBee.c	Generated by ZENA™ software. Contains application-specific information.
zigbee.def	Generated by ZENA software. Contains application-specific information.
ZigbeeF.def	Created by ZENA software for PIC18F microcontrollers.
ZigbeeFSecurity.def	Created by ZENA software for PIC18F microcontrollers with security turned on.
ZigBeeJ.def	Created by ZENA software for PIC18FJ microcontrollers.
zLink.lkr	Generated by ZENA software. Project linker script.
zLinkF.lkr	Created by ZENA software. Project linker script for PIC18F microcontrollers.
zLinkFSecurity.lkr	Created by ZENA software. Project linker script for PIC18F microcontrollers with security turned on.
zLinkJ.lkr	Created by ZENA software. Project linker script for PIC18FJ microcontrollers.

TABLE 8: ZigBee™ PROTOCOL ROUTER DEMO IN `DemoRouter` SUBDIRECTORY

File Name	Description
DemoRouter.mcp	Project file.
myZigBee.c	Generated by ZENA™ software. Contains application-specific information.
Router Template.c	Template for creating ZigBee protocol router applications.
Router.c	Main application source file.
zigbee.def	Generated by ZENA software. Contains application-specific information.
ZigbeeF.def	Created by ZENA software for PIC18F microcontrollers.
ZigBeeJ.def	Created by ZENA software for PIC18FJ microcontrollers.
zLink.lkr	Generated by ZENA software. Project linker script.
zLinkF.lkr	Created by ZENA software. Project linker script for PIC18F microcontrollers.
zLinkJ.lkr	Created by ZENA software. Project linker script for PIC18FJ microcontrollers.

TABLE 9: ZigBee™ PROTOCOL END DEVICE DEMO IN `DemoRFD` SUBDIRECTORY

File Name	Description
DemoRFD.mcp	Project file.
FFD End Device Template.c	Template for creating FFD end device applications.
myZigBee.c	Generated by ZENA™ software. Contains application-specific information.
RFD Template with ACKs.c	Template for creating RFD applications when the RFD requests APS level Acknowledges.
RFD Template.c	Template for creating RFD applications when the RFD does not request APS level Acknowledges.
RFD.c	Main application source file.
zigbee.def	Generated by ZENA software. Contains application-specific information.
ZigbeeF.def	Created by ZENA software for PIC18F microcontrollers.
ZigbeeFSecurity.def	Created by ZENA software for PIC18F microcontrollers with security turned on.
ZigBeeJ.def	Created by ZENA software for PIC18FJ microcontrollers.
zLink.lkr	Generated by ZENA software. Project linker script.
zLinkF.lkr	Created by ZENA software. Project linker script for PIC18F microcontrollers.
zLinkFSecurity.lkr	Created by ZENA software. Project linker script for PIC18F microcontrollers with security turned on.
zLinkJ.lkr	Created by ZENA software. Project linker script for PIC18FJ microcontrollers.

Building Primary Demo Applications

The following is a high-level procedure for building demo applications. This procedure assumes that you are familiar with MPLAB® IDE and will be using MPLAB IDE to build the applications. If not, refer to your MPLAB IDE application-specific instructions to create, open and build a project.

1. Make sure that the source files for the Microchip Stack for the ZigBee protocol are installed. If not, please refer to **"Installing Source Files"**.
2. Launch MPLAB IDE and open the appropriate project file: `DemoCoordinator\DemoCoordinator.mcp` for the demo ZigBee protocol coordinator application, `DemoRFD\DemoRFD.mcp` for the demo RFD application or `DemoRouter\DemoRouter.mcp` for the demo ZigBee protocol router application.
3. Use MPLAB IDE menu commands to build the project. Note that the demo application projects are created to work correctly when the source files are located in the `MpZBee` directory in the root directory of the hard drive. If you have moved the source files to another location, you must recreate or modify existing project settings to build.
4. The build process should finish successfully. If not, make sure your MPLAB IDE and compiler are set up properly, and your project options are correct.

Programming Primary Demo Applications

To program a target with either of the two demo applications, you must have access to a Microchip programmer. The following procedure assumes that you will be using MPLAB ICD 2 as a programmer. If not, please refer to your specific programmer instructions.

1. Connect MPLAB ICD 2 to the PICDEM Z demo board or your target board.
2. Apply power to the target board.
3. Launch MPLAB IDE.
4. Select the Microchip device of your choice (required only if you are importing a hex file previously built).
5. Enable MPLAB ICD 2 as a programmer and select the **Connect** option from the MPLAB ICD 2 programmer menu to connect to MPLAB ICD 2 and perform a self-test.
6. If you have just rebuilt the project as described above, proceed to the next step. If you want to use a previously built hex file, import the `DemoCoordinator\DemoCoordinator.hex` file, the `DemoRFD\DemoRFD.hex` file or the `DemoRouter\DemoRouter.hex` file. In order to simplify identification of the demo coordinator and demo RFD nodes (if you are using PICDEM Z boards), it is recommended that you program the `DemoCoordinator.hex` file into the controller with the "COORD..." label, and the `DemoRFD.hex` file into the controller with the "RFD..." label. If you are programming your custom hardware, make sure that you use some identification method to identify the different nodes.
7. All demo application files contain necessary configuration options required for the PICDEM Z demo board. If you are programming another type of board, make sure that you select the appropriate oscillator mode from the MPLAB ICD 2 configuration settings menu.
8. Select the **Program** menu option from the MPLAB ICD 2 programmer menu to begin programming the target.
9. After a few seconds, you should see the message, "`Programming successful`". If not, double check your board and MPLAB ICD 2 connection. Refer to MPLAB IDE on-line help for further assistance.
10. Remove power from the board and disconnect the MPLAB ICD 2 cable from the target board.

Running the Primary Demo Applications

Before trying to run the demo, ensure that both nodes are configured to demonstrate the same capability. The nodes can demonstrate end device binding by uncommenting the `#define USE_BINDINGS` definition in each main demo source file. The nodes can demonstrate device discovery by commenting out the `#define USE_BINDINGS` definition in each source file. Ensure that both nodes are configured the same way.

Note:	Each device may be configured to act as a "light" or a "switch" through the use of the `I_AM_LIGHT` and `I_AM_SWITCH` definitions. By default, each of these capabilities is enabled in both nodes.

To run the demo, program one PICDEM Z demo board as a ZigBee protocol coordinator, and the other as an RFD, using the demo applications provided. To view node operation, it is recommended that you connect the RS-232 connector on each demo board to a serial port on a PC, and use HyperTerminal or another serial interface software to communicate with the PICDEM Z demo board. Configure the port with the following settings: 19200 bps, 8 data bytes, 1 Stop bit, no parity and no flow control.

Apply power to the coordinator node. You should see the following message on the HyperTerminal window:

```
************************************
Microchip ZigBee(TM) Stack - v1.0-3.8
ZigBee Coordinator
```

The coordinator will then automatically try to find an available wireless channel and form a new network. If successful, it will display the following message:

```
Trying to start network...
PAN #### started successfully.
```

Where #### is a four-digit hexadecimal number, indicates the PAN ID of the network it has successfully formed. It will then enable joining of the network by other nodes and display the following message:

```
Joining permitted.
```

At this point, other nodes may join the network.

Apply power to the RFD node. You should see the following message on the corresponding HyperTerminal window:

```
************************************
Microchip ZigBee(TM) Stack - v1.0-3.8
ZigBee RFD
```

The RFD will then try to find a network to join. If it is successful, it will display the following message:

```
Trying to join network as a new device...
Network(s) found. Trying to join ####.
Join successful!
```

The coordinator will recognize that the new node has joined by displaying the following message:

```
Node #### just joined.
```

Where #### is the assigned short address of the new node.

Since the RFD conserves power by turning off its transceiver and putting the microcontroller into Sleep mode, all incoming messages are buffered by its parent node. When the RFD wakes up, it must request messages from its parent. If its parent has messages, the parent will send them; otherwise, the RFD is free to go back to Sleep. This operation is displayed on the RFD's HyperTerminal window:

```
Requesting data...
No data available.
```

At this point, the RFD has successfully joined the network and is polling for messages. Further operation depends on the configuration of the nodes.

Demonstrating End Device Binding

If both nodes have `#define USE_BINDINGS` uncommented, they will demonstrate end device binding. In this configuration, the "switch" nodes send their messages to one or more "light" nodes through the use of bindings and indirect messages. Refer to **"Message Types and Binding"** for a more detailed description of bindings.

Before a "switch" can send an indirect message to a "light", a binding must be created. The RB5 button on the PICDEM Z is used to send the end device bind request to the ZigBee protocol coordinator. Press the RB5 button on the coordinator node. The following message will be displayed:

```
Trying to perform end device binding.
```

Then press the RB5 button on the RFD node. If the button is not pressed within approximately 5 seconds, the end device bind request on the coordinator will time out and the process must be repeated. If successful, the following message will be displayed on the RFD's terminal window:

```
Trying to send END_DEVICE_BIND_req.
End device bind/unbind successful!
```

The following message will then appear on the ZigBee protocol coordinator's terminal window:

```
End device bind/unbind successful!
```

The "switch" can now send indirect messages to the "light".

Note 1: If both nodes are configured to operate as both a "switch" and a "light", two bindings are actually created with the single end device bind request and each node can send messages to the other.

2: The end device bind process is a toggle function; if the process is repeated, the binding will be removed. The status returned by the end device bind does not indicate if the binding was created or removed, only that the process was successful.

Demonstrating Device Discovery

If both nodes have `#define USE_BINDINGS` commented out, they will demonstrate device discovery. In this configuration, the “switch” nodes send their messages to one “light” node through the use of direct messages to a known network address. Refer to **“Message Types and Binding”** for a more detailed description of message types.

Before a “switch” can send a direct message to a “light”, the “switch” must determine the network address of the “light”. To simplify the demonstration, we assume that the “switch” knows the MAC address of the “light” it wishes to talk to. The RB5 button on the PICDEM Z is used to broadcast the network address request message. Press the RB5 button on the ZigBee protocol coordinator node. The following message will be displayed:

```
Trying to send NWK_ADDR_req.
```

The lower layers of the RFD will see and respond to this message without any support from the application layer. When the coordinator receives the RFD’s response, it will display the following message:

```
Receiving NWK_ADDR_rsp.
```

The coordinator can now act as a “switch” and send messages to the “light”. If both nodes are configured to operate as both a “switch” and a “light”, press RB5 on the RFD PICDEM Z demo board. That node will then get the network address of the ZigBee protocol coordinator. Each node can then send messages to the other.

Demonstrating Messages

After either the bindings have been created, or the network addresses have been discovered, the “switch” node can send messages to the “light” node to toggle its LED. Press RB4 on one of the PICDEM Z demo boards that has been configured as a “switch”. It will display the following message:

```
Trying to send light switch message.
```

If direct messages are used, the application will be notified when the Stack receives a MAC Acknowledge for the transmission. If indirect messages are used, the application will be notified when the Stack buffers the message for later transmission. When the application receives this notification, it will display the following message:

```
Message sent successfully.
```

When the “light” node receives the transmission, it will display the following message:

```
Toggling light.
```

The “light” node will then toggle the state of the RA1 LED.

USING THE MICROCHIP STACK FOR THE ZigBee PROTOCOL

To design a ZigBee protocol system, you must do the following:

1. Obtain an OUI (see **“IEEE Extended Unique Identifiers – EUI-64”**).
2. Determine the radio needed based on data rate and geographical market needs.
3. Select a suitable Microchip MCU.
4. Develop the ZigBee protocol application using the Stack provided with *AN965, “Microchip Stack for the ZigBee™ Protocol”*.
5. Perform all RF compliance certifications.
6. Perform ZigBee protocol interoperability compliance certification.

Follow these basic steps to develop a ZigBee protocol application:

1. Determine the profile that the system will use.
2. Determine the endpoint structure that each device will use.
3. Create a new project directory. Place all application-specific source files and project files in this directory.
4. Use ZENA software to generate configuration files based on the device type, device configuration and endpoint structure.
5. Obtain the appropriate template file from one of the demo directories and use it as the basis for the application code.
6. Add code to the template as directed in the template, including extra initialization, any required ZDO response handling, endpoint message reception and transmission, and any non-protocol processing and interrupt handling.

Interfacing with the Microchip Stack for the ZigBee Protocol

The application source code must include the header file, zAPL.h , to access the ZigBee protocol functions.

```
#include "zAPL.h"
```

A ZigBee protocol coordinator application will need to have one support variable to keep track of the current primitive being executed by the Stack.

```
ZIGBEE_PRIMITIVE currentPrimitive;
```

A ZigBee protocol router or end device will also need to keep track of the current primitive; but in addition, it will need two other support variables to assist in network discovery and joining.

```
NETWORK_DESCRIPTOR * currentNetworkDescriptor;
ZIGBEE_PRIMITIVE currentPrimitive;
NETWORK_DESCRIPTOR * NetworkDescriptor;
```

Next, the application must configure all pins required to interface with the transceiver. The ZENA analyzer will create several labels that may be used to access the required LAT and TRIS bits. Refer to the Readme file for the labels created for the supported transceivers.

Before the Stack can be used, it must be initialized. Interrupts must then be enabled:

```
ZigBeeInit();
RCONbits.IPEN = 1;
INTCONbits.GIEH = 1;
```

The application now interfaces with the Stack through the primitives defined in the ZigBee protocol and IEEE 802.15.4 specifications. Stack operation is triggered by calling the function, ZigBeeTasks(). Stack operation will continue until the requested primitive path is complete or an application-level primitive needs to be processed.

Note:	Refer to the ZigBee protocol and IEEE 802.15.4 specifications for the complete list of primitives and their parameters.

Since only one primitive can be processed at one time, a single data structure (a union) is used to hold all the primitive parameters. This structure can be viewed in the file, ZigBeeTasks.h. Take care when accessing this structure that you do not overwrite a parameter before using it. After processing a primitive, it is critical that the current primitive be set to the next primitive to execute (or NO_PRIMITIVE) to avoid an infinite loop (see Example 1). Refer to the **"Primitive Summary"** section for a list of the common primitives used by the application layer.

Default processing for most primitives is included in the template files. Two primitives will require additional application-specific code: APSDE_DATA_indication and NO_PRIMITIVE.

EXAMPLE 1: THE BASIC STRUCTURE OF THE APPLICATION

```
while (1)
{
    CLRWDT();
    ZigBeeTasks( &currentPrimitive );

    switch (currentPrimitive)
    {
        // Include cases for each required primitive.
        // Be sure to update currentPrimitive!

        default:
            currentPrimitive = NO_PRIMITIVE;
            break;
    }
}
```

Forming or Joining a Network

The process of forming or joining a network is included in the template files. The process is initiated in the NO_PRIMITIVE primitive handling. If the device is a ZigBee protocol coordinator, and if it has not formed a network, then it will begin the process of trying to form a network by issuing the NLME_NETWORK_FORMATION_request primitive.

If the device is not a ZigBee protocol coordinator and it is not currently on a network, it will try to join one. If the device has determined that it was previously on a network, then it will try to join as an orphan by issuing the NLME_JOIN_request with the RejoinNetwork parameter set to TRUE. If that fails, or if the device was not previously on a network, then it will try to join as a new node. It will first issue the NLME_NETWORK_DISCOVERY_request primitive to discover what networks are available. The application code will then select one of the discovered networks and try to join it by issuing the NLME_JOIN_request with the RejoinNetwork parameter set to FALSE. See **"ZigBee Protocol Timing"** for timing requirements used during this process.

Receiving Messages

The Stack notifies the application of received messages through the APSDE_DATA_indication primitive. When this primitive is returned, the APSDE_DATA_indication primitive parameters are populated with information about the message and the received message resides in a buffer. Use the function, APLGet(), to extract each byte of the message from the buffer.

The DstEndpoint parameter indicates the destination endpoint for the message. If it is a valid endpoint, the message can be processed (see Example 2).

Note 1: A case for the ZDO endpoint (endpoint 0) must be included to handle responses to all ZDO messages sent by the application.

2: After the message is processed, it must be discarded using the APLDiscard() function. Failure to discard the message will result in no further messages being processed.

EXAMPLE 2: RECEIVING MESSAGES

```
case APSDE_DATA_indication:
{
        // Declare variables used by this primitive.

        currentPrimitive = NO_PRIMITIVE;  // This may change during processing.
        frameHeader = APLGet();

        switch (params.APSDE_DATA_indication.DstEndpoint)
        {
            case EP_ZDO:
                // Handle all ZDO responses to requests we sent.
                break;

               // Include cases for all application endpoints.
        }
        APLDiscard();
    }
    break;
```

Sending Messages

The Microchip Stack for the ZigBee protocol allows one outgoing message in the application layer at a time. Messages are sent by implementing the following:

1. Verify that the application layer is ready for a new outgoing message by confirming that `ZigBeeReady()` is `TRUE`.
2. Lock the system with `ZigBeeBlockTx()` so subsequent calls to `ZigBeeReady()` will return `FALSE`.
3. Load the message payload into the array `TxBuffer`, using `TxData` to index through the array. When complete, `TxData` must point to the first location after the message (i.e., `TxData` equals the length of the data).
4. Load the `APSDE_DATA_request` primitive parameters.
5. Set `currentPrimitive` to `APSDE_DATA_request` and call `ZigBeeTasks()`.

Messages are typically sent by the application in two places:

- In `APSDE_DATA_indication` processing, in response to a received message.
- In `NO_PRIMITIVE` processing, in response to an application event.

The process of sending a message is identical for both locations. Example 3 shows how to send a direct message to a known destination device and endpoint to toggle a light. The following should be noted:

- Each APS frame must have a unique Transaction Identifier. This is obtained from the Stack by calling `APLGetTransID()`.
- `TxData` must point to the next available location, so `TxBuffer` is loaded using post-increment addressing.
- The destination endpoint has already been determined and is stored in `destinationEndpoint`.
- The 16-bit network address of the destination device has already been determined and is stored in `destinationAddress`.
- We are requesting that the message be routed, if possible.

The status of the transmitted message will be returned via the `APSDE_DATA_confirm` primitive. Note that if the message fails to transmit, the Stack will automatically handle retrying the message, *apscMaxFrameRetries* times.

EXAMPLE 3: SENDING AN OUTGOING MESSAGE

```
if (ZigBeeReady())
{
    if (bLightSwitchToggled)
    {
        bLightSwitchToggled = FALSE;
        ZigBeeBlockTx();

        TxBuffer[TxData++] = APL_FRAME_TYPE_KVP | 1;          // KVP, 1 transaction
        TxBuffer[TxData++] = APLGetTransId();
        TxBuffer[TxData++] = APL_FRAME_COMMAND_SET | (APL_FRAME_DATA_TYPE_UINT8<< 4);
        TxBuffer[TxData++] = OnOffSRC_OnOff & 0xFF;           // Attribute ID LSB
        TxBuffer[TxData++] = (OnOffSRC_OnOff >> 8) & 0xFF;    // Attribute ID MSB
        TxBuffer[TxData++] = LIGHT_TOGGLE;

        params.APSDE_DATA_request.DstAddrMode = APS_ADDRESS_16_BIT;
        params.APSDE_DATA_request.DstEndpoint = destinationEndpoint;
        params.APSDE_DATA_request.DstAddress.ShortAddr = destinationAddress;

        params.APSDE_DATA_request.ProfileId.Val = MY_PROFILE_ID;
        params.APSDE_DATA_request.RadiusCounter = DEFAULT_RADIUS;
        params.APSDE_DATA_request.DiscoverRoute = ROUTE_DISCOVERY_ENABLE;
        params.APSDE_DATA_request.TxOptions.Val = 0;
        params.APSDE_DATA_request.SrcEndpoint = EP_SWITCH;
        params.APSDE_DATA_request.ClusterId = OnOffSRC_CLUSTER;

        currentPrimitive = APSDE_DATA_request;
    }
}
```

Requesting and Receiving Data on an RFD

Since RFDs normally power off their transceiver when they are Idle to conserve power, they must request messages when their transceiver is on by issuing the NLME_SYNC_request primitive. Example 4 demonstrates a typical sequence for going to Sleep, and waking back up using the Watchdog Timer, or a button press to wake-up. We can Sleep if all of the following are true:

- There is no ZigBee protocol primitive ready to be processed.
- The Stack is not performing background tasks.
- The previous data request is complete.
- All application-specific processes are complete.

After waking from Sleep, the RFD must request data from its parent using the NLME_SYNC_request primitive. The RFD will receive one of the following responses from issuing an NLME_SYNC_request:

- If the RFD's parent has messages buffered for the device, it will send one, and the RFD will generate an APSDE_DATA_indication primitive.
- If the parent device does not have any buffered messages for the RFD, the RFD will generate an NLME_SYNC_confirm primitive with a status of SUCCESS.

If the RFD receives no response from its parent, the RFD will generate an NLME_SYNC_confirm primitive with a status of NWK_SYNC_FAILURE.

EXAMPLE 4: REQUESTING AND RECEIVING DATA ON AN RFD

```
// If we don't have to execute a primitive, see if we need to request
// data from our parent, or if we can go to sleep.
if (currentPrimitive == NO_PRIMITIVE)
{
    if (!ZigBeeStatus.flags.bits.bDataRequestComplete)
    {
        // We have not received all data from our parent. If we are not waiting
        // for an answer from a data request, send a data request.
        if (!ZigBeeStatus.flags.bits.bRequestingData)
        {
            if (ZigBeeReady())
            {
                // Our parent still may have data for us.
                params.NLME_SYNC_request.Track = FALSE;
                currentPrimitive = NLME_SYNC_request;
            }
        }
    }
    else
    {
        if (!ZigBeeStatus.flags.bits.bHasBackgroundTasks &&
             myProcessesAreDone())
        {
            // We do not have a primitive to execute, we've extracted all messages
            // that our parent has for us, the stack has no background tasks,
            // and all application-specific processes are complete.
            // Now we can go to sleep. Make sure that the UART is finished,
            // turn off the transceiver, and make sure that we wakeup from key press.
            while (!ConsoleIsPutReady());
            APLDisable();
            INTCONbits.RBIE = 1;
            SLEEP();
            NOP();

            // We just woke up from sleep. Turn on the transceiver and
            // request data from our parent.
            APLEnable();
            params.NLME_SYNC_request.Track = FALSE;
            currentPrimitive = NLME_SYNC_request;
        }
    }
}
```

Secure Transmission

The Microchip Stack for the ZigBee Protocol supports all seven security modes that are defined in ZigBee protocol specification to protect the output packets.

The security modes can be categorized into three groups:

- Message Integrity Code (MIC) Security modes ensure the integrity of the packet. The MIC attached to the packet (the size of which is determined by the particular mode) ensures that the packet, including the header and payload, has not been modified in anyway during transmission. The packet payload is not encrypted in these modes.
- Encryption (ENC) Security mode encrypts the payload. The plaintext content of the payload cannot be exposed without a valid security key. This mode cannot verify frame integrity or the content of the header, including the source of the original packet and the frame counter.
- ENC-MIC Security modes are a combination of the two previous groups. In these modes, the payload is encrypted. At the same time, the header and payload's integrity is protected by the MIC attached at the end of the packet.

In addition, there is also Security mode, 0x00, which specifies no security. Essentially, this is the Stack operating with the security module turned off. The capability of each of the security modes can be found in Table 10.

The ZigBee protocol specification also defines support for Residential and Commercial Security modes, based on the use of security keys. The main difference between the two is that Commercial mode requires the generation of an individual security key between two nodes while communicating, while Residential mode uses the unique network key within the network to secure packets. Currently, the Microchip Stack for the ZigBee Protocol supports only Residential mode.

The Stack supports networks with or without a preconfigured security key. Security is supported in either the NWK or the APL layer, depending on the requirements of the application profile. MAC layer security support can also be enabled.

The Stack adds an auxiliary security header before the security payload of every secured packet. The format of the auxiliary security header format can be found in Table 11.

The ZigBee security protocol specifies the nonce to be the combination of three items:

- the frame counter
- the source long address
- the key sequence number (for MAC layer) or the security control byte (for NWK and APL layers)

As the result, if MAC layer security is turned on, the source address mode in the MAC layer must be Extended Address mode (0x03). If APL layer security is turned on, the device that decrypts the packet must be able to match the packet source short address to its source long address. This is done using the APS address map table.

TABLE 10: ZigBee™ SECURITY SERVICES

Security Mode		Security Service			MIC Length (Bytes)
Identifier	Name	Access Control	Data Encryption	Frame Integrity	
0x01	MIC-32	X		X	4
0x02	MIC-64	X		X	8
0x03	MIC-128	X		X	16
0x04	ENC	X	X		0
0x05	ENC-MIC-32	X	X	X	4
0x06	ENC-MIC-64	X	X	X	8
0x07	ENC-MIC-128	X	X	X	16

TABLE 11: ZigBee™ AUXILIARY SECURITY HEADER FORMAT

Security Location	Packet Header Feature			
	Security Control (1 Byte)	Frame Counter (4 Bytes)	Source Extended Address (8 Bytes)	Key Sequence Number (1 Byte)
MAC Layer Security		X		X
NWK Layer Security	X	X	X	X
APL Layer Security	X	X		X

The Stack is capable of ensuring sequential freshness by checking the transmitted frame counter. Only the frame counter of packets from family members (parent or children) will be checked, since only family member knows when a device joins the network. Packets that are from family members but do not meet the sequential freshness requirement will be discarded.

The maximum length of a transmitted message is 127 bytes. When the security module is turned on, between 5 and 29 additional bytes are required for the auxiliary security header and the MIC, depending on the combination of security mode and secured layer. Users will need to balance the security needs and the impact on the data payload size (and associated performance impact) associated with the combination of security settings.

The security mode and secured layer settings are defined in the application profile. Use the ZENA Wireless Network Analyzer configuration tool to set up all other critical security options.

Once the security mode has been defined, actually sending the secured packet is straightforward; only one modification is required in the application code. Example 4 shows the exact same code as in Example 3, with the additional code to enable secure transmission shown in **bold**.

EXAMPLE 5: SENDING A SECURED OUTGOING MESSAGE

```
if (ZigBeeReady() )
{
    if (bLightSwitchToggled)
    {
        bLightSwitchToggled = FALSE;
        ZigBeeBlockTx();

        TxBuffer[TxData++] = APL_FRAME_TYPE_KVP | 1;
        TxBuffer[TxData++] = APLGetTransId();
        TxBuffer[TxData++] = APL_FRAME_COMMAND_SET | (APL_FRAME_DATA_TYPE_UINT8 << 4);
        TxBuffer[TxData++] = OnOffSRC_OnOff & 0xFF;
        TxBuffer[TxData++] = (OnOffSRC_OnOff >> 8) & 0xFF;
        TxBuffer[TxData++] = LIGHT_TOGGLE;

        params.APSDE_DATA_request.DstAddrMode = APS_ADDRESS_16_BIT;
        params.APSDE_DATA_request.DstEndpoint = destinationEndpoint;
        params.APSDE_DATA_request.DstAddress.ShortAddr = destinationAddress;

        params.APSDE_DATA_request.ProfileId.Val = MY_PROFILE_ID;
        params.APSDE_DATA_request.RadiusCounter = DEFAULT_RADIUS;
        params.APSDE_DATA_request.DiscoverRoute = ROUTE_DISCOVERY_ENABLE;
        params.APSDE_DATA_request.TxOptions.Val = 0;
        params.APSDE_DATA_request.TxOptions.bits.securityEnabled = 1;
        params.APSDE_DATA_request.SrcEndpoint = EP_SWITCH;
        params.APSDE_DATA_request.ClusterId = OnOffSRC_CLUSTER;

        currentPrimitive = APSDE_DATA_request;
    }
}
```

Primitive Summary

The application layer communicates with the Stack primarily through the primitives defined in the ZigBee protocol and IEEE 802.15.4 specifications. Table 12 describes the primitives that are commonly issued by the application layer and their response primitive. Not all devices will issue all of these primitives.

Some primitives that are received by the application layer are generated by the Stack itself, not as a response to an application primitive. The application layer must be able to handle these primitives as well. Table 13 shows all the primitives that can be returned to the application layer. Default processing for most of the primitives is included in the application templates.

TABLE 12: TYPICAL APPLICATION PRIMITIVES AND RESPONSES

Application Issued Primitive	Response Primitive	Description
APSDE_DATA_request	APSDE_DATA_confirm	Used to send messages to other devices.
APSME_BIND_request	APSME_BIND_confirm	Force the creating of a binding. Can be used only on devices that support binding.
APSME_UNBIND_request	APSME_UNBIND_confirm	Force the removal of a binding. Can be used only on devices that support binding.
NLME_NETWORK_DISCOVERY_ request	NLME_NETWORK_DISCOVERY_ confirm	Discover networks available for joining. Not used by ZigBee™ protocol coordinators.
NLME_NETWORK_FORMATION_ request	NLME_NETWORK_FORMATION_ confirm	Start a network on one of the specified channels. ZigBee protocol coordinators only.
NLME_PERMIT_JOINING_ request	NLME_PERMIT_JOINING_ confirm	Allow other nodes to join the network as our children. ZigBee protocol coordinators and routers only.
NLME_START_ROUTER_ request	NLME_START_ROUTER_ confirm	Start routing functionality. Routers and FFD end devices only.
NLME_JOIN_request	NLME_JOIN_confirm	Try to rejoin or join the specified network. Not used by ZigBee protocol coordinators.
NLME_DIRECT_JOIN_ request	NLME_DIRECT_JOIN_ confirm	Add a device as a child device. ZigBee protocol coordinators and routers only.
NLME_LEAVE_request	NLME_LEAVE_confirm	Leave the network or force a child device to leave the network.
NLME_SYNC_request	NLME_SYNC_confirm	Request buffered messages from the device's parent. RFDs only.
ZDO_END_DEVICE_BIND_req	APSDE_DATA_indication	Can be used only on devices that support binding.

TABLE 13: PRIMITIVE HANDLING REQUIREMENTS

Primitive	ZigBee™ Protocol Coordinator	ZigBee™ Protocol Router	FFD End Device	RFD End Device
APSDE_DATA_confirm	X	X	X	X
APSDE_DATA_indication	X	X	X	X
APSME_BIND_confirm	X(6)	X(4,6)		
APSME_UNBIND_confirm	X(6)	X(4,6)		
NLME_DIRECT_JOIN_confirm	X(6)	X(5)		
NLME_GET_confirm	**(Note 2)**	**(Note 2)**	**(Note 2)**	**(Note 2)**
NLME_JOIN_confirm		X	X	X
NLME_JOIN_indication	X	X		
NLME_LEAVE_confirm	X(1)	X(1)	X(1)	X(1)
NLME_LEAVE_indication	X	X	X	X
NLME_NETWORK_DISCOVERY_confirm		X	X	X
NLME_NETWORK_FORMATION_confirm	X			
NLME_PERMIT_JOINING_confirm	X	X		
NLME_RESET_confirm		X	X	X
NLME_SET_confirm	**(Note 2)**	**(Note 2)**	**(Note 2)**	**(Note 2)**
NLME_START_ROUTER_confirm		X	X	
NLME_SYNC_confirm				X
NLME_SYNC_indication				**(Note 3)**
NO_PRIMITIVE	X	X	X	X

Note 1: Required if application will issue an NLME_LEAVE_request to another node.
2: These primitives are not used. Stack attribute manipulation is done directly.
3: Not used by non-beacon networks.
4: Required if binding is supported.
5: Required if application will issue an NLME_DIRECT_JOIN_request.
6: Required if application issues the corresponding BIND/UNBIND_request.

SYSTEM RESOURCE CLEAN-UP

It is required that all unnecessary system resources are cleaned up after invoking a primitive. The Microchip ZigBee protocol Stack already handles most of the cleaning up in the Stack. Currently, there is only one primitive, NLME_JOIN_confirm, which is handled by the application layer and needs to be cleaned up by the user.

ZigBee devices other than the Coordinator usually invoke NLME_NETWORK_DISCOVERY_request to find out the current available networks before deciding which network to join. The primitive, NLME_NETWORK_DISCOVER_confirm, returns a link list of the all available networks for the user to choose from. Upon finished joining the network, the link list of available networks must be removed to free the system resources when receiving primitive NLME_JOIN_confirm. Example 6 shows how to free the available network list in the primitive NLME_JOIN_confirm.

Keep in mind that this procedure has been implemented in the Microchip ZigBee protocol demo projects as well as in the application template.

EXAMPLE 6: CLEANING UP SYSTEM RESOURCES

```
while (NetworkDescriptor)
{
        currentNetworkDescriptor = NetworkDescriptor->next;
        free( NetworkDescriptor );
        NetworkDescriptor = currentNetworkDescriptor;
}
```

Microchip Stack for the ZigBee Protocol Macros and Functions

APLDisable

Description

This function disables the transceiver.

Syntax

```
BOOL APLDisable( void );
```

Inputs

None

Outputs

`TRUE` – If the transceiver was put into Reset.

`FALSE` – If the current Stack activity prohibits putting the transceiver into Reset.

Notes

Typically, this function is used only by RFDs to conserve power while in Sleep.

APLDiscard

Description

This function discards the current received message. It should be called when processing of the current message is complete.

Syntax

```
void APLDiscard( void );
```

Inputs

None

Outputs

None

Notes

Failure to call this function will result in the Stack being unable to process, and ultimately, receive messages. Refer to the template files for typical usage of this function.

APLEnable

Description

This function enables the transceiver.

Syntax

```
void APLEnable(void );
```

Inputs

None

Outputs

None

Notes

Typically, it is used only by RFDs when they wake-up from Sleep. It is not necessary to call this function in any other location.

APLGet

Description

This function retrieves a byte from the current received message.

Syntax

```
BYTE APLGet( void );
```

Inputs

None

Outputs

The next byte of the current received message.

Notes

If this function is called after all message bytes have been retrieved, this function will return 0x00.

APLGetTransId

Description

This function retrieves the next APS Transaction Identification value to use in an outgoing message.

Syntax

```
BYTE APLGetTransId( void );
```

Inputs

None

Outputs

The next Transaction ID value.

Notes

None

ZigBeeBlockTx

Description

This function locks the transmit buffer.

Syntax

```
void ZigBeeBlockTx( void );
```

Inputs

None

Outputs

None

Notes

After calling `ZigBeeReady()` to confirm that the transmit buffer (`TxBuffer`) is ready for use, this function should be called so that subsequent calls to `ZigBeeReady()` will return `FALSE`.

ZigBeeInit

Description

This function initializes the Stack. It must be called before any other Stack functions. All hardware pin configuration and directioning must be performed before this function is called.

Syntax

```
void ZigBeeInit( void );
```

Inputs

None

Outputs

None

Notes

None

ZigBeeReady

Description

This function indicates whether or not the Stack is ready to initiate an outgoing message.

Syntax

```
BOOL ZigBeeReady( void );
```

Inputs

None

Outputs

`TRUE` – A new outgoing message may be loaded into `TxBuffer`.

`FALSE` – An earlier message is still being processed and `TxBuffer` is still in use.

Notes

None

ZigBeeTasks

Description

This function triggers Stack operation. The current primitive to execute must be passed in `*primitive`. If no primitive is required to execute, set `*primitive` to `NO_PRIMITIVE`. The function will continue until a user primitive is generated (including `NO_PRIMITIVE`). On exit, it will return `TRUE` if the Stack still has background tasks to execute. This function must be called on a regular basis in order for the Stack to function properly, even if it returns `FALSE`. Message reception from the transceiver is triggered by an interrupt which requests a background task.

Syntax

```
BOOL ZigBeeTasks( ZIGBEE_PRIMITIVE *primitive );
```

Inputs

`*primitive` – Pointer to the value of the next primitive to execute.

Outputs

`TRUE` – The Stack still has background tasks to execute.

`FALSE` – The Stack does not have background tasks to execute.

`*primitive` – Pointer to the value of the next primitive to execute.

Notes

None

Microchip Stack for the ZigBee Protocol Status Flags

The Stack has several status flags that may be viewed by the application. The application must not modify these files or Stack operation will be corrupted. All flags are located in the `ZigBeeStatus.flags.bits` structure.

TABLE 14: STACK STATUS FLAGS

Flag	Description
`bTxFIFOInUse`	Indicates that the Stack is currently in the process of transmitting an outgoing message. Use the macros, `ZigBeeReady()` to check, and `ZigBeeBlockTx()` to set, this flag.
`bRxOverflow`	Indicates that the receive buffer has overflowed and messages have been dropped. Must be cleared by the application.
`bHasBackgroundTasks`	Updated by `ZigBeeTasks()`. Indicates if the Stack still has background tasks in progress.
`bNetworkFormed`	ZigBee™ protocol coordinator only. Indicates that the device has successfully formed a network.
`bTryingToFormNetwork`	ZigBee protocol coordinator only. Indicates that the device is in the process of trying to form a network.
`bNetworkJoined`	ZigBee protocol routers and end devices. Indicates that the device has successfully joined a network.
`bTryingToJoinNetwork`	ZigBee protocol routers and end devices. Indicates that the device is in the process of trying to join a network.
`bTryOrphanJoin`	ZigBee protocol routers and end devices. Indicates that the device was once part of a network and should try to join as an orphan.
`bRequestingData`	RFD end devices only. Indicates that the device is in the process of requesting data from its parent.
`bDataRequestComplete`	RFD end devices only. Indicates that the current request for data is complete and the device may be able to go to Sleep.

Configuration Parameters

The Microchip Stack for the ZigBee protocol is highly configurable using the ZENA Wireless Network Analyzer configuration tool. Most of the configuration items are straightforward, such as the MAC address of the device. The following items are used to configure the size and performance of the Stack itself. Depending on the selected device type, not all of these options will be available.

MAX FRAMES FROM APL LAYER

Every message sent down from the APL layer using the `APSDE_DATA_request` primitive must be buffered so it can be retransmitted on failure. Additional information must also be stored so the message confirmation can be sent back to the APL layer via the `APSDE_DATA_confirm` primitive. The Stack requires 2 bytes of RAM for each frame. Additional heap space will also be allocated when a message is sent down.

MAX APS ACK FRAMES GENERATED

If the application receives messages requesting APS level Acknowledgement, the Stack will automatically generate and send the Acknowledge.

Like the APL layer frames, these must be buffered for transmission in case of failure. Enter the number of APS level Acknowledge frames that may be buffered concurrently. The Stack requires two bytes of RAM for each frame. Additional heap space will also be allocated when a frame is generated.

MAX APS ADDRESSES

Although all normal messaging between nodes is done using 16-bit network addresses, the ZigBee protocol specification allows the `APSDE_DATA_request` primitive to be invoked with a 64-bit MAC address as the message destination. If so, the APS layer searches an APS address map for the 16-bit address of the specified node. This table is stored in nonvolatile memory and must be maintained by the application. Use of this table is optional. If this value is set to '0', the table is not created; no code is created to search the table and `APSDE_DATA_request` calls with 64-bit addressing will fail. If this value is not set to '0', the Stack requires 10 bytes of nonvolatile memory for each entry, plus 2 bytes of RAM.

MAX BUFFERED INDIRECT MESSAGES

If a device supports bindings (ZigBee protocol coordinators, and optionally, ZigBee protocol routers), then it must buffer all received indirect transmissions so they can be forwarded to one or more destinations. The Stack requires 2 bytes of RAM for each message specified. Additional heap space will also be allocated when an indirect message is received.

BINDING TABLE SIZE

If a device supports bindings (ZigBee protocol coordinators, and optionally, ZigBee protocol routers), then it must possess a binding table. The Stack requires 5 bytes of nonvolatile memory for each binding table entry. Note that minimum binding table size is dictated by the Stack profile.

NEIGHBOR TABLE SIZE

All devices keep track of other nodes on the network by using a neighbor table. End devices require a neighbor table to record potential parents. ZigBee protocol coordinators require a neighbor table to record children. ZigBee protocol routers require a neighbor table for both functions. The Stack requires 15 bytes of nonvolatile memory for each neighbor table entry. Note that minimum neighbor table size is dictated by the Stack profile.

MAX BUFFERED BROADCAST MESSAGES

When FFDs generate or receive a broadcast message, they must buffer the message while they check for passive Acknowledges in case they must rebroadcast the message. The Stack may be configured as to how many broadcast messages may be buffered in the system at one time. It is recommended that this value be at least two, since a typical discovery sequence is a broadcast `NWK_ADDR_req`, followed soon by a broadcast route request. The system requires 2 bytes of RAM for each buffered broadcast message specified. Additional heap space will also be allocated when a broadcast message is received or generated.

ROUTE DISCOVERY TABLE SIZE

The ZigBee protocol specification requires that FFDs use a route discovery table during the route discovery process. Since these entries are required for only a short time, they are stored in heap memory. The system requires 2 bytes of RAM for each table entry specified. Additional heap space will also be allocated when route discovery is underway. Note that the minimum route discovery table size is dictated by the Stack profile.

ROUTING TABLE SIZE

The ZigBee protocol specification requires that FFDs maintain a routing table to route messages to other nodes in the network. The system requires 5 bytes of nonvolatile memory for each entry specified. Note that the minimum routing table size is dictated by the Stack profile.

RESERVED ROUTING TABLE ENTRIES

The ZigBee protocol specification requires that FFDs reserve a portion of the routing table for use during route repair. Note that the minimum reserved table entries are dictated by the Stack profile.

MAX BUFFERED ROUTING MESSAGES

If an FFD receives a message that needs to be routed, and the FFD does not have a route for the required destination, it must buffer the received message and perform route discovery (if possible) for the required destination. The system requires 10 bytes of RAM for each buffered message specified. Additional heap space will also be allocated when a message is received.

CHANNEL ENERGY THRESHOLD

When a ZigBee protocol coordinator selects a channel for a new network, it first scans all of the available channels and eliminates those whose channel energy exceeds a specified limit.

MINIMUM JOIN LQI

When a ZigBee protocol router or end device joins a new network, it examines the link quality of the beacon it received from each possible parent. If the link quality is below this specified minimum, the device will eliminate that device as a potential parent.

TRANSACTION PERSISTENCE

ZigBee protocol coordinators and routers are required to buffer messages for their children whose transceivers are off when they are Idle. This parameter is the amount of time in seconds that the parent device must buffer the messages before it may discard them.

SECURITY MODE

This parameter specifies the use of either Residential or Commercial Security mode, as defined in the ZigBee protocol. The differences between these modes in discussed in **"Secure Transmission"**. Currently, the Microchip Stack for the ZigBee Protocol supports only Residential mode.

TRUST CENTER

The ZigBee protocol defines the concept of a Trust Center to coordinate the operations related to security. A trust center must be an FFD, and there can be only one trust center in a network. The Trust Center address must be defined in the Coordinator as well as in the device defined as the Trust Center.

NETWORK KEY

This parameter specifies the 16-byte network security key. This key is used to secure the outgoing packets as well as to decrypt the incoming packets when security is used in Residential mode. There is also a sequence number for the key, used primarily to identify the key, especially if multiple network keys are transferred and used during runtime. The Network Key must be present for Coordinators and the device that acts as the trust center.

KEY PRESENT IN ALL DEVICES ON THE NETWORK

This parameter is used for Coordinator and Router. If the key is present in all devices on the network, then all devices must contain the Network Key. By defining this parameter, it is assumed that all devices already have the key before joining the network. As a result, the trust center sends the joining device a dummy key, and all packets between devices on the network may be encrypted. If, however, this parameter is not set, the trust center tries to send the joining device the unprotected security key through the joining device's parent.

NONVOLATILE STORAGE

The ZigBee protocol requires that many tables be stored in nonvolatile memory. PIC microcontrollers with an allowable erase block size (smaller than 127 for PIC18F devices) may store these in internal program memory. This is the preferred location, since read and write accesses are relatively fast. However, PIC devices with large erase block sizes, such as the PIC18FJ devices, must store these values externally. The Stack provides support to use an external SPI serial EEPROM to store these values. Since some transceivers require a dedicated SPI peripheral unless external hardware is provided, the SPI selection may be disabled depending on transceiver configuration.

When using external nonvolatile memory, it may be desirable to place each device's MAC address in the serial EEPROM during production rather than using SQTP when programming the PIC. If the MAC address is to be programmed into the serial EEPROM during the manufacturing processes, it should be stored in locations 0 through 7 in the serial EEPROM.

Note:	If the application is to use security and store its nonvolatile information externally, the security keys will be stored in the serial EEPROM. The Stack will encrypt these keys before storing them, using a random key generated by the Stack configuration tool. Unencrypted keys will not be stored externally.

HEAP SIZE

The Microchip Stack uses dynamic memory allocation for many purposes, including those listed in Table 15. RFD end devices may be able to have as little as one bank of heap space. FFDs should have as much space as possible. FFDs with child devices whose transceivers are off when Idle are required to be able to buffer one or more messages for each child. Refer to the appropriate Stack profile for the exact requirement. Heap space will also be required based on the settings above. The selected heap size should take all of these items into consideration, and is therefore, very application dependent.

STACK SIZE (PIC18)

The Microchip Stack requires only one bank of stack space. If your application requires more, ZENA software can generate the appropriate linker script; however, be sure to change the project's memory model to use a multi-bank stack. In MPLAB IDE, select *Project>Build Options>Project* from the menu bar. At the Project dialog, select the **MPLAB C18** tab. Change **Category** to **Memory Model** and select the appropriate **Stack Model**.

TABLE 15: HEAP USAGE

Description	Layer	ZigBee™ Protocol Coordinator	ZigBee™ Protocol Router	FFD End Device	RFD End Device
Checking for descriptor matching	ZDO	X	X	X	X
Checking for end device bind matching	ZDO	X	X(1)		
Buffering messages received from the APL	APS	X	X	X	X
Buffering received indirect messages for retransmission	APS	X	X(1)		
Buffering route requests for rebroadcast	NWK	X	X	X	
Buffering other broadcast messages for rebroadcast	NWK	X	X	X	
Buffering channel information on network formation	NWK	X			
Buffering network information on network join	NWK		X	X	X
Route discovery table entries	NWK	X	X	X	
Buffering messages that require routing	NWK	X	X	X	
Buffering messages for RFD children in Sleep	MAC	X	X		
Buffering a received message	PHY	X	X	X	X
Nonvolatile memory manipulation	NVM	X	X	X	X
Temporary security data during encryption process	SEC	X	X	X	X

Note 1: If binding is supported.

LINKER SCRIPTS

ZENA software generates linker scripts for a small subset of devices. To modify the generated linker script for a different device, change the following items:

`FILES [device].lib` – Change this to the required device name.

`CODEPAGE Sections` – Change these to match those of the required device and environment (MPLAB ICD 2 or production release build).

`ACCESSBANK Sections` – Change these to match those of the required device.

`HEAP Area` – Make sure there is enough room on the required device for the heap size specified. DO NOT MODIFY the START or END parameters of this section. If the size of the heap needs to be changed, regenerate all Stack configuration files using ZENA software.

`RX_BUFFER` – Make sure this section is specified. DO NOT MODIFY the START or END parameters of this section.

`Other DATABANK Sections` – Make sure to copy the SFR and debug areas (if needed) of the required device.

ZigBee Protocol Timing

The data rate for 2.4 GHz operation is 250 kbps. Four data bits are transferred during each symbol period. A symbol period is, therefore, 16 microseconds. Internal Stack timing is based off of the symbol period.

Both beacon and non-beacon networks have timings that are based off of superframes, even though the superframe is not used in non-beacon networks. The superframe duration (*aBaseSuperframeDuration*) is the number of symbols that form a superframe slot (*aBaseSlotDuration*, 60) multiplied by the number of slots contained in a superframe (*aNumSuperframeSlots*, 16). The scan duration required by the `NLME_NETWORK_DISCOVERY_request`, `NLME_NETWORK_FORMATION_request`, and `NLME_JOIN_request` primitives is (*aBaseSuperframeDuration* * (2^n + 1)) symbols, where *n* is the value of the `ScanDuration` parameter. For the Microchip Stack, `ScanDuration` can be between 0 and 14, making the scan time between 0.031 seconds and 4.2 minutes.

For other frequency bands, refer to the IEEE specifications for the data rate. The other times can be calculated from that.

PIC18FJ Family Microcontroller Considerations

Microchip ZigBee Stack version 1.0-3.8 introduced the capability of using the ZigBee protocol Stack with PIC18FJ devices. Due to the large program memory erase block, the ZigBee protocol nonvolatile tables must be stored in an external serial EEPROM with an SPI interface. The Stack configuration tool in the ZENA Wireless Network Analyzer software may be used to configure the interface to the serial EEPROM. Table 16 lists the hardware resources required in the demo projects for PIC18FJ microcontroller family to use the ZigBee Stack.

TABLE 16: PIC18FJ MICROCONTROLLER RESOURCES REQUIRED BY ZigBee™ STACK

Resource	Description
INT0	Used to accept interrupt from MRF24J40 transceiver
TMR0	Used for symbol timer
RC0	Chip selection for MRF24J40
RC1	Voltage regulator/wake pin
RC2	Transceiver reset
RC3	SCK for MRF24J40
RC4	SDI for MRF24J40
RC5	SDO for MRF24J40
RD0	Chip selection for external EEPROM
RD3	SCK for external EEPROM
RD4	SDI for external EEPROM
RD5	SDO for external EEPROM

CONCLUSION

The Microchip Stack for the ZigBee protocol provides a modular, easy-to-use library that is application and RTOS independent. It is specifically designed to support more than one RF transceiver with minimal changes to upper layer software. Applications can be easily ported from one RF transceiver to another. It is targeted for the MPLAB C18 C compiler, but it can be easily modified to support other compilers.

REFERENCES

- *"ZigBee™ Protocol Specification"* http://www.zigbee.org
- *"PICDEM™ Z Demonstration Kit User's Guide"* (DS51524) http://www.microchip.com
- *"IEEE 802.15.4™ Specification"* http://www.ieee.org
- *"ZENA™ Wireless Network Analyzer User's Guide"* (DS51506) http://www.microchip.com

SOURCE CODE

The complete source code, including demo applications, is available for download as a single archive file from the Microchip corporate web site at:

www.microchip.com

ANSWERS TO COMMON QUESTIONS

Q: Is the Microchip Stack for the ZigBee protocol a ZigBee protocol compliant platform?

A: Version 10.-3.8 was the first version of the Microchip Stack for the ZigBee protocol to be certified as a ZigBee compliant platform. Refer to the Readme for a particular version of the Stack for information on ZCP status.

Q: I want to use a wireless protocol, but I do not want all of the ZigBee protocol features. May I modify the Microchip Stack for my own use without receiving any further permissions?

A: No. Microchip has the relevant license rights to distribute this Stack. However, you must be a member of the Zigbee Alliance and have a current license to the Microchip Stack for the Zigbee protocol in order to distribute products using the Microchip Stack. Neither Zigbee Alliance nor Microchip allows modifications to be made to the Microchip Stack.

Q: How do I get the source code for the Microchip Stack for the ZigBee protocol?

A: You may download it from the Microchip web site (www.microchip.com), from either the *AN965, "Microchip Stack for the Zigbee™ Protocol"* or the *"PICDEM™ Z Demonstration Kit User's Guide"* page.

Q: How do I get target hardware design files?

A: You may download it from the *"PICDEM™ Z Demonstration Kit User's Guide"* page on the Microchip web site.

Q: What tools do I need to develop a ZigBee protocol application using the Microchip Stack?

A: You would need:

- At least one PICDEM Z demo kit or at least two of your own ZigBee protocol nodes
- Complete source code for the Microchip Stack for the ZigBee protocol for PIC18 branded products (free of charge)
- The MPLAB C18 C compiler
- MPLAB IDE software
- A device debugger and programmer, such as MPLAB ICD 2

Q: How much program and data memory does a typical ZigBee protocol node require?

A: The exact program and data memory requirements depend on the type of node selected. In addition, the sizes may change as new features and improvements are added. Please refer to the Readme file for more detail.

Q: What is the minimum processor clock requirement for running the different devices?

A: Normally, ZigBee protocol coordinators and routers should run at higher speeds as they must be prepared to handle packets from multiple nodes. The required clock speed depends on the number of nodes in the network, the types of nodes and the frequency at which the end devices request data. The demo coordinator uses 16 MHz (4 MHz with 4x PLL) and can support multiple child devices. We have not performed extensive characterization, since there are so many possible configurations. An end device does not have to run as fast as a coordinator or router. A simple end device may be run at just 4 MHz.

Q: Can I use the internal RC oscillator to run the Microchip Stack?

A: Yes, you may use the internal RC oscillator to run the Microchip Stack. If your application requires a stable clock to perform time-sensitive operations, you must make sure that the internal RC oscillator meets your requirement or you may periodically calibrate the internal RC oscillator to keep it within your desired range.

Q: What is the typical radio range for PICDEM Z demo boards?

A: The exact radio range depends on the type of RF transceiver and the type of antenna in use. A 2.4 GHz-based node with a well designed antenna could reach as high as 100 meters line-of-sight. When placed inside a building, the typical internal range is about 30 meters, but the actual range may be greatly reduced due to walls and other structural barriers.

Q: I have an existing application that uses a wired protocol, such as RS-232, RS-485, etc. How do I convert it to a ZigBee protocol-based application?

A: First, you would need to match your application with one of the ZigBee public profiles. If no public profile is appropriate, you would have to create your own private profile.

If your network is relatively small, the Microchip MiWi™ protocol provides an alternative. (For more information, see *AN1066, "MiWi™ Wireless Networking Protocol Stack"*.)

You would need to develop one ZigBee protocol coordinator and one more ZigBee protocol end-device application. The coordinator is required to create and manage a network. If your existing network has one main controller and multiple end devices or sensor devices, your main controller would become a ZigBee protocol coordinator and sensor devices would become ZigBee protocol end devices. If the existing devices are already mains powered, you may want to consider making the end devices FFDs rather than RFDs. FFDs do not generate as much network traffic and can easily be converted to routers in case one or more of your devices is out of radio range of the coordinator. You must make sure that the radio range offered by a specific RF transceiver is acceptable to your application.

Q: How do I obtain the ZigBee protocol and IEEE 802.15.4 specification documents?

A: Both specifications are freely available on the internet. The IEEE 802.15.4 specification is available at http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf. The ZigBee protocol specification is available at www.zigbee.org.

Q: I have an application that I have built with an earlier version of the Microchip Stack. How do I port my application to the new Stack?

A: The interface to the v1.0-3.8 Stack architecture is much simpler than the interface to the old Stack. Study the template file for the device type you need. The places to insert application-specific code are indicated by large comment blocks:

- **Application-Specific Initialization:** Insert any initialization required by the application before the Stack is started.
- **Received ZDO Responses:** Insert code here to handle responses to ZDO requests that the application issues. If the application does not issue any ZDO requests, this section will be empty.
- **Messages Received for User-Defined Endpoints:** The new architecture handles endpoints differently. There is no need to "open" or "close" an endpoint. Each endpoint is simply a case of a switch statement. Note that the `APLDiscardRx()` function is called after the switch statement, so the individual endpoints do not need to call it.
- **Application Processing that can Generate ZigBee Protocol Messages:** A new outgoing message can only be started if the current primitive is `NO_PRIMITIVE` and another outgoing message is not already waiting (`ZigBeeReady()` returns `TRUE`). Place all message generation processing from all endpoints here. Note that no code is required to retry the message in case it fails to transmit or receive an APS level Acknowledge. That is now handled automatically by the Stack. Also, the Stack now automatically handles all message routing.
- **Non-Related ZigBee Protocol Processing:** If the application has any other processing that does not relate at all to the ZigBee protocol, place that code here. Make sure that this processing does not lock the system for long periods of time or the Stack will miss incoming messages.
- **Hardware Initialization:** The required hardware initialization for the PICDEM Z demo board is included in the template files. If your hardware requirements are different, modify this function appropriately. Note that this function must properly configure all pins required to interface with the transceiver and must be called before `ZigBeeInit()`.

Network formation and association are provided by the templates. You may wish to change some of the parameters of these primitives, but the basic structure will remain unchanged.

REVISION HISTORY

Rev A Document (12/2004)

Original version of this document.

Rev B Document (04/2006)

This document is a complete rewrite of Rev A.

Rev C Document (01/2007)

Added section on **"Secure Transmission"**; added security-related configuration parameters; added additional information on microcontroller hardware requirements and system resource clean-up.

NOTES:

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Trademarks

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona, Gresham, Oregon and Mountain View, California. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Habour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Fuzhou
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Shunde
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7250
Fax: 86-29-8833-7256

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Gumi
Tel: 82-54-473-4301
Fax: 82-54-473-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Penang
Tel: 60-4-646-8870
Fax: 60-4-646-5086

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

12/08/06