

1. 协议栈工作流程和无线收发控制 LED

提示：做此章实验请先接好天线哦，至少接一条天线，因为无线是有阻抗的，设计时有天线就一定要接的，不接天线的请使用板载天线的核心板。

实验内容：

1. ZigBee 协议栈简介
2. ZigBee 基本概念
3. 如何使用 ZigBee 协议栈
4. ZigBee 协议栈的安装、编译与下载
5. 协议栈工作流程
6. 无线收发控制 LED 代码分析与步骤

实现现象：

协调器、终端上电，组网成功后 D1 灯闪烁

前言

前面讲了部分理论和基础实验都是为这章做铺垫的，整个学习中本章的实验是重中之重，也是我们以后实际开发中必须用到的。本套开发板以实战为主，用户可非常简单、方便的应用到实际产品中，本公司开发产品也是基于此开发板开发的，去掉了一些华而不实的东西，如 5 向按键、增加 TFT 显示、增加串口收发指示灯等等，用我们的开发板省去自己研究、裁剪没用的资源和代码，提高开发和学习效率。

ZigBee 无线传感器网络涉及电子、电路、通信、射频等多学科的知识，这对于入门级学习来说，无形中增加了学习难度，很多读者看协议、射频电路……学了半年甚至更长的时间，但是连基本的点对点通信都无法实现，更别说 ZigBee 网络应用了。基于此原因，本章采用一种新学习思路，快速帮大家理解、使用 ZigBee 协议栈。先进行 ZigBee 无线网络通信的学习和实验，有了感性认知后再看 ZigBee 协议栈视频、射频、天线等知识，看完后再实验一遍加深印象，用此法会起到事半功倍的效果，经过多名初学者学习所积累总结出的经验，希望对您有所帮助。

一、ZigBee 协议栈简介

物联网是新一代信息技术的重要组成部分，也是“信息化”时代的重要发展阶段。顾名思义，物联网就是物物相连的互联网。这有两层意思：其一，物联网的核心和基础仍然是互联网，是在互联网基础上的延伸和扩展的网络；其二，其用户端延伸和扩展到了任何物品与物品之间，进行信息交换和通信，也就是物物相息。物联网是指通过各种信息传感设备，实时采集任何需要监控、连接、互动的物体或过程等各种需要的信息，与互联网结合形成的一个巨大网络。其目的是实现物与物、物与人，所有的物品与网络的连接，方便识别、管理和控制；ZigBee 模块是一种物联网无线数据终端，利用 ZigBee 网络为用户提供[无线数据传输](#)功能。

无线传感网络的定义是：大规模、无线、自组织、多跳、无分区、无基础设施支持的网络。其中的节点是同构的、成本较低、体积较小，大部分节点不移动，被随意分布在工作区域，要求网络系统有尽可能长的工作时间。在通信方式上，虽然可以采用有线、无线、红外和光等多种形式，但一般认为短距离的无线低功率通信技术最适合传感器网络使用，为明确起见，一般称无线传感器网络(WSN . Wireless Sensor Network)。

Zigbee 是 IEEE 802.15.4 协议的代名词。根据这个协议规定的技术是一种短距离、低功耗的无线通信技术。这一名称来源于蜜蜂的八字舞，由于蜜蜂(bee)是靠飞翔和“嗡嗡”(zig)地抖动翅膀的“舞蹈”来与同伴传递花粉所在方位信息，也就是说蜜蜂依靠这样的方式构成了群体中的通信网络。其特点是近距离、低复杂度、自组织、低功耗、低数据速率、低成本。主要适用于自动控制和远程控制领域，可以嵌入各种设备。简而言之，ZigBee 就是一种便宜的，低功耗的近距离无线组网通讯技术。

无线传感网络的无线通信技术可以采用 ZigBee 技术、蓝牙、Wi-Fi 和红外等技术。ZigBee 技术是一种短距离、低复杂度、低功耗、低数据速率、低成本的双向无线通信技术或无线网络技术，是一组基于 IEEE802.15.4 无线标准研制开发的组网、安全和应用软件方面的通信技术。

协议栈是指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。使用最广泛的是英特网协议栈，由上到下的协议分别是：应用层 (HTTP , TELNET , DNS , EMAIL 等) , 传输层 (TCP , UDP) , 网络层 (IP) , 链路层 (WI-FI , 以太网 , 令牌环 , FDDI 等) , 物理层。

ZigBee 联盟于 2005 年公布了第一份 ZigBee 规范 “ZigBee Specification V1.0” 。ZigBee 协议规范使用了 IEEE 802.15.4 定义的物理层 (PHY) 和媒体介质访问层 (MAC) , 并在此基础上定义了网络层 (NWK) 和应用层 (APL) 架构。

ZigBee2007/PRO 无线传感器网络与 ZigBee2006 无线传感器网络相比最大区别在于其支持最新 ZigBee2007/PRO 网络，提供更多更精确传感器(如增加高精度温湿度数字传感器等)，提供更多可扩展接口，提供更大网络支持，速度更快/处理能力更强低功耗微控制器等。

ZigBee 的技术特性决定它将是无线传感器网络的最好选择，广泛用于物联网，自动控制和监视等诸多领域。以美国德州仪器 TI 公司 CC2530 芯片为代表的 Zigbee SOC 解决方案在国内高校企业掀起了一股 Zigbee 技术应用的热潮。CC2530 集成了 51 单片机内核，相比于众多的 Zigbee 芯片，CC2530 颇受青睐。

ZigBee 新一代 SOC 芯片 CC2530 是真正的片上系统解决方案，支持 IEEE 802.15.4 标准 ZigBee/ZigBee RF4CE 和能源的应用。拥有庞大的快闪记忆体多达 256 个字节，CC2530 是理想 ZigBee 专业应用。CC2530 集成高性能的 RF 收发器与一个 8051 微处理器，8 kB 的 RAM，32/64/128/256 KB 闪存，以及其他强大的支持功能和外设。

CC2530 提供了 101dB 的链路质量，优秀的接收器灵敏度和健壮的抗干扰性，四种供电模式，多种闪存尺寸，以及一套广泛的外设集——包括 2 个 USART、12 位 ADC 和 21 个通用 GPIO，以及更多。除了通过优秀的 RF 性能、选择性和业界标准增强

8051MCU 内核,支持一般的低 功耗无线通信,CC2530 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,使你更快的获得市场。CC2530 可以用于的应用包括远程控制、消 费型电子、家庭控制、计量和智能能源、楼宇自动化、医疗以及更多领域。

我们开发板主要使用德州仪器 (TI) ZStack-CC2530-2.5.1a 的协议栈。

TI 推出了最丰富、最完整的 ZigBee 系列产品,包括收发器、片上系统以及新近推出的 2.4GHz 距离扩展器 (CC2592)。该系列产品具有无与伦比的高性能、高灵活性与定制功能,从而有助于客户提供特色化设计方案。TI 还为其不断丰富的低功耗 RF 系列产品提供一流的软件、工具、应用知识及全球技术支持,帮助 ZigBee 设计人员在市场中取得成功。

Z-Stack 软件因其出色的 ZigBee 与 ZigBee PRO 特性集被 ZigBee 测试机构国家技术服 务公司 (NTS) 评为 ZigBee 联盟最高业内水平,目前该软件已为全球数以千计的 ZigBee 开发人员广泛采用。

Z-Stack 是在 2007 年 4 月,德州仪器推出业界领先的 ZigBee 协议栈,Z-Stack 符合 ZigBee 2006 规范,支持多种平台,Z-Stack 包含了网状网络拓扑的几近于全功能的协议栈,在竞争激烈的 ZigBee 领域占有重要地位。配合 OSAL 完成整个协议栈的运行。

Z-Stack 只是 ZigBee 协议的一种具体的实现,Z-Stack 是其中一种,也不能把 Z-Stack 等同于 ZigBee 协议,市场还有以下系统:

1、freakz 协议栈和 contiki 操作系统。

freakz 是一个彻底的开源 zigbee 协议,而 contiki 也是一个彻底的开源操作系统,而且这个操作系统短小精悍,非常适合“物联网”时代的 MINI 型设备,同时,这套系统在全球已经拥有了众多的支持与使用者,已经开发了非常多的应用,甚至有像 IPV6 这么强大而且的应用,可以在其官方网站上下载到全套的代码!contiki 是开源的,可移植的,针对存储空间受限的网络化嵌入式系统和无线传感器网络的多任务操作系统。而且 contiki 的代码全部为 C 语言写成,用 GCC 进行编译,对广大应用 C 语言多年的开发者来说,减少了学习另外一种语言与编译平台所带来的时间花费。对很多开发者和初学者来说,对 Linux+GCC 的平台不熟悉,所以,可以选用 IAR 这个极其稳定、易用的编译平台,对 contiki 进行移植。

2、Z-Stack+OSAL 操作系统

2007 年 4 月,德州仪器推出业界领先的 ZigBee 协议栈 (Z-Stack)。Z-Stack 符合 ZigBee 2007 规范,支持多种平台,包括基于 CC2530 收发器以及 TI MSP430 超低功耗单片机的平台,CC2530 SOC 平台 C51RF-3-PK 等。Z-Stack 包含了网状网络拓扑的几近于全功能的协议栈,在竞争激烈的 ZigBee 领域占有重要地位。

OSAL,英文全称 Operating System Abstraction Layer,中文解释操作系统抽象层。它可以看做是一种机制,一种任务分配资源的机制,从而形成了一个简单多任务的操作系统。

3、msstatePAN 协议栈

msstatePAN 协议栈是由密西西比大学的 R .Reese 教授为广大无线技术爱好者开发的

精简版 ZigBee 协议栈 基于标准 C 语言编写 基本具备了 ZigBee 协议标准所规定的功能，最新版本为 V0.2.6，该版本支持多种开发平台，包括 PICDEM Z、CC2530 评估板、MSP430+CC2420(Tmote)以及 WIN32 虚拟平台。源代码是开放的，整个协议栈是基于状态机 (FSM) 实现的。只是其中程序排版不太规范。如果你的程序构架不是基于操作系统的，有限状态机应该是一个很好的选择。而且 OS (operating system) 中进程的状态也是个各个状态间的切换。

4、TinyOS

TinyOS 是 UC Berkeley (加州大学伯克利分校) 开发的开放源代码操作系统，专为嵌入式无线传感网络设计，操作系统基于构件 (component-based) 的架构使得快速的更新成为可能，而这又减小了受传感网络存储器限制的代码长度。TinyOS 的构件包括网络协议、分布式服务器、传感器驱动及数据识别工具。其良好的电源管理源于事件驱动执行模型，该模型也允许时序安排具有灵活性。TinyOS 已被应用于多个平台和感应板中。

5、freaklabs，日本的一个开源协议栈。

IEEE 802.15.4 标准概述

IEEE 802.15.4 是一个低速率无线个人局域网(Low Rate Wireless Personal Area Networks, LR-WPAN)标准。该标准定义了物理层(PHY)和介质访问控制层(MAC)。这种低速率无线个人局域网的网络结构简单、成本低廉、具有有限的功率和灵活的吞吐量。低速率无线个人局域网的主要目标是实现安装容易、数据传输可靠、短距离通信、极低的成本、合理的电池寿命，并且拥有一个简单而且灵活的通信网络协议。

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(carrier sense multiple access with collision avoidance, CSMA-CA)。
- ◆ 用于可靠传输的全应答协议。
- ◆ 低功耗。
- ◆ 能量检测(Energy Detection, ED)。
- ◆ 链路质量指示(Link Quality Indication, LQI)。
- ◆ 在 2450MHz 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(Institute of Electrical and Electronics Engineers，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 functional device, FFD)，另一种是简化功能设备(reduced functional device, RFD)。

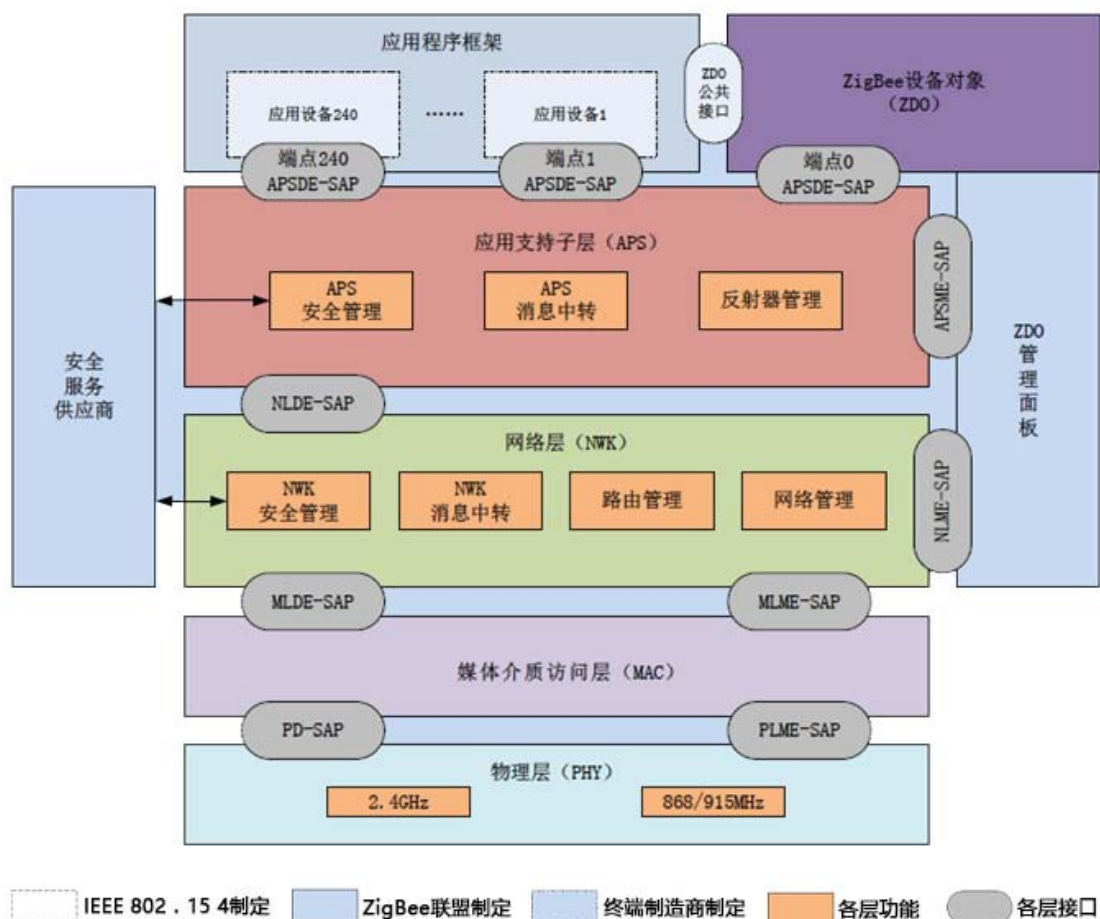
ZigBee 协议体系结构

什么是 ZigBee 协议栈呢？

它和 ZigBee 协议有什么关系呢?协议是一系列的通信标准，通信双方需要共同按照这一标准进行正常的数据发射和接收。协议栈是协议的具体实现形式，通俗点来理解就是协议栈是协议和用户之间的一个接口，开发人员通过使用协议栈来使用这个协议的，进而实现无线数据收发。

ZigBee 协议栈建立在 IEEE 802.15.4 的 PHY 层和 MAC 子层规范之上。它实现了网络层(networklayer, NWK)和应用层(applicationlayer, APL)。在应用层内提供了应用支持子层(application support sub-layer, APS)和 ZigBee 设备对象(ZigBee Device Object, ZDO)。应用框架中则加入了用户自定义的应用对象

ZigBee 的体系结构由称为层的各模块组成。每一层为其上层提供特定的服务：即由数据服务实体提供数据传输服务；管理实体提供所有的其他管理服务。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。ZigBee 协议的体系结构如下图所示：



物理层 (PHY)

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。

物理层内容：

- 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。

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。

MAC 层功能：

- 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；
- 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。

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。

网络层功能：

- 1)网络发现；
- 2)网络形成；
- 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。

应用层 (APL)

ZigBee 应用层框架包括应用支持层(APS)、ZigBee 设备对象(ZDO)和制造商所定义的应用对象。

应用支持层的功能包括：维持绑定表、在绑定的设备之间传送消息。

ZigBee 设备对象的功能包括：定义设备在网络中的角色(如 ZigBee 协调器和终端设备)，发起和响应绑定请求，在网络设备之间建立安全机制。ZigBee 设备对象还负责发现网络中的设备，并且决定向他们提供何种应用服务。

ZigBee 应用层除了提供一些必要函数以及为网络层提供合适的服务接口外，一个重要的功能是应用者可在这层定义自己的应用对象。

应用程序框架 (AF):

运行在 ZigBee 协议栈上的应用程序实际上就是厂商自定义的应用对象，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)

ZigBee 设备对象(ZDO)的功能包括负责定义网络中设备的角色，如：协调器或者终端设备。还包括对绑定请求的初始化或者响应，在网络设备之间建立安全联系等。实现这些功能，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。

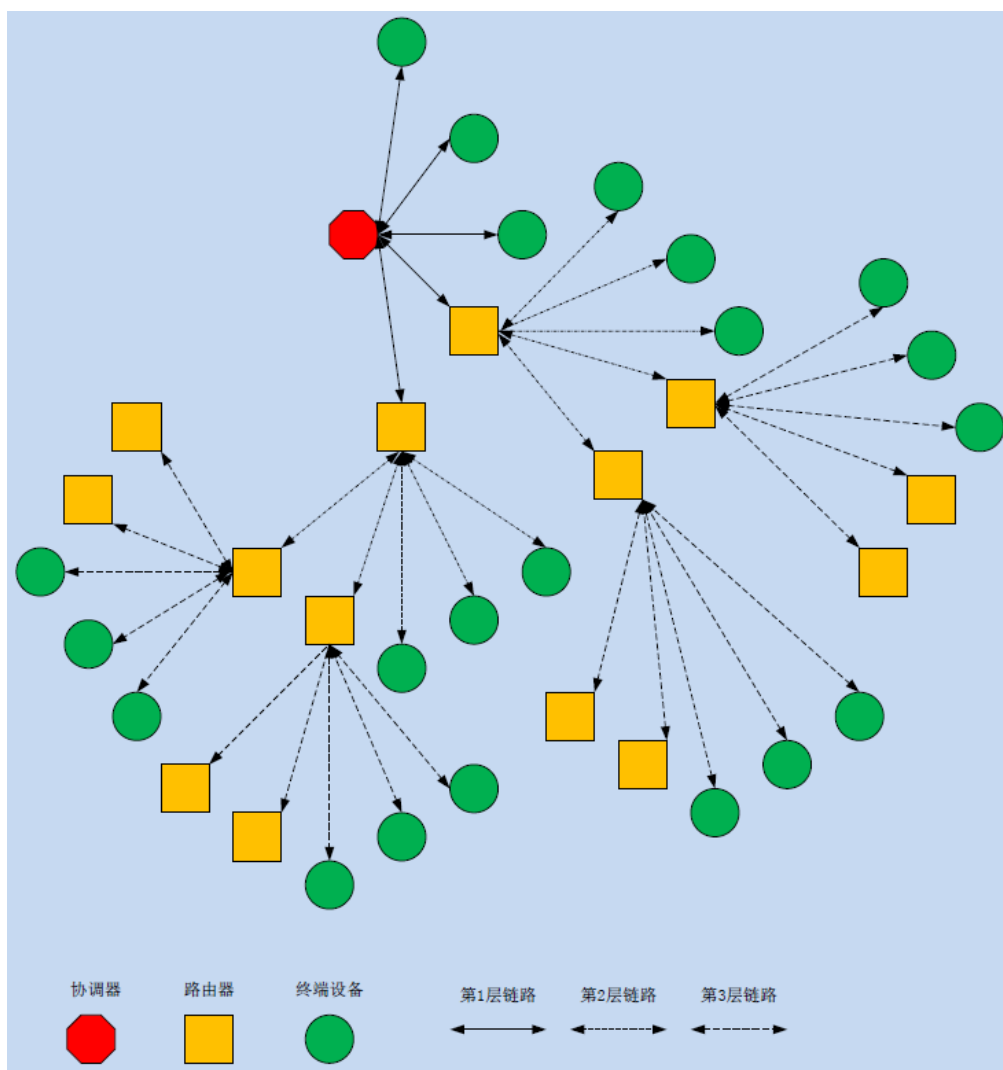
二、ZigBee 基本概念

设备类型(Device Types)

在 ZigBee 网络中存在三种逻辑设备类型：**Coordinator(协调器)**，**Router(路由器)**和 **End-Device(终端设备)**。ZigBee 网络由一个 Coordinator 以及多个 Router 和多个 End_Device 组成。

注意：在 ZStack-CC2530-2.5.1a 中一个设备的类型通常在编译的时候通过编译选项确定。所有的应用例子都提供独立的项目文件来编译每一种设备类型。对于协调器，在 Workspace 区域的下拉菜单中选择 CoordinatorEB；对于路由器，在 Workspace 区域的下拉菜单中选择 RouterEB；对于终端设备，在 Workspace 区域的下拉菜单中选择 EndDeviceEB。

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



上图是一个简单的 ZigBee 网络示意图。其中红色节点为 Coordinator ,黄色节点为 Router ,绿色节点为 End-Device。

Coordinator(协调器)

协调器负责启动整个网络。它也是网络的第一个设备。协调器选择一个信道和一个网络 ID(也称之为 PAN ID ,即 Personal Area Network ID) ,随后启动整个网络。协调器也可以用来协助建立网络中安全层和应用层的绑定(bindings)。

注意,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 go away)。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。

Router(路由器)

路由器的功能主要是 :允许其他设备加入网络,多跳路由和协助它自己的由电池供电的终端设备的通讯。

通常,路由器希望是一直处于活动状态,因此它必须使用主电源供电。但是当使用树状网络拓扑结构时,允许路由间隔一定的周期操作一次,这样就可以使用电池给其供电。

End-Device(终端设备)

终端设备没有特定的维持网络结构的责任，它可以睡眠或者唤醒，因此它可以是一个电池供电设备。通常，终端设备对存储空间(特别是 RAM 的需要)比较小。

协议栈规范 (Stack Profile)

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 ZigBee PRO。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；ZigBee PRO 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：

```
// Controls various stack parameter settings
#define NETWORK_SPECIFIC      0
#define HOME_CONTROLS        1
#define ZIGBEEPRO_PROFILE    2
#define GENERIC_STAR         3
#define GENERIC_TREE         4

#if defined ( ZIGBEEPRO )
    #define STACK_PROFILE_ID    ZIGBEEPRO_PROFILE
#else
    #define STACK_PROFILE_ID    HOME_CONTROLS
#endif
```

在 f8wConfig.cfg 文件定义

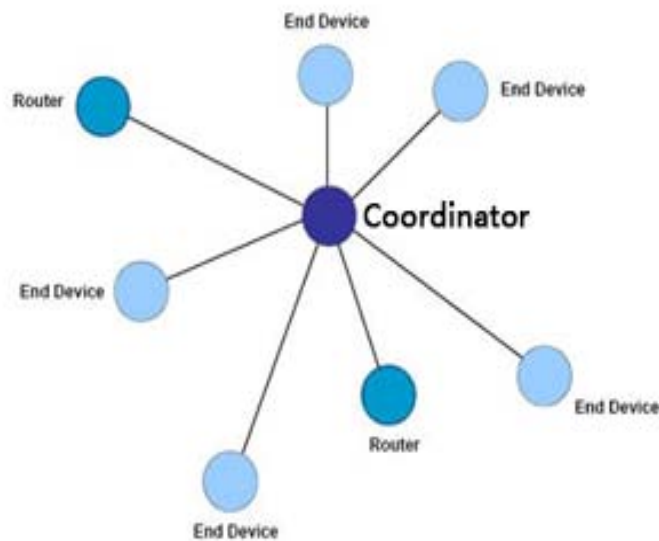
```
/* Enable ZigBee-Pro */
-DZIGBEEPRO
```

拓扑结构

ZigBee 网络支持星状、树状和网状三种网络拓扑结构，如下图所示，分别依次是星状网络，树（簇）状网络和网状网络。

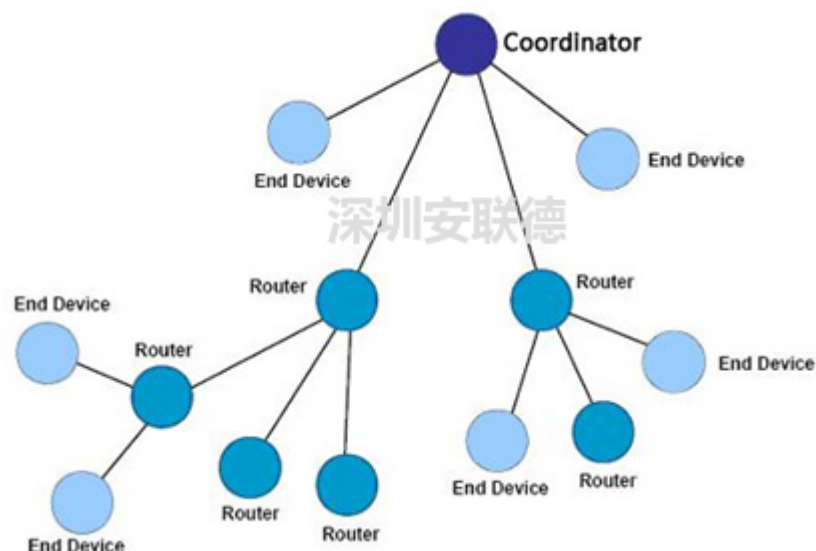
星形拓扑是最简单的一种拓扑形式，他包含一个 Coordinator（协调者）节点和一系列的 End Device（终端）节点。每一个 End Device 节点只能和 Coordinator 节点进行

通讯。如果需要在两个 End Device 节点之间进行通讯必须通过 Coordinator 节点进行信息的转发。



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。Coordinator (协调者) 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。

树形拓扑包括一个 Coordinator (协调者) 以及一系列的 Router (路由器) 和 End Device (终端) 节点。Coordinator 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：

Coordinator 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

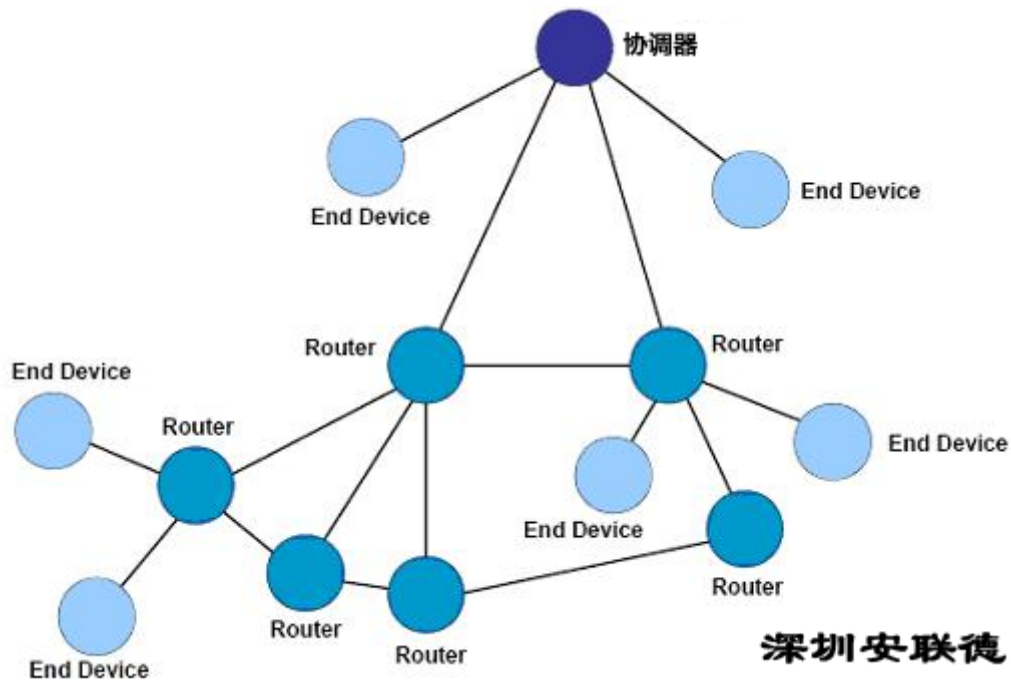
树形拓扑中的通讯规则：

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） 包含一个 Coordinator 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。

MESH 网状网络拓扑结构的网络具有强大的功能，网络可以通过“多级跳”的方式来通信；该拓扑结构还可以组成极为复杂的网络；网络还具备自组织、自愈功能；

星型和族树型网络适合点多多点、距离相对较近的应用。

在 Z-Stack 中网络拓扑结构定义如下：

```
// Controls the operational mode of network
#define NWK_MODE_STAR      0
#define NWK_MODE_TREE     1
```

```

#define NWK_MODE_MESH                2

#if ( STACK_PROFILE_ID == ZIGBEEPRO_PROFILE )
    #define MAX_NODE_DEPTH            20
    #define NWK_MODE                   NWK_MODE_MESH
    #define SECURITY_MODE              SECURITY_COMMERCIAL
    #if ( SECURE != 0 )
        #define USE_NWK_SECURITY      1    // true or false
        #define SECURITY_LEVEL        5
    #else
        #define USE_NWK_SECURITY      0    // true or false
        #define SECURITY_LEVEL        0
    #endif
#endif

```

信标与非信标模式

ZigBee 网络的工作模式可以分为信标 (Beacon) 和非信标 (Non-beacon) 两种模式。信标模式实现了网络中所有设备的同步工作和同步休眠，以达到最大限度的功耗节省，而非信标模式则只允许终端设备进行周期性休眠，协调器 和所有路由器 设备必须长期处于工作状态。

信标模式下，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。

非信标模式下，ZigBee 标准采用父节点为终端设备子节点缓存数据，终端设备主动向其父节点提取数据的机制，实现终端设备的周期性（周期可设置）休眠。网络中所有父节点需为自己的终端设备子节点缓存数据帧，所有终端设备子节点的大多数时间都处于休眠模式，周期性的醒来与父节点握手以确认自己仍处于网络中，其从休眠模式转入数据传输模式一般只需要 15ms。

地址定义

ZigBee 设备有两种类型的地址。一种是 64 位 IEEE 地址，即 MAC 地址，另一种是 16 位网络地址。

64 位地址使全球唯一的地址，设备将在它的生命周期中一直拥有它。它通常由制造商或者被安装时设置。这些地址由 IEEE 来维护和分配。

16 位网络地址是当设备加入网络后分配的。它在网络中是唯一的，用来在网络中鉴别设备和发送数据。其中，协调器的网络地址为 0x00

// Network PAN Coordinator Address

```

#define NWK_PAN_COORD_ADDR 0x0000

```

ZigBee 2007 PRO 使用的随机地址分配机制，对新加入的节点使用随机地址分配，为保证网络内地址分配不重复，使用其余的随机地址再进行分配。当一个节点加入时，将接收

到父节点的随机分配地址，然后产生“设备声明”（包含分配到的网络地址和 IEEE 地址）发送至网络中的其余节点。如果另一个节点有着同样的网络地址，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。所有发生网络地址冲突的节点更改自己的网络地址，然后再发起“设备声明”检测新的网络地址是否冲突。

终端设备不会广播“地址冲突”，他们的父节点会帮助完成。如果一个终端设备发生了“地址冲突”，他们的父节点发送“重新加入”消息至终端设备，并要求他们更改网络地址。然后，终端设备再发起“设备声明”检测新的网络地址是否冲突。

当接收到“设备声明”后，关联表和绑定表将被更新使用新的网络地址，但是路由表不会被更新。

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。这些参数是 MAX_DEPTH（最大网络深度）、MAX_ROUTERS（最多路由数）和 MAX_CHILDREN（最多子节点数）。

```
#if ( STACK_PROFILE_ID == ZIGBEEPRO_PROFILE )
```

```
uint8 CskipRtrs[1] = {0};
```

```
uint8 CskipChldrn[1] = {0};
```

Cm (nwkMaxChildren)：每个父节点可以连接的子节点的总个数；

Rm (nwkMaxRouters)：在 Cm 中，可以是路由节点的个数， $Rm \leq Cm$ ；

Lm：网络最大深度，协调器的深度为 0。

这三个参数的值在 [Z-stack](#) 中分别由变量 CskipChldrn、CskipRtrs、MAX_NODE_DEPTH 决定。这三个变量可以在 NWK 中的 nwk_globals.c 和 nwk_globals.h 两个文件中查找。

寻址

为了向一个在 ZigBee 网络中的设备发送数据，应用程序通常使用 AF_DataRequest()函数。数据包将要发送给一个 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;
    uint8 endPoint;
    uint16 panId; // used for the INTER_PAN feature
} afAddrType_t;
```

注意，除了网路地址之外，还要指定地址模式参数。目的地址模式可以设置为以下几个值：

```
typedef enum
{
    afAddrNotPresent = AddrNotPresent,
    afAddr16Bit      = Addr16Bit,
```

```

afAddr64Bit      = Addr64Bit,
afAddrGroup      = AddrGroup,
afAddrBroadcast  = AddrBroadcast
} afAddrMode_t;

```

因为在 Zigbee 中,数据包可以单点传送(unicast),多点传送(multicast)或者广播传送,所以必须有地址模式参数。一个单点传送数据包只发送给一个设备,多点传送数据包则要传送给一组设备,而广播数据包则要发送给整个网络的所有节点。这个将在下面详细解释。

单点传送(Unicast)

Uicast 是标准寻址模式,它将数据包发送给一个已经知道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。

间接传送(Indirect)

当应用程序不知道数据包的目标设备在哪里的时候使用的模式。将模式设为 AddrNotPresent 并且目标地址没有指定。取代它的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。

当数据向下发送到栈中,从绑定表中查找并且使用该目标地址。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。

广播传送(broadcast)

当应用程序需要将数据包发送给网络的每一个设备时,使用这种模式。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:

NWK_BROADCAST_SHORTADDR_DEVALL(0xFFFF)——数据包将被传送到网络上的所有设备,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)。

NWK_BROADCAST_SHORTADDR_DEVRXON(0xFFFD)——数据包将被传送到网络上的所有在空闲时打开接收的设备(RXONWHENIDLE),也就是说,除了睡眠中的所有设备。

NWK_BROADCAST_SHORTADDR_DEVZCZR(0xFFFC)——数据包发送给所有的路由器,包括协调器。

组寻址(Group Addressing)

当应用程序需要将数据包发送给网络上的一组设备时,使用该模式。地址模式设置为 afAddrGroup 并且 addr.shortAddr 设置为组 ID。

在使用这个功能呢之前,必须在网络中定义组。(参见 Z-stack API 文档中的 aps_AddGroup() 函数)。

注意组可以用来关联间接寻址。再绑定表中找到的目标地址可能是是单点传送或者是一个组地址。另外,广播发送可以看做是一个组寻址的特例。

下面的代码是一个设备怎样加入到一个 ID 为 1 的组当中:

```

// Group Table Element
typedef struct
{
    uint16 ID;                // Unique to this table

```



```
uint8 name[APS_GROUP_NAME_LEN]; // Human readable name of group
} aps_Group_t;
```

设备 (device)

一个节点就是一个设备，对应一个无线单片机 (CC2530)；一个设备有一个射频端，具有唯一的 IEEE 地址(64 位)和网络地址(16 位)。在协议栈中不同的设备有相应的配置文件：

协调器配置文件：f8wCoord.cfg

路由器配置文件：f8wRouter.cfg

终端设备配置文件：f8wEndev.cfg

重要设备地址(Important Device Addresses)

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 ZStack API 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

三、如何使用 ZigBee 协议栈

协议栈是协议的实现，可以理解为代码，函数库，供上层应用调用，协议较底下的层与应用是相互独立的。商业化的协议栈就是给你写好了底层的代码，符合协议标准，提供给你一个功能模块给你调用。你需要关心的就是你的应用逻辑，数据从哪里到哪里，怎么存储，处理；还有系统里的设备之间的通信顺序什么的，当你的应用需要数据通信时，调用组网函数给你组建你想要的网络；当你想从一个设备发数据到另一个设备时，调用无线数据发送函数；当然，接收端就调用接收函数；当你的设备没事干的时候，你就调用睡眠函数；要干活的时候就调用唤醒函数。所以当你做具体应用时，不需要关心协议栈是怎么写的，里面的每条代码是什么意思。除非你要做协议研究。每个厂商的协议栈有区别，也就是函数名称和参数可能有区别，这个要看具体的例子、说明文档。

怎么使用 ZigBee 协议栈？

举个例子，用户实现一个简单的无线数据通信时的一般步骤：

- 1、组网：调用协议栈的组网函数、加入网络函数，实现网络的建立与节点的加入。
- 2、发送：发送节点调用协议栈的无线数据发送函数，实现无线数据发送。
- 3、接收：接收节点调用协议栈的无线数据接收函数，实现无线数据接收。

是不是看上去很简单啊，其实协议栈很多都封装好了，下面我们大概看看无线发送函数：

1. afStatus_t AF_DataRequest(afAddrType_t *dstAddr,
2. endPointDesc_t *srcEP,
3. uint16 cID,

4. `uint16 len,`
5. `uint8 *buf,`
6. `uint8 *transID,`
7. `uint8 options,`
8. `uint8 radius)`







用户调用该函数即可实现数据的无线数据的发送，此函数中有 8 个参数，用户需要将每个参数的含义理解以后，才能熟练使用该函数进行无线数据通信的目的。现在只讲其中最重要的两个参数，其它参数不需要死记硬背，以后用多了自然就记住了。

4. `uint16 len`, //发送数据的长度;
5. `uint8 *buf`, //指向存放发送数据的缓冲区的指针。

至于调用该函数后，如何初始化硬件进行数据发送等工作，用户不需要关心，ZigBee 协议栈已经将所需要的工作做好了，我们只需要调用相应的 API 函数即可，而不必关心具体实现细节。看起来是不是很简单呢，是不是有动手试试的冲动。先别急还要先安装 **ZigBee** 协议栈才能进行开发调试呢，下面就动手安装 **ZigBee** 协议栈吧。

四、ZigBee 协议栈的安装、编译与下载

提示：第 5 章中每个实验都包括了协议栈源码，大家可以不用安装，有兴趣也可以安装双击“..\相关资料与软件\Zigbee 开发软件\ZStack-CC2530-2.5.1a.exe”进行安装，路径你可以选择默认，同样你也可以选择你想要安装的位置。也许有人就困惑了，装完之后不是应该有个桌面图标的么？其实所谓的安装协议栈只是把一些文件解压到你安装的目录下。好了，协议栈是安装好了，可是怎么用它呢？我们先来看看这个协议栈的目录。

计算机 > 本地磁盘 (C:) > Texas Instruments > ZStack-CC2530-2.5.1a		
名称	修改日期	类型
 Components	2015/5/22 10:20	文件夹
 Documents	2015/5/22 10:20	文件夹
 Projects	2015/5/22 10:20	文件夹
 Tools	2015/5/22 10:20	文件夹
 Getting Started Guide - CC2530.pdf	2011/6/25 17:28	Adobe Ac
 README CC2530.txt	2012/4/25 21:19	文本文档

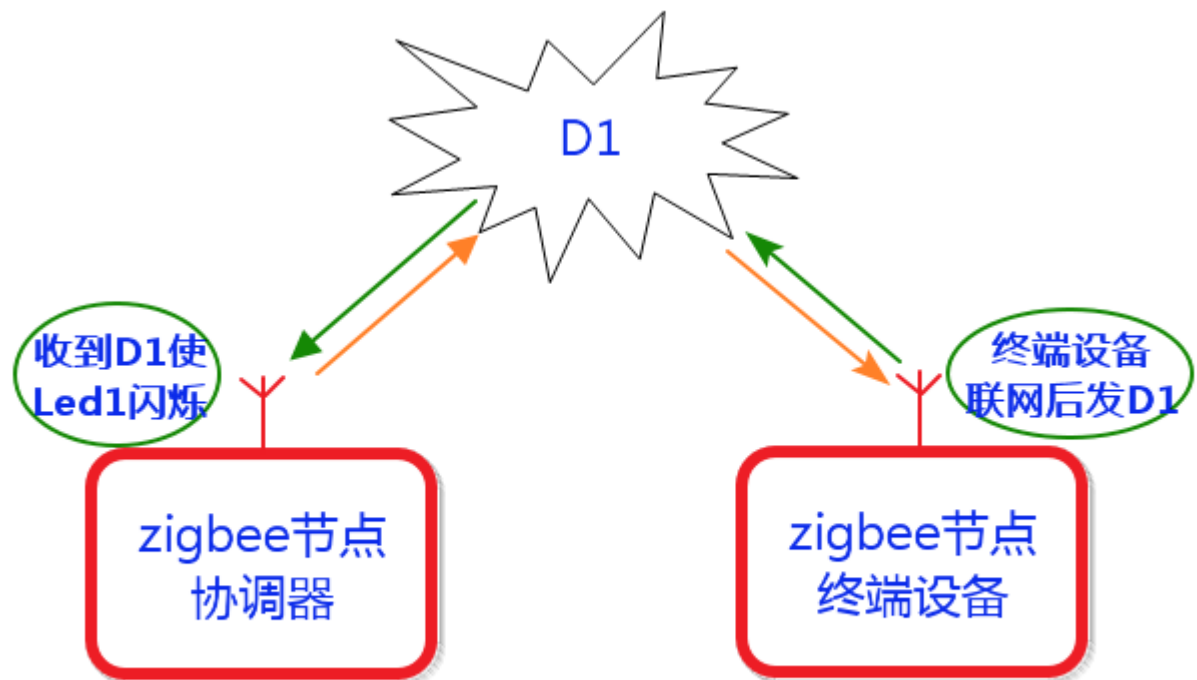
Components，顾名思义这个是放我们的库的文件夹，里面放了一些我们用到的 ZDO，driver，hal，zcl 等库的代码

Documents，这个不用说大家都知道是放 TI 的开发文档的，里面很多都是讲述协议栈的 API 的有空时可以看看

Projects，这个文件夹放的是 TI 协议栈的例子程序，一个个例子程序都是以一个个 project 的形式给我们的，学好这些例子程序里面的一两个，基本你能做事情了。

Tools，这个文件夹是放 TI 的例子程序的一些上位机之类的程序，作为工具使用。

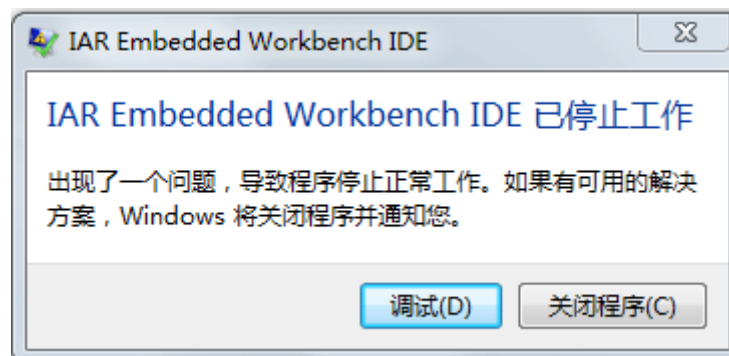
好了，基本明白了基本架构之后，我们以一个简单的实验开始。先掌握一点必要的理论再实验效果比较好。

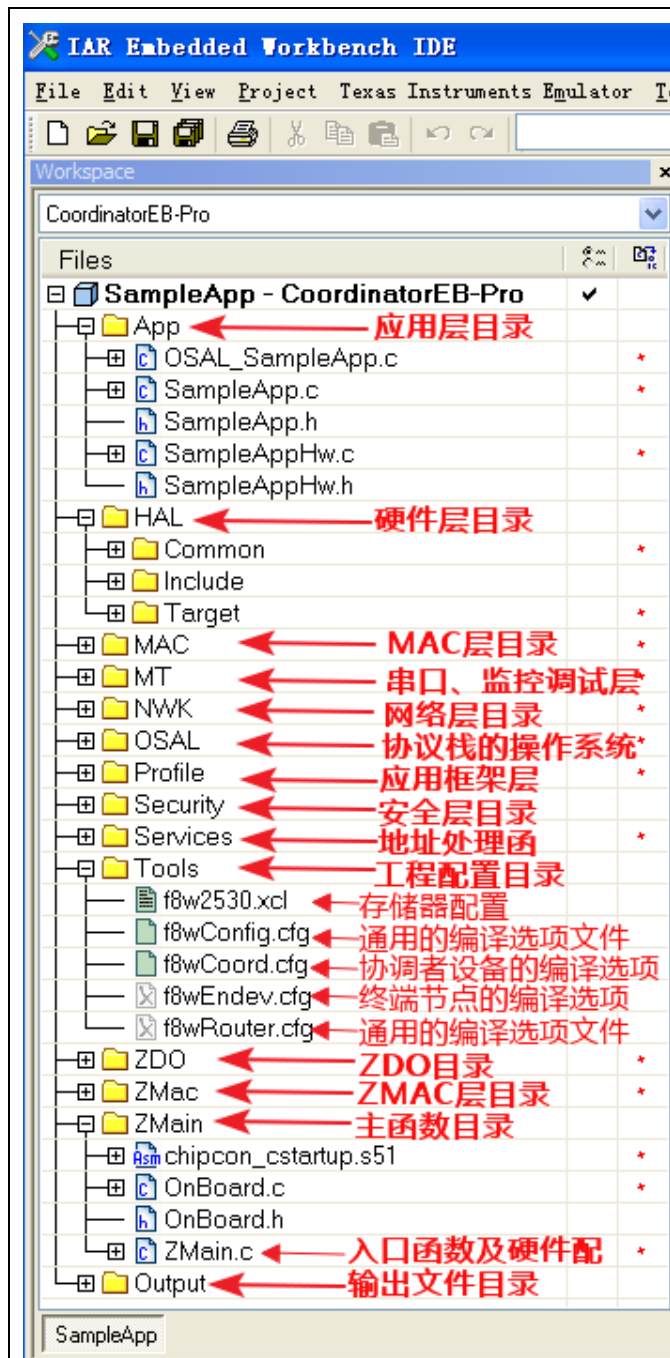


协议栈无线收发控制LED通讯过程

打开工程：[CC2530\第 5 章 zigbee 协议栈应用与组网\1.协议栈工作流程和无线收发控制LED](#)
[\ZStack-2.5.1a\Projects\zstack\Samples\SampleApp\CC2530DB\SampleApp.eww](#)，从软件开发专业角度讲建议大家复制工程到非中文目录，因为有些开发环境对中文路径支持的不好，虽然 IAR 支持但在实际工作中你想别人看到你的工程，认为你专业就照着上面做吧。有时把文件放的太深，目录太长，打开工程时 IAR 会关闭；只要将工程上移几层即可，用英文路径最专业了。我们演示就不修改，容易引起大家误会；打开工程如下图：

如果使用 IAR 打开工程停止响应或关闭，说明你路径太长，IAR 不识别，把路径改短或移上几层目录即可解决。





App：应用层目录，这是用户创建各种不同工程的区域，在这个目录中包含了应用层的内容和这个项目的主要内容。

HAL：硬件层目录，包含有与硬件相关的配置和驱动及操作函数。

MAC：MAC 层目录，包含了 MAC 层的参数配置文件及其 MAC 的 LIB 库的函数接口文件。

MT：实现通过串口可控制各层，并与各层进行直接交互

NWK：网络层目录，包含网络层配置参数文件网络层库的函数接口文件及 APS 层库的函数接口。

OSAL：协议栈的操作系统。

Profile：Application framework 应用框架层目录，包含 AF 层处理函数文件。应用框架层是应用程序和 APS 层的无线数据接口。

Security：安全层目录，包含安全层处理函数，比如加密函数等

Services：地址处理函数目录，包括地址模式的定义及地址处理函数。

Tools：工程配置目录，包括空间划分及 Z-Stack 相关配置信息。

ZDO：ZDO 目录

ZMac：MAC 层目录，包括 MAC 层参数配置及 MAC 层 LIB 库函数回调处理函数。

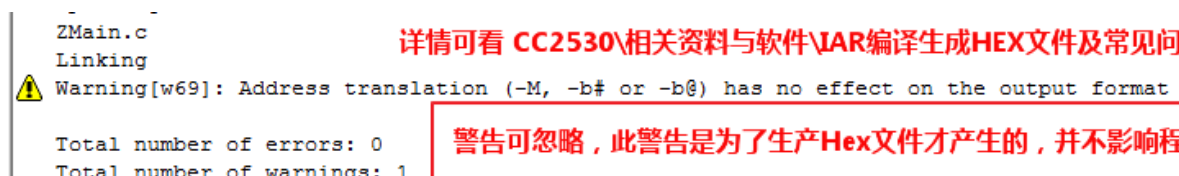
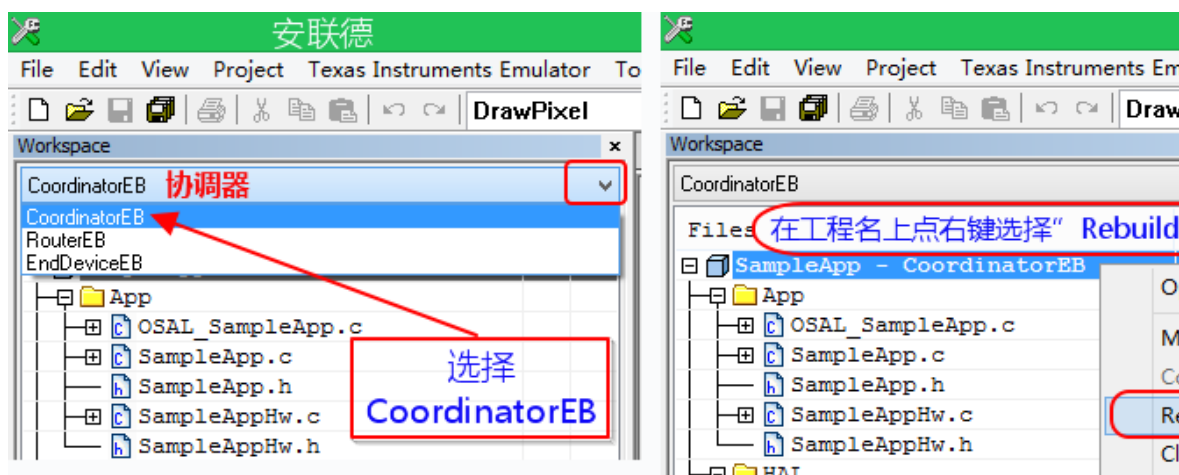
ZMain：主函数目录，包括入口函数及硬件配置文件。

Output：输出文件目录，由 IAR IDE 自动生成。

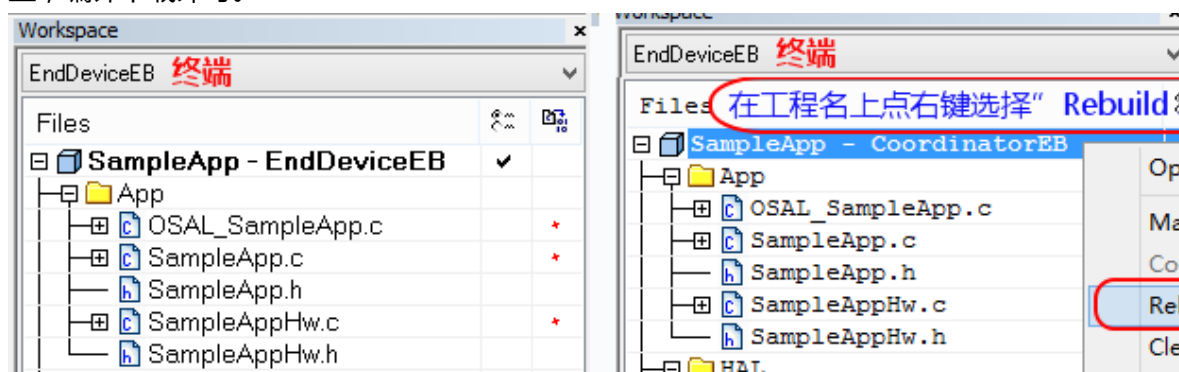
带协议栈的工程怎么这么多文件夹和文件，都有什么用啊？现阶段大家只要带着这个疑问照着做实验就行了，后面实验接触多了自然就懂了。

1. 编译协调器的程序，在 Workspace 下拉框中选择“CoordinatorEB”，在工程名上点右键选择“Rebuild All”，第一次一定要用“Rebuild All”，后面再修改代码只用“Make”即可，没错误提示再下载到开发板当中。尽量教大家用一些快捷方法。编译结果会有以下警告，警告

可忽略的，只要没有错误就可运行的，以下警告是用来生成量产的文件用的，想消除可看看"
2530\相关资料与软件\IAR 编译 CC2530 生成 HEX 文件.pdf"



2. 下载好协调器后，再编译终端设备的程序选择“EndDeviceEB”，把仿真器换到另一块板子上，编译下载即可。

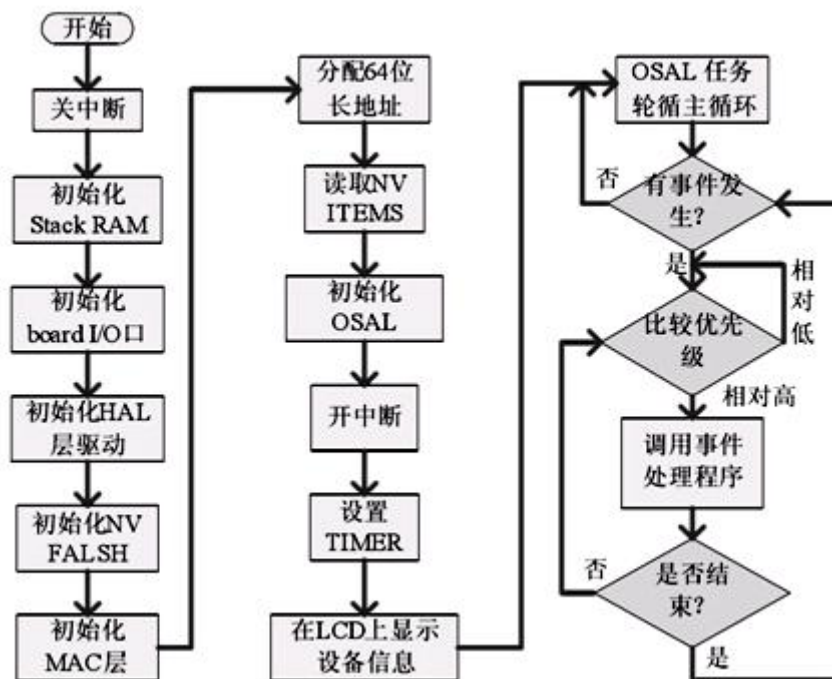


注意：旧版本协议栈显示为 EndDeviceEB-Pro，与 EndDeviceEB 只是显示名称不同，新协议栈取消了-Pro，没影响。

两个 zigbee 节点都下载好后，分别上电看效果吧。协调器、终端通过底板上的 usb 供电并打开开关，组网成功后 D1 灯闪烁。看完后是不是在想怎么实现的？下面我就带着大家分析。

五、协议栈工作流程：

zigbee 工作流程:



看源码推荐大家使用 SourceInsight，十分强大的工具，从事软件必备软件，除非你想做菜鸟，具体用法请参考"相关资料与软件\Zigbee 参考资料\ Source Insight 使用教程.pdf"或者看基础实验 3.16 的视频。下面列出实验中涉及到比较重要的函数进行详解，由于是带协议栈第一个实验，我们对源码也进行注释，方便习惯看源码的同志学习。我建议大家先看看下面的文章，再阅读一次源码加深印象，后面的例子结构基本相同，所以学好此实验，再做后面的实验就得心应手了。用户自己添加的应用任务程序在 Zstack 中的调用过程：

`main()---> osal_init_system()---> osalInitTasks()---> SampleApp_Init()`

下面我们就先从 main()函数开始吧。

提示 :如果你第一次接触 ZStack ,第一个实验的代码看注释只须大概知道它们是做什么的，有点印象就行了，后面实验会徐徐渐进，慢慢带领大家搞懂整个流程和代码的。如果刚开始就啃代码，不但效率低而且信心受损。

六、无线收发控制 LED 代码分析与步骤

1) 打开 ZMain.c 找到 main 函数

```
int main( void )
```

```
{
```

```
    osal_int_disable( INTS_ALL );    //关闭所有中断
    HAL_BOARD_INIT();                //初始化系统时钟
    zmain_vdd_check();               //检查芯片电压是否正常
    InitBoard( OB_COLD );            //初始化 I/O ， LED 、 Timer 等
    HalDriverInit();                 //初始化芯片各硬件模块
```



```

osal_nv_init( NULL );           //初始化 Flash 存储器
ZMacInit();                     //初始化 MAC 层
zmain_ext_addr();               //确定 IEEE 64 位地址
zglInit();                      //初始化非易失变量

#ifndef NONWK
    // Since the AF isn't a task, call it's initialization routine
    aflInit();
#endif

    osal_init_system();         //初始化操作系统
    osal_int_enable( INTS_ALL ); //使能全部中断
    InitBoard( OB_READY );       //最终板载初始化
    zmain_dev_info();            //显示设备信息

#ifdef LCD_SUPPORTED
    zmain_lcd_init();            //初始化 LCD
#endif

#ifdef WDT_IN_PM1
    /* If WDT is used, this is a good place to enable it. */
    WatchDogEnable( WDTIMX );
#endif

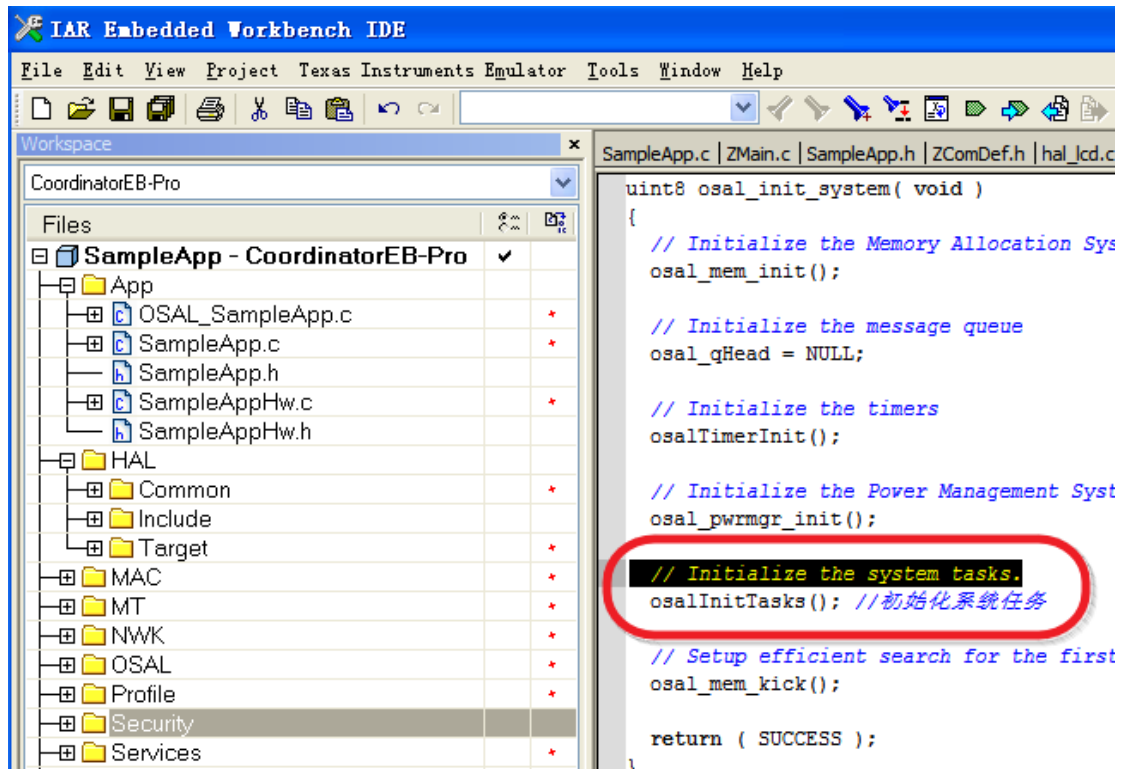
    osal_start_system(); // No Return from here 执行操作系统，进去后不会返回

    return 0; // Shouldn't get here.
} // main()

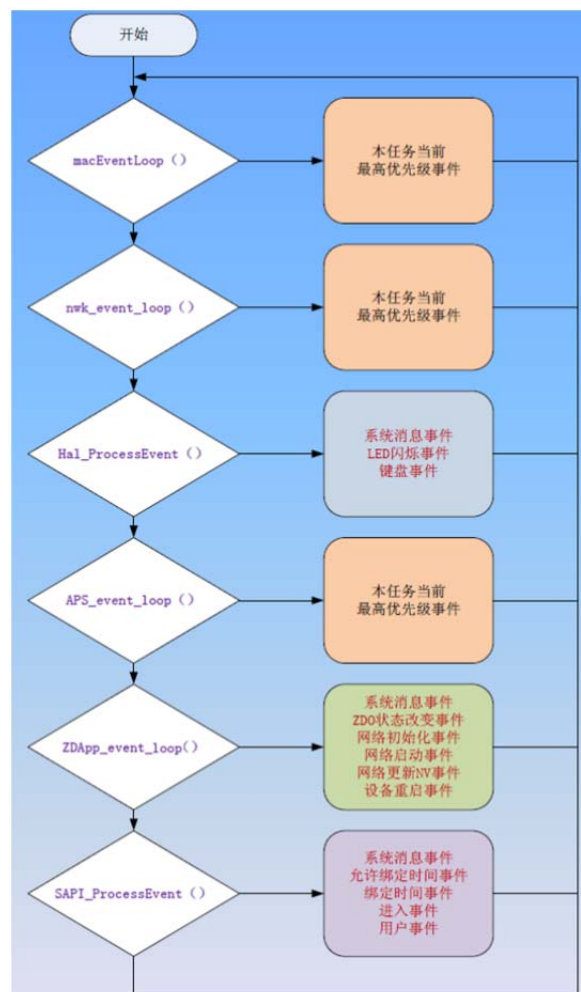
```

看了上面的代码后，可能感觉很多函数不认识。没关系刚开始大概了解流程即可，main 函数先执行初始化工作，包括硬件、网络层、任务等的初始化。然后执行 **osal_start_system();** 操作系统。进去后可不会回来了。在这里，我们重点了解 2 个函数：

- ◆ 初始化操作系统 **osal_init_system();**
 - ◆ 运行操作系统 **osal_start_system();**
- 2) 先来看 **osal_init_system();** 系统初始化函数，进入函数。如果用 IAR 看代码可在函数名上单击右键——**go to definition of...** 便可以进入函数。发现里面有 6 个初始化函数，这里我们只关心 **osalInitTasks();** 任务初始化函数，继续由该函数进入。



Z-Stack 中操作系统是基于优先级的轮转查询式操作系统. 执行流程图如下图所示:



在OSAL_SampleApp.c文件中

```
const pTaskEventHandlerFn tasksArr[] = {
    macEventLoop,          //用户不需要考虑
    nwk_event_loop,        //用户不需要考虑
    Hal_ProcessEvent,      //用户可以考虑
#ifdef MT_TASK
    MT_ProcessEvent,
#endif
    APS_event_loop,        //用户不需要考虑
#ifdef ( ZIGBEE_FRAGMENTATION )
    APSF_ProcessEvent,
#endif
    ZDApp_event_loop,      //用户可以考虑
#ifdef ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
    ZDNwkMgr_event_loop,
#endif
    SampleApp_ProcessEvent //用户可以考虑
};
```

void osallInitTasks(void) //任务初始化函数 在 OSAL_SampleApp.c 文件中

```
{
    uint8 taskID = 0;

    //为当前 OSAL 中各任务分配存储空间，tasksEvents 指向任务数组
    tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
    // 给分配的内存空间清 0
    osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));

    // 任务优先级由高向低依次排列，高优先级对应 taskID 的值反而小
    macTaskInit( taskID++ ); //macTaskInit(0) ，初始化 MAC 层任务 用户不需考虑
    nwk_init( taskID++ );    //nwk_init(1) ，初始化网络层任务 用户不需考虑
    Hal_Init( taskID++ );    //Hal_Init(2) ，初始化硬件任务 用户需考虑
#ifdef MT_TASK              //如果定义 MT_TASK 就初始化 MT 层任务,主要通过串口可控制各层，并与各层进行直接交互
    MT_TaskInit( taskID++ );
#endif
    APS_Init( taskID++ );    //APS_Init(3) ，用户不需考虑 初始化 APS 层任务,应用层由三个部分组成,APS 子层,ZDO(包含 ZDO 管理平台) 和制造商定义的应用对象 APS:提供 NWK 层和 APL 层之间的接口，又名应用支持子层
#ifdef ( ZIGBEE_FRAGMENTATION )
    APSF_Init( taskID++ );  //是否已定义分包传输
#endif
}
```

```

#endif
    ZDApp_Init( taskID++ );    //ZDApp_Init(4) , 用户需考虑
#if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
    ZDNwkMgr_Init( taskID++ );    //初始化网络管理任务
#endif
    //用户创建的任务
    SampleApp_Init( taskID );    // SampleApp_Init_Init(5) , 用户需考虑。重要!
}

```

任务初始化,就是为系统中各个任务分配存储空间,再为各任务分配 taskID;这里的顺序要注意. 系统主循环函数里 tasksEvents[idx]和 tasksArr[idx]的 idx 与这里 taskID 是一一对应关系。大家看到了注释后面有些写着用户需要考虑,有些则写着用户不需考虑。没错,需要考虑的用户可以根据自己的硬件平台或者其他设置,而写着不需考虑的也是不能修改的。TI 公司协议栈已完成。

指针数组 tasksEvents[]里面最终分别指向的是各任务存储空间,指针数组 tasksArr[]里面最终分别指向的是各任务事件处理函数,这两个指针数组里面各元素的顺序要一一对应,后面任务调用会调用相应的事件处理函数。

SampleApp_Init()是我们应用协议栈例程的必要函数,用户通常在这里初始化自己的东西。至此, **osal_init_system()**大概了解完毕。

3) 接下来看第二个函数 **osal_start_system()**;运行操作系统。同样用 go to definition 的方法进入该函数。 最终调用 **osal_run_system()**;

```

void osal_run_system ( void )
{
    uint8 idx = 0;

    osalTimeUpdate();    //扫描哪个事件被触发了,然后置相应的标志位
    Hal_ProcessPoll();    //轮询 TIMER 与 UART

    do {
        if (tasksEvents[idx])    // Task is highest priority that is ready.
        {
            break;                //得到待处理的最高优先级任务索引号 idx
        }
    } while ( ++idx < tasksCnt);

    if (idx < tasksCnt)
    {
        uint16 events;

```

```

halIntState_t intState;

HAL_ENTER_CRITICAL_SECTION(intState); // 进入临界区,保护
events = tasksEvents[idx];           //提取需要处理的任务中的事件
tasksEvents[idx] = 0;                 //清除本次任务的事件
HAL_EXIT_CRITICAL_SECTION(intState); // 退出临界区

events = (tasksArr[idx])( idx, events ); //通过指针调用任务处理函数，关键

HAL_ENTER_CRITICAL_SECTION(intState); //进入临界区
tasksEvents[idx] |= events; // 保存未处理的事件 Add back unprocessed
events to the current task.
HAL_EXIT_CRITICAL_SECTION(intState); // 退出临界区
}
#endif
else // Complete pass through all task events with no activity?
{
osal_pwrmgr_powerconserve(); // Put the processor/system into sleep
}
#endif
}

```

我们看一下 `events = tasksEvents[idx];` 进入 `tasksEvents[idx]` 数组定义，发现恰好是 `osalInitTasks()` 函数里面分配空间初始化过的 `tasksEvents`。而且 `taskID` 一一对应。这就是初始化与调用的关系。`taskID` 把任务联系起来了。

4) SampleApp_Init() 用户应用任务初始化函数

```

void SampleApp_Init( uint8 task_id )
{
    SampleApp_TaskID = task_id; //osal 分配的任务 ID 随着用户添加任务的增多而改变

```

`SampleApp_NwkState = DEV_INIT;` //设备状态设定为 ZDO 层中定义的初始化状态
 初始化应用设备的网络类型，设备类型的改变都要产生一个事件——`ZDO_STATE_CHANGE`，从字面理解为 ZDO 状态发生了改变。所以在设备初始化的时候一定要把它初始化为什么状态都没有。那么它就要去检测整个环境，看是否能重新建立或者加入存在的网络。但是有一种情况例外，就是当 `NV_RESTORE` 被设置的候（`NV_RESTORE` 是把信息保存在非易失存储器中），那么当设备断电或者某种意外重启时，由于网络状态存储在非易失存储器中，那么此时就只需要恢复其网络状态，而不需要重新建立或者加入网络了。这里需要设置 `NV_RESTORE` 宏定义。

```

SampleApp_TransID = 0; //消息发送 ID ( 多消息时有顺序之分 )

#if defined ( BUILD_ALL_DEVICES )
    if ( readCoordinatorJumper() )
        zgDeviceLogicalType = ZG_DEVICETYPE_COORDINATOR;
    else
        zgDeviceLogicalType = ZG_DEVICETYPE_ROUTER;
#endif // BUILD_ALL_DEVICES

//该段的意思是，如果设置了 HOLD_AUTO_START 宏定义，将会在启动芯片的时候会暂停
启动流程，只有外部触发以后才会启动芯片。其实就是需要一个按钮触发它的启动流程。
#if defined ( HOLD_AUTO_START )
    ZDOInitDevice(0);
#endif

    //设置发送数据的方式和目的地址寻址模式
    //发送模式:广播发送
SampleApp_Periodic_DstAddr.addrMode = (afAddrMode_t)AddrBroadcast; //广播
SampleApp_Periodic_DstAddr.endPoint = SAMPLEAPP_ENDPOINT; //指定端点号
SampleApp_Periodic_DstAddr.addr.shortAddr = 0xFFFF; //指定目的网络地址为广播地
址

    //发送模式:组播发送 Setup for the flash command's destination address - Group 1
SampleApp_Flash_DstAddr.addrMode = (afAddrMode_t)afAddrGroup; //组寻址
SampleApp_Flash_DstAddr.endPoint = SAMPLEAPP_ENDPOINT; //指定端点号
SampleApp_Flash_DstAddr.addr.shortAddr = SAMPLEAPP_FLASH_GROUP; //组号
0x0001

    //定义本设备用来通信的 APS 层端点描述符
SampleApp_epDesc.endPoint = SAMPLEAPP_ENDPOINT;
SampleApp_epDesc.task_id = &SampleApp_TaskID; //SampleApp 描述符的任务
ID
SampleApp_epDesc.simpleDesc //SampleApp 简单描述符
    = (SimpleDescriptionFormat_t *)&SampleApp_SimpleDesc;
SampleApp_epDesc.latencyReq = noLatencyReqs; //延时策略

//向 AF 层登记描述符, 登记 endpoint description 到 AF,要对该应用进行初始化并在 AF
进行登记, 告诉应用层有这么一个 EP 已经开通可以使用, 那么下层要是有关于该应用的信息
或者应用要对下层做哪些操作, 就自动得到下层的配合

```



```

afRegister( &SampleApp_epDesc );

// 登记所有的按键事件
RegisterForKeys( SampleApp_TaskID );

// By default, all devices start out in Group 1
SampleApp_Group.ID = 0x0001; //组号
osal_memcpy( SampleApp_Group.name, "Group 1", 7 ); //设定组名
aps_AddGroup( SAMPLEAPP_ENDPOINT, &SampleApp_Group ); //把该组登记添加到 APS 中

#if defined ( LCD_SUPPORTED )
    HalLcdWriteString( "SampleApp", HAL_LCD_LINE_1 ); //如果支持 LCD , 显示提示信息
#endif
}

5) SampleApp_ProcessEvent() 用户应用任务的事件处理函数
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
    afIncomingMSGPacket_t *MSGpkt;
    (void)task_id; // Intentionally unreferenced parameter

    if ( events & SYS_EVENT_MSG ) //接收系统消息再进行判断
    {
        //接收属于本应用任务 SampleApp 的消息, 以 SampleApp_TaskID 标记
        MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );
        while ( MSGpkt )
        {
            switch ( MSGpkt->hdr.event )
            {
                // Received when a key is pressed
                case KEY_CHANGE://按键事件
                    SampleApp_HandleKeys( ((keyChange_t *)MSGpkt)->state,
                    ((keyChange_t *)MSGpkt)->keys );
                    break;

                // Received when a messages is received (OTA) for this endpoint
                case AF_INCOMING_MSG_CMD: //接收数据事件,调用函数 AF_DataRequest()

```

接收数据

SampleApp_MessageMSGCB(MSGpkt); //调用回调函数对收到的数据进行处理

break;

// Received whenever the device changes state in the network

case ZDO_STATE_CHANGE: // 只要网络状态发生改变，就通过ZDO_STATE_CHANGE 事件通知所有的任务。同时完成对协调器，路由器，终端的设置

SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);

//if ((SampleApp_NwkState == DEV_ZB_COORD) //实验中协调器只接收数据所以取消发送事件

if ((SampleApp_NwkState == DEV_ROUTER) || (SampleApp_NwkState == DEV_END_DEVICE))

{

//这个定时器只是为发送周期信息开启的，设备启动初始化后从这里开始触发第一个周期信息的发送，然后周而复始下去。

osal_start_timerEx(SampleApp_TaskID,

SAMPLEAPP_SEND_PERIODIC_MSG_EVT,

SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT);

}

else

{

// Device is no longer in the network

}

break;

default:

break;

}

// Release the memory //事件处理完了，释放消息占用的内存

osal_msg_deallocate((uint8 *)MSGpkt);

//指针指向下一个放在缓冲区的待处理的事件，返回 while (MSGpkt)重新处理事件，直到缓冲区没有等待处理事件为止

MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive(SampleApp_TaskID);

```

    // return unprocessed events //返回未处理的事件
    return (events ^ SYS_EVENT_MSG);
}

// Send a message out - This event is generated by a timer
// (setup in SampleApp_Init()).
if ( events & SAMPLEAPP_SEND_PERIODIC_MSG_EVT )
{
    //处理周期性事件，利用 SampleApp_SendPeriodicMessage()处理完当前的周期性
    //事件，然后启动定时器开启下一个周期性事情，这样一种循环下去，也即是上面说的周期性
    //事件了，可以做为传感器定时采集、上传任务
    SampleApp_SendPeriodicMessage();

    // Setup to send message again in normal period (+ a little jitter)
    osal_start_timerEx(SampleApp_TaskID,SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
        (SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT + (osal_rand() & 0x00FF)) );

    // return unprocessed events 返回未处理的事件
    return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);
}

// Discard unknown events
return 0;
}

6) 分析接收数据函数 SampleApp_MessageMSGCB
//接收数据，参数为接收到的数据
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
    uint16 flashTime;
    byte buf[3];

    switch ( pkt->clusterId ) //判断簇 ID
    {
        case SAMPLEAPP_PERIODIC_CLUSTERID: //收到广播数据
            osal_memset(buf, 0 , 3);
            osal_memcpy(buf, pkt->cmd.Data, 2); //复制数据到缓冲区中
    }
}

```

```

        if(buf[0]=='D' && buf[1]=='1')           //判断收到的数据是否为 "D1"
        {
            HalLedBlink(HAL_LED_1, 0, 50, 500); //如果是则 Led1 间隔 500ms 闪烁
#ifdef ZDO_COORDINATOR //协调器收到"D1"后,返回"D1"给终端,让终端 Led1
也闪烁
            SampleApp_SendPeriodicMessage();
#endif
        }
        else
        {
            HalLedSet(HAL_LED_1, HAL_LED_MODE_ON);
        }
        break;

    case SAMPLEAPP_FLASH_CLUSTERID: //收到组播数据
        flashTime = BUILD_UINT16(pkt->cmd.Data[1], pkt->cmd.Data[2] );
        HalLedBlink( HAL_LED_4, 4, 50, (flashTime / 4) );
        break;
    }
}

7) 分析发送周期信息 SampleApp_SendPeriodicMessage()
void SampleApp_SendPeriodicMessage( void )
{
    byte SendData[3]="D1";

    // 调用 AF_DataRequest 将数据无线广播出去
    if( AF_DataRequest( &SampleApp_Periodic_DstAddr, &SampleApp_epDesc,
                        SAMPLEAPP_PERIODIC_CLUSTERID,
                        2,
                        SendData,
                        &SampleApp_TransID,
                        AF_DISCV_ROUTE,
                        AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
    {
    }
    else
    {
        HalLedSet(HAL_LED_1, HAL_LED_MODE_ON);
    }
}

```

```

        // Error occurred in request to send.
    }
}
8) AF_DataRequest 发送函数
AF_DataRequest( &SampleApp_Periodic_DstAddr, //发送目的地址 + 端点地址和传送
模式
&SampleApp_epDesc, //源(答复或确认)终端的描述（比如操作系统中任务 ID 等）源
EP
SAMPLEAPP_PERIODIC_CLUSTERID, //被 Profile 指定的有效的集群号
2, // 发送数据长度
SendData, // 发送数据缓冲区
&SampleApp_TransID, // 任务 ID 号
AF_DISCV_ROUTE, // 有效位掩码的发送选项
AF_DEFAULT_RADIUS ) //传送跳数，通常设置为 AF_DEFAULT_RADIUS

```

好了，第一次就讲这么多吧，内容很多但非常重要，最好理解后再去做后面的实现，打好坚实的基础后，再去看后面的实验相对容易很多。

实验步骤

- 1.选择 CoordinatorEB, 下载到开发板 A；作为协调器
- 2.选择 EndDeviceEB, 下载到开发板 B；作为终端设备
- 3.给两块开发板上电，通过观察 D3 来判断组网是否成功，协调器 D3 熄灭说明已建立 zigbee 网络，有终端时可入网；当终端 D3 熄灭时说明连网成功，请观察 Led1 灯的变化。下载好程序后可以不需要仿真器了，通过底板供电即可。有显示屏的用户可以通过显示屏来观察组网情况。



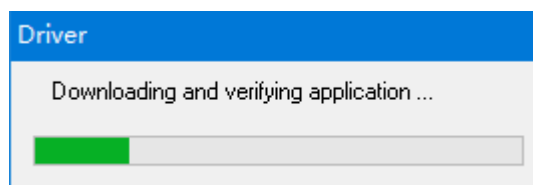


注意：如果模块都接了天线请相距 30cm 以上观察，太近有谐振波可能通讯不上。

显示屏上有节点设备类型、PAN ID、IEEE 地址、联网信息，由于此实验内容过多，显示屏更详细的讲解再后面给出。屏的好处还是比较大的，没有屏的就通过 LED 观察了。部分人可能编译下载没有出现闪烁的现象，可先分别下载“实验固件”里面的 hex 文件，下载 Hex 文件方法请看“cc2530\相关资料与软件\开发板出厂测试程序\烧出厂程序的方法.pdf”也许有人会问 hex 怎么生成的呢？有什么用？带着疑问学习稍后给出答案。

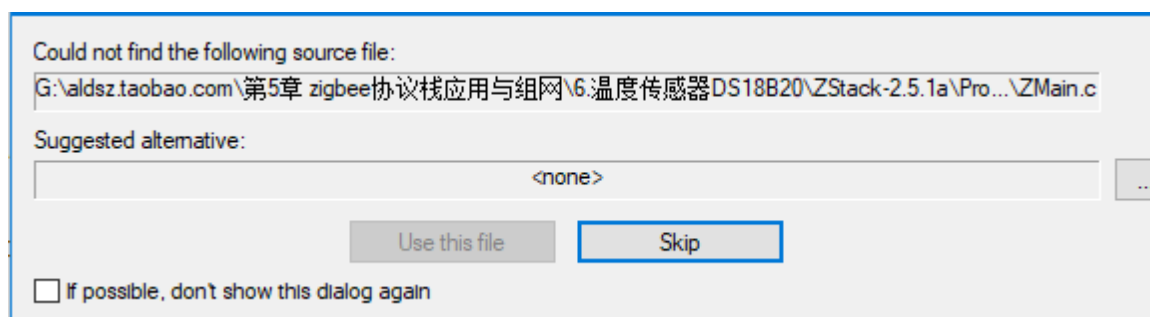
常见问题：

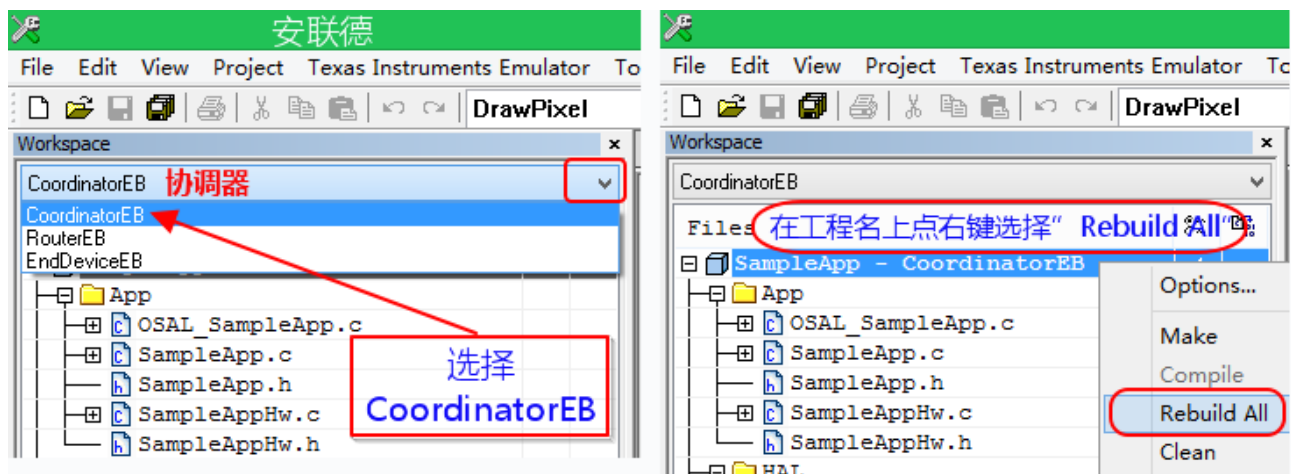
(一) 如图提示



此图为下载进度条，电脑配置不同时间不同，一般在几秒钟左右，耐心等待程序从电脑下载到开发板上

(二) 出现以图说明没有在自己的电脑上编译工程，选择相应的设备重新编译即可





(三) CoordinatorEB，EndDeviceEB 是怎么决定和区分的呢？

请看"zigbee 设备在 Zstack 中的体现.mht", 在后面实验中也增加这个文档的讲解，此实验内容过多就不讲了，有兴趣可以自行先看看。

(四) 厂家提供很多工程模板有什么区别？

看“GenericApp SampleApp SimpleAp 区别” 实际上学会了，用哪个模板都可以实现相同的功能，只是厂家提供模板让我们改动更少，使用更方便而已。

(五) 有些人可能有疑问了？虽然实验做出来了，不知道原理，没明白哦。

学习第 5 章做实验看现象，再看文档补充理论，看视频强化知识。慢慢来相信你一定会成功的，别急。

部分人觉得要背的太多，很容易忘记，其实根本不用背，里面寄存器太多了，有些芯片上千个，你实验做多了，看多了自然就记住了，会利用搜索查到就行了。

有些人学了第一遍感觉心里没底，如果基础不好，加上是新知识在部分人都这样的，可以将第 5 章前面几个实验手册多看几遍，加强印象，新知识需要时间的沉淀，过段时间回头再看你会恍然大悟。