AN965

Microchip ZigBee™ 协议栈

作者： *Nilesh Rajbharti*
Microchip Technology Inc.

引言

ZigBee™ 是专为低速率传感器和控制网络设计的无线网络协议。有许多应用可从 ZigBee 协议受益，其中可能的一些应用有：建筑自动化网络、住宅安防系统、工业控制网络、远程抄表以及 PC 外设。

与其他无线协议相比， ZigBee 无线协议提供了低复杂性、缩减的资源要求，最重要的是它提供了一组标准的规范。它还提供了三个工作频带，以及一些网络配置和可选的安全功能。

如果您正在寻求现有的控制网络技术 （例如 RS-422、RS-485）或专有无线协议的替代方案， ZigBee 协议可能是您所需的解决方案。

此应用笔记旨在帮助您在应用中采用 ZigBee 协议。可以使用在应用笔记中提供的 Microchip ZigBee 协议栈快速地构建应用。为了说明该协议栈的用法，本文包含了两个有效的演示应用程序。可将这两个演示程序作为参考或者根据您的需求经过简单修改来采用它们。

此应用笔记中提供的协议栈函数库实现了一个与物理层无关的应用程序接口。因此，无需做重大修改就可以轻松地在射频（Radio Frequency， RF）收发器之间移植应用程序。

在此文档末尾的**“常见问题解答”**中提供了有关 Microchip 协议栈和用法的一些常见问题及其答案。

假设

此文档假设您熟悉 C 编程语言。文档中大量使用了有关 ZigBee 和 IEEE 802.15.4 规范的术语。此文档没有详细讨论 ZigBee 规范，只提供了对 ZigBee 规范的简要概述。建议您仔细阅读 ZigBee 和 IEEE 802.15.4 规范。

特性

Microchip ZigBee 协议栈设计为随着 ZigBee 无线协议规范的发展而发展。在发布此文档时，该协议栈的 1.0 版本具有以下特点（欲知最新特性，请参阅源代码版本日志文件 `version.log`）：

- 基于 ZigBee 规范的 0.8 版本
- 使用 Chipcon CC2420 RF 收发器支持 2.4 GHz 频带
- 支持简化功能设备 （Reduced Function Device，RFD）和协调器
- 在协调器节点中实现对邻接表和绑定表的非易失性存储
- 支持非时隙的星型网络
- 可以在大多数 PIC18 系列单片机之间进行移植
- 协同多任务处理架构
- 不依赖于 RTOS 和应用
- 支持 Microchip MPLAB® C18 和 Hi-Tech PICC-18™ C 编译器
- 易于添加或删除特定模块的模块化设计

限制

Microchip 协议栈的 1.0 版本包含以下限制。请注意随着时间的推移， Microchip 会添加新特性。如需了解目前的限制，请参阅源代码版本日志文件（`version.log`）。

- 不完全符合 ZigBee 协议
- 不支持群集和点对点网络
- 无安全和访问控制功能
- 无路由器功能
- 不提供标准的配置文件；但是包含创建配置文件必需的所有原始函数
- 不支持一对多绑定

典型的 ZigBee 节点硬件

要使用 Microchip 协议栈来创建典型的 ZigBee 节点，至少必须具备以下组件：

- 一片带 SPI™ 接口的 PIC18F 单片机
- 一个带有所需外部元件的 RF 收发器 （如需了解所支持的收发器，请参见 `version.log`）
- 一根天线，可以是 PCB 上的引线形成的天线或单极天线

如图 1 所示，控制器通过 SPI 总线和一些离散控制信号与 RF 收发器相连。控制器充当 SPI 主器件而 RF 收发器充当从器件。控制器实现了 IEEE 802.15.4 MAC 层和 ZigBee 协议层。它还包含了特定应用的逻辑。它使用 SPI 总线与 RF 收发器交互。Microchip 协议栈提供了完全集成的驱动程序，免除了主应用程序管理 RF 收发器功能的任务。如果您在使用 Microchip ZigBee 节点的参考原理图，那么无需做任何修改就可以开始使用 Microchip 协议栈了。如果需要，可以将某些非 SPI 控制信号重新分配到其他端口引脚以适合你的应用的硬件。在这种情况下，必须修改物理层接口定义来包括正确的引脚分配。

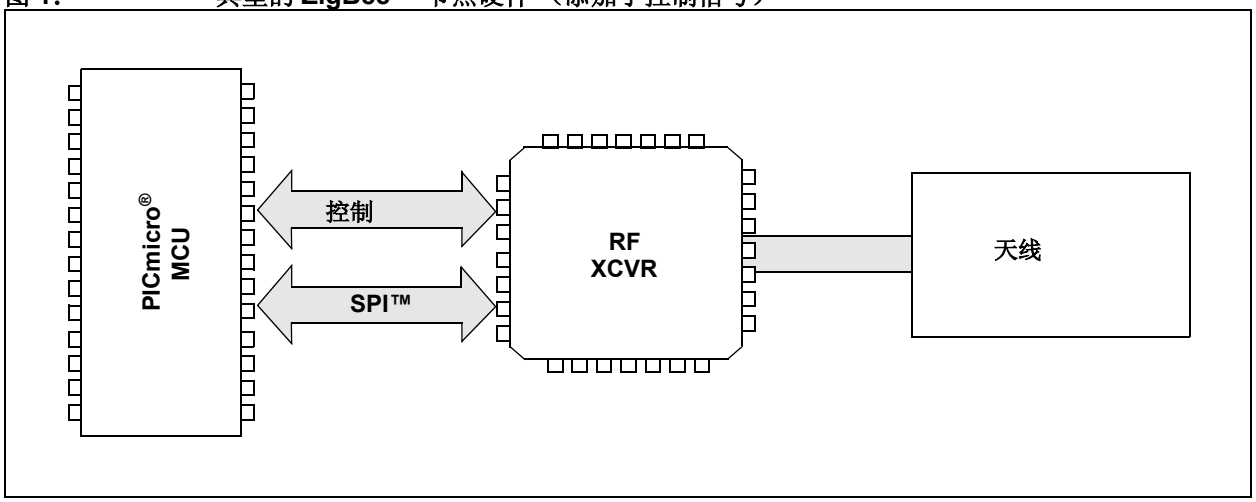
1.0版的Microchip协议栈使用由Chipcon生产的CC2420 RF 收发器。 CC2420 实现了 2.4 GHz 的物理层和一些MAC功能。在Chipcon网站上可以了解更多有关CC2420的信息 （参见 **“参考书目”**）。

Microchip 的 ZigBee 协议参考设计实现了 PCB 引线天线和单极天线两种设计方案。根据您对天线的选择，将可能必须去除或增加一些元件。如需了解更多信息，请参阅 *“PICDEM™ Z Demo Kit User's Guide ”*（参见**“参考书目”**）。

CC2420 需要 3.3V 的电源电压。Microchip 参考设计的控制器和 RF 收发器均使用 3.3V 的电源电压。如果需要，可以将此设计修改为控制器使用 5V 电源电压，RF 收发器使用 3.3V 电源电压。当对控制器使用 5V 电源电压时，在它与 CC2420 之间必须使用逻辑电平转换电路。根据需要，可以使用交流电源或电池。通常，ZigBee 协调器由交流电源供电，而终端设备由电池供电。当使用电池供电时，要确保 CC2420 的电源电压在规定电压范围内。

有关 Microchip ZigBee 节点的参考设计，请参阅 *“PICDEM Z Demo Kit User's Guide ”*。

图 1： 典型的 ZigBee™ 节点硬件 （添加了控制信号）



对 PIC® 单片机的资源要求

Microchip 协议栈使用下列 I/O 引脚与 RF 收发器接口：

表 1： PIC® MCU 与 RF 收发器之间的接口

PIC® 单片机 I/O 引脚	RF 收发器引脚
RB0 （输入）	CC2420: FIFO
RB1 （输入）	CC2420: CCA （未使用）
RB2 （输入）	CC2420: SFD
RB3 （输入）	CC2420: FIFOP
RC0 （输出）	CC2420: CSn
RC1 （输出）	CC2420: VREG_EN
RC2 （输出）	CC2420: RESET
RC3 （输出）	CC2420: SCK
RC4 （输入）	CC2420: SO
RC5 （输出）	CC2420: SI

如需了解对程序和数据存储器的完整要求，请参阅协议栈源文件安装目录中的文档 version.log。

安装源文件

可从 Microchip 网站下载完整的 Microchip 协议栈源文件（参见 **“源代码”**）。源代码以一个 Windows® 安装文件形式发布 （MpZBeeV1.00.00.exe）。

执行下列步骤完成安装：

1. 执行 MpZBeeV1.00.00.exe 文件； Windows 安装向导将指导您完成安装过程。
2. 在继续安装之前，必须通过单击 **I Accept**（我接受）接受软件许可协议。
3. 在完成安装过程之后，应该看到“Microchip Software Stack for ZigBee”程序组。完整的源代码将被复制到您的计算机的根驱动器的 MpZBee\Source 目录中。
4. 如需了解最新版本的特定功能和限制，请参阅文件 version.log。

源文件构成

Microchip 协议栈由多个源文件组成。很多源文件是所有 ZigBee 应用程序共用的，而有些源文件仅供某些特定的 ZigBee 应用程序使用。此外，协议栈文件还包括全部演示应用程序的所有源文件。

为了简化文件管理和应用程序开发，所有源文件均位于 Source 目录下的子目录中。下表给出了目录结构：

表 2： Source 目录结构

目录名称	内容
Stack	Microchip 协议栈源文件
DemoCoordApp	演示协调器应用程序源文件
DemoRFDApp	演示 RFD 应用程序源文件

许多协议栈文件包含了所有支持的ZigBee应用程序类型的逻辑；然而，根据 zigbee.def 文件中的预处理定义，只启用一组逻辑。使用一组共用的协议栈源文件和各自的 zigbee.def 文件可以开发多种 ZigBee 节点应用程序。例如， DemoCoordApp 和 DemoRFDApp 节点应用程序在它们各自的目录中有各自的 zigbee.def 文件。这种方法允许使用共用的源文件开发多种应用程序并根据针对应用程序的选项生成惟一的 hex 文件。

这种方法要求您在编译应用程序项目时提供搜索路径，包含应用程序和协议栈源文件目录中的文件。与此应用笔记一起提供的演示应用程序项目已经包含了必需的搜索路径信息。

演示应用程序

1.0 版的 Microchip 协议栈包含两个演示应用程序：

1. `DemoRFDApp`：演示典型的 ZigBee RFD 设备的应用程序。
2. `DemoCoordApp`：演示典型的 ZigBee 协调器的应用程序。

演示 RFD 的应用程序的功能

1.0 版的演示 RFD 的应用程序实现了下列功能：

- 旨在与 PICDEM Z 演示板一起使用
- 使用系统休眠和看门狗功能演示低功耗功能
- 使用 RS-232 终端驱动菜单命令来配置多个选项
- 通过终端菜单命令对 RF 收发器的性能进行测试的功能
- 在一个节点上可由用户配置的简单远程控制开关和 LED 应用程序
- 使用 D2 作为指示发送 / 接收操作的 LED
- 演示自定义绑定接口
- 自动支持 MPLAB C18 和 Hi-Tech PICC-18 编译器

演示协调器的应用程序的功能

1.0 版的演示协调器的应用程序实现了下列功能：

- 旨在与 PICDEM Z 演示板配合使用
- 使用 RS-232 终端驱动菜单命令来配置多个选项
- 通过终端菜单命令对 RF 收发器的性能进行测试的功能
- 创建非时隙的星型网络
- 使用 D2 作为指示发送 / 接收操作的 LED
- 演示自定义绑定接口
- 自动支持 MPLAB C18 和 Hi-Tech PICC-18 编译器

编译演示应用程序

可以使用 Microchip C18 或 Hi-Tech PICC-18 编译器编译包括在此应用笔记中的演示应用程序。总共有 4 个 MPLAB 项目文件，每个演示应用程序有两个。 MPLAB 项目文件名的前两个字母表示所使用的编译器的类型。例如，项目文件 `MpDemoCoordApp.mcp` 使用 MPLAB C18 编译器，而 `HtDemoCoordApp.mcp` 使用 Hi-Tech PICC-18 编译器。

除了要使用 PIC18F4620 作为器件外，所有的演示应用程序项目还使用在 MPLAB® IDE 的 “Build Options”（编译选项）中定义的附加包含路径。演示协调器项目使用 “`..\Stack`”和 “`..\DemoCoordApp`”，而演示 RFD 项目使用“`..\Stack`”和“`..\DemoRFDApp`”作为附加包含路径。如果正在使用 MPLAB IDE 重新创建任何演示应用程序项目，则必须手动地在 MPLAB 的 “Build Options”对话框中设置这些包含路径。

表 3 和表 4 分别列出了编译演示协调器和演示 RFD 的应用程序所必需的源文件。

表 3：　　演示协调器的应用程序项目文件

源文件	目录名称	用途
MpDemoCoordApp.mcp HtDemoCoordApp.mcp	DemoCoordApp	在 MPLAB® IDE 中使用的演示协调器的应用程序项目文件
DemoZCoordApp.c	DemoCoordApp	协调器的主应用程序文件
zigbee.def	DemoCoordApp	Microchip 协议栈编译时间选项文件
D1OnCoord.c	DemoCoordApp	LED D1 的端点任务，只针对协调器节点
S2OnCoord.c	DemoCoordApp	开关 S2 的端点任务，只针对协调器节点
18f4620i.lkr	DemoCoordApp	PIC® 单片机的 MPLAB C18 链接描述文件， Hi-Tech PICC-18 编译器不需要该文件
Console.c	Stack	仅供演示应用程序使用的 RS-232 终端程序
MSPI.c	Stack	SPI™ 主接口
NeighborTable.c	Stack	协调器上的邻接表和绑定表逻辑
SRAlloc.c	Stack	仅供协调器使用的动态存储管理器
Tick.c	Stack	供协议栈使用的节拍 （tick）管理器，可用于应用程序
zAPL.c	Stack	ZigBee™ 应用层
zAPS.c	Stack	ZigBee 应用支持子层
ZDO.c	Stack	ZigBee 设备对象
zMAC.c	Stack	IEEE 802.15.4 MAC 层
zNVM.c	Stack	非易失性存储器存储程序
zNWK.c	Stack	ZigBee 网络层
zPHYCC2420.c	Stack	针对 CC2420 的物理层程序
zProfile.c	Stack	ZigBee 配置程序

表 4：　　演示 RFD 的应用程序项目文件

源文件	目录名称	用途
MpDemoRFDApp.mcp HtDemoRFDApp.mcp	DemoRFDApp	供 MPLAB® IDE 使用的演示 RFD 的应用程序项目文件
DemoZRFDApp.c	DemoRFDApp	RFD 的主应用程序文件
zigbee.def	DemoRFDApp	Microchip 协议栈编译时间选项文件
D1OnEndDevice.c	DemoRFDApp	LED D1 的端点任务，只针对终端设备
S2OnEndDevice.c	DemoRFDApp	开关 S2 的端点任务，只针对终端设备
18f4620i.lkr	DemoCoordApp	PIC® 单片机的 MPLAB C18 链接描述文件， Hi-Tech PICC-18 编译器不需要该文件
Console.c	Stack	仅供演示应用程序使用的 RS-232 终端程序
MSPI.c	Stack	SPI™ 主接口
Tick.c	Stack	供协议栈使用的节拍管理器，可用于应用程序
zAPL.c	Stack	ZigBee™ 应用层
zAPS.c	Stack	ZigBee 应用支持子层
ZDO.c	Stack	ZigBee 设备对象
zMAC.c	Stack	IEEE 802.15.4 MAC 层
zNVM.c	Stack	非易失性存储器存储程序
zNWK.c	Stack	ZigBee 网络层
zPHYCC2420.c	Stack	针对 CC2420 的物理层程序
zProfile.c	Stack	ZigBee 配置程序

以下是编译演示应用程序的粗略步骤。此处假定您熟悉 MPLAB IDE 并且将使用 MPLAB IDE 来编译应用程序。如果情况并非如此，请参阅 MPLAB IDE 特定应用说明来创建、打开和编译项目。

1. 确认安装了 Microchip 协议栈的源文件。如果尚未安装，请参阅 **“安装源文件”**。
2. 启动 MPLAB IDE 并打开相应的项目文件：
   对于演示协调器的应用程序来说，该项目文件是 Source\DemoCoordApp\??DemoCoordApp.mcp ；
   对于演示 RFD 的应用程序来说，该项目文件是 Source\DemoRFDApp\??DemoRFDApp.mcp。
   项目文件的确切名称取决于您选择的编译器。MPLAB C18 编译器使用的项目文件名为“Mp*.mcp”，而 Hi-Tech PICC-18 编译器使用的项目文件名为 “Ht*.mcp”。
3. 使用 MPLAB IDE 菜单命令来编译项目。注意仅当源文件位于硬盘驱动器根目录的 MpZBee 目录中时，创建的演示应用程序项目才能正确工作。如果您已将源文件移动到了另一个位置，则必须重新创建项目或修改现有的项目设置才能编译项目。更多信息，请参见 **“编译演示应用程序”**。
4. 编译过程应该成功完成。如果未成功编译，请确认正确设置了 MPLAB IDE 和编译器。

将演示应用程序烧写到器件中

要使用两个演示应用程序之一对目标板进行编程，必须要有 PIC 编程器。下面的步骤假定您将使用 MPLAB ICD 2 作为编程器。如果使用其他编程器，请参阅具体编程器的说明。

1. 将 MPLAB ICD 2 连接到 PICDEM Z 演示板或您的目标板。
2. 对目标板通电。
3. 启动 MPLAB IDE。
4. 选择 PIC 器件（仅当导入以前编译生成的 hex 文件时需要）。
5. 使能 MPLAB ICD 2 作为编程器。
6. 如果您希望使用以前编译生成的 hex 文件，只要导入 `DemoCoordApp\MpDemoCoordApp.hex` 文件或 `DemoRFDApp\MpDemoRFDApp.hex` 文件即可。为了简化对演示协调器节点和演示 RFD 节点的辨别 （如果您在使用 PICDEM Z 板），推荐您将 `MpDemoCoordApp.hex` 文件烧写到贴有“COORD...”标签的单片机中；将 `MpDemoRFDApp.hex` 文件烧写到贴有“RFD...”标签的单片机中。如果正在对定制的硬件进行编程，请确保使用某种标识方法来标识协调器节点和 RFD 节点。
7. 如果正在重新编译 hex 文件，请打开相应的演示项目文件并遵循编译步骤来生成应用程序 hex 文件。
8. 两个演示应用程序文件都包含 PICDEM Z 演示板所需的必要的配置选项。如果正在对另一种类型的板编程，请确保从 MPLAB ICD 2 配置设置菜单选择了适当的振荡器模式。
9. 从 MPLAB Programmer（编程器）菜单选择 **Program**（编程）菜单选项，开始对目标板编程。
10. 数秒之后，应该看到“`Programming successful`”（编程成功）的消息。如果未看到这条消息，请仔细检查板和 MPLAB ICD 2 的连接。如需进一步的帮助，请参阅 MPLAB 在线帮助。
11. 除去板的电源并断开 MPLAB ICD 2 电缆与目标板的连接。
12. 重新为该板加上电源，并确认 D1 和 D2 LED 点亮。如果未点亮，请仔细检查您的编程步骤，如果需要还需重复这些编程步骤。

配置演示应用程序

如果这是您第一次运行这两个演示应用程序之一，必须首先为每块板分配一个惟一的节点 ID。节点 ID 是一个惟一的 4 位 10 进制数，用于产生惟一的 MAC 地址。两个演示应用程序都使用 Microchip 组织惟一标识符（Organizational Unique Identifier，OUI）编号编译。这些应用程序使用节点 ID 值来创建满足 IEEE 802.15.4 规范要求的惟一 64 位 MAC 地址。可通过以下网址申请获得您自己的 OUI 编号：

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

要配置演示应用程序，将需要下列工具：

1. 带有至少一个 RS-232 端口的 PC。
2. 一个基于 PC 的 RS-232 终端程序，如 Windows® 操作系统的超级终端。
3. 一条 DB-9 针式到孔式 RS-232 电缆。
4. 带 9V 电源的目标板。

编程节点 ID 值

对每个新编程的演示应用程序执行下列步骤（此步骤假定您使用 Microsoft® 超级终端程序。您可以选用任何终端程序，只要按要求设置了端口即可）：

1. 使用直插针式到孔式 DB9 RS-232 电缆将目标 PICDEM Z 板连接到计算机上的可用串口。
2. 通过选择*开始> 程序> 附件> 通讯*启动超级终端。
3. 在“连接描述”对话框中，为该连接输入任何便捷名称。单击**确定**。
4. 在“连接到”对话框中，选择与 PICDEM Z 板相连的 COM 端口。单击**确定**。
5. 用如下设置配置与 PICDEM Z 节点连接的串口：19200 bps、8 个数据位、1 个停止位、无奇偶校验、无数据流控制。
6. 单击**确定**以发起连接。
7. 通过选择*文件 > 属性*打开属性对话框。
8. 选择“设置”选项卡，并单击 **ASCII 码设置 ...**
9. 选中“本地回显键入的字符”。
10. 单击**确定**依次关闭所有打开的对话框。
11. 在按住 S3 按钮开关或在按住 RESET 和 S3 按钮开关的同时对节点通电，然后释放RESET开关。终端窗口中将出现如例 1 所示的配置菜单（实际的头文本将取决于您尝试重新配置的节点的类型和编译的日期）。
12. 输入 **1** 更改节点 ID 值。
13. 按照说明，输入节点 ID 值。
14. 按下节点上的 RESET 开关或输入 **0** 以退出配置模式并运行应用程序。

要确认新的节点ID值已被正确地保存，只要通过按下演示板上的 RESET 按钮使该板复位，并确认 D1 和 D2 LED *未*点亮即可。还要确认 RS-232 终端上未显示任何菜单。这样就确保了目标板已被正确编程，并且为进一步配置做好了准备。

如果这是新编程的演示 RFD 板，必须执行其他配置以观察全部的演示功能。

例 1：　　演示应用程序配置菜单

```
*************************************************************************
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:
```

编程绑定配置

此刻，您准备执行其余的配置。这要求您有一块演示协调器应用板和一块或多块演示 RFD 应用板。为了简化起见，此步骤假定您只有一块演示协调器板和一块 RFD 板。然而，您可以轻松地将此步骤扩展到任何数量的演示 RFD 板。

作为此配置的一部分，您将把演示 RFD 应用板 （终端设备）与演示协调器应用程序板关联起来。还将把一块板上的 S2 开关绑定到另一块板上的 D1 LED。在成功地完成此配置之后，即可使用这两个演示应用程序进行实验。

为了简化绑定配置，演示 RFD 应用程序提供了一个快速演示绑定菜单选项。快速绑定选项是一个单步绑定操作，演示终端设备和协调器之间的数据传输。除了快速绑定菜单选项之外，还可以使用板上开关来创建其他高级的绑定配置。

执行快速演示绑定

快速演示绑定选项将演示 RFD 板上的 S2 开关绑定到演示协调器上的 D1 LED，并将演示协调器上的 S2 开关绑定到演示 RFD 板上的 D1 LED。在完成快速演示绑定之后，您将能够通过按下演示 RFD 板上的 S2 开关控制演示协调器上的 D1 LED，通过按下演示协调器板上的 S2 开关控制演示 RFD 板上的 D1 LED。

快速演示绑定主要是为双节点（一个协调器和一个 RFD）网络而设计的。如果您在多块演示 RFD 板上执行快速演示绑定，那么演示协调器上的 D1 LED 将被任何演示 RFD 板控制。但是，演示协调器上的 S2 开关将仅控制最后一块执行快速演示绑定的演示 RFD 板上的 D1。

快速演示绑定选项需要使用至少带有一个标准串口和终端软件的 PC。如果您有两个串口和两条串行线缆可用，就可以同时从演示协调器和 RFD 板查看活动日志。如果您只有一个串口可用，则只能将 RFD 板连接到 PC。

执行下列步骤以执行快速演示绑定：

注：	下面的步骤假定演示 RFD 板连接到 COM1，且可将演示协调器板连接到 COM2。

1. 去除所有节点的电源。
2. **可选：**如果您有两个串口和两条串行线缆可用，选择启动一个基于 PC 的 RS-232 终端程序并选择具有如下设置的 COM1 端口：19200 bps，8-N-1，无数据流控制并回显输入的字符。
3. 找到演示协调器板并为其加上电源以启动正常执行模式。确认在 D1 和 D2 闪烁后，D2 短暂闪烁。**可选：**如果连接了一个串口，注意终端显示“`New network successfully started`”（新网络成功启动）的消息。
4. 选择启动一个基于 PC 的 RS-232 终端程序并选择具有以下这些设置的 COM2 端口：19200 bps，8-N-1，无数据流控制并回显输入的字符。
5. 现在，保持演示协调器板通电，在按住 S3 按钮开关时给演示 RFD 板上电，或者同时按住 RESET 和 S3 按钮开关，然后松开 RESET 开关。您应该看到终端程序输出窗口中的文本菜单。
6. 输入 **2** 以启动“`Join a network`”（加入一个网络）命令。注意终端显示“`Successfully associated`”（成功关联）消息。如果您没有看到此消息，请确认演示协调器节点已加电并且在正常模式下运行。**可选：**注意在连接到该演示协调器节点的第二个终端上将显示“`A new node has just joined`”（一个新节点刚连入）消息。
7. 此时，演示 RFD 节点已成功连入由演示协调器建立的网络。现在您准备开始执行快速演示绑定。
8. 输入 **3** 以启动“`Perform quick demo binding`”（执行快速演示绑定）命令。注意终端显示“`Demo binding complete`”（演示绑定完成）消息。如果您没有看到此消息，请确认演示协调器节点已加电并且在正常模式下运行。**可选：**注意在连接到该演示协调器节点的第二个终端上将显示“`Custom binding successful`”（自定义绑定成功）消息。
9. 输入 **0** 以“`Save current changes and exit configuration`”（保存当前更改并退出配置）。终端现在应该显示“`Rejoin successful`”（重新连入成功）消息。
10. 现在可按下演示 RFD 节点上的 S2 并观察演示协调器上 D1 LED 切换状态。同样，按下演示协调器节点上的 S2 并观察演示 RFD 节点上的 D1 LED 切换状态。当您在演示协调器节点上按下 S2 时，将发现演示 RFD 节点上的 D1 LED 并不会立即切换状态。这是由于演示 RFD 节点必须通过定期查询演示协调器才能获得其 LED 状态。查询周期取决于编程到演示 RFD 节点的看门狗预分频值。还应该注意到演示 RFD 节点和演示协调器上的 D2 都定期闪烁。这表明演示 RFD 节点定期查询演示协调器以获得其 D2 的状态。
11. 关联和快速演示绑定配置将永久地存储在演示协调器的闪存存储器中。

执行高级绑定

高级绑定操作使用板上开关在多块演示 RFD 板之间创建总共四种绑定配置的组合。高级绑定操作不需要终端和串行线缆。为了避免重复信息，以下步骤假定您希望在终端窗口上查看活动日志并且已经阅读了**“执行快速演示绑定”**部分和知道如何设置终端软件。

执行下列步骤以执行高级绑定：

1. 去除所有节点的电源。
2. 找到演示协调器板并为其通电以启动正常执行模式。确认在 D1 和 D2 闪烁后，D2 短暂闪烁。**可选：**如果连接了一个串口，注意终端显示“`New network successfully started`”（新网络成功启动）。
3. 保持演示协调器板通电，在按住 S3 按钮开关时给演示 RFD 板上电，或者同时按住 RESET 和 S3 按钮开关，然后松开 RESET 开关。**可选：**您应该看到终端程序输出窗口中的文本菜单。
4. 按下演示 RFD 节点上的 S2 以开始与演示协调器的关联过程。**可选：**终端窗口应该显示“`Successfully associated`”（成功关联）。
5. 如果您有多个演示 RFD 节点，则按下每个演示 RFD 节点上的 S2 以使它们与演示协调器节点关联。
6. 由于绑定配置可能有很多不同的组合，下表用来描述每种组合必须按顺序执行的步骤。
7. 按下每个演示 RFD 节点上的 RESET 开关以开始正常执行。如果连接到终端程序，注意将显示“`Rejoin successful`”（重新连入成功）消息。
8. 根据执行绑定的方式，按下某个节点上的 S2 来确认同一个节点或其他节点上的 D1 的切换状态。

表 5：　　绑定操作

绑定以下设备上的开关 S2	绑定以下设备上的 LED D1	结果
RFD：首先按住 S3，然后按下 S2 并释放 S2，随后释放 S3	**协调器**：首先按住 S3，然后按下 S2 并释放 S3，随后释放 S2	RFD 上的 S3 控制协调器上的 D1
协调器：首先按住 S3，然后按下 S2 并释放 S2，随后释放 S3	**RFD**：首先按住 S3，然后按下 S2 并释放 S3，随后释放 S2	协调器上的 S3 控制 RFD 上的 D1
RFD：首先按住 S3，然后按下 S2 并释放 S2，随后释放 S3	**RFD**：首先按住 S3，然后按下 S2 并释放 S3，随后释放 S2	RFD 上的 S3 控制同一个 RFD 上的 D1
RFD1：首先按住 S3，然后按下 S2 并释放 S2，随后释放 S3	**RFD2**：首先按住 S3，然后按下 S2 并释放 S3，随后释放 S2	RFD #1 上的 S3 控制 RFD #2 上的 D1
协调器：N/A	**协调器**：N/A	不允许

注　1： 当每一步执行时，各个节点上的 LED D1 和 D2 将从点亮 / 熄灭状态切换到熄灭 / 点亮状态。还要注意连接到 RFD 节点的终端程序将显示“`Attempting to bind...`”（尝试绑定 ...）消息，而连接到协调器节点的终端将显示“`Received valid...`”（有效接收 ...）消息。

2： 要完成绑定过程，必须执行“绑定设备上的开关 S2”和“绑定设备上的 LED D1”操作。

执行演示应用程序

在观察演示应用程序的功能之前，您必须至少有一块演示协调器板和一块演示 RFD 板。您还必须已经执行了在 **“配置演示应用程序”**中说明的配置。

演示应用程序实现了简单的远程控制开关和LED功能。有了此功能，您可以通过按下一块板上的开关来控制同一块或另一块演示 RFD 板上的 LED。

演示应用程序是完全独立的，不需要有与主机的接口。然而，如果您可以访问主机，则可以使用主机来查看应用程序的活动日志。与主机的接口对于了解和诊断可能遇到的设置问题很有用。

遵循以下步骤来执行演示应用程序：

1. 去除所有板的电源 （如果以前在通电的话）。
2. 定位演示协调器节点。
3. **可选：**将演示协调器节点连接到一个 PC 串口，并启动您喜欢的终端程序。选择具有以下设置的相应 COM 端口：19200 bps、8-N-1、无数据流控制并回显输入的字符。
4. 为演示协调器节点通电。观察到 D1 和 D2 同时闪烁，然后 D2 独自闪烁。如果连接到 PC，观察到终端程序显示“`New network successfully started`”（新网络成功启动）的消息。
5. 现在定位演示 RFD 节点。
6. **可选：**将 RFD 节点连接到一个 PC 串口，并启动您喜欢的终端程序。选择具有以下设置的相应 COM 端口：19200 bps、8-N-1、无数据流控制并回显输入的字符。
7. 在保持演示协调器节点通电的同时，为演示 RFD 节点加电。观察到 LED D1 和 D2 同时闪烁，然后 D2 多次闪亮。如果连接到 PC，则观察一到两秒钟后，终端程序显示“`Rejoin successful`”（重新连入成功）消息。如果您未看到任何消息或者看到“`Rejoin failed`”（重新连入失败）消息，请确认已经为协调器节点通电并运行正常；复位演示 RFD 节点，并再试一次。
8. 此时， RFD 节点已经与演示协调器节点成功关联。
9. 根据执行绑定配置的方式，按下演示 RFD 节点上的S2并观察同一个节点或其他节点上D1的状态。

演示应用的功能说明

演示协调器和演示 RFD 应用演示了一个简单的 ZigBee 网络。演示协调器和演示 RFD 应用形成了一个非时隙的星型网络。

当一个事先已编程了任一个演示应用程序的节点在启动时，这个演示节点将自动进入配置模式。必须使用终端接口来设置惟一的节点 ID。演示 RFD 节点需要额外的设置 （例如关联和绑定配置）以使其完全地发挥功能。

演示协调器的功能说明

上电时，演示协调器通过扫描空信道尝试建立一个新的网络。作为扫描过程的一步，演示协调器从当前频带的第一个信道开始发送BEACON_REQ帧。如果有另一个协调器处于同一信道，它将响应 BEACON_REQ，而原来的协调器将认为此信道已被占用。然后它会切换到下一个信道并重复该过程直到没有接收到任何对其BEACON_REQ 帧的响应时为止。一旦找到某个信道为空，它会选择一个随机的个人区域网络（Personal Area Network，PAN）ID 并开始侦听该信道。此时就建立了网络。从现在开始，如果另一个协调器广播BEACON_REQ 帧，我们原来的协调器将响应并声明其存在。

该演示协调器现在准备开始接受新的终端设备节点加入其网络。当一个新的终端设备希望加入网络时，它首先会发出BEACON_REQ以检测是否存在协调器。一旦终端设备确认了在某个特定的信道上存在协调器时，它会开始进行关联或孤立通知过程，以加入或重新加入网络。

实际的应用可能不会总是希望一直允许进行新的关联。例如，在一个基于 ZigBee 协议的控制网络中，可能不希望让任何新的传感器加入您的控制网络。您可能希望首先进入一个特殊的模式来控制新的关联。 Microchip 演示协调器对关联没有施加任何限制。任何设备可以在任何时间进行关联或解除关联。协调器不必进入一个特殊模式来执行这些操作。

演示协调器还实现了一个简单的开关和远程控制LED的接口。当协调器被编入程序之初，开关和 LED 没有绑定到任何目标。一旦执行正确的绑定过程，可以按下 S2 开关来控制其他终端设备上的 D1 LED。演示协调器实现了特殊的按键顺序以将板上开关和LED绑定到远程设备上。通过同时按下 S2 和 S3 并遵循 **“配置演示应用程序”**部分中简述的过程可以启动绑定过程。

演示 RFD 的功能说明

除了惟一的节点 ID 值之外，演示 RFD 应用程序要求您将演示 RFD 与附近的演示协调器关联。上电时，演示 RFD 节点尝试查找一个临近的演示协调器。它将扫描所有可用的信道来查找演示协调器。一旦找到了演示协调器，它会尝试重新加入网络。仅当此节点以前加入过该演示协调器的网络时，此次重新加入的尝试才会成功。这也就是为何在以正常模式执行演示 RFD 节点之前必须先将它加入到邻近节点的网络的原因。

该演示RFD节点被置于配置模式以加入一个新的网络。一旦该节点加入网络之后，您还必须将板上开关 S2 和 D1 LED 端点绑定到某些目标节点上。如需了解更多有关如何执行这些配置操作的信息，请参见 **“配置演示应用程序”**。

当完全地配置了演示 RFD 节点之后，在下一次正常执行启动时，它将自动尝试重新加入邻近的演示协调器的网络。由于过去它已经加入过该演示协调器的网络，该演示协调器将允许该演示 RFD 节点重新加入其网络。

一旦重新加入网络之后，演示 RFD 节点将使能看门狗定时器，禁止 RF 收发器并将控制器置于休眠状态。

当按下任何按钮时，演示 RFD 节点使用 PORTB 的电平变化中断功能来唤醒它自己。如果按下了 S2，它会发送一个具有当前开关状态的特殊的 MSG 数据帧（更多信息请参见 ZigBee 规范）到演示协调器。一旦演示协调器应答该 MSG 数据帧，演示 RFD 节点将重新回到休眠状态。

还可以通过看门狗超时唤醒演示 RFD 节点。在控制器退出休眠状态时，演示 RFD 节点将查询演示协调器以得到新的 S2 状态。如果有新的 S2 状态，演示 RFD 节点将相应地更新其 D1 LED。

为了进一步说明演示 RFD 节点如何接收其 S2 状态，假定演示 RFD 节点上的 S2 开关被绑定到同一个节点上的 D1 LED。此绑定配置允许我们通过按下同一块板上的 S2 来控制 D1。当您按下演示 RFD 节点上的 S2 时，该演示 RFD 节点将首先发出一个开关状态更新帧到演示协调器。此时，演示 RFD 节点不知道谁将接收到该开关更新帧。它只是将该帧定向到演示协调器并回到休眠状态。对于协调器来说，当它接收到开关状态帧时，它首先会在绑定表中查找以确定该帧是否有任何已知的目标。由于 S2 开关已被绑定到同一个演示 RFD 节点上的 D1 LED，所以该演示协调器将会发现该帧有一个已分配的目标，并且它会将当前的开关状态帧存储到其间接发送帧缓冲器中。如果没有绑定项，演示协调器将丢弃该帧。演示协调器会将开关状态帧保存到其间接发送帧缓冲器中，直到目标接收节点取出它或者发生超时为止。

假定我们原来的演示 RFD 节点被及时唤醒，并将查询请求发送到演示协调器。演示协调器将会查找其间接发送缓冲器并发现有一个帧正等待发送给该节点。然后它会将该开关状态帧发送给演示 RFD 节点，并等待应答。一旦该帧被应答，它会将该开关状态帧从其间接发送缓冲器中删除。

该演示 RFD 节点现在已接收到了它早些时候发出的新的开关状态。现在它将对该开关状态译码并相应地切换 LED 的状态。由于在第一次按下开关和查询开关状态之间有一个时间差，您将看到 LED 状态变化会有略微的延迟。

由于协调器存储帧并将其转发给相应的接收者，所以只要更改协调器存储器中的绑定表即可在完全不同的节点上控制 LED。

使用 MICROCHIP 协议栈

此应用笔记随附的文件包含 Microchip ZigBee 协议栈的完整源文件（参见 **“源代码”**）。这些源文件还包括了两个演示应用程序：一个 RFD 演示应用程序和一个演示协调器的应用程序。

所有基于 Microchip 协议栈的应用程序必须以协同多任务处理方式编写。协同多任务处理架构由一些按顺序执行的任务组成。协同任务会快速地执行其所需的操作并返回，以便能够执行下一个任务。为了达到此要求，应该使用状态机方法将需要等待某些外部输入或需要执行长操作的任务分解成多个子任务。有关协同多任务处理和状态机编程的进一步讨论不在本文档的范围之内。如需更多详细信息，请参阅软件工程文献。

所编写的 Microchip 协议栈在不做任何更改的情况下就能支持 MPLAB C18 和 Hi-Tech PICC-18 C 编译器。所有源文件自动检测当前正在使用的编译器并相应地调整其代码。Microchip 协议栈以带有 C18 和 PICC-18 特定扩展的标准 ANSI C 编写。如果需要，可以修改这些源文件以支持您选择的编译器。

为了简化文件管理和应用程序开发，所有源文件均位于 `Source` 目录下的子目录中。更多信息，请参见 **“源文件构成”**。

当使用 Microchip 协议栈开发应用程序时，推荐您使用演示应用程序目录结构作为参考，并创建针对您自己的应用程序的子目录。

下面是开发基于 Microchip 协议栈的应用程序的典型步骤。注意这些步骤假定您使用 MPLAB IDE 并且熟悉 MPLAB IDE 界面。

1. 按照前面在 **“安装源文件”** 部分中的描述，安装 Microchip 协议栈源文件。
2. 在 `MpZBee\Source` 目录中创建针对您的应用程序的目录。
3. 根据这是协调器应用程序还是 RFD 应用程序，将 `Source\DemoCoordApp` 或 `Source\DemoRFDApp` 目录中的 `zigbee.def` 文件复制到针对您的应用程序的目录中。
4. 根据您的应用需求修改 `zigbee.def`。更多信息，请参见 **“协议栈配置”**。
5. 使用 MPLAB IDE 来创建您的应用程序项目，并根据您的 ZigBee 节点功能添加协议栈源文件。更多信息，请参见 **“协议栈源文件”**。
6. 如果正在使用 MPLAB C18 编译器，则添加针对您所使用的器件的链接描述文件。
7. 使用 MPLAB Build Option（编译选项）对话框来设置两个附加的包含搜索路径：“`..\Stack`”和“`..\<YourAppDir>`”，其中 `<YourAppDir>` 是包含针对您的应用程序的 `zigbee.def` 文件的目录的名称。
8. 添加针对您的应用程序的源文件。
9. 现在准备开始编译您的应用程序项目。

协议栈源文件

根据应用程序的类型，您将需要在项目中包含一组特定的源文件。表 6 列出了编译典型 ZigBee RFD 应用程序所需的所有源文件，而表 7 列出了编译典型 ZigBee 协调器应用程序所需的所有源文件。

表 6： 典型的 RFD 应用程序文件

源文件	用途
您的应用程序文件	必须包括 main() 入口点
zigbee.def	针对应用程序的 Microchip 协议栈选项
Console.c	RS-232 终端程序，如果定义了 ENABLE_DEBUG 或您的应用程序使用控制台程序则需要此程序
MSPI.c	用于访问 RF 收发器的主 SPI™ 接口程序
Tick.c	节拍管理器，用于跟踪超时和重试条件
zAPL.c	ZigBee™ 应用层
zAPS.c	ZigBee 应用支持子层
ZDO.c	ZigBee 设备对象，如果需要 ZigBee 远程管理则需要 ZigBee 设备对象
zMAC.c	IEEE 802.15.4 MAC 层
zNVM.c	非易失性存储器存储程序，可用您自己的非易失性存储特定文件来替换
zNWK.c	ZigBee 网络层
ZPHY???.c	针对 RF 收发器的程序，对于 Chipcon CC2420 收发器是 zPHYCC2420.c；对于 ZMD 44101 收发器是 zPHYZMD44101.c
zProfile.c	ZigBee 配置程序，如果需要支持标准配置 （在版本 1.00.00 中未完全实现）则需要该程序
18f????.lkr	针对您所选择的器件的链接描述文件，如果使用 C18 则需要此文件

表 7： 典型的协调器应用程序文件

源文件	用途
您的应用程序文件	必须包括 main() 入口点
zigbee.def	针对应用程序的 Microchip 协议栈选项
Console.c	RS-232 终端程序，如果定义了 ENABLE_DEBUG 或您的应用程序使用控制台程序则需要此程序
NeighborTable.c	实现邻接表和绑定表
SRAlloc.c	实现间接发送缓冲器的动态存储器管理器
Tick.c	节拍管理器
zAPL.c	ZigBee™ 应用层
zAPS.c	ZigBee 应用支持子层
ZDO.c	ZigBee 设备对象，如果需要 ZigBee 远程管理则需要 ZigBee 设备对象
zMAC.c	IEEE 802.15.4 MAC 层
zNVM.c	非易失性存储器存储程序，可用您自己的非易失性存储特定文件来替换
zNWK.c	ZigBee 网络层
ZPHY???.c	针对 RF 收发器的程序，对于 Chipcon CC2420 收发器是 zPHYCC2420.c；对于 ZMD 44101 收发器是 zPHYZMD44101.c
zProfile.c	ZigBee 配置程序，如果需要支持标准配置 （在版本 1.00.00 中未完全实现）则需要该程序
18f????.lkr	针对您所选择的器件的链接描述文件，如果使用 C18 则需要此文件

协议栈配置

此 Microchip 协议栈使用多种编译时间选项来使能 / 禁止多个内核逻辑和 RAM 变量。具体的内核逻辑与 RAM 变量的组合由 ZigBee 应用程序的类型决定。为了简化编译时间的配置，所有编译时间选项均被保存在 zigbee.def 文件中。作为应用程序开发的一部分，您必须修改 zigbee.def。

接下来的章节将定义所有的编译时间选项。应该查看 zigbee.def 文件以获取编译时间选项的最新列表。

选项名称 CLOCK_FREQ

用途 定义处理器时钟频率。 Tick.c 和 Debug.c 分别使用此值来计算 TMR0 和 SPBRG 的值。需要的话，也可以在您的应用程序中使用此值。

前提 无

有效值 必须符合 PIC 频率规范。

示例 下面的行将 CLOCK_FREQ 定义为 4 MHz：

```
#define CLOCK_FREQ    4000000
```

选项名称 TICK_PRESCALE_VALUE

用途 Timer0 预分频值。 Tick.c 使用此值来计算 TMR0 载入值。

前提 无

有效值 可能的 TMR0 预分频值请参阅 PIC 器件的数据手册。

示例 以下行设置 TMR0 预分频值为 2：

```
#define TICK_PRESCALE_VALUE 2
```

注 无

选项名称 TICKS_PER_SECOND

用途 一秒钟内的节拍数。由 Tick.c 文件使用。

前提 无

有效值 1-255，必须根据 TICK_PRESCALE_VALUE 对此值进行调整。

示例 以下行将每秒的节拍数设置为 50：

```
#define TICKS_PER_SECOND    50
```

注 无

选项名称 BAUD_RATE

用途 定义 USART 波特率值。 Console.c 文件将使用此值。可以根据应用程序的要求更改此值。

前提 无

有效值 无

示例 以下行将波特率定义为 19200 bps ：

```
#define BAUD_RATE      (19200)
```

注 必须确保当前的波特率选择对于当前的 CLOCK_FREQ 选择是可能的。

选项名称	ENABLE_DEBUG
用途	它使能调试模式。如果在 zigbee.def 文件中定义它，就能为所有源文件使能调试模式。此外，可以通过在特定文件的开头定义 ENABLE_DEBUG 来有选择地使能各个调试模式。
前提	无
有效值	无
示例	以下行使能调试模式： #define ENABLE_DEBUG
注	当定义 ENABLE_DEBUG 时，应用程序代码将会增加。 ENABLE_DEBUG 模式定义很多 ROM 字符串消息。

选项名称	USE_CC24240
用途	用于指示正在使用 Chipcon CC2420 收发器。
前提	不能定义 USE_ZMD44101。
有效值	无
示例	以下行定义正在使用 CC2420： #define USE_CC2420
注	只能定义 USE_CC2420 或 USE_ZMD44101 中的一种。协议栈均设计为同一时刻只使用一种类型的 RF 收发器。

选项名称	USE_ZMD44101
用途	用于指示正在使用 ZMD 44101 收发器 （当前版本不支持）。
前提	不能定义 USE_CC2420。
有效值	无
示例	以下行定义正在使用 ZMD 44101： #define USE_ZMD44101
注	只能定义 USE_CC2420 或 USE_ZMD44101 中的一种。协议栈均设计为同一时刻只使用一种类型的 RF 收发器。

选项名称	I_AM_COORDINATOR
用途	表示此节点是一个协调器。
前提	不能定义 I_AM_ROUTER 和 I_AM_END_DEVICE。
有效值	无
示例	以下行将当前节点设置为协调器： #define I_AM_COORDINATOR
注	一旦定义了 I_AM_COORDINATOR，就不能定义 I_AM_ROUTER 和 I_AM_END_DEVICE。

选项名称	I_AM_ROUTER
用途	指示此节点是一个路由器 （在当前版本中不使用）。
前提	不能定义 I_AM_COORDINATOR 和 I_AM_END_DEVICE。
有效值	无
示例	以下行将当前节点设置为路由器： #define I_AM_ROUTER
注	一旦定义了 I_AM_ROUTER，就不能定义 I_AM_COORDINATOR 和 I_AM_END_DEVICE。当前版本不支持路由器功能。

选项名称	I_AM_END_DEVICE
用途	表示这是一个终端设备，可以是 RFD 或 FFD （在当前版本中，终端设备必须总是 RFD）。
前提	不能定义 I_AM_COORDINATOR 和 I_AM_ROUTER。
有效值	无
示例	以下行将当前节点设置为终端设备： #define I_AM_END_DEVICE
注	一旦定义了 I_AM_END_DEVICE，就不能定义 I_AM_COORDINATOR 和 I_AM_ROUTER。

选项名称	MY_FREQ_BAND_IS_868_MHZ
用途	将工作频带定义为 868 MHz （当前版本中不支持）。
前提	必须定义 USE_ZMD44101 （在以后的版本中会有更多选项）。
有效值	无
示例	以下行设置 868 MHz 频带： #define MY_FREQ_BAND_IS_868_MHZ
注	一旦定义了 MY_FREQ_BAND_IS_868_MHZ，就不能定义 MY_FREQ_BAND_IS_900_MHZ 和 MY_FREQ_BAND_IS_2400_MHZ。当前版本不支持 868/915 MHz 工作。

选项名称	MY_FREQ_BAND_IS_900_MHZ
用途	将工作频带定义为 915 MHz （当前版本中不支持）。
前提	必须定义 USE_ZMD44101 （在以后的版本中会有更多选项）。
有效值	无
示例	以下行设置 915 MHz 频带： #define MY_FREQ_BAND_IS_915_MHZ
注	一旦定义了 MY_FREQ_BAND_IS_915_MHZ，就不能定义 MY_FREQ_BAND_IS_868_MHZ 和 MY_FREQ_BAND_IS_2400_MHZ。当前版本不支持 868/915 MHz 工作。

选项名称	MY_FREQ_BAND_IS_2400_MHZ
用途	将工作频带定义为 2.4 GHz。
前提	必须定义 USE_CC2420 （在以后的版本中会有更多选项）。
有效值	无
示例	以下行设置 2.4 GHz 频带： #define MY_FREQ_BAND_IS_2400_MHZ
注	一旦定义了 MY_FREQ_BAND_IS_2400_MHZ，就不能定义 MY_FREQ_BAND_IS_868_MHZ 和 MY_FREQ_BAND_IS_900_MHZ。

选项名称	I_AM_ALT_PAN_COORD
用途	表示可以将当前 FFD 节点作为备用 PAN 协调器 （当前版本中不支持）。
前提	无
有效值	无
示例	以下行将当前节点定义为备用 PAN 协调器： #define I_AM_ALT_PAN_COORD
注	当前版本不支持 FFD 和备用 PAN 协调器功能。

选项名称 I_AM_MAINS_POWERED

用途 表示当前节点由交流电源供电。通常，协调器和路由器的电源都是交流电源 （在当前版本中不使用）。

前提 不能定义 I_AM_RECHARGEABLE_BATTERY_POWERED 和 I_AM_DISPOSABLE_BATTERY_POWERED。

有效值 无

示例 以下行表示这是一个交流电源供电的设备：

```
#define I_AM_MAINS_POWERED
```

注 一旦定义了 I_AM_MAINS_POWERED，就不能定义 I_AM_RECHARGEABLE_BATTERY_POWERED 和 I_AM_DISPOSABLE_BATTERY_POWERED。当前版本不使用此信息。将使用此信息来创建标准节点的 ZigBee 配置文件。

选项名称 I_AM_RECHARGEABLE_BATTERY_POWERED

用途 表示当前节点使用电池作为电源 （当前版本中不可用）。

前提 不能定义 I_AM_MAINS_POWERED 和 I_AM_DISPOSABLE_BATTERY_POWERED。

有效值 无

示例 以下行表示这是使用电池供电的设备：

```
#define I_AM_RECGARGEABLE_BATTERY_OPERATED
```

注 一旦定义了 I_AM_RECHARGEABLE_BATTERY_POWERED，就不能定义 I_AM_MAINS_POWERED 和 I_AM_DISPOSABLE_BATTERY_POWERED。当前版本不使用此信息。将使用此信息来创建标准节点的 ZigBee 配置文件。

选项名称 I_AM_DISPOSABLE_BATTERY_POWERED

用途 表示当前节点使用一次性电池作为电源 （当前版本中不可用）。

前提 不能定义 I_AM_MAINS_POWERED 和 I_AM_RECHARGEABLE_BATTERY_POWERED。

有效值 无

示例 以下行表示这是使用一次性电池供电的设备：

```
#define I_AM_DISPOSABLE_BATTERY_POWERED
```

注 一旦定义了 I_AM_DISPOSABLE_BATTERY_POWERED，就不能定义 I_AM_RECHARGEABLE_BATTERY_POWERED 和 I_AM_DISPOSABLE_BATTERY_POWERED。当前版本不使用此信息。将使用此信息来创建标准节点的 ZigBee 配置文件。

选项名称 I_AM_SECURITY_CAPABLE

用途 表示此节点使用加密 / 解密来发送和接收数据包 （当前版本中不支持）。

前提 无

有效值 无

示例 以下行表示此节点具有安全功能：

```
#define I_AM_SECURITY_CAPABLE
```

注 当前版本不支持安全功能。

选项名称 MY_RX_IS_ALWAYS_ON_OR_SYNCED_WITH_BEACON

用途 表示此节点总是将接收器保持在开启状态或者周期性地监听信标 （当前版本中不支持）。

前提 不能定义 MY_RX_IS_PERIODICALLY_ON 和 MY_RX_IS_ON_WHEN_STIMULATED。

有效值 无

示例 #define MY_RX_IS_ALWAYS_ON_OR_SYNCED_WITH_BEACON

注 当前版本不使用或支持此信息。

选项名称 MY_RX_IS_PERIODICALLY_ON

用途 表示此节点周期性地开启接收器 （当前版本中不使用）。

前提 不能定义 MY_RX_IS_ALWAYS_ON_OR_SYNCED_WITH_BEACON 和 MY_RX_IS_ON_WHEN_STIMULATED。

有效值 无

示例 #define MY_RX_IS_PERIODICALLY_ON

注 当前版本不使用此信息。

选项名称 MY_RX_IS_ON_WHEN_STIMULATED

用途 表示此节点只有在受到激励时才开启接收器 （当前版本中不支持）。

前提 不能定义 MY_RX_IS_ALWAYS_ON_OR_SYNCED_WITH_BEACON 和 MY_RX_IS_PERIODICALLY_ON。

有效值 无

示例 #define MY_RX_IS_ON_WHEN_STIMULATED

注 当前版本不使用此信息。

选项名称 MAC_LONG_ADDR_BYTEn

用途 为此节点定义默认的 64 位 MAC 地址。总共可定义 8 个 MAC 地址，每个字节一个地址。主应用程序会使用此值来初始化 MAC 地址或根据需要在运行时更改它。

前提 无

有效值 0-255

示例 将默认的 MAC 地址设置为 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)
```

注 此协议栈源不自动设置此地址。主应用程序必须使用此值来初始化 MAC 层提供的 MAC 地址变量 macLongAddr。

选项名称 MAX_EP_COUNT
用途 定义此设备所支持的端点的最大数量。
前提 无
有效值 1-255 （最小值必须为 1。最大值根据可用的 RAM 容量来决定；每个端点需要占用 RAM 的 5 个字节。）
示例
```
#define MAX_EP_COUNT (4)
```
注 必须至少有 1 个端点支持标准的 ZDO 端点。每增加一个端点数就要增加 5 个字节的 RAM 占用。在给定的设备中，最大限制为 255 个端点；但是实际数量将受到可用的 RAM 容量的限制。

选项名称 MAC_USE_RF_TEST_CODE
用途 使能收发器的特定测试功能。
前提 无
有效值 无
示例
```
#define MAC_USE_RF_TEST_CODE
```
注 在当前版本的 Chipcon RF 收发器中，有两个收发器测试功能，一个用于发送随机调制的信号而另一个用于发送未调制的信号。这两个功能对于测试 RF 电路的性能很有用处。

选项名称 MAC_USE_SHORT_ADDR
用途 仅应用于终端设备 （即定义了 I_AM_END_DEVICE 的设备），终端设备在与网络相关联的时候使用此选项来请求新的短地址。
前提 无
有效值 无
示例
```
#define MAC_USE_SHORT_ADDR
```
注 基于当前版本的 ZigBee 终端设备将总是请求来自协调器的短地址。

选项名称 MAC_CHANNEL_ENERGY_THRESHOLD
用途 定义阈值，超过该阈值的信道将被使用 （当前版本中未使用）。
前提 无
有效值 0-255 （具体的值由 RF 收发器决定。）
示例
```
#define MAC_CHANNEL_ENERGY_THRESHOLD    (0x20)
```
注 无

选项名称 MAC_MAX_FRAME_RETRIES
用途 设置在没有接收到应答的情况下的最大帧重试次数。
前提 无
有效值 1-5
示例
```
#define MAC_MAX_FRAME_RETRIES    (3)
```
注 无

选项名称 MAC_ACK_WAIT_DURATION
用途 设置此节点等待来自另一个节点的应答的最大时间。
前提 无
有效值 1-4,294,967,296 tick
示例 将应答等待时间设置为半秒：
```
#define MAC_ACK_WAIT_DURATION (TICK_SECOND/2)
```
注 无

选项名称 MAC_RESPONSE_WAIT_TIME
用途 设置此节点等待来自另一个节点的响应的最大时间。
前提 无
有效值 1-255 节拍
示例
```
#define MAC_RESPONSE_WAIT_TIME    (TICK_SECOND)
```
注 在星型网络中，由于查询请求，终端设备需要等待这么长的时间才能收到响应。

选项名称 MAC_ED_SCAN_PERIOD
用途 设置能量检测周期。在此周期中， RF 接收器保持开启状态以测量 RF 能量。
前提 无
有效值 1-4,294,967,296 节拍
示例 将能量检测扫描周期设置为 1/4 秒：
```
#define MAC_ED_SCAN_PERIOD    (TICK_SECOND/4)
```
注 无

选项名称 MAC_ACTIVE_SCAN_PERIOD
用途 设置有效扫描周期。在有效扫描期间，节点请求来自附近的协调器的信标并期待协调器在这个周期内发出响应。
前提 无
有效值 1-4,294,967,296 节拍
示例 将有效扫描周期设置为 1/2 秒：
```
#define MAC_ACTIVE_SCAN_PERIOD (TICK_SECOND/2)
```
注 无

选项名称 MAC_MAX_DATA_REQ_PERIOD
用途 仅在定义了 I_AM_COORDINATOR 时使用。
设置一个周期，在此周期之内，终端设备必须向此协调器请求它们的数据帧。也可以把这个周期看作是网络中的每个节点必须查询协调器的一段时间。
前提 必须定义 I_AM_COORDINATOR。
有效值 1-4,294,967,296 节拍
示例 将最大数据请求周期设置为 10 秒：
```
#define MAC_MAX_DATA_REQ_PERIOD (TICK_SECOND*10)
```
注 无

选项名称	MAX_HEAP_SIZE
用途	定义间接发送帧缓冲器的最大容量。基于 Microchip 协议栈的协调器使用动态存储管理器在间接发送帧缓冲器中分配各个数据帧。
前提	无
有效值	大于 128，最大值由可用的 RAM 容量决定。
示例	将堆大小设置为 256 字节： `#define MAX_HEAP_SIZE (256)`
注	仅协调器节点可使用此选项 （即定义了 I_AM_COORDINATOR）。 确切的堆大小由 MAX_DATA_REQ_PERIOD （帧的平均大小和网络中节点的总数）决定。网络的 MAX_DATA_REQ_PERIOD 越长，即节点越多并且帧越长，就应该使用越大的堆。通常，堆必须足够大以保存典型数量的帧直到它们被预期的接收者读取为止。如果正在使用 C18，就必须更改链接描述文件并使用程序来将堆定义为大于 256 字节。

选项名称	MAX_NEIGHBORS
用途	定义此协调器所支持的节点的最大数量。
前提	必须定义 I_AM_COORDINATOR。
有效值	大于 2，最大值由可用的程序存储器容量决定。每增加一个邻接节点就需要消耗 12 字节的程序存储空间。
示例	设置此网络中支持的节点的最大数量： `#define MAX_NEIGHBORS (10)`
注	仅协调器节点可使用此选项 （即定义了 I_AM_COORDINATOR）。 根据节点的总数以及查询协调器的频率，可能需要增加协调器的时钟频率以满足增强处理能力的需要。

选项名称	MAX_BINDINGS
用途	定义此协调器所支持的绑定请求的最大数量 （绑定表大小）。
前提	必须定义 I_AM_COORDINATOR。
有效值	2-255 最大值由可用的程序存储器容量决定。每增加一个绑定项就需要消耗 12 字节的程序存储空间。
示例	将最大绑定表大小设置为 10： `#define MAX_BINDINGS (10)`
注	仅协调器节点可使用此选项 （即定义了 I_AM_COORDINATOR）。 每个节点可能有多个绑定项。确切的绑定表大小将根据网络中节点的数量和每个节点的绑定请求数量来决定。

集成应用程序

在根据应用程序的需要修改了编译时间配置之后，下一步将修改主应用程序，初始化并运行协议栈状态机。如果正在开发协调器节点，应该使用 DemoZCoordApp.c 文件作为参考。如果是 RFD 节点，则使用 DemoZRFDApp.c 文件。

回调函数

除了标准的 API 调用之外，应用程序还必须实现一定量的回调函数。回调函数在主应用程序的源文件中。协议栈在做出任何针对应用程序的决定之前调用这些回调函数来通知应用程序或与应用程序交换信息。所有回调函数的名称都以“App”作为前缀，以表示这是一个回调函数。下面的章节将更详细地讨论每个回调函数。

AppOkayToUseChannel

此回调函数询问主应用程序是否可以使用给定的信道。

语法

BOOL AppOkayToUseChannel(BYTE channel)

参数

channel [in]

需要选择的信道编号。此值由频带的频率决定：

频率为 2.4 GHz，信道编号为 11-26

频率为 915 MHz，信道编号为 1-10

频率为 868 MHz，信道编号为 0

返回值

如果应用程序想要使用给定的信道，返回 TRUE

否则返回 FALSE

前提

无

副作用

无

注

即使应用程序使用其频带中的所有信道，也必须实现此回调函数。当应用程序返回 FALSE 时，协议栈将会自动调用此函数以询问是否使用下一个信道，直到应用程序返回 TRUE 为止。

示例

```
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

此回调函数通知应用程序接收到了新的有效数据帧。这仅仅是一个通知（有可能已经处理了或未处理实际的帧）。应用程序可使用此通知点亮 LED 或其他可视的指示器。

语法

```
void AppMACFrameReceived(void)
```

参数

无

返回值

无

前提

无

副作用

无

注

无

示例

```
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

此回调函数通知应用程序数据帧已发送。应用程序可使用此通知点亮 LED 或其他可视的指示器。

语法

```
void AppMACFrameTransmitted(void)
```

参数

无

返回值

无

前提

无

副作用

无

注

无

示例

```
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

此回调函数通知应用程序远程节点未在 MAC_ACK_WAIT_DURATION 内发送应答。应用程序可使用此通知点亮 LED 或其他可视的指示器。

语法

```
void AppMACFrameTimeOutOccurred(void)
```

参数

无

返回值

无

前提

无

副作用

无

注

无

示例

```
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

这是一个主应用程序回调函数。当终端设备尝试连入可用网络时，该协议栈会在其射频范围内找到一个协调器时调用此函数。在一个射频范围内，不同的信道上可以有多个协调器；在这种情况下，请求连入一个特定的协调器之前，协议栈会重复地查找协调器并调用这一函数以通过应用程序的批准。应用程序可能会决定在选择要与之关联的特定协调器之前搜集所有临近的协调器。

仅当已定义 I_AM_END_DEVICE 时该回调函数才可用。

语法

```
BOOL AppOkayToAssociate(void)
```

参数

无

返回值

应用程序想要与当前协调器相关联则返回 TRUE。应用程序可通过访问在 MAC.h 文件中定义的 PANDesc 变量结构来查看当前协调器的信息。

其他情况返回 FALSE。在这种情况下，协议栈会继续通过自动切换到下一个可用信道来寻找新的协调器。

前提

无

副作用

无

注

无

示例

```
// 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

此回调函数询问主应用程序是否要将给定的节点连入其网络。主应用程序可使用它的专用选择标准来允许新的节点连入其网络。

仅当定义了 I_AM_COORDINATOR 时才可使用此回调函数。

语法

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

参数

longAddr [in]

指向想要连入此网络的节点的 64 位 MAC 地址的指针。

返回值

如果应用程序想要允许将给定的节点连入其网络，则返回 TRUE

否则返回 FALSE

前提

无

副作用

无

注

无

示例

```
// 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

此回调函数通知主应用程序新节点刚连入了其网络。

仅当定义了 I_AM_COORDINATOR 时才可使用此回调函数。

语法

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

参数

nodeAddr [in]

指向刚连入此网络的节点的 64 位 MAC 地址的指针。

bIsRejoined [in]

表示这是一个老节点还是新节点。

返回值

无

前提

无

副作用

无

注

当一个新的节点连入网络时，邻接表中会创建一个新的项。但是，在调用 APLCommitTableChanges 函数之前，此项并没有完全保存。应用程序不会总是想要允许新的节点连入其网络。协调器可能会被置于一个特殊的模式以允许新的节点连入其网络。为了提供这种灵活性， Microchip 协议栈要求调用 APLCommitTableChanges() 来发出实际的关联请求。何时调用此函数取决于您的应用程序逻辑。

示例

```
// 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

此回调函数通知主应用程序一个老节点刚刚离开网络。

仅当定义了 I_AM_COORDINATOR 时才可使用此回调函数。

语法

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

参数

nodeAddr [in]

指向刚离开此网络的节点的 64 位 MAC 地址的指针。

返回值

无

前提

无

副作用

无

注

当一个新的节点离开网络时，必须删除相应的关联和绑定表项。一旦删除了关联项，就必须提交更改以便永久保存更改。

示例

```
// 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 协议概述

ZigBee 是为低速率控制网络设计的标准无线网络协议。ZigBee 协议的一些应用包括建筑自动化网络、建筑安防系统、工业控制网络、远程抄表以及 PC 外设。下面的部分将提供与理解 Microchip 协议栈功能相关的 ZigBee 协议概述。有兴趣的读者请在 ZigBee 网站（www.zigbee.org）查找更多信息。

IEEE 802.15.4

ZigBee 协议使用 IEEE 802.15.4 规范作为介质访问层（MAC）和物理层（PHY）。IEEE 802.15.4 总共定义了 3 个工作频带：2.4 GHz、915 MHz 和 868 MHz。每个频带提供固定数量的信道。例如，2.4 GHz 频带总共提供 16 个信道（信道 11-26）、915 MHz 频带提供 10 个信道（信道 1-10）而 868 MHz 频带提供 1 个信道（信道 0）。

协议的比特率由所选择的工作频率决定。2.4 GHz 频带提供的数据速率为 250 kbps，915 MHz 频带提供的数据速率为 40 kbps 而 868 MHz 频带提供的数据速率为 20 kbps。由于数据包开销和处理延迟，实际的数据吞吐量会小于规定的比特率。

IEEE 802.15.4 MAC 数据包的最大长度为 127 字节。每个数据包都由头字节和 16 位 CRC 值组成。

16 位 CRC 值验证帧的完整性。此外，IEEE 802.15.4 还可以选择使用应答数据传输机制。使用这种方法，所有特殊 ACK 标志位置 1 的帧均会被它们的接收器应答。这就可以确定帧实际上已经被传递了。如果发送帧的时候置位了 ACK 标志位而且在一定的超时期限内没有收到应答，发送器将重复进行固定次数的发送，如仍无应答就宣布发生错误。注意接收到应答仅仅表示帧被MAC 层正确接收，而不表示帧被正确处理，这是非常重要的。接收节点的 MAC 层可能正确地接收并应答了一个帧，但是由于缺乏处理资源，该帧可能被上层丢弃。因此，很多上层和应用程序要求其他的应答响应。

网络配置

ZigBee 无线网络可采用多种类型的配置。星型网络配置由一个协调器节点（主设备）和一个或多个终端设备（从设备）组成。协调器是实现了一组很多 ZigBee 服务的一种特殊的全功能设备（Full Function Device，FFD）。终端设备可能是 FFD 或简化功能设备（RFD）。RFD 是最小而且最简单的 ZigBee 节点。它只实现了一组最少的 ZigBee 服务。在星型网络中，所有的终端设备都只与协调器通信。如果某个终端设备需要传输数据到另一个终端设备，它会把数据发送给协调器，然后协调器依次将数据转发到目标接收器终端设备。

除了星型网络之外，ZigBee 还可以采用点对点网络、群集或网状（mesh）网络配置。由于群集和网状网络具有在多个网络之间路由数据包的功能，因而被称为多跳网络，而星型网络则被称为单跳网络。

和任何网络一样，ZigBee 网络也是多点接入网络，这意味着网络中的所有节点对通信介质的访问是同等的。有两种类型的多点接入机制。在没有使能信标的网络中，只要信道是空闲的，在任何时候都允许所有节点发送。在使能了信标的网络中，仅允许节点在预定义的时隙内进行发送。协调器会定期以一个标识为信标帧的超级帧开始发送，并且希望网络中的所有节点与此帧同步。在这个超级帧中为每个节点分配了一个特定的时隙，在该时隙内允许节点发送和接收数据。超级帧可能还含有一个公共时隙，在此时隙内所有节点竞争接入信道。

Microchip 协议栈的当前版本仅支持无信标的星型网络配置。

网络关联

ZigBee 网络可以是 ad-hoc 网络，即可以根据需要组建或不组建新的网络。在星型网络配置中，终端设备在可以执行任何数据传输之前将总是搜索网络。新的网络首先由协调器建立。启动时，协调器会搜索附近的其他协调器，如果没有找到协调器，它就会建立一个自己的网络并选择一个惟一的 16 位 PAN ID。一旦新网络建立，就会允许一个或多个终端设备与此网络相关联。具体是允许还是不允许新关联由协调器决定。

一旦组建了网络，就可能由于物理更改而发生多个网络重叠和PAN ID冲突。在这种情况下，协调器会启动PAN ID 冲突解决过程并且会有一个协调器将更改其 PAN ID 和 / 或信道。受到影响的协调器会指示它所有的终端设备进行必要的更改。 Microchip 协议栈的当前版本不支持 PAN ID 冲突解决。

根据系统需求，协调器会在非易失性存储器中存储所有网络关联，称为邻接表。为了连接到网络，终端设备可能执行孤立通知过程来查找先前与之关联的网络或者执行关联过程来加入一个新网络。在执行孤立通知过程的情况下，协调器将通过查找其邻接表来识别先前与之关联的终端设备。

一旦关联到网络，终端设备就可选择通过执行解除关联过程与该网络解除关联。如果需要的话，协调器本身也会启动解除关联过程来强制节点离开网络。

Microchip 协议栈的当前版本支持新的关联和孤立通知过程。它仅支持由终端设备启动的离开网络过程。

端点、接口、群集、属性和配置文件

典型的 ZigBee 节点可支持多种特性和功能。例如，I/O 节点可能有多种数字和模拟输入 / 输出。一些数字输入可能被一个远程控制器节点用到，而其他数字输入可能被另一个远程控制器节点使用。这种分配将创建一个真正的分布式控制网络。为了便于在 I/O 节点和两个控制器节点之间进行数据传输，所有节点中的应用程序必须保存多个数据链路。为了减少成本，ZigBee 节点仅使用一个无线信道来和多个端点 / 接口来创建多条虚拟链路或信道。

一个 ZigBee 节点支持 31 个端点 （编号为 0-31）和 8 个接口 （编号为 0-7）。端点 0 被保留用于设备配置而端点 31 被保留仅用于广播。剩下的总共 30 个端点用于应用。每个端点总共有 8 个接口。因此，实际上，应用在一个物理信道中最多可能有 240 条虚拟信道。

一个典型的 ZigBee 节点也将有很多属性。例如，I/O 节点包含称为数字输入 1、数字输入 2、模拟输入 1 等的属性。每个属性都有自己的值。例如，数字输入 1 属性可能有值 1 或 0。属性的集合被称为群集。在整个网络中，每个群集都被分配了一个惟一的群集 ID。每个群集最多有 65,536 个属性。

ZigBee 协议还定义了一个称为*配置文件*的术语。配置文件就是指对分布式应用的描述。它根据应用必须处理的数据包和必须执行的操作来描述分布式应用。使用描述符对配置文件进行描述，描述符仅仅是各种值的复杂结构。此配置文件使 ZigBee 设备可以互操作。 ZigBee 联盟已经定义了很多标准的配置文件，比如远程控制开关配置文件和光传感器配置文件等。任何遵循某一标准配置文件的节点都可以与其他实现相同配置文件的节点进行互操作。Microchip 协议栈的当前版本不提供任何标准的配置文件功能。如果需要的话，可以编写一个实现所需要的配置文件的协同任务函数。也可以创建自己的自定义配置文件 （或分布式应用程序）仅与专有节点配合工作。和此应用笔记一起提供的演示应用程序实现了远程控制 LED 的自定义分布式应用程序，其中一个节点的 LED 由另一个节点上的开关控制。每个配置文件可以定义最多 256 个群集，而且和我们在前面所看到的一样，每个群集可以最多有 65,536 个属性。此灵活性允许节点有大量的属性 （或 I/O 点）。

端点绑定

前面提到过，星型网络中的终端设备总是只与协调器通信。协调器负责将端点发送的数据包从一个节点转发到接收终端设备的相应端点。您可能会猜到，当建立一个新的网络时，必须告知协调器如何创建源端点和目标端点之间的链路。ZigBee协议定义了一个称为端点绑定的特殊过程。作为绑定过程的一部分，一个远程网络或一个类似于设备管理器的节点会请求协调器修改其绑定表。协调器节点维护一个基本上包含两个或多个端点之间的逻辑链路的绑定表。每个链路根据其源端点和群集 ID 来惟一定义。

例如，如果需要将来自我们的 I/O 节点的数字输入 1 的数据发送到控制器节点的控制信道 1，我们就必须请求协调器创建一个绑定表项，此绑定表项将 I/O 节点的数字输入 1 端点作为源端点，将控制器节点的控制信道 1 作为目标端点。一旦创建了绑定表项，任何时候 I/O 节点从数字输入 1 端点发送数据时，协调器节点就会查找它的绑定表，并将数据包转发到控制器节点的控制信道 1 端点。数字输入 1 和控制信道 1 将共享同一个群集 ID。根据绑定表的创建方法，可能将数据从一个端点组播到多个节点上的多个端点。Microchip 协议栈的当前版本不支持这种组播绑定表项。

ZigBee 协议定义了称为 ZigBee 设备对象（ZigBee Device Object，ZDO）的特殊软件对象，它在其他服务中提供绑定服务。只有在协调器上运行的 ZDO 才会提供绑定服务。远程网络 / 设备管理器将直接将特殊绑定请求发送给 ZDO（端点 0）以创建或修改绑定表项。根据 ZigBee 规范，运行特殊 ZigBee 节点软件的 PC 或其他高端控制器可以作为网络管理器。

如果不想创建或使用特殊网络管理器节点，可以编写自定义的绑定服务来简化绑定过程。和此应用笔记一起提供的演示应用程序实现了简单的自定义绑定方法。它使能每个节点将各自的对协调器节点的绑定请求发送给端点 0 并定义各自的自定义群集 ID。欲知更多信息，请参阅 ZDO.c 文件中的 CUSTOM_DEMO_BIND 群集 ID。根据此自定义的绑定过程，终端设备必须处于配置模式才能发送绑定请求。在正常执行模式下，协调器可以接收和发出自己的绑定请求。

当检测到某个按开关的过程时，终端设备将使用 CUSTOM_DEMO_BIND 作为群集 ID 发出特殊的二进制数据结构（欲知有关配置模式和绑定过程的更多信息，请参阅 **“配置演示应用程序”**）。它直接将绑定请求数据包发送到协调器的端点 0。演示协调器中的 ZDO 接收群集 ID 为 CUSTOM_DEMO_BIND 的数据包并将此过程委派给 ProcessCustomBind 函数。此函数实际上是在 DemoCoordApp.c 文件中实现的。可以简单地按照 ProcessCustomBind 函数的执行逻辑来充分理解自定义绑定的概念。你可以直接使用此自定义绑定逻辑或将其作为编写自己的绑定逻辑时的参考。

数据传输机制

传输数据到终端设备和从终端设备传输数据的确切机制随网络类型的不同而有所不同。在无信标的星型网络中，当终端设备想要发送数据帧时，它只需等待信道变为空闲。在检测到空闲信道条件时，它将帧发送到协调器。如果协调器想要将此数据发送到终端设备，它会将数据帧保存在其发送缓冲器中，直到目标终端设备明确地来查询该数据为止。此方法确保终端设备的接收器是被开启的，而且可从协调器接收数据。

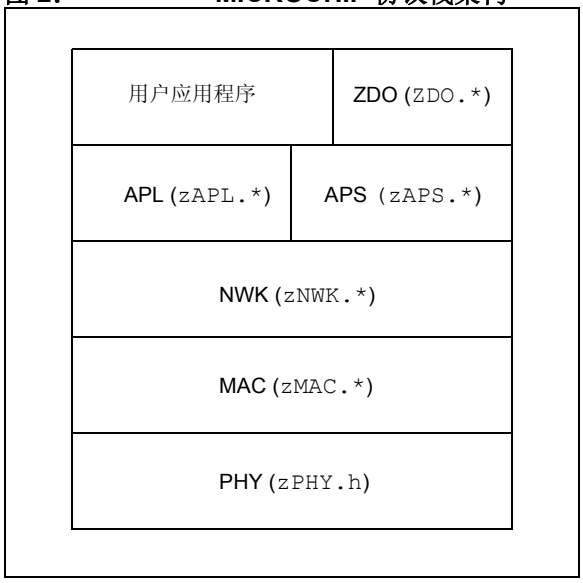
在点对点网络中，每个节点必须一直保持它们的接收器为开启状态或者同意在一个时间段内开启它们的接收器。这将允许节点发送数据帧并确保数据帧会被其它节点接收。

终端设备必须查询协调器以获取其数据，而不是保持接收器开启，从而允许终端设备降低其功耗要求。根据应用的要求，在绝大部分时间内终端设备都处于休眠状态，而仅定期地唤醒设备来发送或接收数据。此方法的一个缺点就是协调器必须将所有数据帧保存在内部缓冲器中，直到目标终端设备唤醒并查询数据。如果网络包含很多休眠时间很长的终端设备，协调器就必须将数据帧保存很长时间。根据节点的数量和交换数据帧的速率，这将大幅增加协调器对 RAM 的需求。协调器可以根据终端设备的设备描述符有选择地决定将一个特定的帧保持一段长时间或短时间。ZigBee 协议要求所有终端设备都保存描述它们的特性和功能的不同方面的各种描述符。 Microchip 协议栈的当前版本不支持描述符。

协议栈架构

Microchip 协议栈是采用 C 语言编写的，可用 MPLAB C18 和 Hi-Tech PICC-18 编译器进行编译。源文件会自动根据所使用的编译器进行必要的更改。 Microchip 协议栈设计为仅在 Microchip PIC18F 系列单片机上运行。Microchip 协议栈使用内部闪存程序存储器来存储可配置的 MAC 地址、网络表和绑定表。因此，必须使用可自编程的闪存存储器单片机。如果需要的话，可以修改非易失性存储器 （NVM）程序来支持任何其他类型的 NVM 而不使用可自编程的单片机。此外，该协议栈旨在在 PICDEM Z 演示板上运行。但是，它可很容易地移植到任何使用兼容单片机的硬件中。

图 2： MICROCHIP 协议栈架构



协议栈层

Microchip 协议栈根据 ZigBee 规范的定义将其逻辑分为多个层。实现每个层的代码位于一个独立的源文件中，而服务和应用程序接口 （Application Programming Interfaces， API）则在头文件中定义。

协议栈的当前版本不实现安全层。每个层为紧接着的上一层定义一组容易理解的函数。要实现抽象性和模块性，顶层总是通过定义完善的 API 和紧接着的下一层进行交互。特定层的 C 头文件（如 zAPS.h）定义该层所支持的所有 API。必须切记，用户应用程序总是与应用编程支持 （Application Programming Support， APS）层和应用层 （Application Layer， APL）交互。由每层提供的很多 API 都是简单的 C 语言宏，调用下一层中的函数。此方法可以避免与模块化相关的典型开销。

协议栈 API

Microchip 协议栈由很多模块组成。典型的应用程序总是与应用层（APL）和应用支持子层（APS）接口。但是，如果需要的话，也可以简单地将应用程序与其他模块接口并且 / 或者根据需要对它们进行自定义。以下部分仅提供对 APL 和 APS 模块的详细 API 描述。如果需要的话，可以在它们各自的头文件中了解其他模块的 API 详细信息。如需最新信息，可以参考实际的源文件。

应用层 （APL）

APL 模块提供高级协议栈管理功能。用户应用程序使用此模块来管理协议栈功能。zAPL.c 文件实现了 APL 逻辑，而 zAPL.h 文件定义 APL 模块支持的 API。用户应用程序将包含 zAPL.h 头文件来访问其 API。

APLInit

该函数初始化所有的协议栈模块。同时还初始化 APL 状态机。

语法

```
void APLInit(void)
```

参数

无

返回值

无

前提

无

副作用

无

注

在上电复位时， RF 收发器掉电。必须调用 `APLEnable()` 来使能 RF 收发器。

示例

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

APLIsIdle

该宏用于检测是否可以禁止 APL 和其他模块。

语法

```
BOOL APLIsIdle(void)
```

参数

无

返回值

TRUE（如果 APL 空闲并能禁止）

FALSE（其他情况）

前提

已调用 APLInit()

副作用

无

示例

```
// 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

该宏用于使能协议栈模块和 RF 收发器。

语法

```
void APLEnable(void)
```

参数

无

返回值

无

前提

已调用 APLInit()。

副作用

无

示例

...

```
// Initialize stack
APLInit();

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

APLDisable

该宏用于禁止 RF 收发器和其他协议栈模块。

语法

```
void APLDisable(void)
```

参数

无

返回值

无

前提

已定义 I_AM_END_DEVICE 并已调用 APLInit()。

副作用

所有等待接收的数据会全部丢失。

注

仅终端设备应禁止 RF 收发器来降低功耗。协调器和路由器应始终开启 RF 收发器。

示例

```
...

// 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

APLTask 是个协同任务函数，按顺序调用每一个协议栈模块任务函数。调用该函数使协议栈获取并处理进入的数据包。

语法

BOOL APLTask(void)

参数

无

返回值

TRUE （如果函数已完成任务并准备就绪于空闲状态）

FALSE （其他情况）

前提

已调用 APLInit()。

副作用

无

示例

```
...

// 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

该宏启动新网络的初始化。应重复调用 APLIsNetworkInitComplete 以使状态机运行并判断网络初始化是否完成。

语法

```
void APLNetworkInit(void)
```

参数

无

返回值

无

前提

已定义 I_AM_COORDINATOR，并已调用 APLInit()。

副作用

无

注

必须调用 APLIsNetworkInitComplete() 来判断网络初始化是否完成。

示例

```
...

// 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

该宏执行网络初始化状态机并指示网络初始化是否完成。这是一个协同任务，必须复调用直到返回 TRUE 为止。

语法

BOOL APLIsNetworkInit(void)

参数

无

返回值

TRUE （网络初始化已完成）。 TRUE 并不表明成功，仅表明网络初始化完成。必须调用 GetLastZError() 才能判断是否成功。

FALSE （其他情况）

前提

已定义 I_AM_COORDINATOR，并已调用 APLNetworkInit()。

副作用

无

注

返回值 TRUE 仅表明网络初始化过程已完成。但初始化过程可能成功也可能失败。调用 GetLastZError() 来判断过程是否成功。

示例

```
...

// 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

该宏指示网络层在当前的信道上组建新的网络。

语法

```
void APLNetworkForm(void)
```

参数

无

返回值

无

前提

已定义 I_AM_COORDINATOR，且 APLIsNetworkInitCompleted() = TRUE 以及 GetLastZError() = ZCODE_NO_ERROR.

副作用

无

注

无

示例

```
...

// 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 &#xD0; form it
            APLNetworkForm();
            ...
        }
        else
        {
            // New network could not be established
            // Do application specific recovery
        ...
}
```

APLPermitAssociation

该宏允许终端设备关联到网络。

语法

```
void APLPermitAssociation(void)
```

参数

无

返回值

无

前提

已定义 I_AM_COORDINATOR，并已调用 APLNetworkForm()。

副作用

无

注

根据应用的要求，您可能希望（或不希望）一个新设备连入网络。

示例

```
...

// 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

该宏不允许新设备连入网络。它与 APLPermitAssociation 相反。

语法

```
void APLDisableAssociation(void)
```

参数

无

返回值

无

前提

已定义 I_AM_COORDINATOR，并已调用 APLNetworkForm()。

副作用

无

注

当正常模式下运行时，协调器可能不希望任何新设备连入网络。在这种情况下，可以调用该函数以阻止任何设备连入网络。

当网络初次组建时，默认情况下禁止新关联。

示例

```
...

// 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

该宏保存目前为止收到的所有的关联和绑定请求，并将它们写入闪存程序存储器。关联请求一旦提交，协调器就会记住这些节点并在以后允许它们重新连入网络。

语法

```
void APLCommitAssociation(void)
```

参数

无

返回值

无

前提

已定义 I_AM_COORDINATOR。

副作用

修改网络和绑定表信息头。

注

一旦调用 APLPermitAssociation，所有的新关联请求将被自动写入闪存存储器。然而，它们只有在使用该宏进行提交后才会标识为有效。关联一旦提交，就会标识为有效并从那时起允许相应的终端设备在以后重新连入网络。

示例

```
...

// 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

终端设备使用该宏连入某个可能可用的网络。该宏仅启动 “Join” 状态机。必须重复调用 APLIsJoinComplete 来允许状态机执行并判断连接是否完成。

仅当已定义 I_AM_END_DEVICE 时，该宏才可用。

语法

```
void APLJoin(void)
```

参数

无

返回值

无

前提

已定义 I_AM_END_DEVICE。

副作用

无

注

在允许终端设备参与网络通信前，终端设备必须总是连入一个网络。在一个典型系统中，可能不希望终端设备连入任一可用网络而是连入先前所在的网络。在这种情况下，终端设备将只在特殊的 “配置” 模式下尝试连入新的网络。终端设备一旦成为某个网络成员，就可在以后简单地 “重新连入” 该网络。

示例

```
...

// 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

终端设备使用该宏来判断先前启动的连入过程是否已完成。

语法

BOOL APLIsJoinComplete(void)

参数

无

返回值

TRUE （如果连入过程已完成，必须调用 GetLastZError() 以判断是否成功）

FALSE （其他情况）

前提

已定义 I_AM_END_DEVICE 并已调用 APLJoin()。

副作用

无

注

返回值 TRUE 仅表明连入过程已完成。实际连入过程可能成功也可能失败。调用 GetLastZError() 以判断过程是否成功。

示例

```
...

// 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

终端设备使用该宏启动重新连接过程。在正常模式下运行时，终端设备应重新连入先前连入的网络，除非应用程序在每次启动时都要求查找并连入一个新的网络。

语法

```
void APLRejoin(void)
```

参数

无

返回值

无

前提

已定义 I_AM_END_DEVICE 并调用了 APLInit() 和 APLEnable()。

副作用

标识那些目前尚未关联的节点。

注

在正常模式下，终端设备将直接重新连入先前连入的网络。设备在能成功重新连入一个网络前必须已经连入过至少一个网络。此外，先前连入的网络必须在其射频范围内，以便能成功地重新连入该网络。

示例

```
...

// 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

终端设备使用该宏来判断先前启动的重新连接过程是否已完成。这是个协同任务；终端设备必须重复调用该宏。

语法

BOOL APLIsRejoinComplete(void)

参数

无

返回值

TRUE （重新连接过程已完成）。调用 GetLastZError 以判定成功 / 失败状态。

FALSE （其他情况）

前提

已定义 I_AM_END_DEVICE 并已调用 APLRejoin()。

副作用

无

注

要判断该节点是否成功重新连入先前的网络，则调用 GetLastZError()。

示例

```
...

// 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

终端设备使用该宏启动离开现有网络的离开过程。

语法

```
void APLLeave(void)
```

参数

无

返回值

无

前提

已定义 I_AM_END_DEVICE。

副作用

无

注

该宏仅设置离开过程的状态机。必须重复调用 APLIsLeaveComplete() 以完成离开过程。

示例

```
...

// 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

终端设备使用该宏来判断先前启动的离开过程是否已完成。

语法

BOOL APLIsLeaveComplete(void)

参数

无

返回值

TRUE （如果离开过程已完成）

FALSE （其他情况）

前提

已定义 I_AM_END_DEVICE。

副作用

无

注

返回值 TRUE 也许并不能完全表示离开尝试已成功。还必须调用 GetLastZError() 才能判断离开是否真正成功。

示例

```
...

// 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
   ...
```

应用支持子层 （APS）

APS 层主要提供 ZigBee 端点接口。应用程序将使用该层打开或关闭一个或多个端点并且获取或发送数据。它还为键值对 （Key Value Pair， KVP）和报文 （MSG）数据传输提供了原语。

APS 层同样也有绑定表。绑定表提供了端点和网络中两个节点间的群集ID对之间的逻辑链路。当首次对协调器编程时绑定表为空。主应用程序必须调用正确的绑定 API 来创建新的绑定项。参阅 DemoCoordApp 和 DemoRFDApp 演示程序作为工作示例。 Microchip APS 层把绑定表存储到闪存存储器内。闪存编程程序位于 zNVM.c 文件中。如果需要，可以用您的特定 NVM（非易失性存储器）程序代替 NVM 程序以支持不同类型的 NVM。

APS 还有一个“间接发送缓冲器”RAM，用来存储间接帧，直到目标接收者请求这些帧为止。根据ZigBee 规范，在星型网络中， RFD 设备总会将这些数据帧转发到协调器中。 RFD 设备可能不知道该数据帧的目标接收者。数据帧的实际接收者由绑定表项决定。协调器一旦接收到数据帧，它就会查找绑定表以确定目标接收者。如果该数据帧有接收者，就会将该帧存储在间接发送帧缓冲器里，直到目标接收者明确请求该帧为止。根据请求的频率，协调器必须将数据帧保存在间接发送帧缓冲器内。 zigbee.def 文件中定义的 MAC_MAX_DATA_REQ_PERIOD 编译时间选项定义了确切的请求时间。注意节点请求数据时间越长，数据包需要保存在间接发送缓冲器里的时间也越长。数据请求时间越长需要的间接缓冲空间越大。间接帧缓冲器包含一个设计时分配的固定大小的 RAM 堆。 zigbee.def 文件中 MAX_HEAP_SIZE 编译时间选项定义了缓冲器的大小。可通过动态分配间接发送帧缓冲器的RAM来添加新的数据帧。动态存储管理可充分利用间接发送帧缓冲空间。动态存储管理程序位于 SRAlloc.c 文件中。

APSInit

该函数通过清零其内部数据变量初始化 APS 层。用户应用程序不需要调用该函数。调用 APLInit 时自动调用该函数。

语法

```
void APSInit(void)
```

参数

无

返回值

无

前提

无

副作用

无

注

无

示例

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

APSTask

APSTask 是 APS 协同任务函数。应用程序不需调用该函数，当应用程序调用 APLTask 时会自动调用该函数。

语法

```
BOOL APSTask(void)
```

参数

无

返回值

TRUE （如果没有未完成的任务需要执行）

FALSE （其他情况）

前提

无

副作用

无

注

无

示例

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

APSDisable

该宏清除任何未完成的端点接收标志并准备为处理器休眠的 APS 模块。应用程序不需直接调用该宏，当调用 APLDisable 时，自动调用该宏。

语法

```
void APSDisable(void)
```

参数

无

返回值

无

前提

无

副作用

无

注

无

示例

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

APSOpenEP

该函数使主应用程序打开一个端点。在 ZigBee 协议中，数据通过端点交换。

语法

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

参数

srcEP [in]

指明源端点号，必须在 0-31 之间。

clusterID [in]

指明端点绑定的群集 ID。

destEP [in]

指明与源端点连接的目标端点。当 bDirect 设置为 TRUE 时使用该值。

bDirect [in]

指明该端点直接连接到指定的目标端点。目标节点取决于当前端点的绑定方式。

返回值

已打开端点的句柄。应用程序使用该句柄在以后访问端点。

前提

已调用 APSInit()。

副作用

将空端点标识为 “在用”，总可用端点数减 1。

注

端点等同于虚拟信道的一端。要完成虚拟信道，必须有两个端点，源端点和目标端点（即点对点连接）。在高级应用中，可能会有一个以上的端点 （即一对多连接），在这种情况下，可能会有多个接收者接收同一个数据包。目前版本的协议栈不支持一对多端点。在 ZigBee 术语中，连接用源端点和群集标识符（ID）定义。群集 ID 只是一个编号而已，用来标识将在虚拟信道上交换的数据变量集。还可把群集 ID 当作是添加到原虚拟信道的虚拟信道。因此，现在一组给定的源端点和目标端点可能会有多个群集 ID，这就要在一个虚拟信道内创建多个虚拟子信道。可以通过协调器建立虚拟连接（即间接连接）或直接连到节点 （即直接连接）。目前版本的协议栈仅支持终端设备和协调器间的间接连接和直接连接。你不能使用目前版本的协议栈在两个终端设备间创建点对点连接。当定义直接连接时 （即 bDirect = TRUE），必须提供目标端点信息。对于间接连接（即 bDirect = FALSE），不必提供目标端点信息。绑定进程将定义确切的目标端点。

要完成信道定义，必须提供源节点和目标节点地址信息。该函数自动使用当前的节点地址作为源节点，但没有定义目标节点地址。调用该函数后，您仅创建了虚拟信道的一部分，缺少目标地址信息。按照 ZigBee 规范，信道的源节点不需要知道目标节点的地址。必须在协调器节点内创建绑定表项，将本节点的源端点绑定到选定的另一节点的目标端点上。

示例

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

APSSetEP

该函数把给定端点设置为活动端点。后面的函数调用都将在该端点 （即活动端点）上进行。

语法

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

参数

h [in]

要激活的端点句柄。

返回值

无

前提

无

副作用

调用此函数后，APSCloseEP()、APSPut()、APSBeginMSG() 和 APSBeginKVP() 以及其他与端点相关的函数都将对给定的端点进行操作。

注

无

示例

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

APSCloseEP

该函数允许主应用程序关闭目前活动的已打开端点。

语法

```
void APSCloseEP(void)
```

参数

无

返回值

无

前提

已调用 APSSetEP()。

副作用

将给定端点标识为空闲。

注

无

示例

```
// 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

该函数启动 MSG 数据帧传输。MSG 是 ZigBee 规范定义的特殊数据传输机制。MSG 格式允许应用传输二进制数据字节流。 MSG 对于传输专用数据流或文件数据非常有用。请参阅 ZigBee 规范了解更多详细信息。

语法

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

参数

```
length  [in]
```

在该 MSG 帧中要发送的字节总数。

返回值

标识当前 MSG 帧的顺序事务 ID。

如果在启动 MSG 帧时发生错误，则返回 TRANS_ID_INVALID。

前提

已调用 APSSetEP()。

副作用

无

注

应用程序必须知道其要发送的作为 MSG 服务一部分的确切字节数。调用该函数后，应用程序必须使用 APSPut 装入实际数据字节。

示例

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

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

APSBeginKVP

该函数启动键值对（KVP）数据帧。KVP 是 ZigBee 规范定义的特殊数据传输机制。KVP 允许主应用程序传输变量－值对数据。当交换的数据是简单的变量值格式时，KVP 非常有用。例如，带温度传感器的节点可能会交换温度信道和相应的温度值。通常，标准 ZigBee 配置应用程序使用 KVP 来标准化数据传输的格式和内容。参阅 ZigBee 规范了解更多详细信息。

语法

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

参数

type [in]

事务类型。必须为下列中的一种：

类型	用途
TRANS_SET	设置变量值
TRANS_EVENT	设置事务
TRANS_GET_ACK	请求 ACK
TRANS_SET_ACK	发送 ACK
TRANS_EVENT_ACK	发送事务 ACK
TRANS_GET_RESP	发送 “获取”响应
TRANS_SET_RESP	发送 “设置”响应
TRANS_EVENT_RESP	发送事务响应

dataType [in]

事务数据类型。必须为下列中的一种：

类型	用途
TRANS_NO_DATA	无值
TRANS_UINT8	包含 8 位无符号值
TRANS_INT8	包含 8 位有符号值
TRANS_ERR	发生了错误
TRANS_UINT16	包含 16 位无符号值
TRANS_INT16	包含 16 位有符号值
TRANS_SEMI_PRECISE	包含 16 位半精度值 （基于 IEEE 754）
TRANS_ABS_TIME	包含 32 位值 （自 2000 年 1 月 1 日开始所经过的秒数）

attribId[in]

16 位属性 ID，指定目标节点内的特定属性。

返回值

用于发送该事务的顺序事务 ID。如果在启动 KVP 帧时发生错误，则返回 TRANS_ID_INVALID。

前提

已调用 APSSetEP()。

副作用

该函数不启动帧发送。必须根据属性值的要求，不调用或调用多个 APSPut，然后调用 APSSend 来启动发送。

注

无

示例

```
// 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

应用程序使用该函数来判断是否可以加载新帧。

语法

```
BOOL APSIsPutReady(void)
```

参数

无

返回值

TRUE （如果可以开始加载新帧）

FALSE （其他情况）

前提

已调用 APSSetEP()。

副作用

无

注

无

示例

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

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

APSSetClusterID

该函数设置与当前端点发送相关联的群集 ID。

语法

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

参数

clusterID[in]

要设置的群集 ID。

返回值

无

前提

```
APSIsGetReady() = TRUE
```

副作用

无

注

端点号和群集 ID 构成一个惟一对。ZigBee 应用程序使用该惟一对进行通信。ZigBee 规范建议应用程序为每个这样的惟一对保存绑定项。如果有端点使用多个群集 ID 通信， ZigBee 规范就建议应用程序保存多个绑定项，即使它们均与同一个端点相关联也是如此。但是为了节约 RAM， Microchip 协议栈仅对每个端点使用一个绑定项，而不管该端点使用多少个群集 ID。因此，如果端点使用多个群集 ID，就必须为该端点的每一个发送特别设置一个群集 ID。相似地，它会从接收到的帧中获取群集 ID 并对其进行相应的处理。

示例

```
// 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

该函数将给定字节加载到当前的发送缓冲器中。

语法

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

参数

v [in]

将装入帧缓冲器的数据字节。

返回值

无

前提

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

副作用

无

注

该函数仅把给定字节加载到发送缓冲器。必须调用 APSSend 以启动帧发送。

示例

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

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

APSPutArray

该函数将一个给定的字节数组装入发送缓冲器。

语法

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

参数

v [in]

将装入发送缓冲器的数据字节。

length [in]

将装入发送缓冲器的字节数。

返回值

无

前提

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

副作用

无

注

该函数仅把给定字节装入发送缓冲器。必须调用 APSSend 以启动帧发送。

示例

```
// 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

该函数发送当前发送缓冲器中的数据。

语法

```
void APSSend(void)
```

参数

无

返回值

无

前提

APSIsPutReady() == TRUE 并且至少调用了一个 APSPut() 或 APSPutArray()。

副作用

无

注

帧一旦被发送，帧的序号就保存在应答队列里，用于识别应答帧。调用函数必须重复调用 APSIsConfirmed 以检查应答，并当得到应答或超时时将该帧从队列中删除。

示例

```
// 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

该函数检查远程节点是否应答了活动帧。

语法

BOOL APSIsConfirmed(void)

参数

无

返回值

TRUE （如果活动帧被应答）

FALSE （其他情况）

前提

已定义 I_AM_END_DEVICE 并已调用 APSSetEP()。

副作用

无

注

该函数仅用于终端设备。终端设备发送所有的数据帧到协调器，然后必须等待来自协调器的应答。如果协调器发送数据帧，该帧就会被立即保存在间接发送缓冲器内，直到目标接收终端设备明确查询该帧为止。这就是为什么当协调器发送数据帧时会立即得到应答，而不需要通过调用 APSIsConfirmed 进行确认的原因。

示例

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

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

APSIsTimedOut

该函数检查应答是否超时。

语法

BOOL APSIsTimedOut(void)

参数

无

返回值

TRUE （如果活动帧超时）

FALSE （其他情况）

前提

已调用 APSSend()。

副作用

无

注

无

示例

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

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

APSRemoveFrame

该宏将活动帧从应答队列中删除。

语法

```
void APSRemoveFrame(void)
```

参数

无

返回值

无

前提

已调用 APSSend()。

副作用

无

注

当第一次发送帧时，该帧的序号被保存在由 MAC 层管理的应答队列中。当接收到应答帧时，MAC 层将应答帧的序号与应答队列中的相应项进行比较。如果匹配，表示队列中相应的项得到确认。当帧得到应答或超时或者不需要应答时，应用程序必须调用该函数将该帧从队列中删除。

示例

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

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

APSIsGetReady

该函数检查活动端点是否有等待接收的数据。

语法

BOOL APSIsGetReady(void)

参数

无

返回值

TRUE（如果有等待接收的数据）

FALSE（其他情况）

前提

已调用 APSSetEP()。

副作用

无

注

无

示例

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

        ...

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

APSGetDataLen

该函数获取当前与活动端点相关联的接收帧中剩下的数据字节数。

语法

```
BYTE APSGetDataLen(void)
```

参数

无

返回值

要获取的剩下的字节数。

前提

```
APSIsGetReady() = TRUE
```

副作用

无

注

无

示例

```
// 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

该函数从接收缓冲器中获取一个数据字节。

语法

```
BYTE APSGet(void)
```

参数

无

返回值

实际获取的数据字节。

前提

```
APSIsGetReady() = TRUE
```

副作用

无

注

无

示例

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

        ...

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

APSGetArray

该函数从接收缓冲器获取一个数据字节数组。

语法

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

参数

buffer [out]

保存数据数组的缓冲器。

count [in]

要获取的字节总数。

返回值

实际获取的字节数。

前提

```
APSIsGetReady() = TRUE
```

副作用

无

注

无

示例

```
// 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

该函数将当前接收的帧从接收缓冲器中删除。

语法

```
void APSDiscardRx(void)
```

参数

无

返回值

无

前提

```
APSIsGetReady() = TRUE
```

副作用

无

注

无

示例

```
// 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

该函数获取与当前端点接收相关联的群集 ID。

语法

```
BYTE APSGetClusterID(void)
```

参数

无

返回值

与接收端点数据相关联的群集 ID。

前提

```
APSIsGetReady() = TRUE
```

副作用

无

注

无

示例

```
// 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();
}
...
```

网络层

网络层 （NWK）负责建立和维护网络连接。它独立处理传入数据请求、关联、解除关联和孤立通知请求。

典型的应用不需要直接调用 NWK 层。如有需要，可以参阅 NWK.c 和 NWK.h 文件了解对 NWK 层 API 的详细描述。

ZigBee 设备对象

ZigBee 设备对象（ZDO）打开和处理 EP 0 接口。ZDO 负责接收和处理远程设备的不同请求。不同于其他的端点，EP 0 总是在启动时就被打开并假设绑定到任何发往 EP 0 的输入数据帧。

ZDO 对象允许远程设备管理服务。远程设备管理器会向 EP 0 发出请求， ZDO 会处理这些请求。下面是一些远程服务实例：NWK_ADDR_REQ（给出网络地址），NODE_DESC_REQ（给出节点描述符）和 BIND_REQ（绑定源 EP、群集 ID 和目标 EP）。一些请求仅适用于协调器。参阅 ZDO.c 获取完整列表。还应参阅 ZigBee 规范了解更多信息。

应用程序除非在启动时进行初始化，否则无需直接调用任何 ZDO 函数。如有需要，可以参阅 ZDO.c 和 ZDO.h 文件了解对 ZDO API 的详细描述。

ZigBee 设备配置层

ZigBee 设备配置层提供标准的 ZigBee 配置服务。它定义和处理描述符请求。远程设备可能通过 ZDO 接口请求任何标准的描述符信息。当接收到这些请求时，ZDO 会调用配置对象以获取相应的描述符值。在目前的版本中，还没有完全实现设备配置层。

应用程序不需要直接调用任一配置函数。如有需要，可以参阅 zProfile.c 和 zProfile.h 文件了解对 API 的详细描述。

介质访问控制层

介质访问控制（Medium Access Control ， MAC）层实现了 IEEE 802.15.4 规范所要求的功能。MAC 层负责同物理（Physical， PHY）层进行交互。为支持不同类型的 RF 收发器， Microchip 协议栈将不同的 PHY 交互归类到不同的文件中。每个支持的收发器都有一个独立的文件。注意由于 RF 收发器功能上的差异，MAC 和 PHY 文件不能完全独立。MAC 文件根据当前的 RF 收发器调整自身的某些逻辑。

在当前版本的协议栈中， zPHYCC2420.c 文件实现 Chipcon CC2420 2.4 GHz 收发器特定的功能。将来，随着对 RF 收发器支持的添加，也将会相应地添加新的 PHY 文件。所有的 RF 收发器 PHY 文件使用 zPHY.h 文件作为它们与 MAC 层的主接口。

典型应用不需要直接调用 MAC 层。如有需要，可以参阅 MAC.c、MAC.h 和 PHY.h 文件了解对 MAC/PHY API 的详细描述。

总结

Microchip ZigBee 协议栈提供了一种易于使用的不依赖于应用和 RTOS 的函数库。该函数库是专为仅需对上层软件做极小更改就可支持多个 RF 收发器而设计的。应用程序能容易地从一个 RF 收发器移植到另一个收发器。此外，它还自动支持 MPLAB C18 和 Hi-Tech PICC-18 C 编译器。如有需要，能很容易地修改该协议栈以支持其他编译器。

参考书目

- *“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

源代码

可从 Microchip 公司网站 **www.microchip.com** 下载完整源代码 （包括所有演示应用程序和必要的支持文件）的压缩文件。

常见问题解答

问： Microchip ZigBee协议栈与ZigBee协议兼容吗？

答： 不。目前版本的 Microchip 协议栈与 ZigBee 协议不完全兼容。

问： 如何获取 Microchip ZigBee 协议栈的源代码？

答： 您可以从 Microchip 网站（www.microchip.com）的 AN965 或 PICDEM Z 页下载。

问： 如何获取目标硬件设计文件？

答： 您可以从 Microchip 网站的 PICDEM Z 页下载。

问： 我需要使用哪些工具来使用 Microchip 协议栈开发 ZigBee 应用程序？

答： 需要：

- 至少一个 PICDEM 演示工具包或至少两个 ZigBee 节点
- 完整的 Microchip ZigBee 协议栈源代码
- MPLAB C18 或 Hi-Tech PICC-18 C 编译器
- MPLAB IDE 软件
- 器件调试器和编程器，如 MPLAB ICD 2

问： 一个典型的ZigBee节点需要多大的程序和数据存储空间？

答： 确切的程序和数据存储空间要求取决于节点和选定的编译器的类型。此外，随着对原有功能的改进和新功能的增加，对存储容量的要求也会随之改变。请参阅 `version.log` 文件了解更多详细信息。

问： 要运行 ZigBee 协调器和终端设备所需的最小处理器时钟速率是多大？

答： 通常，ZigBee 协调器应以较高的速度运行，因为它必须准备好处理多个节点的数据包。协调器时钟速率取决于网络中的节点数。演示协调器使用 16 MH（4 MHz，带有 4 倍频 PLL）的时钟速率，并可支持至少 10 个节点。我们还未提供时钟频率及其对应网络中的节点数的特性数据。终端设备运行速度不需要与协调器一样快。一个简单的终端设备可能只要运行在 4 MHz 时钟速率下。

问： 我可以使用内部 RC 振荡器运行 Microchip 协议栈吗？

答： 可以，您可以使用内部 RC 振荡器运行 Microchip 协议栈。如果您的应用要求稳定的时钟来执行对时间敏感的操作，您必须确保内部 RC 振荡器满足您的要求或者您可以周期性将内部 RC 振荡器校准在理想范围内。

问： PICDEM Z 板的典型射频范围是多少？

答： 确切的射频范围取决于 RF 收发器和使用天线的类型。对于带 PCB 引线天线，基于 Chipcon 2.4 GHz 的节点，应该大约为 100 米的距离。当在建筑物内，因墙壁和其他的障碍物会造成实际范围会有所减小。

问： 我已有一个使用有线协议，如 RS-232 和 RS-485 等的应用。我怎样才能把它转成基于 ZigBee 协议的应用呢？

答： 您需要开发一个 ZigBee 协调器和多个 ZigBee 终端设备应用。协调器用于创建和管理网络。如果您现有的网络有一个主控制器和多个终端设备或传感器，则主控制器可以作为 ZigBee 协调器，传感器可以作为 ZigBee 终端设备。您必须确保您的应用可以接受特定 RF 收发器提供的射频范围。

问： 我现已经创建了一个协调器和多个终端设备。如何绑定设备，使我可以在设备间传输数据？

答： 处理这个问题有两种方法。您可以编写自定义绑定逻辑或使用标准的 ZigBee 设备管理器接口，来创建或修改协调器表中的绑定表项。在发布该文档之际，市场上还没有标准的 ZigBee 设备管理器。本应用笔记中包含的演示应用程序使用自定义的绑定机制，使用 MSG 帧发送绑定请求。请参阅 `DemoRFDApp.c` 文件中的自定义绑定函数 `InitCustomBind`、`StartCustomBind` 和 `IsCustomBindComplete`。演示协调器同样包含处理自定义绑定请求的逻辑。请参阅 `DemoCoordApp.c` 文件中的 `ProcessCustomBind` 函数了解更多信息。

问： 如何获取 ZigBee 和 IEEE 802.15.4 规范文档？

答： IEEE 802.15.4 规范可从 IEEE 网站免费获取。

在发布本文档之际，还没有发布 ZigBee 规范，只有 ZigBee 成员可以获取该规范。查看 ZigBee 网站了解与获取发布的规范有关的信息。

注：

请注意以下有关 Microchip 器件代码保护功能的要点：

- Microchip 的产品均达到 Microchip 数据手册中所述的技术指标。
- Microchip 确信：在正常使用的情况下， Microchip 系列产品是当今市场上同类产品中最安全的产品之一。
- 目前，仍存在着恶意、甚至是非法破坏代码保护功能的行为。就我们所知，所有这些行为都不是以 Microchip 数据手册中规定的操作规范来使用 Microchip 产品的。这样做的人极可能侵犯了知识产权。
- Microchip 愿与那些注重代码完整性的客户合作。
- Microchip 或任何其它半导体厂商均无法保证其代码的安全性。代码保护并不意味着我们保证产品是 “牢不可破”的。

代码保护功能处于持续发展中。 Microchip 承诺将不断改进产品的代码保护功能。任何试图破坏 Microchip 代码保护功能的行为均可视为违反了《数字器件千年版权法案（Digital Millennium Copyright Act）》。如果这种行为导致他人在未经授权的情况下，能访问您的软件或其它受版权保护的成果，您有权依据该法案提起诉讼，从而制止这种行为。

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

Microchip 位于美国亚利桑那州 Chandler 和 Tempe 及位于加利福尼亚州 Mountain View 的全球总部、设计中心和晶圆生产厂均于 2003 年 10 月通过了 ISO/TS-16949:2002 质量体系认证。公司在 PICmicro® 8 位单片机、 KEELOQ® 跳码器件、串行 EEPROM、单片机外设、非易失性存储器和模拟产品方面的质量体系流程均符合 ISO/TS-16949:2002。此外， Microchip 在开发系统的设计和生产方面的质量体系也已通过了 ISO 9001:2000 认证。

全球销售及服务网点

美洲

公司总部 Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 1-480-792-7200
Fax: 1-480-792-7277
技术支持：
http://support.microchip.com
网址：www.microchip.com

亚特兰大 Atlanta
Alpharetta, GA
Tel: 1-770-640-0034
Fax: 1-770-640-0307

波士顿 Boston
Westborough, MA
Tel: 1-774-760-0087
Fax: 1-774-760-0088

芝加哥 Chicago
Itasca, IL
Tel: 1-630-285-0071
Fax: 1-630-285-0075

达拉斯 Dallas
Addison, TX
Tel: 1-972-818-7423
Fax: 1-972-818-2924

底特律 Detroit
Farmington Hills, MI
Tel: 1-248-538-2250
Fax: 1-248-538-2260

科科莫 Kokomo
Kokomo, IN
Tel: 1-765-864-8360
Fax: 1-765-864-8387

洛杉矶 Los Angeles
Mission Viejo, CA
Tel: 1-949-462-9523
Fax: 1-949-462-9608

圣何塞 San Jose
Mountain View, CA
Tel: 1-650-215-1444
Fax: 1-650-961-0286

加拿大多伦多 Toronto
Mississauga, Ontario,
Canada
Tel: 1-905-673-0699
Fax: 1-905-673-6509

亚太地区

中国 - 北京
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

中国 - 成都
Tel: 86-28-8676-6200
Fax: 86-28-8676-6599

中国 - 福州
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

中国 - 香港特别行政区
Tel: 852-2401-1200
Fax: 852-2401-3431

中国 - 青岛
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

中国 - 上海
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

中国 - 沈阳
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

中国 - 深圳
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

中国 - 顺德
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

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

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

台湾地区 - 高雄
Tel: 886-7-536-4818
Fax: 886-7-536-4803

台湾地区 - 台北
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

台湾地区 - 新竹
Tel: 886-3-572-9526
Fax: 886-3-572-6459

亚太地区

澳大利亚 Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

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

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

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

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

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

马来西亚 Malaysia - Penang
Tel:604-646-8870
Fax:604-646-5086

菲律宾 Philippines - Manila
Tel: 011-632-634-9065
Fax: 011-632-634-9069

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

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

欧洲

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

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

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

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

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

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

西班牙 Spain - Madrid
Tel: 34-91-352-30-52
Fax: 34-91-352-11-47

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

07/01/05