1.Zigbee 协议栈相关概念

1.1 近距离通信技术比较：

近距离无线通信技术有 wifi、蓝牙、红外、zigbee，在无线传感网络中需求的网络通信恰是近距离需求的，故，四者均可用做无线传感网络的通信技术。而，其中（1）红外（infrared）：能够包含的信息过少；频率低波衍射性不好只能视距通信；要求位置固定；点对点传输无法组网。（2）蓝牙（bluetooth）：可移动，手机支持；通信距离 10m；芯片价格贵；高功耗（3）wifi：高带宽；覆盖半径 100m；高功耗；不能自组网；（4）zigbee:价格便宜；低功耗；自组网规模大。□□□□⇨WSN 中 zigbee 通信技术是最佳方案，但它连接公网需要有专门的网关转换→进一步学习 stm32。

1.2 协议栈

协议栈是网络中各层协议的总和，其形象的反映了一个网络中文件传输的过程：由上层协议到底层协议，再由底层协议到上层协议。

1.2.1Zigbee 协议规范与 zigbee 协议栈

Zigbee 各层协议中物理层（phy）、介质控制层（mac）规范由 IEEE802.15.4 规定，网络层（NWK）、应用层（apl）规范由 zigbee 联盟推出。Zigbee 联盟推出的整套 zigbee 规范：2005 年第一版 ZigBeeSpecificationV1.0，zigbee2006，zigbee2007、zigbeepro

zigbee 协议栈：很多公司都有自主研发的协议栈，如 TI 公司的：RemoTI，Z-Stack，SimpliciTI、freakz、msstatePAN 等。

1.2.2z-stack 协议栈与 zigbee 协议栈

z-stack 协议栈与 zigbee 协议栈的关系：z-stack 是 zigbee 协议栈的一种具体实现，或者说是 TI 公司读懂了 zigbee 协议栈，自己用 C 语言编写了一个软件—---z-stack，是由全球几千名工程师共同开发的。ZStack-CC2530-2.3.1-1.4.0 软件可与 TI 的 SmartRF05 平台协同工作，该平台包括 MSP430 超低功耗微控制器(MCU)、CC2520RF 收发器以及 CC2591 距离扩展器，通信连接距离可达数公里。

Z-Stack 中的很多关键的代码是以库文件的形式给出来，也就是我们只能用它们，而看不到它们的具体的实现。其中核心部分的代码都是编译好的，以库文件的形式给出的，比如安全模块，路由模块，和 Mesh 自组网模块。与 z-stack 相比 msstatePAN、freakz 协议栈都是全部真正的开源的，它们的所有源代码我们都可以看到。但是由于它们没有大的商业公司的支持，开发升级方面，性能方面和 z-stack 相比差距很大，并没有实现商业应用，只是作为学术研究而已。

还可以配备 TI 的一个标准兼容或专有的网络协议栈（RemoTI，Z-Stack，或 SimpliciTI）来简化开发,当网络节点要求不多在 30 个以内，通信距离 500m-1000m 时用 simpliciti。

1.2.3IEEE802.15.4 标准概述

IEEE802.15.4 是一个低速率无线个人局域网(LowRateWirelessPersonalAreaNetworks，LR-WPAN)标准。定义了物理层(PHY)和介质访问控制层(MAC)。

LR-WPAN 网络具有如下特点：

◆实现 250kb/s，40kb/s，20kb/s 三种传输速率。

◆支持星型或者点对点两种网络拓扑结构。

◆具有 16 位短地址或者 64 位扩展地址。

◆支持冲突避免载波多路侦听技术(carriersensemultipleaccesswithcollisionavoidance，CSMA/CA)。（mac 层）

◆用于可靠传输的全应答协议。(RTS-CTS)

◆低功耗。

◆能量检测(EnergyDetection，ED)。

◆链路质量指示(LinkQualityIndication，LQI)。

◆在 2.45GHz 频带内定义了 16 个通道；在 915MHz 频带内定义了 10 个通道；在 868MHz 频带内定义了 1 个通道。

为了使供应商能够提供最低可能功耗的设备，IEEE(InstituteofElectricalandElectronicsEngineers，电气及电子工程师学会)定义了两种不同类型的设备：一种是完整功能设备(full．functionaldevice，FFD)，另一种是简化功能设备

(reduced．functionaldevice，RFD)。

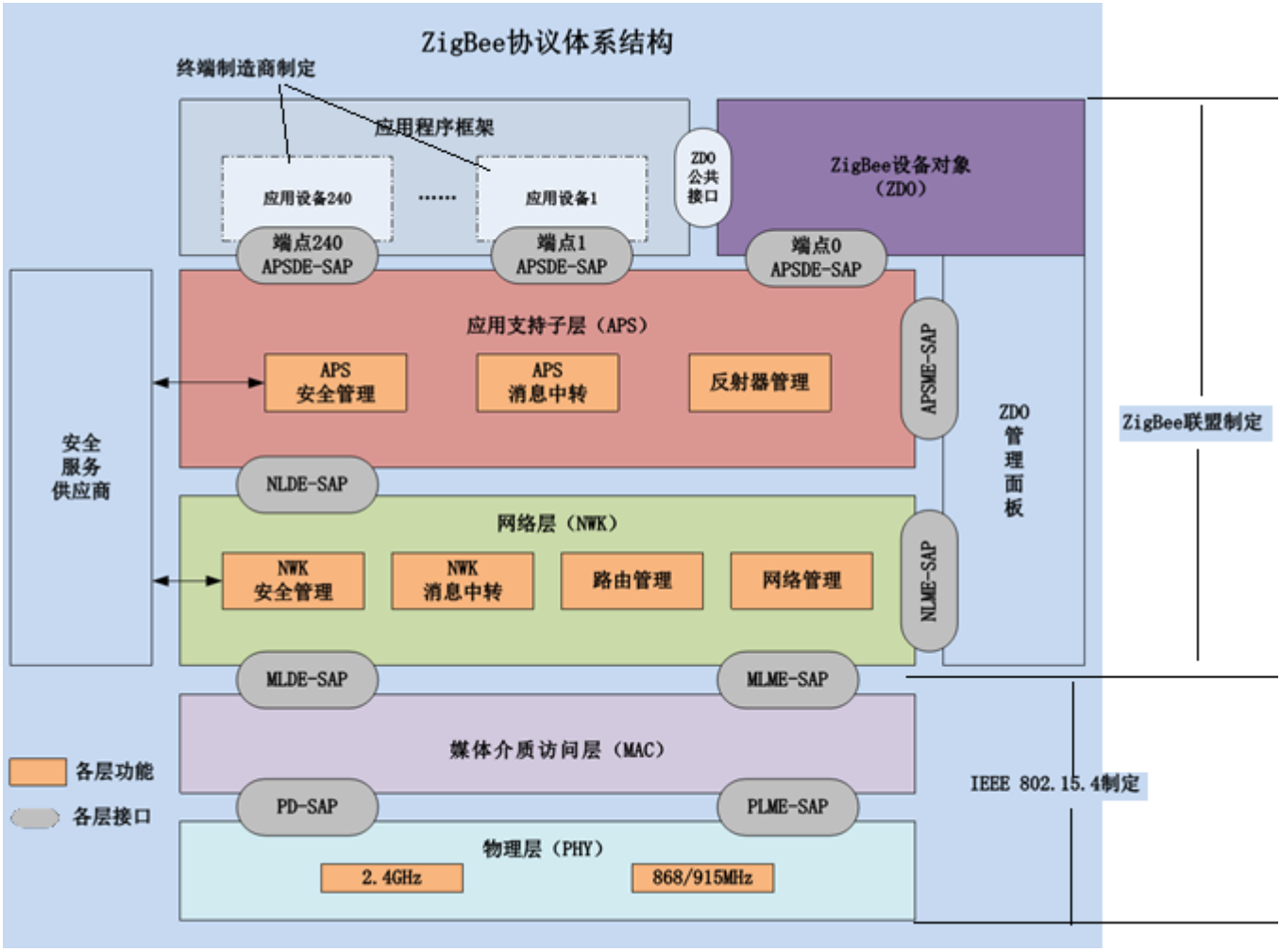
1.2.4ZigBee 协议体系结构

IEEE802.15.4 定义物理层、介质访问控制层

ZigBee 联盟定义网络层(networklayer，NWK)、应用层(applicationlayer，APL)。

应用层内定义：应用支持子层(applicationsupportsub—layer，APS)、ZigBee 设备对象(ZigBeeDeviceObject，ZDO)（端点号 0)、应用框架中用户自定义应用对象（端点号 1-240，可以定义 0-240 个应用)。

每一层为其上层提供特定的服务--数据服务实体→数据传输服务；管理实体提供→管理服务。每个服务实体通过相应的服务接入点(SAP)为其上层提供一个接口，每个服务接入点通过服务原语来完成所对应的功能。



1.2.4.1 物理层：

物理层定义了物理无线信道和 MAC 子层之间的接口，提供物理层数据服务和物理层管理服务。具体内容：

1)ZigBee 的激活；

2)当前信道的能量检测；

3)接收链路服务质量信息；

4)ZigBee 信道接入方式；

5)信道频率选择；

6)数据传输和接收。

1.2.4.2 介质接入控制子层（MAC）

MAC 层负责处理所有的物理无线信道访问，并产生网络信号、同步信号；支持 PAN 连接和分离，提供两个对等 MAC 实体之间可靠的链路。具体功能：

1）网络协调器产生信标；

2）与信标同步；

3）支持 PAN(个域网)链路的建立和断开；

4）为设备的安全性提供支持（加密解密功能）；

5）信道接入方式采用免冲突载波检测多址接入(CSMA-CA)机制；

6）处理和维护保护时隙(GTS)机制；

7）在两个对等的 MAC 实体之间提供一个可靠的通信链路。

1.2.4.3 网络层（NWK）

ZigBee 协议栈的核心部分在网络层。网络层主要实现节点加入或离开网络、接收或抛弃其他节点、路由查找及传送数据等功能。具体功能：

1)网络发现；（路由器、终端）

2)网络形成；（协调器）

3)允许设备连接；

4)路由器初始化；

5)设备同网络连接；

6)直接将设备同网络连接；

7)断开网络连接；

8)重新复位设备；

9)接收机同步；

10)信息库维护。

1.2.4.4 应用层（APL）

应用层包括：应用支持层(APS)、ZigBee 设备对象(ZDO)、制造商所定义的应用对象(AF)。

（1）APS 功能：维持绑定表、在绑定的设备之间传送消息。

（2）ZDO 功能：定义设备在网络中的角色(如物理实体节点被定义为协调器、路由器还是终端设备)，发起和响应绑定请求，在网络设备之间建立安全机制（加解密），发现网络中的设备并且决定向他们提供何种应用服务。

- ZDO 使用 APS 层的 APSDE-SAP 和网络层的 NLME-SAP。ZDO 是特殊的应用对象，它在端点(entire)0 上实现。
- 远程设备通过 ZDO 请求描述符信息，接收到这些请求时，ZDO 会调用配置对象获取相应描述符值（eg 设备什么时候出厂的、需不需要电池、传输距离多少、使用什么规范）。

（3）AF（应用程序框架）：用户自定义的应用对象，并且遵循规范（profile）运行在端点 1~240 上。在 ZigBee 应用中，提供 2 种标准服务类型：键值对（KVP）或报文（MSG）。

2.ZigBee 基本概念

2.1 设备类型

三种逻辑设备类型：协调器、路由器、终端设备。

节点类型	.cfg 配置文件
协调器	-DZDO_COORDINATOR -DRTR_NWK
路由器	-DRTR_NWK
终端设备	空

协调器的角色主要涉及网络的启动和配置。一旦这些都完成后，协调器的工作就像一个路由器(或者消失goaway)。由于 ZigBee 网络本身的分布特性，因此接下来整个网络的操作就不在依赖协调器是否存在。

路由器一直活跃，须使用主电源供电。但当树状拓扑结构时，允许其间隔一定的周期操作一次，可使用电池。

终端设备没有特定的维持网络结构的责任，可以睡眠或者唤醒，可用电池供电。对存储空间(特别是 RAM 的需要)比较小。

2.2 协议规范

协议栈规范由 ZigBee 联盟定义指定。在同一个网络中的设备必须符合同一个协议栈规范（同一个网络中所有设备的协议栈规范必须一致）。

ZigBee 联盟为 ZigBee 协议栈 2007 定义了 2 个规范：ZigBee 和 ZigBeePRO。所有的设备只要遵循该规范，即使在不同厂商买的不同设备同样可以形成网络。

如果应用开发者改变了规范，那么他的产品将不能与遵循 ZigBee 联盟定义规范的产品组成网络，也就是说该开发者开发的产品具有特殊性，我们称之为“关闭的网络”，也就是说它的设备只有在自己的产品中使用，不能与其他产品通信。更改后的规范可以称之为“特定网络”规范。

协议栈规范的 ID 号可以通过查询设备发送的 beacon 帧获得。在设备加入网络之前，首先需要确认协议栈规范的 ID。“特定网络”规范 ID 号为 0；ZigBee 协议栈规范的 ID 号为 1；ZigBeePRO 协议栈规范的 ID 号为 2。协议栈规范的 ID（STACK_PROFILE_ID）在 nwk_globals.h 中定义：

```
#defineNETWORK_SPECIFIC0
#defineHOME_CONTROLS1//zigbee 首先应用于智能家居，故直接把 zigbee 协议栈规范定义为 home_control
#defineZIGBEEPRO_PROFILE2
#defineGENERIC_STAR3
#defineGENERIC_TREE4
#ifdefined(ZIGBEEPRO)
    #defineSTACK_PROFILE_IDZIGBEEPRO_PROFILE
#else
     #defineSTACK_PROFILE_IDHOME_CONTROLS
#endif
```

2.3 拓扑结构

星型、树状、网状

```
#defineNWK_MODE_STAR0
#defineNWK_MODE_TREE1
#defineNWK_MODE_MESH2
#if(STACK_PROFILE_ID==ZIGBEEPRO_PROFILE)
    #defineNWK_MODENWK_MODE_MESH
#elif(STACK_PROFILE_ID==HOME_CONTROLS)
   #defineNWK_MODENWK_MODE_MESH
#elif(STACK_PROFILE_ID==GENERIC_STAR)
   #defineNWK_MODENWK_MODE_STAR
#elif(STACK_PROFILE_ID==NETWORK_SPECIFIC)
   #defineNWK_MODENWK_MODE_MESH
#endif
```

一般拓扑结构定义为网状网络

2.4 信标与非信标模式

Zigbee 网络的工作模式可以分为信标（Beaeon）和非信标（Non-beaeon）两种模式。

信标：所有设备同步工作、休眠。

协调器负责以一定的间隔时间（一般在 15ms-4mins 之间）向网络广播信标帧，两个信标帧发送间隔之间有 16 个相同的时槽，这些时槽分为网络休眠区和网络活动区两个部分，消息只能在网络活动区的各时槽内发送。

非信标：终端可休眠，路由器、协调器一直工作。

父节点为终端缓存数据，终端主动向父节点提取数据，故，终端大多处于休眠状态，周期性醒来与父节点握手以确认自己仍处于网络中，醒来一般需要 15ms。

实际使用中非信标模式使用更多，因为路由器、协调器往往还要担任一些其他功能，且常常加入功放扩大传输距离，一般加主电源供电。

2.5 地址

两种地址：64 位 IEEE 地址，即 MAC 地址，16 位网络地址（协调器网络地址为 0x00）。

#defineNWK_PAN_COORD_ADDR0x0000

2.5.1 网络地址分配

- 分布式寻址方案：ZigBee2006、ZigBee2007 使用分布式寻址方案来分配网络地址，保证唯一。

  设备只能从父设备接受网络地址，不需要全网通讯分配，有助于测量。

- 随机地址分配机制：ZigBee2007PRO 采用。

新节点加入时，父节点为其随机分配地址，然后产生“设备声明”（包含分配到的网络地址和 IEEE 地址）发送至网络。若有冲突，则通过路由器广播“网络状态-地址冲突”至网络中的所有节点。所有发生冲突的节点更改自己的网络地址，然后再发起“设备声明”直到无冲突。

在每个路由加入网络之前，寻址方案需要知道和配置一些参数：MAX_DEPTH（最大网络深度）、MAX_ROUTERS（最多路由数）和 MAX_CHILDREN（最多子节点数）。这些参数是栈配置的一部分，ZigBee2007 协议栈已经规定了这些参数的值：

```
#if ( STACK_PROFILE_ID == ZIGBEEPRO_PROFILE )
    #define MAX_NODE_DEPTH        20
#elif ( STACK_PROFILE_ID == HOME_CONTROLS )
    #define MAX_NODE_DEPTH        5
#elif ( STACK_PROFILE_ID == GENERIC_STAR )
    #define MAX_NODE_DEPTH        5
#elif ( STACK_PROFILE_ID == NETWORK_SPECIFIC )
#define MAX_NODE_DEPTH            5
#endif

#define NWK_MAX_ROUTERS           6

#define NWK_MAX_DEVICES           21
```

如果要改动，首先要确保整个地址空间不能超过 2^{16}，这就限制了参数能够设置的最大值。可以使用 projects\ZStack\tools 文件夹下的 CSkip.xls 文件来确认这些值是否合法。当在表格中输入了这些数据后，如果你的数据不合法的话就会出现错误信息。

当选择了合法的数据后，开发人员还要保证不再使用标准的栈配置，取而代之的是网络自定义栈配置(例如：在 nwk_globals.h 文件中将 STACK_PROFILE_ID 改为 NETWORK_SPECIFIC)。

然后 nwk_globals.h 文件中的 MAX_DEPTH 参数将被设置为合适的值。

此外，还必须设置 nwk_globals.c 文件中的 Cskipchldrn 数组和 CskipRtrs 数组。这些数组的值由 MAX_CHILDREN 和 MAX_ROUTER 构成。

```
#if ( STACK_PROFILE_ID == ZIGBEEPRO_PROFILE )
  uint8 CskipRtrs[1] = {0};
  uint8 CskipChldrn[1] = {0};
#elif ( STACK_PROFILE_ID == HOME_CONTROLS )
  uint8 CskipRtrs[MAX_NODE_DEPTH+1] = {6,6,6,6,6,0};
  uint8 CskipChldrn[MAX_NODE_DEPTH+1] = {20,20,20,20,20,0};
#elif ( STACK_PROFILE_ID == GENERIC_STAR )
  uint8 CskipRtrs[MAX_NODE_DEPTH+1] = {5,5,5,5,5,0};
  uint8 CskipChldrn[MAX_NODE_DEPTH+1] = {5,5,5,5,5,0};
#elif ( STACK_PROFILE_ID == NETWORK_SPECIFIC )
  uint8 CskipRtrs[MAX_NODE_DEPTH+1] = {5,5,5,5,5,0};
  uint8 CskipChldrn[MAX_NODE_DEPTH+1] = {5,5,5,5,5,0};
#endif // STACK_PROFILE_ID
```

2.5.2 寻址

应用程序通常使用 AF_DataRequest()函数发送数据，给一个 afAddrType_t(在 ZComDef.h 中定义)类型的目标设备。

```
typedef struct
{
  union
  {
    uint16        shortAddr;
    ZLongAddr_t extAddr;
  } addr;
  afAddrMode_t addrMode;
  byte endPoint;
  uint16 panId;   // used for the INTER_PAN feature
} afAddrType_t;
```

目标设备网络地址

```
typedef enum
{
  afAddrNotPresent = AddrNotPresent,
  afAddr16Bit         = Addr16Bit,
  afAddr64Bit         = Addr64Bit,
  afAddrGroup          = AddrGroup,
  afAddrBroadcast    = AddrBroadcast
} afAddrMode_t;
```

目标设备模式

数据包传送模式分三种：

（1） 单点传送(Unicast)：发给已知网络地址的网络设备。将 afAddrMode 设置为 Addr16Bit 并且在数据包中携带目标设备地址。

（2） 间接传送(Indirect)：应用程序不知道目标设备位置时。将模式设置为 AddrNotPresent，目标地址没有指定，而是从发送设备的栈的绑定表中查找目标设备。这种特点称之为源绑定。数据到达栈中后，从绑定表中查找并使用目标设备地址，这样，数据包将被处理成为一个标准的单点传送数据包。如果在绑定表中找到多个设备，则向每个设备都发送一个数据包的拷贝。【ZigBee2004 中有一个选项可以讲绑定表保存在协

调器(Coordinator)当中。发送设备将数据包发送给协调器，协调器查找它栈中的绑定表，然后将数据发送给最终的目标设备。这个附加的特性叫做协调器绑定(CoordinatorBinding)。2004 中绑定表只能在协调器，2006 和 2007 中可以将绑定表放在任意节点】

（3） 广播传送(broadcast)：应用程序将数据包发给每一个设备。地址模式设置为 AddrBroadcast。目标地址可以设置为下面广播地址的一种：❶NWK_BROADCAST_SHORTADDR_DEVALL(0xFFFF)——发给所有设备，包括睡眠中的设备。对于睡眠中的设备，数据包将被保留在其父亲节点直到查询到它，或者消息超时(NWK_INDIRECT_MSG_TIMEOUT 在 f8wConifg.cfg 中)。

❷NWK_BROADCAST_SHORTADDR_DEVRXON(0xFFFD)——发给所有除睡眠之外的所有设备。

❸NWK_BROADCAST_SHORTADDR_DEVZCZR(0xFFFC)——发给所有路由器、协调器。

（4） 组寻址(GroupAddressing)：发送给网络上的一组设备。地址模式设置为 afAddrGroup 并且 addr.shortAddr 设置为组 ID。使用这个功能，必须先定义组。(参见 Z-stackAPI 文档中的 aps_AddGroup()函数)。

下面的代码是一个设备怎样加入到一个 ID 为 1 的组当中：

```
aps_Group_t group;
// Assign yourself to group 1
group.ID = 0x0001;
group.name[0] = 0; // This could be a human readable string
aps_AddGroup( SAMPLEAPP_ENDPOINT, &group );
```

2.5.3 重要设备地址(ImportantDeviceAddresses)

应用程序可能需要知道它的设备地址和父亲地址。使用下面的函数获取设备地址(在 ZStackAPI 中定义)：

NLME_GetShortAddr()——返回本设备的 16 位网络地址

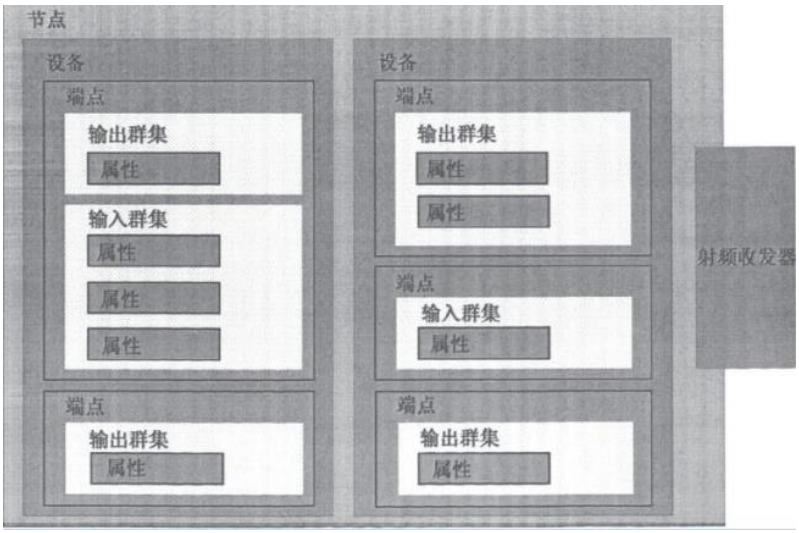
NLME_GetExtAddr()——返回本设备的 64 位扩展地址

使用下面的函数获取该设备的父亲设备的地址：

NLME_GetCoordShortAddr()——返回本设备的父亲设备的 16 位网络地址

NLME_GetCoordExtAddr()——返回本设备的父亲设备的 64 位扩展地址

2.6zigbee 术语



一个节点：一个射频收发器、多个设备（最常见的为仅一个），

一个设备中包含端点 0（ZDO），端点 1-端点 240

一个端点中包含群集（按功能分为输入群集、输出群集）

一个群集中包含多个属性

（1） 属性 Attribute：是一个反映物理数量或状态的数据值，比如开关值（On/Off）,温度值、百分比等。

（2） 群集 Cluster：是包含一个或多个属性（attribute）的集合。每个群集都被分配一个唯一的群集 ID 且每个群集最多有 65536 个属性。

（3） 端点 EndPoint：是协议栈应用层的入口，也可以理解应用对象（ApplicationObject）存在的地方，它是为实现一个设备描述而定义的一组群集。每个 ZigBee 设备可以最多支持 240 这样的端点，这也意味着在每个设备上可以定义 240 个应用对象。端点 0 被保留用于与 ZDO 接口，这是每个 ZigBee 设备必须使用的端点，而端点 255 被保留用于广播，端点 241-254 则被保留用于将来做扩展使用

（4） 设备描述 DeviceDescription：是指一个大型目标应用的一部分，包括一个或多个群集，并且指定群集是输入还是输出。描述符有：节点描述符、电源描述符、简单描述符、端点描述符

端点描述符：

```
typedef struct
{
  byte endPoint;
  byte *task_id;   // Pointer to location of the Application task ID.
  SimpleDescriptionFormat_t *simpleDesc;
  afNetworkLatencyReq_t latencyReq;
} endPointDesc_t;
```

简单描述符：

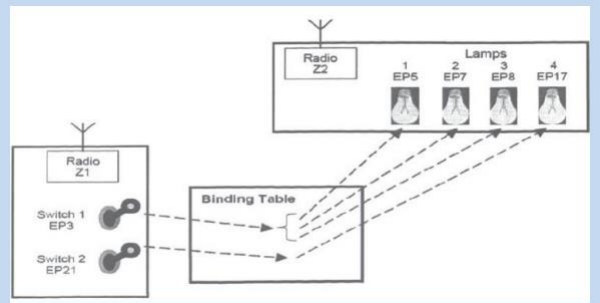
```
typedef struct
{
  byte                EndPoint;
  uint16              AppProfId;
  uint16              AppDeviceId;
  byte                AppDevVer:4;
  byte                Reserved:4;              // AF_V1_SUPPORT uses for AppFlags:4.
  byte                AppNumInClusters;
  cId_t               *pAppInClusterList;
  byte                AppNumOutClusters;
  cId_t               *pAppOutClusterList;
} SimpleDescriptionFormat_t;
```

。

（5）节点 Node：也可以理解为一个容器，包含一组 ZigBee 设备，分享一个无线信道。每个节点有且只有一个无线信道使用。

2.7 绑定

通过使用 ClusterID 为不同节点上的独立端点建立一个逻辑上的连接。



要实现绑定操作，端点必须向协调器发送绑定请求，协调器在有限的时间间隔内接收到两个端点的绑定请求后，便通过建立端点之间的绑定表在这两个不同的端点之间形成了一个逻辑链路。

因此，在绑定后的两个端点之间进行消息传送的过程属于消息的间接传送。其中一个端点首先会将信息发送到 ZigBee 协调器中，ZigBee 协调器在接收到消息后会通过查找绑定表，将消息发送到与这个端点相绑定的所有端点中，从而实现了绑定端点之间的间接通信。

也可以不在协调器中建立绑定表，而在任意一个路由器中建立。

2.8 路由

Zstack 协议栈的路由代码我们不需要写，也无法写（已被封装）。整个路由功能如何实现的需要了解，有助于对 zigbee

协议栈进行整体的把握。

zigbee 路由还能够自愈网络，如果某个无线连接断开了，路由功能又能自动寻找一条新的路径避开那个断开的网络连接。这就极大的提高了网络的可靠性，同时也是 ZigBee 网络的一个关键特性。

2.8.1 路由协议(Routing Protocol

ZigBee 执行基于用于 AODV 专用网络的路由协议。简化后用于传感器网络。ZigBee 路由议有助于网络环境有能力支持移动节点，连接失败和数据包丢失。

当路由器从他自身的应用程序或者别的设备那里收到一个单点发送的数据包，则网络层 NWK Layer)根据一下程序将它继续传递下去。如果目标节点是它相邻路由器中的一个，则数据包直接被传送给目标设备。否则，路由器将要检索它的路由表中与所要传送的数据包的目标地址相符合的记录。如果存在与目标地址相符合的活动路由记录，则数据包将被发送到存储在记录中的下一级地址中去。如果没有发现任何相关的路由记录，则路由器发起路径寻找，数据包存储在缓冲区中知道路径寻找结束。

ZigBee 终端节点不执行任何路由功能。终端节点要向任何一个设备传送数据包，它只需简单的将数据向上发送给它的父亲设备，由它的父亲设备以它自己的名义执行路由。同样的，任何一个设备要给终端节点发送数据，发起路由寻找，终端节的的父亲节点都已它的名义来回应。

注意 ZigBee 地址分配方案使得对于任何一个目标设备，根据它的地址都可以得到一条路径。在 Z-Stack 中，如果万一正常的路径寻找过程不能启动的话(通常由于缺少路由表空间)，那么 Z-Stack 拥有自动回退机制。

此外，在 Z-Stack 中，执行的路由已经优化了路由表记录。通常，每一个目标设备都需要一条路由表记录。但是，通过把一定父亲节点记录与其子所有子结点的记录合并，这样既可以优化路径也可以不丧失任何功能。

ZigBee 路由器，包括协调器执行下面的路由函数：

(i)路径发现和选择；

(ii)路径保持维护；

(iii)路径期满。

2.8.2 路径的发现和选择(Route Discovery and Selection)

路径发现是网络设备凭借网络相互协作发现和建立路径的一个过程。路由发现可以由任意一个路由设备发起，并且对于某个特定的目标设备一直执行。路径发现机制寻找源地址和目标地址之间的所有路径，并且试图选择可能的最好的路径。

路径选择就是选择出可能的最小成本的路径。每一个结点通常持有跟它所有邻接点的“连接成本(link costs)”。通常，连接成本的典型函数是接收到的信号的强度。沿着路径，求出所有连接的连接成本总和，便可以得到整个路径的“路径成本”。路由算法试图寻找到拥有最小路径成本的路径。

路径通过一系列的请求和回复数据包被发现。源设备通过向它的所有邻接节点广播一个路由请求数据包，来请求一个目标地址的路径。当一个节点接收到 RREQ 数据包，它依次转发 REQ 数据包。但是在转发之前，它要加上最新的连接成本，然后更新 RREQ 数据包中的成本值。这样，沿着所有它通过的连接，RREQ 数据包携带着连接成本的总和。这个过程一直持续到 RREQ 数据包到达目标设备。通过不同的路由器，许多 RREQ 副本都将到达目标设备。目标设备选择最好的 RREQ 数据包，然后发回一个路径答复数据包(a Route Reply)RREP 给源设备.RREP 数据包是一个单点发送数据包，它沿着中间节点的相反路径传送直到它到达原来发送请求的节点为止。

一旦一条路径被创建，数据包就可以发送了。当一个结点与它的下一级相邻节点失去了连接(当它发送数据时，没有收到 MAC ACK)，该节点向所有等待接收它的 RREQ 数据包的节点发送一个 RERR 数据包，将它的路径设为无效。各个结点根据收到的数据包 RREQ、RREP 或者 RERR 来更新它的路由表。

2.8.3 路径保持维护(Route maintenance)

网状网提供路径维护和网络自愈功能。中间节点沿着连接跟踪传送失败，如果一个连接被认定是坏链，那么上游节点将针对所有使用这条连接的路径启动路径修复。节点发起重新发现直到下一次数据包到达该节点，标志路径修复完成。如果不能够启动路径发现或者由于某种原因失败了，节点则向数据包的源节点发送一个路径错误包(RERR)，它将负责启动新路径的发现。这两种方法，路径都自动重建。

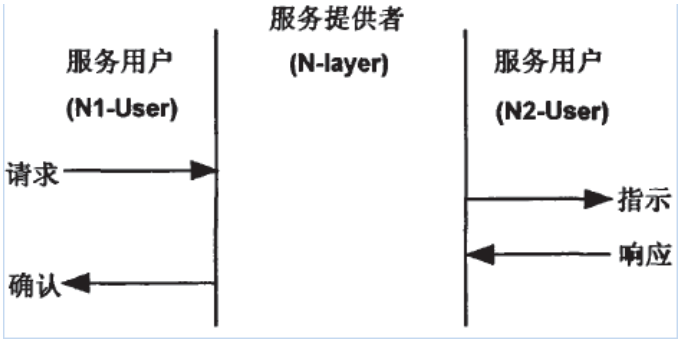
2.8.4 路径期满(Route expiry)

路由表为已经建立连接路径的节点维护路径记录。如果在一定的时间周期内，没有数据通过沿着这条路径发送，这条路径将被表示为期满。期满的路径一直保留到它所占用的空间要被使用为止。这样，路径在绝对不使用之前不会被删除掉的。

在配置文件 f8wConfig.cfg 文件中配置自动路径期满时间。设置 ROUTE_EXPIRY_TIME 为期满时间，单位为秒。如果设置为 0，则表示关闭自动期满功能。

2.9zigbee 原语

ZigBee 协议按照开放系统互联的 7 层模型将协议分成了一系列的层结构，各层之间通过相应的服务访问点来提供服务。这样使得处于协议中的不同层能够根据各自的功能进行独立的运作，从而使整个协议栈的结构变得清晰明朗。另一方面，由于 ZigBee 协议栈是一个有机的整体，任何 ZigBee 设备要能够正确无误的工作，就要求协议栈各层之间共同协作。因此，层与层之间的信息交互就显得十分重要。ZigBee 协议为了实现层与层之间的关联，采用了称为服务“原语"的操作。利用下图来说明原语操作的概念。



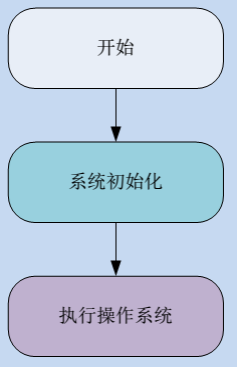
服务由 N 用户和 N 层之间信息流的描述来指定。这个信息流由离散瞬时事件构成，以提供服务的特征。每个事件由服务原语组成，它将在一个用户的某一 层，通过与该层相关联的层服务访问 A(SAP)与建立对等连接的用户的相同层之 间传送。层与层之间的原语一般情况下可以分为 4 种类型：

- 请求：请求原语从 NI 用户发送到它的 N 层，请求发起一个服务。
- 指示：指示原语从 N 层到 N2 用户，指示一个对 N2 用户有重要意义外部 N 层事件。这个事件可能与一个远程的服务请求有关，或者由内部事件产生。
- 响应：响应原语由 N2 用户向它的 N 层传递，用来响应上一个由指示原语引 起的过程。
- 确认：确认原语由 N 层向 NI 用户传递，用来传递与前面一个或多个服务请 求相关的执行结果。

3.Z-Stack 协议栈总体设计

Chipcon 公司为自己设计的 Z-Stack 协议栈中提供了一个名为操作系统抽象层 OSAL 的协议栈调度程序。对于用户来说，除了能够看到这个调度程序外，其它任何协议栈操作的具体实现细节都被封装在库代码中。用户在进行具体的应用开发时只能够通过调用 API 接口来进行，而无权知道 Zig,Bee 协议栈实现的具体细节。(即物理层、mac 层、网络层封装到库函数中，无法查看；应用层的 aps、zdo 代码完全开源)

Z-Stack 由 main（）函数开始执行， main（）函数共做了 2 件事：一是系统初始化，一是轮转查询式操作系统，如下图所示：



Osal 初始化完成后，一经进入操作系统则永远无休止执行下去，除非板子坏掉。

3.1osal 初始化：

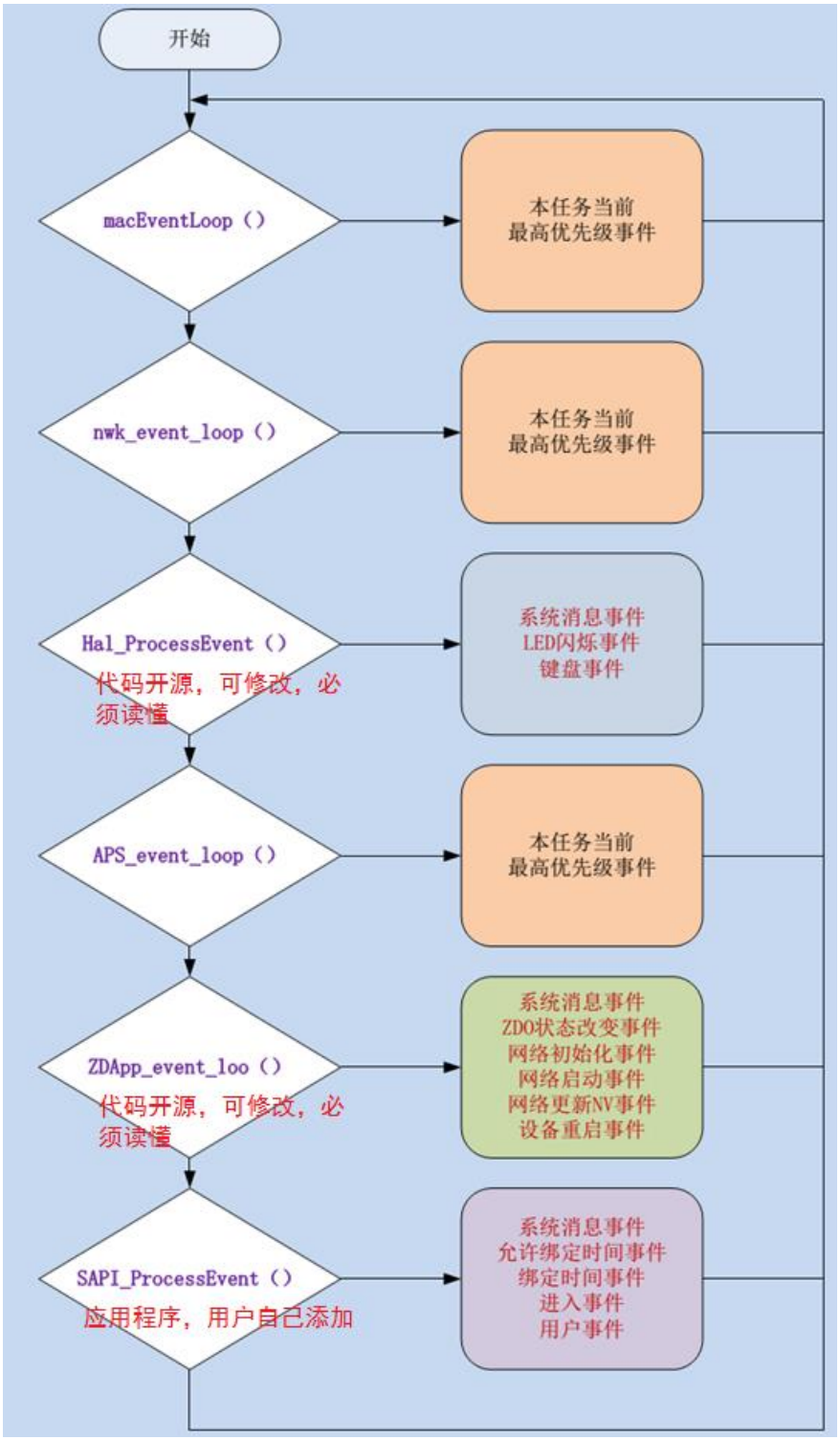
```
void osalInitTasks( void )
{
  uint8 taskID = 0;

  tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
  osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));

  macTaskInit( taskID++ );
  nwk_init( taskID++ );
  Hal_Init( taskID++ );
#if defined( MT_TASK )//通过串口调试的一个任务
  MT_TaskInit( taskID++ );
#endif
  APS_Init( taskID++ );
#if defined ( ZIGBEE_FRAGMENTATION )//如果定义了数据分割的分包，则执行分包初始化
  APSF_Init( taskID++ );
#endif
  ZDApp_Init( taskID++ );
#if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )//如果定义了调频初始化，和个域网 id 冲突初始化，则执行网络管理层初始化
  ZDNwkMgr_Init( taskID++ );
#endif
  SampleApp_Init( taskID );//应用对象初始化
}
```

3.2 任务调度

ZigBee 协议栈中的每一层都有很多原语操作要执行，因此对于整个协议栈来说，就会有很多并发操作要执行。协议栈中的每一层都设计了一个事件处理函数，用来处理与这一层操作相关的各种事件。将这些事件处理函数看成是与协议栈每一层相对应的任务，由 ZigBee 协议栈中调度程序 OSAL 来进行管理。这样，对于协议栈来说，无论何时发生了何种事件，我们都可以通过调度协议栈相应层的任务，即事件处理函数来进行处理。这样，整个协议栈便会按照时间顺序有条不紊的运行。



ZigBee 协议栈的实时性要求并不高，因此在设计任务调度程序时， OSAL 只采用了轮询任务调度队列的方法来进行任务调度管理。

任务列表：

```
    const pTaskEventHandlerFn tasksArr[] = {
      macEventLoop,
      nwk_event_loop,
      Hal_ProcessEvent,
    #if defined( MT_TASK )
      MT_ProcessEvent,
    #endif
      APS_event_loop,
    #if defined ( ZIGBEE_FRAGMENTATION )
      APSF_ProcessEvent,
    #endif
      ZDApp_event_loop,
    #if defined ( ZIGBEE_FREQ_AGILITY ) || defined
( ZIGBEE_PANID_CONFLICT )
```

- 任务调度主循环

```
void osal_start_system( void )
{
  for(;;)  // Forever Loop
{uint8 idx = 0;

  osalTimeUpdate();
  Hal_ProcessPoll();  // This replaces MT_SerialPoll() and osal_check_timer().

  do {
    if (tasksEvents[idx])  // Task is highest priority that is ready.
    {
      break;
    }
  } while (++idx < tasksCnt);

  if (idx < tasksCnt)
  {
    uint16 events;
    halIntState_t intState;

    HAL_ENTER_CRITICAL_SECTION(intState);
    events = tasksEvents[idx];
    tasksEvents[idx] = 0;  // Clear the Events for this task.
    HAL_EXIT_CRITICAL_SECTION(intState);

    events = (tasksArr[idx])( idx, events );

    HAL_ENTER_CRITICAL_SECTION(intState);
    tasksEvents[idx] |= events;  // Add back unprocessed events to the current task.
    HAL_EXIT_CRITICAL_SECTION(intState);
  }
#if defined( POWER_SAVING )
  else  // Complete pass through all task events with no activity?
  {
    osal_pwrmgr_powerconserve();  // Put the processor/system into sleep
  }
#endif

  /* Yield in case cooperative scheduling is being used. */
#if defined (configUSE_PREEMPTION) && (configUSE_PREEMPTION == 0)
  {
    osal_task_yield();
  }
#endif
}
}
```

设置事情发生标志

当协议栈中有任何事件发生时，我们可以通过设路 event_flag 来标记有事件发生，以便主循环函数能够及时加以处理。其函数说明如下：

3.3 时间管理

假设 A 事件发生后要求 l0 秒之后执行，B 事件在 A 事件发生 1 秒后产生，且 B 事件要求 5 秒后执行。为了按照合理的时间顺序来处理不同事件的执行，这就需要对各种不同的事件进行时间管理。OSAL 调度程序设计了与时间管理相关的函数，用来管理各种不同的要被处理的事件。

对事件进行时间管理， OSAL 也采用了链表的方式进行，每当发生一个要被处理的事件后，就启动一个逻辑上的定时器，并将此定时器添加到链表之中。利用硬件定时器作为时间操作的基本单元。设路时间操作的最小精度为 lms，每 lms 硬件定时器便产生一个时间中断，在时间中断处理程序中去更新定时器链表。每次更新，就将链表中的每一项时间计数减 I，如果发现定时器链表中有某一表项时间计数已经减到 0，则将这个定时器从链表中删除，并设路相应的事件标志。这样任务调度程序便可以根据事件标志进行相应的事件处理。根据这种思路，来自协议栈中的任何事件都可以按照时间顺序得到处理。从而提高了协议栈设计的灵活性。

在设计过程中需要经常使用这样一个时间管理函数，其函数说明如下：

```
uint8 osal_start_timerEx( uint8 taskID, uint16 event_id, uint16 timeout_value )
{
    halIntState_t intState;
    osalTimerRec_t *newTimer;

    HAL_ENTER_CRITICAL_SECTION( intState );    // Hold off interrupts.

    // Add timer
    newTimer = osalAddTimer( taskID, event_id, timeout_value );

    HAL_EXIT_CRITICAL_SECTION( intState );     // Re-enable interrupts.

    return ( (newTimer != NULL) ? SUCCESS : NO_TIMER_AVAIL );
}
```

这个函数为事件 event id 设路超时等待时间 timeout value。一旦等待结束，便为 task id 所对应的任务设路相应的事件发生标记，从而达到对事件进行延迟处理的目的。

3.4 原语通信

请求(Request)、响应(Response)原语分别由协议栈中处于较高位路的层向较低层发起；可直接使用函数调用实现

确认(Confirm)、指示(Indication)原语则从较低层向较高层返回结果或信息；需要间接处理的机制来完成。

在设计与请求、响应原语操作相对应的函数时，一旦调用了下层相关函数后，就立即返回。下层处理函数在操作结束后，将结果以消息的形式发送到上层并产生一个系统事件，调度程序发现这个事件后就会调用相应的事件处理函数对它进行处理。OSAL 调度程序设计了两个相关的函数来完成这个过程。

3.4.1 向目标任务发送消息的函数

这个函数主要用来将原语操作的结果以消息的形式向上层任务发送，并产生一个系统事件来通知调度程序。其函数说明如下：

```
uint8 osal_msg_send( uint8 destination_task, uint8 *msg_ptr )
{
  if ( msg_ptr == NULL )
    return ( INVALID_MSG_POINTER );

  if ( destination_task >= tasksCnt )
  {
    osal_msg_deallocate( msg_ptr );
    return ( INVALID_TASK );
  }

  // Check the message header
  if ( OSAL_MSG_NEXT( msg_ptr ) != NULL ||
       OSAL_MSG_ID( msg_ptr ) != TASK_NO_TASK )
  {
    osal_msg_deallocate( msg_ptr );
    return ( INVALID_MSG_POINTER );
  }

  OSAL_MSG_ID( msg_ptr ) = destination_task;

  // queue message
  osal_msg_enqueue( &osal_qHead, msg_ptr );

  // Signal the task that a message is waiting
  osal_set_event( destination_task, SYS_EVENT_MSG );

  return ( SUCCESS );
}
```

3.4.2 消息提取函数

这个函数用来从内存空间中提取相应的消息。

```
uint8 *osal_msg_receive( uint8 task_id )
{
  osal_msg_hdr_t *listHdr;
  osal_msg_hdr_t *prevHdr = NULL;
  osal_msg_hdr_t *foundHdr = NULL;
  halIntState_t     intState;

  // Hold off interrupts
  HAL_ENTER_CRITICAL_SECTION(intState);

  // Point to the top of the queue
  listHdr = osal_qHead;

  // Look through the queue for a message that belongs to the asking task
  while ( listHdr != NULL )
  {
    if ( (listHdr - 1)->dest_id == task_id )
    {
      if ( foundHdr == NULL )
      {
        // Save the first one
        foundHdr = listHdr;
      }
```

```
        else
        {
          // Second msg found, stop looking
          break;
        }
      }
      if ( foundHdr == NULL )
      {
        prevHdr = listHdr;
      }
      listHdr = OSAL_MSG_NEXT( listHdr );
    }

    // Is there more than one?
    if ( listHdr != NULL )
    {
      // Yes, Signal the task that a message is waiting
      osal_set_event( task_id, SYS_EVENT_MSG );
    }
    else
    {
      // No more
      osal_clear_event( task_id, SYS_EVENT_MSG );
    }

    // Did we find a message?
    if ( foundHdr != NULL )
    {
      // Take out of the link list
      osal_msg_extract( &osal_qHead, foundHdr, prevHdr );
    }

    // Release interrupts
    HAL_EXIT_CRITICAL_SECTION(intState);

    return ( (uint8*) foundHdr );
}
```