

ZigBee 软件开发人员指南（基于 Jennic JN5121）

2007.5.25

BOCCN 张宏亮

第一章 基础概念	3
第二章 平台介绍	5
硬件环境	5
软件平台:	7
第三章 快速入门	9
软件安装	9
编译和下载	11
WSN 例程的代码解释	13
修改代码	18
第四章 基于 ZigBee 协议栈进行开发	23
第一节 协议栈架构简介。	23
第二节 ZigBee 协议栈的开发接口 API	28
第三节 应用框架接口函数	36
第四节 ZigBee Device Profile API	42
第五章 基于 802.15.4 协议栈进行开发	49
第一节: IEEE 802.15.4 协议栈的架构, 接口和中断说明	50
第二节: IEEE 802.15.4 网络的建立过程	53
第三节: 应用程序的代码框架	57
第六章 外围部件的操作	68
如何实现定时休眠唤醒	68
如何使用 SPI 接口	69
如何使用 UART	69

第一章 基础概念

为了使我们更好的完成 ZigBee 开发技术的学习，首先我们希望通过概念的介绍在大家的头脑中形成一些统一的概念，这样在我们下面的文档中提到这些概念的时候我们可以不用再进行解释。

ZigBee

“ZigBee”是一个协议的名称，这一协议基于 IEEE 802.15.4 标准，其目的是为了适用于低功耗，无线连接的监测和控制系统。这一协议标准由 ZigBee 联盟维护。

IEEE 802.15.4 标准

IEEE802.15.4 是 ZigBee 协议的底层标准，主要规范了物理层和 MAC 层的协议，其标准由国际电工学协会 IEEE 组织制定并推广。

2.4G 免费频段

免费频段，是指各个国家根据各自的实际情况，并考虑尽可能与世界其他国家规定的一致性，而划分出来的一个频段，专门用于工业，医疗以及科学研究使用的，不需申请就可以免费使用的频段。我们国家的 2.4G 频段就是这样一个频段。

PAN

Personal Area Network 的缩写，用于区别同一 Channel 中，不同的节点群组，只有属于同一个 PAN 的节点之间才能相互通讯。

Channel

通常翻译成通道，ZigBee 所使用的频率范围从 2400MHz 到 2483.5MHz 共 16 个通道，同一个网络的设备必须位于同一个通道中。

MAC 地址 / Extended Address

MAC 地址是网络设备的一个唯一标识码，这一编码具有全球唯一性，由 IEEE 进行管理。

短地址 / Network Address

当 ZigBee 装置加入一个 PAN 中时，会由上一层父节点分配一个 16 位地址，用于网络内节点之间的标识和通讯，以减小包的大小。

Coordinator

ZigBee 网络中的一种网络设备的角色定义，用以控制整个 PAN，每一个 PAN 都必

须有一个 Coordinator

Router

ZigBee 网络中的一种网络设备的角色定义，用以转发数据，延伸 ZigBee 网络的规模。

End-Device

ZigBee 网络中的一种网络设备的角色定义，作为网络的最终端节点。

第二章 平台介绍

硬件环境

JN5121 芯片

JN5121 是一款兼容 IEEE 802.15.4 的低功耗 SoC 芯片。虽然在笔者撰写这个培训手册的时候已经有消息 Jennic 将要推出更加经济和性能更高的 5139 芯片，但是考虑到产品的稳定性和 5139 与 5121 全面兼容的特性，在本教材中我们仍然以 5121 作为硬件基础。

如果您需要了解 JN5121 芯片的管脚和更多的参数细节，您可以参考 JN5121 的 [DataSheet JN-DS-JN5121](#)。这个文档可以从 Jennic 的网站或者我们公司的 [ftp\(ftp://bocon.com.cn/jennic\)](ftp://bocon.com.cn/jennic) 上下载到。

JN5121-Z01-M 模块

JN5121-Z01 模块是基于 JN5121 芯片所开发的一一系列表帖形式的模块产品。该系列模块集成了所有的射频组件和无线微控制器。采用模块进行开发可以大大的减少开发人员的工作量，缩短产品的开发周期。

这一系列的模块包含下列不同的型号

- JN5121-Z01-M00 内置陶瓷天线
- JN5121-Z01-M01 带有天线连接接口
- JN5121-Z01-M02 带有功率放大器和天线连接接口

如果您需要了解关于模块产品的更多细节，可以参考模块产品的 DataSheet [JN-DS-JN5121MO](#)。

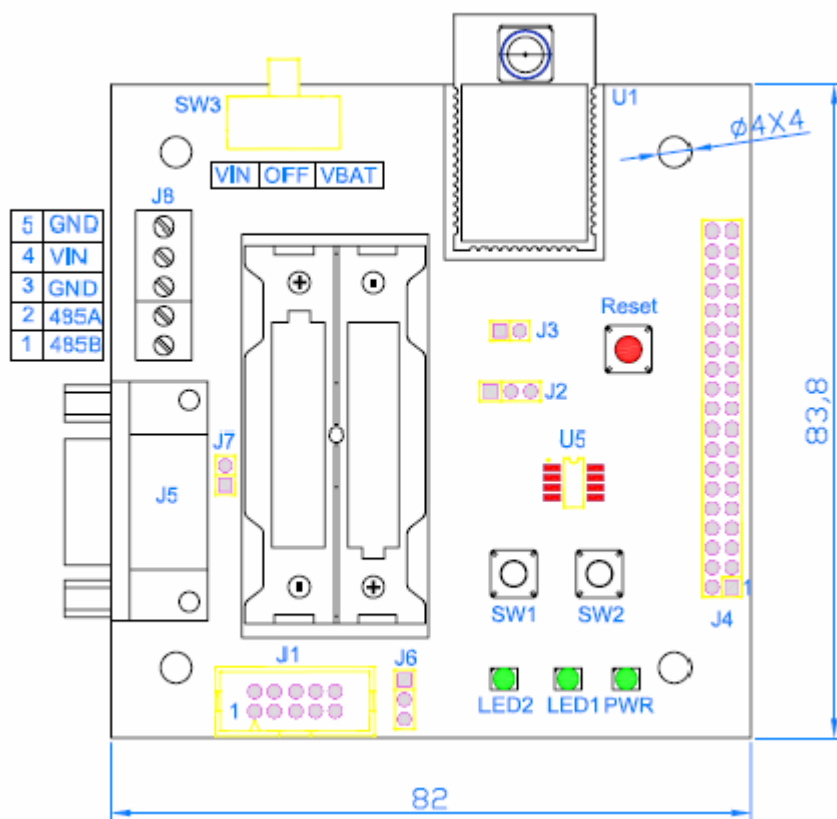
开发包

基于 Jennic 产品的 ZigBee 开发包有很多种型号，包括 Jennic 原产的 EK 系列，和 BOCCN 生产的 DK 系列，您还可以从其他的供货商查到更多的开发包型号，不过

在本手册中我们选择 BOCCN 的原产开发包系列中的 JN5121-DK103 作为学习的硬件基础。如图所示：

DK103 包含三个传感器板，软件和文档光盘以及两根串口连接线和电池。该款开发包的售价是 1990 元整，非常适合用来开发基于 802.15.4 或者 ZigBee 的应用。

传感器板



开发包中的主要部件就是传感器板，之所以叫做传感器板是因为每一个板子上都集成了相应的温湿度一体传感器，这样用户就可以直接利用传感器板进行一些应用的测试和开发。

传感器板包含的部件如下所介绍：

- JN5121 模块
- 板载温湿度传感器
- RS232 接口：用于编程或者连接其他的串口设备
- JN5121 的 I/O 扩展端口
- RS485
- J1 CMOS Program

- J2 Prog Run
- J3 Flash
- J4 I/O 脚
- J5 RS232/Program Port
- J6 J1 和 J5 编程/串口选择
- J7 RS485 端电阻
- J8 如图标注
- SW1, SW2 可编程按键
- SW3 开关(可选择外供电或电池)
- PWR 电源指示灯
- LED1, LED2 可编程 LED

在本手册中我们就将使用这一款传感器板进行各种应用的开发和测试。

软件平台:

需要说明的是,在笔者撰写这个文档的时候, **Jennic** 正在为改进他的软件开发平台做着巨大的努力,所以以下所提到的各种软件都频繁的推出新的版本,但是为了不至于对我们的读者造成混乱,我将所有的软件版本锁定到一个稍老的版本号上,如果有特殊的需要我将在文档的部分特别的说明。如果没有特别的说明本手册的所有范例代码都以此处提到的软件版本为准:

Cygwin 2v0

这个软件是一个 **Linux** 的模拟环境,之所以要安装这个程序是因为最初 **Jennic** 的开发环境是基于 **Linux Gcc** 编译器的。

Codeblocks 1v0

这个软件是 **Jennic** 所提供的代码编辑和编译环境,这个软件和上面提到的 **cygwin** 的 **gcc** 编译器进行连接完成代码的编译工作。**Codeblocks** 是一款开源的 **C/C++** 开发工具, **Jennic** 基于这个工具对其进行扩展形成了自己的开发平台,所以您必须使用从我们网站或者 **Jennic** 网站下载 **codeblocks** 才能进行开发。

Flash programmer 1.4

这个程序是用来将编译好的代码下载到传感器板中的工具。**1.4** 版本是相对稳定的

版本。

mac stack 2.6.1

这个安装包是 **mac** 协议栈的安装包。请注意我们所选用的协议栈的版本号。

zigbee stack 1.5.1

这个安装包是 **zigbee** 协议栈的安装包。请同样安装和我们一样的版本。

第三章 快速入门

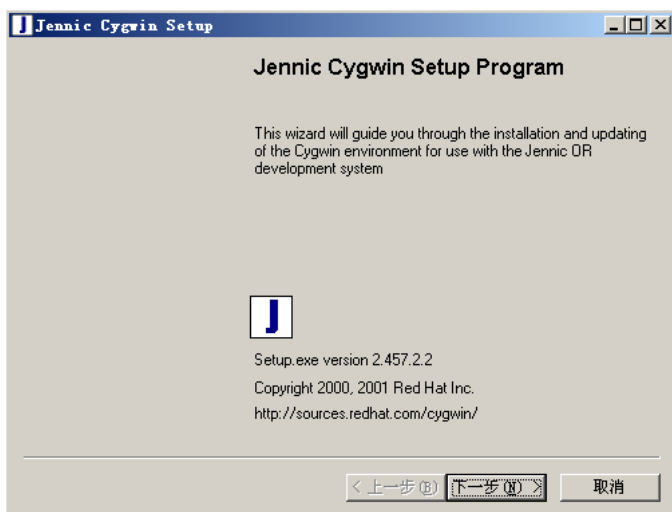
现在我们有了一些共同的概念认识和共同的软硬件平台，那么接下来我们就可以开始一点儿修改代码的工作，让东西转起来。

软件安装

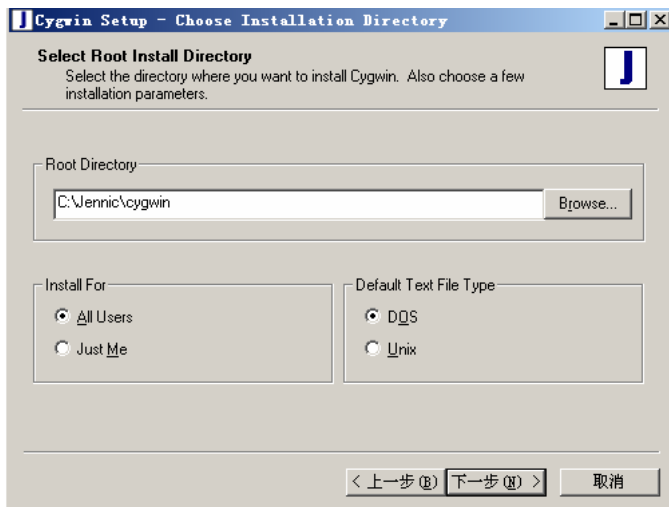
首先我们需要把所有的软件安装好。

先从 **cygwin** 开始

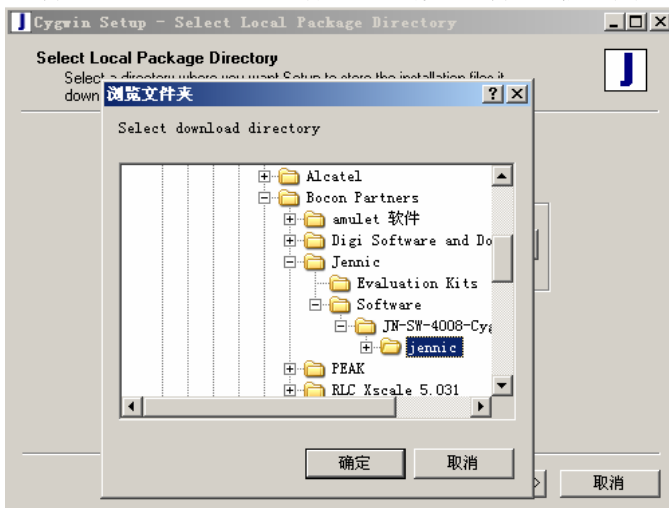
运行 **Setup.exe**



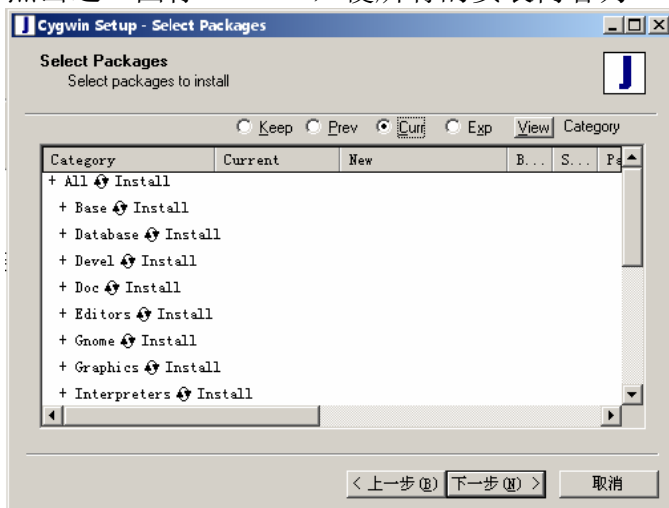
选择安装目录



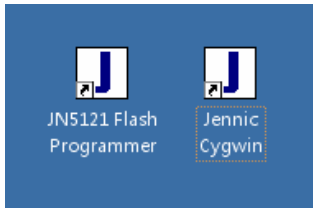
选择源目录：注意，请点击浏览选择压缩包解压目录下的 Jennic 目录



点击这一图标 All , 使所有的安装内容为 install 状态如图



点击下一步完成软件的安装



安装完成后，桌面将出现上图所示的图标，其中 **JN5121 Flash Programmer** 为程序下载程序，详细的使用方法请参考文档光盘中[用户手册] [JN-UG-3007.Flash Loader](#)

接下来是 codeblocks

运行 **JN-SW-4024-JennicCodeblocks-1v0.msi**

然后按照安装指示完成软件的安装，这个过程很简单，一路 **next** 基本就可以了。

接下来升级你的 **Jennic Flash Programmer** 到 **1.4.0** 版本，老版本的 **Flash Programmer** 有一些问题，会把模块的 **MAC** 址刷掉。

如果开发包的光盘中没有，请从我们的 **FTP** 上下载 **/Jennic/Software/JN-SW-4007-FlashLoader-1v4.msi**

运行后一路 **NEXT** 完成安装。

然后是协议栈

JN-SW-4011-ZigbeeStack-1v5v1.msi

JN-SW-4013-MAC-Setup-2v6v1.msi

这两个都需要安装我们指定的版本。新版本的协议栈可能和本手册的部分范例有冲突。

另外需要说明的是，最好将上面的软件都安装到默认的 **C** 盘目录，尽量不要修改。虽然这不会造成什么大的问题，但是如果您安装在默认的目录，在接下来的教程中凡是提到目录的地方您就都可以按图索骥了。

编译和下载

软件安装完成后，我们就可以开始调试一个例程了。

现在我们从一个很有意思的例子入手，无线传感器网络例程，也叫做 **WSN** 例程。这个范例的源代码可以从开发包的光盘 **Application Notes** 目录下，或者我们的 **FTP** 相应的目录下找到，

JN-AP-1015-Zigbee-Wireless-Sensor-Network-1v4

下载后是一个压缩文件。将其解压到 **C:\Jennic\cygwin\jennic\developer** 目录下

解压后的 **JN-AP-1015-Zigbee-Wireless-Sensor-Network** 目录下有下面这

样两个文件

JN5121_WSN_Coordinator.cbp

JN5121_WSN_Router.cbp

JN5121_WSN_Coordinator.cb 是 Coordinator 的工程文件，

JN5121_WSN_Router.cbp 是 Router 的工程文件。

用 CodeBlocks 打开这两个文件。然后在每一个工程名上点击右键，选择 **Build Options** 命令

确认工程的编译器选择为 **JN5121 Compiler**。

分别编译两个工程。编译完成后您将在工程目录下找到一个 **JN5121_Build** 目录，您将在这个目录下找到 **WSN_Coordinator.bin** 和 **WSN_Router.bin** 这两个文件。

这就是这个例程的二进制程序文件。

在把程序文件写入传感器板之前，让我们介绍一下 **WSN** 例程的大致应用。

这个范例演示了多个传感器板将自己的传感器数据通过 **Mesh** 网络传输到 **Coordinator** 节点，然后 **Coordinator** 节点通过串口将数据上传到 PC 的过程。

在这个应用中我们有两种角色的设备，分别使用不同的程序来实现。

一种是 **Router**，这种角色的设备负责采集传感器的数据和电池的电压，并将数据周期性的发送到 **Coordinator** 节点，而且 **Router** 设备还将自动的转发网络中其他节点的数据。

另一种是 **Coordinator** 节点，这种角色的设备负责接收 **Router** 上传的数据，并将所得到的数据通过串口上传到 PC。

下面我们就通过 **Jennic Flash Programmer** 程序将两个 **bin** 文件分别写入到不同的传感器板中。

硬件连接：首先将开发包中的串口连接线取出，连接到 PC 的串口，然后将另一端连接到传感器板的编程口，将传感器板的 **J6** 跳线跳到 **PROG** 端 (**J6** 跳线在电池盒和板子底端之间)。安装电池和天线。

启动 **Jennic Flash Programmer**，然后给板子上电，点击 **Read Addr** 按钮。如果能够看到正常的 **MAC** 地址被读取上来，可以确认所有的连接和供电都是正常的。

点击 **Browse** 按钮，选择刚才编译好的 **WSN_Router.bin** 文件。点击 **Program** 按钮把程序写入板子。断电，然后把 **J6** 跳到 **NORM** 端。

重复上述步骤将另外一块 **Router** 也写好。

将最后一块板子写成 **WSN_Coordinator.bin** 文件。记住每个板子的角色分配。

现在您手头的三块板子应该是两块 **Router**，一块 **Coordinator**。并且

Coordinator 已经连接到了 PC 上。

打开一个串口调试工具。这里我们选择 Windows 默认的超级终端，设置一个串口连接，每秒位数 115200，数据位 8，奇偶校验 无，停止位 1，数据流控制 无。

按顺序打开 **Coordinator, Router**。可以看到两个不同地址的 **Router** 周期性的上报自己的温湿度传感器数据和电池电压。关闭一个 **Router**，可以看到上报的数据中就少了一组。

经过上述的步骤您就已经完成了一个程序从编译到下载调试的全过程。

现在让我们更深入一点，看看如何修改这个程序让他变得更意思一点。在修改之前让我们从整体了解一下程序的各个函数的用途。

WSN 例程的代码解释

让我们先从 **Router** 的代码开始。如果我将 **Router** 的代码全部的帖到这里会有 15 页之长，所以我只打算简单介绍每个函数的用途。

PUBLIC void AppColdStart

这个函数是整个程序的入口，Jennic 的开发程序虽然使用标准的 C 语言进行开发，但是它的程序构建在 **Application Support Layer** 基础上，所以没有我们熟悉的 **main** 函数入口，**AppColdStart** 就是我们的起点了。对于这个范例程序，我们在 **AppColdStart** 里面设置了基本的网络参数并且调用了一个初始化函数。

PUBLIC void AppWarmStart

这个函数是程序热启动的入口，系统在休眠后重新启动的时候会自动的调用这个函数。在这个应用中我们简单的调用了 **AppColdStart**。

PRIVATE void vInit

一个私有函数，被前面的 **AppColdStart** 函数调用，用来完成一系列的初始化工作。包括初始化系统，初始化指示灯，初始化传感器，最后启动了 **BOS** 一个小型的任务系统，然后我们的程序就在这个小型系统的调度下开始工作，进入不同的事件处理函数。

PRIVATE void vInitSensors

这个函数初始化了传感器，这里有调用了一系列的 AD 系统函数用来读取电池电压数据，如果您以后在您的应用中需要操作 AD 的话，别忘了参考这个函数的代码。

PRIVATE void vAppTick

这个函数是整个程序的核心内容了。他完成了读取传感器数据，控制状态显示灯闪烁，发送数据等的功能。这个函数是一个时钟周期性调用函数，在函数的最后他又创建一个时钟，以便固定的周期后仍然调用自己。

PRIVATE void vReadBatteryVoltage

这个函数用来读取电池的电压值，他展示了一个完整的 AD 采样的过程，如果您在以后自己的应用中需要用到 AD 采样，这段代码是非常好的参考。

PRIVATE void vReadTempHumidity

这个函数完成了温湿度传感器的数据的读取过程。

PRIVATE void vSendData

这个函数是数据的发送过程，他构建了相应的数据包并向协议栈发送了相应的数据发送请求将所有的传感器数据发送到 **Coordinator**。

接下来是一系列的协议栈的回调函数，这一系列的函数利用回调的机制接收协议栈的各种事件通知并执行相应的函数过程，您会注意到并不是每一个函数都有可以执行的动作，很多函数都是空的函数体，但是作为基于 **Jennic ZigBee** 协议栈的应用我们必须在代码中留出这些函数的入口。

回调函数：

简单说就是，由程序员编写的，给协议栈调用的函数。也就是说，函数的功能由你编写，由协议栈来决定调用的时机，通常是发生了相应的事件。你所要做的只是把函数的参数按照规定格式写好，编写好函数的内容。

PUBLIC void JZA_vZdpResponse

收到 ZDP 回应的时候协议栈调用的函数

PUBLIC uint8 JZA_u8AfMsgObject

收到 MsgObject 调用的函数。

PUBLIC void JZA_vAfKvpResponse

收到 Kvp 回应的时候调用的函数

PUBLIC AF_ERROR_CODE JZA_eAfKvpObject

收到 KvpObject 的时候调用的函数

PUBLIC void JZA_vAppDefineTasks

协议栈要求用户定义自定义任务的时候调用的函数

PUBLIC void JZA_vPeripheralEvent

发生外围部件事件的时候调用的函数

PUBLIC void JZA_vAppEventHandler

周期性调用函数，协议栈会周期性的调用这个函数，这里我们定义了一个时钟。

PUBLIC bool_t JZA_boAppStart

系统初始化完成后会调用这个函数，通常我们在这个函数中进行设备描述。

PUBLIC void JZA_vStackEvent

协议栈发生网络事件的时候调用这个函数，在这个例程中我们在这个函数中判断是否已经加入了网络。

Router 的代码我们以已经走马观花的浏览了一遍，其实他实现的功能并不复杂，除去 **ZigBee** 应用程序的框架代码外，它主要就是定义了一个时钟函数然后周期性的读取各种传感器数据并向 **Coordinator** 发送数据。现在我们暂且把所有的细节都忽略掉，您只需要掌握一个整体的框架就可以了。

下面我们再来看一下 **Coordinator** 的代码，同样我们还是整体的粗略浏览一遍，这回我们换个方式来分析代码，刚才通过亲手的下载运行代码我们大致可以猜到 **Coordinator** 的工作方式。一定是通过某个函数将各 **Router** 的数据读取上来然后向串口发送数据。那么我们就主要看这两个功能的代码。接收 **Router** 的数据和向串口发送数据。

```
PUBLIC uint8 JZA_u8AfMsgObject(AF_ADDRTYPE sAfSrcAddr,
                                uint8      u8ClusterID,
                                uint8      u8DstEndPoint,
                                uint8      u8LQI,
                                uint8      *pau8AfduInd,
                                uint8      *pu8ClusterIDRsp,
                                uint8      *pau8AfduRsp)
{
    uint16 u16BattVoltage;
    uint16 u16Temperature;
    uint16 u16Humidity;

    if ((sAfSrcAddr.hAddrMode == DEV_16BIT_ADDR) &&
        (u8DstEndPoint == WSN_DATA_SINK_ENDPOINT))
    {
        if(u8ClusterID == WSN_CID_SENSOR_READINGS)
        {
            u16BattVoltage = *(pau8AfduInd + 1);
            u16BattVoltage = u16BattVoltage << 8;
            u16BattVoltage |= *pau8AfduInd;

            u16Temperature = *(pau8AfduInd + 3);
```

```
        u16Temperature = u16Temperature << 8;
        u16Temperature |= *(pau8AfdInd + 2);

        u16Humidity = *(pau8AfdInd + 5);
        u16Humidity = u16Humidity << 8;
        u16Humidity |= *(pau8AfdInd + 4);

        vTxSerialDataFrame(sAfSrcAddr.u16Address,
                           u16Humidity,
                           u16Temperature,
                           u16BattVoltage);
    }
}
return 0;
}
```

这个回调函数负责接收 **Router** 发送来的数据, 他通过解析 **pau8AfdInd** 携带的数据内容将每个传感器的数据读取出来。然后调用了 **vTxSerialDataFrame** 函数将数据写到串口。

```
PRIVATE void vTxSerialDataFrame(uint16 u16NodeId,
                                uint16 u16Humidity,
                                uint16 u16Temperature,
                                uint16 u16BattVoltage)
{
    printf("\n\r\n\rAddress = %x", u16NodeId);
    printf("\n\rHumidity = %d", u16Humidity);
    printf("\n\rTemperature = %d", u16Temperature);
    printf("\n\rVoltage = %d", u16BattVoltage);
}
```

这个函数将 **Router** 的地址和相应的传感器数据写到串口, 您大概已经注意到我们使用了一个熟悉的函数 **printf**, 但是这个可不是我们在 C 语言中所使用的标准输出函数, 这个函数在 **Coordinator** 工程中的另外一个 **printf.c** 文件中进行了定义。您可以查看我们 **Coordinator** 工程的另外其他几个文件 **serial.c**, **serialq.c**, **uart.c**, **printf.c**, 这些都是已经编写好的串口操作的功能文件, 您也可以在自己的工程中使用他们。

Coordinator 的其他代码和 **Router** 的代码大同小异。不过还是有一些地方需要提示。

PUBLIC bool_t JZA_boAppStart(void) 这个函数定义了 **Coordinator** 的

Simple Descriptor, 这个概念我们将在后面讨论, 现在您可以认为这个过程定义了设备的一个端口的描述。

PRIVATE void vToggleLed(void *pvMsg, uint8 u8Dummy) 这个函数被自己定义的时钟周期性的调用。用来闪灯来表示设备的正在运行。

好了, 现在不管您是一头雾水还是了然于胸, 我想您都已经掌握了足够的信息来修改这个范例代码。 下一个小节我们就将修改一下让这个小的范例按照我们的要求工作。

修改代码

下面为了让这次修改对您以后的开发工作有点帮助, 我们希望能够给出一个有点意义的演示。这次我们打算为传感器节点引入一个读取串口数据的功能, 然后将读到的串口数据通过 **ZigBee** 网络发向 **Coordinator**。基于这个范例以后您就可以连接各种串口设备采集数据了。为了将代码变得不太复杂, 我们只修改 **Router** 的代码, 而且我们不引入串口数据的 **buffer** 数据结构, 只在程序中用一个 **Char** 类型的变量保存串口读到的最后一个字符。在这个基础上您完全可以添加更加强大的功能。

下面是我们添加或者修改的内容。

1. 首先我们添加两个串口初始化的函数。

```
PRIVATE void vUART_Init(void)
{
    /* Enable UART 0 */
    vAHI_UartEnable(UART);

    vAHI_UartReset(UART, TRUE, TRUE);
    vAHI_UartReset(UART, FALSE, FALSE);

    /* Set the clock divisor register to give required buad, this has to be done
       directly as the normal routines (in ROM) do not support all baud rates */
    vUART_SetBuadRate(UART_BAUD_RATE);

    vAHI_UartSetControl(UART, FALSE, FALSE, E_AHI_UART_WORD_LEN_8, TRUE, FALSE);
    vAHI_UartSetInterrupt(UART, FALSE, FALSE, TRUE, TRUE, E_AHI_UART_FIFO_LEVEL_1);
}
```

```
}
```

```
PRIVATE void vUART_SetBuadRate(uint32 u32BaudRate)
{
    uint8 *pu8Reg;
    uint8  u8TempLcr;
    uint16 u16Divisor;
    uint32 u32Remainder;

    /* Put UART into clock divisor setting mode */
    pu8Reg = (uint8*)(UART_START_ADR + UART_LCR_OFFSET);
    u8TempLcr = *pu8Reg;
    *pu8Reg = u8TempLcr | 0x80;

    /* Write to divisor registers:
       Divisor register = 16MHz / (16 x baud rate) */
    u16Divisor = (uint16)(16000000UL / (16UL * u32BaudRate));

    /* Correct for rounding errors */
    u32Remainder = (uint32)(16000000UL % (16UL * u32BaudRate));

    if (u32Remainder >= ((16UL * u32BaudRate) / 2))
    {
        u16Divisor += 1;
    }

    pu8Reg = (uint8 *)UART_START_ADR;
    *pu8Reg = (uint8)(u16Divisor & 0xFF);
    pu8Reg = (uint8*)(UART_START_ADR + UART_DLM_OFFSET);
    *pu8Reg = (uint8)(u16Divisor >> 8);

    /* Put back into normal mode */
    pu8Reg = (uint8*)(UART_START_ADR + UART_LCR_OFFSET);
    u8TempLcr = *pu8Reg;
    *pu8Reg = u8TempLcr & 0x7F;
}
```

将这两个函数添加到 Router 代码中，然后在代码的前部 Local Function Prototypes 的部分添加下面的函数签名

```
PRIVATE void vUART_Init(void);
```

```
PRIVATE void vUART_SetBuadRate(uint32 u32BaudRate);
```

2. 这两个函数中用到了很多常量参数，把下面的代码添加到 **Macro Definitions** 部分

```
#define UART                E_AHI_UART_0
#define UART_BAUD_RATE      19200

#if UART == E_AHI_UART_0
    #define UART_START_ADR   0x30000000UL
#else
    #define UART_START_ADR   0x40000000UL
#endif

#define UART_LCR_OFFSET     0x0C
#define UART_DLM_OFFSET     0x04
```

3. 接下来在 **Local Variables** 部分定义一个字符类型的数据用来保存串口得到的最后一个字符数据

```
PRIVATE char cCharIn=0;
```

4. 在 **JZA_vPeripheralEvent** 回调函数中添加串口事件的处理过程，把串口数据保存到 **cCharIn** 变量中

```
PUBLIC void JZA_vPeripheralEvent(uint32 u32Device, uint32 u32ItemBitmap)
{

    if (u32Device == E_AHI_DEVICE_UART0)
    {
        /* If data has been received */
```

```
        if ((u32ItemBitmap & 0x000000FF) == E_AHI_UART_INT_RXDATA)
        {
            cCharIn = ((u32ItemBitmap & 0x0000FF00) >> 8);
        }
    }
}
```

5. 在 `vInit` 过程中添加下面的函数调用，用来初始化串口。把这句代码添加到 `(void)bBosRun(TRUE);` 前面

```
/* set uart */
vUART_Init();
```

6. 把 `vSendData` 过程中的下面两句

```
au8Afdu[0] = sBattSensor.u16Reading;
au8Afdu[1] = sBattSensor.u16Reading >> 8;
```

改为

```
au8Afdu[0] = cCharIn;
au8Afdu[1] = 0;
```

这样我们发送的就不是电池的电压而是串口接收到的字符数据了。

好了现在重新 `build Router` 工程，我们将得到新的 `WSN_Router.bin` 文件。现在把这个文件重新写到传感器板子中。还记得步骤吗？如果您有点忘记了让我在重复一遍大致的过程

- 连接串口线，
- J6 跳到 PROG，
- 打开 Flash Program，
- 板子上电，
- Read Addr ，
- 选择 bin 文件，Programm。
- 接下来稍等片刻。

下面我们需要把两根串口线都连接好。一个连接 **coordinator** 一个连接我们刚刚写好的 **Router**。用超级终端建立两个连接，注意速率的选择 **coordinator** 是 **115200**, **Router** 是 **19200**。然后打开电源。现在试着用 **Router** 发送一个字符。**Coordinator** 端应该看到电压值地方显示的应该是发送过来的字符的 **ASCII** 码。

很好。我们已经做了很多工作，现在您应该对 **Jennic** 的开发有了大致的概念。现在我们需要更多的细节来满足实际的应用需要。

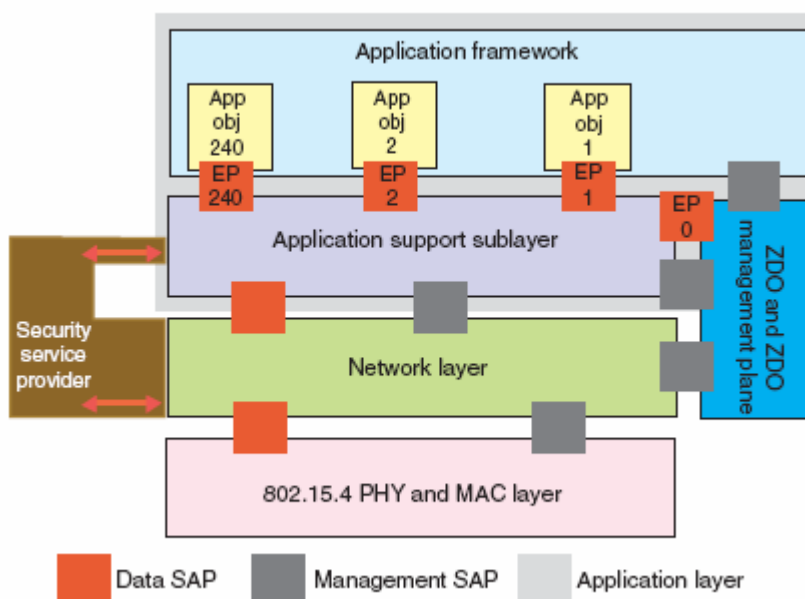
在接下来的几个章节中我们将逐一向您展示 **ZigBee** 开发的各个部分。他们分别是

- 第四章 基于 **ZigBee** 协议栈进行开发
- 第五章 基于 **802.15.4** 协议栈进行开发
- 第六章 外围部件的操作

我们将以小的专题的形式展开这些章节，您既可以按照章节的顺序阅读也可以在日常的工作中跳到相应地小结进行查阅。

第四章 基于 ZigBee 协议栈进行开发

第一节 协议栈架构简介。



对于深入的 ZigBee 应用开发，我们需要花一点时间来详细的了解我们所使用的工具---ZigBee 协议栈，建立基本的协议栈工作原理的概念对于我们后续的工作有非常大的帮助。

首先，ZigBee 标准定义了一种网络协议，这种协议能够确保无线设备在低成本、低功耗和低数据速率网络中的互操作性。ZigBee 协议栈构建在 IEEE 802.15.4 标准基础之上，802.15.4 标准定义了 MAC 和 PHY 层的协议标准。

MAC 和 PHY 层定义了射频以及相邻的网络设备之间的通讯标准。而 ZigBee 协议栈则定义了网络层，应用层和安全服务层的标准。上面的示意图表示了 ZigBee 协议的层次架构。越向下越贴近硬件越向上越贴近软件本身和应用。

下面我们会涉及到一系列的新的概念。

首先是**Profile**，如果需要译成中文的话我习惯叫它规约，但是在这个手册里面我还是保留它的英文名称不再翻译，这样可以和Jennic的手册保持一致。每一个 ZigBee 的网络设备都应该使用一个**Profile**，**Profile**定义了设备的应用场景，

比如是家庭自动化(HC)或者是无限传感器网络(WSN),另外定义了设备的类型还有设备之间的信息交换规范。**Profile**分为两种,一种是公共的**Profile**,这种**Profile**通常由某个组织发布,用于实现不同厂商生产的**ZigBee**设备之间可以互相的通讯使用。私有的**Profile**通常只是在公司内部或者项目的内部的一个默认的标准。

App Obj 这个概念的全名叫**Application objects**, 这个概念目前是一个纯概念范畴的东西,在**Jennic**的开发中我们并看不到这个概念的具体表现, 目前我们可以理解为凡是和一个应用的相关的操作和数据都可以算属于这个应用的**Application Object**.

每一个**App Obj**连接一个**End Point**, **End Point**类似于端口号的概念,他是一个数据交换的接口,在**Jennic**开发中它表现为一个整形的数值,设备之间的通讯实际上表现为**end point**和**end point**之间的数据交换。数据在通过协议栈请求发送的时候都需要指定发往哪个**end point**.

Cluster, 通常我们翻译成“簇”。它定义了**endpoint**和**endpoint**之间的数据交换格式。**Cluster**包含一系列有着逻辑含义的属性。通常**profile**都会定义自己的一系列**cluster**。每一个**endpoint**上都会定义自己发送和接收的**cluster**.

另外需要说明的是两个特殊的**endpoint**定义。**Endpoint 0** 用于配置和管理整个**ZigBee**设备,通过这个**endpoint**,应用可以和**ZigBee**协议栈的其他层进行通讯,进行相关的初始化和配置工作。和这个**endpoint**接口的是**ZigBee Device Object (ZDO)**。 另外一个特殊的**endpoint**是 255, 这个**endpoint**用来向所有的**endpoint**进行广播。 241-254 是保留的**endpoint**,用户在自己的应用中不能使用。

Application support sublayer提供了数据安全和绑定的功能,绑定(**Binding**)就是将不同的但是兼容的设备进行匹配的一种能力,比如开关和灯。

Network layer完成了大部分的网络功能,包括设备之间的通讯,设备的初始化,数据的路由等等。

上面的图中还提到了**SAP**的概念, **SAP**就是**Service Access Point**,如果要翻译成中文的话我们习惯叫作服务访问接口,也就是数据或者管理的接口。不同层之间通过这些接口进行数据的交换和管理。这又是一个纯概念上的含义,没有具体的表现形式和固定的实现形式。每两个层之间都有自己的**SAP**的实现方法。

网络层 (**Network layer**) 概念

正如我们前面所说的网络层完成了 **ZigBee** 网络的大部分功能,包括网络的建立,拓扑,数据的通讯,路由等等。我们的应用程序就是构建在这个层次之上的。**ZigBee** 协议栈的主要工作内容就是实现网络层的各种功能,并保证其标准性和兼容性。

ZigBee 的节点

ZigBee 标准规定可以在一个单一的网络中容纳 65535 个节点，所有的 ZigBee 网络节点都属于以下三种类型中的一种。

- Co-ordinator
- Router
- End Device

讲到这里就需要对一些用户常问的问题进行解释了，通常用户都会过于关注于节点的类型，但实际上以上所说的三种节点类型都是网络层的概念，他们决定了您的网络的拓扑形式，而通常来说 ZigBee 网络采用任何一种拓扑形式只是为了实现网络中信息高效稳定的传输，用户在实际的应用中是不必关心 ZigBee 网络的组织形式的。节点类型的定义和节点在应用中所起到的功能并不相关，比如说一个 ZigBee 网络节点不论他是 Co-ordinator、Router 还是 End-Device 它都可以运行相应的程序测量传感器的温度和湿度。

下面对这些节点的类型进行粗略的解释

Co-ordinator

不论 ZigBee 网络采用何种拓扑方式，网络中都需要有一个并且只能有一个 Co-ordinator 节点。

在网络层上，Co-ordinator 通常只在系统初始化的时候起到重要的作用。在一些应用中网络初始化完成后，即便是关闭了 Co-ordinator 节点，网络仍然可以正常的工作。但是如果 Co-ordinator 还负责提供路由路径，比如说在星形网络的拓扑结构中(我们将很快提到这种拓扑结构)，Co-ordinator 就不能被关闭，而必须持续的处于工作状态。同样如果 Co-ordinator 在应用层提供一些服务，比如 Co-ordinator binding.Co-ordinator 也必须持续的处于工作状态。

Co-ordinator 在网络层的任务是：

- 选择网络所使用的频率通道，通常应该是最安静的频率通道。
- 开始网络
- 将其他节点加入网络
- Co-ordinator 通常还会提供信息路由，安全管理和其他的服务。

Router

如果 ZigBee 网络采用了树形和星形拓扑结构就需要用到 Router 这种类型的节点。(关于这两种拓扑结构我们马上就会提到)

Router 类型节点的主要功能就是：

- 在节点之间转发信息
- 容许子节点通过他加入网络
- 需要注意的是通常 Router 节点不能够休眠。

End Device

End Device 节点的主要任务就是发送和接收信息。通常一个 End Device 节点

是电池供电的，并且当它不在数据收发状态的时候它通常都是处于休眠状态以节省电能。

需要说明的是 **End Device** 节点不能够转发信息也不能够让其他人加入网络。

网络拓扑形式

ZigBee 网络可以实现下面三种网络拓扑形式

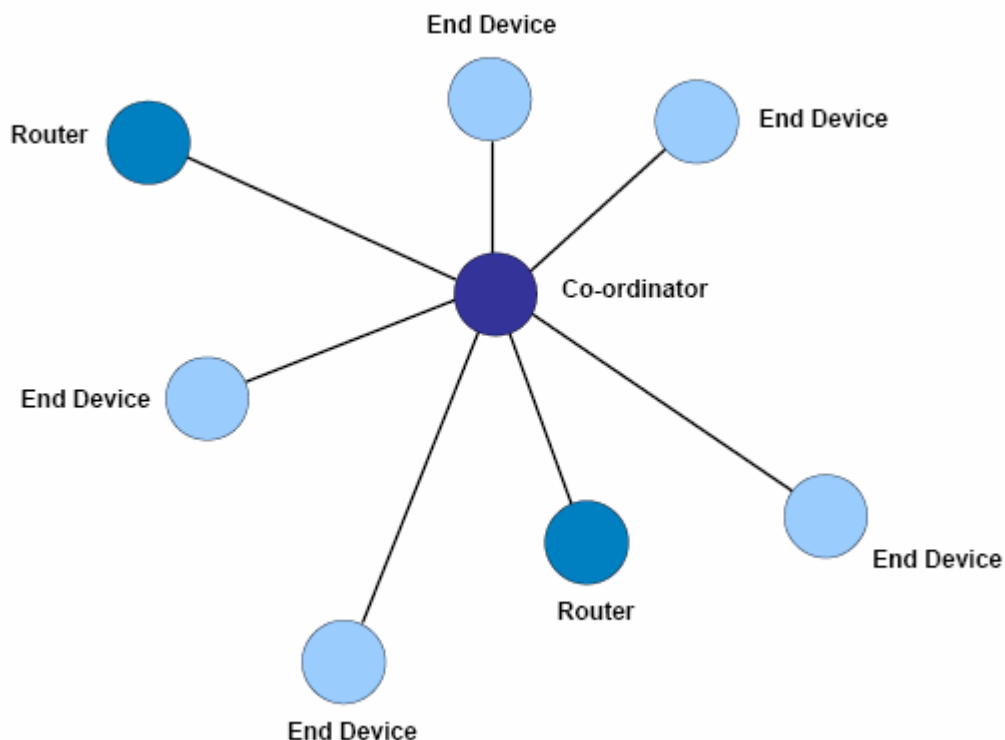
- 星形
- 树形
- 网状

星形拓扑

这种拓扑形式是最简单的一种拓扑形式

星形拓扑包含一个 **Co-ordinator** 节点和一系列的 **End Device** 节点。每一个 **End Device** 节点只能和 **Co-ordinator** 节点进行通讯。如果需要在两个 **End Device** 节点之间进行通讯必须通过 **Co-ordinator** 节点进行信息的转发。

下图表示了星形拓扑结构的示意图

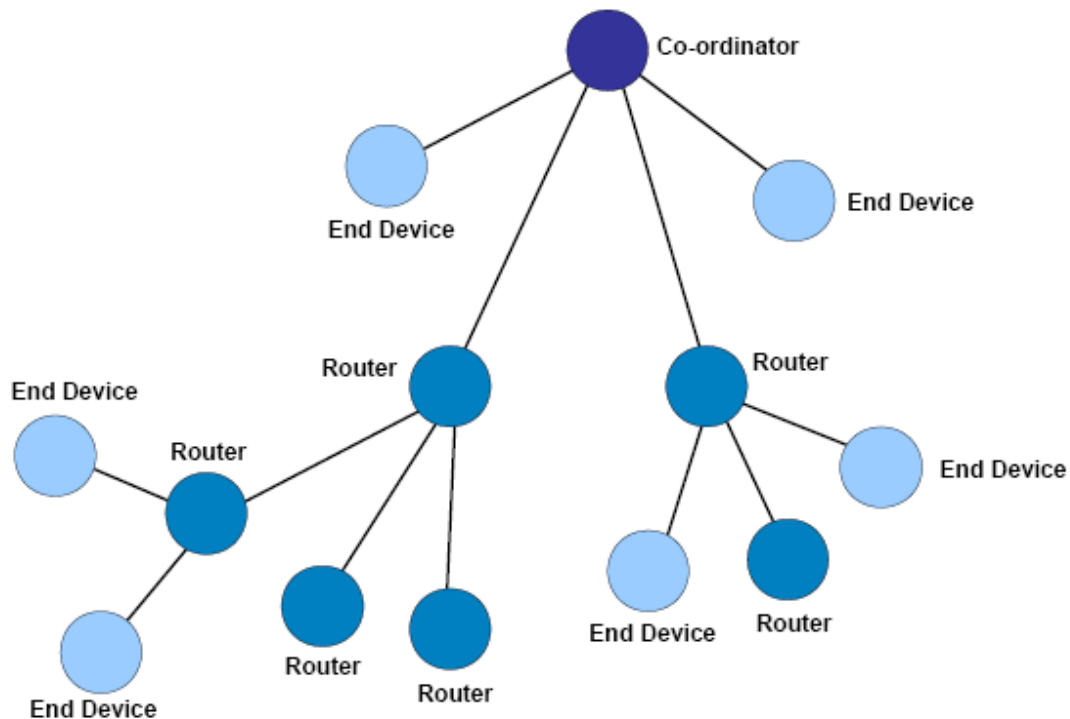


这种拓扑形式的缺点是节点之间的数据路由只有唯一的一个路径。**Co-ordinator** 有可能成为整个网络的瓶颈。

实现星形网络拓扑不需要使用 ZigBee 的网络层协议，因为本身 IEEE 802.15.4 的协议层就已经实现了星形拓扑形式，但是这需要开发者在应用层作更多的工作，包括自己处理信息的转发。

树形拓扑

树形拓扑包括一个 Co-ordinator 以及一系列的 Router 和 End Device 节点。Co-ordinator 连接一系列的 Router 和 End Device，他的子节点的 Router 也可以连接一系列的 Router 和 End Device。这样可以重复多个层级。树形拓扑的结构如下图所示：



需要注意的是：

- Co-ordinator 和 Router 节点可以包含自己的子节点。
- End Device 不能有自己的子节点。
- 有同一个父节点的节点之间称为兄弟节点
- 有同一个祖父节点的节点之间称为堂兄弟节点

树形拓扑中的通讯规则：

每一个节点都只能和他的父节点和子节点之间通讯。

如果需要一个节点向另一个节点发送数据，那么信息将沿着树的路径向上传递到最近的祖先节点然后再向下传递到目标节点。

这种拓扑方式的缺点就是信息只有唯一的路由通道。另外信息的路由是由协议栈层处理的，整个的路由过程对于应用层是完全透明的。

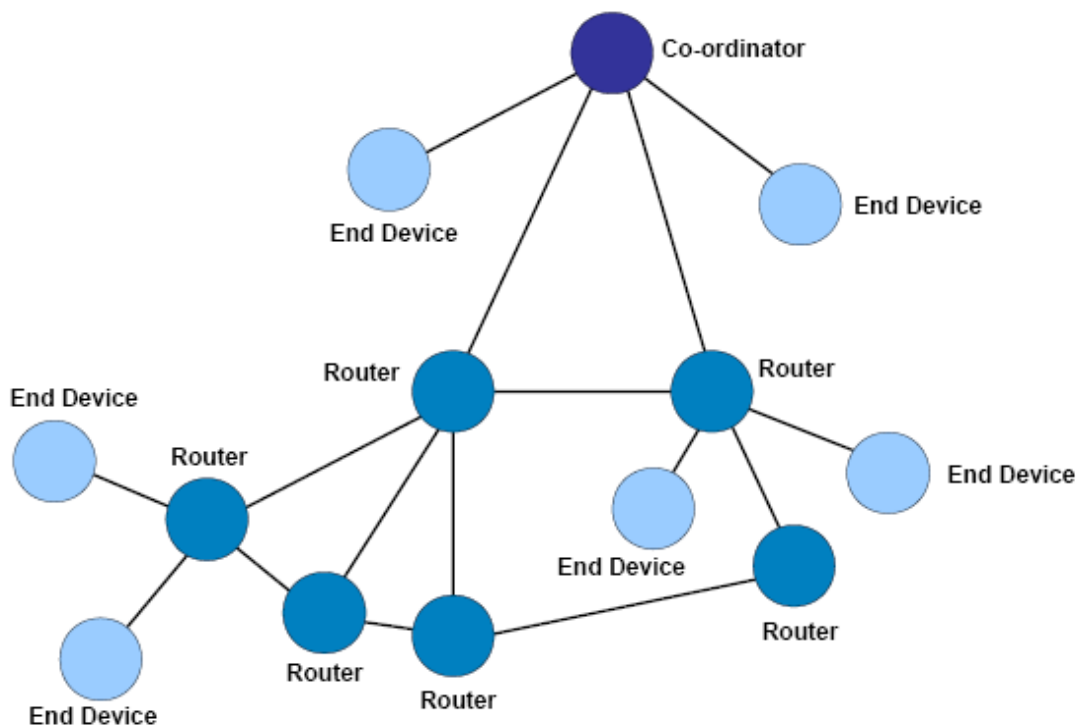
Mesh拓扑(网状拓扑)

Mesh 拓扑包含一个 Co-ordinator 和一系列的 Router 和 End Device。

这种网络拓扑形式和树形拓扑相同；请参考上面所提到的树形网络拓扑。但是，网

网状网络拓扑具有更加灵活的信息路由规则，在可能的情况下，路由节点之间可以直接的通讯。这种路由机制使得信息的通讯变得更有效率，而且意味这一旦一个路由路径出现了问题，信息可以自动的沿着其他的路由路径进行传输。

网状拓扑的示意图如下所示：



通常在支持网状网络的实现上，网络层会提供相应的路由探索功能，这一特性使得网络层可以找到信息传输的最优化的路径。

需要注意的是，以上所提到的特性都是由网络层来实现，应用层不需要进行任何的参与。

地址模式

IEEE MAC 地址：

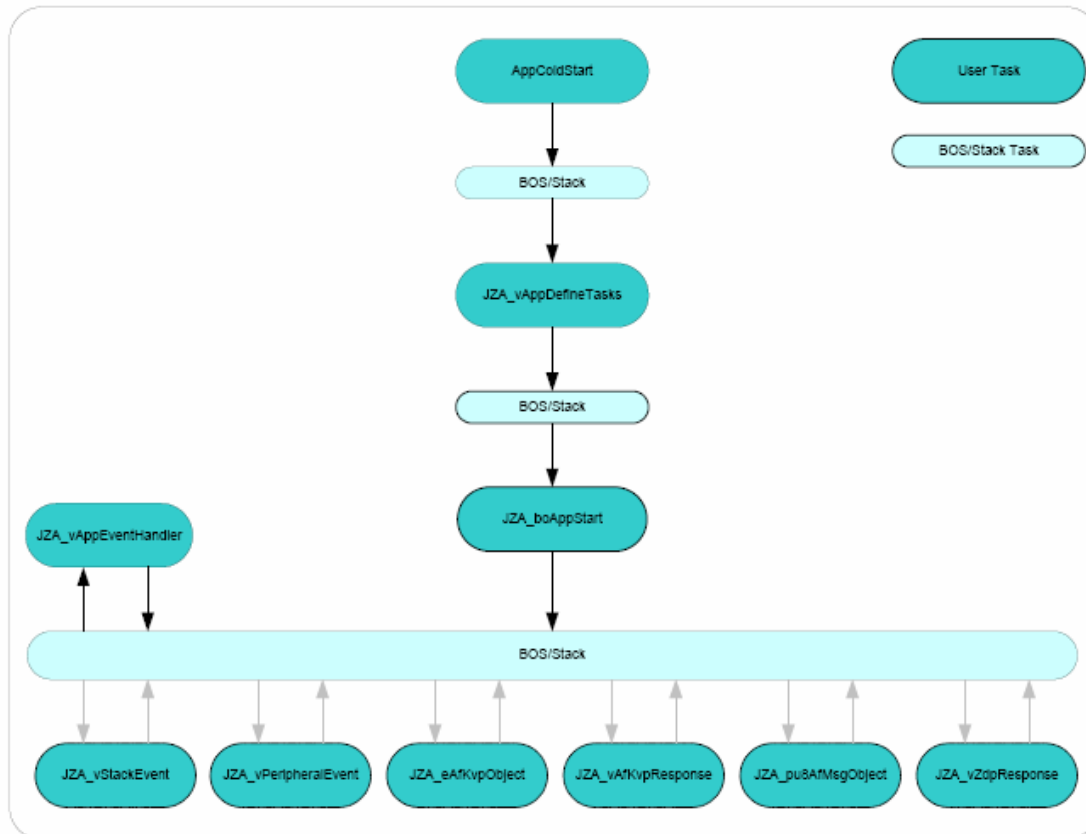
这是一种 64 位的地址，这个地址由 IEEE 组织进行分配，用于唯一的标识设备，全球没有任何两个设备具有相同的 MAC 地址。在 zigbee 网络中，有时也叫 MAC 地址为扩展地址。

16 位短地址：

16 位短地址用于在本地网络中标识设备，所以如果是处于不同的网络中有可能具有相同的短地址。当一个节点加入网络的时候将由它的父节点给它分配短地址。

第二节 ZigBee 协议栈的开发接口 API

首先我们看一下一个 ZigBee 设备的程序的大体架构是什么样的。



正如上图所示，用户的 ZigBee 应用程序实际上是和 ZigBee 协议栈交替的对处理器和外围部件进行操作，为了实现这个目标 Jennic 在 ZigBee 协议栈的基础上提供了 Basic Operating System BOS。上图所示的深绿色的部分就是 BOS 调用用户程序的接口。

我将在这一章里面介绍基于 Jennic ZigBee 协议栈开发的基本接口函数，您可以打开任何一个 ZigBee 的应用，找到以下一些函数的接口。

JZA_boAppStart
JZA_vStackEvent
JZA_vPeripheralEvent
JZA_vAppEventHandler
JZA_vAppDefineTasks
JZA_eAfKvpObject
JZA_vAfKvpResponse
JZA_u8AfMsgObject
JZA_vZdpResponse

还有

AppColdStart
AppWarmStart

还有对于下面这两个函数的调用

JZS_u32InitSystem
JZS_vStartStack

这些函数是您的应用和 ZigBee 协议栈进行交互的基本接口。主要分成三类

■ 第一类是应用的初始化函数，它们用于在设备上电时对协议栈进行初始化。

AppColdStart AppWarmStart

■ 第二类是应用程序调用协议栈的函数，这类函数通常由第一类函数进行调用

JZS_u32InitSystem JZS_vStartStack

■ 第三类是协议栈调用应用程序的函数，这类函数通常作为协议栈和应用程序进行通讯的接口。

JZA_boAppStart
JZA_vStackEvent
JZA_vPeripheralEvent
JZA_vAppEventHandler
JZA_vAppDefineTasks
JZA_eAfKvpObject
JZA_vAfKvpResponse
JZA_u8AfMsgObject
JZA_vZdpResponse

下面我们来分别介绍这三类函数。我们建议您同时打开一个 ZigBee 工程来对照的查看这些函数在实际项目中的功能。

应用的初始化函数

AppColdStart: 这个函数是用户应用程序的入口。设备上电后应用程序就从这个函数开始运行了。Jennic 的程序并没有 C 语言程序中的 main 函数。在这个函数的函数体中应该调用一系列的协议栈和 BOS(Basic Operation System)的初始化函数。下面我们看一个实际的例子，代码中的注释说明了每个语句的具体用途。

```
PUBLIC void AppColdStart(void)
{
/* 初始化协议栈 */
(void)JZS_u32InitSystem(TRUE);
/* 设置网络所使用的通道和网络 ID */
JZS_sConfig.u32Channel = 0x13;
JZS_sConfig.u16PanId = 0x1aab;
...
/* 启动 BOS */
bBosRun(TRUE);
}
```

AppWarmStart: 当设备从内存供电的休眠模式唤醒的时候将进入这个函数。启动后所有的内存数据都没有丢失。如果您的设备不需要休眠唤醒功能, 这个函数可以为空。

通常我们在范例代码中会看到这样的代码:

```
PUBLIC void AppWarmStart(void)
{
    AppColdStart();
}
```

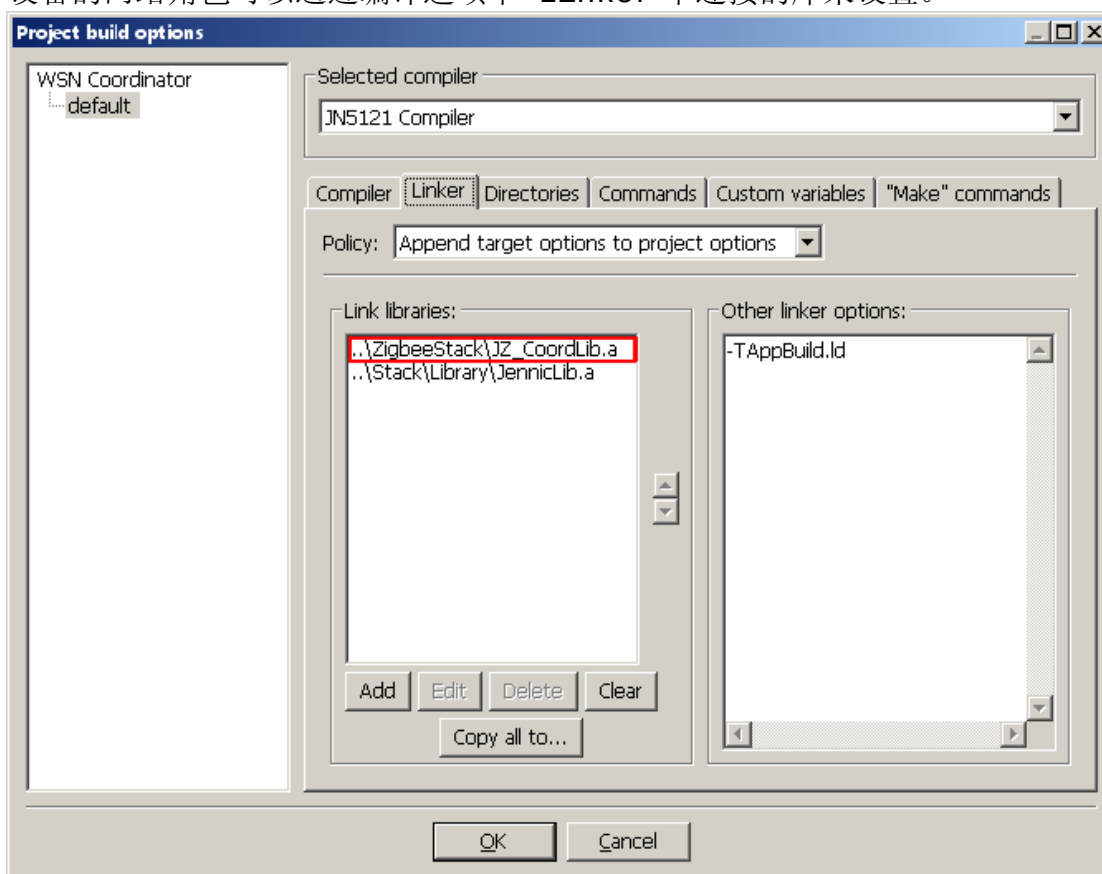
它简单的调用了 **AppColdStart** 来重新启动整个设备。

应用程序调用协议栈的函数

JZS_u32InitSystem : 这个函数初始化了 Jennic 的 ZigBee 协议栈

JZS_vStartStack : 调用这个函数后, 设备将作为 **Co-ordinator**、**Router** 或者 **End Device** 启动。如果是 **Co-ordinator** 将启动网络, 如果是 **Router** 或者 **End Device** 将试着加入网络。

设备的网络角色可以通过编译选项中 **Linker** 中连接的库来设置。



协议栈调用应用程序的函数

这些函数是协议栈在运行过程中如果需要应用程序进行相应的处理把控制权交给应用程序的接口。需要注意的是，所有这些函数都需要在您的应用中定义接口，即便您不使用其中的一些函数。

JZA_boAppStart: 这个函数让用户可以在协议栈启动前定义endpoint的descriptor。通常开发人员应该在这个函数中调用JZS_vStartStack来启动协议栈。

JZA_vStackEvent: 协议栈将通过这个函数反馈网络层的一些网络事件，比如网络启动成功或者节点加入成功，或者数据发送完成等等。
具体的我们看两个例子。

```
PUBLIC void JZA_vStackEvent(teJZS_EventIdentifier eEventId,
                           tuJZS_StackEvent *puStackEvent)
{
    if (eEventId == JZS_EVENT_NWK_STARTED)
    {
        bNwkStarted = TRUE;
    }
}
```

上面这段代码就是判断出 Co-ordinator 已经把网络启动完成了。

```
PUBLIC void JZA_vStackEvent (teJZS_EventIdentifier eEventId,
tuJZS_StackEvent *puStackEvent)
{
    switch (eEventId)
    {
    case JZS_EVENT_APS_DATA_CONFIRM:
        if (puStackEvent->sApsDataConfirmEvent.u8Status != APSDE_SUCCESS)
        {
            bTxProblems = TRUE;
        }
        break;
    }
}
```

而这段代码就是判断出数据发送出现了问题，需要应用层进行相应的处理。

关于这个函数的参数的具体说明请参考[JN-RM-2014](#)的相关章节。

JZA_vPeripheralEvent 这个函数主要用来处理外部的硬件中断，比如说时钟还

有串口等等。
还是来看两个例子。

```
PRIVATE bool_t bTimerFired = FALSE;
PUBLIC void JZA_vPeripheralEvent (uint32 u32Device,
uint32 u32ItemBitmap)
{
    if ((u32Device == E_AHI_DEVICE_SYSCTRL)
    && (u32ItemBitmap & (1 << E_AHI_SYSCTRL_WK0))
    {
        bTimerFired = TRUE;
    }
}
```

上面的代码检查了硬件中断是否来自于唤醒时钟 0 。

```
PUBLIC void JZA_vPeripheralEvent(uint32 u32Device, uint32 u32ItemBitmap)
{
    if (u32Device == E_AHI_DEVICE_UART0)
    {
        /* If data has been received */
        if ((u32ItemBitmap & 0x000000FF) == E_AHI_UART_INT_RXDATA)
        {
            /* Process UART0 RX interrupt */
            cCharIn = ((u32ItemBitmap & 0x0000FF00) >> 8);
        }
        else if (u32ItemBitmap == E_AHI_UART_INT_TX)
        {
            vUART_TxCharISR();
        }
    }
}
```

上面这段代码检测 UART 0 的事件并完成相应的串口传输。

JZA_vAppDefineTasks: 这个函数用于向BOS注册自己的用户任务。属于比较少用到的接口。这里就不详细讲解了。如果您感兴趣可以参考[BOS API Reference Manual](#)。

JZA_eAfKvpObject 用于用户程序接收处理其它节点发送来的KVP数据。这里把两个函数放在一起讲比较好。

JZA_u8AfMsgObject 用于用户程序接收处理其它节点发送来的MSG数据。

当远程节点发送数据的时候可以选择是发送 KVP 数据，还是 MSG 数据。从某种程度上看这两种发送方式没有什么本质的不同，在接收端的处理方式上就是从不同的函数来接收。

详细的代码可以参考 WSN Coordinator 的代码的部分或者 Wireless Uart 的相应部分。

这两个范例分别展示了 MsgObject 和 KvpObject 的处理过程。繁琐但不困难。这里就不详细的进行解释了。关于数据发送的函数我们将在下一个小节来介绍。

JZA_vAfKvpResponse 这个函数用来接收发送的KVP包的回应。这一回应由远程节点发出。

通常这个函数用来判断和远程节点通讯是否通畅。

JZA_vZdpResponse 这个函数用来接收所发送的ZDP请求的回应，比如说 Binding或者Match Descriptor的请求。

```
PRIVATE void vPerformMatchRequest(void)
{
    AF_SIMPLE_DESCRIPTOR *pAfSimpleDesc;

    /* obtain simple descriptor */
    pAfSimpleDesc = afmeSearchEndpoint(sWuart.sSystem.u8WuartEndpoint);

    if(pAfSimpleDesc)
    {
        /* Send Match Descriptor request */
        zdpMatchDescReq(0xffff,
                        pAfSimpleDesc->u16ProfileId,
                        pAfSimpleDesc->u8OutClusterCount,
                        pAfSimpleDesc->au8OutClusterList,
                        pAfSimpleDesc->u8InClusterCount,
                        pAfSimpleDesc->au8InClusterList,
                        APS_TXOPTION_NONE);

        sWuart.sSystem.eBound = E_BIND_BINDING;
    }
}
```

```
PUBLIC void JZA_vZdpResponse(uint8 u8Type, uint8 u8Lqi, uint8 *pu8Payload, uint8 u8PayloadLen)
{
    /* if already matched with another node, ignore any incoming responses */
    if (sWuart.sSystem.eBound == E_BIND_MATCHED)
    {
        return;
    }

    switch (u8Type)
    {
```

```
case Match_Desc_rsp:
    /* response to a MatchDescriptor request has been received. If valid
    extract the short address and endpoint form the response and store it.
    Turn off LED to indicate successful matching */
    if ((pu8Payload[0] == ZDP_SUCCESS_VALID) && (pu8Payload[3] > 0))
    {
        sWuart.sSystem.u16MatchAddr = (uint16) (pu8Payload[2] << 8);
        sWuart.sSystem.u16MatchAddr |= pu8Payload[1];

        sWuart.sSystem.u8MatchEndpoint = pu8Payload[4];
        sWuart.sSystem.eBound = E_BIND_MATCHED;
        /* turn off LED2 to indicate matching successful */
        vAHI_DioSetOutput(LED2_MASK, 0);
    }
    break;

case End_Device_Bind_rsp:
    /* this would be used to receive responses if a Bind Request
    had been used */
    break;

default:
    break;

}
}
```

上面的两段代码是从 **zigbee wireless uart** 中摘出的,它们演示了一对 **zigbee** 无线串口设备如何完成 **match** 操作的。

上面的全部函数的具体的参数说明可以参考[JN-RM-2014](#)。在实际的开发过程中我们就需要在上面所介绍的这些开发接口上添加自己的应用逻辑,定义自己的数据处理过程并且通过这些接口函数在适当的时机调用。这些函数就好比整个应用的骨架,把我们应用程序的代码和**ZigBee** 协议栈紧密地联系在一起。下一个小节我们将继续学习**Application Framework API**(应用框架接口函数)。具体讲解决如何利用协议栈发送数据和处理设备描述。

第三节 应用框架接口函数

这一小节我们将利用应用框架接口函数来解决发送数据和处理设备描述的问题。详细的函数文档可以参考[JN-RM-2018](#)。在本培训文档里我们只关注于具体的函数使用。

应用框架接口函数主要分为两大类

- 一类是 AF sub-layer Data Entity (AFDE) API：用来创建和发送数据请求，这类函数定义在 `af.h` 文件中
- 另一类是 AF sub-layer Management Entity (AFME) API：用来添加、修改、删除设备的描述(device descriptor)，这一系列函数在 `afProfile.h` 中定义。

AFDE类函数只有一个，是`afdeDataRequest()`，这个函数用来向网络层发出数据发送的请求。这个函数非常的重要，我们会多花一些笔墨来详细的解释他的每个参数的含义。

```
bool_t afdeDataRequest(  
    AF_ADDRTYPE sDstAddr,  
    uint8 u8SrcEP,  
    uint16 u16ProfileId,  
    uint8 u8ClusterId,  
    AF_FRAME_TYPE eFrameType,  
    uint8 u8TransCount,  
    AFDE_DATA_REQ_INFO *psAfdeDataReqInfo,  
    uint8 *pau8Afd,  
    uint8 txOptions,  
    uint8 u8DiscoverRoute,  
    uint8 u8RadiusCounter  
);
```

我们先来看这个函数的原形，下面是每一个参数的具体含义，如果涉及到需要解释的 `struct` 或者 `union` 的数据类型，我们也一起解释。

- **sDstAddr**：这个参数数据要发送的目标地址，它是一个 `AF_ADDRTYPE` 类型的数据。那么他的具体定义是这样的

```
typedef struct  
{  
    DEV_ADDR_MODE hAddrMode;  
    uint8 u8EndPoint;  
    uint16 u16Address;  
} AF_ADDRTYPE;
```

相应的解释如下：

字段名	描述	取值范围
hAddrMode	地址的模式, 设置选用 16 位短地址还是 64 位扩展地址	DEV_ADDR_NOT_PRESENT DEV_16BIT_ADDR DEV_64BIT_ADDR
U8EndPoint	目标 endpoint	0x01 到 0xF0
U16Address	目标 16 位地址(这里可能有点问题, 如果我设置地址模式使用 64 位的话, 这里应该怎么设置呢?)	0x0000 到 0xFFFF

- **U8SrcEP** 源地址的Endpoint 范围是 0x01 到 0xF0
- **U16Profileid** 所采用的profile ID
- **U8ClusterId**所采用的cluster ID
- **eFrameType**使用的数据帧类型 0x01=KVP 0x02=MSG
- **u8TransCount** 本次请求发送的数据事务的数量。这里可能需要解释一下, 就目前的理解数据请求一次可以发送多个数据包, 这个参数就表示了数据包的数量, 不过我们通常在应用中只发送一个, 所以这个参数通常就是 1。
- ***pasAfdeDataReqInfo** 这个参数是一个AFDE_DATA_REQ_INFO类型数据的数组。其中每一个数据都描述了每个数据包的一些信息。具体的内容可以参考 AFDE_DATA_REQ_INFO的数据结构

```
typedef struct
{
    uint8 u8SequenceNum;
    AF_COMMAND_TYPE_ID hCommandTypeID;
    AF_ATTRIBUTE_DATA_TYPE hAttributeDataType;
    uint16 u16AttributeID;
    AF_ERROR_CODE hErrorCode;
    uint8 u8DividedAfdLen;
}AFDE_DATA_REQ_INFO;
```

字段名	描述
U8SequenceNum	标识 KVP 和 MSG 数据帧的序列号, 这个号应该是单增的, 用来跟踪数据的丢包。
hCommandTypeID	命令类型, 只用于 KVP 类型的数据
hAttributeDataType	属性数据的类型, 只用于 KVP 类型的数据
U16AttributeID	属性标识, 只用于 KVP 类型的数据
hErrorCode	错误代码, 只用于 KVP 类型的数据

U8DividedAfdulLen

包中数据长度，用于 KVP 和 MSG 数据

这里又涉及了很多具体的struct类型的定义，这里不在详述，如果您需要了解更多的信息请参考[RM-2018](#) 12 页到 13 页的内容。

■ ***pauAfdul** 所发送的数据区。

■ **txOptions** 发送模式。可以选择下面的值，并且下面的值可以用 or 的方式联合采用

APS_TXOPTION_NONE (0x00) 没有任何选项

SECURITY_ENABLE_TRANSMISSION (0x01) 使用安全传输

USE_NWK_KEY (0x02) 使用网络键

ACKNOWLEDGED_TRANSMISSION (0x03) 采用确认传输模式

■ **U8DiscoverRoute** 设定所采用的路由发现模式

SUPPRESS_ROUTE_DISCOVERY (0x00) 使用强制路由发现模式，采用这种模式如果路由表已经建立，那么数据将使用现有的路由表路由，如果路由表没有建立那么数据将沿着树状路径路由。

ENABLE_ROUTE_DISCOVERY (0x01) 路由发现使能，采用这种模式，如果路由表已经建立，那么数据将使用现有路由，如果路由表没有建立那么此次数据发送请求将引发路由探索动作。

FORCE_ROUTE_DISCOVERY (0x02) 这一模式将明确的引发路由探索操作，路由表将重新建立。

■ **u8RadiusCounter** 数据发送的深度，也就是数据包所发送的转发次数限制，如果设置为 0 那么协议栈将采用 2 倍的MaxDepth。

下面我们看一个典型的数据发送请求例程

```
void afdeMsgReqExample(void)
{
    AF_ADDRTYPE sAddrDst;
    AFDE_DATA_REQ_INFO sAfdeDataReqInfo[3];
    uint8 au8Afdul[3] = {0x10, 0x20, 0x30};
    sAddrDst.hAddrMode = DEV_ADDR_NOT_PRESENT;
    sAddrDst.u16Address = COORD_SHORT_ADDRESS;
    sAddrDst.u8EndPoint = 0x01;
    sAfdeDataReqInfo[0].u8SequenceNum = ais.u8AfTransactionSequence++;
    sAfdeDataReqInfo[0].u8DividedAfdulLen = 1;
    afdeDataRequest(sAddrDst,
    0x01,
    0x0103,
    0x1C,
    MSG,
```

```
1,  
sAfdeDataReqInfo,  
au8Afd,  
APS_TXOPTION_NONE,  
SUPPRESS_ROUTE_DISCOVERY,  
0);  
}
```

接下来是 AFME 函数，讲到这里需要详细的解释 **Descriptor** 的作用和含义。**ZigBee** 协议的作用就是为了实现低成本的设备之间的无线通讯，那么这就涉及到未来来自不同厂商的设备之间能够互相的兼容并且连通的问题，于是在 **ZigBee** 协议的规范中定义了一种标准的设备用于自我描述的机制，便于 **ZigBee** 兼容设备之间进行互相的识别和访问。

那么在 **ZigBee** 规范中有三种主要的 **descriptor** 和两种可选的 **descriptor**。三种主要的 **descriptor** 分别是 **Node**，**Node Power** 和 **Simple descriptor**。两种可选的 **descriptor** 是 **Complex** 和 **User descriptor**。

下面简单分别介绍每种 **descriptor** 的含义：

- **Node Descriptor**：描述了网络节点的各种基本特性，比如节点类型（End Device，Router 还是 Co-ordinator），所使用的频率，等等
- **Node Power Descriptor**：描述了网络节点的供电特性，比如供电的模式（常供电还是其他）可选的电源模式，当前电源模式等等。

以上种类的 **Descriptor** 每个网络节点设备通常只有一份，用户可以根据设备的实际情况进行修改和更新。

- **User Descriptor** 通常都是用户定义的一些设备描述。
- **Complex Descriptor** 是扩展的设备描述。
- **Simple Descirptor**：用于对网络节点上的 **endpoint** 进行描述。通常设备上每一个 **end point** 都需要定义自己的 **simple descriptor**。所以一个网络节点设备可以包含多个 **simple descriptor**。它描述了 **endpoint** 所定义的 **Profile ID**，设备标识和版本，以及发送和接收的 **cluster**。

需要注意的是：如果一个 **endpoint** 上没有正确定义的 **simple descriptor** 那么它就不能正确地接受别的节点发送来的数据，这是特别需要注意的。

那么我们下面详细介绍如何为一个 **endpoint** 添加 **simple descriptor** 如果开发者需要了解更多关于 **descripotr** 的操作可以参考 [JN-RM-2018](#) 的第三章。

我们使用 **afmeAddSimpleDesc** 函数来对 **endpoint** 添加 **descriptor**。

我们先看一下这个函数的原型

```
uint8 afmeAddSimpleDesc(  
uint8 u8Endpoint,  
uint16 u16ProfileID,  
uint16 u16DeviceID  
uint8 u8DeviceVersion,  
uint8 u8Flags,  
uint8 u8InClusterCount,  
uint8 *pau8InClusterList,  
uint8 u8OutClusterCount,  
uint8 *pau8OutClusterList);
```

参数说明

- **u8Endpoint** EndPoint序号 (范围 0x01 到 0xF0)
- **u16ProfileID** 所使用的Profile ID (范围 0 到 0xFFFF)
- **u16DeviceID** 设备ID (范围 0 到 0xFFFF)
- **u8DeviceVersion** 设备版本 (范围 0 到 0xFF)
- **u8Flags** 标志参数
 - bit 0 表示是否有complex descriptor
 - bit 1 表示是否有 user descriptro
 - bit 2、3 是保留的
- **u8InClusterCount** 输入cluster数量
- ***pau8InClusterList** cluster数组
- **u8OutClusterCount** 输出cluster数量
- ***pau8OutClusterList** cluster数组

接下来是一个实际的例子

```
PUBLIC bool_t JZA_boAppStart(void)  
{  
    uint8 u8InputClusterCnt      = 1;  
    uint8 au8InputClusterList[]  = {WSN_CID_SENSOR_READINGS};  
    uint8 u8OutputClusterCnt     = 0;  
    uint8 au8OutputClusterList[] = { };  
  
    (void)afmeAddSimpleDesc(WSN_DATA_SINK_ENDPOINT,  
                             WSN_PROFILE_ID,  
                             0x0000,
```

```
        0x00,  
        0x00,  
        u8InputClusterCnt,  
        au8InputClusterList,  
        u8OutputClusterCnt,  
        au8OutputClusterList);  
  
    JZS_vStartStack();  
  
    return TRUE;  
}
```

通常 **Simple Descriptor** 应该在协议栈启动前进行定义，所以我们一般应该在 **JZA_boAppStart** 中定义相应的 **descriptor** 然后调用 **JZS_vStartStack** 来启动协议栈。上面的代码是 **WSN** 例程中数据汇聚节点的定义，这个定义出现在 **co-ordinator** 的代码中，开发者可以参考前面的函数说明来分析代码。

其他的操作函数请大家参考[RM-2018](#) 基本上各种**descriptor**的操作都是大同小异的，这里不再详细介绍了。

第四节 ZigBee Device Profile API

通过前几节的内容，我们已经完全可以完成 ZigBee 网络的创建和数据发送的功能，这一节我们将介绍一些关于 ZigBee Device Profile 的 API。这一系列 API 可以帮助我们完成更复杂的应用和服务。

ZDP API 是用来和远程节点的 ZigBee Device Objects 打交道的函数接口。它主要包含以下三类函数：

- **ZDP Device Discovery API**：用来获得远程节点的网络标识。
- **ZDP Service Discovery API**：用来获得远程节点所能提供的服务
- **ZDP Binding API**：用来实现设备之间的绑定和反绑定。

ZDP 系列的 API 通过请求应答的模式来工作。发送请求的节点将相应的请求通过函数调用的形式发往网络(指定地址，或者广播)，然后由网络中能够应答的节点发出回应信息。比如说如果需要根据 MAC 地址找到相应设备的网络短地址，那么相应的请求就被发往网络，符合这个 MAC 地址的节点就会发出回应告知自己的网络短地址，回应的信息通过发出请求的节点 JZA_vZdpResponse() 接口函数返回给应用程序。

了解了 ZDP API 的基本运行模式，我们就可以参考 JN-RM-2017 来查找需要的函数以及其相应的回应说明。

下面我们介绍一些典型的 ZDP API 的应用。

Binding

Binding 在节点之间提供了一种逻辑上的对应关系，为数据在相关节点之间的传输提供了一种更加方便的方式。

只有使用同一个 Profile ID 并且 cluster 之间的输入输出关系相互匹配的节点之间才能够进行绑定。

绑定关系被保存在叫做 **binding table** 的数据结构中。

绑定关系包括下面几种基本的形式，

- one-to-one,
- one-to-many,
- many-to-one.

Binding table 根据保存位置的不同可以分为直接绑定和间接绑定。

直接绑定: **binding table** 保存在数据的发送节点，对于直接绑定而言，当需要发送数据时协议栈会搜索整个 **binding table** 然后找到所有匹配的数据传输关系，然后将信息发送到所有匹配的目标节点。

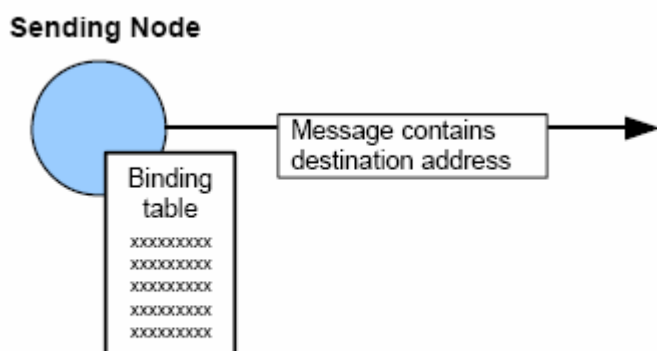


Figure 8: Direct (Local) Binding

间接绑定: **binding table** 保存在 **co-ordinator** 节点。发送数据的节点通过 **Co-ordinator** 来转发数据。**Co-ordinator** 扫描整个 **binding table** 找到匹配的数据接收节点，然后将数据发送到所有匹配的节点。

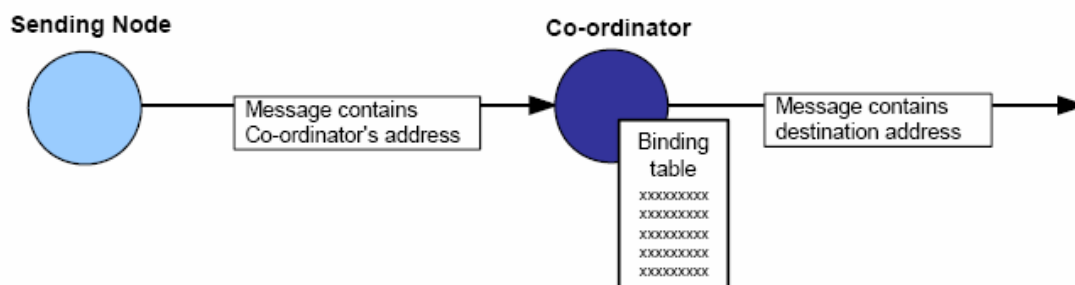


Figure 9: Indirect (Co-ordinator) Binding

下面我们就看一下如何使用 **ZDP API** 来实现 **binding**

我们从一个例子入手，实现家庭控制中开关和灯泡的绑定。

```
PRIVATE void vPerformEndBindRequest(uint8 u8Endpoint)
{
    AF_SIMPLE_DESCRIPTOR *pAfSimpleDesc;
    uint16                u16BindingTarget = 0x0000;

    /* recall endpoints simple descriptor from the application framework */
    pAfSimpleDesc = afmeSearchEndpoint(u8Endpoint);

    /* create and send a request for binding */
    if(pAfSimpleDesc)
    {
        zdpEndDeviceBindReq(u16BindingTarget,
                            pAfSimpleDesc->u8Endpoint,
                            pAfSimpleDesc->u16ProfileId,
                            pAfSimpleDesc->u8OutClusterCount,
                            pAfSimpleDesc->au8OutClusterList,
                            pAfSimpleDesc->u8InClusterCount,
                            pAfSimpleDesc->au8InClusterList,
                            APS_TXOPTION_NONE);
    }

    /* update programs binding state variable */
    sSwitch.eBound = E_BIND_BINDING;
}
```

```
PRIVATE void vPerformEndBindRequest(uint8 u8Endpoint)
{
    AF_SIMPLE_DESCRIPTOR *pAfSimpleDesc;
    uint16                u16BindingTarget = 0x0000;

    /* recall endpoints simple descriptor from the application framework */
    pAfSimpleDesc = afmeSearchEndpoint(u8Endpoint);

    /* create and send a request for binding */
    if(pAfSimpleDesc)
    {
        zdpEndDeviceBindReq(u16BindingTarget,
                            pAfSimpleDesc->u8Endpoint,
                            pAfSimpleDesc->u16ProfileId,
                            pAfSimpleDesc->u8OutClusterCount,
                            pAfSimpleDesc->au8OutClusterList,
                            pAfSimpleDesc->u8InClusterCount,
```

```
        pAfSimpleDesc->au8InClusterList,
        APS_TXOPTION_NONE);

    }

    /* update programs binding state variable */
    sLight.eBound = E_BIND_BINDING;
}
```

上面是相应的灯泡和开关的绑定请求的过程。可以发现这个例程使用的是间接绑定，所以 `BindingTarget = 0x0000`，把 `binding` 的请求发送到了

`Co-ordinator`。这样就在 `Co-ordinator` 建立了相应的 `binding table`。

相应的 API 函数使用的是 `zdpEndDeviceBindReq`。

接下来是 `binding` 请求的回应处理过程。请阅读下面代码的相应注释。

```
PUBLIC void JZA_vZdpResponse(uint8 u8Type, uint8 u8Lqi, uint8 *pu8Payload,
                           uint8 u8PayloadLen)
{
    /* if already matched with another node, ignore any incoming responses */
    if (sSwitch.eBound != E_BIND_BINDING)
    {
        return;
    }
    //根据回应的类型 u8Type 来决定具体的操作
    switch (u8Type)
    {
        case Match_Desc_rsp: // 匹配请求的处理过程

            break;

        case End_Device_Bind_rsp: //绑定请求的处理过程
            if (pu8Payload[0] == ZDP_SUCCESS_VALID) //数据负载的第一个字节标志着请求是否有效
            {
                sSwitch.eBound = E_BIND_BOUND; //请求成功了，设定设备的状态
                vLedControl(LED2, FALSE); //用指示灯标志绑定的成功
            }
            else
            {
                sSwitch.eBound = E_BIND_NONE; //绑定失败，设定设备的状态
            }
            break;
    }
```

```
default:
    break;

}
}
```

下面我们来看，绑定成功后数据发送的过程有什么差别：

```
hDstAddr.hAddrMode = DEV_ADDR_NOT_PRESENT;
hDstAddr.u16Address = 0x0000;
```

可以看到在 switch 的数据发送端，数据发送请求的地址模式设置为了 DEV_ADDR_NOT_PRESENT。然后地址设置为了 0x0000，也就是 Co-ordinator 的地址，因为 binding table 是保存在 co-ordinator 的，co-ordinator 会自动地根据 binding table 来处理数据的转发。

Zigbee Stack 的Match服务。

关于 match 服务，我们可以在 Jennic Application Notes 的 JN-AP-1016-Zigbee-Wireless-UART 找到具体的应用案例。这个 API 的使用就是通过广播的模式找到网络中可以匹配的相应的 endpoint。我们来看具体的代码：

```
PRIVATE void vPerformMatchRequest(void)
{
    AF_SIMPLE_DESCRIPTOR *pAfSimpleDesc;

    /* obtain simple descriptor */
    pAfSimpleDesc = afmeSearchEndpoint(sWuart.sSystem.u8WuartEndpoint);

    if(pAfSimpleDesc)
    {
        /* Send Match Descriptor request */
        zdpMatchDescReq(0xffff,
                        pAfSimpleDesc->u16ProfileId,
                        pAfSimpleDesc->u8OutClusterCount,
                        pAfSimpleDesc->au8OutClusterList,
                        pAfSimpleDesc->u8InClusterCount,
```

```
        pAfSimpleDesc->au8InClusterList,  
        APS_TXOPTION_NONE);  
  
        sWuart.sSystem.eBound = E_BIND_BINDING;  
    }  
}
```

这个函数发出了相应的匹配请求，以找到网络中可以通讯的无线串口节点。请求的发送地址是 **0xffff**，那么这个请求将在整个无线网络中进行广播。匹配节点 **ZDO** 将返回回应信息，使发出请求的节点找到自己。我们来看回应的处理过程：

请参考代码中的注释信息

```
PUBLIC void JZA_vZdpResponse(uint8 u8Type, uint8 u8Lqi, uint8 *pu8Payload,  
                             uint8 u8PayloadLen)  
{  
    /* 如果该设备已经和其他的节点匹配了，那么就无条件的返回，这样保证无线串口节点只和一个其他节点进行匹配 */  
    if (sWuart.sSystem.eBound == E_BIND_MATCHED)  
    {  
        return;  
    }  
  
    switch (u8Type)  
    {  
    case Match_Desc_rsp:  
        /* 匹配成功*/  
        if ((pu8Payload[0] == ZDP_SUCCESS_VALID) && (pu8Payload[3] > 0))  
        {  
            //将相应的匹配节点的地址信息保存，以备发送数据使用  
            sWuart.sSystem.u16MatchAddr = (uint16) (pu8Payload[2] << 8);  
            sWuart.sSystem.u16MatchAddr |= pu8Payload[1];  
            //将匹配节点的 endpoint 信息保存，以备发送数据使用  
            sWuart.sSystem.u8MatchEndpoint = pu8Payload[4];  
            sWuart.sSystem.eBound = E_BIND_MATCHED;  
            /*通过指示灯告知用户匹配成功*/  
            vAHI_DioSetOutput(LED2_MASK, 0);  
        }  
        break;  
  
    case End_Device_Bind_rsp:  
        /* 处理绑定请求的回应，这个例子中没有用到 */  
        break;  
    }
```

```
default:
    break;

}
```

好了我们已经通过一些简短的例子了解了 ZDP API 的大致应用方法，ZDP API 包含了大量的函数，其使用方法大同小异，您可以参考[JN-RM-2017](#)来了解所有的接口函数，在本手册中不再详述。

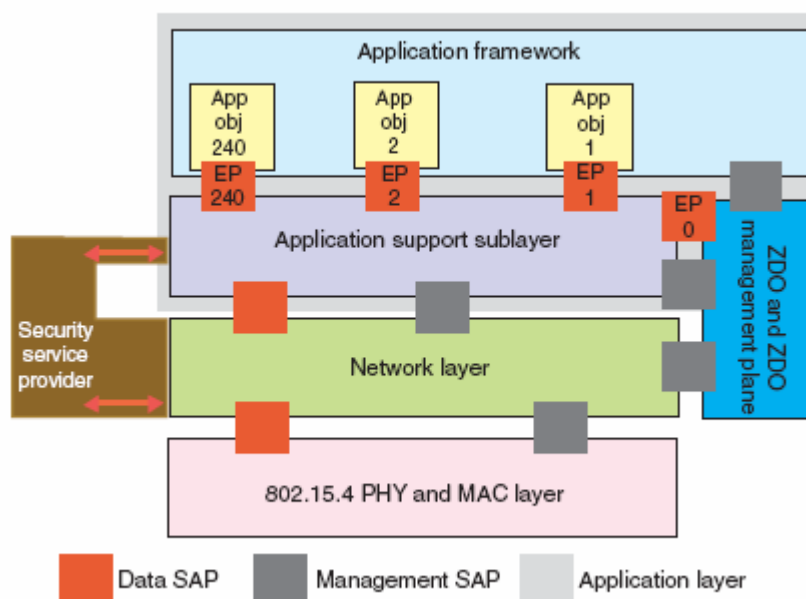
到本节的结束我们已经基本上全面的了解了基于 ZigBee 协议栈 API 进行无线应用开发的基本内容。现在让我们回顾一下

在第一节中我们了解了 ZigBee 网络的一些基本概念和术语。第二节我们了解了 ZigBee 应用程序的运行过程以及协议栈的基本接口函数，第三节我们学习了如何发送数据和定义 endpoint 上的 simple descriptor, 第四节，我们学习了一些 ZDP 的基本内容以及相应的 binding 和 match 操作。

如果您已经迫不及待的开始写您的程序，我想本手册到现在为止的内容已经足够了。[JN-RM-2014.2017.2018](#)应该是您手头必备的开发手册。不过为了让这本手册更加的完整，我们在下两章还将花费一些篇幅来讲解如何开发基于 802.15.4 的网络应用以及模块外围部件的接口函数。

第五章 基于 802.15.4 协议栈进行开发

对于基于 802.15.4 的无线应用开发，我们首先要搞清楚的一个问题是它和基于 ZigBee 协议栈进行开发有什么不同，以及什么情况下才会用到 802.15.4 的协议栈。



从上一章 ZigBee 协议栈的框架图我们就可以了解到，802.15.4 协议栈是 ZigBee 协议栈的基础，换句话说各厂商的 ZigBee 协议栈都是基于 MAC 层的 802.15.4 协议栈进行开发的。ZigBee 在 802.15.4 协议栈的基础上提供了路由、网络拓扑、节点管理等等众多的网络层的功能，使得用户在开发应用层的程序的时候可以不用考虑这些问题，直接获得联通的无线网络连接，并且可以保证协议栈层的市场兼容性。所以说基于 ZigBee 协议栈进行应用的开发会更为简单。

那么反过来说，802.15.4 协议栈则更为基础，只能够实现基本的星形网络拓扑，用户需要了解和管理更多的内容，开发的灵活性也足够大，而且从所开发出的最终产品的成本上来说，可以为每个网络节点省出一点 ZigBee 协议栈的版权使用费用。

既然如此，什么样的开发者才需要从 802.15.4 协议栈开始呢？对于网络的拓扑方式有特殊的要求，需要自己严格的控制网络的拓扑方式，对于应用程序的规模有着较大需求，采用相对更小的 802.15.4 协议栈可以使开发者获得更大的编程空间。开发者出于研究或者商业的目的希望自己开发类似 ZigBee 协议栈的嵌入式软件产品。等等。
所以如果您的应用需求如果属于以上几种情况，那么您就有了足够的理由进入这片荒蛮之地。

本章主要以[JN-AN-1046](#)作为范例代码，讲解如何利用范例代码模版开发基于 802.15.4 协议栈的网络应用，代码模版是非信标网络的，如果您需要使用信标网络可以参考Home Sensor Demo 的代码

[JN-AP-1050-802-15-4-Home-Sensor-Demo](#)

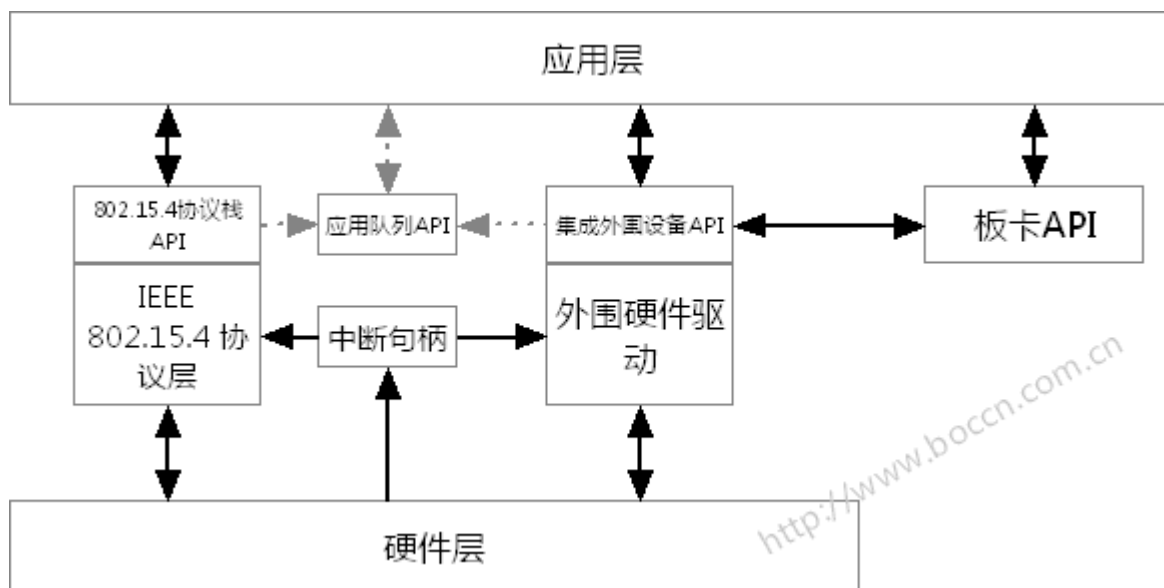
本章主要包含下面几节

- 第一节： IEEE 802.15.4 协议栈的架构，接口和中断说明
- 第二节： IEEE 802.15.4 网络的建立过程
- 第三节： 应用程序的代码框架

第一节： IEEE 802.15.4 协议栈的架构，接口和中断说明

IEEE 802.15.4 的协议栈架构概述，开发人员可以参考如下的图理解

802.15.4 的协议栈架构



下面对这个图进行一下解释

- 应用程序通过 **802.15.4** 的协议栈 **API** 和 **IEEE 802.15.4** 的协议层进行交互。这一交互用来实现 **MCPS/MLME** 的请求和确认，消息的标识和回应。
IEEE 802.15.4 协议层和更底层的硬件以及寄存器进行交互。
- 应用程序通过集成外围设备 **API** 和芯片上集成的外围设备（比如 **AD**，**DA**，**DIO**，**Timers...**）进行交互。这一 **API** 访问硬件寄存器
- 应用程序通过板卡 **API** 和开发包中的传感器板或者控制器板访问板卡上的设备，比如传感器等等。
- 硬件层产生各种中断通过中断句柄将其转发给各个软件模块
- 开发人员还可以使用我们提供的**应用队列 API** 来简化对于通讯协议层和硬件 **API** 中断的处理过程。

1. 802.15.4 协议栈 API

我们提供的 **802.15.4** 的协议栈**API**可以使应用访问 **802.15.4** 的协议栈以及控制基于**JN51xx** 无线微控制器的**IEEE 802.15.4** **MAC**层。详细的**API**说明可以参考[JN-RM-2002](#)

2. 集成外围设备 API

集成外围设备 API 可以使应用程序创建控制以及处理 JN51xx 无线微处理器的集成外围设备。比如 UART，计时器和通用 IO

详细的API说明可以参考[JN-RM-2001](#)

3. 板卡 API

板卡 API 可以使应用程序使用 无线传感器板和无线控制器板的板载设备，比如 LCD 屏，LED，按钮，温湿度传感器等等。这一 API 封装了对于硬件寄存器和中断的处理

详细的API说明可以参考[JN-RM2003](#)

4. 应用队列 API

开发人员在开发无线应用的时候可以选择性的使用我们提供的 应用队列 API(Application Queue API). 这一系列的 API 负责处理各种中断，并向应用层提供了一个基于队列的接口，这样开发人员就可以不用自己处理各种中断的回调函数。当来自下层的中断产生的时候，中断的入口就会被分类压入到三个专用的队列中（MLME（网络管理队列），MCPS（网络数据队列），硬件事件队列）应用程序灵活的掌握时机从队列中取出事件进行处理。

应用队列 API 也容许开发人员使用传统的方法注册自己的回调函数，但是我们强烈推荐开发人员采用应用 API 来处理队列事件以加快开发速度并同时保证程序的稳定性。

这一API的详细说明请参考[JN-RM-2025](#)

中断和回调函数

需要说明的是如果您使用应用队列 API(Application Queue API) 来开发程序的话您就可以无需了解中断和回调函数的详细细节。但是了解这一细节可以

帮助您更好的理解应用程序的整体软件架构和工作原理。

现在我们从应用程序如何使用协议栈的 **API** 说起，应用程序将在各种情况下调用协议栈 **API** 的入口函数。很多的 **802.15.4** 的调用请求将导致协议栈的程序进行一系列的操作，而这些操作将在调用结束后持续的进行，比如对于传输数据帧。为了避免使用多任务的操作系统，协议栈将尽可能的在中断上下文中工作较长的时间。

当需要有信息通知应用层的时候，不论是应用层前一个请求的回复还是协议栈或者硬件产生的状态报告，协议栈层都将调用事先注册的回调函数。需要注意的是回调函数的运行周期仍然在中断上下文中，所以回调函数中的处理过程应该尽可能的短。

所有的中断都由硬件产生，中断将通过中断句柄传递给 **802.15.4** 协议栈或者硬件的驱动程序，这些协议栈和驱动将决定是自己处理这些中断还是将其通过注册回调函数继续传递给上层的应用程序层。

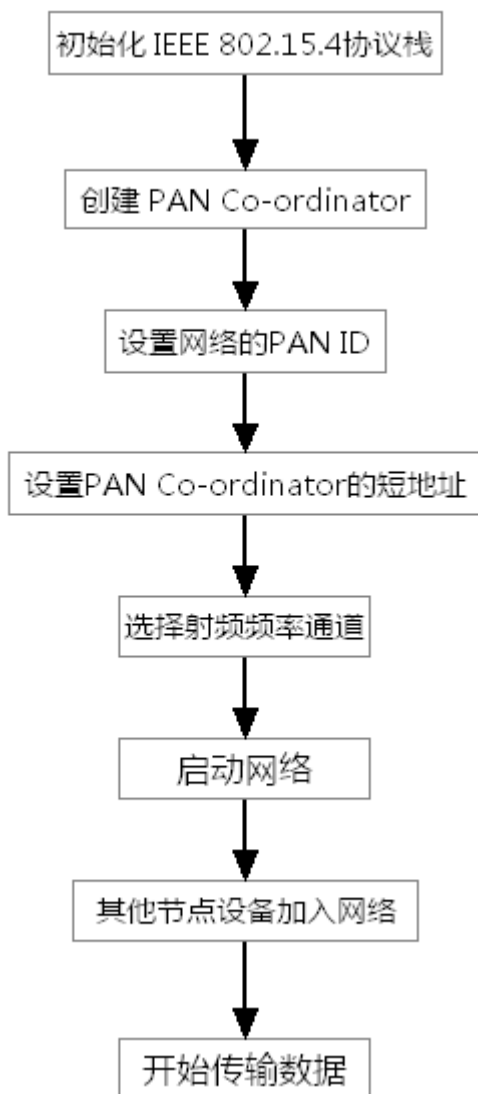
第二节： IEEE 802.15.4 网络的建立过程

这篇文章将详细的描述 **IEEE 802.15.4** 无线网络的建立过程，我们描述的是一个星形网络的建立过程。需要注意的是这里我们所建立的网络是非信标网络。

概述

下图描述了一个 **IEEE 802.15.4** 网络的建立过程

<http://www.boccn.com.cn>



下面将详细解释一下整个网络的建立过程。

建立网络的过程

1. 首先，每一个设备的 IEEE 802.15.4 的协议栈必须要对其的 PHY 和 MAC 层进行初始化的工作。
2. 创建 PAN Co-ordinator，每一个网络必须有一个也只能有一个 PAN Co-ordinator，建立网络的第一个步骤就是需要选择并且初始化这个 Co-ordinator. 初始化 PAN Co-ordinator 的动作只在相应的被事先约定的设备上进行。

3. 选择 PAN ID 和 Co-ordinator 的短地址

PAN Co-ordinator 一旦初始化完成就必须为它的网络选定一个 PAN ID 作为网络的标识。PAN ID 可以被人为的预定义。

这里需要说明的是 PAN ID 可以通过侦听其他网络的 ID 然后选择一个不会冲突的 ID 的方式来获取。PAN Co-ordinator 可以扫描多个频率通道，当然开发人员也可以指定设备优先扫描指定的通道来确定不和其他网络冲突的 PAN ID。

每一个 PAN Co-ordinator 设备都已经具有了一个唯一的固定的 64 位 IEEE MAC 地址，通常我们叫做扩展地址。但是作为组网的标识他还必须分配给自己一个 16 位的网络地址，通常我们叫做短地址。使用短地址进行通讯可以使网络通讯更轻量级更加高效。这一短地址是由开发人员预先定义的，PAN Co-ordinator 的短地址通常被定义为 0x0000。

4. 选择射频频率

PAN Co-ordinator 必须选择一个网络所建立的射频频率通道。PAN Co-ordinator 可以通过进行一次能量扫描检测来找到一个相对安静的通道。通过通道能量扫描检测 API 将返回每一个通道的能量水平，能量水平高就标志着这个通道的无线信号比较活跃。接下来 PAN Co-ordinator 就可以根据这些信息选择一个可以利用的通道来建立自己的无线网络。

5. 启动网络

一个无线网络的启动过程是从初始化配置 PAN Co-ordinator 开始的，然后这个设备就将以 Co-ordinator 的模式启动。然后 PAN Co-ordinator 就将开放对于加入网络的请求应答。

6. 设备加入网络

一旦网络中出现了可以利用的 **Co-ordinator**，其他的网络设备就可以加入网络了。一个设备如果需要加入网络首先其要完成自己的初始化过程，然后他需要找到 **PAN Co-ordinator**。

为了找到 **PAN Co-ordinator**，设备需要进行频道扫描，它将在特定的频率通道中发送信标请求。当 **PAN Co-ordinator** 检测到信标请求后，

Co-ordinator 将回应相应的信标来向设备标识自己。

对于信标网络（所谓的信标网络就是 **PAN Co-ordinator** 将周期性的发送信标），需要加入网络的设备可以被动的侦听来自 **Co-ordinator** 的信标。

设备找到 **PAN Co-ordinator** 之后就将发出加入网络的申请。**Co-ordinator** 将决定是否具有足够的资源接受新的设备，并且决定是否接受和拒绝设备加入网络。

如果 **PAN Co-ordinator** 接受了设备，它将发送一个 16 位的短地址给设备，作为设备在网络中的标识。

在设备之间传输数据

当网络中出现了 **PAN Co-ordinator** 和至少一个端节点设备后，网络就可以进行数据传输了。数据传输的过程如下所述。

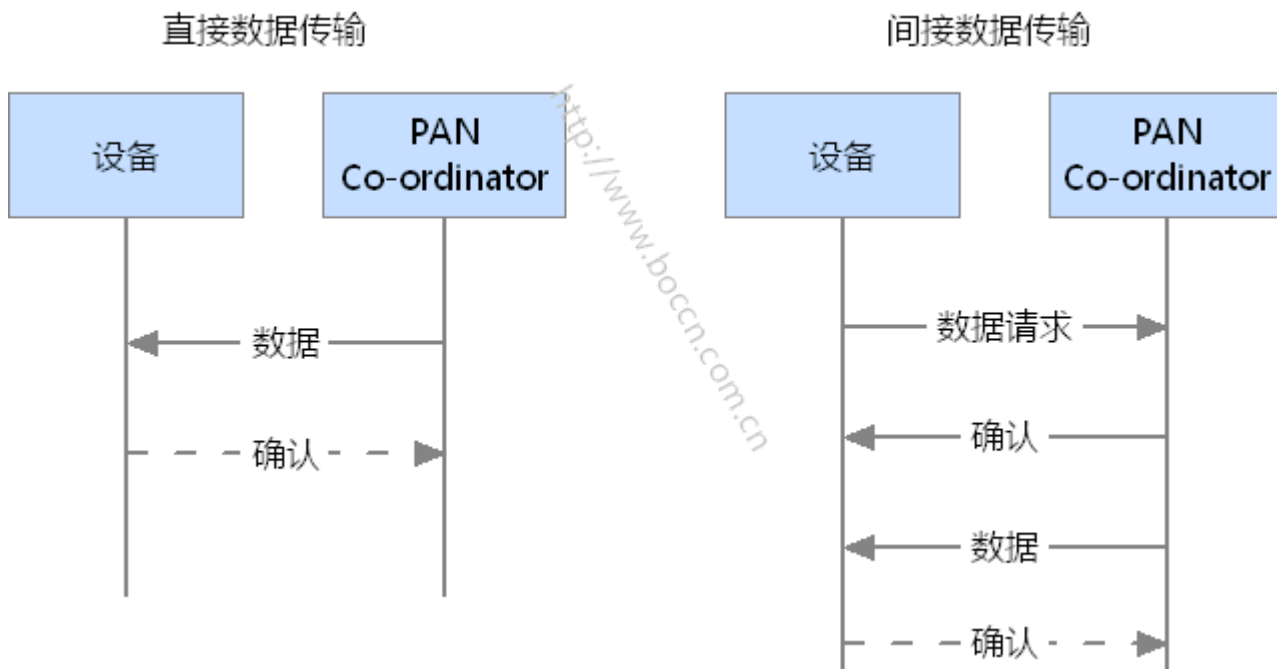
Co-ordinator 向 **End Device** 传输数据

有两种方法可以实现 **Co-ordinator** 向 **End Device** 传输数据：

1. 直接传输：**PAN Co-ordinator** 可以将数据直接发送给 **End Device**。**End Device** 接收到数据后可以发送确认消息给 **Co-ordinator**。这种数据传输方式就要求 **End Device** 随时都处于数据接收的状态，也就是要求其随时都要处

于唤醒的状态。后面我们将详细的解释这种工作方式。

2. 间接传输：另外一种传输方式就是 **Co-ordinator** 可以将数据保存起来等待 **End Device** 请求读取数据。采用这种方式，**End Device** 为了获得数据必须先要发送数据请求。发送数据请求后，**Co-ordinator** 就会判断是否有需要发送给这个设备的数据，如果有就发送相应的数据给 **End Device**。接到数据的设备将发送确认信息。这一方式适用于 **End Device** 设备需要较低功耗的情况，其大部分的工作状态都处于休眠状态以节省能量。以上所述的数据传输方式可以用如下图表示：



End Device 向 Co-ordinator 传输数据

End Device 通常向 PAN Co-ordinator 直接发送数据，Co-ordinator 接到数据后可以发送确认消息。

第三节：应用程序的代码框架

本节详细的讲解应用模版代码，我们将先介绍支持文件，然后再介绍代码中的函数调用。

我们建议您使用我们提供的IDE集成开发平台**Code::Blocks**，这个平台是基于

开源的系统并且集成了Jennic的编译处理器，这一工具具有相对友好的编程开发环境而且正在不断地改进，这个工具可以从我们的FTP下载到
<ftp://bocon.com.cn/jennic>

应用模版

应用模版提供了 IEEE 802.15.4 无线网络应用开发的基础框架代码(非信标网络)。您可以修改我们提供的范例代码来生成您自己的应用。

范例代码可以从我们的FTP下载到，代码的编号是 [JN-AN-1046](#)

预先需要知道的知识：

我们所讲解的范例代码需要以下的环境：

- 您应该有一个设备作为 PAN Co-ordinator
- 您应该至少再有一个设备作为 End Device
- 您可以使用预先定义的 PAN ID 和短地址作为范例程序的网路参数，作为一个演示的应用这些参数已经可以演示基本的概念了，但是如果您需要开发一款实际的产品或者项目，您应该规定自己的网络地址的分配逻辑。
- 我们将采用星形的网络拓扑结构
- 我们将采用非信标的网络(这就意味着 PAN Co-ordinator 将不会周期的发送信标)
- 我们将使用短地址来标识网络中的设备
- 数据的传输模式采用直接传输，并且进行数据的接收确认
- 我们将不采用任何的安全措施

文件的说明：

在这个范例程序中我们使用三个文件：

- Config.h: 这个头文件包含了一些网络参数设置，比如 PAN ID, 短地址，和需要扫描的通道。
- coordinator.c: 这个文件是 PAN Co-ordinator 的源代码。
- enddevice.c: 这个文件是 End Device 的源代码。

在您构建自己应用的过程中您将主要修改上面这三个文件。

代码描述

这部分内容从函数的级别详细解释了代码。我们将分别解释 PAN

Co-ordinator 和 End Device 的代码。

config.h 头文件将被引用到两个源代码文件中,同时两个源代码文件也引用了

以下的头文件:

jendefs.h, AppHardwareApi.h, AppQueueApi.h, mac_sap.h,
mac_pib.h

coordinator.c 的内容

开发者最常问的问题之一就是为什么 Jennic 的程序都没有 Main 函数,这个熟悉的函数哪里去了呢? 这是因为 Jennic 程序都由 boot loader 来启动和引导, boot loader 引导完成后就将自动的调用 AppColdStart 函数, 您可以认为 AppColdStart 就是我们通常所说的 Main()。

AppColdStart 将进行下面的操作:

1.AppColdStart 将调用函数 vInitSystem(), 这一函数将完成以下任务:

- 初始化设备的 IEEE 802.15.4 的协议栈
- 设置 PAN ID 和 PAN Co-ordinator 的短地址, 在这个应用中这些参数都由我们预定义在 config.h 这个文件中
- 打开射频接收器
- 使 Co-ordinator 可以接受其他的设备加入网络

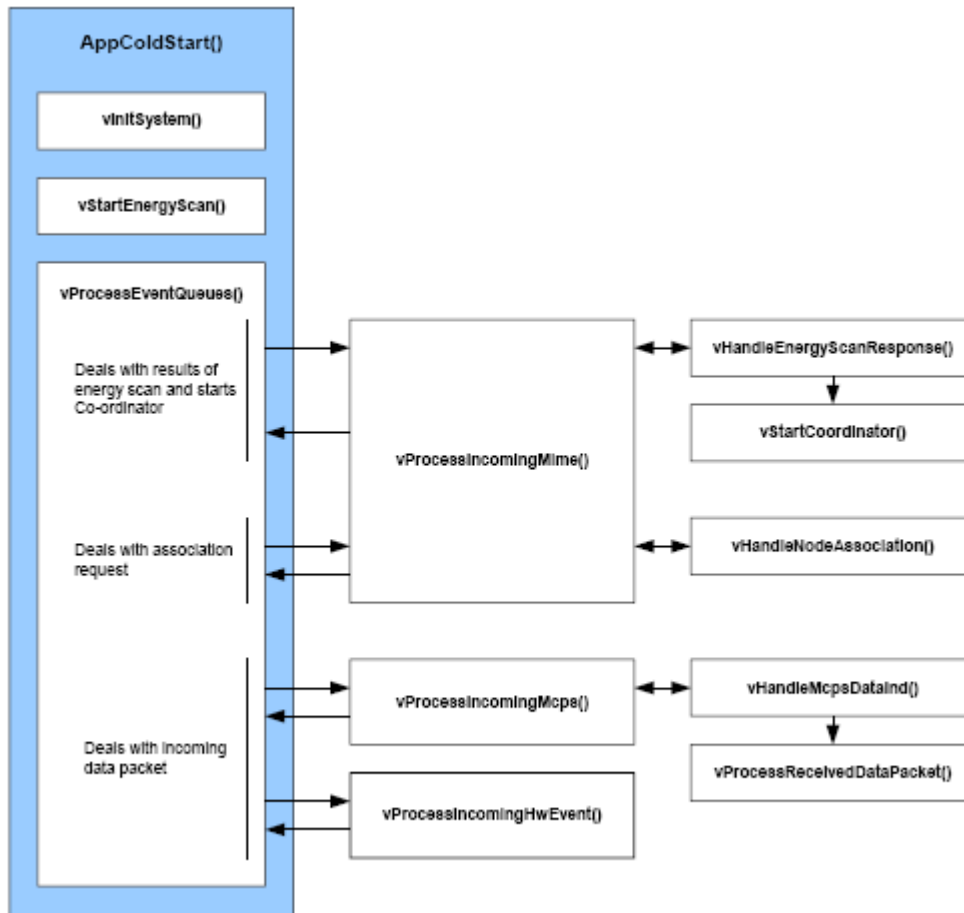
2.AppColdStart()会调用 vStartEnergyScan(), 这一函数将会开始在各个通道进行能量扫描以获得各个通道的能量级别. 所扫描的通道以及速率都定义在 config.h 中。扫描将通过初始化一个 MLME 请求并将其发送给 IEEE 802.15.4 的 MAC 层来实现。

3. `AppColdStart()` 将通过调用 `vProcessEventQueues()` 的方式等待 MLME 的回应。`vProcessEventQueues()` 函数将检查三个不同类型的事件队列并将接到的事件交给不同的事件处理函数处理。比如这个函数将调用 `vProcessIncomingMlme()` 函数来处理 MLME 回应。而这个函数将调用 `vHandleEnergyScanResponse()` 来处理能量检测扫描的回应结果。这个函数将检查所有通道的能量级别，并挑选一个最安静的通道作为建立网络的通道。接下来将调用 `vStartCoordinator()` 函数，这个函数将设置必要的参数并且递交 MLME 请求来启动网络，启动网络的请求不需要处理任何的回复信息。

4. `AppColdStart()` 循环调用 `vProcessEventQueues()` 来等待其他设备的加入网络的请求，入网请求将以 MLME 请求的方式发送到 `codinator`。当请求到达的时候函数将调用 `vHandleNodeAssociation` 来处理。接下来 `codinator` 将创建并发送入网请求回复。

5. `AppColdStart` 将循环调用 `vProcessEventQueues` 来处理来自于 MCPS 的消息队列和来自于硬件的消息队列。

- 当数据到达 MCPS 队列后，`vProcessEventQueues` 首先调用函数 `vProcessIncomingMcps()` 来接收到达的数据帧。`vProcessIncomingMcps()` 调用 `vHandleMcpsDataInd()`，这个函数将调用 `vProcessReceivedDataPacket`，在这个函数里面您可以自定义您自己的数据处理过程。
- 当硬件事件到达硬件队列后，`vProcessEventQueues` 将调用函数 `vProcessIncomingHwEvent` 来接收到来的事件。您需要在这个函数中自定义自己的事件处理过程。
您可以参考下面的示意图来理解



enddevice.c 的内容介绍

End Device 的运行过程仍然是从 `AppColdStart` 开始。这一函数和 `Co-ordinator` 的运行方式完全的不同，下面将详细的讲解这个过程。

1. `AppColdStart` 调用 `vInitSystem`，这个函数将初始化 IEEE 802.15.4 的协议栈

2. `AppColdStart()` 调用 `vStartActiveScan()` 开始对于活动通道的扫描，
End Device 将向扫描的通道发送信标请求，并接收 PAN Co-ordinator 的信标请求回应。需要扫描的通道和速率将在 `config.h` 中定义。扫描请求的初始化和发送的工作可以通过 MLME 请求的方式通过 IEEE 802.15.4 的 MAC 层发送。

3. `AppColdStart()` 将通过 `vProcessEventQueues` 来检查和处理 MLME 回应。这个函数将调用 `vProcessIncomingMlme()` 来处理收到的 MLME 回应。

`vHandleActiveScanResponse()` 会被调用处理返回的活动通道扫描结果：

- 如果找到 PAN Co-ordinator, 函数将保存相应的 Co-ordinator 信息 (比如 PAN ID, 短地址, 逻辑通道), 并且调用 `vStartAssociate()` 向 Co-ordinator 来提交入网请求, 这一请求将通过 MLME 请求的方式提交.
- 如果 PAN Co-ordinator 没有被找到 (可能是由于 Co-ordinator 还没有初始化完成). 这一函数将重新调用 `vStartActiveScan()` 来重新启动扫描。

4. `AppColdStart` 将循环的调用 `vProcessEventQueues()` 等待来自 Co-ordinator 的入网回复。当收到回复后就将调用 `vProcessIncomingMlme()`, 然后将调用 `vHandleAssociateResponse` 来处理回复, 接下来的函数将检查回复的状态:

- 如果 PAN Co-ordinator 接受的入网请求, 将设备置于联网状态。
- 如果 PAN Co-ordinator 拒绝了入网的请求, 函数就将重新调用 `vStartActiveScan()` 来开始搜索另外一个 PAN Co-ordinator。

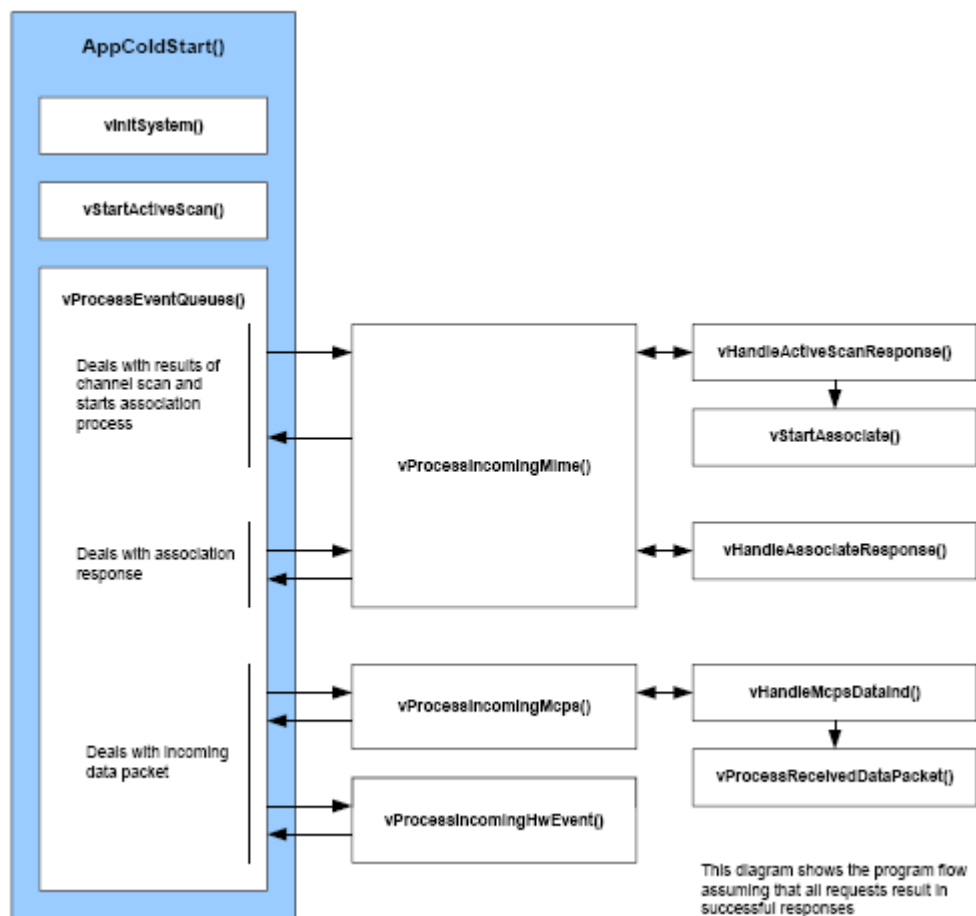
5. `AppColdStart()` 接下来将循环的调用 `vProcessEventQueues` 来等待来自于 PAN Co-ordinator 的 MCPS 信息或者硬件的队列信息。

- 当数据到达了 MCPS 队列, `vProcessEventQueue()` 首先使用函数 `vProcessIncomingMcps()` 来接收数据帧, 接着调用 `vHandleMcpsDataInd()`, 接着调用 `vProcessReceivedDataPacket()`, 开发人员可以在这个函数里面编写

自己的数据处理过程。

- 当硬件事件到达硬件事件队列, `vProcessEventQueues()` 将调用 `vProcessIncomingHwEvent()` 来接收到达的事件, 您可以在这个过程中编写自己的事件处理逻辑。

下面的图表示了 End Device 的工作过程。



如何调整骨架代码:

下面的内容提供了一些指导来告诉开发人员如何利用骨架代码进行修改以符合不同的应用需求。内容包括:

- 如何修改预定义的 PAN ID?
- 如何修改预定义的短地址?
- 如何向网络中添加新的 End Device?
- 如何修改通道扫描?
- 如何定义数据包的接收过程?

■ 如何编写数据传输过程？

如何修改预定义的PAN ID？

在我们提供的范例骨架代码中 PAN ID 被定义在 config.h 中，他被预先定义为 0xCAFE。

如果您需要使用不同的 PAN ID, 打开 config.h 您就可以将 PAN ID 修改为您需要的 16 进制地址

```
#define PAN_ID 0xCAFE
```

需要注意的是所定义的 PAN ID 不要和在同一区域内的其他基于 IEEE 802.15.4 的网络冲突。

如何修改预定义的短地址

对于 PAN Co-ordinator 和 End Device 的短地址定义也都在 config.h 中。

在我们提供的范例代码中，Co-ordinator 的短地址被定义为 0x0000 而第一个 End Device 的地址被定义为 0x0001。后面这个地址也是 End Device 的起始短地址, 如果你需要增加多个 End Device, 那么短地址将自动的被生成分配，每一个短地址都比上一个增加 0x0001。

如果您需要使用不同的短地址，打开 config.h 并且修改下面的定义

```
#define COORDINATOR_ADR 0x0000
#define END_DEVICE_START_ADR 0x0001
```

需要注意的是，我们通常都是设置 PAN Co-ordinator 的地址为 0x0000, 您在应用中最好也遵从这个约定。

如何让更多的End Device 加入网络

我们提供的范例代码被设计为网络中最少有两个节点：一个 PAN

Co-ordinator 和 End Device. 默认的这个范例代码最大可以接受 10 个 End Device，这就是说您可以使用 10 个 End Device 加入网络而不用修改任何代

码。如果您需要使用更多的 **End Device**，您可以参考下面的说明来修改代码。

修改 **config.h** 文件

首先您需要修改 **config.h** 文件来定义所能接受的最大节点数量的数值

#define MAX_END_DEVICES	10
-------------------------	----

修改 **enddevice.c** 文件

这个文件是 **End Device** 的源代码文件，如果您所使用的新的 **End Device** 具有一些特殊的功能和特性，比如额外的温湿度传感器，或者 **AD**，**DA** 的功能，那么您需要修改这个文件来生成新的设备。

关于 **coordinator.c** 文件

这一文件是 **Co-ordinator** 的源代码文件，一般来说您不需要修改这个文件来接收新的 **End Device** 节点的加入。

如何修改通道扫描

我们提供的范例代码提供了两种通道扫描的模式

- 能量检测扫描，**PAN Co-ordinator** 在建立网络的时候会进行这个扫描来选择合适的通道建立网络
- 活动通道扫描，**End Device** 在需要加入网络的时候进行这个扫描来找到通道中的 **PAN Co-ordinator**

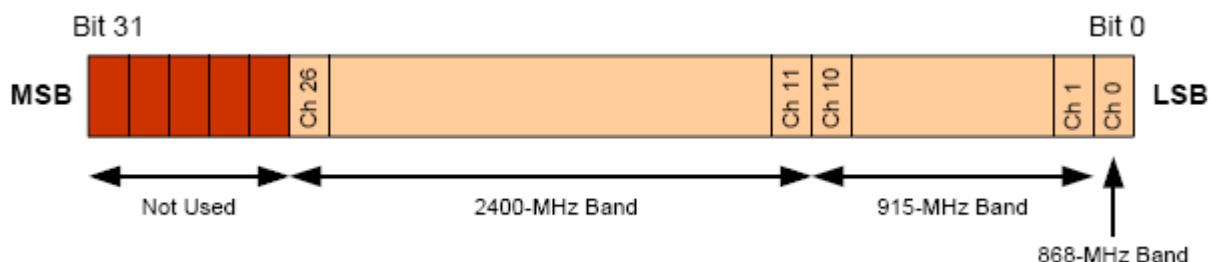
通常情况下是没有必要检查所有可能的频率通道的。**IEEE 802.15.4** 所定义的 27 个通道分布在三个频段(868, 915 和 2400MHz)。因为网络只能建立在某一个频段内，所以扫描其他的两个频段的通道就没有太大的意义。对于 **Jennic** 产品而言，通常工作在 2400 MHz 频段，也就是 11 到 26 通道。而且通常来说您在开发的过程中也可以发现一个特定的通道总是被本地的一些无线网络所占用，所以对于这些通道的扫描也可以从通道扫描中排除。您可以预先定义需要扫描的通道，而且可以定义每一个通道所检查的时间。这些参数都可以通过下面的定义实现。

定义所扫描的通道

找到 config.h 中下面的位置

```
#define SCAN_CHANNELS 0x07fff800UL
```

SCAN_CHANNELS 参数定义了需要扫描的通道。每一个位都代表一个通道，最低的位代表通道 0。如果一个通道需要扫描那么相应的位就是 1，如果一个通道不需要扫描相应的位就是 0。 您可以参考下面这个图来理解参数的定义。



需要注意：SCAN_CHANNELS 的定义既被用于能量检测扫描也被用于活动通道扫描，另外 Jennic 的产品只能工作在 2.4G 这个频段，所以对于其他通道的扫描是没有意义的。
定义通道扫描速率

在 config.h 中找到下面的定义

```
#define ACTIVE_SCAN_DURATION 3
#define ENERGY_SCAN_DURATION 3
```

上面两个参数定义了相应的扫描时间，

ACTIVE_SCAN_DURATION 定义了 活动通道扫描的时间，

ENERGY_SCAN_DURATION 定义了能量检测扫描的时间。这些参数定义了花在

每一个通道的扫描时间，如果需要换算成 ms, 可以参考下面的公式

Active Channel Scan

Channel scan duration(ms) = $15.36 \times ((2^{\text{ACTIVE_SCAN_DURATION}}) + 1)$

Energy Detection Scan

Channel scan duration(ms) = $15.36 \times ((2^{\text{ENERGY_SCAN_DURATION}}) + 1)$

打个比方说，如果值设置为 3，那么通道扫描的时间就使 138.24ms

需要注意的是，这两个参数必须被设置为 0 到 14 的整数，也就是说通道扫描的

时间将在 30.72ms 和 251.6736s 之间

如何定义处理接收到的数据的过程？

IEEE 802.15.4 的协议栈会将接收到的数据压入到 MCPS 队列中。我们提供的范例代码不论是 PAN Co-ordinator 还是 End Device 都将从队列中取出数据，但是不对数据做任何的处理，您必须自己定义自己的数据处理过程。但是范例代码中提供了一个空的函数 `vProcessReceivedDataPacket()` 来给你使用。您可以修改 `coordinator.c` 或者 `enddevice.c` 中这个函数的过程来定义自己的数据处理过程。

如何编写数据的发送过程？

在 `coordinator.c` 和 `enddevice.c` 这两个源文件中，都已经定义好了一个数据传输函数 `vTransmitDataPacket()`。如果您需要传输数据只需要简单的调用这个函数就可以了。

第六章 外围部件的操作

这一章我们将介绍一些JN5121 模块外围部件的操作方法，所有的函数都可以在[JN-RM-2001-Integrated-Peripherals-API](#)这个参考手册中找到，所以我们不打算逐个的介绍每个API函数，而是采用主题问答的形式面向实际的问题来提供可供参考的代码片断和相应的解释。本章的内容来自于我们和我们的客户的实际工程案例，其主题列表将不断丰富，请随时更新本手册最新版本以获得更多的内容。

如何实现定时休眠唤醒

最简单的方法就是在 JZA_vStackEvent 事件中来实现，这样就会在数据发送完成后实现定时的休眠唤醒。

```
PUBLIC void JZA_vStackEvent(teJZS_EventIdentifier eEventId,
                          tuJZS_StackEvent *puStackEvent)
{
    if (eEventId == JZS_EVENT_NWK_JOINED_AS_ROUTER)
    {
        bNwkJoined = TRUE;
    }
    if (eEventId == JZS_EVENT_APS_DATA_CONFIRM)
    {
        if (puStackEvent->sApsDataConfirmEvent.u8Status == APSDE_SUCCESS)
        {
            /* 唤醒时钟使能 */
            vAHI_WakeTimerEnable(E_AHI_WAKE_TIMER_0, TRUE);
            /* 设置休眠周期 */
            vAHI_WakeTimerStart(E_AHI_WAKE_TIMER_0, 1920000);

            u8AHI_WakeTimerFiredStatus();
            vAHI_PowerDown( FALSE ); // False -> Sleep; TRUE -> Deep Sleep
        }
    }
}
```

这段代码将在数据发送完成后休眠一分钟，然后 warmstart。

如何使用 SPI 接口

下面这段代码展示了一个基本的 SPI 接口的命令发送和数据接收的过程，请参考代码中的注释信息。

```
vAHI_SpiConfigure( 1, /* 所使用的 SPI 设备数量*/
                  E_AHI_SPIM_MSB_FIRST, /* send data MSB first */
                  E_AHI_SPIM_TXPOS_EDGE, /* TX 上升沿*/
                  E_AHI_SPIM_RXPOS_EDGE, /* RX 上升沿 */
                  3, /* SPI 时钟速率 16MHz /2^n*/
                  E_AHI_SPIM_INT_DISABLE, /* 是否使用 SPI 中断 */
                  E_AHI_SPIM_AUTOSLAVE_DSABL); /*自动片选设置 */

vAHI_SpiSelect(E_AHI_SPIM_SLAVE_ENBLE_1); //片选第一个 SPI 设备

uint8 u8Cmd = 0x9f;
uint8 u8Temp=0;
vAHI_SpiStartTransfer8(u8Cmd); //发送 SPI 命令
vAHI_SpiWaitBusy();

//Bit 1
vAHI_SpiStartTransfer8(0xFF); //发送一个无意义的数据来激活 SPI 时钟
vAHI_SpiWaitBusy();
u8Temp = u8AHI_SpiReadTransfer8(); //读取返回的数据
vAHI_SpiWaitBusy();

vAHI_SpiSelect(SPI_SLCT_NONE); //取消片选
```

如何使用 UART

关于这个问题最简单的方法就是使用 JN-AN-1015 中的标准串口操作函数。他们包括
Uart.c, uart.h, serial.c, serial.h, printf.c, printf.h,
serialq.c, serialq.h

把这几个文件加入到您的工程中，然后在主程序的代码头部 include 下面这些

文件

```
#include "serial.h"
#include "uart.h"
#include "printf.h"
```

然后在您的代码的初始化设备的函数里加入

```
vSerial_Init();
```

一般应该在 `(void)bBosRun(TRUE);` 的前面加入。

这样就可以在您的代码的任何地方使用 `printf` 来向串口输出信息了。比如

```
printf("Temp= %d \r\n",u16Temp);
```

如果您还需要在程序中从串口接收数据那么就需要做如下的修改:

首先把下面的代码加入到

`PUBLIC void JZA_vPeripheralEvent(uint32 u32Device, uint32 u32ItemBitmap)`这个事件中，以便接收到串口中断。

```
if (u32Device == E_AHI_DEVICE_UART0)
{
    /* If data has been received */
    if ((u32ItemBitmap & 0x000000FF) == E_AHI_UART_INT_RXDATA)
    {
        /* Process UART0 RX interrupt */
        cCharIn = ((u32ItemBitmap & 0x0000FF00) >> 8);
    }
    else if (u32ItemBitmap == E_AHI_UART_INT_TX)
    {
        vUART_TxCharISR();
    }
}
```

根据您的具体的应用，您可能需要对上面的代码做一些修改。

然后为了不和 `uart.c` 中的中断处理函数冲突，您还需要做下面的修改。把

`uart.c` 中 `PUBLIC void vUART_Init(void)` 的下面这段代码注释掉。

```
// /* Register function that will handle UART interrupts */
// #if UART == E_AHI_UART_0
//     vAHI_Uart0RegisterCallback(vUART_HandleUart0Interrupt);
// #else
//     vAHI_Uart1RegisterCallback(vUART_HandleUart1Interrupt);
```

```
// #endif
```

然后您就可以使用串口接收到的数据了。