

Microchip ZigBee® PRO Feature Set Protocol Stack

Author: Derrick P. Lattibeaudiere
Microchip Technology Inc.

INTRODUCTION

The ZigBee® protocol is a wireless network protocol developed specifically for low data rate sensor and control networks. Wireless applications that may benefit from ZigBee include, but are not limited to, industrial controls, home and building automation, PC peripherals, medical sensor applications and security networks.

The latest protocol specifications ratified by the ZigBee Alliance is collectively referred to as ZigBee-2007. ZigBee-2007 defines two distinct levels of functionality or feature sets. The standard set is called ZigBee, and the more advanced feature set is named ZigBee PRO. When compared against ZigBee, the ZigBee PRO feature set offers many enhanced networking capabilities, and, under certain conditions, is backwards compatible with ZigBee.

The purpose of this application note is to provide a description of the ZigBee PRO Features Set as implemented by the Microchip ZigBee PRO Stack. These features are described in the “**ZigBee Pro Feature Set Overview**” section of this document.

ASSUMPTION

This application note is not self contained, and does not provide an introduction to the ZigBee protocol. Rather, it builds upon the information provided in a previous application note AN1232, “*Microchip ZigBee-2006 Residential Stack Protocol*”. For readers who are not familiar with ZigBee or previous releases of Microchip’s ZigBee Stack, it is highly advised that application note AN1232 be carefully reviewed prior to reading this document or attempting to use the ZigBee PRO Feature Set Stack. Many introductory concepts related to the ZigBee protocol and how the Microchip ZigBee Stacks are structured are covered in that application note and are therefore not repeated here.

Furthermore, this document assumes that the reader is familiar with the ZigBee protocol and its terminology. The reader is also expected to be familiar with the C programming language, as well as the IEEE 802.15.4-2003 specifications in detail. For additional technical information on the IEEE 802.15.4™ specifications, please refer to <http://standards.ieee.org/catalog/>. For additional technical information on the ZigBee specifications, please refer to www.zigbee.org.

DISTRIBUTION NOTICE

Companies wishing to distribute a product that uses the Microchip ZigBee PRO Stack for the wireless network protocol portion of their product must be members of the ZigBee Alliance. Additionally, companies may only use the Microchip ZigBee PRO Stack in their products when it is used in conjunction with a Microchip transceiver and microcontroller. Please refer the software license that accompanies the stack. For additional information regarding ZigBee licenses and product certification, refer to www.zigbee.org and specifically to document “053594r03_ZQG_ZigBee_Certification”.

Note: The Microchip ZigBee PRO Stack is available for purchase from the www.microchipdirect.com website. Due to governmental security regulations regarding 128-bit encryption software, the ZigBee PRO stack is not available for download from the Microchip website.

ZigBee COMPLIANT PLATFORM

The Microchip ZigBee PRO Protocol Stack has been certified as a ZigBee Compliant Platform (ZCP) consisting of the following modules:

- Processor: PIC24F and dsPIC33 families of microcontrollers
- Transceiver: MRF24J40
- Firmware: Version 2.0.PRO.2.0 of the Microchip ZigBee PRO Stack

FEATURES

The Microchip ZigBee PRO Stack is designed to evolve with the ZigBee wireless protocol specifications. At the time of this publication, the current applicable ZigBee specification document is 05347 r17. This document applies to Microchip’s ZigBee PRO Stack releases v2.0.PRO.2.0 and greater. The Microchip ZigBee-2006 Stack is described in application note AN1232.

The Microchip ZigBee PRO Stack offers the following features:

- A certified ZigBee PRO Compliant Platform (ZCP)
- Support for the 2.4 GHz frequency band using the MRF24J40 transceiver
- Support for all ZigBee protocol device types (Coordinator, Routers and Reduced Function End Devices)
- Stochastic Address and Address Conflict Resolution mechanisms are supported

- Support for Data Fragmentation and Reassembly
- Support for Frequency Agility and Dynamic Channel Selection
- Support for Source Routing
- Support for Many-to-One Routing
- PANID Conflict Detection and Resolution Mechanism is supported
- Support for High Security Key Exchange
- Support for Centralized Data Collection (ZigBee Concentrator Device)
- Support for Commissioning via the Startup Attribute Set (SAS)
- RTOS and application independent
- Portable across the PIC24 MCU and dsPIC33 DSC families
- Out-of-box support for Microchip MPLAB® C Compiler for PIC24 MCUs and dsPIC DSCs
- Implements nonvolatile storage for critical network parameters such as Neighbor and Routing Tables

CONSIDERATIONS

Version 2.0.PRO.2.0 of the Microchip Stack for the ZigBee Protocol is the third version to be granted the status of ZigBee Compliant Platform (ZCP).

For information on the ZCP status of version v2.0-2.6, please refer to AN1232, “Microchip ZigBee-2006 Residential Stack Protocol”.

The first version, v1.0-3.8, of the Microchip's ZigBee Stack has been deprecated and is no longer supported. Users are encouraged to migrate to either version v2.0-2.6 or to the ZigBee PRO version 2.0-PRO.2.0 that is described in this document.

LIMITATIONS

The ZigBee protocol specifications leave many higher level decisions up to the developer and product designer. As such, the Microchip ZigBee PRO Stack provides no explicit support for the following functions in the current release:

- Beacon networks are not supported in this version of the ZigBee PRO protocol stack.
- The Smart Energy Profile is not implemented
- The Home Automation Profile is not implemented
- The SKKE security mechanism is not implemented
- The ZigBee Cluster Library (ZCL) is not implemented
- Alternate PAN coordinator capability is not supported in ZigBee protocol networks. Only a single ZigBee protocol coordinator is permitted.
- The Zena™ Packet Sniffer does not currently support this released version of the ZigBee PRO Feature Set Stack. Microchip recommends using Daintree Sensor Network Analyzer (SNA) for customers' ZigBee PRO developments.

DEVELOPMENT TOOLS REQUIREMENTS

In order to use the Microchip ZigBee PRO Stack to create a ZigBee protocol network consisting of ZigBee devices, the following development tools are required.

The hardware platform consists the following (one each per network node):

- Explorer 16 (DM240001) board
- PIC24FJ128GA010 Plug-In-Module (PIM) (MA240011)
- MRF24J40 2.4 GHz Daughter Card (AC163027-4) or MRF24J40MA PICtail™ Plus 2.4G Hz RF Card (AC164134)

Miscellaneous Hardware

- At least one RS-232 Serial Cable in order to configure and communicate with the hardware
- Personal Computer with RS-232 COM port or USB to RS-232 adapter
- A programmer such as MPLAB REAL ICE™ in-circuit emulator or MPLAB ICD 3

Software Tools

- MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs
- MPLAB IDE v8.10 or later
- The source code for Microchip ZigBee PRO Stack version v2.0.PRO.2.0 or higher
- Daintree Sensor Network Analyzer (SNA) or similar ZigBee Packet Sniffer for those intending to do moderate to complex application development with this stack (optional)

Together these development tools will allow the user to create a ZigBee network using the Microchip ZigBee PRO Stack. The exact procedure of how to do this is covered in the ZigBeePROQuickStartGuide.chm document that accompanies the stack software.

ZigBee PRO FEATURE SET OVERVIEW

The following sections provide an in-depth discussion of the important features that make up the ZigBee PRO Feature Set. Where appropriate, specifics of how each feature is implemented and its impact on the future development of other ZigBee profiles is also covered.

Stochastic Addressing

The Stochastic Addressing feature of the ZigBee PRO Stack allows each device that joins the network to be randomly assigned a unique 16-bit network address. The only exception is the ZigBee Coordinator, which still retains a network address of 0x0000.

This random network address assignment stands in contrast to the CSKIP mechanism employed in the ZigBee-2006 Stack, where network addresses were pre-determined and distributed based on device type and the network topology.

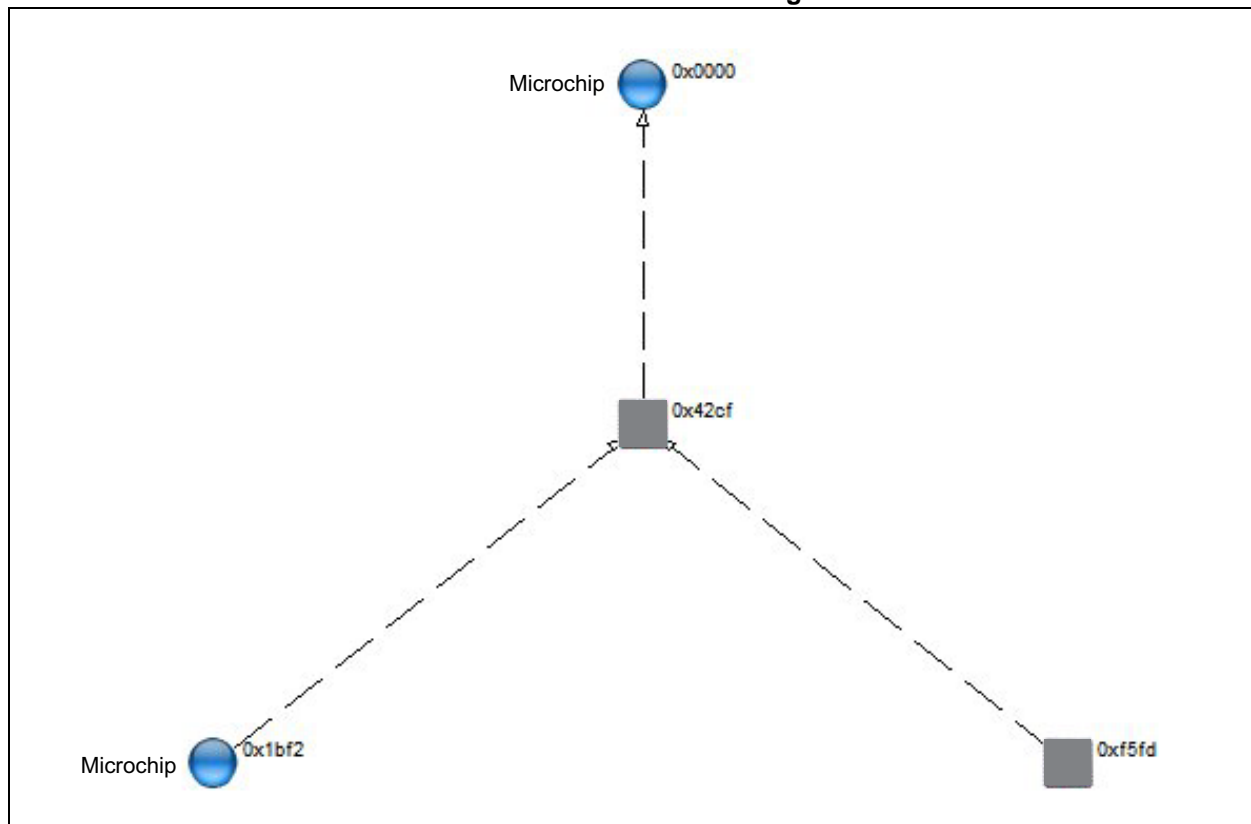
Under the stochastic addressing scheme, once a device has been assigned its network address, it may choose not to relinquish it, unless that address comes in conflict with another device on the network. This is true even during the rejoin process, when a device may be switching parents.

The Stochastic addressing mechanism in ZigBee PRO simplifies the network address calculation (the CSKIP algorithm vs. generating a random number), and removes the linkage between the network address of an individual device and its position within the network topology. One benefit of this feature is that the entire addressing space is now made available to each potential parent device on the network.

Figure 1 shows a sample ZigBee PRO network consisting of four devices, and their associated randomly generated network addresses. Note the difference in the network address assignment when compared against the ZigBee-2006 Stack.

Version 2.0-2.6 of the Microchip ZigBee-2006 Stack does not support the stochastic addressing feature.

FIGURE 1: STOCHASTIC NETWORK ADDRESSES OF ZigBee® PRO DEVICES



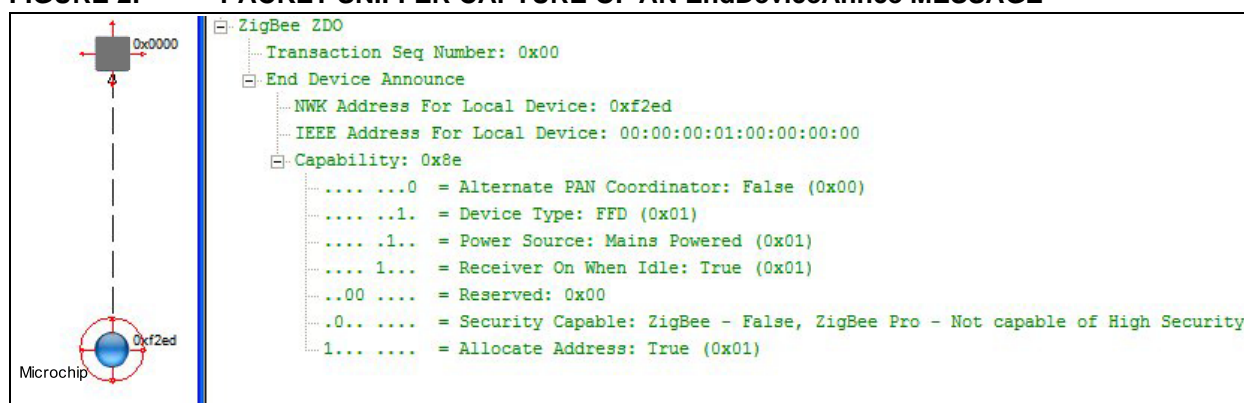
Address Conflict Detection and Resolution

An address conflict occurs when two devices on the same network have identical network addresses. More precisely, an address conflict arises when the same network address gets associated with two different MAC addresses. This conflict situation can arise, for example, when two parent devices generate the same random network address for each of their respective child devices, where both child devices have different MAC addresses.

The detection of an address conflict usually occurs when a device that has just joined the network broadcasts an announcement to notify the other devices that it is now a member of the network. This announcement is called a ZigBee **EndDeviceAnnce** message and carried within its payload are both the MAC and network address of the newly joined device, as well as a byte that identifies the capabilities of the device. An example of the information contained in the Capability byte would be that the device is an RFD, its transmitter is turned off when the device is idle, and it does not support security.

Figure 2 shows a packet sniffer capture of the EndDeviceAnnce message that is broadcasted to all devices in the network.

FIGURE 2: PACKET SNIFFER CAPTURE OF AN EndDeviceAnnce MESSAGE

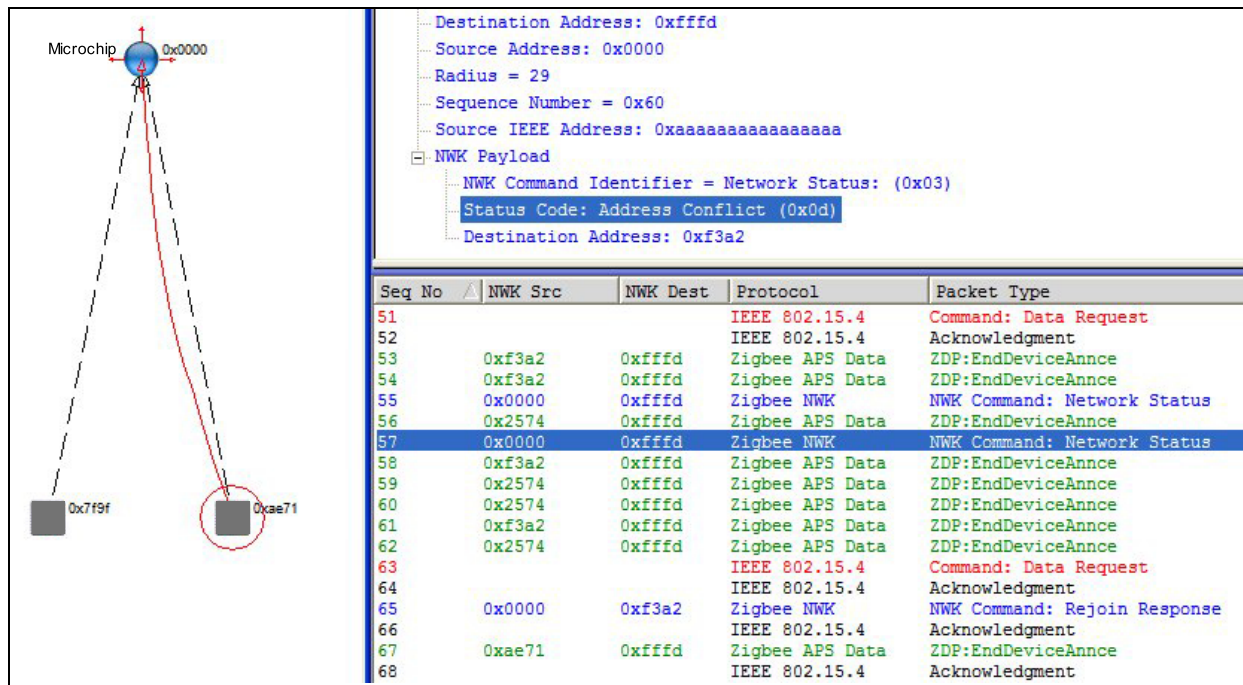


After every device announcement is received, routers and the Coordinator will compare the new device's network address against all the known addresses in their address map and neighbor tables. If another device with the same network address as the newly joined device is found, or the new network address is the same as the router's own, then a status command is broadcasted throughout the network indicating an address conflict situation has been detected.

If the device with the address conflict is a Reduced Function End Device, the parent of that device will choose a new random network address for the child. It will then send an unsolicited rejoin command to the child, forcing it to accept the new network address, thus alleviating the address conflict. This new network address is embedded in the payload of the unsolicited rejoin command.

If a router is the device that is the source of the address conflict, it will assign a new address to itself and announce its new address via a new EndDeviceAnnce message.

Figure 3 shows an example of the sniffer capture of the address conflict detection and resolution mechanism at work.

FIGURE 3: PACKET SNIFFER ILLUSTRATION OF ADDRESS CONFLICT DETECTION

- **Seq No. 57:** Shows the Coordinator detecting an address conflict, with NWK Address 0xf3a2 being the source of the conflict.
- **Seq. No. 65:** Since the Coordinator is the parent of that device, it sends an unsolicited Rejoin Response command to device 0xf3a2. The response carries the new randomly chosen NWK address of 0xae71 in its payload.
- **Seq. No. 67:** The device, upon receiving this unsolicited rejoin, accepts its new NWK address and rejoins the network. The device sends out a new EndDeviceAnnce message to all the FFDs in the network alerting every device of its new address.

This feature is only supported in the Microchip ZigBee PRO Stack and not in any of the earlier versions.

The ZigBee PRO Network Channel Manager

The ZigBee PRO device that implements a particular subset of the network management functions, including PANID Conflict Resolution and Frequency Agility measures when interference is encountered, is called the Network Channel Manager (NCM). For the sample application that accompanies Microchip's ZigBee PRO Stack, the Coordinator is the overall Network Channel Manager Device. However, the Coordinator may designate another FFD to perform the role of Network Channel Manager.

Within Microchip's ZigBee PRO Stack, the Network Channel Manager device performs two key duties:

- First, it is the central device that receives channel interference reports and facilitates the changing of the current operating channel in order to mitigate the interference. In order to effect a channel change, the Network Channel Manager maintains a list of channels to be used during the channel scanning process (i.e., determines the value of the channelMask).
- Second, the Network Channel Manager handles the PANID conflict resolution process. Whenever it gets a report of a PANID conflict (the process is discussed in section **"PANID Conflict Detection and Resolution"**), it selects the new PANID on which the network will operate, and broadcasts it to all the devices.

In the sample application that accompanies the Microchip ZigBee PRO Stack, after the Coordinator has formed a network, it may designate another device to become the Network Channel Manager. The Coordinator does so by broadcasting (0xffff), a ZigBee Mgmt_NWK_Update_request command message. Inside the payload of this message is the network address of the designated Network Channel Manager device as well as the list of channels to be scanned during the energy detection phase. After receiving the Mgmt_NWK_Update_request command, the designated NCM device will begin to perform its duties, and the other devices will register the designated NCM as such. Subsequently, they will send their channel management notification messages to the NCM device.

PANID Conflict Detection and Resolution

Note: A ZigBee PRO network has two Personal Area Network Identifiers (PANID). The first is a 64-bit globally unique PAN identifier named the Extended PANID, that should be unique within any overlapping network area. The second is 16-bit PAN Identifier called the Short PANID. When used together, this pair, the Extended and Short PANIDs, shall uniquely identify the network.

A PANID conflict occurs when any device operating on a ZigBee PRO network receives a beacon frame via a MLME-BEACON-NOTIFY, indication primitive, in which the PANID of the beacon frame matches that of its own PAN Identifier, but the Extended PANID contained within the beacon frame's payload does not match its own Extended PANID, at which point the device has detected a PANID conflict.

Any device that detects a PANID conflict will report it to the current device that is designated as the Network Channel Manager via a ZigBee defined Network Report Pan Identifier Report Conflict command frame.

Upon receipt of the Network Report Pan Identifier Report Conflict command frame, the Network Manager creates a new random 16-bit PANID, and transmits it to the other devices via a Network Update PAN Identifier Update command broadcast (destination address 0xffff).

All the devices on the network, upon receipt of the PAN Identifier Update command, will extract the new PAN Identifier from the command payload, and update their beacon payloads accordingly.

From an implementation perspective, this new PANID is also written in the transceiver hardware, effectively switching the network to a new PAN for the purpose of PAN filtering incoming packets.

Figure 4 and Figure 5 show packet sniffer traces of the of network commands that are transmitted during the process of detecting and resolving a PAN Identifier conflict on Microchip's ZigBee PRO network.

FIGURE 4: PACKET SNIFFER CAPTURE OF PANID 0x1aaa CONFLICT DETECTION

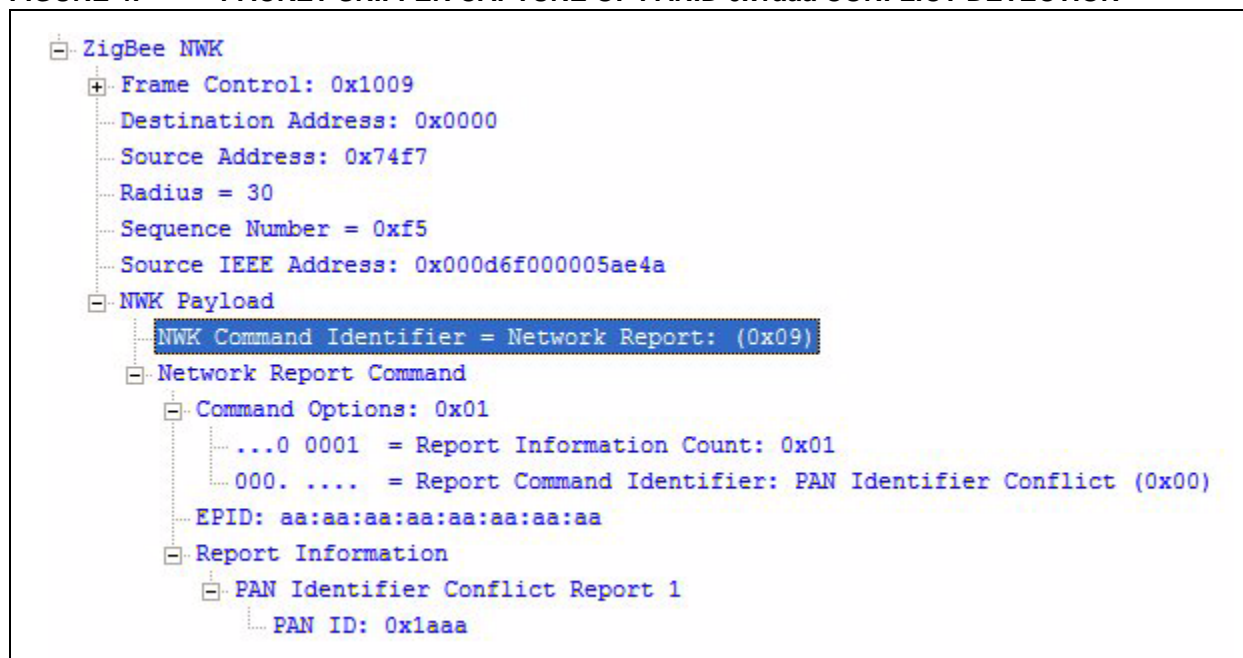
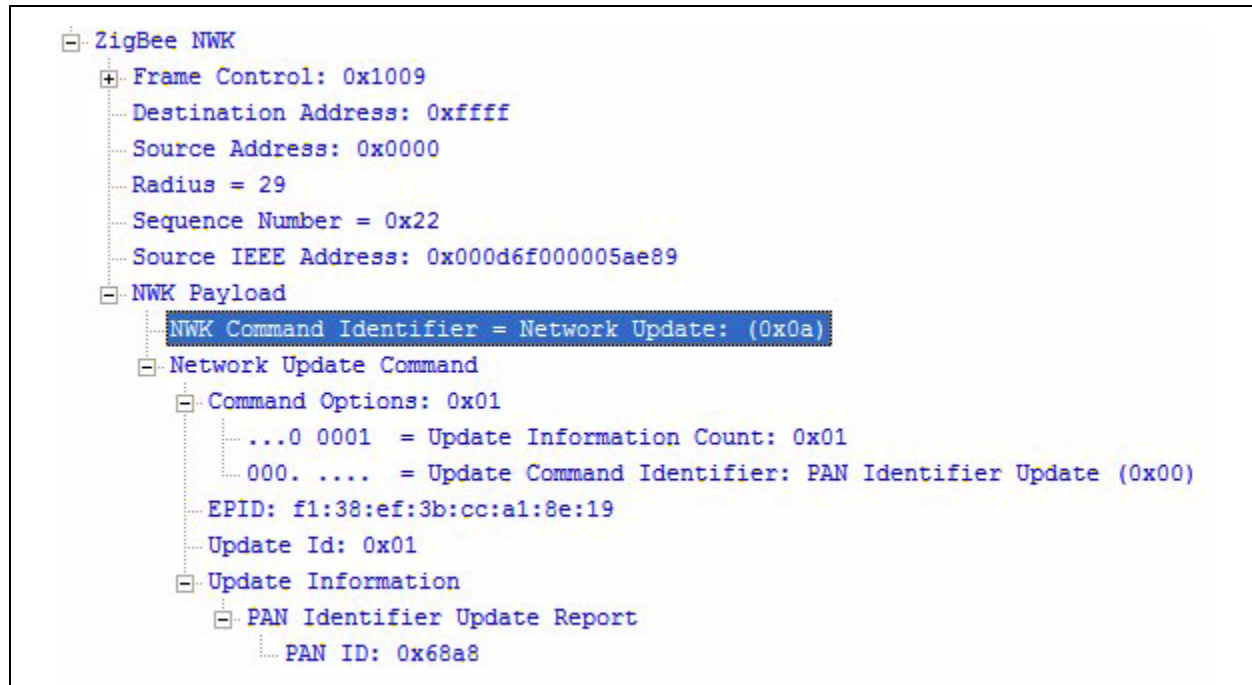


FIGURE 5: PACKET SNIFFER CAPTURE OF PANID 0x1aaa CONFLICT RESOLUTION



Fragmentation and Reassembly

The maximum size of a ZigBee frame, including all the information that is carried in the headers, is 127 bytes. Fragmentation allows applications that have the need to transmit large payloads, which would exceed the 127-byte frame limit, to segment the data into smaller transmittable chunks.

On the receiver side, a Reassembly mechanism allows for the reconstruction of those payload chunks into a single network layer frame, which is then passed into the application layer as a single whole unit.

In order to setup and control the transmittal and reception of a fragmented frame, several parameters mandated by the ZigBee specifications are used within the Microchip ZigBee PRO Stack to implement this feature. The definition of these parameters will be explained by using an example.

Consider an application that has a 325 byte payload to be transmitted. Clearly this exceeds the 127-byte frame limit and must be split up into transmittable blocks or fragments:

- The **fragmentTotalDataLength** parameter represents the total number of payload bytes to be transmitted. This parameter is 325 in our example.
- The network application developer has the freedom to choose, at the application level, the size of each payload chunk when transmitting a fragmented packet. The parameter that governs this is called the **fragmentDataSize**. For this example, if 50 bytes is chosen, then each block size or

fragmentDataSize is 50 bytes long.

- In order to transmit the entire 325 bytes of the payload, 325-bytes divided by 50 bytes per block = 7 blocks total. Six blocks, each carrying 50 bytes, plus a final seventh block with only 25 bytes are needed to complete the transmittal.
- The parameter **fragmentWindowSize** represents the number of blocks that can be sent to a receiving device before the receiver is required to send back to the transmitting device an explicit acknowledgement packet, indicating that all fragmentWindowSize number of blocks have been received. This helps keep the two devices in sync, and not have the sending device transmit too many blocks without receiving explicit acknowledgement from the receiver, stating that the transmission process is proceeding well.

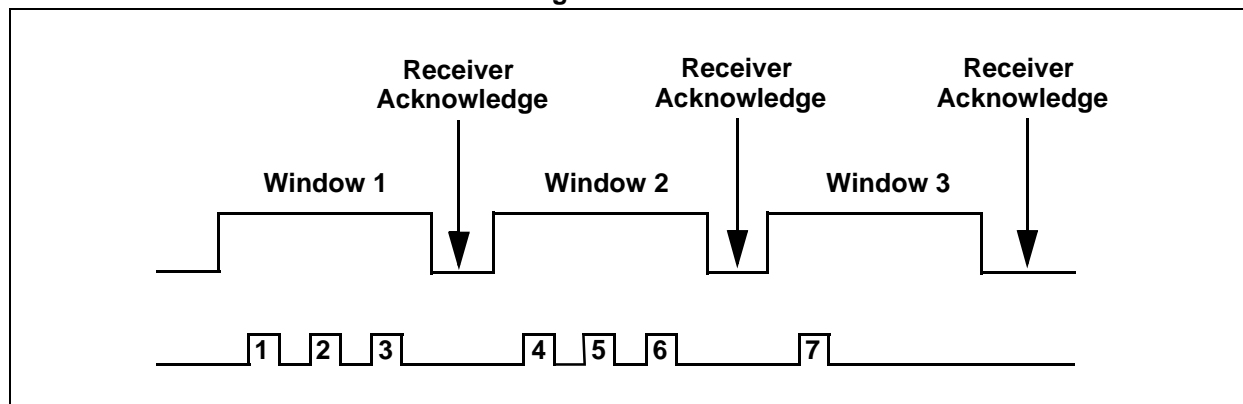
For our example, if fragmentWindowSize is set to 3, then after every group of 3 blocks are successfully transmitted, the receiver would respond with an acknowledge packet. At that time, the sender would then proceed to send the second set of 3 blocks, and then wait for the second acknowledgement. Finally, the last frame, containing the last block, the 7th one, would be transmitted. The receiver, recognizing that the transaction is over, would send back the final acknowledgement packet, reassemble the entire 325-byte packet, and send it up to the application layer as a whole entity.

- The **fragmentInterframeDelay** parameter represents the time delay in milliseconds between transmissions of each block of a fragmented frame. This delay provides the receiving device with adequate time to process the packet and do the internal bookkeeping necessary to keep track of all the packets.

Internal to the ZigBee PRO Stack, the total number of blocks to be transmitted, in addition to which block has currently been transmitted, is kept via an internal bookkeeping mechanism. If a block is dropped or fails to be successfully transmitted for whatever reason, the internal bookkeeping mechanism will facilitate an automatic retransmission without any application code involvement.

Figure 6 shows a pictorial representation of the fragmentation and reassembly process in relationship to the parameters described.

FIGURE 6: ILLUSTRATION OF THE ZigBee® PRO FRAGMENTATION PARAMETERS



```
/******
```

Function:

```
void ZIGAPLUpdateFragmentParams ( BYTE WindowSize, BYTE InterframeDelay, BYTE
DataLengthPerBlock, BYTE TotalFragmentDataLength)
```

Summary:

Initializes the parameters that controls how a fragmented packet will be transmitted.

Description:

This function is used to set the parameters which are used by fragmentation feature to control how the actual fragmentation hand shaking between the transmitting and receiving device is carried out.

Precondition:

The ZigBee devices must support fragmentation in order to use this function

Parameters:

WindowSize - BYTE specifies the number of blocks that can be Txd/Rxd in one window
InterframeDelay - BYTE specifies the time delay between the block transmission
DataLengthPerBlock - BYTE specifies the data length that can be fit in one block transmission
TotalFragmentDataLength - BYTE specifies the total number bytes that needs to be transmitted using fragmentation

Returns:

None

Remarks:

None

```
*****/
```

```
void ZIGAPLUpdateFragmentParams( BYTE WindowSize, BYTE InterframeDelay, BYTE
DataLengthPerBlock, BYTE TotalFragmentDataLength );
```


It is up to the application developer to set the default values for all the fragmentation/reassemble parameters that were discussed above, prior to calling the `ZIGAPLSendFragmentedPacket()` function. For an example of this, see the Sample demo application code that is shipped with the Microchip ZigBee PRO Stack sample application.

Frequency Agility

The Frequency Agility feature gives a ZigBee PRO application the ability to dynamically switch the current channel on which the network operates, primarily in response to detected “interference”.

In order to support this feature, a Network Channel Manager device is required. A ZigBee PRO Network Channel Manager is the device that is designated as such in either the profile's Node Descriptor, or dynamically by the Coordinator after the network has started. It has the responsibility of the managing the channels on which the ZigBee network operates.

Internally, the Microchip ZigBee PRO Stack keeps track of the number of packets that are transmitted by each transceiver. By the standards set forth by the 802.15.4 specifications, for each transmitted packet, there must be an associated MAC level acknowledgement. If no acknowledgement is received, then the packet is judged to be lost and is counted as failure internally by the stack. For a given channel, whenever the ratio of the total transmitted packets vs. the number of transmit failures exceeds a 50% threshold, then the stack assumes there is some “interference” on that channel and this will automatically trigger the start of a corrective action.

The device that experiences a high level of transmit failures will notify the Network Channel Manager by sending it a `MGMT_NWK_UPDATE_NOTIFY` command. This command will indicate the total number of attempted transmissions, the total failures, a list of channel scanned and their energy values. The scanned channels and energy values provide useful information that the Network Channel Manager makes use of as it takes corrective action in order to avoid further interference.

The designer of the network application must decide what precise action to take in response to an interference `MGMT_NWK_UPDATE_NOTIFY` command. Here are a few possibilities:

- The Network Channel Manager, upon receipt of the `MGMT_NWK_UPDATE_NOTIFY`, may ask all devices to scan their channels, and use that information to decide which is the best channel for all the devices. Subsequently, it would tell all the devices to move to that new channel.

- The Network Channel Manager, upon receipts of the `MGMT_NWK_UPDATE_NOTIFY`, may use the information that is received from the single notifying device to decide which is the best channel for all the devices. Subsequently it would tell all the devices to move to that new channel.
- The Network Channel Manager may decide to do nothing and continue to operate the network on the degraded channel.

The Network Channel Manager uses the `MGMT_NWK_UPDATE_req` command to broadcast the channel change to all the devices in the network. After receiving this request, all the devices will switch to the new channel and resume operation. The Network Channel Manager will also do the same.

It should be noted that the Microchip ZigBee PRO Stack provides the entire infrastructure to manage Frequency Agility, but it is up to the application designer to make the final policy decision on how this feature actually functions in their network. The sample application that is provided with the Microchip ZigBee PRO Stack demonstrates this feature.

Link Status Commands

The coordinator and all routers operating on a ZigBee PRO network are required to periodically broadcast a Link Status Command. The Link Status Command carries a list of the device's neighbors that are within a one-hop radio range.

Additionally, the link costs, both out-going and incoming, for each neighbor device are also included in the Link Status Command. The purpose of the Link Status Command is tri-fold:

First, the link status command is used to keep the number of entries in the neighbor table as small as possible. Since each router must periodically broadcast a Link Status Command, each of its neighbors uses the reception of this command as an indication that the device that originated the command is alive and operational.

If a router does not receive this command from a previously neighboring device after a certain interval of time, then it removes that device from its neighbor table. Thus, if a device has moved outside the radio range, or has become inactive, it will be no longer occupy a valuable entry slot inside a neighboring device's neighbor table.

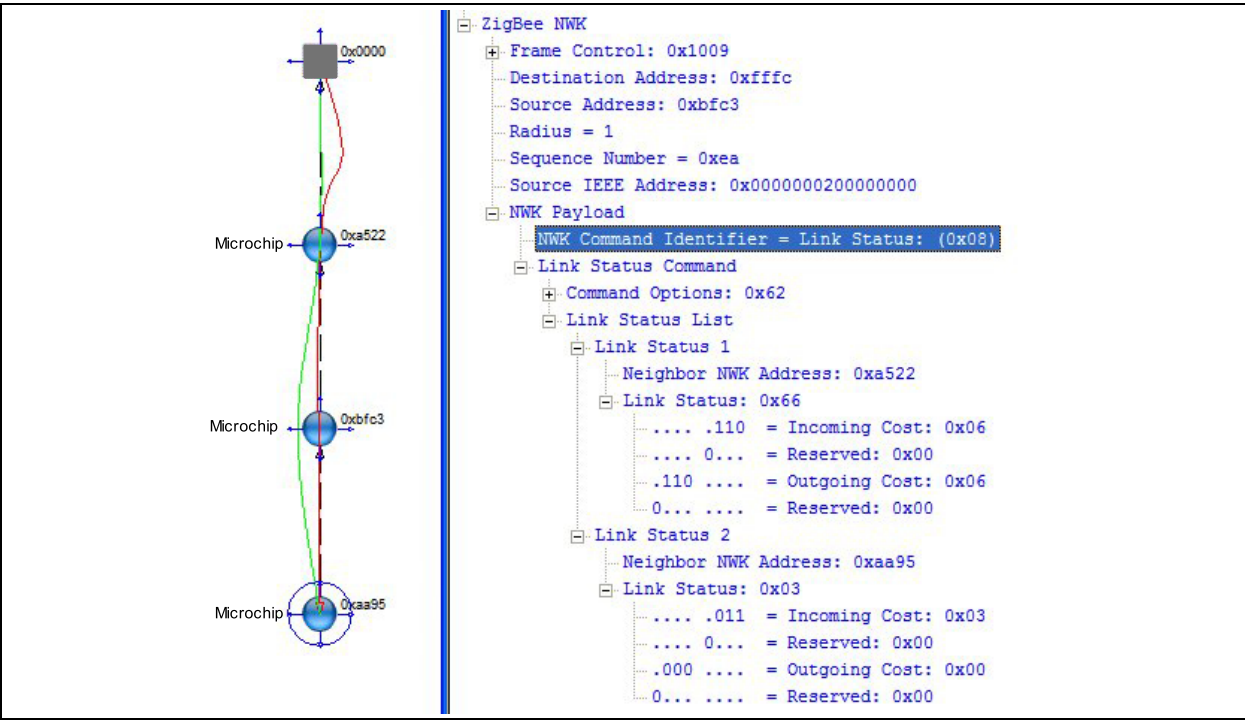
Second, the Link Status Command is used to exchange the link cost information with a devices' one-hop neighbors. The Link Status command carries within its payload the incoming and outgoing link cost of all its known neighbors.

Each neighbor device, upon receiving this command, will search for its own address within the list carried by the link command payload. The incoming and outgoing cost will be updated in its neighbor table to reflect the information that it received from its neighbor.

Third, the link status command makes it easier to calculate the route to any given device within the network. This is because, instead of having to send out route discovery requests to get the path cost, this information can often be calculated from the incoming and outgoing link cost information already stored in the neighbor table.

Figure 7 shows a packet capture example of a Link Status Command.

FIGURE 7: ZigBee® PRO LINK STATUS COMMAND PACKET CAPTURE



The Startup Attribute Set and Nonvolatile Storage Feature

In order to support the future development of ZigBee profiles such as Smart Energy, provide interoperability with other manufacturers’ devices, and to supply the infrastructure necessary to support device commissioning, the Microchip ZigBee PRO Stack implements the Startup Attribute Set (SAS) mechanism.

SAS provides the means by which each device stores, in Nonvolatile Memory (NVM), a set of parameters that is necessary for it to either join or rejoin a specific network.

Table 1 depicts the parameters that are stored in a single instance of the Startup Attribute Set.

TABLE 1: THE PARAMETERS OF THE STARTUP ATTRIBUTE SET (SAS)

Parameter	Size (Bytes)
STARTUP_PARAMETERS_ATTRIBUTE_SET	
The Devices' Network Address	2
The Network Extended PANID	8
The Short PAN ID	2
Channel Mask	4
Protocol Version	1
Stack Profile ID	1
Startup Control	1
UseInsecureJoin	1
The Trust Centers' MAC Address	8
The Trust Centers' MasterKey	16
The Devices' Preconfigured LinkKey	16
The Network Key Sequence Number	1
The Network KeyType	1
The Channel Managers' Network Address	2
JOIN_PARAMETERS_ATTRIBUTE_SET	
Scan Attempts	1
Time Between Scans	2
RejoinInterval	2
MaxRejoinInterval	2
END_DEVICE_PARAMETERS_ATTRIBUTE_SET	
IndirectPollRate	2
ParentRetryThreshold	1
CONCENTRATOR_PARAMETERS_ATTRIBUTE_SET	
Concentrator Indicator Flag	1
Concentrator Radius	1
Concentrator Discovery Time	1
Miscellaneous	2

The SAS consists of a total of 94 bytes. In the Microchip ZigBee PRO Stack, three distinct instances of the SAS are stored in nonvolatile memory. They are known as the Default_SAS, SAS_1 and SAS_2.

Table 2 shows the three SAS blocks. Internal to the ZigBee PRO Stack, it uses an 8-bit index to reference each distinct SAS. For example, the Default_SAS is referenced by using an index value of 0xFF, while the other two use Index 0x00 and 0x01, respectively.

TABLE 2: REPRESENTATION OF THE THREE SAS BLOCKS IN THE ZigBee® PRO STACK

SAS Blocks	Index Value
Default_SAS	0xFF
SAS_1	0
SAS_2	1

The Default_SAS obtains most of its parameter values from the `zigbee.def` file. Therefore, product developers that use the Microchip ZigBee PRO Stack can customize the Default_SAS by placing the appropriate values in the `zigbee.def` file prior to compiling and building their device application. When the application starts, it copies the values from the `zigbee.def` file into the Default_SAS memory location of nonvolatile storage.

In practice, the Default_SAS initially represents what is traditionally referred to as the “factory default settings”. Parameters such as the device's MAC address, the preconfigured security keys to be used, etc., are all logical candidates to be included in the Default_SAS.

However, because each device is destined to be deployed into a wireless network, whose operating environment is unknown at manufacturing time, the SAS mechanism must be made flexible such that the device can be deployed into any network that the installer specified. SAS_1 and SAS_2 provides this flexibility.

In practice, the Default_SAS can be used to hold the factory default settings. SAS_1 and SAS_2 can be used by an installer to program the application and deployment specific parameters that will be used by the device in the field.

The following steps describe how a product's application code, the Microchip ZigBee PRO Stack, and the SAS infrastructure may be used together to ensure that the device starts up and join or rejoin any installer's desired network.

1. Compile and build the application of the ZigBee devices using the parameters in the `zigbee.def` file. The Default_SAS block of nonvolatile memory, will then be initialized with many of the parameters whose values originate from the `zigbee.def` file definitions. Parameters such as the Extended PANID, and Channel Mask in the Default_SAS should be configured to direct the device to a specific channel, or set of potential channels, from which the Coordinator can establish the network and from where other devices may join.

2. After the Coordinator has established the network, the application code for each device can use the appropriate API functions that are described in this section to update the appropriate SAS_1 and SAS_2 block(s), to the parameters that will be used when the device is deployed in their “permanent” network.
3. The devices, now with the deployable SAS_1 and SAS_2 in nonvolatile memory, can be restarted using SAS_1 or SAS_2. This uses the Default_SAS to get the devices configured, after which the devices are restarted using either SAS_1 or SAS_2, while retaining the factory settings in Default_SAS.

Interface functions to create instances of any of the SAS (Default, SAS_1 or SAS_2) are provided.

The Microchip ZigBee PRO Stack provides the following Application Programming Interface (API) functions to program the SAS.

```
/* *****  
 * Function:          void Initdefault_SAS ()  
 *  
 * PreCondition:      None  
 *  
 * Input:             None  
 *  
 * Output:            default_SAS initialized in Nonvolatile Memory  
 *  
 * Side Effects:      Old Default_SAS is overwritten with new one  
 *  
 * Overview:          This function initializes the default_SAS with factory default settings  
 * *****/  
void Initdefault_SAS (void)
```

```
Function SaveSAS() is used to save data into a particular SAS block  
/* *****  
 * Function:          void SaveSAS (STARTUP_ATTRIBUTE_SET* ptr_current_SAS, unsigned char index)  
 *  
 * PreCondition:      None  
 *  
 * Input:             *current_SAS - pointer to the data structure that holds the SAS parameters  
 *                   index - which SAS entry to write (Default_SAS (0xff), SAS_1(0x00), SAS_2 (0x01)  
 *  
 * Output:            None  
 *  
 * Side Effects:      Old SAS will be overwritten with new values into NVM  
 *  
 * Overview:          This API is used by application to store a new set of SAS at a particular index  
 *  
 * Note:              This function does not free the pointer ptr_current_SAS  
 * *****/  
void SaveSAS (STARTUP_ATTRIBUTE_SET* ptr_current_SAS, BYTE options, BYTE index)
```

Function that is used by the application to determine which one of the three SAS block is currently the active and in use SAS.

```

/*****
* Function:          void GetActiveSASIndex (BYTE &index)
*
* PreCondition:      None
*
* Input:             index - address of the variable to hold the active index
*
* Output:            None
*
* Side Effects:      The 'index' variable is update to the either 0xff, 0x01 or 0x02
*
* Overview:          This API will be used to get the index of the currently active SAS
*
* Note:              None
*****/
GetActiveSASIndex(&index)

```

The function that is used to select between Default_SAS, SAS_1 and SAS_2 to be used as the active and in use SAS block.

```

/*****
* Function:          BYTE SetActiveIndex (BYTE index)
*
* PreCondition:      None
*
* Input:             index - 0xff, 0x01 or 0x02
*
* Output:            None
*
* Side Effects:      None
*
* Overview:          This API will be used to set "index" parameter to
*                    "activeSASIndex" which determines the application's currently
*                    active SAS to use
*
* Note:              None
*****/
void SetActiveIndex (BYTE index)

```

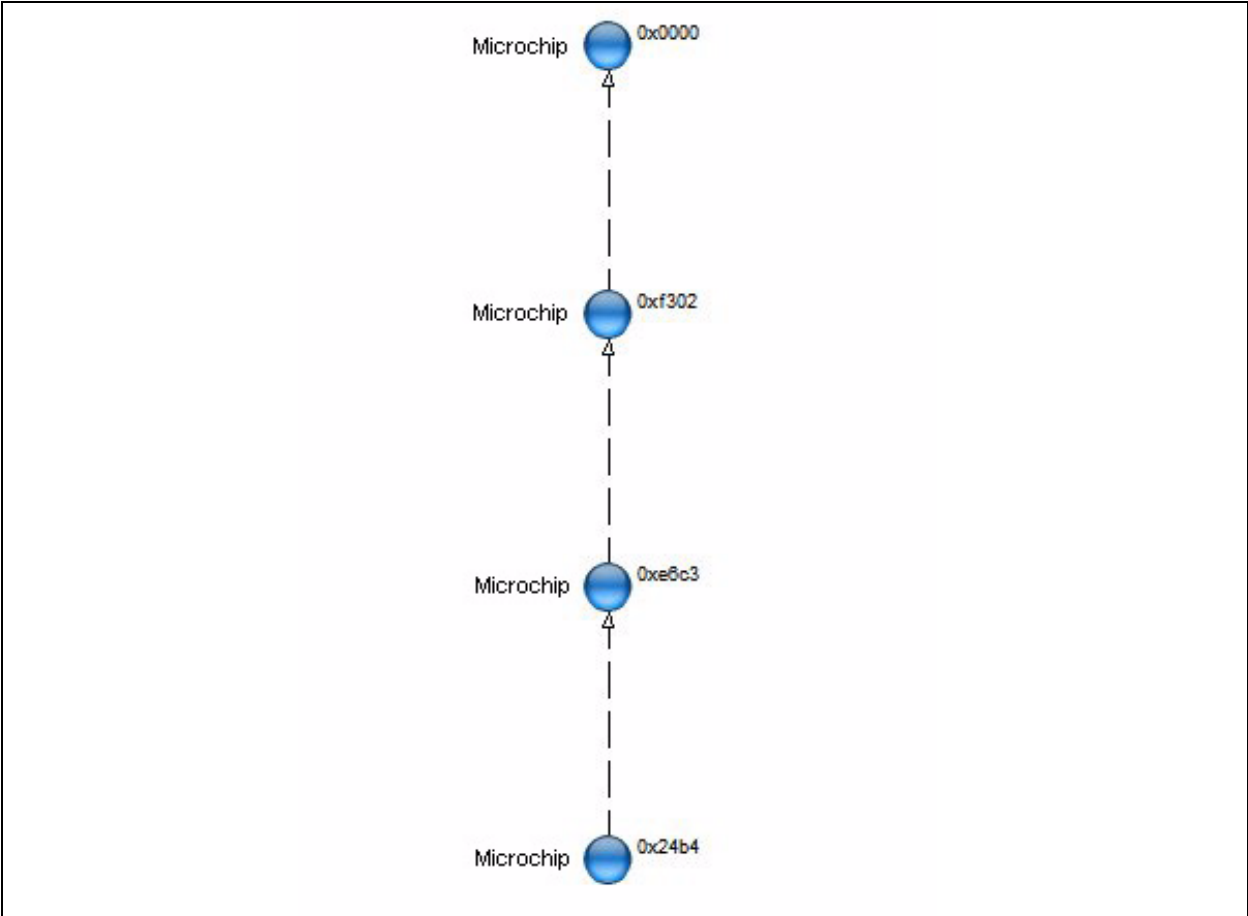
Collectively, these APIs provide the user with the interfaces needed to create, select and use the SAS mechanism within an application.

The benefit of having the three SAS is that the device may be started using one of the SAS, and the others can then be loaded with new parameters, after which the operation of the device may be switched to a different SAS.

Many-to-One Routing

A ZigBee PRO Concentrator device is one that is able to store the routes to other devices in network. It stores these routes in its **Route Record Table**. Refer to Figure 8 for an example.

FIGURE 8: SOURCE ROUTING ILLUSTRATION



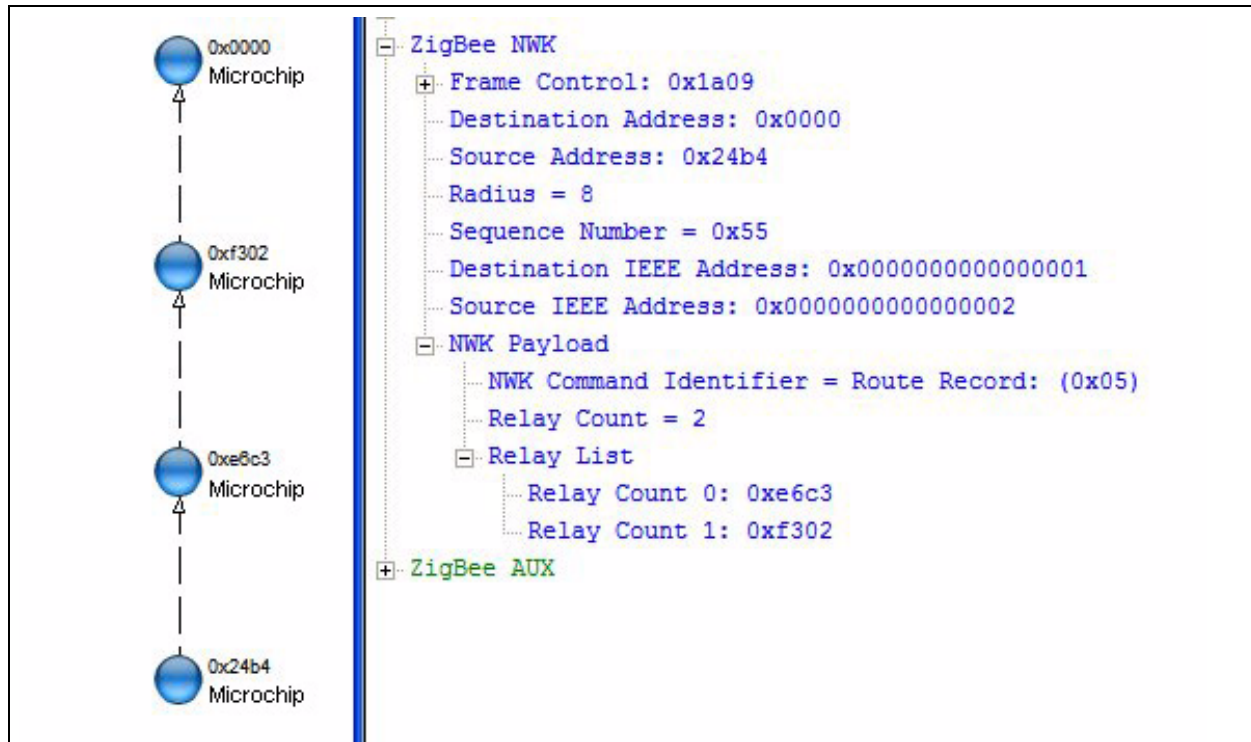
Assuming the Coordinator with network address 0x0000 is the Concentrator device, the route from itself (0x0000) to device 0x24b4 would be the list of network addresses 0xf302 and 0xe63c. This list is stored in the Concentrator's Route Record Table. The structure of the Route Record Table for this specific example is shown in Table 3.

TABLE 3: ROUTE RECORD TABLE ENTRY FORMAT

Field Name	Range/Value	Comment
Target Network Address	0x24b4	The destination address for this route record.
Relay Count	2	The relay count of the number of nodes from destination address to Concentrator.
Path List	0xe63c 0xf302	Set of network address that represents, in order, the route from Concentrator to the Destination.

Figure 9 shows an actual packet Sniffer capture of the route record command that was transmitted to the Concentrator from the illustrative example been discussed.

FIGURE 9: ROUTE RECORD COMMAND FRAME CAPTURE



A ZigBee PRO Concentrator device triggers the process of gathering the routes from the other network devices to itself by using a special route discovery command. That special command is referred to as a many-to-one route discovery. The devices will eventually send back their routes to the Concentrator as a result of receiving a many-to-one route discovery request, but they will not do so immediately to prevent flooding the network with route replies.

In summary, a ZigBee PRO Concentrator device has the capability to perform two operations:

- The first is the ability to establish routes to itself from all the routers and coordinator within a given radius in its network.
- Secondly, a Concentrator is capable of storing and managing those routes in a Route Record Table.

There can be several Concentrator devices in one network. When a device receives a many-to-one route discovery request from a concentrator, it performs three operations:

- First, it creates a routing table entry in its routing table and records the next hop route back to the Concentrator.
- Second, it stores a set of flags in its routing table that indicates it received a many-to-one route request from a Concentrator device in the network.
- Third, when it subsequently needs to send any data to the Concentrator, it will first send a Route Record Command. The purpose of the Route Record Command is to document the route (list of nodes) from itself back to the concentrator. The concentrator will store this list in its Route Record Table.

Therefore, by sending out a single many-to-one route request, Concentrator devices are able to build efficient routes to the devices in the network and can later use these routes to transmit data without first having to first issue route requests.

Source Routing

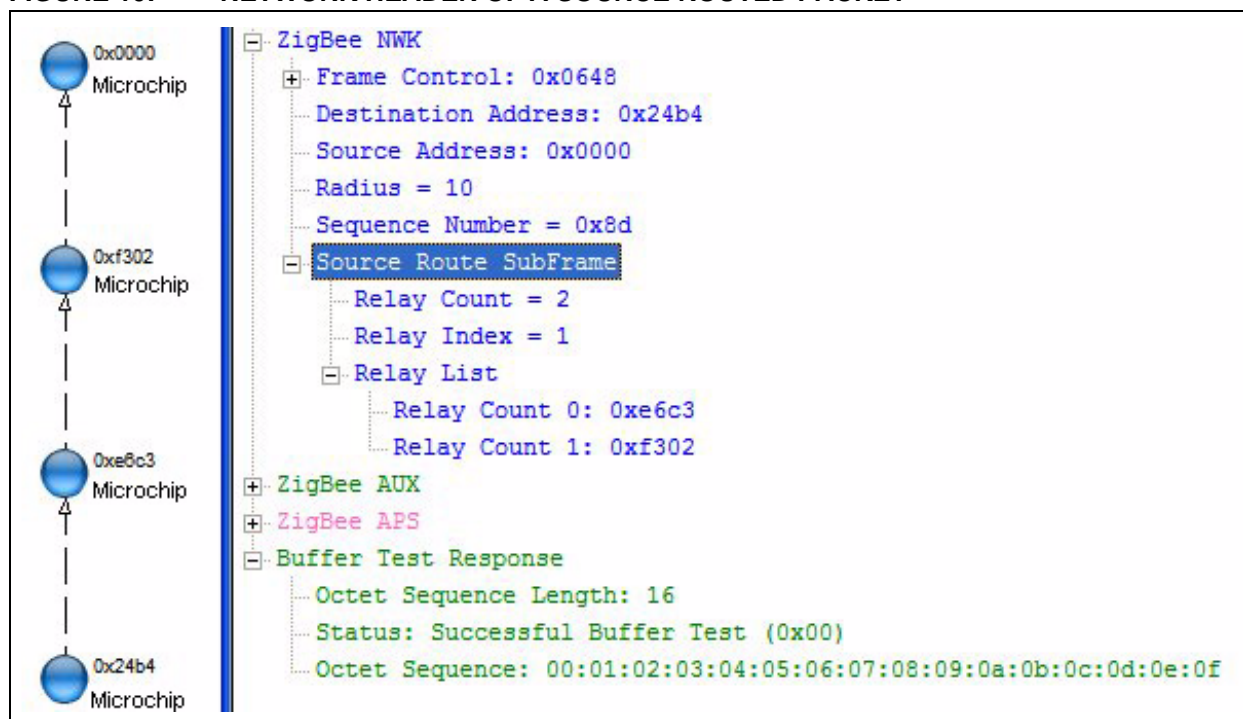
Whenever a Concentrator device needs to send a data packet to another device, it must determine the most effective path by which to send the packet. One of those choices is to use source routing. The source routing process works as follows:

If the destination device address is present in its Route Record Table, it will extract the list of nodes that comprises the path to that device. A packet will then be created that includes the list of nodes in the network header. The `source_routed_packet` indicator flag will be set in the network header as well. This flag is used to alert other devices along the path from the Concen-

trator to the destination device, to extract the `next_hop` device address from the network header, and to use that address to relay to packet until it reaches its final destination.

In addition to the list of nodes that comprise the path from Concentrator to destination device, there is also an index pointer that tells each successive node that receives a source routed packet, which is the next address in the list to use as the `next_hop` address. This index is decremented by one at each hop, and reaches zero on the final hop. See Figure 10 for a packet sniffer capture of the source routed packet from the example being described.

FIGURE 10: NETWORK HEADER OF A SOURCE ROUTED PACKET



High Security Application Master Key Exchange

The Microchip ZigBee PRO Feature Set Stack supports the high security mode of operation. To accomplish this, the stack supports three types of keys:

- The Network Key – used to secure packets at the ZigBee NWK level, and is used by all the devices in the network.
- The Application Link Key – used to secure packets at the ZigBee APS level, and is used by all the devices in the network.
- The End-to-End Application Master Key – used to secure the communication between a pair of devices. This key is unique to each pair of devices, and is not known by any of the other devices in the network, except the device pair that established the key and the Trust Center that created it.

The Microchip ZigBee PRO Feature Set Stack is configurable to use either preconfigured or non-preconfigured network and application link keys. In the preconfigured mode, the keys are defined as part of the stack's `zigbee.def` files are then compiled and linked into the stack's data section when each ZigBee device type is built. These keys are then copied into nonvolatile storage at device initialization time.

In the non preconfigured key mode, the keys are transmitted from the Trust Center to the devices as they join the network.

The high security end-to-end application master key mode operation works as follows:

Any pair (2) of devices wishing to use high security to communicate exclusively with each other must first get an application master key from the Trust Center. They acquire the application master key by sending a ZigBee defined `APSME_REQUEST_KEY_Request` primitive to the Trust Center. This request must be secured. Both devices must send their individual request to the Trust Center at approximately the same time, with each device stating the other device as its partner in the transaction.

Upon receiving the simultaneous requests, the Trust Center will then calculate a unique end-to-end application master key, and securely transport it to the two partnering devices.

After receiving their Master key, subsequent communications between the two partnering devices will use their unique Master Key, which is known only by themselves and the Trust Center. No other device on the network can eavesdrop on their communication.

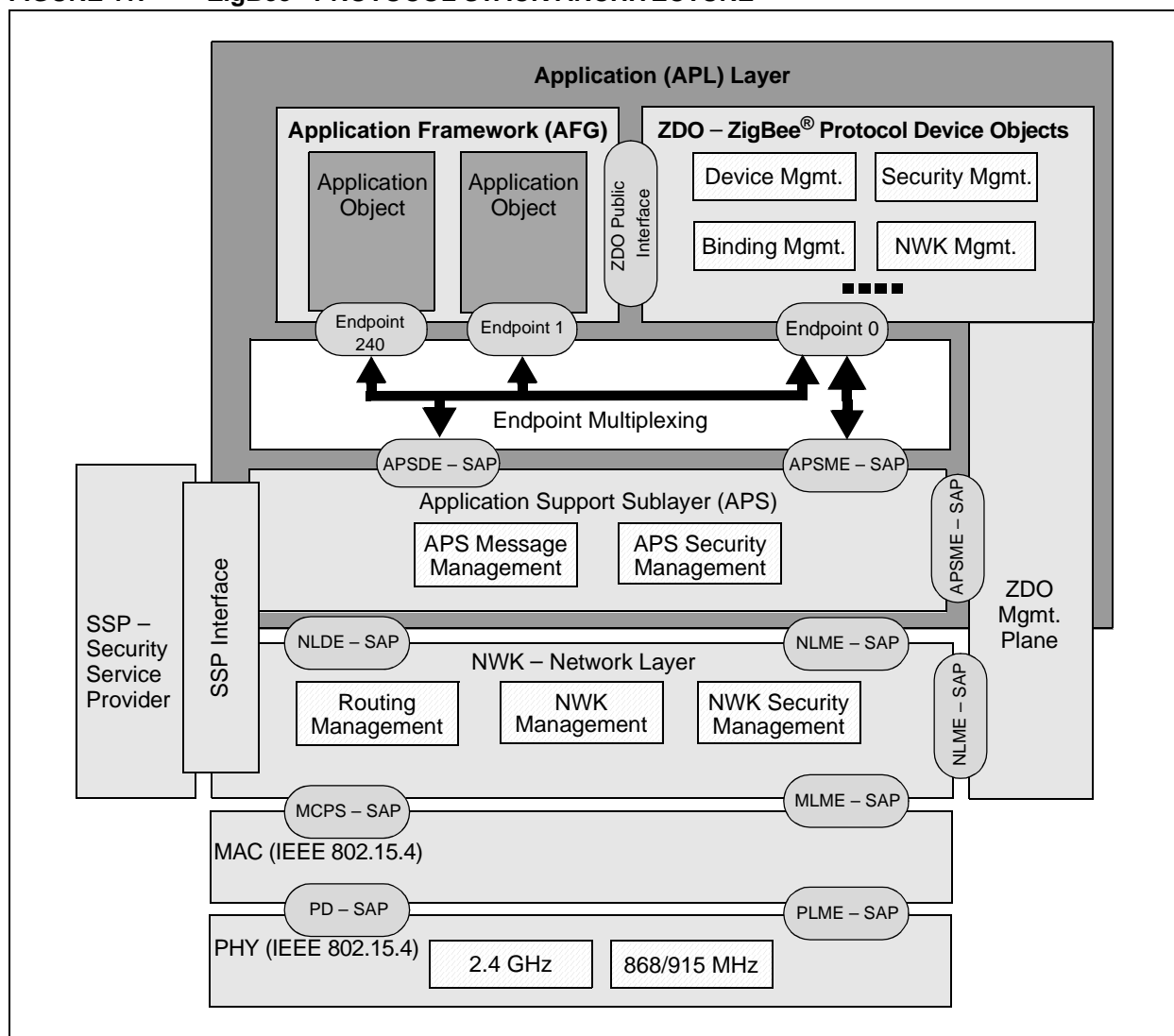
In the current version of the Microchip ZigBee PRO Feature Set Stack, eight such end-to-end application master keys can be supported by the Trust Center, but this a configurable parameter.

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 can use either internal Flash program memory, or external Nonvolatile Memory (NVM) to store a number of persistent stack parameters across resets of a device. The Designer has a choice of which type of NVM to use. The current default stack operation is an external EEPROM used on the Explorer 16 platform.

The Microchip Stack is designed to follow the ZigBee protocol and IEEE 802.15.4-2003 specifications, with each layer in its own source file. Refer to Figure 11 for a diagram of the ZigBee stack. 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 11: 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 MRF24J40 RF transceiver with required external components
- An antenna – may be a PCB trace antenna or monopole antenna
- External serial EEPROM (optional)

As shown in Figure 12, the microcontroller connects to the MRF24J40 transceiver via the SPI bus and a few discrete control signals. The microcontroller is the SPI master and the MRF24J40 transceiver acts as a slave. The

controller implements the IEEE 802.15.4 Medium Access Control (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 PIC24F microcontroller family to drive the RF transceiver in the default implementation (provided in the Explorer 16 platform) are listed in Table 4. 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.

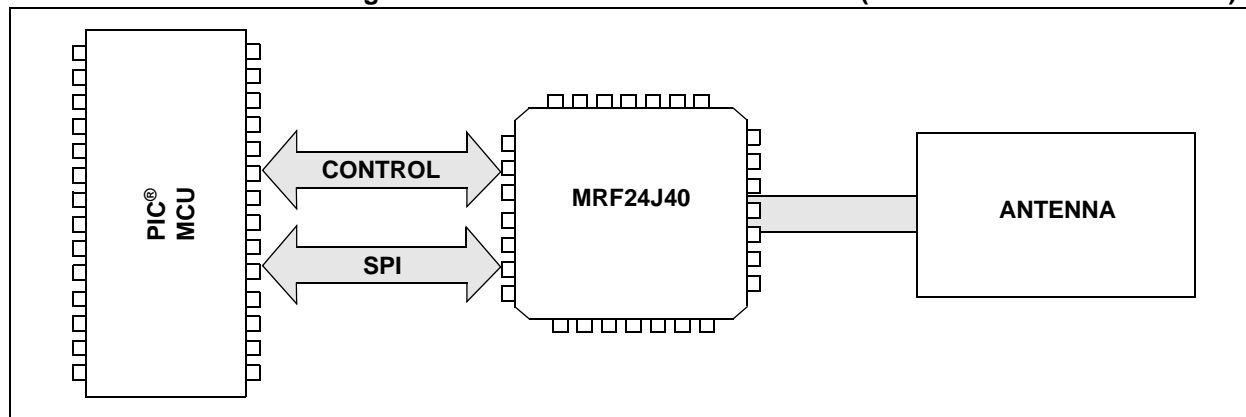
TABLE 4: PIC24FJ128GA010 MICROCONTROLLER RESOURCES REQUIRED BY THE ZigBee® PRO PROTOCOL STACK

PIC24F Resource	Description	MRF24J40
INT1	Used to accept interrupts from MRF24J40 transceiver	INT
TMR2	Used for symbol timer	—
RB2	Chip selection	\overline{CS}
RG3	Wake-up pin	WAKE
RG2	Transceiver Reset	\overline{RESET}
RF6	SPI SCK	SCK
RF7	SPI SDI	SDO
RF8	SPI SDO	SDI

The Microchip reference design for the 802.15.4 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*” (DS51524) for more information (see “References”).

The Microchip RF transceiver is a 3.3V device. Depending on the requirements, the designer 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, care must be taken to operate the transceiver within the specified voltage range.

FIGURE 12: TYPICAL ZigBee® PROTOCOL NODE HARDWARE (CONTROL SIGNALS ADDED)



INSTALLING THE MICROCHIP ZigBee PRO STACK

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. 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 ZigBeePRO subdirectory in the C:\Microchip Solutions directory of your computer.

4. 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 5 shows the directory structure.

TABLE 5: SOURCE FILE DIRECTORY STRUCTURE

Directory Name	Contents
Documents	Microchip Stack for the ZigBee® Protocol documentation.
Microchip	Microchip Stack for the ZigBee Protocol source files. Files contained in this directory should not be changed.
Sample Applications	Source code for a demonstration ZigBee protocol Coordinator, Router and End Device. These files can be changed to create a custom application.

The stack files contain logic for all supported device types of ZigBee protocol applications; however, only one set of logic will be enabled based on the preprocessor definitions in the zigbee.def. A designer 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, Microchip\Common, and Microchip\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

Versions 2.0.PRO.2.0 of the Microchip Stack include three primary demonstration applications:

- **Coordinator** – Demonstrates a typical ZigBee protocol coordinator device application.
- **Router** – Demonstrates a typical ZigBee protocol router device application.
- **End Device** – Demonstrates a typical ZigBee protocol RFD device application.

Demo Application Features

The demo applications implement the following features:

- Targeted for use with the Explorer 16 demo board
- RS-232 terminal output to view device operation, as well as a menu system to send commands to the operating devices
- Sending and receiving data
- Operates a simple multicast addressing application
- Simulates frequency agility
- Sending fragmented data packets

One Explorer 16 Demonstration Board must be programmed as a ZigBee protocol coordinator using the `Coordinator` project. A second board must be programmed as a full function device using the `Router` project. If more Explorer 16 Demonstration Boards are available, they can be programmed either as more end devices or as routers using the appropriate project.

Demo Applications Project and Source Files

Table 6 through Table 10 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 6: MICROCHIP STACK SOURCE FILES IN `ZigBeeStack` SUBDIRECTORY

File Name	Description
<code>SymbolTime.c, .h</code>	Performs timing functions for the Microchip Stack for the ZigBee® protocol.
<code>zAPL.h</code>	Application level interface header file for the stack. This is the only file that the application code needs to include.
<code>zAPS.c, .h</code>	ZigBee protocol APS layer.
<code>zTest.h</code>	ZigBee ZCP profile information. This changes depending on the profile.
<code>zigbee.h</code>	Generic ZigBee protocol constants.
<code>ZigBeeTasks.c, .h</code>	Directs program flow through the stack layers.
<code>zMAC.h</code>	Generic IEEE 802.15.4™ MAC layer header file.
<code>zMAC_MRF24J40.c, .h</code>	IEEE 802.15.4 MAC layer for the Microchip MRF24J40 transceiver.
<code>zNVM.c, .h</code>	Performs nonvolatile memory storage functions.
<code>zNWK.c, .h</code>	ZigBee protocol NWK layer.
<code>zPHY.h</code>	Generic IEEE 802.15.4 PHY layer header file.
<code>zPHY_MRF24J40.c, .h</code>	IEEE 802.15.4 PHY layer for the Microchip MRF24J40 transceiver.
<code>zSecurity.h</code>	ZigBee protocol security layer header file.
<code>zSecurity_MRF24J40.c, .h</code>	ZigBee protocol security layer for the Microchip MRF24J40 transceiver.
<code>zZDO.c, .h</code>	ZigBee protocol's ZDO (ZDP) layer.
<code>zStack_Configuration.h</code>	ZigBee PRO Stack information.
<code>zStack_Profile.h</code>	ZigBee PRO ZCP profile information.

TABLE 7: 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 8: ZigBee® PROTOCOL COORDINATOR DEMO IN Sample Applications DIRECTORY

File Name	Description
Coordinator.c	Main application source file.
Coordinator.mcp	Project file.
Coordinator.mcw	Work space file.
myZigBee.c	Contains application-specific information.
zigbee.def	Contains application-specific information.
Coordinator.h	Main application header file.
zCoordInitialization.c	Main application initialization code.

TABLE 9: ZigBee® PROTOCOL ROUTER DEMO IN Sample Applications DIRECTORY

File Name	Description
Router.c	Main application source file.
Router.mcp	Project file.
Router.mcw	Work space file.
myZigBee.c	Contains application-specific information.
zigbee.def	Contains application-specific information.
Router.h	Main application header file.
zRouterInitialization.c	Main application initialization code.

TABLE 10: ZigBee® PROTOCOL END DEVICE DEMO IN Sample Applications DIRECTORY

File Name	Description
EndDevice.c	Main application source file.
EndDevice.mcp	Project file.
EndDevice.mcw	Work space file.
myZigBee.c	Contains application-specific information.
zigbee.def	Contains application-specific information.
EndDevice.h	Main application header file.
zEndDeviceInitialization.c	Main application initialization code.

Demonstrating Sample Applications

Please consult the ZigBeePROQuickStartGuide.pdf and ZigBeePROQuickStartGuide.chm documents in the stack install directory for a complete guide on how to run the sample applications that came with this version of the stack.

USING THE MICROCHIP STACK FOR THE ZigBee PROTOCOL

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

1. Obtain an Organizationally Unique Identifier (OUI).
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 application note.
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. Make appropriate changes to the ZigBee.def file as needed based on your specific hardware requirements.
5. Use the sample application that came with the stack as a guide in creating a new application.
6. Add code in the new application, including extra initialization, any required ZDO response handling, endpoint message reception and transmission, and any non-protocol processing and interrupt handling.

The ZigBee Stack Nonvolatile Storage

The ZigBee protocol requires many parameters and tables be storage in Nonvolatile Memory (NVM) so that information critical to the device's deployment and operation may be recovered across device resets or power failures. The Microchip ZigBee PRO Stack utilizes the external 25LC256 (32K x 8) serial EEPROM that is available on the Explorer 16 platform for this purpose. It interfaces with the PIC24 microcontroller via the SPI interface. Other types of NVM devices may be used by the application developer, provided that the appropriate support driver utility is used.

For Microchip's ZigBee PRO Stack, Table 11 shows the parameters and tables that are stored in the external EEPROM, as the default setting for the stack. This information is used by the stack to ensure that all the ZigBee PRO Feature Set requirements for persistent data usage is met, and to ensure that all ZigBee devices operate correctly on the network even across resets and power failures.

TABLE 11: ZigBee® PRO STACK PARAMETERS AND TABLES STORED IN EEPROM

Parameter or Table Name	Total Size in Bytes (Including Support Data Structures)	Size Adjustable in Stack
MAC Address	10	No
Binding Table	200	Yes
Group Table	256	Yes
Neighbor Table	925	Yes
Routing Table	80	Yes
Node Descriptor	15	No
Power Descriptor	2	No
Simple Descriptor	12	No
Persistence PIB	48	No
APS Address Map Table	202	Yes
Network Keys	37	No
Link Keys	232	Yes
Start Attribute Sets (2)	190	No
Total	2213	

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. Refer to the `ZigBee.def` file for the labels created for the supported transceivers.

Before the stack can be used, it must be initialized. Interrupts must then be enabled. 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`. Care needs to be taken when accessing this structure to avoid overwriting 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 sample application 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();  
  
    /* Trigger the current ZigBee primitive */  
    StackStatus = ZigBeeTasks( &currentPrimitive );  
  
    /* Process the next ZigBee Primitive */  
    ZIGAPLProcessZigBeePrimitives();  
  
    /* Check if the user has any menu inputs */  
    if ( ConsoleIsGetReady() )  
    {  
        ZIGAPLProcessMenu();  
    }  
  
    /* Check if the user has activated a pushbutton */  
    ZIGAPLProcessPushButtons();  
  
    /* do any non ZigBee related tasks in this function */  
    ZIGAPLProcessNONZigBeeTasks();  
}
```

Forming or Joining a Network

The process of forming or joining a network is shown in the sample applications. 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:
{
    ProcessReceivedPacket();
}
break;

void ProcessReceivedPacket(void)
{
    BYTE          data;
    BYTE          sequenceNumber = 0;

    currentPrimitive = NO_PRIMITIVE;

    switch (params.APSDE_DATA_indication.DstEndpoint)
    {
        /* Process anything sent to ZDO i.e EndPoint Zero here */
        case EP_ZDO:
            //Handle ZDO responses here...see sample application for example

            break;

        /******
        // Place a case here for each user defined endpoints.
        //*****
        case MY_APP_USER_EP:
        {
            /* This EndPoint is to demonstrate the SendPacket
            * and GetPacket functions. Note that the EndPoint
            * and clusterID must match below for the packet to be
            * sent up. This is for demo, change it if you'd like
            */

            switch(params.APSDE_DATA_indication.ClusterId.Val)
            {
                /* Showing how to support Endpoint & Cluster pairs */
                case MY_APP_CLUSTER:
                case TRANSMIT_COUNTED_PACKETS_CLUSTER:
                {
                    // See example application for example code

                    break;
                }
            }
        }

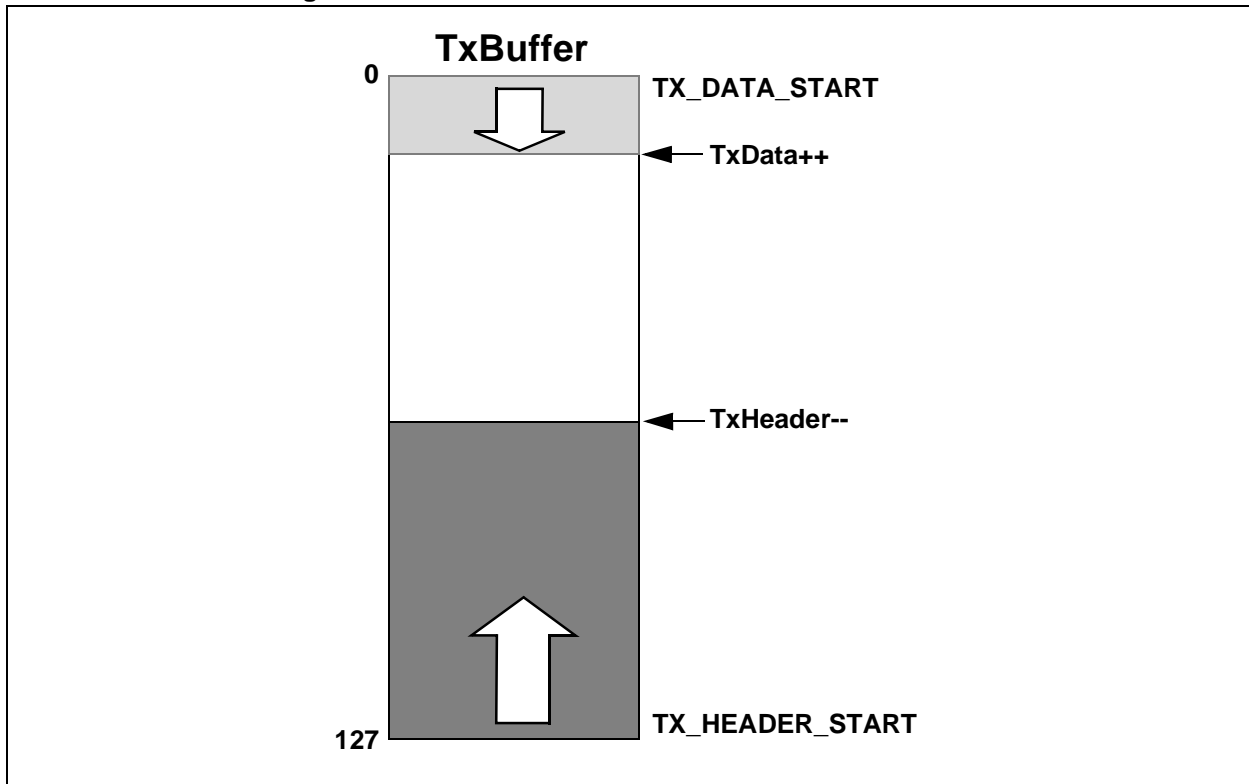
        /* Discard packet that is received so that buffer can be reused */
        APLDiscardRx();
    }
}
```


The Stack Transmit Buffer

The Microchip Stack for the ZigBee PRO Feature Set allows one outgoing message in the application layer at a time. That message is placed in the Transmit Buffer, referred to in the stack code as TxBuffer, and passed down through the ZigBee protocols' architectural lay-

ers, to the transceiver, where it is eventually transmitted over the air. The size of the TxBuffer, as per 802.15.4 specifications, is fixed at 127 bytes. Figure 13 shows a diagram of the TxBuffer, and will be used to illustrate how a designer's application may use this buffer to transmit messages over the air.

FIGURE 13: THE ZigBee® PRO STACK TRANSMIT BUFFER



The operating dimensions of TxBuffer are governed the following C language parameter definitions:

```
#define TX_BUFFER_SIZE 128
#define TX_DATA_START 0
#define TX_HEADER_START (TX_BUFFER_SIZE-1)
BYTE TxBuffer[TX_BUFFER_SIZE]; /* the
transmit buffer */
BYTE TxData = TX_DATA_START; /* the
Data section */
BYTE TxHeader = TX_HEADER_START; /* the
Header section */
```

From an architectural perspective, the TxBuffer has two distinct sections: a Data section and a Header section. The Data section starts at address offset 0 (TX_DATA_START) and grows toward higher offsets, 1, 2, 3 etc., The Header section starts at address offset 127 ((TX_BUFFER_SIZE-1), and decreases toward lower offsets: 126, 125, 124, etc.

In terms of the ZigBee protocol, a messages' payload data is placed in the Data section, while the message's header information is placed in the Header section. The TxData and TxHeader parameters are simply used by the stack as indexes into the Data and Header sections of the TxBuffer, respectively.

The following code fragment shows how a user's application code sets up the transmit buffer to send out a ZigBee defined message (i.e. a BUFFER_TEST_REQUEST) to ask the device with a network address of 0x7eaf for sixteen (0x10) bytes of data.

EXAMPLE 3: REQUESTING DATA FROM ANOTHER DEVICE

```
outGoingPacket[0] = 0x10; /* requesting 16-bytes from device 0x7eaf */
destinationAddress.v[1] = 0x7e;
destinationAddress.v[0] = 0xaf;

/* See the help file for complete description of this function */
ZIGAPSSendPacket(outGoingPacket,
    1, /* payload is only 1-byte long */
    BUFFER_TEST_REQUEST_CLUSTER,
    MY_PROFILE_ID,
    EP_DEFAULT,
    APS_ADDRESS_16_BIT,
    destinationAddress,
#ifdef I_SUPPORT_SECURITY
    TRUE);
#else
    FALSE);
#endif
```

Sending Messages

The Microchip Stack for the ZigBee Protocol allows one outgoing message in the application layer at a time. Messages are sent by calling the `ZIGAPSSendPacket()` function.

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 4 shows how to send a unicast message to a specific device of interest. The following should be noted:

- Each APS frame must be directed to a particular end point and Cluster within that Profile.
- Load up the outgoing packet with the data to be sent.
- Direct message to a specific endpoint (Applet) on the target device.

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, *apsMaxFrameRetries* times.

EXAMPLE 4: SENDING AN OUTGOING MESSAGE

```
/* Length of packet is first byte */
outGoingPacket[0] = packetLen;

/* Load the payload buffer with the data to send */
for(i = 0; i < packetLen; i++)
{
    outGoingPacket[i+1] = i;
}

destinationAddress.v[1] = 0x7e;
destinationAddress.v[0] = 0xaf;

ZIGAPSSendPacket(outGoingPacket,
                 packetLen+1, /* length in byte[0] + payload in slots 1 onwards */
                 TRANSMIT_COUNTED_PACKETS_CLUSTER,
                 MY_PROFILE_ID,
                 MY_APP_USER_EP,
                 APS_ADDRESS_16_BIT,
                 destinationAddress,
                 #ifdef I_SUPPORT_SECURITY
                     TRUE);
                 #else
                     FALSE);
                 #endif
```

Secure Transmission

The Microchip Stack for the ZigBee Protocol supports all seven security modes that are defined in the 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 any way 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 12.

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 pre-configured 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 13.

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 12: ZigBee® PROTOCOL 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 13: ZigBee® PROTOCOL 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.

Once the security mode has been defined, sending the secured packet is straightforward; only one modification is required in the application code. Example 5 shows the exact same code as in Example 4, with the additional code to enable secure transmission shown in **bold**.

EXAMPLE 5: SENDING A SECURED OUTGOING MESSAGE

```
/* Length of packet is first byte */
outGoingPacket[0] = packetLen;

/* Load the payload buffer with the data to send */
for(i = 0; i < packetLen; i++)
{
    outGoingPacket[i+1] = i;
}

destinationAddress.v[1] = 0x7e;
destinationAddress.v[0] = 0xaf;

ZIGAPSSendPacket(outGoingPacket,
                packetLen+1, /* length in byte[0] + payload in slots 1 onwards */
                TRANSMIT_COUNTED_PACKETS_CLUSTER,
                MY_PROFILE_ID,
                MY_APP_USER_EP,
                APS_ADDRESS_16_BIT,
                destinationAddress,
                #ifdef I_SUPPORT_SECURITY
                TRUE);
                #else
                FALSE);
                #endif
```

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 14 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 15 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 14: 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 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.
APSME_ADD_GROUP_request	APSME_ADD_GROUP_confirm	Request membership in particular group to an endpoint. Can be used only on devices that support multicast addressing.
APSME_REMOVE_GROUP_request	APSME_REMOVE_GROUP_confirm	Remove membership in particular group from an endpoint. Can be used only on devices that support multicast addressing.
APSME_REMOVE_ALL_GROUPS_request	APSME_REMOVE_ALL_GROUPS_confirm	Remove membership in all groups from an endpoint. Can be used only on devices that support multicast addressing.
NETWORK_ROUTE_DISCOVERY_request	NETWORK_ROUTE_DISCOVERY_confirm	Initiate route discovery to another device. ZigBee protocol Coordinator and Routers only.

TABLE 15: 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 ⁽⁵⁾	X ^(3,5)		
APSME_UNBIND_confirm	X ⁽⁵⁾	X ^(3,5)		
NLME_DIRECT_JOIN_confirm	X ⁽⁵⁾	X ⁽⁴⁾		
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	
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: Required if binding is supported.

4: Required if application will issue an NLME_DIRECT_JOIN_request.

5: 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 clean 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 protocol devices other than the Coordinator usually invoke NLME_NETWORK_DISCOVERY_request to find the current available networks before deciding which network to join. The primitive,

NLME_NETWORK_DISCOVERY_confirm, returns a link list of the available networks for the user to choose from. Upon 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 Status Flags

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

TABLE 16: STACK STATUS FLAGS

Flag	Description
<code>bTxFIFOInUse</code>	Indicates that the Stack is currently in the process of transmitting an outgoing message. Use the macros, <code>ZigBeeReady()</code> to check, and <code>ZigBeeBlockTx()</code> to set, this flag.
<code>bRxBufferOverflow</code>	Indicates that the receive buffer has overflowed and messages have been dropped. Must be cleared by the application.
<code>bHasBackgroundTasks</code>	Updated by <code>ZigBeeTasks()</code> . Indicates if the Stack still has background tasks in progress.
<code>bNetworkFormed</code>	ZigBee® protocol coordinator only. Indicates that the device has successfully formed a network.
<code>bTryingToFormNetwork</code>	ZigBee protocol coordinator only. Indicates that the device is in the process of trying to form a network.
<code>bNetworkJoined</code>	ZigBee protocol routers and end devices. Indicates that the device has successfully joined a network.
<code>bTryingToJoinNetwork</code>	ZigBee protocol routers and end devices. Indicates that the device is in the process of trying to join a network.
<code>bTryOrphanJoin</code>	ZigBee protocol routers and end devices. Indicates that the device was once part of a network and should try to join as an orphan.
<code>bRequestingData</code>	RFD end devices only. Indicates that the device is in the process of requesting data from its parent.
<code>bDataRequestComplete</code>	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. 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, then it must possess a binding table. The stack requires 5 bytes of nonvolatile memory for each binding table entry. Note that the 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.

MAX NUMBER OF GROUPS

If the device supports Group Addressing, then it must have a Group Table. This parameter governs the maximum number of records that the Group Table will support. The stack requires 22 bytes of nonvolatile memory for each Group Table entry.

MAX END POINTS PER GROUP

If the device supports Group Addressing, then each Group ID can be associated with up to this many endpoints.

MAX NUMBER OF DUPLICATE PACKETS

All devices keep track of each packet they receive. Individual packets are distinguished from each other by their unique sequence number. If two packets are the received that bear the same sequence number, the second packet is tagged as a duplicate and is discarded. The number of packet sequence numbers that are maintained and checked against the latest packet received is governed by the parameter.

DUPLICATE TABLE EXPIRATION

This parameter governs how long a packet sequence number is maintained by the device before the sequence number is discarded. If two packets are received that bear the same sequence number before the first sequence number has expired, the second packet is tagged as a duplicate and discarded. The expiration time interval for the duplicate packet is governed by this parameter.

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 run time. 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 MCU devices with large erase block sizes, such as the PIC24F 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 MCU. If the MAC address is to be programmed into the serial EEPROM during the manufacturing process, 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.
--------------	---

STOCHASTIC_ADDRESSING

This parameter specifies the use of the device stochastic addressing method, instead of the older CSKIP method that was employed in the ZigBee-2006 Stack. The ZigBee PRO ZCP profile mandates the use of stochastic addressing.

SAS_TABLE_SIZE

This parameter specifies how many Startup Attribute Sets (SAS) will be stored in NVM. The default setting is 2. Therefore, in addition to a default SAS, two additional instances are maintained in NVM.

COMMISSIONING

This parameter specifies that a Startup Attribute Set (SAS) will be stored in NVM. By design, space is reserved in NVM for three SAS instances and the API functions that support the reading from and writing to the SAS are included in the builds for each ZigBee device type.

CONCENTRATOR

This ZigBee PRO Feature Set defines the concept of a Concentrator device that is capable of managing many-to-one and source routing. This parameter indicates which device has the built-in infrastructure to support the requirements of a Concentrator device, such as maintaining a Route Record Table.

CONCENTRATOR RADIUS

When a ZigBee PRO Concentrator device collects the routes to other devices in the network, it does so only within a specified maximum range. This parameter specifies the neighborhood range, in radius hops, for which a given concentrator stores routes. In a large network, this parameter can be used to establish the “neighborhood” in which each Concentrator operates.

HIGH CONCENTRATOR

This ZigBee PRO Feature Set defines the concept of a Concentrator device that is capable of managing many-to-one and source routing. This parameter indicates that the device has the built-in infrastructure to support the requirements of a Concentrator and it maintains a Route Record Table.

LOW CONCENTRATOR

This ZigBee PRO Feature Set defines the concept of a Concentrator device that is capable of managing many-to-one and source routing. This parameter indicates that the device has the built-in infrastructure to support the requirements of a Concentrator but does not maintain a Route Record Table. All data requests to this type of Concentrator must be preceded by a Route Record Command.

ROUTE RECORD TABLE SIZE

The ZigBee PRO Feature Set requires that all High Concentrator devices maintain a Route Record Table in order to route messages to target devices using source routing. The system requires 240 bytes of RAM for the table, and the default maximum number of entries that can be held in this table is 60.

ROUTING_TABLE_AGING

The Link Status Command is used to update both the neighbor and routing table entries. If it is determined that a device has moved or is no longer on the network, this parameter is used to decide when its entry in the routing table would be removed.

FRAGMENTATION

The ZigBee PRO Feature Set allows for the transmission of packets that will exceed the 127-byte packet length set forth in the 802.15.4 specifications. This parameter is used to allow the stack to support the chopping up of larger packets into smaller transmittable blocks that are reassembled at the receiver. If this parameter is not set, then fragmentation is disabled.

FREQUENCY_AGILITY

This parameter is used to specify support for the frequency agility feature. If not specified, then the dynamic channel change and notification mechanism within the stack will be disabled.

PANID_CONFLICT

This parameter is used to specify support for the PANID conflict detection and resolution mechanism. If not specified, then PANID conflicts are ignored and notifications are not sent to the channel manager.

PRECONFIGURED_LINK_KEY

This parameter specifies the 16-bit application link security key. This key is used to secure the outgoing packets as well as to decrypt the incoming packets at the application level when the high security mode is used. The Link Key must be present for Coordinators and the device that acts as the trust center.

HEAP SIZE

The Microchip Stack uses dynamic memory allocation for many purposes, including those listed in Table 17. 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, therefore, is very application dependent.

TABLE 17: 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.

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 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.

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 C Compiler for PIC24 MCUs and dsPIC® DSCs, but it can be easily modified to support other compilers.

REFERENCES

- “ZigBee® Protocol Specification”
<http://www.zigbee.org>
- “IEEE 802.15.4™ Specification”
<http://www.ieee.org>

SOURCE CODE

The complete source code, including demo applications, is available from microchipDIRECT.

ANSWERS TO FREQUENTLY ASKED QUESTIONS (FAQs)

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

A: Yes.

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 purchase it from the Microchip web site (www.microchipDIRECT.com).

Q: How do I get target hardware design files?

A: You may download it from the PICDEM™ Z Demonstration Kit 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 two Explorer 16 boards
- Complete source code for the Microchip Stack for the ZigBee Protocol for dsPIC33 and PIC24 branded products (free of charge)
- The MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs
- MPLAB IDE software
- A device debugger and programmer, such as MPLAB ICD 3

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 help 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 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?

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 v2.0-2.6 stack architecture is the same as v2.0.PRO.2.0:

- **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 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 sample applications.

REVISION HISTORY

Rev A Document (04/2009)

Original version of this document.

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.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KEELOQ, KEELOQ logo, MPLAB, PIC, PICmicro, PICSTART, rfPIC, SmartShunt and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.


FilterLab, Hampshire, Linear Active Thermistor, MXDEV, MXLAB, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, nanoWatt XLP, PICkit, PICDEM, PICDEM.net, PICtail, PIC³² logo, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rfLAB, Select Mode, Total Endurance, TSHARC, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2009, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

 Printed on recycled paper.

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 design centers in California and India. 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

Cleveland

Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

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
Harbour 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 - Hong Kong SAR

Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing

Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

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 - Wuhan

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

China - Xiamen

Tel: 86-592-2388138
Fax: 86-592-2388130

China - Xian

Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Zhuhai

Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore

Tel: 91-80-3090-4444
Fax: 91-80-3090-4080

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 - Daegu

Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul

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

Malaysia - Kuala Lumpur

Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang

Tel: 60-4-227-8870
Fax: 60-4-227-4068

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: 886-3-6578-300
Fax: 886-3-6578-370

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

03/26/09