Microchip ZigBee 协议栈

一. 引言

1.1 Microchip ZigBee 协议栈介绍

ZigBee™ 是专为低速率传感器和控制网络设计的无线网络协议。有许多应用受益于ZigBee 协议，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、医学领域以及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，本文包含了一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。

1.2 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IEEE802.15.4 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 协议和IEEE 802.15.4 规范。

1.3特性

Microchip ZigBee 协议栈设计随着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3.5版本具有以下特点：

1. 基于ZigBee 协议规范的3.5版本；
2. 使用Chipcon CC2420 RF 收发器支持2.4 GHz 频带；
3. 支持所有ZigBee设备类型（包括协调器、路由器和终端设备）；
4. 在协调器节点中实现对相邻表和绑定表的非易失性存储；
5. 可以在大多数PIC18 系列单片机之间进行移植；
6. 不依赖于RTOS 和应用；
7. 支持Microchip MPLAB® C18 编译器；
8. 模块化设计和标准术语使用ZigBee协议和IEEE 802.15.4 规范所提供的术语。

1.4 限制

Microchip 协议栈的3.5 版本中有安全方面的限制。请注意随着时间的推移， Microchip会添加新特性。

1.5 注意事项

ZigBee 协议所描述的一些高级决议，Microchip 协议栈没有给出相应的支持功能：

1. 仅支持时隙网络（不支持信标网络）；
2. 离开网络的节点的网络地址不能再分配；
3. 不能完成节点自动从相邻表中移出；
4. 不支持 PAN ID 的冲突解决；
5. 不支持自动路由修复；
6. 改变 PAN 协调器的能力。

二． ZigBee 协议概述

2.1 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（冗余校验） 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。

2.2 设备类型

IEEE 802.15.4定义了两种设备类型。表1中给出了定义的设备类型，表2中列出了ZigBee协议中的三种设备类型，以及他们与IEEE设备类型的关系。

表1：IEEE 802.15.4定义的设备类型

设备类型	提供的功能	供电电源	接收器配置
全功能设备（FFD）	所有	交流电	空闲时开启
简化功能设备（RFD）	受限	电池	空闲时关闭

表2：ZigBee协议的设备类型

ZigBee协议设备	IEEE设备类型	典型功能
协调器	FFD	每个网络只有一个。用来构建网络，分配网络地址，建立保存绑定表。
路由器	RFD	可选择的。用来扩展网络的物理范围，允许多个节点加入网络，可能用来监测、控制。
终端	FFD 或RFD	完成监测和（或）控制功能

2.3 网络构造

ZigBee协议的无线网络可以有多种构造类型，在所有的网络构造中，至少有两个重要的组成部分：

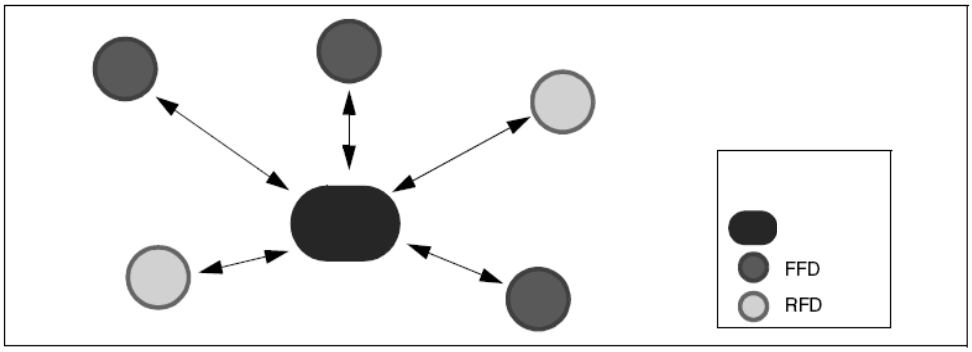
- 协调器
- 终端设备

ZigBee 协调器是一种特殊FFD变量，它可以完成ZigBee 协议所设置的大量服务。而终端设备可能是FFD 或者RFD。RFD是一个小的、简单的ZigBee 协议节点。它实现的仅仅是ZigBee 协议所提供服务中的最小部分。网络中可选用的是ZigBee协议路由器。

2.3.1 星型网络结构

一个星形网络结构包括一个ZigBee 协调器和一个或多个终端设备。在这种网络结构中，所有的终端设备通信仅仅靠协调器来完成。如果一个终端设备需要传输数据给另一个终端设备，它先发送数据给协调器，协调器再转发数据给已经确定的接收设备。

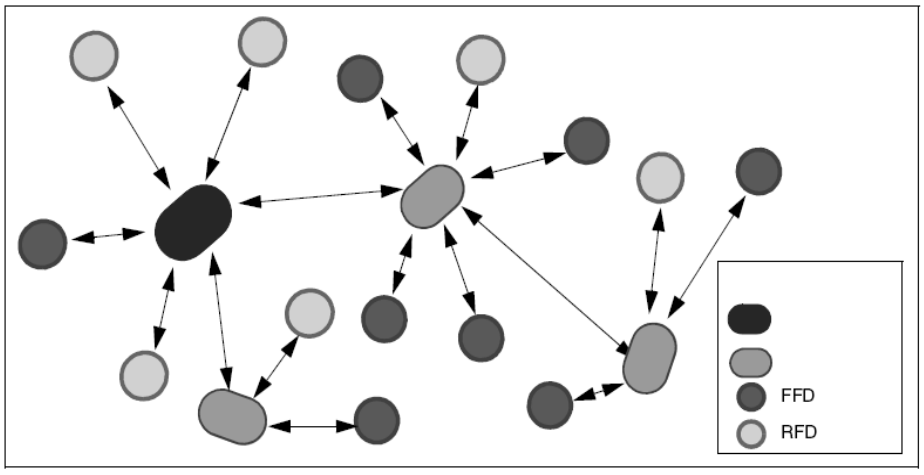
图1：星型网络结构



2.3.2 簇—树型拓扑

另外一种网络结构是簇—树型拓扑。在这种网络中，终端设备可能加入ZigBee 协调器，也有可能是ZigBee 路由器。路由器有两种功能。一种是增加网络中节点数量，另一种是扩展网络的物理范围。增加路由，终端设备便可以在协调器的网络覆盖范围已外。在簇—树型拓扑中，消息沿树型传送。

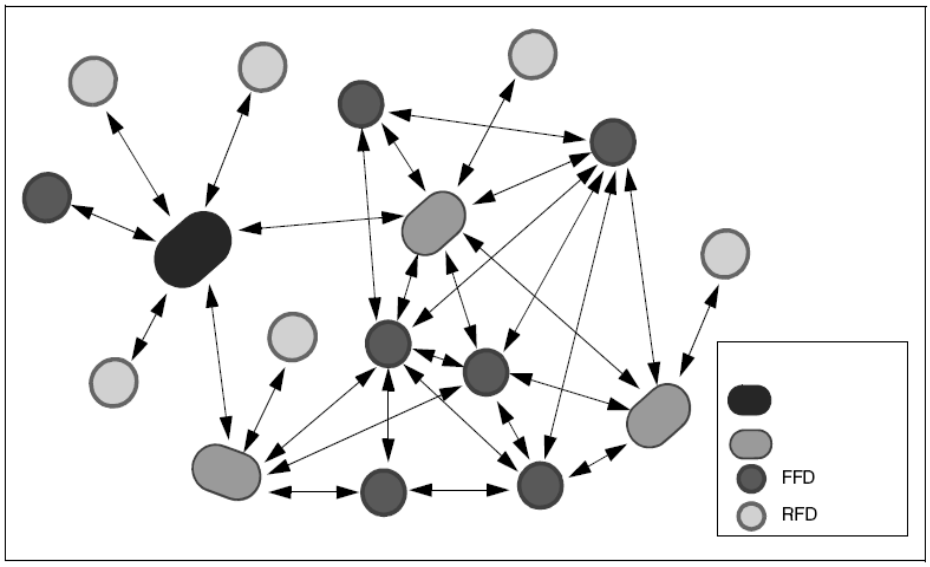
图2：簇—树型拓扑



2.3.3 网型网络

网型网络与簇—树型网络相似，除了FFD 可以直接与另外的FFD 进行消息路由，代替了树型结构。RFD 发送消息仍必须要经过它的父节点。这种拓扑结构的优势在于可以减少消息的反应时间，增加消息的可靠性。

图3：网型网络



簇—树型网络和网型网络都属于多跳型网络，这是由于它们有能力通过多个设备发送信息包，而星型网络属于单跳型网络。ZigBee 网络属于多通道网络，这意味着网络中的每一个节点都有合适的通道作为传输媒介。在这里存在两种类型的多通道机制，信标和无信标。在无信标使能的网络中，网络中的所有节点都被允许在任何时候使用空闲信道进行传输。在信标使能的网络中，节点仅在预定的时间间隙内进行传输。协调器周期性的使用超帧识别作为信标帧，所有的网络节点需要与此帧同步，在节点被允许发送或接收数据时，每一个节点在超帧中被分配一个明确的时隙。在所有的节点竞争访问通道时，每一个超帧将包括一个公共的时隙。Microchip 堆栈当前的版本仅仅支持无信标网络。

2.4 ZigBee 协议术语

ZigBee 协议的规范（profile）是一种简单的物理和界面描述，通常没有代码与框架联系。每一个数据块可以在不同的设备之间进行传输，像开关状态和电位计读数，这些被称之为属性（attribute）。每一个属性分配给一个唯一的标识符。一些属性被联合起来形成簇（clusters）。每一个群也被分配给唯一的标识符。界面被列入簇标准而不是属型标准，通过属性进行个别调用。

规范定义了属性的ID 值和簇的ID 值，也就是每一个属型的格式。例如，在家居控制中照明部分，簇OnOffDRC（Dimmer Remote Control 调光远程控制器）包括一个属性——打开/关闭，它将使用一个无符号8位数，0xFF 表示打开，0x00 表示关闭，0xF0 表示固定输出。

规范描述了簇的必要性，并且对于每一个设备它是可选择的。总之，规范定义了一些可选择的ZigBee 协议服务。用户可以调用这些定义并且用它编写自己的代码。可以用任何方法写自己想要的代码，也可以进行任意的功能分组，只要支持命令群和服务，并且使用的这

些属型在规范中都有所定义。按照这种方法，不同厂商生产的东西将可达到交融。

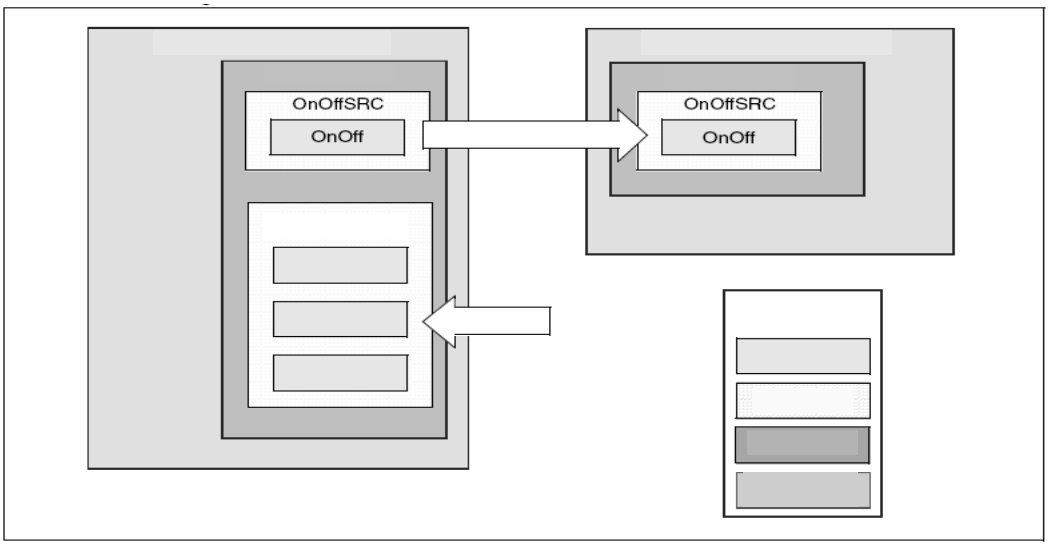
还是前面的那个例子，家居控制中，照明框架部分指定了六种设备。ZigBee协议的Microchip堆栈提供了一个由下列信息组成的文件来支持这一框架：

- 规范ID
- 设备ID和版本
- 簇ID
- 属性ID
- 属性数据类型

每一个支持一个或多个群的功能代码块被称为一个终端（endpoint）。不同的设备之间通信是由它们的终端和簇支持的。

图4种显示了不同的术语之间怎样发生关系。图中显示了家居控制照明部分的两个设备，每一个是一个终端。开关的本地控制器有一个输入簇在终端上，开关的远程控制器有一个输入簇和一个输出簇在它的终端上。这个开关的使用，可以使这两个群被分离在不同的终端上，而数据流在群层上传输。

图4：ZigBee协议的框架体系



2.5 消息

2.5.1 消息类型和绑定

如果这个设备知道网络中其它设备的网络地址，那么它便可以和其它网络中的设备进行通信。这种消息被称为直接消息。然而这种方式需要大量的信道发现和维持原地址和目的地址的关系。ZigBee 协议提供了一种服务称为“绑定”，用于简单消息传输。基于簇/终端水平，在服务和网络设备的需求之间，ZigBee 协议协调器可以创建相匹配的表，即绑定表。建立的一对关系称之为一个“绑定”。绑定可以由设备自身或一个授权的设备创建，可以由协调器或其它设备创建。一旦绑定建立，两个设备将可以通过协调器进行通信。间接消息则是指原地址发送消息给协调器，再由协调器转发给一个或者多个目的地址。

开关远程控制器

终端7

2.5.2 消息格式

一条ZigBee 协议消息包括127字节，具体如下：

SRC程序

不考虑

1. MAC 报头——该报头包含当前被传输消息的源地址及目的地址．若消息被路由，则该地址有可能不是实际地址，产生及使用该报头对于应用代码是透明的．
2. NWK 报头——该报头包含了消息的实际源地址及最终的目的地址，该报头的产生及使用对于应用代码是透明的．
3. APS 报头——该报头包含了配置ID，簇ID及当前消息的目的终端．同样，报头的产生及使用是透明的．
4. APS 有效负载——该域包含了待应用层处理的Zigbee协议帧．

2.5.3 ZigBee 协议帧结构

ZigBee 协议有两种帧结构格式：KVP（Key Value Pair 键值对）帧结构和MSG（Message 消息）帧结构。KVP是ZigBee 规范定义的特殊数据传输机制，通过一种规定来标准化数据传输格式和内容，主要用于传输较简单的变量值格式。MSG是ZigBee 规范定义的特殊数据传输机制，其在数据传输格式和内容上并不作更多规定，主要用于专用的数据流或文件数据等数据量较大的传输机制。

KVP 帧是专用的比较规范的信息格式，采用键值对的形式，按一种规定的格式进行数据传输．通常用于传输一个简单的属性变量值；而MSG 帧还没有一个具体格式上的规定，通常用于多信息，复杂信息的传输。KVP、MSG 是通讯中的两种数据格式。如果将帧比作一封邮件，那么信封、邮票、地址人名等信息都是帧头、帧尾，里面的信件内容就是特定的数据格式KVP 或MSG。根据具体应用规范，KVP一般用于简单属性数据，MSG 用于较复杂的，数据量较大信息。

2.5.3.1 KVP 帧

一个KVP 帧包含以下信息，依次是：

1. 处理计数；
2. 结构类型；
3. 处理——处理序列数
   命令类型和属性数据类型
   属性ID
   误码（可选择）
   属性数据（可变长度）

2.5.3.2 MSG 帧

一个MSG帧包含以下信息，依次是：

1. 处理计数；
2. 结构类型；
3. 处理——处理序列数
   处理长度
   处理数据

发送和接收设备需要知道处理数据的格式。

2.6 寻址

ZigBee 协议网络中的每一个节点都有两个地址：一个是64位的MAC 地址，一个是16位的网络地址。因此有两种可选择的寻址方式。

2.6.1 IEEE 扩展唯一标识符（Extended Unique Identifiers）– EUI-64

每一个使用 ZigBee 协议通信的终端必须有一个全球唯一的 64 位 MAC 地址，前 24 位为有组织的唯一的辨识符（Organizationally Unique Identifier OUI），后 40 位由厂商自定。OUI 必须从 IEEE 购买，保证全球的唯一性。

2.6.2 网络地址

在终端加入网络的过程中，终端使用它的扩展地址进行通信。而当它成功的加入网络后，它被分配给一个 16 位的网络地址，并用它进行通信。

2.7 数据传输

2.7.1 数据传输机制

对于非信标网络，当一个设备想要发送一个数据帧时，它会等待信道空闲，直到检测到信道为空后设备会传输该帧。

若目的设备为FFD 全功能设备，它的接收器应始终保持开启状态，以便其它的设备可随时向它传输数据。但是若设备为RFD 精简功能设备，无操作时设备将关闭收发器以节约能量。此时RFD 设备无法接收到任何数据。因此，其它设备只能通过RFD 的FFD 父亲向RFD 设备请求或发送数据。直到RFD 上电RX 收发器后，它会向父亲请求自己的信息数据，若父亲缓冲区中存有发给孩子的信息，则将该信息发给孩子设备。该操作模式可降低RFD 的功耗，但相应的父亲FFD 节点应拥有足够的RAM 空间，以便为孩子设备缓冲信息。若孩子设备没有在规定的时间内请求信息，信息将被丢失。

2.7.2 单播

当单播一个消息时，数据包的MAC 报头中应含有目的节点的地址，只有知道了接收设备的地址，消息方可以单播方式进行发送。

2.7.3 广播

要想通过广播来发送消息，应将信息包MAC报头中的目的地址域置为0xFF。此时，所有射频收发使能的终端皆可接收到该信息。

该寻址方式可用于加入一个网络、查找路由及执行ZigBee协议的其它查找功能。ZigBee协议对广播信息包实现一种被动应答模式。即当一个设备产生或转发一个广播信息包时，它将侦听所有邻居的转发情况。如果所有的邻居都没有在应答时限*nwkPassiveAckTimeout* 秒内复制数据包，设备将重复转发信息包，直到它侦听到该信息包已被所有邻居转发，或者广播传输时间*nwkNetworkBroadcastDeliveryTime* 秒被耗尽时停止。

2.7.4 路由

Microchip 堆栈提供了路由消息的能力。在没有任何来自终端应用的干涉下，路由由堆

栈自动建立。路由使得网络范围进行扩展，ZigBee 协调器也可以通过路由器加入网络。

当发送消息时，路由的类型需要说明。这里有三种路由形式提供给大家：

1. 禁止——如果是网型网，消息沿路由路线传输。否则，沿树型发送。
2. 使能——如果是网型网，消息沿路由路线传输。如果网型网内路线不能被确定，路由器寻找路线，当找到后，消息沿着给定路由路线发送。如果路由器没能找到新的路线，则沿树型发送。
3. 强制——如果路由器有路由能力，它将进行路由发现，即使已经存在路由。当发现完成，选择最优的路由进行传输。如果路由器没有路由能力，它将沿树型传输。这种方法节省资源，产生了大量的网络通信路线。他最初的目的是用来修复路由路线。

2.8 网络关联

ZigBee 协调器首先建立一个ZigBee 协议网络。协调器一启动，在它自己被允许的通道上便开始搜索其它的协调器。在信道被激活的基础上，协调器才可以建立它自己的网络，并选定唯一的16位的PAN ID。一旦新的网络建立，ZigBee 路由器和其它终端设备才被允许加入网络。

一旦网络建立，由于物理上的变化，可能有多个网络发生重叠，PAN ID 发生冲突。在这种情况下，两个发生冲突的网络，其中一个网络的协调器需要更改PAN ID和（或者）信道，另外一个不作变化。受到影响的协调器，需要通知子节点进行必要的改变。当前版本的Microchip 堆栈不支持PAN ID 冲突解决。

ZigBee 设备存储网络中其它节点的信息，包括父节点和子节点，在一个非易失性网络中，这被称之为相邻表。当功能唤醒时，一个从前属于网络中的子设备，它决定查找它的相邻表，并向相邻表中它以前的父节点发出作为孤节点的通知程序，用来查找它曾经加入的网络。设备收到孤节点的通知后，它将查找他的相邻表，看孤节点是否是它的子节点，如果是，此设备将通知孤节点它在这个网络中的位置。如果孤节点通知失败，或者在它的相邻表中没有父节点，它将以一个新的设备加入网络。它将建立一个现有的可能的父亲设备的目录，并试着加入一个最佳的。

加入网络后，设备同样也可以与网络脱离。与网络脱离有两种情况。一种是父节点发出请求，要求你离开；另一种是子设备自动要求离开。

一个设备花费在确定信道激活和获得信道的总时间由ScanDuration 参数决定。需要关于扫描参数的具体内容请参阅“ZigBee协议定时器”。对于2.4GHz 的频带宽度来说，扫描时间可以由下列公式计算得出：

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

对于Microchip 堆栈，ScanDuration 在0到14之间，给定的扫描时间为0.031秒/信道~4.2分钟/信道。如果ScanDuration 在8和全部的16个信道上设置，一个设备将花费超过一分钟来完成一次扫描。ZigBee 路由器和终端设备完成一次扫描来确定可获得的网络，而ZigBee 协调器则需要两次扫描，一次是用来对激活信道进行采样，一次是用来确定网络。确定扫描周期要考虑到指定信道扫描完一次所需的时间和系统启动所分配的时间。

2.9典型的zigbee协议节点硬件

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™ 接口的微处理器
- 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

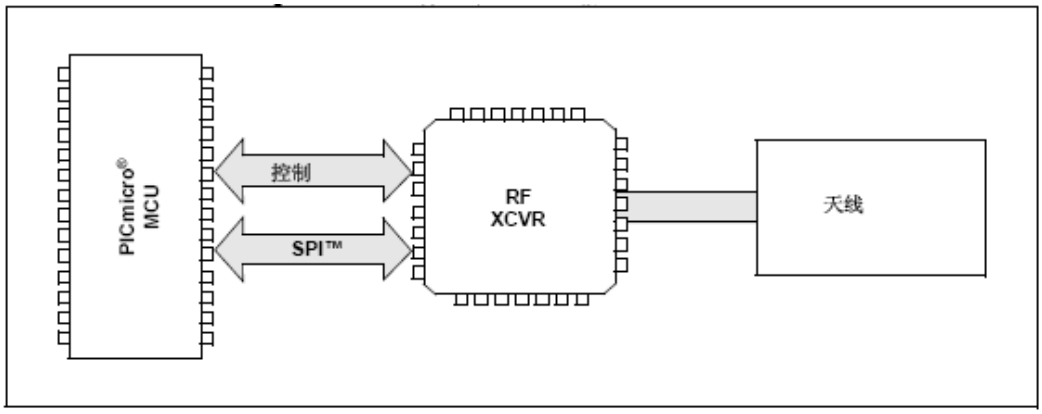
如图6所示，控制器通过SPI 总线和一些离散控制信号与RF 收发器相连。控制器充当SPI 主器件而RF 收发器充当从器件。控制器实现了IEEE 802.15.4 MAC 层和ZigBee 协议层。它还包含了特定应用的逻辑。它使用SPI 总线与RF 收发器交互。Microchip 协议栈提供了完全集成的驱动程序，免除了主应用程序管理RF 收发器功能的任务。如果您在使用Microchip ZigBee 节点的参考原理图，那么无需做任何修改就可以开始使用Microchip 协议栈了。如果需要，可以将某些非SPI 控制信号重新分配到其他端口引脚以适合你的应用的硬件。在这种情况下，必须修改物理层接口定义来包括正确的引脚分配。

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”）。

Microchip 堆栈设计提供了3.3v电压给控制器和射频收发器。根据需要，可以选择用交流电源供电和电池供电两种供电模式。典型的，协调器和路由器使用交流电源供电，终端设备使用电池供电。当使用电池供电时，必须保证供电电压在指定的范围内。

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

图6：典型的zigbee协议节点硬件

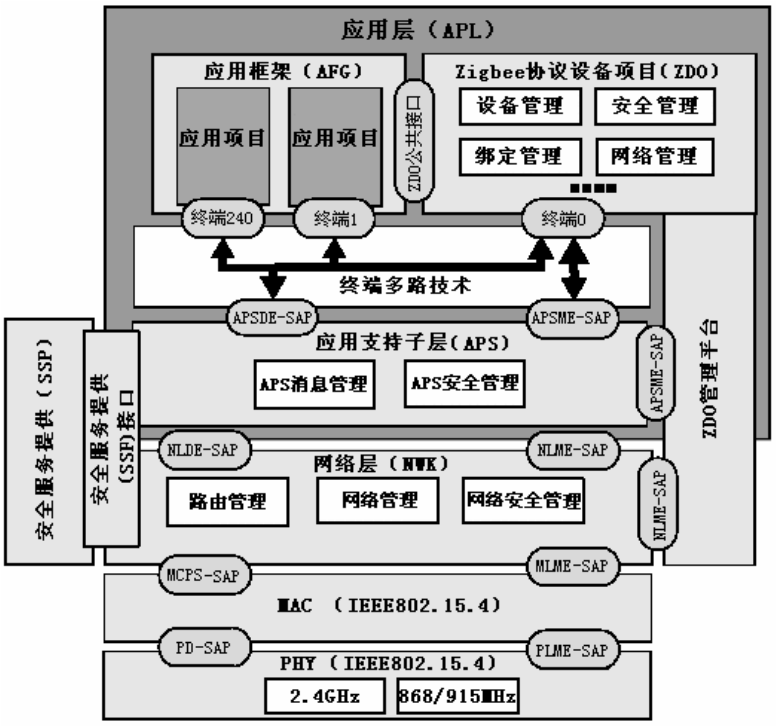


三． Microchip ZigBee 协议栈架构

Microchip 堆栈用C 语言编写，并运行在MicrochipPIC18F 的微处理器系列。Microchip 堆栈使用嵌入式程序闪存存储一些参数，包括MAC 地址，相邻表和绑定表。因此您必须使用带有程序闪存的微处理器。如果需要，您可以更改非易失性存储器（Nonvolatile Memory——NVM）程序用来支持其它类型的NVM，而不使用自带可编程的微处理器。

Microchip 堆栈设计遵循ZigBee协议和IEEE802.15.4规范，每一层在自己的源文件中都有定义。术语来自规范当中。两种规范中最初的定义通过一些简单的功能调用，用于界面及堆栈，用参数目录说明规范中的原始定义。参阅“ZigBee协议的Microchip堆栈的界面连接”，里面详细描述的典型的原始流程。参阅ZigBee协议和IEEE802.15.4规范，可以了解更多关于原始定义和他们的参数目录。

图7：ZigBee协议栈体系结构



3.1 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。

3.2 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。

3.2.1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
...
```

3.2.2 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	用途
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();
}
...
```

3.3 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。

3.4 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。

3.5 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。

3.6 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。

四． 演示

4.1 ZENA

ZENA 是Microchip 的zigbee 协议栈配置工具和无线网络分析器。

在zigbee 协议硬功发展上，Microchip 提供的低成本的网络分析器软件被称为ZENA。

ZENA 包括进特殊应用配置的文件和zigbee 协议应用的连接器原程序。ZENA 演示软件作为zigbee 协议Microchip 堆栈的一部分被免费提供，可以在MpZBee 目录中查得。关于ZENA软件的使用更多信息可以参考“ZENA无线网络分析器用户指南”。

ZENA 软件的演示本提供了构造特殊应用的源文件的能力和分析以前所获取的无线网络信道。ZENA 软件的所有特点都可作为一个单独的工具包被获得。包括获取当前无线网络的能力和通过USB 口连接PC 的RF 探测器。

注：当ZENA 软件被用来配置zigbee 协议应用时，他将创建3个文件zigbee.def，myZigBee.c和zLink.lkr。zigbee.def 和myZigBee.c 文件包括堆栈配置的确认信息，zLink.lkr 文件是连接器的源文件。这3个文件相互依赖。

4.2 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“MicrochipSoftware 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README 文件和源代码。

4.3 源文件组成

Microchip 协议栈由多个源文件组成。Zigbee 应用程序公用的多个源文件，被存放在一个单独的目录下，另外一些仅供某些特定的ZigBee 应用程序使用的源文件被存放在另一个目录下。每一个演示应用被存放在自己的目录下。表3给出了目录结构。

表3：目录结构

目录名称	内容
Common	Microchip Zigbee 协议栈的普通源文件，和其它的Microchip应用记录
DemoCoordinator	Zigbee 协议协调器应用范例的源代码，和构造其它Zigbee 协议协调器的模版
DemoRFD	Zigbee 协议RFD 终端应用范例的源代码，和构造其它Zigbee 协议RFD和FFD终端设备应用的模版
DemoRouter	Zigbee 协议路由器应用范例的源代码，和构造其它Zigbee 协议路由器的模版
Documentation	Microchip Zigbee 协议栈文件
TempDemoCoord	Zigbee 协议协调器应用范例（应用PICDEM™ 2 演示板的温度传感器）的源代码
TempDemoRFD	Zigbee 协议RFD 应用范例（应用PICDEM™ 2 演示板的温度传感器）的源代码
ZigBeeStack	Microchip Zigbee 协议栈源文件

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，每一个演示节点应用程序在它们各自的目录

中有各自的zigbee.def （和 myZigBee.c）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。

4.4 演示应用程序

1.0~3.5版的Microchip 协议栈包含三个最初演示应用程序：

- DemoCoordinator——演示典型的ZigBee 协调器的应用程序；
- DemoRFD——演示典型的ZigBee RFD 设备的应用程序；
- DemoRouter——演示典型的ZigBee 路由器的应用程序。

两个二级的演示应用程序包括：

- TempDemoCoord——演示一个ZigBee 协调器，此协调器去获取RFD的温度信息；
- TempDemoRFD——演示一个ZigBee RFD 设备，此设备给协调器提供所需温度信息。

4.5 演示应用功能

演示应用程序可以实现了下列功能：

- 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；
- 用RS—232 终端输出显示设备运行状态；
- 在一个节点上可运行简单远程控制开关和LED 应用程序；
- 可以在编译时进行配置，来演示绑定或者设备搜索（直接或间接消息传输）

将DemoCoordinator 或TempDemoCoord 程序烧入PICDEM Z 演示板中，作ZigBee协调器使用；另一块演示板中烧入相应的RFD程序，作ZigBee终端设备。如果有更多的PICDEM Z 演示板，可以给他们写入终端设备程序或者是路由器程序（DemoRouter）。注意：路由器程序不能作为“开关”和“灯”使用，但它可以扩展网络的物理范围。

4.6 演示应用的项目文件和源文件

下面七个表中列出了Microchip Zigbee 协议栈和演示应用程序实现所必须的源文件。

表4：ZigBeeStack子目录下的Microchip堆栈源文件

文件名	描述
SymbolTime.c, .h	完成定时功能
zAPL.h	堆栈应用层界面头文件。这里包括了所需的应用代码
zAPS.c, .h	Zigbee 协议的APS 层
zHCLighting.h	Zigbee 协议的家居控制，照明规范信息
zigbee.h	普通的Zigbee 协议常量
ZigBeeTasks.c, .h	穿过堆栈各层的程序流程
zMAC.h	普通IEEE 802.15.4 MAC 层的报头文件
zMAC_CC2420.c, .h	针对CC2420 的IEEE 802.15.4 MAC 层程序
zNVM.c, .h	非易失性存储器存储程序
zNWK.c, .h	ZigBee 网络层

zPHY.h	普通IEEE 802.15.4 物理层的头文件
zPHY_CC2420.c, .h	针对CC2420 的IEEE 802.15.4 物理层程序
zZDO.c, .h	ZigBee ZDO(ZDP) 层

表5：Common子目录中Microchip命令源文件

文件名	描述
Compiler.h	编译器特殊定义
Console.c, .h	USART 界面代码
Generic.h	普通常量和典型定义
MSPI.c, .h	SPI 界面代码
sralloc.c, .h	动态存储器分配代码

表6：DemoCoordinator子目录中Zigbee协调器演示应用程序

文件名	描述
Coordinator Template.c	构造ZigBee 协调器应用程序的模版
Coordinator.c	主应用程序源文件
DemoCoordinator.mcp	项目文件。
myZigBee.c	由ZENA™ 产生。包含特殊应用信息。
zigbee.def	由ZENA 产生。包含特殊应用信息。
zLink.lkr	由ZENA 产生。工程链接脚本文件。

表7：DemoRouter子目录中Zigbee协调器演示应用程序

文件名	描述
DemoRouter.mcp	项目文件。
myZigBee.c	由ZENA™ 产生。包含特殊应用信息。
Router Template.c	构造ZigBee 路由器应用程序的模版
Router.c	主应用程序源文件
zigbee.def	由ZENA 产生。包含特殊应用信息。
zLink.lkr	由ZENA 产生。工程链接脚本文件。

表8：DemoRFD子目录中Zigbee终端设备演示应用程序

文件名	描述
DemoRFD.mcp	项目文件。
FFD End Device Template.c	构造ZigBee FFD 终端设备应用程序的模版。
myZigBee.c	由ZENA™ 产生。包含特殊应用信息。
RFD Template with ACKs.c	当RFD 要求APS 层认证时，构造ZigBee 路由器应用程序的模版。
RFD Template.c	当RFD 不要求APS 层认证时，构造ZigBee路由器应用程序的模版。
RFD.c	主应用程序源文件。
zigbee.def	由ZENA 产生。包含特殊应用信息。
zLink.lkr	由ZENA 产生。工程链接脚本文件。

表9：TempDemoCoord子目录中温度（协调器）演示应用程序

文件名	描述
myZigBee.c	由ZENA™ 产生。包含特殊应用信息。
TempDemoCoord.c	主应用程序源文件
TempDemoCoord.mcp	项目文件。

zigbee.def	由ZENA 产生。包含特殊应用信息。
zLink.lkr	由ZENA 产生。工程链接脚本文件。

表10：TempDemoRFD子目录中温度（RFD）演示应用程序

文件名	描述
myZigBee.c	由ZENA™ 产生。包含特殊应用信息。
TempDemoRFD.c	主应用程序源文件
TempDemoRFD.mcp	项目文件。
zigbee.def	由ZENA 产生。包含特殊应用信息。
zLink.lkr	由ZENA 产生。工程链接脚本文件。

4.7 演示应用程序

4.7.1编译演示应用程序

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DemoCoordinator\Demo-Coordinator.mcp。对于演示RFD 的应用程序来说，该项目文件是DemoRFD\DemoRFD.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。
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 、编译器和方案选项。

4.7.2 将第一个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Microchip 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DTD242A 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。选择MPLAB ICD 2编程器菜单中的“Connect”选项去连接MPLAB ICD 2并经行自我检测。
6. 如果您只是像上述那样重新建立了项目文件，那么请您继续执行下一步。如果您希望使用以前编译生成的hex 文件，只要导入DemoCoordinator\DemoCoordinator 文件、DemoRFD\DemoRFD.hex文件或者是DemoRouter\DemoRouter.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DemoCoordinator.hex文件烧写到贴有“COORD...”标签的单片机中； 将DemoRFD.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.演示应用程序文件都包含DTD242A 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。
8. 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。
9. 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。
10. 除去板的电源并断开MPLAB ICD 2 电缆与目标板的连接。

4.7.3 运行演示应用程序

在进行演示之前，首先要确定两个节点均被配置。演示终端设备绑定使用未屏蔽（uncommenting）的#define USE_BINDINGS，而#define USE_BINDINGS已在演示源文件的主文件中被定义。演示设备发现使用已屏蔽(commenting)的#define USE_BINDINGS，而#define USE_BINDINGS已在每一个源文件中被定义。

运行演示程序时，一块DTD242A 演示板作为ZigBee 协调器，另一块DTD242A 演示板作为RFD。为了观察节点的运行情况，我们采用RS—232 线，分别将这两块演示板连接在两台计算机上，并且使用HyperTermina l串口调试软件或者其它的连续串口软件与DTD242A 演示板进行通信，实时对演示版进行观测。运行演示时，端口的配置如下：19200 bps，8 data bytes，1 Stop bit，no parityand和 no flow control。

此时，给协调器节电供电，你将会在HyperTerminal视窗中看到如下显示：

****

西安达泰电子有限责任公司ZigBee开发模板 v1.0
ZigBee 协调器（Coordinator）

协调器开始寻找一个可用的无线信道并建立一个新的网络。如果成功，将会有如下显示：
正在建立网络…
网络 #### 建立成功。

这里####是一个4位的16进制数，表示协调器成功建立网络后的网络PAN ID。此时，协调器允许其他节点加入网络。在视窗中有如下信息显示：

允许加入。

此时，其他节点可以加入此网络。
接着给RFD节点供电，你将会看到HyperTerminal 视窗中有如下显示：

****

西安达泰电子有限责任公司ZigBee开发模板 v1.0
ZigBee节点（ RFD）

RFD开始寻找网络加入，一旦成功，将有如下显示：

尝试作为新节点加入网络…

找到网络.尝试加入 ####.
成功加入网络！

协调器同意新节点加入网络，并显示如下消息：

节点 *#### 已加入*。
.
####表示协调器分配给新节点的短地址，也就是网络地址。

RFD 为了保存能量，通常关闭无线电收发器，使微控制器处于休眠状态。父节点接收到的信息将被存放在父节点的缓冲区内。当 RFD 从休眠状态中醒来，必须向父节点请求接收信息。如果父节点的缓冲区内有给它的信息，就直接传给它。如果没有，RFD 空闲，返回睡眠状态。这一运行过程在 HyperTerminal 视窗中有如下显示：

请求数据…
无数据。

此时，RFD 成功的加入网络并开始监测接收消息。如须更进一步的操作，需要对节点进行其它配置。

4.7.4 演示终端设备绑定

如果两个节点都有未屏蔽 (uncommented) 的 #define USE_BINDINGS 配置，他们就可以演示终端设备绑定。在这种配置下，“开关”节点可以通过绑定发送消息给一个或更多的“灯”节点，这种消息属于间接消息。如需更多了解绑定，请参阅“Message Types and Binding”。

在“开关”节点发送间接消息给“灯”节点之前，首先要建立绑定。DTD242A 演示板上的K3 按键用来发送终端设备绑定请求给ZigBee 协调器。按下协调器演示板上的K3 按键，HyperTerminal 视窗中有如下显示：

正在绑定节点。

于是接着按下RFD DTD242A 演示板上的K3 按键，用来响应绑定请求。如果在5 秒内没有按下，协调器节点的绑定请求超时，被删掉，需要重新进行请求。如果成功响应，RFD 端的HyperTerminal视窗中有如下显示：

发送 END_DEVICE_BIND_req.
节点成功绑定/解绑！

协调器端的HyperTerminal视窗中有如下显示：

节点成功绑定/解绑！

此时绑定成功建立，“开关”K4 可以给“灯”发送间接消息。

4.7.5 演示设备发现

如果两个节点都有屏蔽（commented）过的#define USE_BINDINGS 配置，他们便可以演示设备发现。在这种配置下，“开关”节点可以采用直接消息的消息类型控制“灯”节点，此时，“灯”节点的网络地址是已知的。如需更多了解，请参阅“Message Types and Binding”。

在“开关”发送直接消息给“灯”之前，需要知道“灯”的网络地址。为了演示简单，我们假设“开关”知道“灯”的MAC地址，并且希望同它进行通信。

“开关”节点将通过广播网络地址的请求信息，请求知道“灯”的网络地址。在协调器的 DTD242A 演示板上按下 K3 按键，发送这一广播，HyperTerminal 视窗中有如下显示：

发送 NWK_ADDR_req.

RFD 的底层将会收到这条广播信息，并对它进行响应。这一过程，不需经过 RFD 的应用层。当协调器收到来自 RFD 的响应时，HyperTerminal 视窗中有如下显示：

接收 *NWK_ADDR_rsp.*

此时，“开关”便可以发送消息给“灯”。如果这两个节点已经被配置成“开关”和“灯”，那么按下RFD 上的K3 按键，就可以获得协调器分配给它的网络地址。于是，每个节点都可以发送消息给其它的节点。

4.7.6 演示消息传送

建立绑定或者寻找到目的终端的网络地址后，“开关”节点就可以给“灯”节点发送消息，“灯”节点的终端绑定的发光二极管就将产生“亮/灭”。按下其中一个 DTD242A 演示板上的 K4 按键，作为“开关”控制。HyperTerminal 视窗中有如下显示：

正在发送灯的控制命令。

如果采用直接消息传输，当堆栈接收到一个MAC确认帧，直接报告给应用层，表示发送成功；如果采用间接消息传输，堆栈缓存器内的消息稍后转发，也报告给应用层。当应用层收到报告时，HyperTerminal 视窗中有如下显示：

消息发送成功。

当“灯”节点收到消息，HyperTerminal 视窗中有如下显示：

控制灯光。

“灯”节点绑定 D2 发光二极管，即可以控制此发光二极管 LED 的状态。

五． Microchip ZigBee 协议栈应用

5.1 使用 Microchip ZigBee 协议栈

为了设计一个 ZigBee 协议系统，你必须做到以下几点：

1. 获得一个OUI（参见“IEEE Extended UniqueIdentifiers – EUI-64”）；
2. 在数据传输速率和实际市场需求的基础上，选择合适的无线电收发装置；
3. 选择一个合适的Microchip MCU；
4. 利用堆栈提供的AN965，“Microchip Stack for the ZigBee™ Protocol”，发展ZigBee协议应用；
5. 依次完成所有射频检验的；
6. 依次完成ZigBee协议互用性的证明。

可以按照以下步骤发展 ZigBee 协议应用：

1. 确定这个系统所需的整体框架；
2. 确定每一个设备所需的终端结构；
3. 建立一个新的项目目录，将所有的特殊应用文件和项目文件写入这一目录；
4. 在设备类型、设备配置和终端结构的基础上，使用 ZENA 软件产生配置文件；
5. 从某个演示目录中获得模板文件，使用它最基础的的应用代码。
6. 在模板上添加代码，包括特殊的初始化定义，必要的 ZDO 响应处理，和一些无协议处理和中断处理。

5.2 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11 列出了编译典型ZigBee RFD 应用程序所需的所有源文件，而表12 列出了编译典型ZigBee 协调器应用程序所需的所有源文件。

表11： 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12： 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

5.3 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
```

注 无

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。

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。

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（当前版本不支持）。

前提　必须定义USE_ZMD44101。

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。

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。

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。
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。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　表示当前节点使用一次性电池作为电源。
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。
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。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（当前版本不支持）。
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　无

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。

5.4 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Coordinator Template.c 文件作为参考。如果是RFD 节点，则使用RFD Template with ACKs.c 文件或者RFD Template.c。

在应用程序中，除了标准的API 调用之外，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。下面将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();
}
```

5.5 使用Microchip ZigBee 协议栈的连接接口

应用源代码必须包括头文件：zAPL.h，来访问ZigBee 协议功能。

```
#include “zAPL.h”
```

ZigBee 协调器应用，必须要有一个变量，用来追踪堆栈执行当前原语时的过程。

```
ZIGBEE_PRIMITIVE currentPrimitive;
```

ZigBee 路由器或者终端设备，同样也需要追踪当前原语执行的过程；但是，它还需要另外两个变量来帮助查找网络、加入网络。

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

接着，应用层必须配置所有的引脚功能。ZENA 分析器将创建标志位用来访问LAT 位和TRIS 位。详情请参阅“README”文件。
在堆栈使用前，首先将其初始化，中断必须在那个时候设置为使能：

```
ZigBeeInit();
IPEN = 1;
GIEH = 1;
```

通过ZigBee 协议和IEEE 802.15.4 规范中定义的原语，应用程序此时可以和堆栈连接。堆栈操作由呼叫功能触发，使用ZigBeeTasks() 进行触发。堆栈操作将会一直持续下去，直到被请求的原语路径完成，或者是应用层的原语需要被处理。

由于在一段时间里只有一个原语被处理，那么一个单独的数据结构要用来支持所有的原语参数。这一数据结构在文件ZigBeeTasks.h 中可以找到。访问这一结构时要小心，在使用它之前不能过多的写入参数。处理完一个原语后，它将由当前原语处跳到下一条原语处去执行（或者是NO_PRIMITIVE），以避免进入死循环。如例1所示。关于更多的应用层上的原语的使用，请参考“Primitive Summary”。

例1：应用层基本结构

```
while (1)
{
```

```
CLRWDT();
ZigBeeTasks( &currentPrimitive );
switch (currentPrimitive)
{
// Include cases for each required primitive.
// Be sure to update currentPrimitive!
default:
currentPrimitive = NO_PRIMITIVE;
break;
}
}
```

5.6 建立或加入网络

模板文件中包含了建立网络和加入网络的过程。在NO_PRIMITIVE 原语中，这个过程开始处理。如果这个设备是ZigBee 协调器，并且它还没有建立网络，它将试着去建立一个网络，调用NLME_NETWORK_FORMATION_request 原语完成这一过程。

如果这个设备不是协调器，并且它没有加入网络，它将试着加入一个网络。如果这个设备决定加入它以前加入过的网络，那么它将以孤节点的身份加入，调用NLME_JOIN_request 原语，并且给参数RejoinNetwork 置为TRUE。如果失败，或者是这个设备没有以前曾经加入过的网络，那么它将以一个新节点试着加入网络。首先它调用NLME_NETWORK_DISCOVERY_request 原语，来寻找所有可获得的网络，应用程序将会选择其中的一个网络，去试着加入。此时调用NLME_JOIN_request 原语，并给参数RejoinNetwork 置为FALSE。详情参阅“ZigBee Protocol Timing”。

5.7 接收消息

堆栈通过APSDE_DATA_indication 原语报告接收消息。当这一原语被返回时，这一原语的所有参数和消息信息，以及接收到的消息，一同存放在缓冲区内。调用APLGet() 原语，可以从缓冲区内读取每一字节的消息。

DstEndpoint 参数中存放消息的目的终端，如果这是一有效的目的终端，消息才可以被处理。

例2：接收消息

```
case APSDE_DATA_indication:
{
// Declare variables used by this primitive.
currentPrimitive = NO_PRIMITIVE; // This may change during processing.
frameHeader = APLGet();
switch (params.APSDE_DATA_indication.DstEndpoint)
{
case EP_ZDO:
// Handle all ZDO responses to requests we sent.
break;
// Include cases for all application endpoints.
```

```
}
APLDiscard();
}
break;
```

5.8发送消息

Microchip ZigBee 协议栈允许应用层在一个时刻内发送一条消息。消息发送按以下各步进行：

1. 检查应用层是否已经准备好了发送消息，如果准备好了，ZigBeeReady() 为TRUE。
2. ZigBeeBlockTx() 用来锁住系统。如果被锁，调用ZigBeeReady() 返回FALSE。
3. 消息中的有效部分进入TxBuffer 对列，并使用TxData 进行检索。当装载完成，TxData 必须指向消息后的第一个位置（TxData 等于数据长度）。
4. 装载APSDE_DATA_request 原语参数。
5. 设置currentPrimitive 给APSDE_DATA_request，并且调用 ZigBeeTasks().

消息发送有两个特殊的地方：

1. 在APSDE_DATA_indication 处理中，响应给接收消息；
2. 在NO_PRIMITIVE 处理中，响应给应用事件。

发送消息的过程，在这两个地方被区分。例3给出了发送一直接消息给已知目的设备，并且终端绑定灯。

例3：发送消息

```
if (ZigBeeReady())
{
if (bLightSwitchToggled)
{
bLightSwitchToggled = FALSE;
ZigBeeBlockTx();
TxBuffer[TxData++] = APL_FRAME_TYPE_KVP | 1; // KVP, 1 transaction
TxBuffer[TxData++] = APLGetTransId();
TxBuffer[TxData++] = APL_FRAME_COMMAND_SET |
(APL_FRAME_DATA_TYPE_UINT8<< 4);
TxBuffer[TxData++] = OnOffSRC_OnOff & 0xFF; // Attribute ID LSB
TxBuffer[TxData++] = (OnOffSRC_OnOff >> 8) & 0xFF; // Attribute ID MSB
TxBuffer[TxData++] = LIGHT_TOGGLE;
params.APSDE_DATA_request.DstAddrMode = APS_ADDRESS_16_BIT;
params.APSDE_DATA_request.DstEndpoint = destinationEndpoint;
params.APSDE_DATA_request.DstAddress.ShortAddr = destinationAddress;
params.APSDE_DATA_request.ProfileId.Val = MY_PROFILE_ID;
params.APSDE_DATA_request.RadiusCounter = DEFAULT_RADIUS;
params.APSDE_DATA_request.DiscoverRoute = ROUTE_DISCOVERY_ENABLE;
params.APSDE_DATA_request.TxOptions.Val = 0;
params.APSDE_DATA_request.SrcEndpoint = EP_SWITCH;
params.APSDE_DATA_request.ClusterId = OnOffSRC_CLUSTER;
currentPrimitive = APSDE_DATA_request;
```

```
}
}
```

关于例3：

1. 每一个APS帧必须有一个唯一的处理标识符。这一标识符是调用APLGetTransID()原语从堆栈中获得。
2. TxData必须指向下一个可获得的位置，所以TxBuffer采用负增加地址。
3. 目的终端已知，被存放在destinationEndpoint中。
4. 目的终端的16位的网络地址已知，被存放在destinationAddress。
5. 如果可能的话，我们请求消息路由。

传输消息状态通过APSDE_DATA_confirm原语返回。如果消息传输失败，堆栈将自动重新传送，共有apscMaxFrameRetries次重传机会。

5.9 RFD上的请求接收消息

由于当RFD 处于空闲时保存能量，它通常关闭无线电收发器。当调用NLME_SYNC_request 原语打开无线电收发器，它必须发送请求命令。例4中给出了一个典型的顺序，进入睡眠状态，唤醒状态。我们可以采用看门狗时钟或者是按键唤醒。只有在下列情况下，RFD 才能进入睡眠状态。

1. 没有ZigBee 原语等待被处理；
2. 堆栈没有正在运行的后台作业；
3. 以前的数据请求已经完成；
4. 所有的特殊应用过程已经完成。

在唤醒后，RFD必须调用NLME_SYNC_request 原语，向父节点请求数据。调用NLME_SYNC_request 原语，RFD将收到响应，这条响应可以是下列中的一种：

1. 如果RFD 的父节点的数据缓冲区有给它的消息，就直接传给它。RFD 将产生一条APSDE_DATA_indication 原语。
2. 如果父节点的缓冲区内没有给它的消息，RFD将产生一条NLME_SYNC_confirm 原语和SUCCESS 状态。
3. 如果RFD没有收到来自父节点的响应，将产生一条NLME_SYNC_confirm 原语，和NWK_SYNC_FAILURE 状态。

例4 ： RFD上的请求响应数据

```
// If we don't have to execute a primitive, see if we need to request
// data from our parent, or if we can go to sleep.
if (currentPrimitive == NO_PRIMITIVE)
{
if (!ZigBeeStatus.flags.bits.bDataRequestComplete)
{
// We have not received all data from our parent. If we are
// not waiting for an answer from a data request, send a data
// request.
if (!ZigBeeStatus.flags.bits.bRequestingData)
{
if (ZigBeeReady())
{
```

```
// Our parent still may have data for us.
params.NLME_SYNC_request.Track = FALSE;
currentPrimitive = NLME_SYNC_request;
}
}
}
else
{
if (!ZigBeeStatus.flags.bits.bHasBackgroundTasks &&
myProcessesAreDone())
{
// We do not have a primitive to execute, we've extracted all messages that our parent has for us,
//the stack has no background tasks, and all application-specific processes are complete. Now we
//can go to sleep. Make sure that the UART is finished, turn off the transceiver, and make sure that
//we wakeup from key press.
while (!ConsoleIsPutReady());
APLDisable();
RBIE = 1;
SLEEP();
NOP();
// We just woke up from sleep. Turn on the transceiver and
// request data from our parent.
APLEnable();
params.NLME_SYNC_request.Track = FALSE;
currentPrimitive = NLME_SYNC_request;
}
}
}
```