AN965

Microchip Stack for the ZigBee™ Protocol

Author: *Nilesh Rajbharti*
Microchip Technology Inc.

INTRODUCTION

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

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

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

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

The Stack library provided in this application note implements a PHY-independent application interface. As a result, you can easily port your application from one Radio Frequency (RF) transceiver to another without significant changes.

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

ASSUMPTION

This document assumes that you are familiar with the C programming language. This document uses extensive terminology from the ZigBee and IEEE 802.15.4 specification. This document does not discuss the full details of ZigBee specifications. It does provide a brief overview of the ZigBee specification. You are advised to read the ZigBee and IEEE 802.15.4 specifications in detail.

FEATURES

The Microchip Stack for the ZigBee protocol is designed to evolve with the ZigBee wireless protocol specifications. At the time this document was published, version 1.0 of the Stack offered the following features (for the latest features, refer to the source code version log file, `version.log`):

- Based on version 0.8 of ZigBee specifications
- Support for 2.4 GHz frequency band using the Chipcon CC2420 RF transceiver
- Support for Reduced Function Device (RFD) and Coordinator
- Implements nonvolatile storage for neighbor and binding tables in coordinator nodes
- Supports non-slotted star network
- Portable across the majority of the PIC18 family of microcontrollers
- Cooperative multitasking architecture
- RTOS and application independent
- Out-of-box support for Microchip MPLAB® C18 and Hi-Tech PICC-18™ C compilers
- Modular design to easily add or remove specific modules

LIMITATIONS

Version 1.0 of the Microchip Stack contains the following limitations. Please note that MIcrochip is planning to add new features as time progresses. Refer to the source code version log file (`version.log`) for current limitations.

- Not ZigBee protocol-compliant
- No cluster and peer-to-peer network support
- No security and access control capabilities
- No router functionality
- Does not provide standard profiles; however, it contains all necessary primitive functions to create profiles
- Does not support one-to-many bindings

TYPICAL ZigBee NODE HARDWARE

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

- One PIC18F microcontroller with an SPI™ interface
- One RF transceiver (see `version.log` for supported transceivers) with required external components
- An antenna – may be PCB trace antenna or monopole antenna

As shown in Figure 1, the controller connects to the RF transceiver via the SPI bus and a few discrete control signals. The controller acts as an SPI master and the RF transceiver acts as a slave. The controller implements the IEEE 802.15.4 MAC layer and ZigBee protocol layers. It also contains application specific logic. It uses the SPI bus to interact with the RF transceiver. The Microchip Stack provides a fully integrated driver which relieves the main application from managing RF transceiver functions. If you are using a Microchip reference schematic for a ZigBee 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 which case, you will have to modify PHY interface definitions to include the correct pin assignments.

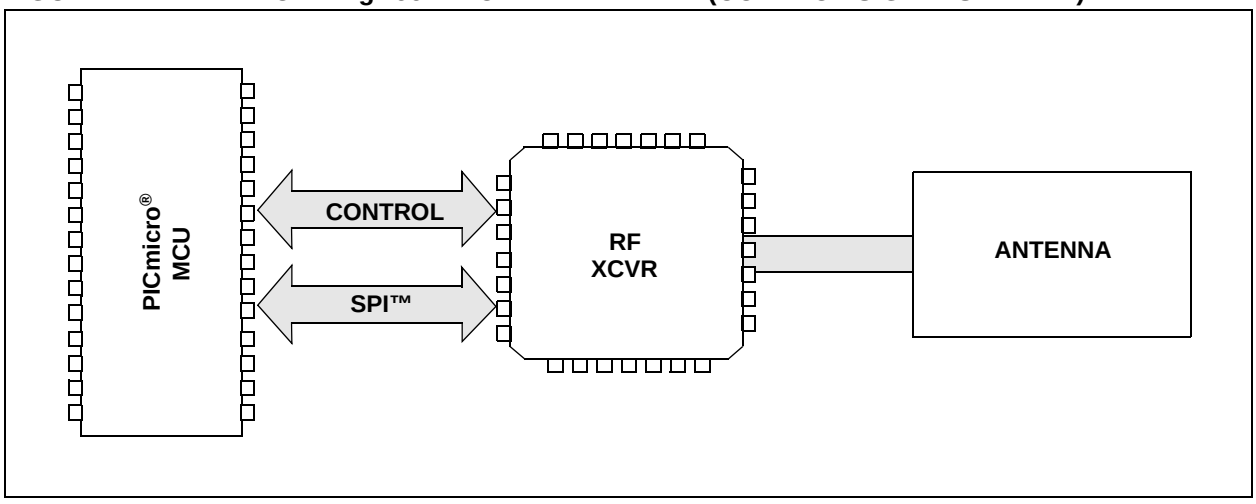
Version 1.0 of the Microchip Stack uses a CC2420 RF transceiver manufactured by Chipcon. The CC2420 implements the 2.4 GHz physical layer along with some of the MAC functions. You may read more about the CC2420 at the Chipcon Web site (see **"References"**).

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

The CC2420 requires a 3.3V voltage supply. The Microchip reference design uses a 3.3V supply for both the controller and the RF transceiver. If required, you may modify this design to use 5V for the controller and 3.3V for the RF transceiver. When using a 5V supply for the controller, a logic level translator circuit to interface to and from the CC2420 must be used. Depending on your requirements, you may either use mains or a battery power supply. Typically, ZigBee coordinator devices would operate on mains power supply and end devices would operate on a battery. When using a battery power supply, you must make sure that you operate the CC2420 within the specified voltage range.

Refer to the *"PICDEM™ Z Demo Kit User's Guide"* for a Microchip reference design for a ZigBee node.

FIGURE 1: TYPICAL ZigBee™ NODE HARDWARE (CONTROL SIGNALS ADDED)



PIC® RESOURCE REQUIREMENTS

The Microchip Stack uses the following I/O pins to interface to the RF transceiver:

TABLE 1: PIC® MCU TO RF TRANSCEIVER INTERFACE

PIC® I/O Pin	RF Transceiver Pin
RB0 (Input)	CC2420: FIFO
RB1 (Input)	CC2420: CCA (Not Used)
RB2 (Input)	CC2420: SFD
RB3 (Input)	CC2420: FIFOP
RC0 (Output)	CC2420: CSn
RC1 (Output)	CC2420: VREG_EN
RC2 (Output)	CC2420: RESET
RC3 (Output)	CC2420: SCK
RC4 (Input)	CC2420: SO
RC5 (Output)	CC2420: SI

For complete program and data memory requirements, refer to the document `version.log` located in the Stack source installation directory.

INSTALLING SOURCE FILES

The complete Microchip Stack source is available for download from the Microchip web site (see **"Source Code"**). The source code is distributed in a single Windows® installation file (`MpZBeeV1.00.00.exe`).

Perform the following steps to complete the installation:

1. Execute the file `MpZBeeV1.00.00.exe`; a Windows installation wizard will guide you through the installation process.
2. Before continuing with the installation, 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" program group. The complete source code will be copied in the `MpZBee\Source` directory in the root drive of your computer.
4. Refer to the file `version.log` for the latest version specific features and limitations.

SOURCE FILE ORGANIZATION

The Microchip Stack consists of multiple source files. Many of the source files are common to all ZigBee applications, while some are specific to certain ZigBee applications only. In addition, the Stack files also include all source files for all demo applications.

To simplify file management and application development, all source files are located in subdirectories under the `Source` directory. The following table shows the directory structure:

TABLE 2: SOURCE FILE DIRECTORY STRUCTURE

Directory Name	Contents
`Stack`	Microchip Stack source files
`DemoCoordApp`	Demo coordinator application source files
`DemoRFDApp`	Demo RFD application source files

Many of the Stack files contain logic for all supported types of ZigBee applications; however, only one set of logic is enabled based on the preprocessor definitions defined in the `zigbee.def` file. You may develop multiple ZigBee node applications using the common set of Stack source files but individual `zigbee.def` files. For example, the `DemoCoordApp` and `DemoRFDApp` node applications have their own `zigbee.def` file in their respective directory. This approach allows the development of multiple applications using common source files and generates unique hex file depending on application-specific options.

This approach requires that when you compile an application project, you provide search paths to include files from both application and Stack source directories. The demo application projects supplied with this application note already include the necessary search path information.

DEMO APPLICATIONS

Version 1.0 of the Microchip Stack includes two demonstration applications:

1. `DemoRFDApp` – to demonstrate a typical ZigBee RFD device application.
2. `DemoCoordApp` – to demonstrate a typical ZigBee coordinator device application.

Demo RFD Application Features

Version 1.0 of the demo RFD application implements the following features:

- Targeted for use with the PICDEM Z demo board
- Demonstrates low-power functionality using system Sleep and Watchdog functionality
- RS-232 terminal driven menu commands to configure various options
- RF transceiver performance test functions via terminal menu commands
- User-configurable simple remote control switch and LED application on one node
- Uses D2 as transmit/receive activity LED
- Demonstrates custom binding interface
- Automatically supports MPLAB C18 and Hi-Tech PICC-18 compilers

Demo Coordinator Application Features

Version 1.0 of the demo coordinator application implements the following features:

- Targeted for use with the PICDEM Z demo board
- RS-232 terminal driven menu commands to configure various options
- RF transceiver performance test functions via terminal menu commands
- Creates a non-slotted star network
- Uses D2 as transmit/receive activity LED
- Demonstrates custom binding interfaces
- Automatically supports MPLAB C18 and Hi-Tech PICC-18 compilers

Building Demo Applications

The demo applications included in this application note can be built using either the Microchip C18 or the Hi-Tech PICC-18 compiler. There are a total of four MPLAB project files – two for each demo application. The first two letters in the name of the MPLAB project file identifies the type of compiler being used. For example, the project file, `MpDemoCoordApp.mcp`, uses the MPLAB C18 compiler, while `HtDemoCoordApp.mcp` uses the Hi-Tech PICC-18 compiler.

In addition to using the PIC18F4620 as a device, all demo application projects also use additional include paths as defined in the “Build Options” of MPLAB® IDE. The demo coordinator project uses “`..\Stack`” and “`..\DemoCoordApp`” and the demo RFD project uses “`..\Stack`” and “`..\DemoRFDApp`” as additional include paths. If you are recreating any of the demo application projects using MPLAB IDE, you must manually set these include paths in the MPLAB “Build Options” dialog box.

Table 3 and Table 4 list the necessary source files needed to build the demo coordinator and demo RFD applications.

TABLE 3: DEMO COORDINATOR APPLICATION PROJECT FILES

Source Files	Directory Name	Purpose
MpDemoCoordApp.mcp HtDemoCoordApp.mcp	DemoCoordApp	Demo coordinator application project file for MPLAB® IDE
DemoZCoordApp.c	DemoCoordApp	Main coordinator application file
zigbee.def	DemoCoordApp	Microchip Stack compile time options file
D1OnCoord.c	DemoCoordApp	Endpoint task for LED D1 – specific to coordinator node
S2OnCoord.c	DemoCoordApp	Endpoint task for switch S2 – specific to coordinator node
18f4620i.lkr	DemoCoordApp	MPLAB C18 linker script file for PIC® microcontroller – not required for Hi-Tech PICC-18™ compiler
Console.c	Stack	RS-232 terminal routines used by demo application only
MSPI.c	Stack	SPI™ master interface
NeighborTable.c	Stack	Coordinator neighbor and binding table logic
SRAlloc.c	Stack	Dynamic memory manager used by coordinator only
Tick.c	Stack	Tick manager used by Stack, available to application
zAPL.c	Stack	ZigBee™ application layer
zAPS.c	Stack	ZigBee application support sublayer
ZDO.c	Stack	ZigBee device object
zMAC.c	Stack	IEEE 802.15.4 MAC layer
zNVM.c	Stack	Nonvolatile memory storage routines
zNWK.c	Stack	ZigBee network layer
zPHYCC2420.c	Stack	CC2420 specific PHY routines
zProfile.c	Stack	ZigBee profile routines

TABLE 4: DEMO RFD APPLICATION PROJECT FILES

Source Files	Directory Name	Purpose
MpDemoRFDApp.mcp HtDemoRFDApp.mcp	DemoRFDApp	Demo RFD application project file for MPLAB® IDE
DemoZRFDApp.c	DemoRFDApp	Main RFD application file
zigbee.def	DemoRFDApp	Microchip Stack compile time options file
D1OnEndDevice.c	DemoRFDApp	Endpoint task for LED D1 – specific to end device
S2OnEndDevice.c	DemoRFDApp	Endpoint task for switch S2 – specific to end device
18f4620i.lkr	DemoCoordApp	MPLAB C18 linker script file for PIC® microcontroller – not required for Hi-Tech PICC-18™ compiler
Console.c	Stack	RS-232 terminal routines used by demo application only
MSPI.c	Stack	SPI™ master interface
Tick.c	Stack	Tick manager used by Stack, available to application
zAPL.c	Stack	ZigBee™ application layer
zAPS.c	Stack	ZigBee application support sublayer
ZDO.c	Stack	ZigBee device object
zMAC.c	Stack	IEEE 802.15.4 MAC layer
zNVM.c	Stack	Nonvolatile memory storage routines
zNWK.c	Stack	ZigBee network layer
zPHYCC2420.c	Stack	CC2420 specific PHY routines
zProfile.c	Stack	ZigBee profile routines

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

1. Make sure that the source files for the Microchip Stack are installed. If not, please refer to **"Installing Source Files"**.
2. Launch MPLAB IDE and open the appropriate project file:

   Source\DemoCoordApp\??DemoCoordApp.mcp
   for the demo coordinator application; or

   Source\DemoRFDApp\??DemoRFDApp.mcp
   for the demo RFD application.

   The exact name of the project file depends on your choice of compiler. Use "Mp*.mcp" for MPLAB C18 and "Ht*.mcp" for the Hi-Tech PICC-18 compiler.
3. Use MPLAB IDE menu commands to build the project. Note that the demo applications projects are created to work correctly when the source files are located in the MpZBee directory in the root directory of the hard drive. If you have moved the source files to another location, you must recreate or modify existing project settings to build. See **"Building Demo Applications"** for more information.
4. The build process should finish successfully. If not, make sure that your MPLAB IDE and compiler are set up properly.

Programming Demo Applications

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

1. Connect MPLAB ICD 2 to the PICDEM Z demo board or your target board.
2. Apply power to the target board.
3. Launch MPLAB IDE.
4. Select the PIC device of your choice (required only if you are importing a hex file previously built).
5. Enable MPLAB ICD 2 as a programmer.
6. If you want to use a previously built hex file, simply import the `DemoCoordApp\MpDemoCoordApp.hex` file or the `DemoRFDApp\MpDemoRFDApp.hex` file. In order to simplify identification of the demo coordinator and demo RFD nodes (if you are using PICDEM Z boards), it is recommended that you program the `MpDemoCoordApp.hex` file into the controller with the "COORD..." label and the `MpDemoRFDApp.hex` file into the controller with the "RFD..." label. If you are programming your custom hardware, make sure that you use some identification method to identify the coordinator and RFD node.
7. If you are rebuilding the hex file, open the appropriate demo project file and follow the build procedure to create the application hex file.
8. Both demo application files contain necessary configuration options required for the PICDEM Z demo board. If you are programming another type of board, make sure that you select the appropriate oscillator mode from the MPLAB ICD 2 configuration settings menu.
9. Select the **Program** menu option from the MPLAB programmer menu to begin programming the target.
10. After a few seconds, you should see the message "`Programming successful`". If not, double check your board and MPLAB ICD 2 connection. Refer to MPLAB on-line help for further assistance.
11. Remove power from the board and disconnect the MPLAB ICD 2 cable from the target board.
12. Reapply power to the board and make sure that the D1 and D2 LEDs are lit. If not, double check your programming steps and repeat, if necessary.

Configuring Demo Applications

If this is the first time you are running either of the two demo applications, you must first assign a unique node ID to each board. A node ID is a unique four digit decimal number to create a unique MAC address. Both demo applications are built using a Microchip Organizational Unique Identifier (OUI) number. These applications use the node ID value to create a unique 64-bit MAC address as required by IEEE 802.15.4 specifications. You may obtain your own OUI number by applying at the following Web address:

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

To configure the demo applications, you would need the following tools:

1. A PC with at least one RS-232 port.
2. A PC-based RS-232 terminal program such as HyperTerminal for Windows® operating system.
3. One DB-9 male-to-female RS-232 cable.
4. Target board(s) with a 9V power supply.

Programming Node ID Value

Perform the following steps for each of the newly programmed demo applications (This procedure assumes that you are using the Microsoft® HyperTerminal program. You may use any terminal program of your choice provided the required port settings are set):

1. Connect the target PICDEM Z board to an available serial port on the computer, using a straight male-to-female DB9 RS-232 cable.
2. Launch HyperTerminal by selecting *Start> Programs>Accessories >Communications*.
3. At the Connection Description dialog box, enter any convenient name for the connection. Click **OK**.
4. At the Connect To dialog box, select the COM port that the PICDEM Z board is connected to. Click **OK**.
5. Configure the serial port connected to the PICDEM Z node with these settings: 19200 bps, 8 data bytes, 1 Stop bit, no parity and no flow control.
6. Click **OK** to initiate the connection.
7. Open the Properties dialog box by selecting *File > Properties*.
8. Select the Settings tab and click **ASCII Setup...**
9. Check “Echo typed characters locally”.
10. Click **OK** to close all open dialog boxes.
11. Apply power to the node while holding the S3 switch, or press and hold both the RESET and S3 switches; then release the RESET switch.

    The configuration menu, as shown in Example 1, would appear in the terminal window (exact header text would depend on the type of node you are trying to reconfigure and date of build).
12. Type **1** to change the node ID value.
13. Follow the instructions to enter the node ID value.
14. Press the RESET switch on the node or type **0** to exit Configuration mode and run the application.

To confirm that the new node ID was saved properly, simply reset the board by pushing the RESET button on the board and make sure that the D1 and D2 LEDs are *not* lit. Also make sure that no menu is displayed on the RS-232 terminal. This confirms that your board is properly programmed and is ready for further configurations.

If this is a newly programmed demo RFD board, you must perform other configurations to observe the full demo functionality.

EXAMPLE 1: DEMO APPLICATION CONFIGURATION MENU

```
********************************************************************************
ZigBee Demo RFD Application v1.0 (Microchip Stack for ZigBee v1.0.0)
    Built on Nov 11 2004
********************************************************************************
    1. Set node ID...
    2. Join a network.
    3. Perform quick demo binding (Must perform #2 first)
    4. Leave a previously joined network (Must perform #2 first)
    5. Change to next channel.
    6. Transmit unmodulated signal.
    7. Transmit random modulated signal.
    0. Save changes and exit.

Enter a menu choice:
```

Programming Binding Configuration

At this point, you are now ready to perform the rest of the configurations. This requires that you have one demo coordinator application board and one or more demo RFD application boards. For simplicity, this procedure assumes that you have only one demo coordinator and one demo RFD board. However, you may easily extend this procedure to any number of demo RFD boards.

As part of this configuration, you will be associating the demo RFD application board (an end device) to the demo coordinator application board. You will also bind the S2 switch on one board to the D1 LED on another board. After successful completion of this configuration, you will be ready to experiment with the demo applications.

To simplify the binding configuration, the demo RFD application provides a quick demo binding menu option. The quick binding option is a single-step binding operation that demonstrates data flow between an end device and a coordinator. In addition to the quick binding menu option, you may also use on-board switches to create other advanced binding configurations.

PERFORMING QUICK DEMO BINDING

The quick demo binding option binds the S2 switch on the demo RFD board to the D1 LED on the demo coordinator and the S2 switch on the demo coordinator to the D1 LED on the demo RFD board. After completing the quick demo binding, you will be able to control the D1 LED on the demo coordinator by pressing the S2 switch on the demo RFD board and the D1 LED on the demo RFD board by pressing the S2 switch on the demo coordinator board.

The quick demo binding is primarily designed for a two-node (one coordinator and one RFD) network only. If you perform the quick demo binding on multiple demo RFD boards, the D1 LED on the demo coordinator will now be controlled by any of the demo RFD boards. However, the S2 switch on the demo coordinator will only control the D1 on the last demo RFD board that performed the quick demo binding.

The quick demo binding option requires the use of a PC with at least one standard serial port and terminal software. If you have access to two serial ports and two serial cables, you may view activity logs from both the demo coordinator and RFD boards simultaneously. If you only have one serial port, you would only connect the RFD board to PC.

Perform the following steps to do quick demo binding:

Note:	The following procedure assumes that the demo RFD board is connected to COM1 and the demo coordinator board is optionally connected to COM2.

1. Remove power from all nodes.
2. **Optional:** If you have two serial ports and two serial cables, launch your choice of a PC-based RS-232 terminal program and select the COM1 port with these settings: 19200 bps, 8-N-1, no flow control and echo typed characters.
3. Find the demo coordinator board and apply power to start its normal mode of execution. Make sure that D1 and D2 are flashed followed by a brief flash of D2. **Optional:** If connected to a serial port, note that the terminal displays the message "`New network successfully started`".
4. Launch your choice of a PC-based RS-232 terminal program and select the COM2 port with these settings: 19200 bps, 8-N-1, no flow control and echo typed characters.
5. Now, while keeping the demo coordinator board powered, apply power to the demo RFD board while holding the S3 switch, or press and hold both the RESET and S3 switches, then release the RESET switch. You should see a text menu in the output window of the terminal program.
6. Type **2** to start the "`Join a network`" command. Note that the terminal displays the message "`Successfully associated`". If you do not see this message, make sure that the demo coordinator node is powered and running in normal mode. **Optional:** On the second terminal connected to the demo coordinator node, note that the message "`A new node has just joined`" is displayed.
7. At this point, the demo RFD node has successfully joined the network established by the demo coordinator. You are now ready to perform the quick demo binding.
8. Type **3** to start the "`Perform quick demo binding`" command. Note that the terminal displays the message "`Demo binding complete`". If you do not see this message, make sure that demo coordinator node is powered and running in normal mode. **Optional**: On the second terminal connected to the demo coordinator node, note that the message "`Custom binding successful`" is displayed.
9. Enter **0** to "`Save current changes and exit configuration`". The terminal should now display "`Rejoin successful`".
10. You may now press S2 on the demo RFD node and observe that the D1 LED on the demo coordinator node toggles. Similarly, press S2 on the demo coordinator node and observe that the D1 LED on the demo RFD node toggles. You will observe that when you press S2 on the demo coordinator node, the D1 LED on the demo RFD node does not toggle immediately. This is due to the fact that the demo RFD node has to periodically poll the demo coordinator to obtain its LED status. The polling period depends on the watchdog prescaler value programmed into the demo RFD node. You should also note that D2 on both the demo RFD node and the demo coordinator are blinking periodically. This indicates that the demo RFD node is periodically polling the demo coordinator for its D2 status.
11. The association and quick demo binding configuration are stored permanently in the demo coordinator Flash memory.

PERFORMING ADVANCED BINDING

The advanced binding operation uses on-board switches to create a total of four combinations of binding configurations among multiple demo RFD boards. The advanced binding operation does not require terminal and serial cables. In order to eliminate duplicate information, the following procedure assumes that if you want to view activity logs on terminal window, you have already read the section **"Performing Quick Demo Binding"** and understand how to set up the terminal software.

Perform the following steps to do advanced binding:

1. Remove power from all nodes.
2. Find the demo coordinator board and apply power to start its normal mode of execution. Make sure that D1 and D2 are flashed followed by a brief flash of D2. **Optional:** If connected to a serial port, note that the terminal displays "`New network successfully started`".
3. While keeping the demo coordinator board powered, apply power to the demo RFD board while holding the S3 switch, or press and hold both RESET and S3 switches; then release the RESET switch. **Optional:** You should see a text menu in the output window of the terminal program.
4. Press S2 on the demo RFD node to begin the association sequence with the demo coordinator node. **Optional:** The terminal window should display "`Successfully associated`".
5. If you have more than one demo RFD node, press S2 on each demo RFD node to associate them to the demo coordinator node.
6. Since there are many different possible combinations of binding a configuration, the following table is used to describe the necessary sequence of steps for each combination
7. Press the RESET switch on each demo RFD node to begin normal execution. If connected to a terminal program, note that the message "`Rejoin successful`" is displayed.
8. Depending on how binding was performed, press S2 on one node to confirm that D1 on the same or other node toggles.

TABLE 5: BINDING OPERATION

To Bind Switch S2 On	To Bind LED D1 On	Result
RFD: Press and hold S3 first, then press S2 and release S2, followed by S3	**Coordinator**: Press and hold S3 first, then press S2 and release S3, followed by S2	S3 on RFD controls D1 on coordinator
Coordinator: Press and hold S3 first, then press S2 and release S2, followed by S3	**RFD**: Press and hold S3 first, then press S2 and release S3, followed by S2	S3 on coordinator controls D1 on RFD
RFD: Press and hold S3 first, then press S2 and release S2, followed by S3	**RFD**: Press and hold S3 first, then press S2 and release S3, followed by S2	S3 on RFD controls D1 on same RFD
RFD1: Press and hold S3 first, then press S2 and release S2, followed by S3	**RFD2**: Press and hold S3 first, then press S2 and release S3, followed by S2	S3 on RFD #1 controls D1 on RFD #2
Coordinator: N/A	**Coordinator**: N/A	Not allowed

Note 1: As each step is performed, LEDs D1 and D2 on the respective node will be toggled between ON and OFF, to OFF and ON. Also note that the terminal program connected to the RFD node displays the message "`Attempting to bind...`" and the terminal connected to the coordinator node displays the message "`Received valid...`".

2: To complete the binding process, you must perform both "To Bind Switch S2 On" and "To Bind LED D1 On" actions.

Executing Demo Applications

Before you can observe the demo application's functionality, you must have at least one demo coordinator and one demo RFD board. You must also have performed the configuration as described in **"Configuring Demo Applications"**.

The demo applications implement a simple remote control switch and LED functionality. With this functionality, you can press the switch on one board and control the LED on the same or another demo RFD board.

The demo applications are completely stand-alone and do not require an interface to a host computer. However, if you have access to a host computer, you may use it to observe the activity logs of the applications. An interface to a host computer is useful to understand and troubleshoot any setup issues you might have.

Do the following to execute demo applications:

1. Remove power from all boards if it was previously applied.
2. Locate the demo coordinator node.
3. **Optional:** Connect the demo coordinator node to a PC serial port and launch your favorite terminal program. Select the appropriate COM port with these settings: 19200 bps, 8-N-1, no flow control and echo typed characters.
4. Apply power to the demo coordinator node. Observe that both D1 and D2 flash simultaneously, followed by D2 flashing by itself. If connected to a PC, observe that the terminal program displays the message "`New network successfully started`".
5. Now locate the demo RFD node.
6. **Optional:** Connect the demo RFD node to a PC serial port and launch your favorite terminal program. Select the appropriate COM port with these settings: 19200 bps, 8-N-1, no flow control and echo typed characters.
7. While keeping the demo coordinator node still powered, apply power to the demo RFD node. Observe that both LEDs D1 and D2 flash simultaneously, followed by multiple flashes of D2. If connected to a PC, observe that in one to two seconds, the terminal program displays the message "`Rejoin successful`". If you do not see any message or see the message "`Rejoin failed`", make sure that you have the coordinator node powered and running properly; reset the demo RFD node and try again.
8. At this point, the RFD node has successfully associated with the demo coordinator node.
9. Depending on how the binding configuration was performed, press S2 on the demo RFD node and observe that D1 on the same or another node toggles.

Functional Description of Demo Applications

Both the demo coordinator and demo RFD applications demonstrate a simple ZigBee network. The demo coordinator and demo RFD applications form a non-slotted star network.

When a node is first programmed with any of the demo applications, on startup, a demo node will automatically enter into the Configuration mode. You must use a terminal interface to set a unique node ID. A demo RFD node needs additional setup, such as association and binding configuration, to make it fully functional.

FUNCTIONAL DESCRIPTION OF DEMO COORDINATOR

On power-up, the demo coordinator attempts to establish a new network by scanning for an empty channel. As part of its scanning procedure, the demo coordinator transmits the BEACON_REQ frame starting from the first channel in the current frequency band. If there is another coordinator in the same channel, it would respond to BEACON_REQ and the original coordinator would consider that channel as occupied. It would then switch to the next channel and repeat the procedure until it does not receive any response to its BEACON_REQ frame. Once a channel is found to be empty, it selects a random Personal Area Network (PAN) ID and starts listening on that channel. At this point a network is said to be established. From now on, if another coordinator were to broadcast a BEACON_REQ frame, our original coordinator would respond and declare its presence.

The demo coordinator is now ready to accept new end device nodes in its network. When a new end device wants to join a network, it first sends out a BEACON_REQ to detect the presence of a coordinator. Once the end device verifies the presence of a coordinator on a specific channel, it would begin association, or the orphan notification procedure, to join or rejoin the network.

A real world application may not always want to allow new associations at all times. For example, in a ZigBee protocol-based control network, you may not want any new sensor to join your control network. You may want to first enter into a special mode to control the new associations. The Microchip demo coordinator application does not place any restrictions on associations. Any device may associate or disassociate at any time. The coordinator does not have to be put in a special mode to perform these actions.

The demo coordinator also implements a simple switch and remote controlled LED interface. When a coordinator is first programmed, the switch and LED are not bound to any destination. Once the proper binding procedure is performed, you may press the S2 switch to control the D1 LED on the other end device. The demo coordinator implements a special sequence of switch presses to bind the on-board switch and LED to a remote device. You may initiate the binding sequence by pressing S2 and S3 simultaneously and follow the procedure outlined in the section **"Configuring Demo Applications"**.

FUNCTIONAL DESCRIPTION OF DEMO RFD

In addition to a unique node ID value, the demo RFD application requires that you associate it with a nearby demo coordinator. On power-up, the demo RFD node attempts to find a nearby demo coordinator. It will scan all available channels to find a demo coordinator. Once a demo coordinator is found, it attempts to rejoin its network. The rejoin attempt will succeed only if this node was previously joined to that demo coordinator. This is why you must first join this demo RFD node to a nearby node before executing the demo RFD node in normal mode.

The demo RFD node is put in Configuration mode to join to a new network. Once the node has joined to a network, you must also bind on-board switch S2 and D1 LED endpoint to some destination node. See **"Configuring Demo Applications"** for more information on how to perform these configuration actions.

After a demo RFD node is fully configured, on next normal execution start-up, it will automatically attempt to rejoin the nearby demo coordinator. Since it had already joined the demo coordinator in the past, the demo coordinator will allow the demo RFD node to rejoin its network.

Once rejoined to a network, the demo RFD node enables the Watchdog Timer, disables the RF transceiver and puts the controller to Sleep.

The demo RFD node uses PORTB interrupt-on-change functionality to wake itself up when any of the push buttons is pressed. If S2 was pressed, it sends out a special MSG data frame (see the ZigBee specification for more information) to the demo coordinator with the current switch status. Once the MSG data frame is acknowledged by the demo coordinator, the demo RFD node goes back to Sleep.

The demo RFD node may also be awakened by a Watchdog time-out. Upon exiting controller Sleep, the demo RFD node polls the demo coordinator for the new S2 status. If there is a new S2 status, the demo RFD node will update its D1 LED accordingly.

To further explain how a demo RFD node receives its S2 status, assume that the S2 switch on the demo RFD node is bound to the D1 LED on the same node. This binding configuration would allow us to control D1 by pressing S2 on the same board. When you press S2 on the demo RFD node, the demo RFD node would first send out a switch status update frame to the demo coordinator. At this point, the demo RFD node does not know who will receive the switch update frame. It simply directs the frame to the demo coordinator and goes back to Sleep. On the coordinator side, when it receives the switch status frame, it first looks up in its binding table to see if there is any known destination for this frame. Since the S2 switch was bound to the D1 LED on the same demo RFD node, the demo coordinator would find that there is an assigned destination for this frame and it would simply store the current switch status frame into its indirect transmit frame buffer. If there was no binding entry, the demo coordinator would have discarded the frame. The demo coordinator would hold the switch status frame in its indirect transmit frame buffer until either the intended recipient node retrieves it or a time-out occurs.

Assume that our original demo RFD node wakes up in time and sends a poll request to the demo coordinator. The demo coordinator would look in its indirect transmit buffer and find that there is a frame pending for this node. It will then transmit the switch status frame to the demo RFD node and wait for an Acknowledgement. Once the frame is Acknowledged, it will remove the switch status frame from its indirect transmit buffer.

The demo RFD node has now received a new switch status which it had sent out earlier. It would now decode the switch status and toggle the LED accordingly. Since there is a time difference between when the switch was first pressed and when it was polled, you will see a slight delay in the LED status change.

Since the coordinator stores and forwards a frame to an appropriate recipient, you may control an LED on a completely different node by simply changing the binding table in the coordinator memory.

USING THE MICROCHIP STACK

The files accompanying this application note contain the full source for the Microchip Stack ZigBee protocol (see **"Source Code"**). These source files also include two demo applications: one RFD demo application and one demo coordinator application.

All applications based on the Microchip Stack must be written in a cooperative multitasking manner. Cooperative multitasking architecture consists of a number of tasks executing in sequence. A cooperative task would quickly perform its required operation and return so that the next task would be able to execute. Because of this requirement, a task that needs to wait for some external input, or needs to perform a long operation, should be broken down into multiple subtasks using a state machine approach. Further discussion of cooperative multitasking and state machine programming is beyond the scope of this document. You should refer to software engineering literature for more detail.

The Microchip Stack is written to support both the MPLAB C18 and Hi-Tech PICC-18 C compilers without any changes. All source files automatically detect the current compiler in use and adjust its code accordingly. The Microchip Stack is written in standard ANSI C with C18 and PICC-18 specific extensions. If required, you may modify the source files to support your choice of compiler.

To simplify file management and application development, all source files are located in subdirectories under the `Source` directory. See **"Source File Organization"** for more information.

When you develop your application using the Microchip Stack, it is recommended that you use the demo application directory structure as a reference and create your own application-specific subdirectory.

Following are the typical steps you would use to develop an application based on the Microchip Stack. Note that these steps assume that you are using MPLAB IDE and are familiar with the MPLAB IDE interface.

1. Install the Microchip Stack source as previously described in the section **"Installing Source Files"**.
2. Create your application specific directory in the `MpZBee\Source` directory.
3. Depending on whether this is a coordinator application or RFD application, copy the `zigbee.def` file from either the `Source\DemoCoordApp` or `Source\DemoRFDApp` directory into your application-specific directory.
4. Modify `zigbee.def` as per your application requirements. See **"Stack Configuration"** for more information.
5. Use MPLAB IDE to create your application project and add the Stack source files as per your ZigBee node functionality. See **"Stack Source Files"** for information.
6. If you are using the MPLAB C18 compiler, add your device specific linker script file.
7. Use the MPLAB Build Option dialog box to set two additional include search paths: "`..\Stack`" and "`..\<YourAppDir>`", where `<YourAppDir>` is the name of the directory that contains your application specific `zigbee.def` file.
8. Add your application specific source files.
9. Now your application project is ready for build.

Stack Source Files

Depending on the type of your application, you would need to include a specific set of source files in your project. Table 6 lists all source files needed to build a typical ZigBee RFD application, and Table 7 lists all source files needed to build a typical ZigBee coordinator application.

TABLE 6: TYPICAL RFD APPLICATION FILES

Source Files	Purpose
Your App Files	Must include `main()` entry point
`zigbee.def`	Microchip Stack options specific to your application
`Console.c`	RS-232 terminal routines – needed if `ENABLE_DEBUG` is defined or your application uses console routines
`MSPI.c`	Master SPI™ interface routines to access RF transceiver
`Tick.c`	Tick manager, used to keep track of time-out and retry conditions
`zAPL.c`	ZigBee™ application layer
`zAPS.c`	ZigBee application support sublayer
`ZDO.c`	ZigBee device object – required if ZigBee remote management is needed
`zMAC.c`	IEEE 802.15.4 MAC layer
`zNVM.c`	Nonvolatile memory storage routines – may be replaced with your own nonvolatile storage specific file
`zNWK.c`	ZigBee network layer
`ZPHY???.c`	RF transceiver specific routines – `zPHYCC2420.c` for Chipcon CC2420 and `zPHYZMD44101.c` for ZMD 44101 transceiver
`zProfile.c`	ZigBee profile routines – required if standard profile support is needed (not fully implemented in version 1.00.00)
`18f????.lkr`	Linker script file specific your selection of device – required if using C18

TABLE 7: TYPICAL COORDINATOR APPLICATION FILES

Source Files	Purpose
Your App Files	Must include `main()` entry point
`zigbee.def`	Microchip Stack options specific to your application
`Console.c`	RS-232 terminal routines – needed if `ENABLE_DEBUG` is defined or your application uses console routines
`NeighborTable.c`	Implements neighbor and binding table
`SRAlloc.c`	Dynamic memory manager to implement indirect transmit buffer
`Tick.c`	Tick manager
`zAPL.c`	ZigBee™ application layer
`zAPS.c`	ZigBee application support sublayer
`ZDO.c`	ZigBee device object – required if ZigBee remote management is needed
`zMAC.c`	IEEE 802.15.4 MAC layer
`zNVM.c`	Nonvolatile memory storage routines – may be replaced with your own nonvolatile storage specific file
`zNWK.c`	ZigBee network layer
`ZPHY???.c`	RF transceiver specific routines – `zPHYCC2420.c` for Chipcon CC2420 and `zPHYZMD44101.c` for ZMD 44101 transceiver
`zProfile.c`	ZigBee profile routines – Required if standard profile support is needed (not fully implemented in version 1.00.00)
`18f????.lkr`	Linker script file specific your selection of device – required if using C18

Stack Configuration

The Microchip Stack uses many compile time options to enable/disable many of the core logic and RAM variables. Exact composition of core logic and RAM variables is dependent on the type of ZigBee application. To simplify this compile time configuration, all compile-time options are maintained in the `zigbee.def` file. As part of your application development, you must modify `zigbee.def`.

The following section defines all compile time options. You should review the `zigbee.def` file for the latest list of compile time options.

Option Name	`CLOCK_FREQ`
Purpose	Defines processor clock frequency. This value is used by `Tick.c` and `Debug.c` files to calculate TMR0 and SPBRG values, respectively. If required, you may also use this in your application.
Precondition	None
Valid Values	Must be within the PIC frequency specification.
Example	Following line defines `CLOCK_FREQ` as 4 MHz: `#define CLOCK_FREQ 4000000`

Option Name	`TICK_PRESCALE_VALUE`
Purpose	Timer0 prescale value. Used by `Tick.c` file to calculate TMR0 load value.
Precondition	None
Valid Values	Refer to the PIC device data sheet for possible TMR0 prescale value.
Example	Following line sets 2 as TMR0 prescale value: `#define TICK_PRESCALE_VALUE 2`
Note	None

Option Name	`TICKS_PER_SECOND`
Purpose	Number of ticks in one second. This is used by `Tick.c` file.
Precondition	None
Valid Values	1-255. This value must be adjusted depending on `TICK_PRESCALE_VALUE`.
Example	Following line sets 50 ticks in one second: `#define TICKS_PER_SECOND 50`
Note	None

Option Name	`BAUD_RATE`
Purpose	Defines USART baud rate value. This value is used by the `Console.c` file. You may change this value as per your application requirements.
Precondition	None
Valid Values	None
Example	Following line defines a 19200 bps baud rate: `#define BAUD_RATE (19200)`
Note	You must make sure that current selection of baud rate is possible with current selection of `CLOCK_FREQ`.

Option Name	ENABLE_DEBUG
Purpose	It enables Debug mode. If defined in zigbee.def file, Debug mode is enabled for all source files. Alternatively, you may selectively enable individual Debug mode by defining ENABLE_DEBUG in the beginning of a specific file.
Precondition	None
Valid Values	None
Example	Following line enables Debug mode: #define ENABLE_DEBUG
Note	When ENABLE_DEBUG is defined, your application code will be increased. ENABLE_DEBUG mode defines many ROM string messages.

Option Name	USE_CC24240
Purpose	Used to indicate that the Chipcon CC2420 transceiver is in use.
Precondition	USE_ZMD44101 must not be defined.
Valid Values	None
Example	Following line defines that CC2420 is in use: #define USE_CC2420
Note	You must only define USE_CC2420 or USE_ZMD44101. The Stack is designed to use only one type of RF transceiver at a time.

Option Name	USE_ZMD44101
Purpose	Used to indicate that ZMD 44101 transceiver is in use (not supported in current version).
Precondition	USE_CC2420 must not be defined.
Valid Values	None
Example	Following line defines that ZMD 44101 is in use: #define USE_ZMD44101
Note	You must only define USE_CC2420 or USE_ZMD44101. The Stack is designed to use only one type of RF transceiver at a time.

Option Name	I_AM_COORDINATOR
Purpose	Indicates that this node is a coordinator.
Precondition	I_AM_ROUTER and I_AM_END_DEVICE must not be defined.
Valid Values	None
Example	Following line sets current node as a coordinator: #define I_AM_COORDINATOR
Note	Once I_AM_COORDINATOR is defined, you must not define I_AM_ROUTER and I_AM_END_DEVICE.

Option Name	I_AM_ROUTER
Purpose	Indicates that this node is a router (not used in current version).
Precondition	I_AM_COORDINATOR and I_AM_END_DEVICE must not be defined.
Valid Values	None
Example	Following line sets current node as a router: #define I_AM_ROUTER
Note	Once I_AM_ROUTER is defined, you must not define I_AM_COORDINATOR and I_AM_END_DEVICE. Current version does not support Router functionality.

Option Name	I_AM_END_DEVICE
Purpose	Indicates that this is an end device – may be RFD or FFD (in current version, an end device must always be RFD).
Precondition	I_AM_COORDINATOR and I_AM_ROUTER must not be defined.
Valid Values	None
Example	Following line sets current node as an end device: #define I_AM_END_DEVICE
Note	Once I_AM_END_DEVICE is defined, you must not define I_AM_COORDINATOR and I_AM_ROUTER.

Option Name	MY_FREQ_BAND_IS_868_MHZ
Purpose	Defines 868 MHz as the frequency band of operation (not supported in current version).
Precondition	USE_ZMD44101 must be defined (in future versions, there may be more options).
Valid Values	None
Example	Following line sets 868 MHz frequency band: #define MY_FREQ_BAND_IS_868_MHZ
Note	Once MY_FREQ_BAND_IS_868_MHZ is defined, you must not define MY_FREQ_BAND_IS_900_MHZ and MY_FREQ_BAND_IS_2400_MHZ. Current version does not support 868/915 MHz operation.

Option Name	MY_FREQ_BAND_IS_900_MHZ
Purpose	Defines 915 MHz as the frequency band of operation (not supported in current version).
Precondition	USE_ZMD44101 must be defined (in future versions, there may be more options).
Valid Values	None
Example	Following line sets 915 MHz frequency band: #define MY_FREQ_BAND_IS_915_MHZ
Note	Once MY_FREQ_BAND_IS_915_MHZ is defined, you must not define MY_FREQ_BAND_IS_868_MHZ and MY_FREQ_BAND_IS_2400_MHZ. Current version does not support 868/915 MHz operation.

Option Name	MY_FREQ_BAND_IS_2400_MHZ
Purpose	Defines 2.4 GHz as the frequency band of operation.
Precondition	USE_CC2420 must be defined. (In future versions, there may be more options).
Valid Values	None
Example	Following line sets 2.4 GHz frequency band: #define MY_FREQ_BAND_IS_2400_MHZ
Note	Once MY_FREQ_BAND_IS_2400_MHZ is defined, you must not define MY_FREQ_BAND_IS_868_MHZ and MY_FREQ_BAND_IS_900_MHZ.

Option Name	I_AM_ALT_PAN_COORD
Purpose	Indicates that current FFD node can be an alternate PAN coordinator (not supported in current version).
Precondition	None
Valid Values	None
Example	Following line defines current node as an alternate PAN coordinator: #define I_AM_ALT_PAN_COORD
Note	Current version does not support FFD and alternate PAN coordinator functionality.

Option Name	I_AM_MAINS_POWERED
Purpose	Indicates that current node is AC powered. Normally, coordinator and router device will be AC powered (not used in current version).
Precondition	I_AM_RECHARGEABLE_BATTERY_POWERED and I_AM_DISPOSABLE_BATTERY_POWERED must not be defined.
Valid Values	None
Example	Following line indicates that this is a mains powered device: `#define I_AM_MAINS_POWERED`
Note	Once I_AM_MAINS_POWERED is defined, you must not define I_AM_RECHARGEABLE_BATTERY_POWERED and I_AM_DISPOSABLE_BATTERY_POWERED. Current version does not use this information. This information will be used to create standard node ZigBee profile.

Option Name	I_AM_RECHARGEABLE_BATTERY_POWERED
Purpose	Indicates that current node is battery-powered (not used in current version).
Precondition	I_AM_MAINS_POWERED and I_AM_DISPOSABLE_BATTERY_POWERED must not be defined.
Valid Values	None
Example	Following line indicates that this is a battery-powered device: `#define I_AM_RECGARGEABLE_BATTERY_OPERATED`
Note	Once I_AM_RECHARGEABLE_BATTERY_POWERED is defined, you must not define I_AM_MAINS_POWERED and I_AM_DISPOSABLE_BATTERY_POWERED. Current version does not use this information. This information will be used to create standard node ZigBee profile.

Option Name	I_AM_DISPOSABLE_BATTERY_POWERED
Purpose	Indicates that current node is disposable battery-powered (not used in current version).
Precondition	I_AM_MAINS_POWERED and I_AM_RECHARGEABLE_BATTERY_POWERED must not be defined.
Valid Values	None
Example	Following line indicates that this is a disposable battery-powered device: `#define I_AM_DISPOSABLE_BATTERY_POWERED`
Note	Once I_AM_DISPOSABLE_BATTERY_POWERED is defined, you must not define I_AM_RECHARGEABLE_BATTERY_POWERED and I_AM_DISPOSABLE_BATTERY_POWERED. Current version does not use this information. This information will be used to create standard node ZigBee profile.

Option Name	I_AM_SECURITY_CAPABLE
Purpose	Indicates that this node uses encryption/decryption to transmit and receive packets (not supported in current version).
Precondition	None
Valid Values	None
Example	Following line indicates that this node is security capable: `#define I_AM_SECURITY_CAPABLE`
Note	Current version does not support security.

Option Name	MY_RX_IS_ALWAYS_ON_OR_SYNCED_WITH_BEACON
Purpose	Indicates that this node keeps its receiver always ON or periodically listens for beacon (not supported in current version).
Precondition	MY_RX_IS_PERIODICALLY_ON and MY_RX_IS_ON_WHEN_STIMULATED must not be defined.
Valid Values	None
Example	#define MY_RX_IS_ALWAYS_ON_OR_SYNCED_WITH_BEACON
Note	Current version does not use or support this information.

Option Name	MY_RX_IS_PERIODICALLY_ON
Purpose	Indicates that this node periodically turns on its receiver (not used in current version).
Precondition	MY_RX_IS_ALWAYS_ON_OR_SYNCED_WITH_BEACON and MY_RX_IS_ON_WHEN_STIMULATED must not be defined.
Valid Values	None
Example	#define MY_RX_IS_PERIODICALLY_ON
Note	Current version does not use this information.

Option Name	MY_RX_IS_ON_WHEN_STIMULATED
Purpose	To indicate that this node turns on its receiver only when stimulated (not supported in current version).
Precondition	MY_RX_IS_ALWAYS_ON_OR_SYNCED_WITH_BEACON and MY_RX_IS_PERIODICALLY_ON must not be defined.
Valid Values	None
Example	#define MY_RX_IS_ON_WHEN_STIMULATED
Note	Current version does not use this information.

Option Name	MAC_LONG_ADDR_BYTEn
Purpose	To define default 64-bit MAC address for this node. There are a total of 8 defines, one for each byte. The main application may use this value to initialize MAC address or change it at run time as required.
Precondition	None
Valid Values	0-255
Example	Set default MAC address of 04:a3:00:00:00:00:01 #define MAC_LONG_ADDR_BYTE0 (0x01) #define MAC_LONG_ADDR_BYTE1 (0x00) #define MAC_LONG_ADDR_BYTE2 (0x00) #define MAC_LONG_ADDR_BYTE3 (0x00) #define MAC_LONG_ADDR_BYTE4 (0x00) #define MAC_LONG_ADDR_BYTE5 (0xa3) #define MAC_LONG_ADDR_BYTE6 (0x04) #define MAC_LONG_ADDR_BYTE7 (0x00)
Note	The Stack source does not automatically set this address. The main application must use this value to initialize mac address variable, macLongAddr, exposed by MAC layer.

Option Name	MAX_EP_COUNT
Purpose	Defines maximum number of endpoints supported by this device.
Precondition	None
Valid Values	1-255 (Must be at least 1. Maximum value depends on available RAM size; each EP takes 5 bytes of RAM.)
Example	`#define MAX_EP_COUNT (4)`
Note	There must be at least one EP to support standard ZDO endpoint. Each addition of EP count increases RAM usage by 5 bytes. There is a maximum limit of 255 endpoints in a given device; however, actual count will be limited by available RAM.

Option Name	MAC_USE_RF_TEST_CODE
Purpose	Enables transceiver specific test functions.
Precondition	None
Valid Values	None
Example	`#define MAC_USE_RF_TEST_CODE`
Note	In current version for Chipcon RF transceiver, there are two transceiver test functions – one to transmit a random modulated signal and another to transmit an unmodulated signal. These functions are useful to characterize RF circuit performance.

Option Name	MAC_USE_SHORT_ADDR
Purpose	Applies to end device only (i.e., I_AM_END_DEVICE is defined). This is used by an end device to request new short address when it associates with a network.
Precondition	None
Valid Values	None
Example	`#define MAC_USE_SHORT_ADDR`
Note	A ZigBee end device based on current version will always request short address from the coordinator.

Option Name	MAC_CHANNEL_ENERGY_THRESHOLD
Purpose	Defines the threshold over which a channel is said to be in use (not used in current version).
Precondition	None
Valid Values	0-255 (Exact unit depends on RF transceiver.)
Example	`#define MAC_CHANNEL_ENERGY_THRESHOLD (0x20)`
Note	None

Option Name	MAC_MAX_FRAME_RETRIES
Purpose	Sets the maximum frame retries count if no Acknowledge is received.
Precondition	None
Valid Values	1-5
Example	`#define MAC_MAX_FRAME_RETRIES (3)`
Note	None

Option Name	MAC_ACK_WAIT_DURATION
Purpose	Sets the maximum time this node should wait for Acknowledgement from another node.
Precondition	None
Valid Values	1-4,294,967,296 ticks
Example	Set ACK wait duration equal to half a second: `#define MAC_ACK_WAIT_DURATION (TICK_SECOND/2)`
Note	None

Option Name	MAC_RESPONSE_WAIT_TIME
Purpose	Sets the maximum time this node should wait for response from another node.
Precondition	None
Valid Values	1-255 ticks
Example	`#define MAC_RESPONSE_WAIT_TIME (TICK_SECOND)`
Note	In a star network, the end device would wait this long to receive response as a result of poll request.

Option Name	MAC_ED_SCAN_PERIOD
Purpose	Sets the energy detection period. During this period, the RF receiver is kept ON to measure RF energy.
Precondition	None
Valid Values	1-4,294,967,296 ticks
Example	Set 1/4 second as an ED scan period: `#define MAC_ED_SCAN_PERIOD (TICK_SECOND/4)`
Note	None

Option Name	MAC_ACTIVE_SCAN_PERIOD
Purpose	Sets the active scan period. During active scan, a node requests beacon from nearby coordinator(s) and expects coordinator(s) to respond within this period.
Precondition	None
Valid Values	1-4,294,967,296 ticks
Example	Set 1/2 second as an active scan period: `#define MAC_ACTIVE_SCAN_PERIOD (TICK_SECOND/2)`
Note	None

Option Name	MAC_MAX_DATA_REQ_PERIOD
Purpose	Used only when I_AM_COORDINATOR is defined. Sets the period during which end devices must request their data frame from this coordinator. You may also think of this as a period where each node in the network must poll the coordinator.
Precondition	I_AM_COORDINATOR must be defined.
Valid Values	1-4,294,967,296 ticks
Example	Set 10 seconds as max data request period: `#define MAC_MAX_DATA_REQ_PERIOD (TICK_SECOND*10)`
Note	None

Option Name	MAX_HEAP_SIZE
Purpose	Defines the maximum indirect transmit frame buffer size. The coordinator based on the Microchip Stack uses dynamic memory manager to allocate individual data frames within the indirect transmit frame buffer.
Precondition	None
Valid Values	128+. Maximum value depends on available RAM.
Example	Set 256 bytes big heap size: `#define MAX_HEAP_SIZE (256)`
Note	This is used by coordinator node only (i.e., I_AM_COORDINATOR is defined). The exact heap size depends on the MAX_DATA_REQ_PERIOD, the average size of frames and total number of nodes in a network. A network with longer MAX_DATA_REQ_PERIOD, more nodes and longer frames should have larger heap size. Typically, the heap size must be large enough to hold a typical number of frames until they are read by intended recipients. If C18 is in use, you must modify the linker script file and use the program to define a heap larger than 256 bytes.

Option Name	MAX_NEIGHBORS
Purpose	Defines the maximum number of nodes supported by this coordinator.
Precondition	I_AM_COORDINATOR must be defined.
Valid Values	2+. Maximum value depends on the available program memory. Each additional neighbor consumes 12 bytes of program memory.
Example	Set maximum number of nodes supported in this network: `#define MAX_NEIGHBORS (10)`
Note	This is used by the coordinator node only (i.e., I_AM_COORDINATOR is defined). Depending on the total number of nodes and how often they poll the coordinator, you may need to increase clock frequency of the coordinator to keep up with increased processing requirements.

Option Name	MAX_BINDINGS
Purpose	Defines the maximum number of binding requests (binding table size) supported by this coordinator.
Precondition	I_AM_COORDINATOR must be defined.
Valid Values	2-255. Maximum value depends on available program memory. Each additional binding entry consumes 12 bytes of program memory.
Example	Set maximum binding table size of 10: `#define MAX_BINDINGS (10)`
Note	This is used by coordinator node only (i.e., I_AM_COORDINATOR is defined). There may be multiple binding entry per node. Exact binding table size would depend on the number of nodes in the network and number of binding requests per node.

Integrating Your Application

After modifying the compile time configurations as required by your application, the next step would be to modify your main application to initialize and run a Stack state machine. If you are developing a coordinator node, you should use the DemoZCoordApp.c file as a reference. For an RFD node, use the DemoZRFDApp.c file.

CALLBACK FUNCTIONS

In addition to standard API calls, your application must also implement a number of callback functions. Callback functions reside in the main application source files. The Stack calls these callback functions to notify or confer with the application before making any application-specific decision. All callback function names are prefixed with "App" to signify callback functions. The following section discusses each callback function in more detail.

AppOkayToUseChannel

This callback function asks main application if it should use given channel.

Syntax

```
BOOL AppOkayToUseChannel(BYTE channel)
```

Parameters

channel [in]

Channel number that needs to be selected. This value depends on the frequency of bands:

For 2.4 GHz, 11-26

For 915 MHz, 1-10

For 868 MHz, 0

Return Values

TRUE if application wants to use given channel

FALSE, if otherwise

Precondition

None

Side Effects

None

Note

Application must implement this callback function even if it uses all channels in its frequency band. When an application returns FALSE, the Stack will automatically call this function with next channel until the application returns TRUE.

Example

```
BOOL AppOkayToUseChannel(BYTE channel)
{
        // We are operating in 2.4 GHz band and we only want to use channel 11-15
        return ( channel <= 15 );
}
```

AppMACFrameReceived

This callback function notifies the application that a new valid data frame is received. This is just a notification – the actual frame may or may not be processed. Application may use this notice to blink an LED or other visual indicator.

Syntax

```
void AppMACFrameReceived(void)
```

Parameters

None

Return Values

None

Precondition

None

Side Effects

None

Note

None

Example

```
void AppMACFrameReceived(void)
{
        // RD1 LED is used to indicate receive activities
        RD1 = 1;
        // Assume that LED will be turned off by the timer interrupt.
}
```

AppMACFrameTransmitted

This callback function notifies the application that a data frame has just been transmitted. Application may use this notice to blink an LED or other visual indicator.

Syntax

```
void AppMACFrameTransmitted(void)
```

Parameters

None

Return Values

None

Precondition

None

Side Effects

None

Note

None

Example

```
void AppMACFrameTransmitted(void)
{
        // RD1 LED is used to indicate transmit activities
        RD1 = 1;
        // Assume that LED will be turned off by the timer interrupt.
}
```

AppMACFrameTimeOutOccurred

This callback function notifies the application that a remote node did not send Acknowledgement within MAC_ACK_WAIT_DURATION. Application may use this notice to blink an LED or other visual indicator.

Syntax

```
void AppMACFrameTimeOutOccurred(void)
```

Parameters

None

Return Values

None

Precondition

None

Side Effects

None

Note

None

Example

```
void AppMACFrameTimeOutOccurred(void)
{
        // RD2 LED is used to indicate timeout conditions
        RD2 = 1;
        // Assume the LED will be turned off by the timer interrupt.
}
```

AppOkayToAssociate

This is a callback function to the main application. When an end device is attempting to join an available network, the Stack would call this function when it finds a coordinator in its radio sphere. In one radio sphere, there could be more than one coordinator on different channels; in which case, the Stack would repeatedly find those coordinators and call this function for application's approval before requesting to join that specific coordinator. The application may decide to accumulate all nearby coordinators before selecting a specific one to associate.

This callback is available only when `I_AM_END_DEVICE` is defined.

Syntax

`BOOL AppOkayToAssociate(void)`

Parameters

None

Return Values

`TRUE`, the application wants to associate with the current coordinator. Application may check information of current coordinator by accessing `PANDesc` variable structure defined in `MAC.h` file.

`FALSE`, if otherwise. In this case, the Stack would continue to look for a new coordinator by automatically switching to next available channel.

Precondition

None

Side Effects

None

Note

None

Example

```
// Callback resides in main application source file.
BOOL AppOkayToAssociate(void)
{
        // Let's say that we will associate with a coordinator whose first three bytes
        // of its MAC is same as mine (i.e. it belongs to my devices)
        if ( PANDesc.CoordAddress.longAddr.v[0] == macInfo.longAddr.v[0] &&
             PANDesc.CoordAddress.longAddr.v[1] == macInfo.longAddr.v[1] &&
             PANDesc.CoordAddress.longAddr.v[2] == macInfo.longAddr.v[2] )
                return TRUE;
        else
                return FALSE;
}
```

AppOkayToAcceptThisNode

This callback function asks the main application if it wants to accept given node into its network. The main application may implement its private selection criteria to allow new nodes to join its network.

This callback is available only when I_AM_COORDINATOR is defined.

Syntax

```
BOOL AppOkayToAcceptThisNode(LONG_ADDR *longAddr)
```

Parameters

longAddr [in]

Pointer to a 64-bit MAC address of node who wants to join this network.

Return Values

TRUE, if application wants to allow the given node to join its network

FALSE, if otherwise

Precondition

None

Side Effects

None

Note

None

Example

```
// Callback resides in main application source file.
BOOL AppOkayToAcceptThisNode(LONG_ADDR *longAddr)
{
        // Let's say that we will only allow nodes, whose first three bytes of its MAC
        // is same as ours (i.e. it belongs to my devices)
        if (longAddr->v[0] == macInfo.longAddr.v[0] &&
            longAddr->v[1] == macInfo.longAddr.v[1] &&
            longAddr->v[2] == macInfo.longAddr.v[2] )
                return TRUE;
        else
                return FALSE;
}
```

AppNewNodeJoined

This callback function notifies the main application that a new node has just joined its network.

This callback is available only when `I_AM_COORDINATOR` is defined.

Syntax

```
void AppNewNodeJoined(LONG_ADDR *nodeAddr, BOOL bIsRejoined)
```

Parameters

`nodeAddr [in]`

Pointer to a 64-bit MAC address of node who has just joined the network.

`bIsRejoined [in]`

Indicates if this is a familiar node or new node.

Return Values

None

Precondition

None

Side Effects

None

Note

When a new node joins the network, a new entry is created into neighbor table. However, the entry is not fully saved until the `APLCommitTableChanges` function is called. An application may not always want to allow a new node to join its network. The coordinator may be put in a special mode to allow a new node to join its network. To provide this flexibility, the Microchip Stack requires that you call `APLCommitTableChanges()` to commit the actual association request. Exact decision as to when to call this function depends on your application logic.

Example

```
// Callback resides in main application source file.
void NewNodeJoined(LONG_ADDR *nodeAddr, BOOL bIsRejoined)
{
        // If this node is new, save its association information to NVM.
        if ( bIsRejoined == FALSE )
        {
                APLCommitTableChanges();
        }
        // Else don't do anything.
}
```

AppNodeLeft

This callback function notifies the main application that a familiar node has just left the network.

This callback is available only when `I_AM_COORDINATOR` is defined.

Syntax

```
void AppNodeLeft(LONG_ADDR *nodeAddr)
```

Parameters

`nodeAddr [in]`

Pointer to a 64-bit MAC address of node who has just left the network.

Return Values

None

Precondition

None

Side Effects

None

Note

When a new node leaves the network, the corresponding association and binding table entries must be removed. Once associated entries are removed, the changes must be committed to permanently save the changes.

Example

```
// Callback resides in main application source file.
void AppNodeLeft(LONG_ADDR *nodeAddr)
{
        // Before this function was called, the stack has already deleted table entries
        // for this node. We just need to commit the changes.
        APLCommitTableChanges();
}
```

ZigBee PROTOCOL OVERVIEW

ZigBee is a standard wireless network protocol designed for low rate control networks. Some of the applications for the ZigBee protocol include building automation networks, building security systems, industrial control networks, remote meter reading and PC peripherals. The following sections provide a brief overview of the ZigBee protocol that is relevant in understanding Microchip Stack functionality. Interested readers should refer to the ZigBee web site (www.zigbee.org) for more information.

IEEE 802.15.4

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

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

The maximum length of an IEEE 802.15.4 MAC packet is 127 bytes. Each packet consists of header bytes and a 16-bit CRC value.

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

Network Configurations

A ZigBee wireless network may assume many types of configurations. A star network configuration consists of one coordinator node (a “master”) and one or more end devices (“slaves”). The coordinator is a special variant of a Full Function Device (FFD) that implements a larger set of ZigBee services. The end devices may be FFD or a Reduced Function Device (RFD). An RFD is the smallest and simplest ZigBee node. It implements only a minimal set of ZigBee services. In a star network, all end devices communicate to the coordinator only. If an end device needs to transfer data to another end device, it sends its data to the coordinator and the coordinator, in turn, forwards the data to the intended receiver end device.

In addition to the star network, a ZigBee network may also assume peer-to-peer, cluster, or mesh network configurations. The cluster and mesh network are also known as a multi-hop network, due to their abilities to route packets between multiple networks, while the star network is called a single-hop network.

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

The current version of the Microchip Stack supports non-beacon star network configuration only.

Network Association

ZigBee networks can be ad-hoc, meaning that a new network is formed and unformed as needed. In a star network configuration, end devices would always search for a network before they can perform any data transfer. A new network is first established by a coordinator. On start-up, a coordinator searches for other coordinators nearby and if none is found, it establishes its own network and selects a unique 16-bit PAN ID. Once a new network is established, one or more end devices are allowed to associate with the network. The exact decision to allow or disallow new associations depends on the coordinator.

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

Depending on system requirements, a coordinator may store all of the network associations in nonvolatile memory, called a neighbor table. In order to connect to a network, an end device may either execute the orphan notification procedure to locate its previously associated network or execute the association procedure to join a new network. In the case of the orphan notification procedure, the coordinator will recognize a previously associated end device by looking up its neighbor table.

Once associated to a network, an end device may choose to disassociate from the network by performing the disassociate procedure. If required, a coordinator itself may also initiate a disassociate procedure to force a node to leave the network.

The current version of the Microchip Stack supports new association and orphan notification procedures. It only supports a network leave procedure initiated by an end device.

Endpoints, Interfaces, Clusters, Attributes and Profiles

A typical ZigBee node may support multiple features and functionality. For example, an I/O node may have multiple digital and analog inputs/outputs. Some of the digital inputs may be used by a remote controller node and others may be used by another remote controller node. This arrangement creates a truly distributed control network. To facilitate data transfer between the I/O node and two controller nodes, the applications in all nodes must maintain multiple data links. In order to reduce cost, a ZigBee node uses only one radio channel and multiple endpoint/interfaces to create multiple virtual links or channels.

One ZigBee node supports 31 endpoints (numbered 0-31) and 8 interfaces (numbered 0-7). The endpoint 0 is reserved for device configuration and endpoint 31 is reserved for broadcasts only. This leaves a total of 30 endpoints for application use. For each endpoint, there can be a total of 8 interfaces. Thus, in reality, an application may have up to 240 virtual channels in one physical channel.

A typical ZigBee node would also have many attributes. For example, our I/O node contains attributes called digital input #1, digital input #2, analog input #1, etc. Each attribute would have its own value. For example, the digital input #1 attribute may have a value of '1' or '0'. A collection of attributes is called a cluster. Each cluster is assigned a unique cluster ID in the entire network. Each cluster may have up to 65,536 attributes.

The ZigBee protocol also defines a term called *profile*. A profile is synonymous to the description of a distributed application. It describes a distributed application in terms of the packets it must handle and actions it must perform. A profile is described using a descriptor, which is nothing but a complex structure of various values. It is the profile that makes ZigBee devices interoperable. The ZigBee Alliance has defined many standard profiles, such as remote control switch profile, light sensor profile, etc. Any node that conforms to one of these standard profiles will be interoperable with other nodes implementing the same profile. The current version of the Microchip Stack does not provide any standard profile functionality. If required, you may easily write a cooperative task function that implements the desired profile. You may also create your own custom profile (or distributed application) that works with your proprietary nodes only. The demo application provided with this application note implements a custom distributed application of a remote controlled LED, where an LED of one node is controlled by a switch on another node. Each profile can define up to 256 clusters and as we saw earlier, each cluster can have up to 65,536 attributes. This flexibility allows a node to have a very large number of attributes (or I/O points).

Endpoint Binding

As mentioned earlier, end devices in a star network always communicate to the coordinator only. The coordinator is responsible for forwarding the data packet sent from an endpoint from one node to the appropriate endpoint(s) in the receiving end device. As you might have guessed, when a new network is established, the coordinator must be told how to create source and destination endpoint links. The ZigBee protocol defines a special procedure called endpoint binding. As a part of the binding process, a remote network/device manager-like node may ask the coordinator to modify its binding table. The coordinator node maintains a binding table that essentially contains a logical link between two or more endpoints. Each link is uniquely defined by its source endpoint and cluster ID.

For example, if the data from digital input #1 of our I/O node needs to be sent to control channel #1 of the controller node, we must ask the coordinator to create a binding table entry that consists of digital input #1 endpoint of the I/O node as a source and control channel #1 of the controller node as a destination. Once a binding table entry is created, any time the I/O node sends data from its digital input #1 endpoint, the coordinator node will look up its binding table and forward the packet to the control channel #1 endpoint of the controller node. Both digital input #1 and control channel #1 will share a common cluster ID. Depending on how the binding table is created, it is possible to multicast data from one endpoint to multiple endpoints on multiple nodes. The current version of the Microchip Stack does not support such multicast binding table entries.

The ZigBee protocol defines a special software object, called the ZigBee Device Object (ZDO), that provides binding services among other services. Only the ZDO running on the coordinator will provide the binding services. A remote network/device manager would issue a special binding request directed to the ZDO (endpoint 0) to create or modify a binding table entry. As per the ZigBee specifications, a PC or other high-end controller running special ZigBee node software may act as a network manager.

If you do not want to create or use a special network manager node, you may write your own custom binding services that simplify the binding procedure. The demo applications included with this application note implement a simple custom binding method. It enables each node to send its own binding request to the coordinator node to the endpoint 0 and defines its own custom cluster ID. For more information, please see the `CUSTOM_DEMO_BIND` cluster ID in the `ZDO.c` file. According to this custom binding procedure, the end device must be in Configuration mode to send binding requests. The coordinator can receive and originate its own binding requests in normal mode of execution.

When a certain sequence of switch presses is detected, the end device sends out a special binary data structure using the `CUSTOM_DEMO_BIND` as a cluster ID (refer to **"Configuring Demo Applications"** to learn about the Configuration mode and binding sequence). It directly sends the binding request packet to the endpoint 0 of the coordinator. The ZDO in the demo coordinator receives the packet identified as the `CUSTOM_DEMO_BIND` cluster ID and delegates the processing to the `ProcessCustomBind` function. This function is actually implemented in the `DemoCoordApp.c` file. You may easily follow the execution logic of the `ProcessCustomBind` function to fully understand the custom binding concept. You may use this custom binding logic as it is or write your own using it as a reference.

Data Transfer Mechanism

Depending on the type of network, exact mechanisms to transfer data to and from the end device differ. In a non-beacon star network, when an end device wants to send a data frame, it simply waits for the channel to become idle. Upon detecting an Idle channel condition, it transmits its frame to the coordinator. If a coordinator wants to send data to an end device, it holds the data frame in its transmit buffer until the intended end device explicitly polls for the data. This method ensures that the receiver of the end device is turned ON and it is capable of receiving data from the coordinator.

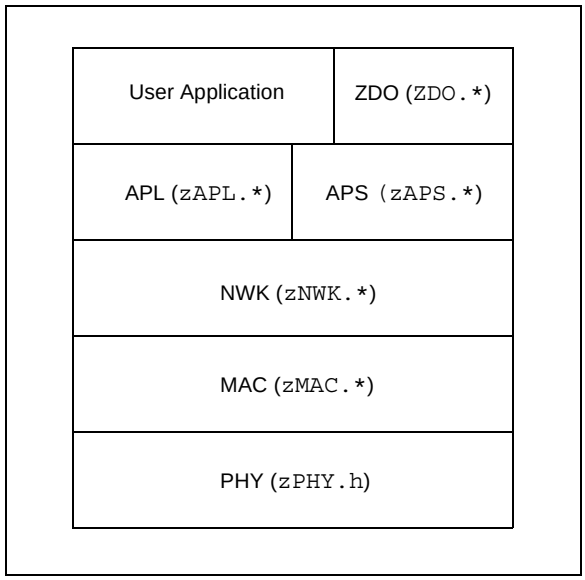
In a peer-to-peer network, each node must either keep their receiver ON all the time or agree upon an interval period during which they will switch ON their receivers. This will allow a node to transmit a data frame and ensure that the frame will be received by the other node.

The fact that the end device must poll the coordinator for its data, rather than keep its receiver ON, allows the end devices to lower their power requirement. Depending on the application requirements, an end device may spend most of its time sleeping and only periodically wake-up to transmit or receive data. The one disadvantage to this approach is that the coordinator must hold all data frames in its internal buffer until the intended end device wakes up and polls for the data. If a network contains many end devices that sleep for long time, the coordinator must keep the data frame for that long period. Depending on the number of nodes and rate of data frame exchanges, this would drastically increase the coordinator RAM requirements. A coordinator may selectively decide to hold a specific frame for a long time, or a short time, based on the device descriptor for the end device. The ZigBee protocol requires that all end devices will maintain various descriptors that describe various aspects of their features and capabilities. The current version of the Microchip Stack does not support descriptors.

STACK ARCHITECTURE

The Microchip Stack is written in the C programming language, intended for both MPLAB C18 and Hi-Tech PICC-18 compilers. Depending on which is used, the source files automatically make the required changes. The Microchip Stack is designed to run on Microchip's PIC18F family of microcontrollers only. The Microchip Stack uses internal Flash program memory to store its configurable MAC address, network table and binding table. Consequently, you must use a self-programmable Flash memory microcontroller. If required, you may modify the Nonvolatile Memory (NVM) routines to support any other type of NVM and not use a self-programmable microcontroller. In addition, the Stack is targeted to run on the PICDEM Z demonstration board. However, it can be easily ported to any hardware equipped with a compatible microcontroller.

FIGURE 2: MICROCHIP STACK ARCHITECTURE



Stack Layers

The Microchip Stack divides its logic into multiple layers as defined by the ZigBee specification. The code implementing each layer resides in a separate source file, while the services and Application Programming Interfaces (APIs) are defined in the include file.

The current version of the Stack does not implement a security layer. Each layer defines a set of easy to understand functions to its immediate upper layer. To create an abstraction and modularity, a top-level layer always interacts with a layer immediately below it through well-defined APIs. A C header file for a specific layer (`zAPS.h`, for example) defines all APIs supported by that specific layer. With that in mind, a user application would always interact with the Application Programming Support (APS) layer and Application Layer (APL). Many of the APIs provided by each layer are simply C macros that call functions one layer down. This method avoids the typical overhead associated with modularization.

Stack APIs

The Microchip Stack consists of many modules. A typical application would always interface to the Application Layer (APL) and the Application Support Sublayer (APS). However, if required, you may easily interface to other modules and/or customize them as needed. The following sections provide detailed API descriptions of the APL and APS modules only. If required, you may learn the details of APIs for other modules in their respective header files. For up-to-date information, you should refer to the actual source files.

APPLICATION LAYER (APL)

The APL module provides the high-level Stack management functions. A user application would use this module to manage the Stack functionality. The `zAPL.c` file implements the APL logic and the `zAPL.h` file defines the APIs supported by the APL module. A user application would include the `zAPL.h` header file to access its APIs.

APLInit

This function initializes all Stack modules. It also initializes APL state machine.

Syntax

```
void APLInit(void)
```

Parameters

None

Return Values

None

Precondition

None

Side Effects

None

Note

On POR, RF transceiver is powered down. You must call `APLEnable()` to enable RF transceiver.

Example

```
// Initialize stack
APLInit();
```

APLIsIdle

This macro is used to detect if it is okay to disable APL and other modules.

Syntax

`BOOL APLIsIdle(void)`

Parameters

None

Return Values

`TRUE`, if APL is Idle and can be disabled

`FALSE`, if otherwise

Precondition

`APLInit()` is called.

Side Effects

None

Example

```
// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// Enter into main application loop
while(1)
{
    // Let stack execute
    ZAPLTask();

    // If stack is idle, disable it and put micro to sleep
    if ( APLIsIdle() )
    {
        APLDisable();

        // May be now the micro should go to sleep...
        SLEEP();

        ...
}
```

APLEnable

This macro is used to enable Stack modules and RF transceiver.

Syntax

`void APLEnable(void)`

Parameters

None

Return Values

None

Precondition

`APLInit()` is called.

Side Effects

None

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();
```

APLDisable

This macro is used to disable RF transceiver and other Stack modules.

Syntax

`void APLDisable(void)`

Parameters

None

Return Values

None

Precondition

`I_AM_END_DEVICE` is defined and `APLInit()` is called.

Side Effects

All pending receptions are lost.

Note

Only end devices should disable RF transceiver to conserve power. Coordinator and router devices should always keep their RF transceiver ON.

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// Enter into main application loop
while(1)
{
    // Let stack execute
    ZAPLTask();

    // If stack is idle, disable it and put micro to sleep
    if ( APLIsIdle() )
    {
        APLDisable();

        // May be now the micro should go to sleep...
        SLEEP();

        ...
}
```

APLTask

This is a cooperative task function that calls each Stack module task function sequentially. Calls to this function allow the Stack to fetch and process incoming data packets.

Syntax

`BOOL APLTask(void)`

Parameters

None

Return Values

`TRUE`, if the function has completed its task and is ready in Idle state

`FALSE`, if otherwise

Precondition

`APLInit()` is called.

Side Effects

None

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        // Now perform your app task(s)
        ProcessIO();

        // If stack is idle, disable it and put micro to sleep
        if ( APSIsIdle() )
        {
                APSDisable();

                // May be now the micro should go to sleep...
                SLEEP();

                ...
}
```

APLNetworkInit

This macro starts the new network initialization. You must repeatedly call `APLIsNetworkInitComplete` to let the state machine run and determine if network initialization is complete.

Syntax

`void APLNetworkInit(void)`

Parameters

None

Return Values

None

Precondition

`I_AM_COORDINATOR` is defined and `APLInit()` is called.

Side Effects

None

Note

Must call `APLIsNetworkInitComplete()` to determine if network initialization is complete.

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// A coordinator will always try to set its own network.
APLNetworkInit();

// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        if ( APLIsNetworkInitComplete() )
                ...
}
```

APLIsNetworkInitComplete

This macro executes the network initialization state machine and indicates if network initialization is complete. This is a cooperative task – you must call it repeatedly until it returns TRUE.

Syntax

BOOL APLIsNetworkInit(void)

Parameters

None

Return Values

TRUE, if network initialization is complete. TRUE does not indicate success, it simply means that network initialization is complete. You must call GetLastZError() to determine if it was successful or not.

FALSE, if otherwise

Precondition

I_AM_COORDINATOR is defined and APLNetworkInit() is called.

Side Effects

None

Note

A return value of TRUE simply indicates that a network initialization process is complete. An initialization process may be complete because either it has succeeded or failed. Call GetLastZError() to determine if process was successful.

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// A coordinator will always try to set its own network.
APLNetworkInit();

// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        if ( APLIsNetworkInitComplete() )
        {
                // Check to see if it was successful
                if ( GetLastZError() == ZCODE_NO_ERROR )
                {
                        // A network is established
                        ...
                }
                else
                {
                        // New network could not be established
                        // Do application specific recovery
                ...
}
```

APLNetworkForm

This macro instructs network layer to form a new network on the current channel.

Syntax

`void APLNetworkForm(void)`

Parameters

None

Return Values

None

Precondition

`I_AM_COORDINATOR` is defined and `APLIsNetworkInitCompleted() = TRUE` and `GetLastZError() = ZCODE_NO_ERROR`.

Side Effects

None

Note

None

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// A coordinator will always try to set its own network.
APLNetworkInit();

// Enter into main application loop
while(1)
{
    // Let stack execute
    APLTask();

    if ( APLIsNetworkInitComplete() )
    {
        // Check to see if it was successful
        if ( GetLastZError() == ZCODE_NO_ERROR )
        {
            // Network init is successful – form it
            APLNetworkForm();
            ...
        }
        else
        {
            // New network could not be established
            // Do application specific recovery
        ...
}
```

APLPermitAssociation

This macro allows end devices to associate to network.

Syntax

`void APLPermitAssociation(void)`

Parameters

None

Return Values

None

Precondition

`I_AM_COORDINATOR` is defined and `APLNetworkForm()` is called.

Side Effects

None

Note

Depending on the application requirements, you may or may not want to allow a new device to join your network.

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// A coordinator will always try to set its own network.
APLNetworkInit();

// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        ...

        // At this point, a new network is formed.

        // If in config mode, allow new associations
        if ( bInConfigMode )
        {
                APLPermitAssociation();
        }

        ...
}
```

APLDisableAssociation

This macro disallows new devices from joining the network. This is complementary of `APLPermitAssociation`.

Syntax

`void APLDisableAssociation(void)`

Parameters

None

Return Values

None

Precondition

`I_AM_COORDINATOR` is defined and `APLNetworkForm()` is called.

Side Effects

None

Note

When running in normal mode, a coordinator may not want to allow any new devices to join the network. In this case, you would call this function to stop any device from joining the network.

When a network is first formed, new associations are disabled by default.

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// A coordinator will always try to set its own network.
APLNetworkInit();

// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        ...

        // At this point, a new network is formed.

        // If in config mode, allow new associations
        if ( bInConfigMode )
                APLPermitAssociation();
        else
                APLDisableAssocation();

        ...
```

APLCommitTableChanges

This macro saves all association and binding requests received so far and writes them in Flash program memory. Once associations are committed, the coordinator would remember its nodes and allow them to rejoin the network in the future.

Syntax

`void APLCommitAssociation(void)`

Parameters

None

Return Values

None

Precondition

`I_AM_COORDINATOR` is defined.

Side Effects

Modifies the network and binding table information header.

Note

Once `APLPermitAssociation` is called, all new association requests are automatically written into Flash memory. However, they are marked valid only after they are committed using this macro. Once associations are committed, they are marked valid and from then on, corresponding end devices will be allowed to rejoin the network in the future.

Example

```
...

// Initialize stack
APLInit();
// Enable RF transceiver
APLEnable();
// A coordinator will always try to set its own network.
APLNetworkInit();
// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        ...

        // At this point, a new network is formed.

        // If in config mode, allow new associations
        if ( bInDebugMode )
                APLPermitAssociation();
        else
                APLDisableAssocation();

        // After some predetermined time, stop accepting
        // new association requests
        if ( bAssocTimeOut )
        {
                APLCommitAssociations();
        }
        ...
```

APLJoin

This macro is used by an end device to join to one of the many potentially available networks. This macro simply starts the “Join” state machine. You must repeatedly call `APLIsJoinComplete` to allow the state machine to execute and determine if the join is complete.

This macro is available only when `I_AM_END_DEVICE` is defined.

Syntax

```
void APLJoin(void)
```

Parameters

None

Return Values

None

Precondition

`I_AM_END_DEVICE` is defined.

Side Effects

None

Note

An end device must always join a network before it will be allowed to participate in network communication. In a typical system, you may not want your end device to join any available network but a familiar one. In that case, the end device will attempt to join a new network in special “Configuration” mode only. Once an end device becomes a member of a network, it would simply “rejoin” that network in the future.

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// If in special mode, start the join procedure
if ( bInConfigMode )
        NWKJoin();

// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        // Check to see if join is complete
        if ( bInConfigMode )
        {
                if ( APLIsJoinComplete() )
                {
        ...
}
```

APLIsJoinComplete

This macro is used by the end device to determine if a previously started join process is complete.

Syntax

`BOOL APLIsJoinComplete(void)`

Parameters

None

Return Values

`TRUE`, if join process is complete – must call `GetLastZError()` to determine if it was successful

`FALSE`, if otherwise

Precondition

`I_AM_END_DEVICE` is defined and `APLJoin()` is called.

Side Effects

None

Note

A return value of `TRUE` simply indicates that a join process is complete. A join process may be complete because either it has succeeded or failed. Call `GetLastZError()` to determine if process was successful.

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// If in special mode, start the join procedure
if ( bInDebugMode )
        NWKJoin();

// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        // Check to see if join is complete
        if ( bInDebugMode )
        {
                if ( APLIsJoinComplete() )
                {
                        // Check to see if it is successful
                        if ( GetLastZError() == ZCODE_NO_ERROR )
                        {
        ...
}
```

APLRejoin

This macro is used by the end device to start the rejoin process. When running in normal mode, an end device should rejoin to a previously joined network unless your application requires it to find and join a new network on every start-up.

Syntax

```
void APLRejoin(void)
```

Parameters

None

Return Values

None

Precondition

`I_AM_END_DEVICE` is defined and `APLInit()` and `APLEnable()` are called.

Side Effects

Marks that the node is not currently associated.

Note

In normal mode, an end device would simply rejoin the already joined network. A device must have joined at least one network before it can successfully rejoin a network. In addition, the previously joined network must be in its radio sphere to be able to successfully rejoin it.

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// If in special mode, start the join procedure
if ( bInConfigMode )
        APLJoin();

else
        // Else initiate rejoin process.
        APLRejoin();

// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        ...
}
```

APLIsRejoinComplete

This macro is used by the end device to determine if a previously started rejoin process is complete. This is a cooperative task; the end device must call this macro repeatedly.

Syntax

`BOOL APLIsRejoinComplete(void)`

Parameters

None

Return Values

`TRUE`, if rejoin process is complete. Call `GetLastZError` to determine the success/fail status.

`FALSE`, if otherwise

Precondition

`I_AM_END_DEVICE` is defined and `APLRejoin()` is called.

Side Effects

None

Note

To determine if this node successfully rejoined previously known network or not, call `GetLastZError()`.

Example

```
...

// Initialize stack
APLInit();

// Enable RF transceiver
APLEnable();

// If in special mode, start the join procedure
if ( bInConfigMode )
        NWKJoin();

else
        // Else initiate rejoin process.
        NWKRejoin();

// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        if ( APLIsRejoinComplete() )
        {
                if ( GetLastZError() == ZCODE_NO_ERROR )
                {
                        // Successfully rejoined the network
   ...
```

APLLeave

This macro is used by the end device to initiate the leave sequence from an existing network.

Syntax

`void APLLeave(void)`

Parameters

None

Return Values

None

Precondition

`I_AM_END_DEVICE` is defined.

Side Effects

None

Note

This macro simply sets up the state machine for leave procedure. You must repeatedly call `APLIsLeaveComplete()` to complete the leave procedure.

Example

```
...

// Start the leave process.
APLLeave();

// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        if ( APLIsLeaveComplete() )
        {
                if ( GetLastZError() == ZCODE_NO_ERROR )
                {
                        // Successfully left the network
   ...
```

APLIsLeaveComplete

This macro is used by the end device to determine if previously started leave process is complete or not.

Syntax

`BOOL APLIsLeaveComplete(void)`

Parameters

None

Return Values

`TRUE`, if leave process is complete

`FALSE`, if otherwise

Precondition

`I_AM_END_DEVICE` is defined.

Side Effects

None

Note

A return value of `TRUE` may not always mean that the leave attempt was successful. You must call `GetLastZError()` to determine if leave was indeed successful.

Example

```
...

// Start the leave process.
APLLeave();

// Enter into main application loop
while(1)
{
        // Let stack execute
        APLTask();

        if ( APLIsLeaveComplete() )
        {
                if ( GetLastZError() == ZCODE_NO_ERROR )
                {
                        // Successfully left the network
   ...
```

APPLICATION SUPPORT SUBLAYER (APS)

The APS layer primarily provides the ZigBee endpoint interface. An application would use this layer to open or close one or more endpoints and retrieve or send the data. It also provides primitives for Key Value Pair (KVP) and Message (MSG) data transfer.

The APS layer also maintains the binding table. The binding table provides a logical link between the endpoint and cluster ID pair between two nodes in a network. When a coordinator is first programmed, the binding table is empty. The main application must call the appropriate binding APIs to create new binding entries. See the `DemoCoordApp` and `DemoRFDApp` demo applications for working examples. The Microchip APS layer stores binding tables into Flash memory. The Flash programming routines are located in the `zNVM.c` file. If required, you may replace NVM routines with your NVM-specific routines to support different types of NVM.

The APS layer also maintains an "indirect transmit buffer" RAM to store indirect frames until they are requested by intended recipients. According to the ZigBee specification, in a star network, an RFD device would always forward its data frame to the coordinator. The RFD device may not know about the intended recipient of that data frame. The exact recipient of a data frame is determined by the binding table entry. Once the coordinator receives the data frame, it looks up its binding table to determine the intended recipient. If a recipient exists for this data frame, it would place that frame in its indirect transmit frame buffer until the intended recipient explicitly requests it. Depending on the frequency of requests, the coordinator must keep the data frame in its indirect transmit frame buffer. The exact time is defined by the `MAC_MAX_DATA_REQ_PERIOD` compile time option defined in the `zigbee.def` file. Note that the longer a node takes to request its data, the longer the packet would need to remain in an indirect transmit buffer. A longer data request time would require a larger indirect buffer area. The indirect frame buffer consists of a design time allocated fixed size of RAM heap. The size of the buffer is defined by the `MAX_HEAP_SIZE` compile time option in the `zigbee.def` file. A new data frame is added by dynamically allocating RAM from the indirect transmit frame buffer. The dynamic memory management allows the efficient use of an indirect transmit frame buffer area. The dynamic memory management routines are located in the `SRAlloc.c` file.

`APSInit`

This function initializes APS layer by clearing its internal data variables. A user application does not need to call this function. Call to `APLInit` automatically calls this function.

Syntax

```
void APSInit(void)
```

Parameters

None

Return Values

None

Precondition

None

Side Effects

None

Note

None

Example

```
// Following is part of APLInit() in zAPL.c
APSInit();
...
```

APSTask

This is an APS cooperative task function. Application does not have to call this function – it is automatically called when application calls `APLTask`.

Syntax

```
BOOL APSTask(void)
```

Parameters

None

Return Values

`TRUE`, if there are no outstanding states to execute

`FALSE`, if otherwise

Precondition

None

Side Effects

None

Note

None

Example

```
// Following is part of APLTask() in zAPL.c
APSTask();
...
```

APSDisable

This macro clears any outstanding endpoint receive flags and prepares APS module for processor Sleep. The application does not have to call this macro directly – it is automatically called when `APLDisable` macro is called.

Syntax

```
void APSDisable(void)
```

Parameters

None

Return Values

None

Precondition

None

Side Effects

None

Note

None

Example

```
// Following is a definition of APLDisable macro
#define APLDisable()            … APSDisable() …
```

APSOpenEP

This function allows main application to open an endpoint. In ZigBee protocol, data is exchanged via endpoints.

Syntax

```
EP_HANDLE APSOpenEP(BYTE srcEP,
                BYTE clusterID,
                BYTE destEP,
                BOOL bDirect)
```

Parameters

`srcEP [in]`

Indicates the source endpoint number – must be between 0-31.

`clusterID [in]`

Indicates the cluster ID to which endpoint is bound.

`destEP [in]`

Indicates the destination endpoint to which the source EP is connected. This value is used when `bDirect` is set `TRUE`.

`bDirect [in]`

Indicates that this endpoint is a direct connection to the specified destination EP. The destination node depends on how current endpoint will be bound.

Return Values

Handle to an opened endpoint. Application would use this handle to access this endpoint in future.

Precondition

`APSInit()` is already called.

Side Effects

Marks an empty endpoint as “in use” and total available endpoint count is decreased by one.

Note

The endpoint is synonymous to one end of a virtual channel. To complete the virtual channel, there would be two endpoints – source and destination (i.e., one-to-one connection). In advanced applications, there may be more than one endpoint (i.e., one-to-many connection), in which case, a data packet may be received by multiple receivers. The current version of the Stack does not support one-to-many endpoints. In ZigBee terminology, a connection is defined by a source endpoint and a cluster identifier (or cluster ID). The cluster ID is simply a number that identifies a collection of data variables that will be exchanged over a virtual channel. You may also think of a cluster ID as an added virtual channel within the original virtual channel. As a result, now you may have multiple cluster IDs for a given set of source and destination endpoints, thus creating multiple virtual subchannels within one virtual channel. The virtual connection may be established via a coordinator (i.e., indirect connection) or directly to a node (i.e., direct connection). The current version of the Stack supports indirect connection and direct connection to/from an end device to the coordinator only. You may not use this version of the Stack to establish peer-to-peer connection between two end devices. When a connection is defined as direct (i.e., `bDirect = TRUE`), you must supply destination endpoint information. For indirect connection (i.e., `bDirect = FALSE`), destination endpoint information is not necessary. The exact destination endpoint will be defined by the binding process.

To complete the definition of a channel, we must also provide source and destination node address information. This function automatically uses the current node address as the source node; however, the destination node address is not defined. When you call this function, you are creating a part of a virtual channel that is missing the destination address information. As per the ZigBee specification, the originator of the channel does not have to know the destination node address. You must create a binding table entry into the coordinator node that binds the source endpoint from this node to a destination endpoint on another node of your choice.

Example

```
EP_HANDLE hMyEP;
hMyEP = APSOpenEP(20, 5, 0, FALSE); // srcep = 20, clusterid = 5, destEP = N/A, Indirect
```

APSSetEP

This function sets the given endpoint as an active endpoint. All subsequent function calls will operate on this (i.e., active) endpoint.

Syntax

```
void APSSetEP(EP_HANDLE h)
```

Parameters

`h` `[in]`

Endpoint handle that is to be set active.

Return Values

None

Precondition

None

Side Effects

After this call, `APSCloseEP()`, `APSPut()`, `APSBeginMSG()`, `APSBeginKVP()` and other endpoint related functions operate on given endpoint.

Note

None

Example

```
// Set hMyEP as active endpoint
APSSetEP(hMyEP);
...
```

APSCloseEP

This function allows the main application to close an already open endpoint that is currently active.

Syntax

```
void APSCloseEP(void)
```

Parameters

None

Return Values

None

Precondition

`APSSetEP()` is called.

Side Effects

Marks given endpoint as free.

Note

None

Example

```
// Must first set the desired EP as an active to make sure that you close only that EP
APSSet(hMyEP);
// Close active endpoint
APSCloseEP();
...
```

APSBeginMSG

This function begins MSG data frame. MSG is a special data transfer mechanism defined by the ZigBee specification. The MSG format allows an application to transfer a stream of binary data bytes. MSG is useful for transferring a proprietary data stream or file data. See the ZigBee specification for more detail.

Syntax

```
TRANS_ID APSBeginMSG(BYTE length)
```

Parameters

`length  [in]`

Total number of bytes to be sent in this MSG frame.

Return Values

A sequential transaction ID to identify current MSG frame.

`TRANS_ID_INVALID` if there was an error beginning the MSG frame.

Precondition

`APSSetEP()` is called.

Side Effects

None

Note

Application must know the exact number of bytes it wants to send as part of the MSG service. After this function call, The application must load actual data bytes using `APSPut`.

Example

```
// Set the active EP.
APSSetEP(hMyEP);

// Initiate MSG frame of 20 data bytes
myTransID = APSBeginMSG(20);
...
```

APSBeginKVP

This function begins a Key Value Pair (KVP) data frame. KVP is a special data transfer mechanism defined by the ZigBee specification. KVP allows the main application to transfer variable-value pair data. KVP is useful when the data being exchanged is simple variable, value format. For example, a node with a temperature sensor may exchange temperature channel and corresponding temperature value. Typically, KVP is used by the standard ZigBee profile applications to standardize the format and the meaning of data transfer. See ZigBee specification for more detail.

Syntax

```
TRANS_ID APSBeginKVP(TRANS_TYPE type,
                     TRANS_DATA dataType,
                     WORD attribId)
```

Parameters

`type [in]`

Transaction type. Must be one of the following:

Type	Purpose
TRANS_SET	Set a variable value
TRANS_EVENT	Set an event
TRANS_GET_ACK	Request an ACK
TRANS_SET_ACK	Send an ACK
TRANS_EVENT_ACK	Send an event ACK
TRANS_GET_RESP	Send Get response
TRANS_SET_RESP	Send Set response
TRANS_EVENT_RESP	Send event response

`dataType[in]`

Transaction data type. Must be one of the following:

Type	Purpose
TRANS_NO_DATA	There is no value
TRANS_UINT8	Contains 8-bit unsigned value
TRANS_INT8	Contains 8-bit signed value
TRANS_ERR	An error has occurred
TRANS_UINT16	Contains 16-bit unsigned value
TRANS_INT16	Contains 16-bit signed value
TRANS_SEMI_PRECISE	Contains 16-bit semi-precise value – based on IEEE 754
TRANS_ABS_TIME	Contains 32-bit value – number of seconds passed since January 1, 2000

`attribId[in]`

16-bit attribute ID that specifies specific attribute within the target node.

Return Values

Sequential transaction ID used to send this transaction. `TRANS_ID_INVALID` if there was an error beginning the KVP frame.

Precondition

`APSSetEP()` is called.

Side Effects

This function does not start the frame transmission. You must call zero or more `APSPut` as per attribute value requirement, followed by `APSSend` to begin the transmission.

Note

None

Example

```
// First of all make sure that desired EP is made active.
APSSetEP(hMyEP);

// Assume that current node has two attributes - '0' meaning that switch is pushed and
// '1' meaning that switch is open.
// We want to send our switch status to a remote node that is already properly bound.
// When we first opened the EP, we had already specified srcEP and clusterID.
// As a result, we just have to load the current attribute value
// Since the attribute itself explains the state of the switch, we will not transmit any
// data for this attribute. You may have an attribute (e.g. Temperature) that might
// assume different values. In that case, you will pass appropriate TRANS_DATA type and
// load corresponding value using APSPut().
APSBeginKVP(TRANS_SET, TRANS_NO_DATA, swValue);
...
```

APSIsPutReady

This function is used by the application to determine if it is okay to load a new frame.

Syntax

`BOOL APSIsPutReady(void)`

Parameters

None

Return Values

`TRUE`, if it is okay to start loading a new frame

`FALSE`, if otherwise

Precondition

`APSSetEP()` is called.

Side Effects

None

Note

None

Example

```
// Set the active EP
APSSetEP(hMyEP);

// Check to see if we can load new frame
if ( APSIsPutReady() )
{
        APSPut(0x55);
        APSPut(0xaa);
}
...
```

APSSetClusterID

This function sets the cluster ID associated for the current endpoint transmission.

Syntax

```
void APSSetClusterID(BYTE clusterID)
```

Parameters

`clusterID[in]`

Cluster ID to be set.

Return Values

None

Precondition

`APSIsGetReady() = TRUE`

Side Effects

None

Note

Endpoint number and cluster ID form a unique pair. ZigBee applications communicate using this unique pair. ZigBee specification recommends that application maintain a binding entry for each such unique pair. If there exists an endpoint that communicates using multiple cluster ID, the ZigBee specification recommends that application maintain multiple binding entries even if they are all associated with one endpoint. To conserve RAM usage, the Microchip Stack, instead, uses only one binding entry per an endpoint no matter how many cluster IDs it uses. As a result, if an endpoint uses multiple cluster IDs, it must specifically set a cluster ID for each of its transmissions. Similarly, it would fetch the cluster ID from a received frame and process it accordingly.

Example

```
// Set active EP
APSSetEP(hMyEP);

// Check to see if we can load new frame
if ( APSIsPutReady() )
{
        // A cluster ID value is set when we first called APSOpenEP. However, if an EP
        // uses multiple cluster ids to communicate, we would set cluster ID here.
        APSSetClusterID(0x02);

        APSPut(0x55);
        APSPut(0xaa);
}
...
```

APSPut

This function loads given byte into the current transmit buffer.

Syntax

```
void APSPut(BYTE v)
```

Parameters

```
v       [in]
```

Data byte that is to be loaded into frame buffer.

Return Values

None

Precondition

```
APSIsPutReady() == TRUE
```

Side Effects

None

Note

This function simply loads the given bytes into the transmit buffer. You must call `APSSend` to begin the frame transmission.

Example

```
// Set active EP
APSSetEP(hMyEP);

// Check to see if we can load new frame
if ( APSIsPutReady() )
{
        APSPut(0x55);
        APSPut(0xaa);
}
...
```

APSPutArray

This function loads a given array of bytes into transmit buffer.

Syntax

```
void APSPutArray(BYTE *v, BYTE length)
```

Parameters

`v` `[in]`

Data bytes that are to be loaded into transmit buffer.

`length` `[in]`

Number of bytes to be loaded into transmit buffer.

Return Values

None

Precondition

`APSIsPutReady() == TRUE`

Side Effects

None

Note

This function simply loads the given bytes into the transmit buffer. You must call `APSSend` to begin the frame transmission.

Example

```
// Set active EP
APSSetEP(hMyEP);

// Check to see if we can load new frame
if ( APSIsPutReady() )
{
        // Prepare the array
        myBuffer[0] = 0x55;
        myBuffer[1] = 0xaa;

        // Now load it.
        APSPutArray(myBuffer, 2);
}
...
```

APSSend

This function transmits the current transmit buffer.

Syntax

`void APSSend(void)`

Parameters

None

Return Values

None

Precondition

`APSIsPutReady() == TRUE` and at least one `APSPut()` or `APSPutArray()` is called.

Side Effects

None

Note

Once a frame is transmitted, its frame sequence number is held in the Acknowledgement queue to identify its Acknowledgement frame. Caller must repeatedly call `APSIsConfirmed` to check for Acknowledgement and remove it from queue after it is Acknowledged or timed out.

Example

```
// Set the active EP
APSSetEP(hMyEP);

// Check to see if we can load new frame
if ( APSIsPutReady() )
{
        myBuffer[0] = 0x55;
        myBuffer[1] = 0xaa;

        APSPutArray(myBuffer, 2);

        // Send it
        APSSend();
}
...
```

APSIsConfirmed

This function checks to see if active frame is Acknowledged by remote node.

Syntax

`BOOL APSIsConfirmed(void)`

Parameters

None

Return Values

`TRUE`, if active frame is Acknowledged

`FALSE`, if otherwise

Precondition

`I_AM_END_DEVICE` is defined and `APSSetEP()` is called.

Side Effects

None

Note

This function is available to end devices only. An end device sends all of its data frame to the coordinator and it must wait for an Acknowledgment from the coordinator. If a coordinator sends data frame, it is immediately held in the indirect transmit buffer until the intended recipient end device explicitly polls it. That is why when a coordinator sends data frame, it is said to be Acknowledged immediately and there is no need to confirm it by calling `APSIsConfirmed`.

Example

```
// Set active EP
APSSetEP(hMyEP);

// Check to see if active frame is acknowledged.
if ( APSIsConfirmed() )
{
        // Now remove it
        APSRemoveFrame();
}
...
```

APSIsTimedOut

This function checks to see if Acknowledgement time-out has occurred.

Syntax

`BOOL APSIsTimedOut(void)`

Parameters

None

Return Values

`TRUE`, if active frame is timed out

`FALSE`, if otherwise

Precondition

`APSSend()` is called.

Side Effects

None

Note

None

Example

```
// Set the active EP
APSSetEP(hMyEP);

// Check to see if active frame is timed out.
if ( APSIsTimedOut() )
{
        // Now remove it
        APSRemoveFrame();
}
...
```

APSRemoveFrame

This macro removes the active frame from the Acknowledgement queue.

Syntax

`void APSRemoveFrame(void)`

Parameters

None

Return Values

None

Precondition

`APSSend()` is called.

Side Effects

None

Note

When a frame is first sent, its frame sequence number is stored in Acknowledgement queue managed by MAC layer. When an Acknowledgement frame is received, the MAC layer compares its frame sequence number with items in Acknowledgement queue. If a match is found, corresponding entry in queue is said to be confirmed. When a frame is Acknowledged or timed out or not needed, the application must call this function to remove it from the queue.

Example

```
// Set the active EP
APSSetEP(hMyEP);

// Check to see if active frame is acknowledged.
if ( APSIsConfirmed() )
{
        // Now remove it
        APSRemoveFrame();
}
...
```

APSIsGetReady

This function checks to see if the active endpoint has any receive data pending.

Syntax

`BOOL APSIsGetReady(void)`

Parameters

None

Return Values

`TRUE`, if some receive data is pending

`FALSE`, if otherwise

Precondition

`APSSetEP()` is called.

Side Effects

None

Note

None

Example

```
// Check to see if hMyEP has received any data
APSSetEP(hMyEP);
if ( APSIsGetReady() )
{
        // Get it
        myDataByte = APSGet();

        ...

        // Now discard it
        APSDiscardRx();
}
...
```

APSGetDataLen

This function retrieves remaining data bytes in current receive frame associated with the active endpoint.

Syntax

```
BYTE APSGetDataLen(void)
```

Parameters

None

Return Values

Number of bytes remaining to be fetched.

Precondition

```
APSIsGetReady() = TRUE
```

Side Effects

None

Note

None

Example

```
// Check to see if hMyEP has received any data
APSSetEP(hMyEP);
if ( APSIsGetReady() )
{
        // Get total data bytes in this frame
        myDataLen = APSGetdataLen();

        // Get it all
        APSGetArray(myData, myDatLen);

        ...

        // Now discard it
        APSDiscardRx();
}
...
```

APSGet

This function fetches one data byte from the receive buffer.

Syntax

```
BYTE APSGet(void)
```

Parameters

None

Return Values

Actual data byte that was fetched.

Precondition

```
APSIsGetReady() = TRUE
```

Side Effects

None

Note

None

Example

```
// Check to see if hMyEP has received any data
APSSetEP(hMyEP);
if ( APSIsGetReady() )
{
        // Get it
        myDataByte = APSGet();

        ...

        // Now discard it
        APSDiscardRx();
}
...
```

APSGetArray

This function retrieves an array of data bytes from the receive buffer.

Syntax

```
BYTE APSGetArray(BYTE *buffer, BYTE count)
```

Parameters

`buffer  [out]`

Buffer to hold the array of data.

`count  [in]`

Total number of bytes to fetch.

Return Values

Actual number of bytes fetched.

Precondition

```
APSIsGetReady() = TRUE
```

Side Effects

None

Note

None

Example

```
// Check to see if hMyEP has received any data
APSSetEP(hMyEP);
if ( APSIsGetReady() )
{
        // Get total data bytes in this frame
        myDataLen = APSGetdataLen();

        // Get it all
        APSGetArray(myData, myDatLen);

        ...

        // Now discard it
        APSDiscardRx();
}
...
```

APSDiscardRx

This function removes current received frame from the receive buffer.

Syntax

```
void APSDiscardRx(void)
```

Parameters

None

Return Values

None

Precondition

```
APSIsGetReady() = TRUE
```

Side Effects

None

Note

None

Example

```
// Check to see if hMyEP has received any data
APSSetEP(hMyEP);
if ( APSIsGetReady() )
{
        // Get total data bytes in this frame
        myDataLen = APSGetdataLen();

        // Get it all
        APSGetArray(myData, myDatLen);

        ...

        // Now discard it
        APSDiscardRx();
}
...
```

APSGetClusterID

This function retrieves a cluster ID associated for current endpoint reception.

Syntax

```
BYTE APSGetClusterID(void)
```

Parameters

None

Return Values

Cluster ID associated with received endpoint data.

Precondition

```
APSIsGetReady() = TRUE
```

Side Effects

None

Note

None

Example

```
// Check to see if hMyEP has received any data
APSSetEP(hMyEP);
if ( APSIsGetReady() )
{
        // Retrieve cluster ID in this data frame.
        myClusterID = APSGetCluserID();

        // Get total data bytes in this frame
        myDataLen = APSGetdataLen();

        // Get it all
        APSGetArray(myData, myDatLen);

        ...

        // Now discard it
        APSDiscardRx();
}
...
```

NETWORK LAYER

The Network Layer (NWK) is responsible to establish and maintain network connection. It independently handles incoming data request, association, disassociation and orphan notification requests.

A typical application would not need to make direct calls to the NWK layer. If required, you may refer to the NWK.c and NWK.h files for a detailed description of NWK APIs.

ZigBee DEVICE OBJECT

The ZigBee Device Object (ZDO) opens and handles the EP 0 interface. The ZDO is responsible for receiving and processing various requests from a remote device. Unlike other endpoints, EP 0 is always opened on start-up and assumed to be bound to any incoming data frames that are directed to EP 0.

The ZDO object allows remote device management services. A remote device manager would issue requests to EP 0 and the ZDO would process those requests. Some examples of remote services are NWK_ADDR_REQ (give me your network address), NODE_DESC_REQ (give me your node descriptor) and BIND_REQ (bind this source EP, cluster ID and destination EP). Some of the requests are available to the coordinator device only. Refer to the ZDO.c file for the complete list. You should also refer to the ZigBee specification for more information.

An application would not need to call any of the ZDO functions directly except to initialize it on start-up. If required, you may refer to the ZDO.c and ZDO.h files for detailed descriptions of ZDO APIs.

ZigBee DEVICE PROFILE LAYER

The ZigBee device profile layer provides standard ZigBee profile services. It defines and processes descriptor requests. A remote device may request any of the standard descriptor information via the ZDO interface. Upon receiving such requests, ZDO would call a profile object to retrieve the corresponding descriptor value. In the current version, the device profile layer is not fully implemented.

An application would not need to call any of the profile functions directly. If required, you may refer to the zProfile.c and zProfile.h files for detailed description of APIs.

MEDIUM ACCESS CONTROL LAYER

The Medium Access Control (MAC) layer implements functions required by the IEEE 802.15.4 specification. The MAC layer is responsible for interacting with the Physical Layer (PHY). In order to support different types of RF transceivers, the Microchip Stack separates PHY interactions in its separate file. There is a separate file for each supported transceiver. Note that due to the differences in RF transceiver capabilities, MAC and PHY files are not completely independent. The MAC file adjusts some of its logic based on the current RF transceiver.

In the current version of the Stack, the zPHYCC2420.c file implements Chipcon CC2420, 2.4 GHz transceiver-specific functions. In the future, as support for new RF transceivers is added, a new PHY file will be added. All RF transceiver PHY files use the zPHY.h file as their main interface to the MAC layer.

A typical application would not need to make direct calls to the MAC layer. If required, you may refer to the MAC.c, MAC.h, and PHY.h files for detailed descriptions of MAC/PHY APIs.

CONCLUSION

The Microchip Stack for 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. In addition, it automatically supports MPLAB C18 and Hi-Tech PICC-18 C compilers. If required, it can be easily modified to support other compilers.

REFERENCES

- *"Chipcon CC2420"*
  http://www.chipcon.com
- *"ZigBee™ Protocol Specification"*
  http://www.zigbee.org
- *"PICDEM™ Z Demo Kit User's Guide"* (DS51524)
  http://www.microchip.com
- *"IEEE 802.15.4 Specification"*
  http://www.ieee.org

SOURCE CODE

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

www.microchip.com

ANSWERS TO COMMON QUESTIONS

Q: Is the Microchip Stack for the ZigBee protocol ZigBee protocol-compliant?

A: No. The current version of the Microchip Stack is not ZigBee protocol-compliant.

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

A: You may download it from the Microchip web site (www.microchip.com) from either the AN965 or PICDEM Z page.

Q: How do I get target hardware design files?

A: You may download it from the PICDEM Z page on the Microchip web site.

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

A: You would need:

- At least one PICDEM Z demo kit or at least two of your own ZigBee nodes
- Complete source code for the Microchip Stack for the ZigBee protocol
- Either the MPLAB C18 or Hi-Tech PICC-18 C Compiler
- MPLAB IDE software
- A device debugger and programmer such as MPLAB ICD 2

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

A: The exact program and data memory requirement depends on the type of node and compiler selected. In addition, the sizes may change as new features and improvements are added. Please refer to `version.log` file more detail.

Q: What is the minimum processor clock requirement for running the ZigBee coordinator and end devices?

A: Normally, the ZigBee coordinator should run at a higher speed as it must be prepared to handle packets from multiple nodes. The coordinator clock speed depends on the number of the nodes in the network. The demo coordinator uses 16 MHz (4 MHz with 4 x PLL) and found to be supporting at least 10 nodes. We have not characterized the clock frequency and number of nodes in the network. The end device does not have to run as fast as the coordinator. A simple end device may be run at just 4 MHz.

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

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

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

A: The exact radio range depends on the type of RF transceiver and the type of antenna in use. For a Chipcon 2.4 GHz-based node with a PCB trace antenna, you should expect to get about a 100-meter line-of-sight. When placed inside a building, the actual range would reduce 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: You would need to develop one ZigBee coordinator and one more ZigBee 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 coordinator and sensor devices would become ZigBee end devices. You must make sure that the radio range offered by a specific RF transceiver is acceptable to your application.

Q: I have now created a coordinator and multiple end devices. How do I bind the devices so I can transfer data among devices?

A: There are two approaches to this issue. You may either write your custom binding logic or use a standard ZigBee device manager interface to create/modify a binding table entry in the coordinator table. At the time this document was published, there was no standard ZigBee device manager available on the market. The demo applications included with this application note use a custom binding mechanism, where binding requests are sent using a MSG frame. Refer to the custom binding functions, `InitCustomBind`, `StartCustomBind` and `IsCustomBindComplete`, in the `DemoRFDApp.c file`. The demo coordinator also contains logic to process custom binding requests. See the `ProcessCustomBind` function in the `DemoCoordApp.c` file for more information.

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

A: The IEEE 802.15.4 specification is freely available from the IEEE web site.

At the time this document was published, the ZigBee specifications were not released and were available to ZigBee members only. Check the ZigBee web site for information on obtaining the released specifications.

NOTES:

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

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

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

Trademarks

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

Microchip received ISO/TS-16949:2002 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona and Mountain View, California in October 2003. The Company's quality system processes and procedures are for its PICmicro® 8-bit MCUs, 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
Alpharetta, GA
Tel: 770-640-0034
Fax: 770-640-0307

Boston
Westford, MA
Tel: 978-692-3848
Fax: 978-692-3821

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

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

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

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

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

San Jose
Mountain View, CA
Tel: 650-215-1444
Fax: 650-961-0286

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

ASIA/PACIFIC

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: 86-28-8676-6200
Fax: 86-28-8676-6599

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

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

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

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

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

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

China - Qingdao
Tel: 86-532-502-7355
Fax: 86-532-502-7205

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-2229-0061
Fax: 91-80-2229-0062

India - New Delhi
Tel: 91-11-5160-8631
Fax: 91-11-5160-8632

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

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

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

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

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

Taiwan - Hsinchu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

EUROPE

Austria - Weis
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

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

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

Germany - Ismaning
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

England - Berkshire
Tel: 44-118-921-5869
Fax: 44-118-921-5820

10/20/04