

无线传感网-ZigBee

无线组网技术

- ▶ 无线组网提供灵活性、流动性，省去花在布线上的费用和精力，并且它也符合家庭网络通信的特点，因此，在无线传感器网络中，采用无线组网技术已经成为势不可挡的趋势。无线组网技术主要包括
- ▶ 802.11（WLAN）
- ▶ 蓝牙(BlueTooth)
- ▶ ZigBee等。

ZIGBEE技术

- ▶ 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。
- ▶ ZigBee联盟成立于2001年8月。2002年下半年，英国Invensys公司、日本三菱电气公司、美国摩托罗拉公司以及荷兰飞利浦半导体公司四大巨头共同宣布，它们将加盟“ZigBee联盟”，以研发名为“ZigBee”的下一代无线通信标准，这一事件成为该项技术发展过程中的里程碑。

ZigBee与IEEE 802.15.4标准的关系

- ▶ ZigBee的诞生和生长和两个组织密不可分，这两个组织分别是IEEE 802.15工作组和ZigBee联盟。
- ▶ 如下图所示，网络的最下面两层是由IEEE 802.15.4标准所定义的，该标准是由IEEE802标准委员会所开发并于2003年最初发布的，IEEE 802.15.4标准定义了无线网络PHY层和MAC层的详细信息，但它没有为更高的层规定任何要求。
。
- ▶ ZigBee标准仅仅定义了协议的网络层、应用层和安全层，并采用IEEE 802.15.4的PHY层和MAC层作为其部分协议。因此，任何遵循ZigBee标准的设备也同样遵循IEEE 802.15.4标准。

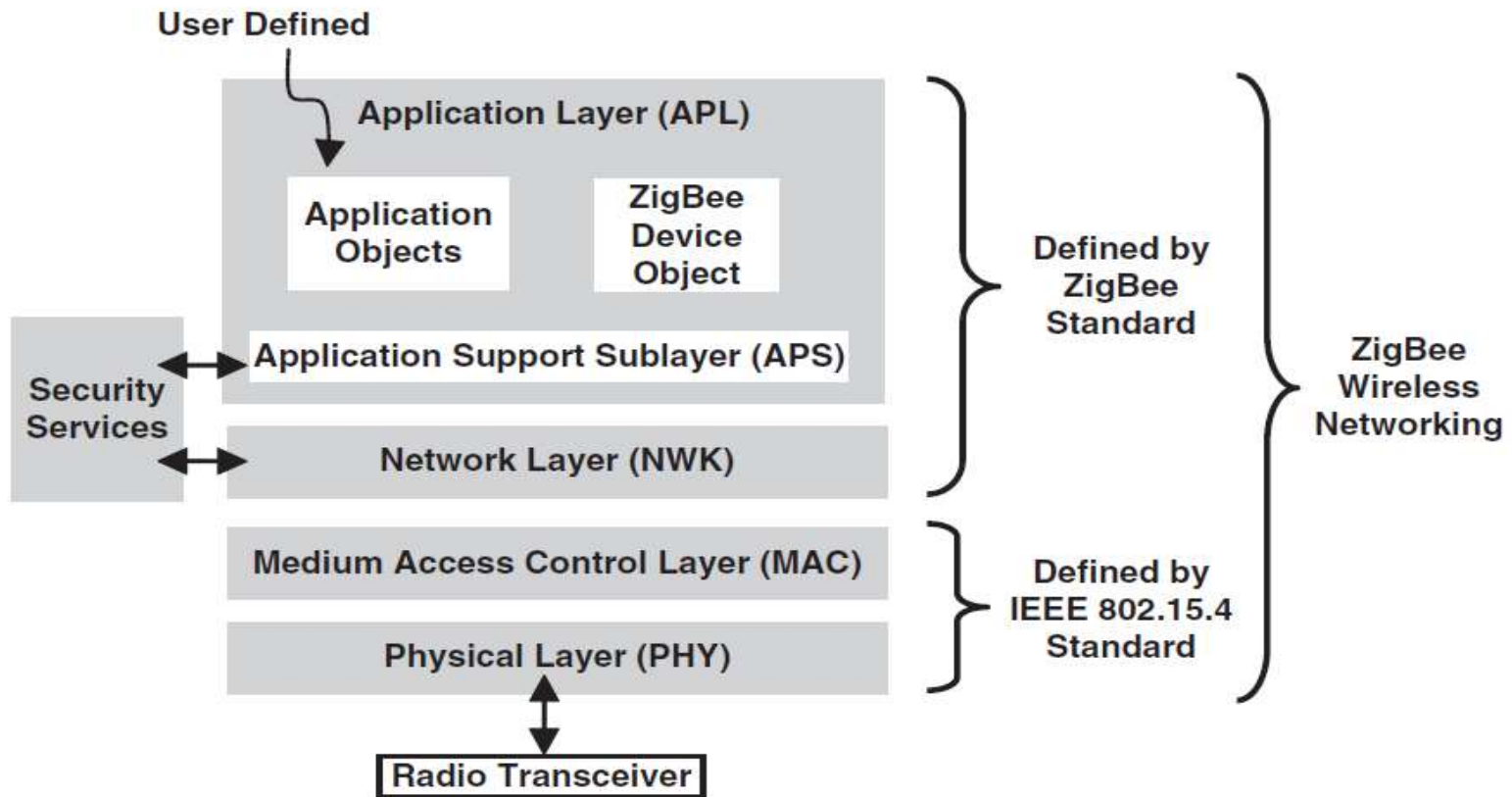


Figure 1.3: ZigBee Wireless Networking Protocol Layers

ZigBee技术介绍

- ▶ Zigbee自身的技术优势
- ▶ ①低功耗。在低耗电待机模式下, 2 节5 号干电池可支持1个节点工作6~24个月, 甚至更长。这是Zigbee的突出优势。相比较, 蓝牙能工作数周、[WiFi](#)可工作数小时。
- ▶ ②低成本。通过大幅简化协议(不到蓝牙的1/10), 降低了对通信控制器的要求, 按预测分析, 以8051的8位微控制器测算, 全功能的主节点需要32KB代码, 子功能节点少至4KB代码, 而且Zigbee免协议专利费。
- ▶ ③ 低速率。Zigbee工作在20~250 kbps的较低速率, 分别提供250 kbps(2.4GHz)、40kbps (915 MHz)和20kbps(868 MHz) 的原始数据吞吐率, 满足低速率传输数据的应用需求。
- ▶ ④近距离。传输范围一般介于10~100 m 之间, 在增加RF 发射功率后, 亦可增加到1~3 km。这指的是相邻[节点](#)间的距离。如果通过路由和节点间通信的接力, 传输距离将可以更远。
- ▶ ⑤短时延。Zigbee 的响应速度较快, 一般从睡眠转入工作状态只需15 ms , 节点连接进入网络只需30 ms , 进一步节省了电能。相比较, 蓝牙需要3~10 s、WiFi 需要3 s。
- ▶ ⑥高容量。Zigbee 可采片状和网状网络结构, 由一个主节点管理若干子节点, 最多一个主节点可管理254 个子节点;同时主节点还可由上一层用星状、网络节点管理, 最多可组成65000节点的大网个。
- ▶ ⑦高安全。Zigbee 提供了三级安全模式,包括无安全设定、使用接入控制清单(ACL)防止非法获取数据以及采用高级加密标准(AES 128) 的对称密码,以灵活确定其安全属性。
- ▶ ⑧免执照[频段](#)。采用直接序列扩频在工业科学医疗(ISM) 频段,2. 4 GHz (全球)、915 MHz(美国) 和868 MHz(欧洲) 。

ZigBee VS 蓝牙和IEEE 802.11

将ZigBee标准与蓝牙和IEEE802.11WLAN进行比较有助于我们理解ZigBee与现有一些标准的区别。下图总结了这几个标准的一些基本特性。

在这三种标准中，ZigBee具有最低的数据速率和复杂度，但提供了最长的电池寿命。由此我们知道ZigBee主要用在短距离无线控制系统，传输少量的控制信息。特别适用于电池供电的系统。

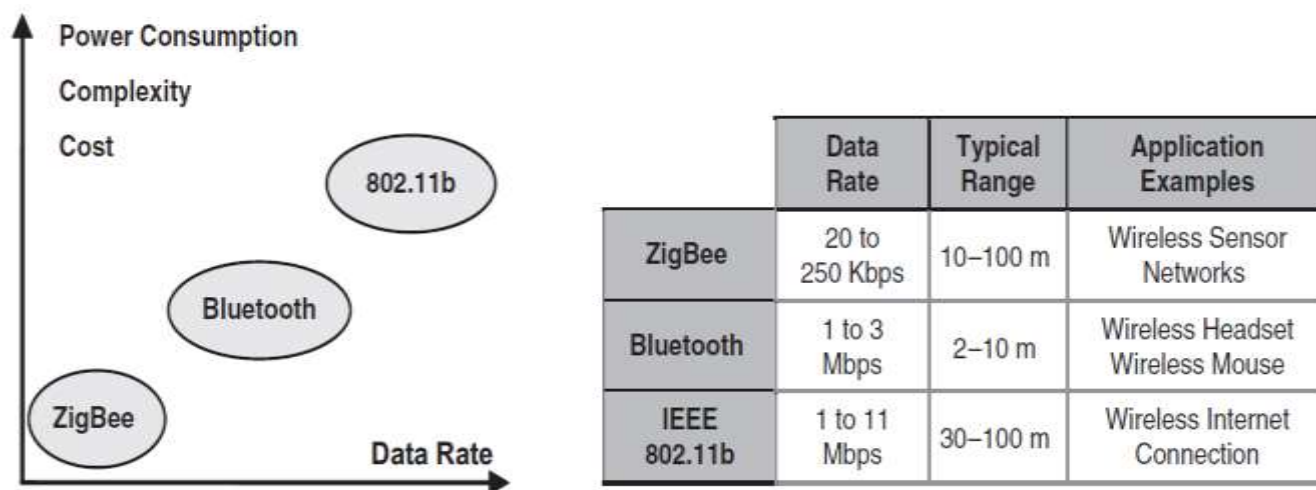
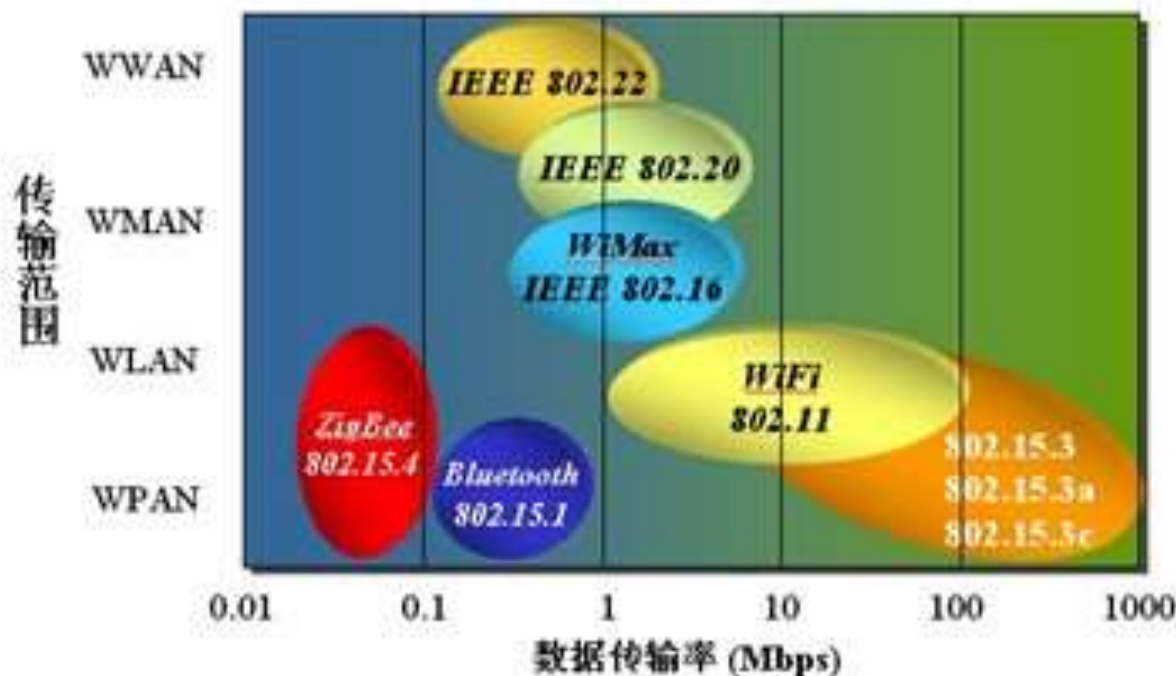


Figure 1.1: Comparing the ZigBee Standard with Bluetooth and IEEE 802.11b

短程无线网络的比较

ZigBee技术简介 — 短程无线网络标准



ZigBee 技术的应用定位是低速率、复杂网络、低功耗和低成本应用。

ZigBee 和 802.15.4 标准都适合于低速率数据传输，最大速率为 250K，与其他无线技术比较，适合传输距离相对较近。

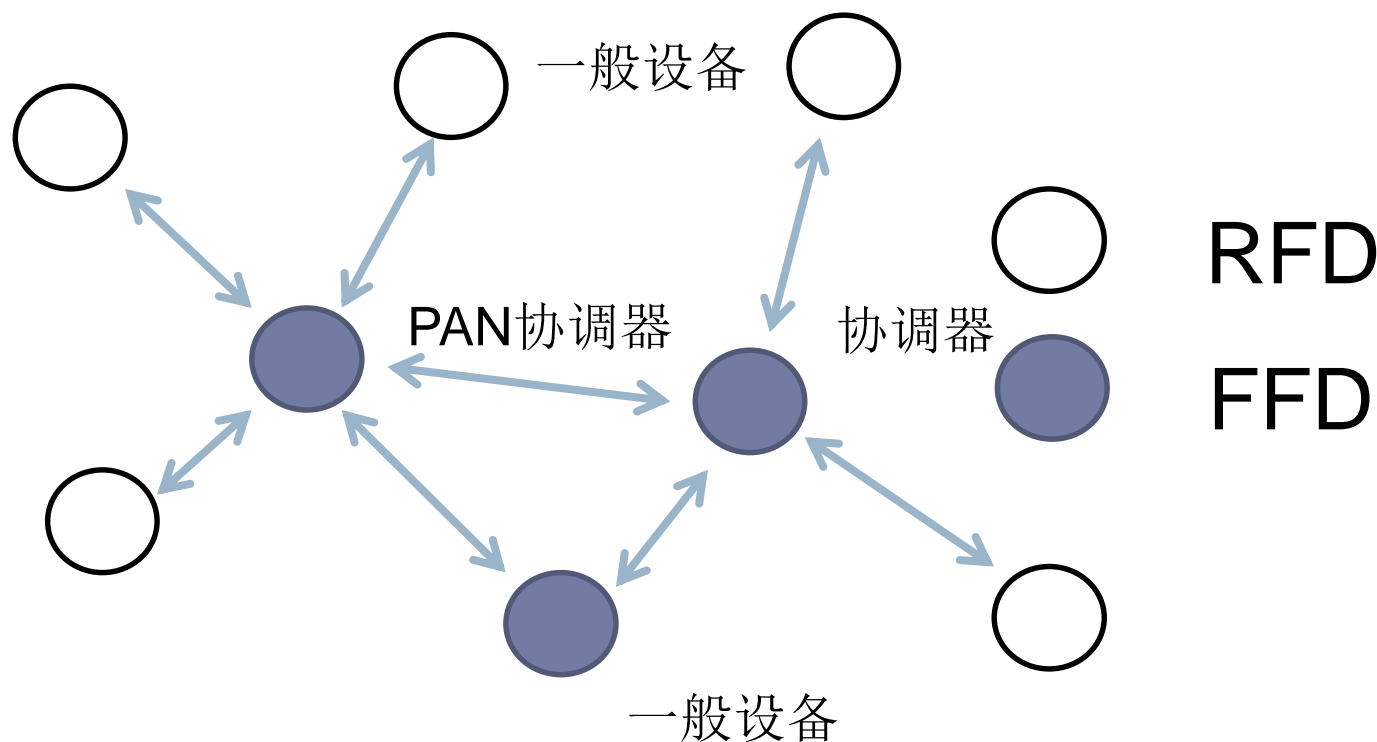
ZigBee 无线技术适合组建 WPAN 网络，就是无线个人设备的联网，对于数据采集和控制信号的传输是非常合适的。

ZigBee的应用

Zigbee 的市场，它有着广阔的应用前景。Zigbee 联盟预言在未来的并不是用来与蓝牙或者其他已经存在的标准竞争，它的目标定位于现存的系统还不能满足其需求的特定四到五年，每个家庭将拥有50个Zigbee器件，最后将达到每个家庭150个。其应用领域主要包括：

- ◆家庭和楼宇网络：空调系统的温度控制、照明的自动控制、窗帘的自动控制、煤气计量控制、家用电器的远程控制等；
- ◆工业控制：各种监控器、传感器的自动化控制；
- ◆商业：智慧型标签等；
- ◆公共场所：烟雾探测器等；
- ◆农业控制：收集各种土壤信息和气候信息；
- ◆医疗：老人与行动不便者的紧急呼叫器和医疗传感器等。

ZigBee的网络组件

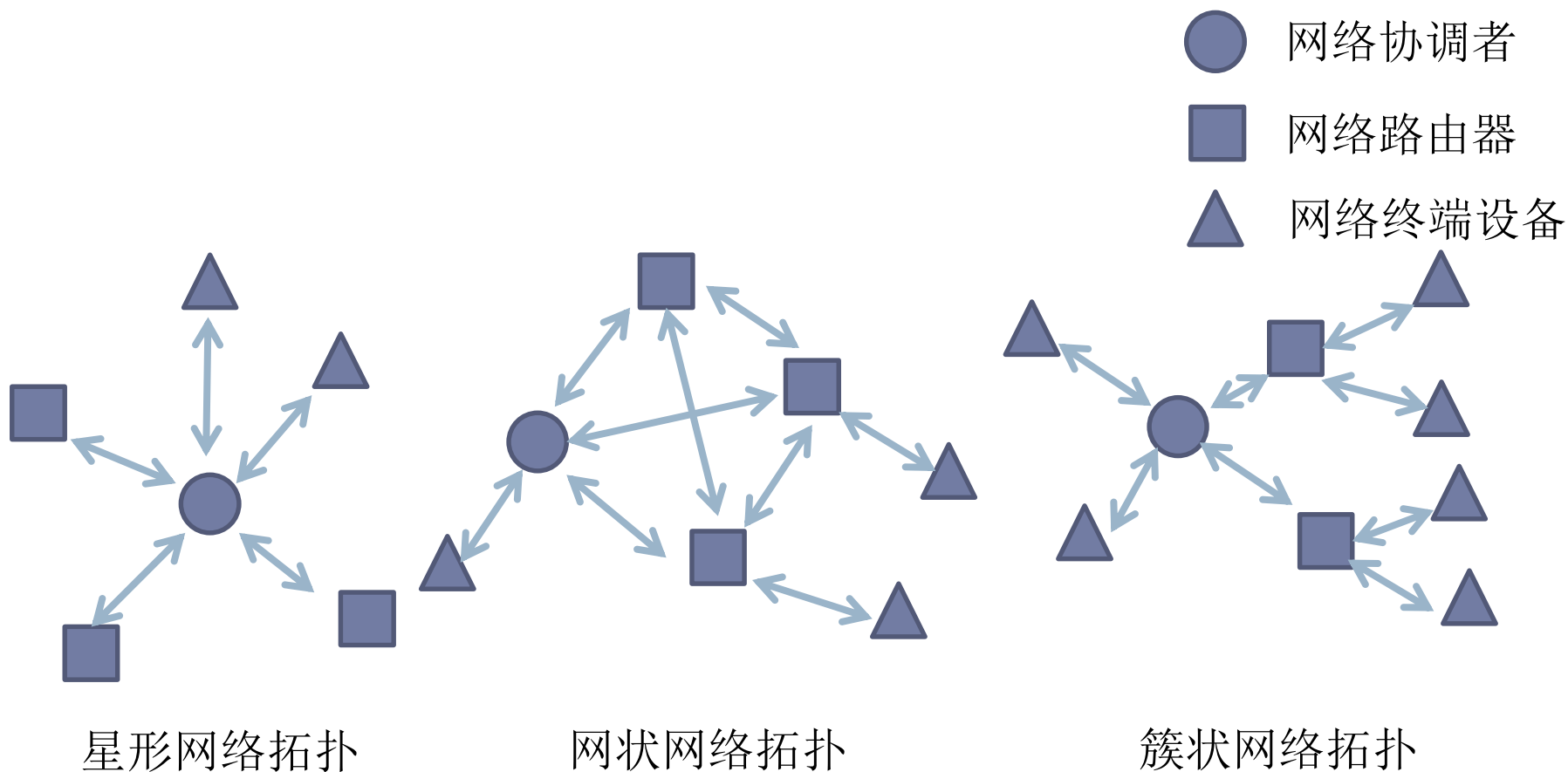


RFD:半功能设备

FFD:全功能设备

- ▶ IEEE 802.15.4无线网络中有两种设备类型：全功能设备（FFD）和精简功能设备（RFD）。FFD可以执行IEEE 802.15.4标准中描述的所有功能，并且可以扮演网络中的任何角色。另一方面，RFD只有部分功能，例如，FFD可以和网络中的任何设备通信，但RFD却只能和FFD设备通信。RFD设备的目的是应用于简单的应用中，如打开或闭合一个开关。RFD设备的处理能力和内存大小通常小于FFD设备。

ZigBee的网络拓扑结构



- 星状网络：在星状网络中，所有的终端设备都只与 PAN 协调器通信。且只允许 PAN 协调器与终端设备通信，终端设备和终端设备不能直接通信，终端设备间的消息通信需通过 PAN 协调器进行转发。
- 簇状网络：由一个 PAN 协调器和一个或多个星状网络结构组成。终端设备可以选择加入 PAN 协调器或者路由器。设备能与自己的父节点或子节点直接通信，但与其他设备的通信只能依靠树状节点组织路由进行。
- 网状网络：网状网络中任意两个路由器能够直接通信，且具有路由功能的节点不用沿着树来通信而可以直接把消息发送给其它的路由节点。

ZigBee通信频段和信道

频段就是设备在工作时可以使用的一定范围的频率段，信道是在这个频段内可供选择、用于传输信息的通道。

ZigBee物理层工作在868MHz、915MHz和2.4GHz这三个频段上，这三个频段共计分为27个信道，且分别拥有1个、10个、16个信道。

信道编号	中心频率/MHz	信道间隔/MHz	频率上限/MHz	频率下限/MHz
k=0	868.3	--	868.6	868.0
k=1,2,...,10	$906+2(k-1)$	2	928.0	902.0
k=11,12,...,26	$2401+5(k-11)$	5	2483.5	2400.0

ZigBee协议术语

- 规约 (profile)
- 协调器 (coordinator)
- 路由器 (router)
- 终端节点 (end)
- 设备 (device)
- 端点 (endpoint)
- 属性 (attribute)
- 簇 (cluster)

规约 (profile)

- ▶ 像学习一门新的语言一样，读者必须要知道这门语言的单词、语法规约含义才可以有效地与使用这门语言的人交流。规约就是对逻辑设备及其接口描述进行规定的协议集合，它是面向某个具体应用场合的公约、准则。它在分布式应用设备间的消息类型（向外部提供和接收什么消息）、消息格式、消息请求和应答、消息帧处理行为等方面达成了共识。**Profile**的目的是为了制定标准，以兼容不同制造商间的产品。

- ▶ 在ZigBee应用层规范中，应用对象可理解为在协议栈上运行的应用程序，并于ZigBee网络节点连接的设备对应，如传感器、灯、和控制器等。一旦确定了Profile，就决定了应用对象外部接口。一个Profile具有2字节（uint16）的Profile ID。标准的ProfileID是由ZigBee联盟分配的，并且为唯一标识号。
- ▶ 一般而言，profile指的是应用层profile，他说明了设备的类型、接口、数据传输等。ZigBee协议中还有一种特殊的Profile，它主要定义网络类型、网络深度等协议栈配置信息，其称为协议栈Profile

协议栈规约

- ▶ 协议栈规约（Stack Profile）的参数需要配置成特定的值，并由ZigBee联盟定义。在同一个网络中的设备必须使用相同的协议栈规约（即设备的协议栈参数必须配置成相同的值）。ZigBee联盟定义了两个不同的协议栈规约ZigBee 2007和ZigBee PRO，其目的是为了保证不同厂商产品的互操作性。如果设备都符合这个规范，在同一网络中的设备则能够与其他厂商生产的符合该规约的设备一起工作。但是，如果应用开发者改变了这些协议栈规约参数，他们的产品将不能再与遵从ZigBee特定协议栈规约的产品组成网络。改变之后的协议栈被称为“封闭网络”或“专用网络”协议栈规约。
- ▶ 设备可以通过信标帧获取协议栈规约标识符，并使得其能够在加入网络前确定协议栈规约。“专用网络”的协议标识符为0，ZigBee协议栈规约标识符为1，ZigBeePRO协议栈标识符为2。ZigBee2007协议栈规约标识符为HOME_CONTROLS，这是因为ZigBee规约首先在智能家居应用中使用，所以ZigBee协议栈规约标识符默认为智能家居的Profile ID。不同的协议栈规约对应不同的网络配置和功能特性。协议栈规约在nwk_globals.h文件中的STACK_PROFILE_ID参数项进行配置。

▶ 协议栈Profile参数在nwk_globals.h中的定义:

- ▶ `// 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`

- ▶ 一般来说，一个规约号为m的设备（如ZigBeePRO, 2）能够加入一个同样规约号为m的网络。如果一个规约号为m的路由器（如ZigBeePRO,2）加入一个具有不同规约号n（如ZigBee-2007,1）的网络，它将以不睡眠终端设备的身份加入网络。一个规约号为m的终端设备始终以终端设备的身份加入到具有不同规约号n的网络中。

协调器 (coordinator)

- ▶ 每个zigbee网络只允许有一个zigbee的协调器，协调器首先选择一个信道和网络标识(PAN ID)，然后开始这个网络.因为协调器是整个网络的开始，它具有网络的最高权限，是整个网络的维护者，还可以保持间接寻址用的表格绑定，同时还可以设计安全中心和执行其他动作，保持网络其他设备的通信。

路由器（router）

- ▶ 路由器是一种支持关联的设备，既可以做为普通设备使用，也能够实现其他节点的消息转发功能。Zigbee的树形网络可以有多个zigbee路由器设备，星型网络不支持zigbee的路由器设备。

终端节点（end）

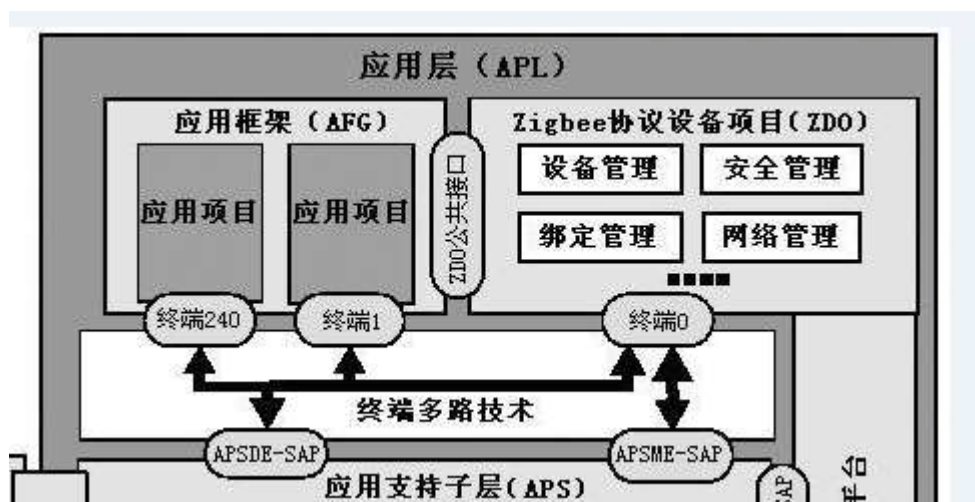
- ▶ 终端节点是一个ZigBee网路的最终端，完成用户功能，比如信息的收集，他不能转发其他节点的消息

设备/节点（device）

- ▶ 一般来说一个节点（FFD/RFD）就是一个设备，对应于一个无线单片机，如CC2530；一个设备有一个无线射频端，具有唯一的IEEE地址（Mac 地址）和网络地址（短地址）。
- ▶ 因为每一个节点可能存在着多种功能，所以在在一个成熟的分布式控制网络中，所有的节点中的应用程序必须保存多个数据链路，也就是说可以将我们的每一个节点能够同时实现多个功能。

端点 (Endpoint)

- ▶ 端点是节点真正的收发数据的目标。0号端点是保留给ZDO (zigbee device object) 使用的端点号，255号端点用于广播，1~240为用户可以使用的端点号（在一个ZDO可以有多个EP），其余号保留。

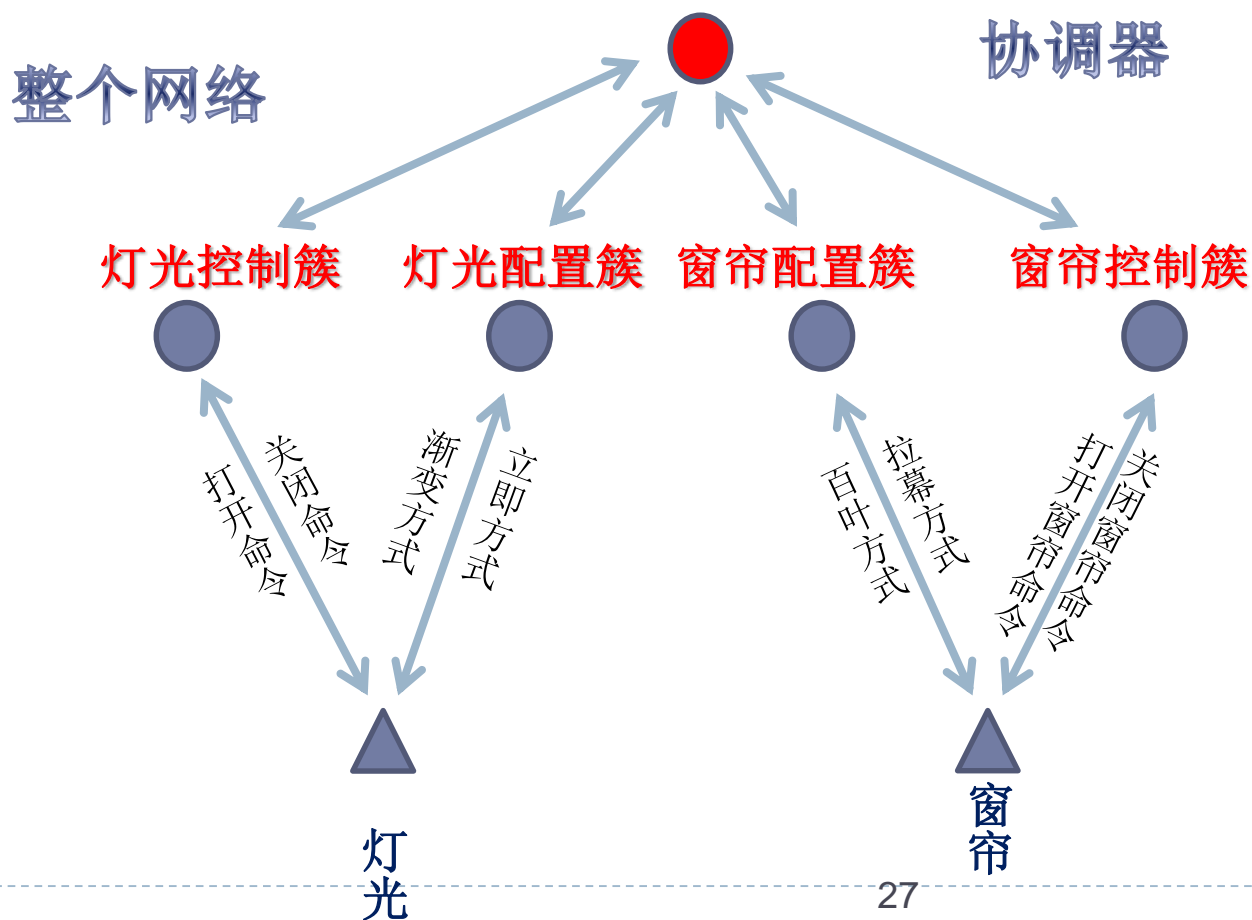


配置文件（**profile**）

- ▶ 它是面向某个应用类别的公约、准则是对逻辑设备及其接口描述的集合，比如：智能家居（profileID=0x104）。
- ▶ 作用：它使得节点在消息、消息格式、请求数据或请求创建一个共同的分布式应用程序的处理行为上达成了共识。

簇 (Cluster)

- ▶ 簇是一个Profile大方向下的一个特定对象，比如智能家居（profile）里面的灯光控制（灯光簇）、温湿度控制（温度簇）、窗帘控制（窗帘簇）
- ▶ Profile规范了应该包括哪些cluster。每个簇都有不同的命令。

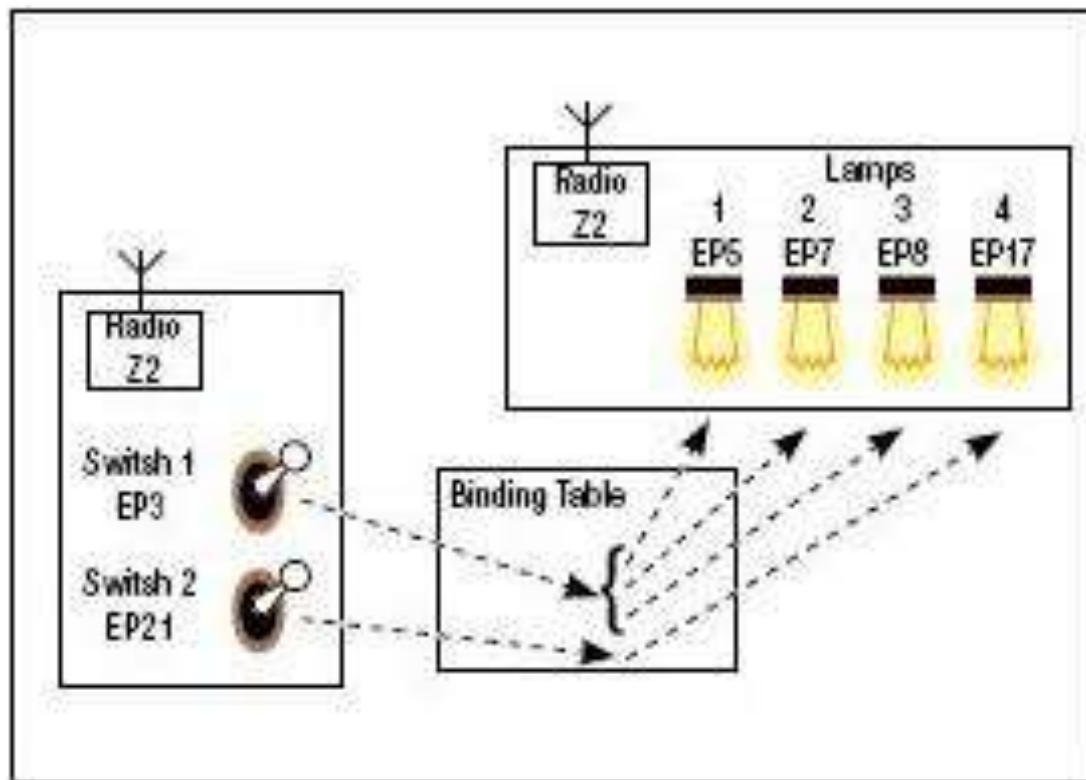


属性 (Attribute)

- ▶ 属性是对于我们节点的状态的描述
- ▶ 比如对于一个控制灯光的节点，我们需要描述这个灯的各种属性：比如网络地址，自身短地址，位置，命令接口，等等。
- ▶ 在我们的程序当中我们通常是用一个或者几个嵌套的结构体去描述节点的各种属性。

绑定

绑定是ZigBee协议中定义的一种特殊的操作，它能够通过使用ClusterID为不同节点上的独立端点建立一个逻辑上的连接。



- ▶ 绑定允许应用程序不需要知道目标地址，而向目标节点发送一个数据包。在TCP/IP socket通信中，客户端和服务端也可以建立绑定。在绑定完成后，应用程序直接调用send/recv函数发送/读取数据，不需要进行过多的操作。绑定寻址是通过APS层从它的绑定表中确定目标地址，然后向目标设备或者目标组发送数据。在ZigBee 2004协议中，绑定表只存在于协调器中。在ZigBee2006以后的协议版本中，绑定表保存在发送信息的设备中，这种绑定方式称为源绑定方式。源绑定方式需要在f8wConfig.cfg配置文件中添加REFLECTOR编译选项完成。

协议栈简介

- ▶ Z-Stack协议栈的开发是ZigBee模块软硬件开发的重要部分，其主要是要搞清楚ZigBee协议栈的结构和协议栈各个层之间的功能和重要函数关系。特别是利用协议栈能完成什么功能，要完成什么内容，以及怎样完成等。
- ▶ Z-Stack协议栈是ZigBee协议的一种具体实现。它是一种半开源协议栈，其安全子模块，路由模块，Mesh网络支持等关键代码都以库的方式封装，不能随便修改。用户一般只需在应用层进行编写即可。这种协议栈稳定性高，成本低，适用于工程人员使用。
- ▶ 另外，Freescale公司的协议栈BeeStack不提供源代码，但会提供一些封装函数。许多组织也实现了免费的开源协议栈，如密西西比大学R.Reese教授开发的msstatePAN协议栈。msstatePAN协议栈是为广大无线技术爱好者开发的精简版ZigBee协议栈，它是基于标准C语言编写的，具备ZigBee协议标准所规定的基本功能。Freakz也是一个开源ZigBee协议栈，它配合Contik操作系统运行。这种模式与Z-Stack+OSAL类似，非常适合读者深入学习。

Z-Stack与ZigBee规范版本对照表

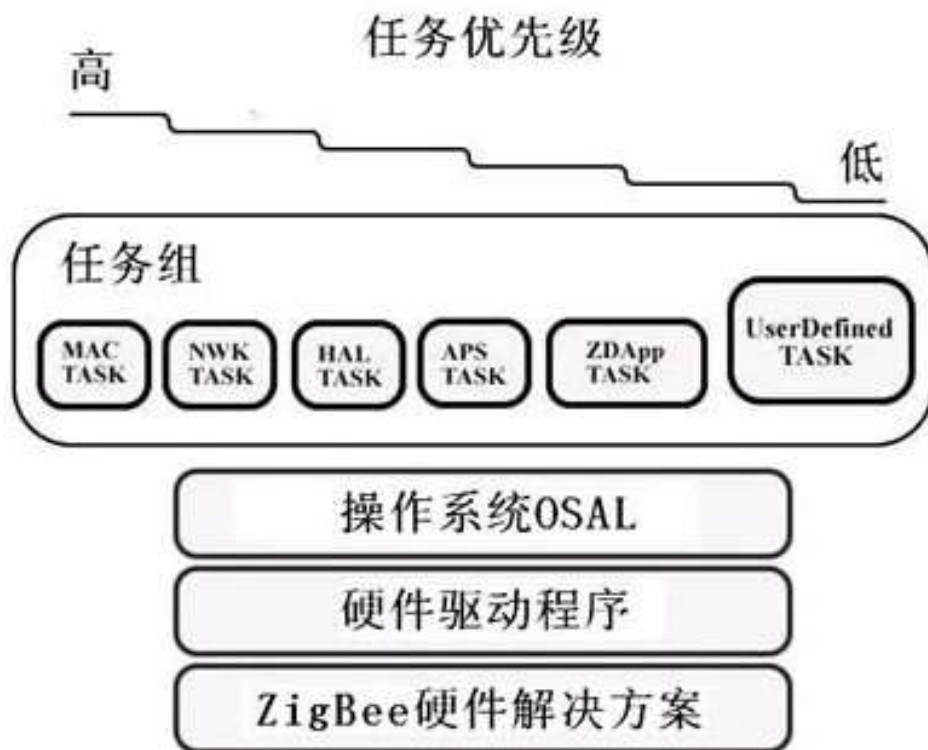
TI公司针对ZigBee协议规范实现了相应的ZigBee协议栈（Z-Stack）。ZigBee协议规范目前按照发布年份主要分为三种：ZigBee 2004， ZigBee2006和ZigBee 2007.

ZigBee规范版本	ZigBee 2004	ZigBee 2006	ZigBee 2007
协议栈Profiles	Home Controls	ZigBee	ZigBee、 ZigbeePro
应用Profiles	家庭灯光控制	自动家居、 制造业特定Profile	自动家居、 商业建筑自动化、 敏捷能源
Z-Stack版本	1.3.X-	1.4.X+	2.X.X+
发布	2005	2006	2008.3

ZigBee2007框架

ZigBee2007协议栈的介绍

- ▶ 它是一个公开的半开源的无线通信协议
- ▶ 它源于一个最简单的操作系统思想（轮询）

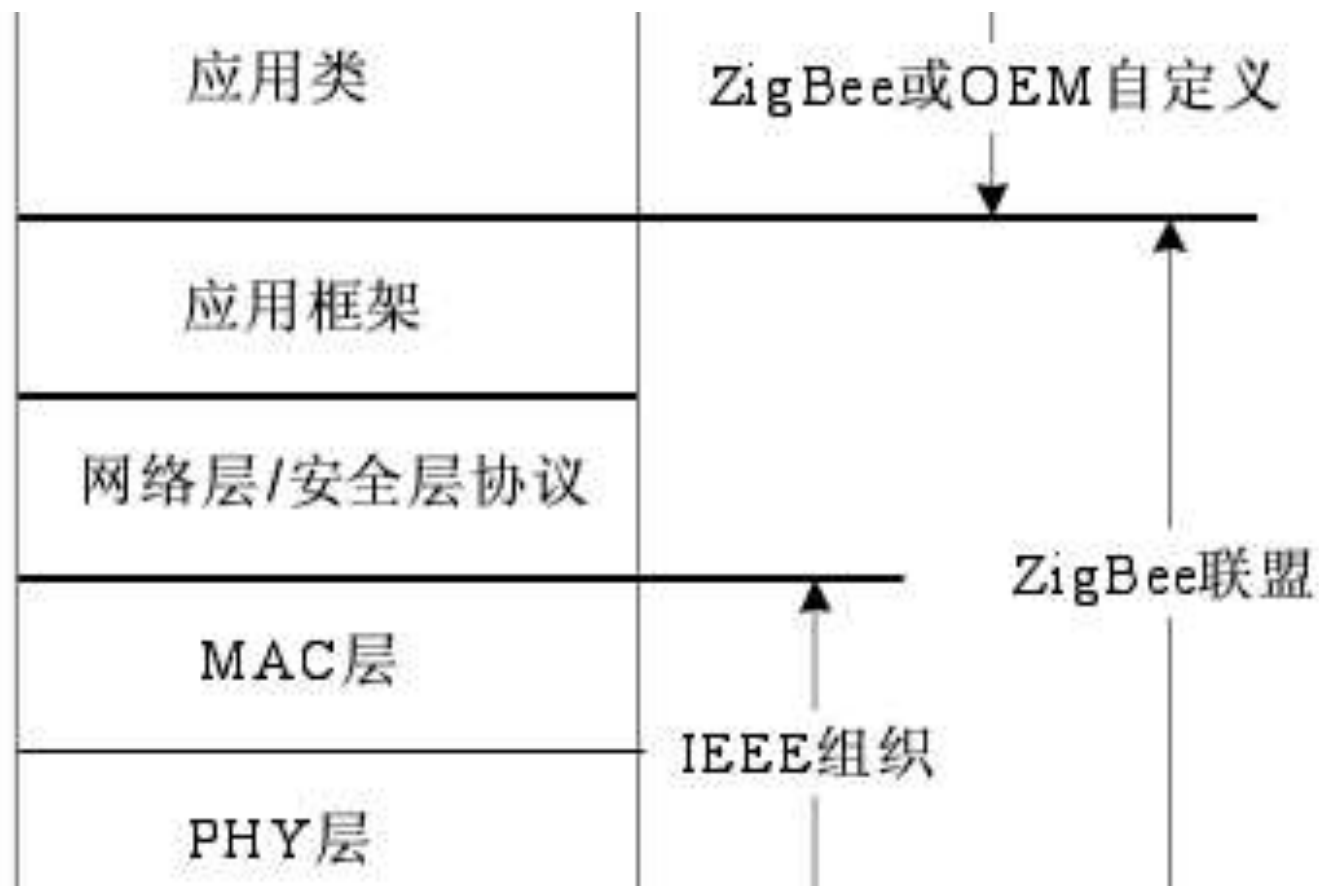


ZigBee 2007 协议栈中的各个层次

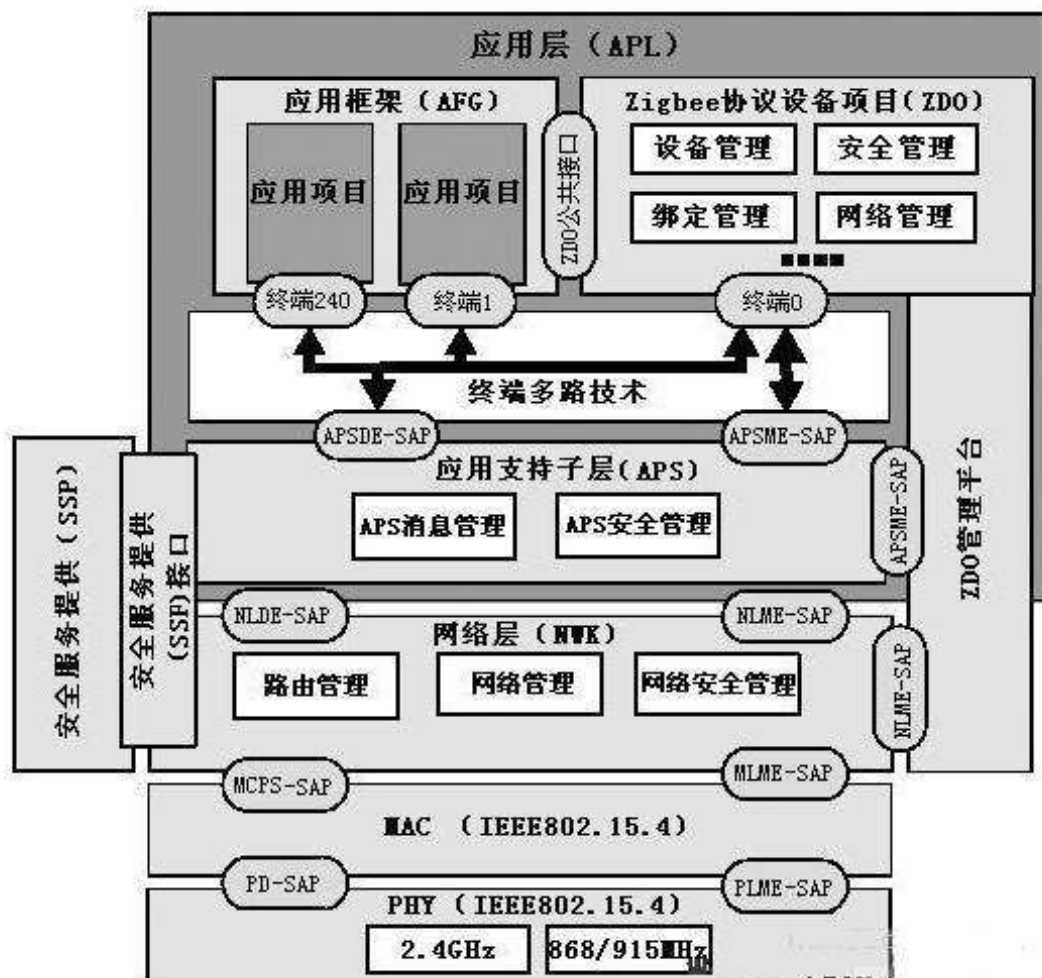
- (1)应用层
- (2)网络层
- (3)IEEE MAC层
- (4)IEEE PHY层

基于标准的系统开放互联模型（OSI）,在IEEE 802.15.4 2003 标准定义了MAC 层、PHY层，zigbee联盟定义了应用层、网络层，其中应用层包括应用支持子APS（Application Support Sublayer）、zigbee设备对象ZDO（Zigbee Device Objects）、制造商定义的应用对象。

ZigBee 2007 协议栈中的各个层次



ZigBee 2007 协议栈中的各个层次



应用层

ZigBee应用层由三个部分组成：

- ZDO（包含ZDO管理平台）
- APS子层
- 制造商定义的应用对象。

应用层—设备对象层（ZDO）

- ▶ 提供了管理一个 ZigBee 节点所要使用的功能函数。ZDO API 为协调器、路由器和终端设备提供了应用端点的管理函数,其中包括:建立、发现和加入一个ZigBee 网络,绑定应用端点和安全管理
- ▶ ZDP（ZigBee Device Profile）层描述了 ZDO 内部ZigBee 设备功能是如何实现的。它定义了使用命令和响应对的设备描述和簇。ZDP 为 ZDO 和应用程序提供如下功能:
 - 设备网络启动
 - 设备和服务发现
 - 终端设备绑定和取消绑定服务
 - 网络管理服务

应用层—应用支持子层 (APS)

- ▶ 应用支持子层给网络层和应用层通过ZigBee设备对象和制造商定义的应用对象使用的一组服务提供了接口。应用支持子层提供这些服务：
 - ▶ 数据服务
 - ▶ 管理服务
 - ▶ APS数据实体DE(Data Entity)通过与之连接的APSDE-SAP (Service Access Point) 来提供数据传输服务。
 - ▶ APS管理实体ME (Manage Entity) 通过与之连接的APSME-SAP提供管理服务
-



端点管理

- ▶ 在 ZigBee 网络中每个设备都是一个节点,每个节点具有唯一的一个 IEEE 地址(64 位)和一个网络地址(16 位)。网络中的其他节点发送数据时必须指定目标节点的短地址,数据才能被接收。每个节点有 241 个端点,其中端点 0 由 ZDO 层使用,它是不可缺少的。端点 1~240 由应用程序分配使用,在 ZigBee 网络中应用程序必须登记注册一个或多个端点,这样才能发送和接收数据。

网络层功能

- ▶ ZigBee网络层主要功能是发现设备并建立设备间无线链路，其网络层支持三种网络拓扑结构，星型结构（star）、簇状结构（Cluster tree）和网状结构（Mesh）。其中树形结构和网状结构都是属于点对点的拓扑结构，它们是点对点拓扑结构的复杂化形式。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。

IEEE MAC层

- 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。



ZigBee寻址方式

- ▶ ZigBee网络有两种类型的地址：扩展地址、短地址。
- ▶ 扩展地址:又称IEEE地址、MAC地址。扩展地址位数为64位，由设备商固化在设备里。任何ZigBee网络设备都具有全球唯一的扩展地址，在PAN网络中，此地址可以直接用于通信。
- ▶ 短地址:又称为网络地址,它用于在本地网络中标识设备节点。短地址位数为16位。在协调器建立网络后，使用0x0000作为自己的短地址，在设备需要关联得时候，由父设备分配16位短地址。设备可以使用16位短地址在网络中进行通信。不同的ZigBee网络可能具有相同的短地址。

ZigBee硬件开发平台

ZigBee常用射频芯片介绍

ZigBee应用前景广阔，市场需求巨大，世界各大半导体生产商纷纷推出了支持IEEE802.15.4标准的无线芯片，比如TI公司的CC2420、CC2430和CC2530，Freescale公司的MC13191/13192/13193和MC13211/13222/13223/13224，Ember公司的EM250和EM351/357等。这些芯片集成了ZigBee物理层的功能，并且所需外围器件少，使用起来方便，大大降低了射频电路设计、制作的难度。即使没有相关射频（RF）专业知识和昂贵的射频仪器，用户也能快速实现嵌入式ZigBee特定领域的应用。

ZigBee模块与射频芯片不是同一个概念。ZigBee射频芯片是符合IEEE802.15.4标准、具有数据调制调解功能的射频收发芯片，但是不能够直接用来收发数据。ZigBee模块是在射频芯片的基础上，增加一些外围电路、传感器、外置天线等器件的模块。



ZigBee技术应用开发主要涉及两方面内容：**ZigBee**协议栈和承载协议栈运行的硬件芯片模块。**ZigBee**协议栈主要完成网络建立、数据路由等通信功能，而硬件芯片模块承载**ZigBee**协议栈运行并完成相关数据的感知功能。以本书使用的**Z-stack+CC2530**的模式为例，读者主要需要学习**Z-Stack**协议栈网络编程以及**CC2530 8051**单片机编程。**ZigBee**技术是一个软硬件结合的技术，读者需要认真学习这两方面的内容。

如果初学**ZigBee**技术，读者可以直接购买市面上提供的**ZigBee**模块，并使用**TI**等公司提供的**Z-Stack**协议栈源代码和实例程序，在**IAR**等开发环境下编写应用程序。这样读者可以尽快地入门并进入开发阶段，能够有效地缩短开发时间，降低项目开发成本。

当然，如果具备高频设计方面的知识和经验，用户也可以采用**CC2530**等芯片进行模块设计，这可以使得设备成本降低和产品化。其优点是模块结构更为灵活，如何调整模块尺寸和外观，以便嵌入到其他设备（如玩具、灯具、仪表等设备）中；与其他电子产品（如**RFID**读卡器，网关等设备）进行一体化设计；根据应用需求，在模块上增加和减少各种类型的传感器。用户可以根据项目需求、开发成本、日后拓展等因素决定是否要进行模块设计。

CC2530芯片介绍



CC2530是专门针对IEEE 802.15.4和Zigbee应用的单芯片解决方案，经济且低功耗。

CC2530有四种不同的版本：CC2530-F32 / 64 / 128 / 256。分别带有32 / 64 / 128 / 256 KB的闪存空间；它整合了全集成的高效射频收发机及业界标准的增强型8051微控制器，8 KB的RAM和其他强大的支持功能和外设。

主要特点：

- 高达256kB的闪存和20kB的擦除周期，以支持无线更新和大型应用程序
- 8kB RAM用于更为复杂的应用和Zigbee应用
- 可编程输出功率达+4dBm
- 在掉电模式下，只有睡眠定时器运行时，仅有不到1uA的电流损耗
- 具有强大的地址识别和数据包处理引擎

利益：

- 支持Zigbee / Zigbee PRO , Zigbee RF4CE, 6LoWPAN, WirelessHART 及其他所有基于802.15.4标准的解决方案；
- 卓越的接收机灵敏度和可编程输出功率；
- 在接收、发射和多种低功耗的模式下具有极低的电流消耗，能保证较长的电池使用时间；
- 一流的选择和阻断性能（50-dB ACR）

应用：

- 智能能源/自动化仪表读取
- 远程控制
- 居家及楼宇自动化
- 消费类电子产品
- 工业控制及监测
- 低功耗无线传感器网络



硬件开发平台介绍

- ▶ 单纯地学习ZigBee理论犹如“纸上谈兵”，读者还需要在合适的硬件平台上下载、调试程序来实践ZigBee技术。现在介绍一套由华清远见研发中心开发设计的ZigBee开发模块，其包括CC2530最小模块和学习底板。
- ▶ CC2530模块-图中绿色板子所示，它采用TI公司的ZigBee芯片CC2530（红色方框内）。
- ▶ 学习底板-图中蓝色板子所示，可外接各种传感器实现数据采集的功能，也可以接入控制设备实现远程控制的功能。

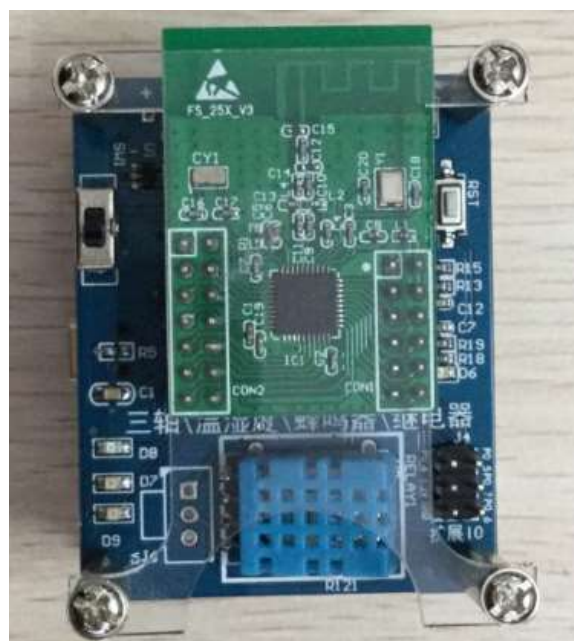


ZigBee学习底板

光线感应器终端



温湿度采集终端



风扇控制终端





ZigBee开发环境搭建

软件开发环境主要包括Z-Stack协议栈、IAR集成开发环境。另外，TI还提供了许多实用的软件工具，如协议栈分析仪Packet Sniffer、烧写工具Flash Programmer等。我们需要熟练掌握以下内容：

- ◆ IAR的工程建立与修改
- ◆ 工程的下载与调试



TI Z-Stack协议栈安装

TI 主页 > 半导体 > 无线连接 > Z-Stack - ZigBee 协议栈

Z-Stack - ZigBee 协议栈

(正在供货) Z-STACK



描述/特性



技术文档



支持和社区

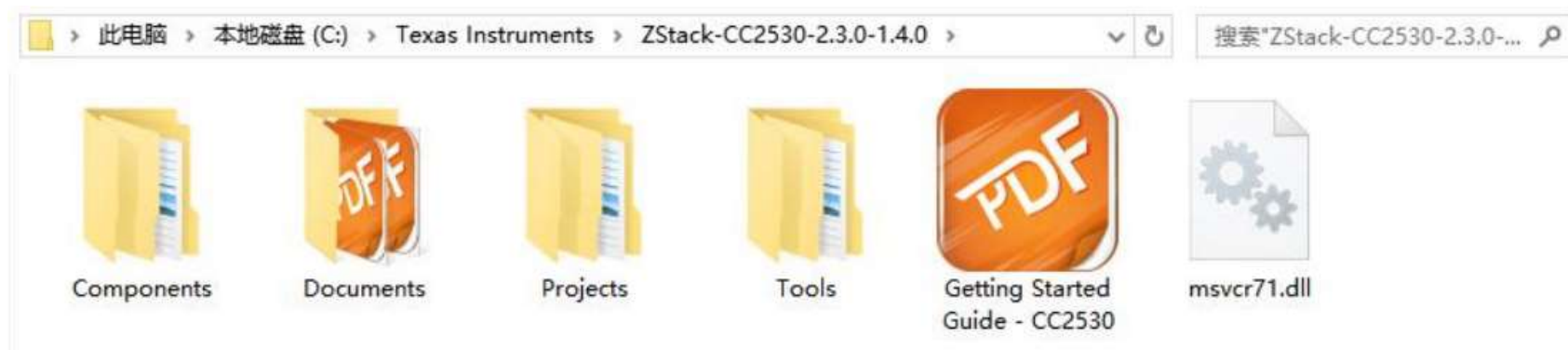
立即订购

器件型号	从德州仪器 (TI) 或第三方购买	通知我	状态
Z-STACK-LINUX-GATEWAY: Z-Stack™Linux Ubuntu Gateway installer	免费 下载	通知我	ACTIVE
Z-STACK-HOME: ZigBee Home Automation Solutions	免费 下载	通知我	ACTIVE
Z-STACK-LIGHTING: ZigBee Light Link Solutions	免费 下载	通知我	ACTIVE
Z-STACK-MESH: ZigBee Mesh Solutions	免费 下载	通知我	ACTIVE
Z-STACK-ENERGY: ZigBee Smart Energy Solutions	免费 下载	通知我	ACTIVE

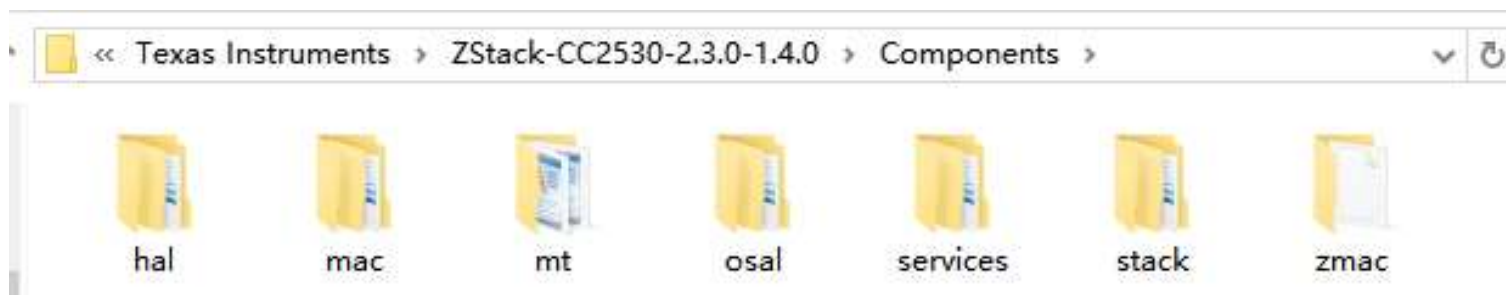
- ▶ Z-Stack安装包可以通过登录TI公司的官方网站进行免费下载:
<http://www.ti.com.cn/tool/cn/z-stack>
- ▶ 我们选择Z-Stack版本是Zstack-CC2530-2.3.1-1.4.0,默认必须安装在C盘,并且安装路径尽量不要出现汉字或者空格.

Z-Stack安装目录结构

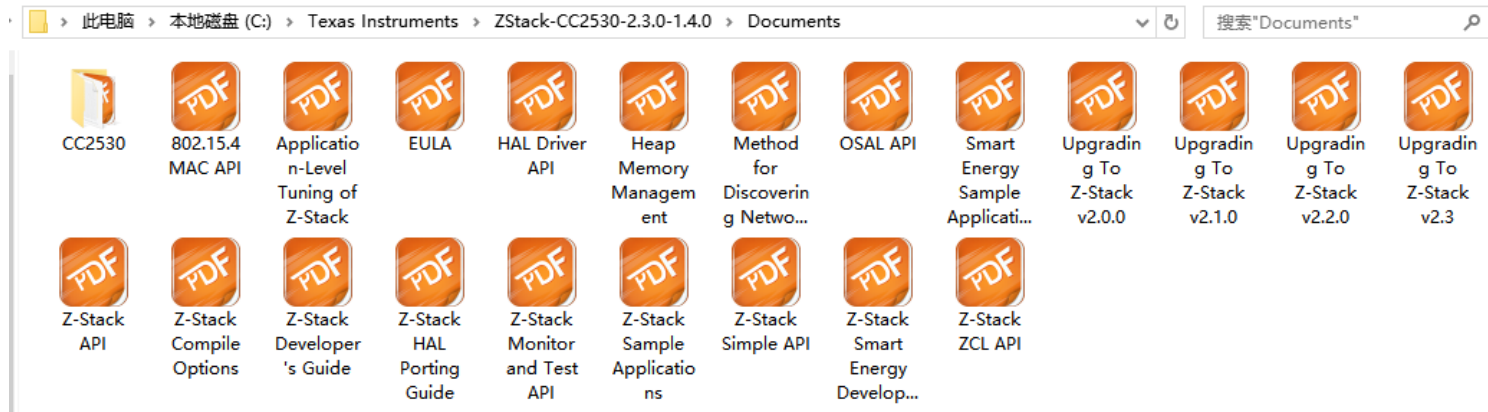
- ▶ 安装完成后,我们在 C 盘的根目录下出现了 Texas Instruments 目录,我们进入该文件夹。由于我们安装的协议栈的版本是 ZStack-CC2530-2.3.0-1.4.0,将会出现一个相同命名的文件夹。打开文件夹,文件目录结构如下图所示:



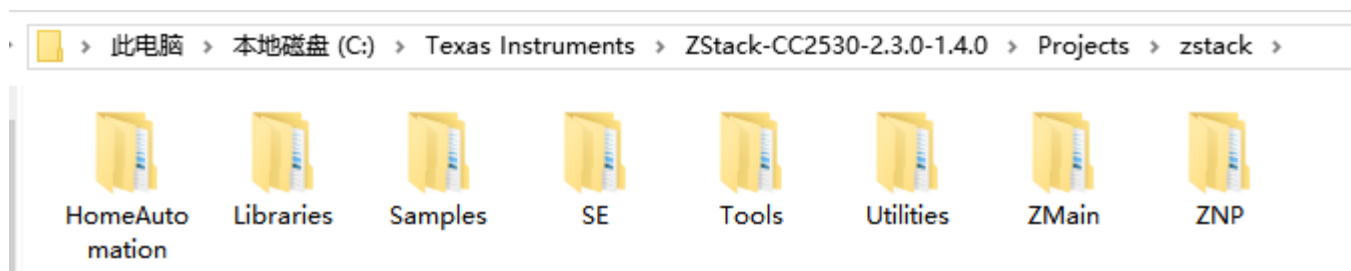
- **Components** 文件夹:存放 Z-Stack 开源的主要程序代码,主要包括硬件接口层、MAC层、操作系统代码等。结构如下图所示:



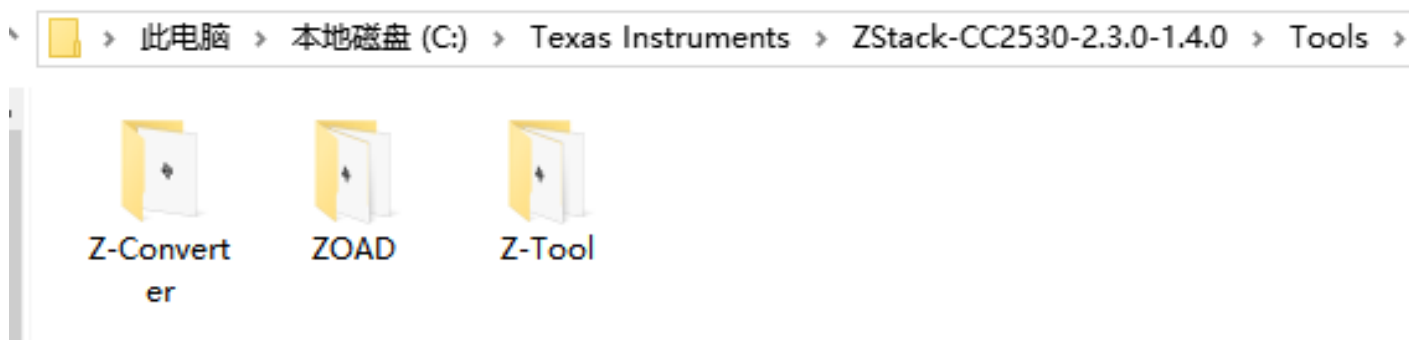
- **Documents** 文件夹: 这个目录中的文档在开发中很实用, 希望读者可以查阅学习。



- ▶ **Projects 文件夹：** 存放用户自己的工程，同时也包含了 TI 公司提供的几个官方例程和编程模板，我们以后的例程都是基于 Samples 中的工程模版。



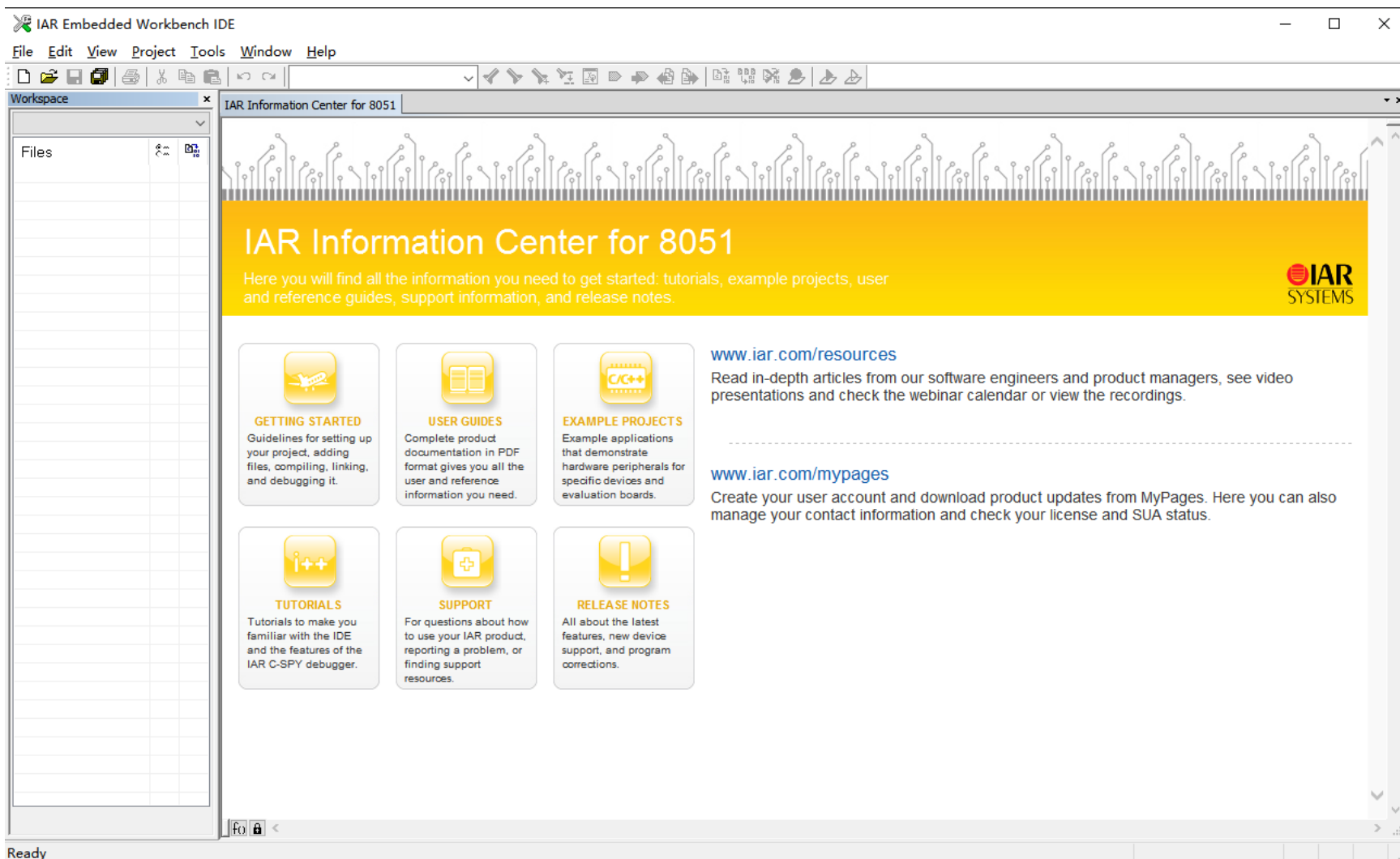
- ▶ **Tools 文件夹：** 存放了几个 ZigBee 相关的实用工具，读者可以根据文档自行安装使用。



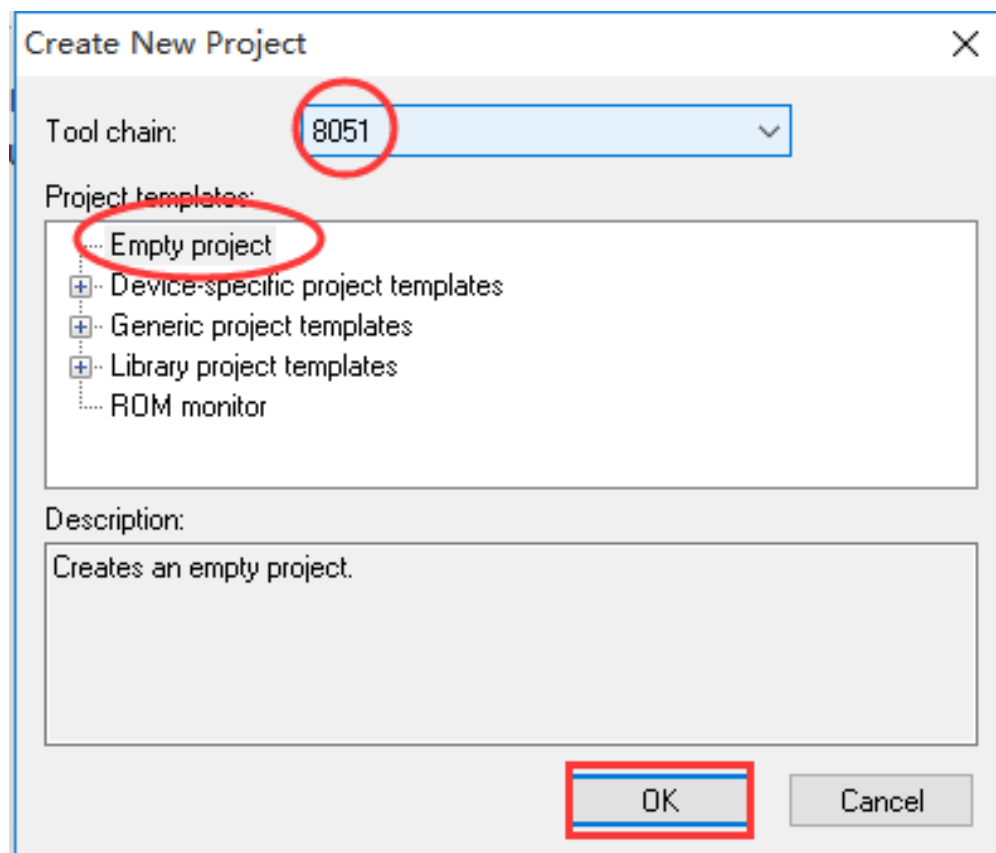
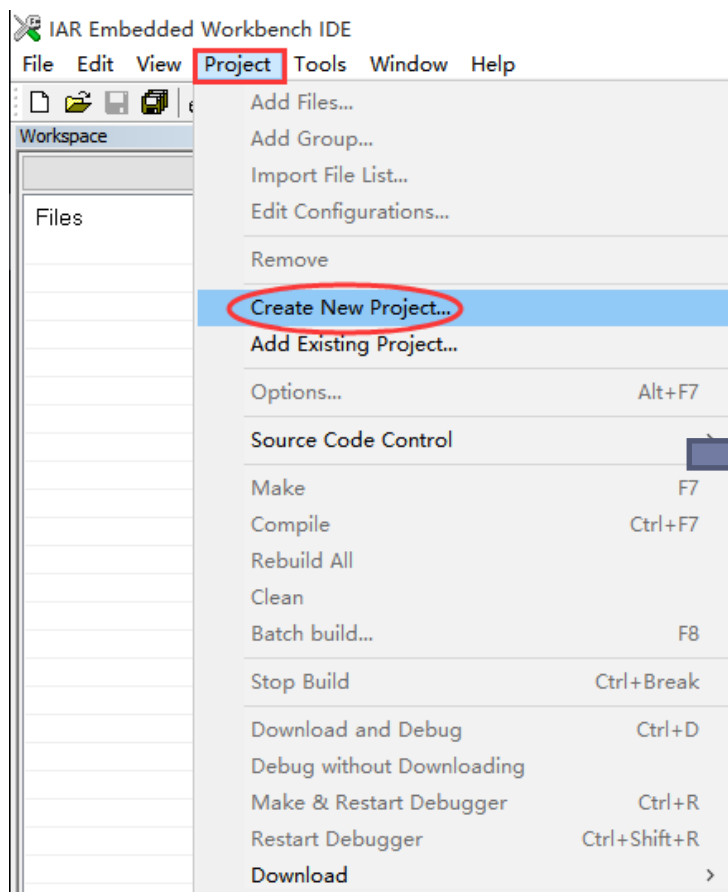
IAR 集成开发环境

- ▶ 嵌入式 IAR Embedded Workbench 适用于大量 8 位、16 位以及 32 位的微处理器和微控制器，使用户在开发新的项目时也能在所熟悉的开发环境中进行。它为用户提供一个易学和具有最大量代码继承能力的开发环境，以及对大多数和特殊目标的支持。嵌入式 IAR Embedded Workbench 有效提高用户的工作效率，通过 IAR 工具，用户可以大大节省工作时间。我们称这个理念为：“不同架构，同一解决方案”。
- ▶ 嵌入式 IAR Embedded Workbench IDE 提供一个框架，任何可用的工具都可以完整地嵌入其中，这些工具包括：
 - ▶ 高度优化的 IAR AVR C/C++ 编译器；
 - ▶ AVR IAR 汇编器；
 - ▶ 通用 IAR XLINK Linker；
 - ▶ IAR XAR 库创建器和 IAR XLIB Librarian；
 - ▶ 一个强大的编辑器；
 - ▶ 一个工程管理器；
 - ▶ TM IAR C-SPY 调试器；

IAR软件界面

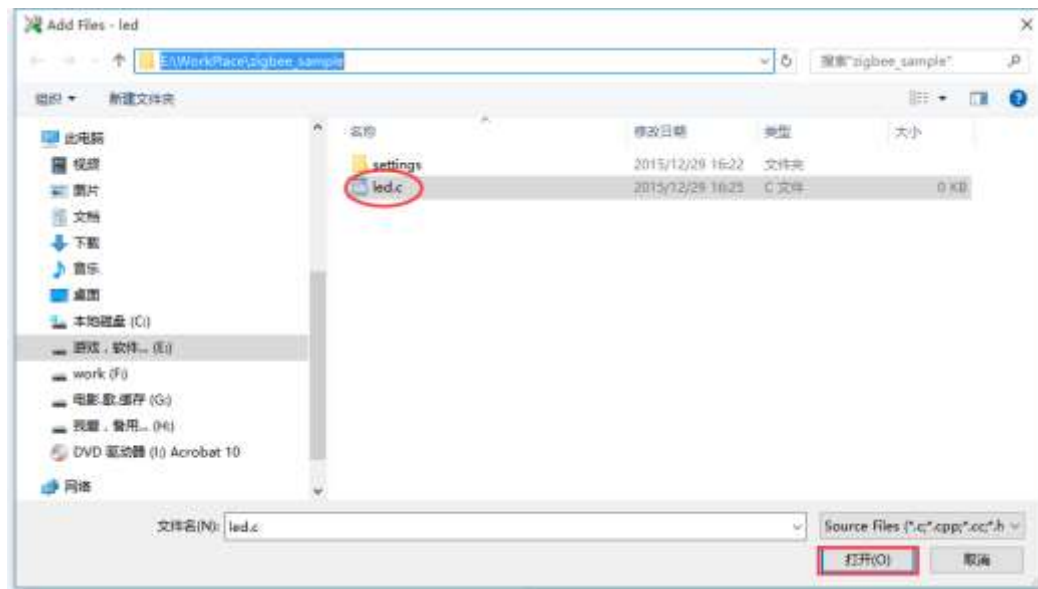
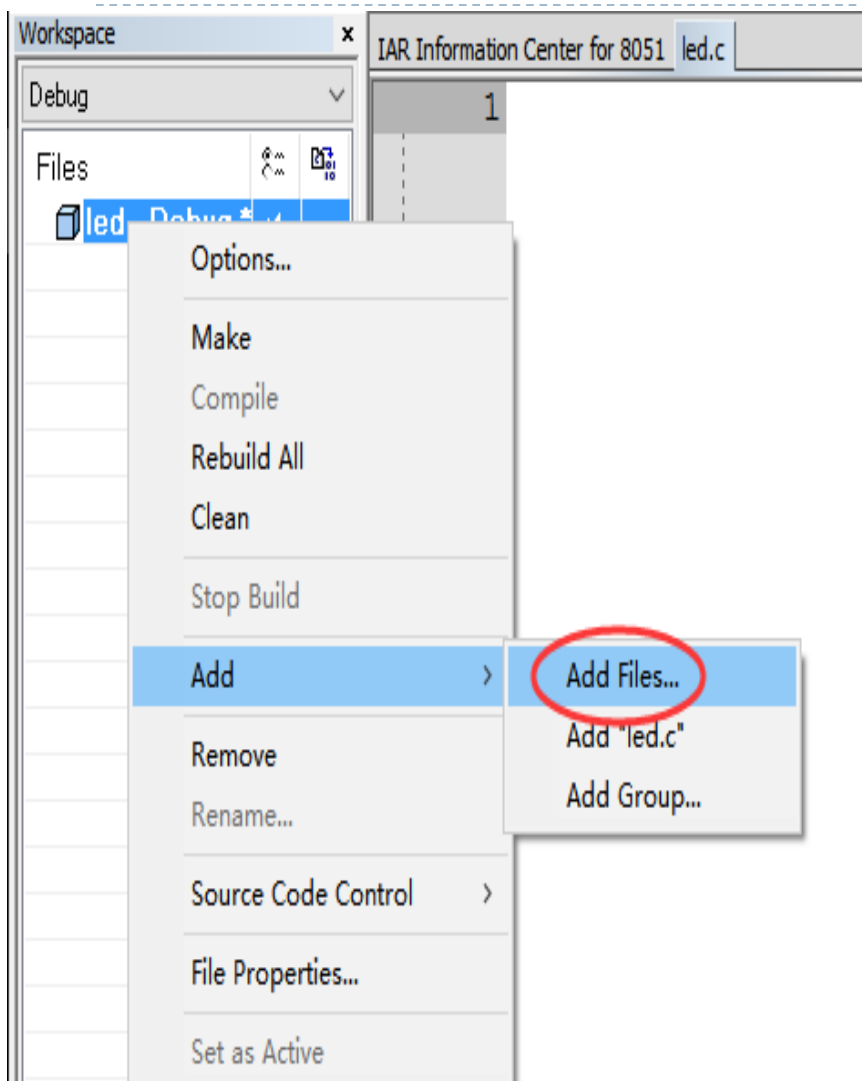


新建一个LED工程



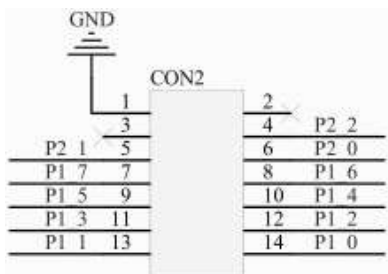
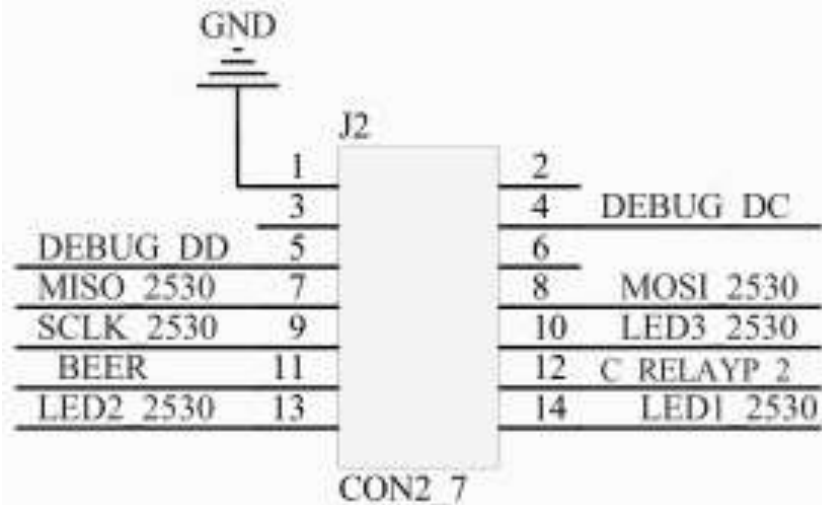
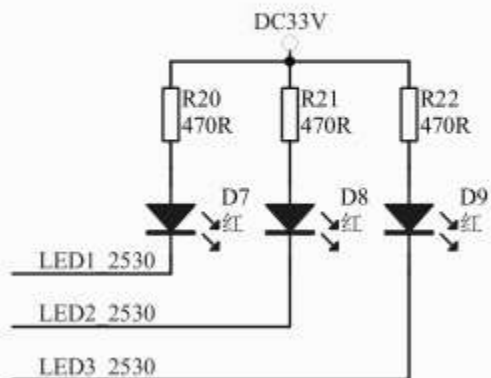
添加文件

在工程中添加我们
所需的.c/.h文件



实现LED点灯

LED灯



根据原理图我们可以知道LED1、LED2、LED3分别连接的I/O为P1_0、P1_1、P1_4.

实现代码

以下为LED的IO初始化函数

```
void led_init(void)
{
    P1SEL &= ~(1 << 0); //功能设置寄存器,
    设置为普通 I/O 口
    P1DIR |= (1 << 0); //设置为输出模式
    LED1 = LED_OFF; //默认关灯
    P1SEL &= ~(1 << 1);
    P1DIR |= (1 << 1);
    LED2 = LED_OFF;
    P1SEL &= ~(1 << 4);
    P1DIR |= (1 << 4);
    LED3 = LED_OFF;
}
```

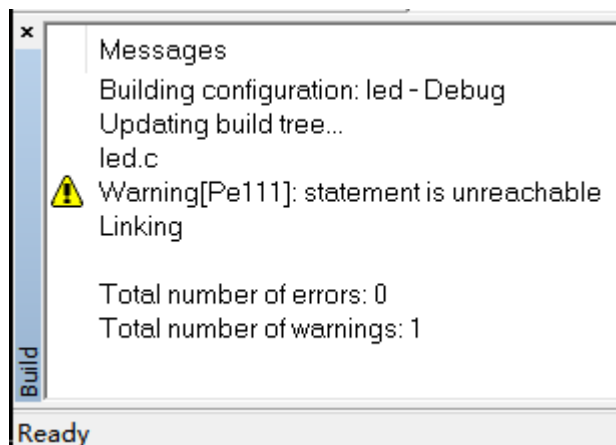
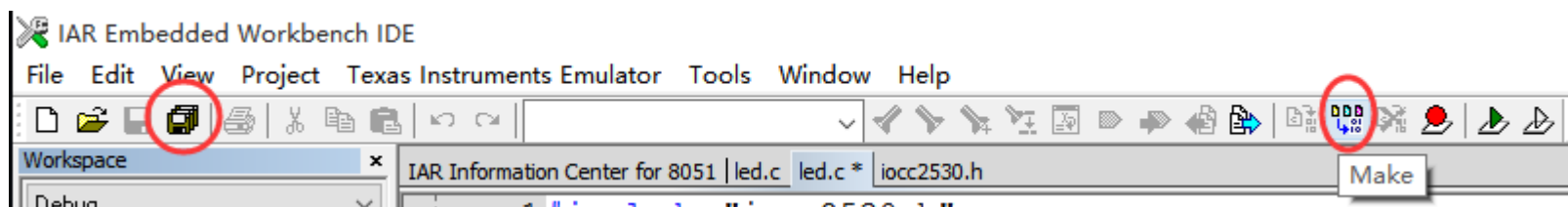
主函数实现LED流动的效果

```
int main(void)
{
    led_init();
    while (1) {
        LED1 = LED_ON;
        delay(10);
        LED1 = LED_OFF;
        delay(10);
        LED2 = LED_ON;
        delay(10);
        LED2 = LED_OFF;
        delay(10);
        LED3 = LED_ON;
        delay(10);
        LED3 = LED_OFF;
        delay(10);
    }
    return 0;
}
```



编译工程

- 当我们进行裸机开发的时候，需要对工程做出相关的配置，但是在使用协议栈开发时无需进行配置，直接使用即可

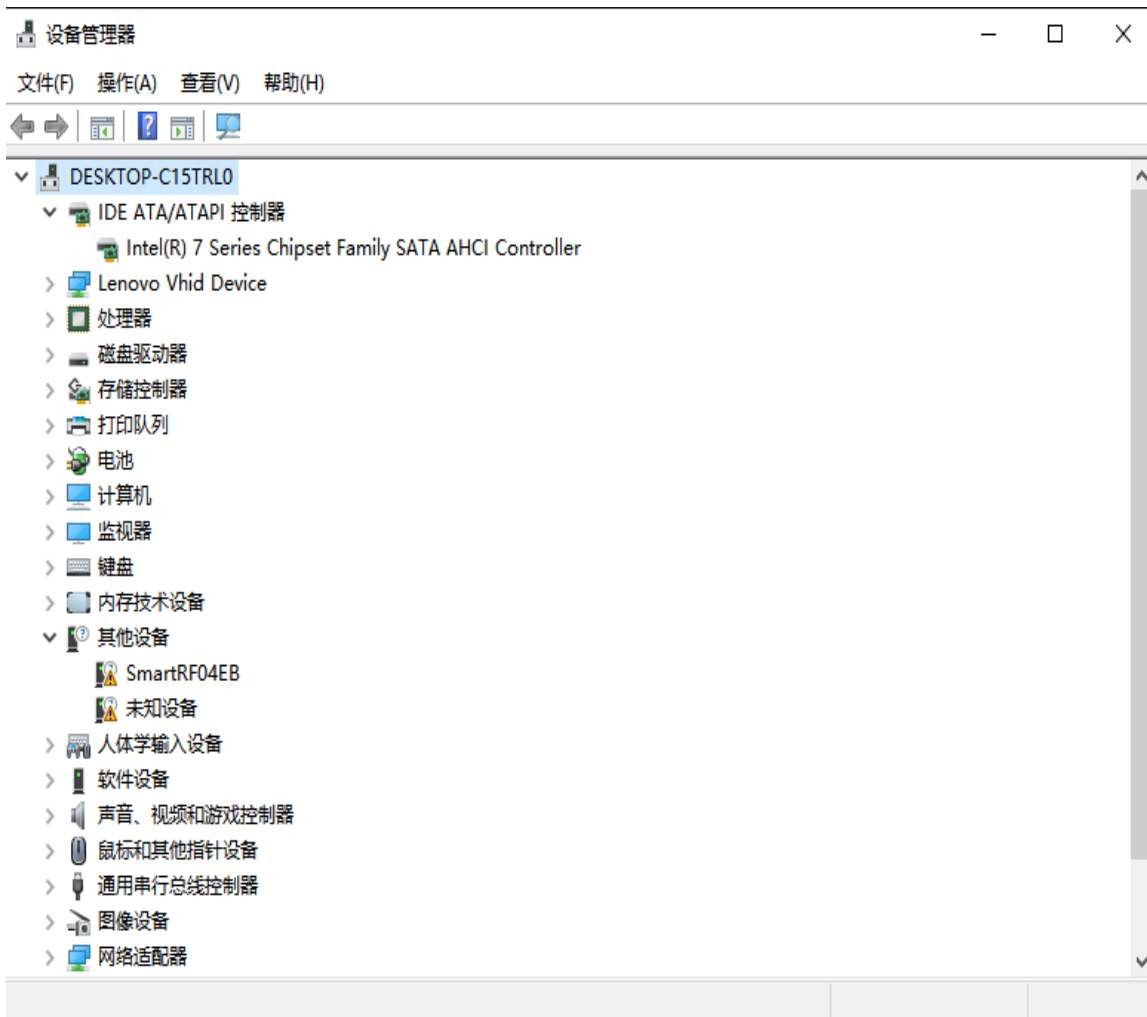


我们对工程进行保存并编译，随后下载至开发板。

仿真器调试与下载

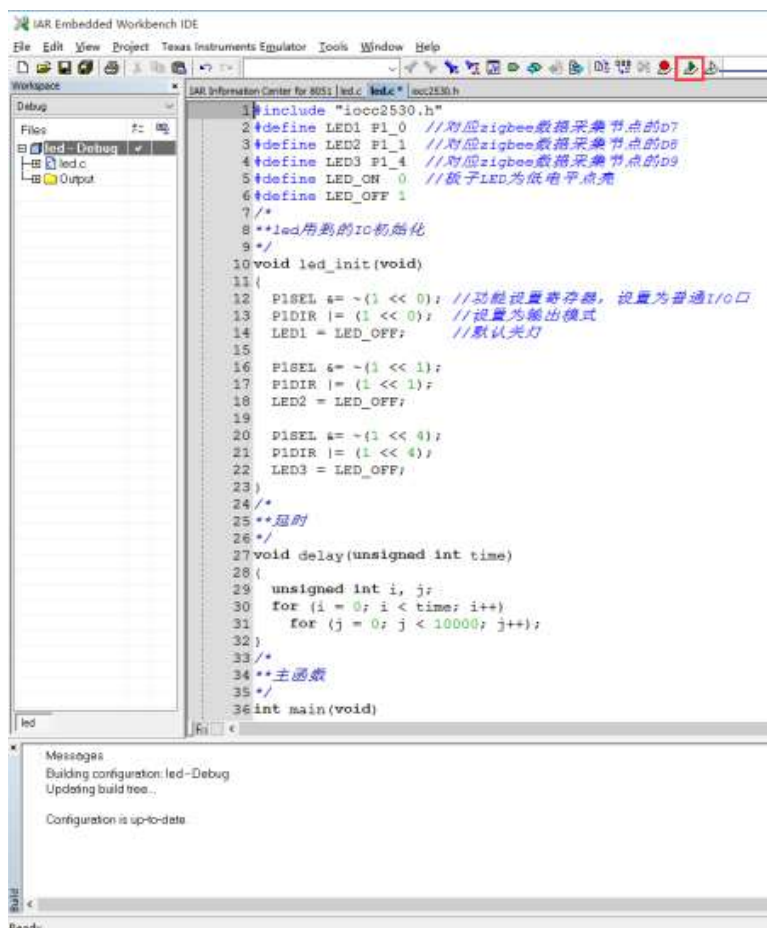
SmartRF04EB 仿真器的一端为 USB 连接头，通过 USB 线可以连接下载器和电脑（PC），另一端为十芯牛角下载接头（配备防呆缺口），通过十芯下载线可以连接各个节点。

仿真器、PC 和 ZigBee 已连接，接下来我们开始安装下载器的驱动程序，首先保证硬件连接正常。我们打开设备管理器，我们看到其他设备中的 SmartRF04EB 左边有一个黄色的叹号，代表驱动没有安装，我们手动安装它的驱动即可。

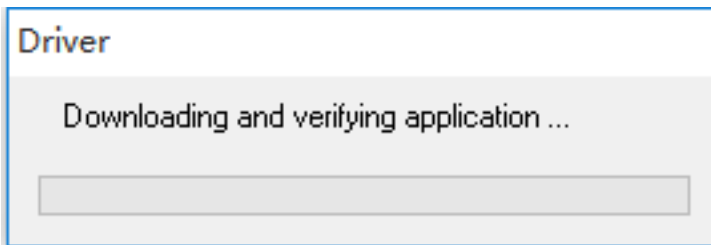


下载至开发板

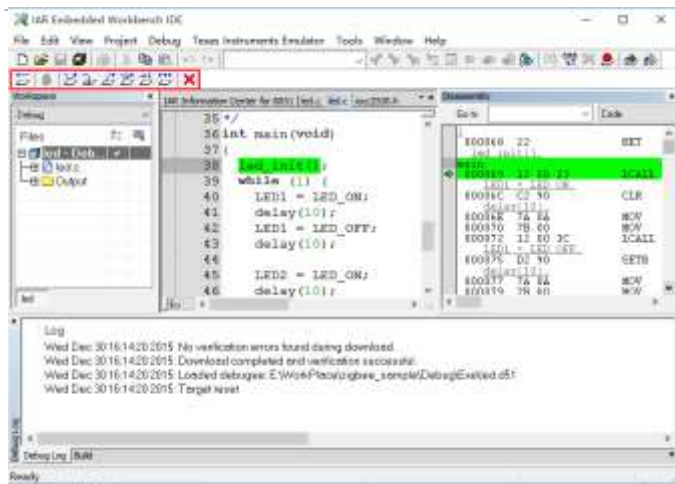
确保 SmartRF04EB 仿真器驱动已经安装成功，接下来我们将刚刚建立的 LED 工程下载至我们的开发板。



点击该按钮会对工程进行重新编译并下载至开发板，最后进入到调试阶段



调试功能



当程序出问题时，调试往往可以事半功倍，希望大家可以多使用 **DEBUG** 来调试程序，在这里我们不进行调试，退出调试（点击红色叉号）并复位节点模块即完成了程序的下载。

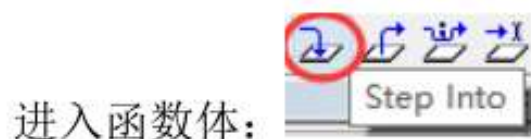
我们来介绍每个按钮的作用。



复位按钮：



单步调试：



进入函数体：



跳出函数体：



全速运行：



每次执行一个语句：



运行至光标所在处：

ZigBee协议栈架构

协议栈架构

使用 ZigBee 协议栈进行应用程序开发过程中，虽然说读者可以不必关心 ZigBee 协议栈的具体细节，但是读者需要对 ZigBee 协议栈的基本构成与内部工作原理有个清晰的认识，只有这样才能将 ZigBee 协议栈提供的函数充分地融入自己的实际项目开发过程中。ZigBee 协议栈包含了 ZigBee 协议所规定的基本功能，这些功能是以函数的形式实现的，为了便于管理这些函数集，从 ZigBee2006 协议栈开始，ZigBee 协议栈内加入了实时操作系统，称为 OSAL（操作系统抽象层，Operating System Abstraction Layer）。

- ◆ OSAL 运行机制
 - ◆ 事件和消息
 - ◆ 添加用户任务
-

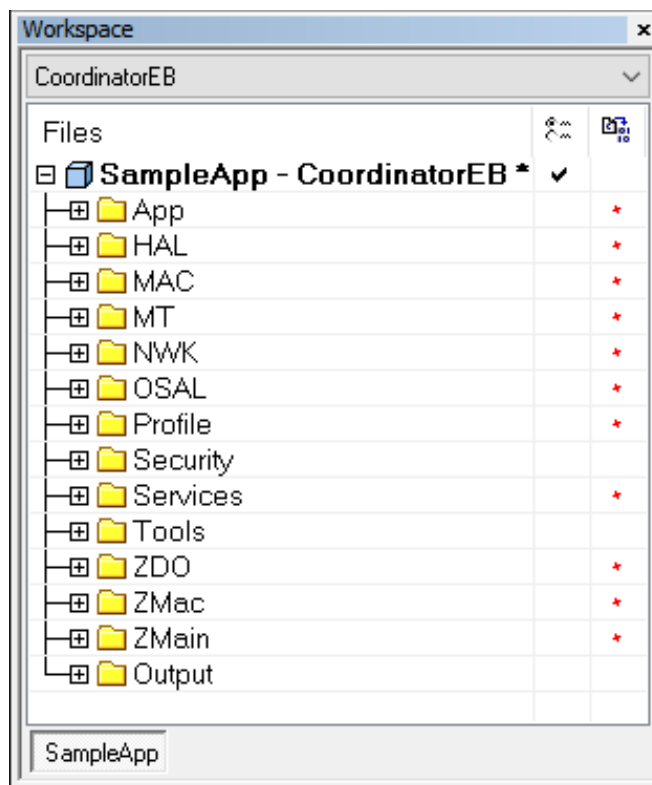


TI Z-Stack协议栈架构分析

- ▶ 在软件层次上，ZigBee无线网络需要依靠ZigBee协议栈才能实现。与传统TCP/IP协议类似，Z-Stack协议栈采用分层结构。分层的目的是为了使协议栈各层能够独立，每一层向上层提供一些服务，应用开发程序员只需关心与他们相关层的协议。分层的结构脉络清晰，方便设计和调试。Z-Stack协议栈需要配合操作系统抽象层（OSAL）才能够运行。在操作系统运行前，系统需要完成硬件平台和软件架构所需要的各个模块的初始化工作。操作系统抽象层OSAL通过时间片轮转函数实现任务调度，并提供多任务处理机制。IEEE802.15.4标准和ZigBee协议规范在协议栈层间定义了大量的原语操作。

TI Z-Stack项目文件组织

- ▶ IAR工具为TI官方指定的开发环境。在IAR中，打开一个Z-Stack工程文件，读者可以看到Z-Stack项目组织。一个Z-Stack工程文件大约有10万行代码。Z-Stack目录结构如下图所示，在Z-Stack项目中大约有14个目录文件，目录文件下面又有很多的子目录和文件，它定义了许多变量、结构体，以及函数声明，通过这些声明，读者可以大体了解此目录的作用。



● APP(应用层)目录

这个目录包含了应用层的内容。读者需要在这个文件夹下进行应用程序编写，并创建各种不同工程的区域。

● HAL(硬件层)目录

这个目录提供各种硬件模块的驱动及操作函数。例如：定时器Timer，通用I/O口GPIO，通用异步收发传输器UART，模数转换ADC

● MAC(介质访问控制层)目录

此目录包含了MAC层参数配置文件和库的函数接口文件。High Level和Low Level两个目录表示MAC层的高层和底层，在Include目录下则有MAC层的参数配置文件及基于MAC的LIB库函数接口文件，这里的MAC层协议是不开源的，而以库的形式给出。

● MT(监制调试层)目录

该目录下的文件主要用于调试，利用Ztool工具，并经过串口实现对各层的调试，同时与各层进行直接交互。

● NMK(网络层)目录

此目录含有网络层配置参数文件、网络层库函数接口文件，以及APS层库函数接口。



- **OSAL(协议栈的操作系统抽象层)目录**

Z-Stack协议栈需要配合OSAL操作系统才能够运行。OSAL层主要管理系统软硬件资源，具体包括：电源管理、内存管理、任务管理、中断管理、信息管理、时间管理、任务同步和非易失存储(NV)管理。

- **Profile(AF应用框架层)目录**

此目录包括AF层处理函数接口文件。

- **Security(安全层)目录**

此目录包含安全层处理函数接口文件，如加密函数。

- **Tools(工作配置)目录**

此目录包括空间划分及Z-Stack相关配置信息。大部分Z-Stack配置信息在f8wConfig.cfg文件中。

- **ZDO(ZigBee设备对象)目录**

此目录可认为是一个公共功能集合，用户可用自定义对象调用APS子层和NWK层的服务。



● ZMAC(MAC导出层)目录

此目录提供802.15.4MAC和网络层之间的接口。其中Zmac.c是Z-Stack MAC导出层接口文件，并包括参数配置等。Zmac_cb.c是ZMAC回调网络层函数。

● Zmain(主函数)目录

Zmain.c主要包括了整个项目的入口函数main(),而OnBoard.c文件包含硬件开发平台上各类外设进行控制的接口函数。

● Output(输出文件)目录

此目录由EW8051 IDE自动生成。

操作系统抽象层OSAL实现了一个易用的操作系统平台,并通过时间片轮转函数实现任务调度,提供多任务处理机制.用户可以调用OSAL提供的相关API进行多任务编程,以将自己的应用程序作为一个独立的任务来实现.



OSAL 运行机制

ZigBee 协议栈与 ZigBee 协议之间并不能完全画等号。

ZigBee 协议栈仅仅是 ZigBee 协议的具体实现。**OSAL 就是一种支持多任务运行的系统资源分配机制。在 ZigBee 协议栈中，OSAL 负责调度各个任务的运行，如果有事件发生了，则会调用相应的事件处理函数进行处理。**

那么，**事件和任务的事件处理函数**是如何联系起来的呢？ZigBee 中采用的方法是：建立一个事件表，保存各个任务的对应的事件，建立另一个函数表，保存各个任务的事件处理函数的地址，然后将这两张表建立某种对应关系，当某一事件发生时则查找函数表找到对应的事件处理函数即可。



在 ZigBee 协议栈中，有三个变量至关重要。

- **tasksCnt**—该变量保存了任务的总个数。

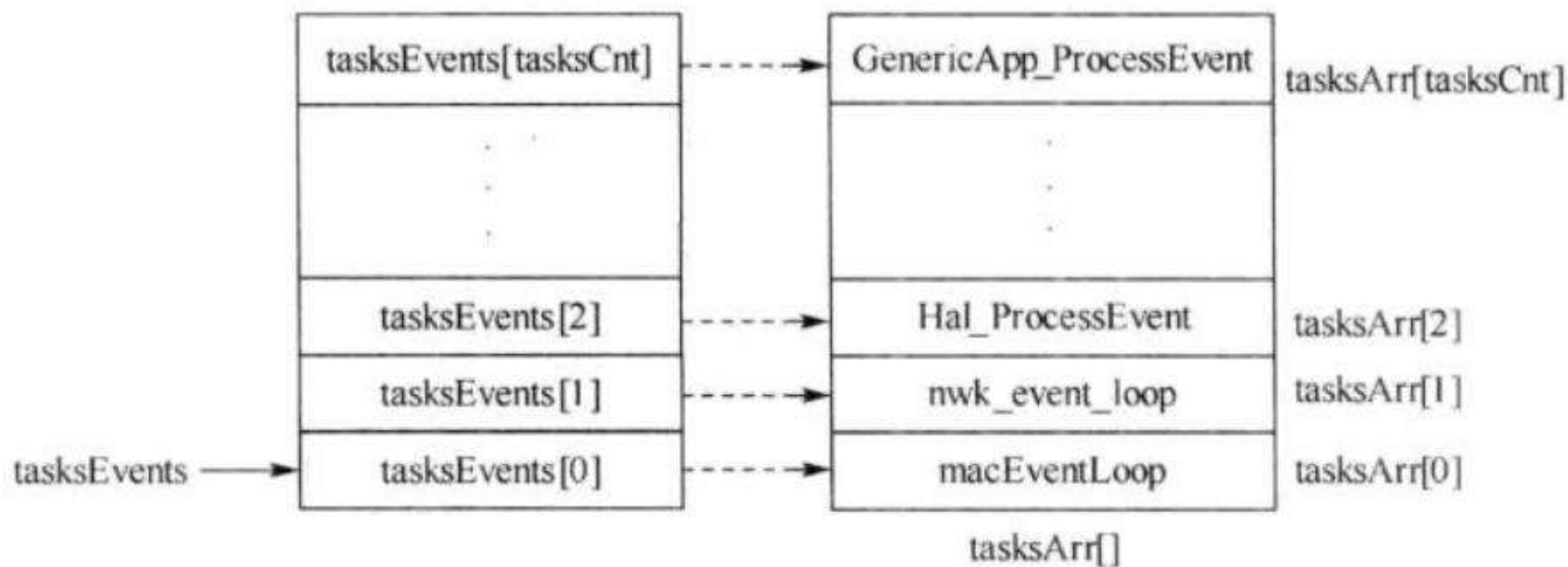
该变量的声明为：uint8 tasksCnt，其中 uint8 的定义为：typedef unsigned char uint8

- **tasksEvents**—这是一个指针，指向了事件表的首地址。

该变量的声明为：uint16 *tasksEvents，其中 uint16 的定义为：typedef unsigned short uint16

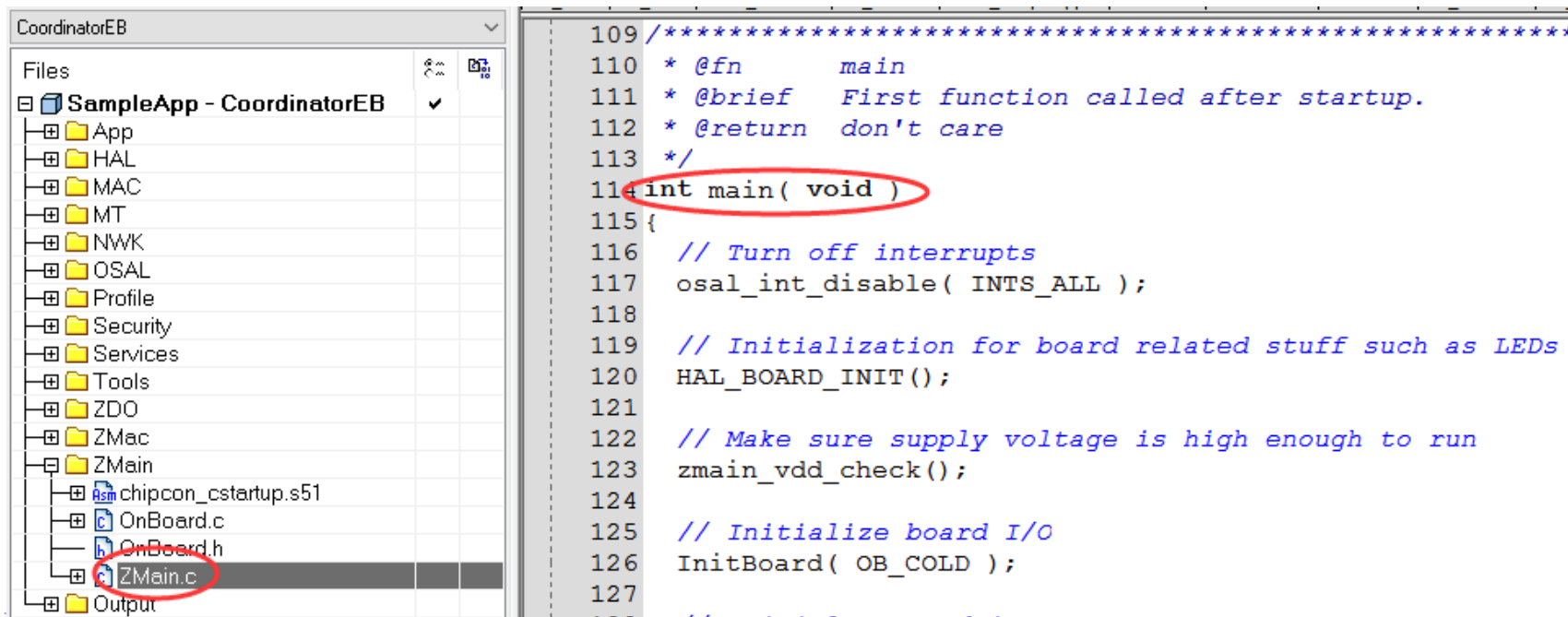
- **tasksArr**—这是一个数组，数组的每一项都是一个函数指针，指向了事件处理函数
该数组的声明为：const pTaskEventHandlerFn tasksArr[]，其中 pTaskEventHandlerFn 的定义为：typedef unsigned short (*pTaskEventHandlerFn)(unsigned char task_id, unsigned short event)，这是定义了一个函数指针。tasksArr 数组的每一项都是一个函数指针，指向了事件处理函数。

▶ 事件表和函数表的关系如下图



程序的入口：main（）函数

总结下 OSAL 的工作原理：通过 tasksEvents 指针访问事件表的每一项，如果有事件发生，则查找函数表找到事件处理函数进行处理，处理完后，继续访问事件表，查看是否有事件发生，无限循环。**OSAL 就是一种基于事件驱动的红询式操作系统。**事件驱动是指发生事件后采取相应的事件处理方法，红询指的是不断地查看是否有事件发生。整个协议栈是从哪里开始执行的呢？在 Zmain 文件夹下有个 Zmain.c 文件，打开该文件可以找到 main()函数，这就是整个协议栈的入口点。



osal_start_system () 函数

在 main()函数中调用了很多函数，在此可以暂不考虑，重点是 osal_start_system()函数，在此之前的函数都是对板载硬件以及协议栈进行的初始化，直到调osal_start_system()函数，整个 ZigBee 协议栈才算是真正地运行起来了。

```
do {  
    // Task is highest priority that is ready.  
    if (tasksEvents[idx])  
    {  
        break;  
    }  
} while (++idx < tasksCnt);  
if (idx < tasksCnt)  
{  
    uint16 events;  
    halIntState_t intState;  
    HAL_ENTER_CRITICAL_SECTION(intState)  
    ;  
    events = tasksEvents[idx];
```

以下代码实现基于事件驱动的轮询操作系统

```
// Clear the Events for this task.  
tasksEvents[idx] = 0;  
HAL_EXIT_CRITICAL_SECTION(intState);  
events = (tasksArr[idx])( idx, events );  
HAL_ENTER_CRITICAL_SECTION(intState);  
// Add back unprocessed events to the  
current task.  
tasksEvents[idx] |= events;  
HAL_EXIT_CRITICAL_SECTION(intState);  
}
```

事件如何表示？

ZigBee 协议栈使用一个 unsigned short 型的变量，因为 unsigned short 类型占两个字节，即 16 个二进制位，因此，可以使用每个二进制位表示一个事件，我们来看下协议栈定义的系统事件SYS_EVENT_MSG，十六进制：0x8000，二进制0b10000000000000000。它用的就是最高位来表示该事件：在系统初始化时，所有任务的事件初始化为 0，

```
150 /*****
151  * Global System Events
152  */
153
154 #define SYS_EVENT_MSG                0x8000    // A message is waiting event
```

在事件处理函数中返回未处理的事件？

```
138 // return unprocessed events
139 return (events ^ SYS_EVENT_MSG);
```

在事件处理函数结束的时候我都有一个 return 语句，还使用了异或运算，我们通过下面的例子来理解异或运算的作用。

我们假设同时发生了 SYS_EVENT_MSG 事件和 SAMPLEAPP_SEND_PERIODIC_MSG_EVT 事件：

```
#define SAMPLEAPP_SEND_PERIODIC_MSG_EVT    0x0001
#define SYS_EVENT_MSG                      0x8000 // A message is waiting event
```

则此时 events = 0b10000000000000001，即 0x8001，假设现在处理完了系统事件 SYS_EVENT_MSG，则应该将最高位清 0，这时我们只需要使用异或运算即可，即 events^SYS_EVENT_MSG=0x8001^0x8000=0x01，即 0b00000000000000001，刚好将我们未处理完的事件 SAMPLEAPP_SEND_PERIODIC_MSG_EVT 留下。



消息与事件的区别

提到事件，我们就不得不提到消息。事件是驱动任务去执行某些操作的条件，当系统中产生了一个事件，OSAL 将这个事件传递给相应的任务后，任务才能执行一个相应的操作（调用事件处理函数去处理）。通常某些事件发生时，又伴随着一些附加信息的产生，例如：从天线接收到数据后，会产生 `AF_INCOMING_MSG_CMD` 消息，但是任务的事件处理函数在处理这个事件的时候，还需要得到收到的数据。因此，这就需要**将事件和数据封装成一个消息**，将消息发送到消息队列，然后在事件处理函数中就可以使用 `osal_msg_receive` 从消息队列中得到该消息。

添加用户任务

在使用 ZigBee 协议栈进行应用程序开发时，如何在应用程序中添加一个新任务呢？

要添加新任务，只需要编写两个函数：

- ◆ 新任务的初始化函数。
- ◆ 新任务的事件处理函数。

将新任务的初始化函数添加在 osalInitTasks()函数的最后，如下代码所示：

```

106 void osalInitTasks( void )
107 {
108     uint8 taskID = 0;
109
110     tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
111     osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));
112
113     macTaskInit( taskID++ );
114     nwk_init( taskID++ );
115     Hal_Init( taskID++ );
116     #if defined( MT_TASK )
117     MT_TaskInit( taskID++ );
118     #endif
119     APS_Init( taskID++ );
120     #if defined ( ZIGBEE_FRAGMENTATION )
121     APSF_Init( taskID++ );
122     #endif
123     ZDApp_Init( taskID++ );
124     #if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
125     ZDNwkMgr_Init( taskID++ );
126     #endif
127     SampleApp_Init( taskID );
128 }

```

将事件处理函数的地址加入 tasksArr[]数组，如下代码所示

```
71 // The order in this table must be identical to the task initialization calls below in osalInitTask.
72 const pTaskEventHandlerFn tasksArr[] = {
73     macEventLoop,
74     nwk_event_loop,
75     Hal_ProcessEvent,
76 #if defined( MT_TASK )
77     MT_ProcessEvent,
78 #endif
79     APS_event_loop,
80 #if defined( ZIGBEE_FRAGMENTATION )
81     APSF_ProcessEvent,
82 #endif
83     ZDApp_event_loop,
84 #if defined( ZIGBEE_FREQ_AGILITY ) || defined( ZIGBEE_PANID_CONFLICT )
85     ZDNwkMgr_event_loop,
86 #endif
87     SampleApp_ProcessEvent
88 };
```

注意：

tasksArr[]数组里各事件处理函数的排列顺序要与 osalInitTasks 函数中调用各任务初始化函数的顺序保持一致，只有这样才能保证当任务有事件发生时调用每个任务对应的事件处理函数。

ZigBee—LED实验

在OSAL中实现LED闪烁

我们这次的实验是基于TI的官方例程修改。由于协议栈中已经提供了 led 的驱动，我们可以直接使用协议栈中提供的 LED 灯控制函数。但是 TI 的 ZigBee 协议栈 Z-stack 是针对 TI 官方套件的，我们需要修改底层驱动才能适配于我们的开发板。

- ◆ 修改LED底层驱动
- ◆ 熟悉有关LED的API
- ◆ 添加用户任务
- ◆ 熟悉事件处理函数的流程结构
- ◆ 编写用户功能代码



修改驱动

修改LED灯的数量以及LED用到的GPIO口

```

98
99 /* -----
100 *                               LED Configuration
101 * -----
102 */
103
104 #if defined (HAL_BOARD_CC2530EB_REV17) && !defined (HAL_PA_LNA) && !defined (HAL_PA_LNA_CC2590)
105     #define HAL_NUM_LEDS        3
106 #elif defined (HAL_BOARD_CC2530EB_REV13) || defined (HAL_PA_LNA) || defined (HAL_PA_LNA_CC2590)
107     #define HAL_NUM_LEDS        1
108 #else
109     #error Unknown Board Identifier
110 #endif
111
112 #define HAL_LED_BLINK_DELAY()    at( ( volatile uint32 i; for (i=0; i<0x5900; i++) { } ) )
113
114 /* D7 */
115 #define LED1_BV                BV(0)
116 #define LED1_SBIT               P1_0
117 #define LED1_DDR                P1DIR
118 #define LED1_POLARITY          ACTIVE_HIGH
119
120 #if defined (HAL_BOARD_CC2530EB_REV17)
121     /* D8 */
122     #define LED2_BV                BV(1)
123     #define LED2_SBIT               P1_1
124     #define LED2_DDR                P1DIR
125     #define LED2_POLARITY          ACTIVE_HIGH
126
127     /* D9 */
128     #define LED3_BV                BV(4)
129     #define LED3_SBIT               P1_4
130     #define LED3_DDR                P1DIR
131     #define LED3_POLARITY          ACTIVE_HIGH
132 #endif

```

头文件 `hal_board_cfg.h` 包含了硬件信息。我们开发板上有 3 个由 IO 控制的 LED 灯，分别是 D7、D8、D9 三盏灯，分别对应的 IO 是 P1_0、P1_1、P1_4。由于我们用到的 LED 和 TI 官方套件上一致，我们暂且不必修改；

由于 TI 的开发板是高电平点亮 LED，而我们的开发板是低电平点亮 LED，找到文件中下图所示的代码并按红线标识处修改代码。

```
OnBoard.c | hal_board_cfg.h | SampleApp.h | SampleApp.c | comdef.h | ZComDef.h | OSAL.s51 | SampleAppHw.c | OSAL_SampleApp.c | MT_UART.c | hal_uart.c | _hal_uart_dma.c | ioCC2530.h | hal_led.h | hal_led.c |
286
287 /* ----- LED's ----- */
288 #if defined (HAL_BOARD_CC2530EB_REV17) && !defined (HAL_PA_LNA) && !defined (HAL_PA_LNA_CC2590)
289
290 #define HAL_TURN_OFF_LED1()      st( LED1_SBIT = LED1_POLARITY (1); )
291 #define HAL_TURN_OFF_LED2()      st( LED2_SBIT = LED2_POLARITY (1); )
292 #define HAL_TURN_OFF_LED3()      st( LED3_SBIT = LED3_POLARITY (1); )
293 #define HAL_TURN_OFF_LED4()      HAL_TURN_OFF_LED1()
294
295 #define HAL_TURN_ON_LED1()        st( LED1_SBIT = LED1_POLARITY (0); )
296 #define HAL_TURN_ON_LED2()        st( LED2_SBIT = LED2_POLARITY (0); )
297 #define HAL_TURN_ON_LED3()        st( LED3_SBIT = LED3_POLARITY (0); )
298 #define HAL_TURN_ON_LED4()        HAL_TURN_ON_LED1()
299
300 #define HAL_TOGGLE_LED1()         st( if (LED1_SBIT) { LED1_SBIT = 0; } else { LED1_SBIT = 1; } )
301 #define HAL_TOGGLE_LED2()         st( if (LED2_SBIT) { LED2_SBIT = 0; } else { LED2_SBIT = 1; } )
302 #define HAL_TOGGLE_LED3()         st( if (LED3_SBIT) { LED3_SBIT = 0; } else { LED3_SBIT = 1; } )
303 #define HAL_TOGGLE_LED4()         HAL_TOGGLE_LED1()
304
305 #define HAL_STATE_LED1()          (LED1_POLARITY (LED1_SBIT))
306 #define HAL_STATE_LED2()          (LED2_POLARITY (LED2_SBIT))
307 #define HAL_STATE_LED3()          (LED3_POLARITY (LED3_SBIT))
308 #define HAL_STATE_LED4()          HAL_STATE_LED1()
```

LED相关的API

```
95 /*
96  * Set the LED ON/OFF/TOGGLE.
97  */
98 extern uint8 HalLedSet( uint8 led, uint8 mode );
99
100 /*
101  * Blink the LED.
102  */
103 extern void HalLedBlink( uint8 leds, uint8 cnt, uint8 duty, uint16 time );
104
105 /*
106  * Put LEDs in sleep state - store current values
107  */
108 extern void HalLedEnterSleep( void );
109
110 /*
111  * Retore LEDs from sleep state
112  */
113 extern void HalLedExitSleep( void );
114
115 /*
116  * Return LED state
117  */
118 extern uint8 HalLedGetState ( void );
```



添加用户任务

● 任务初始化函数

```
79 void SampleApp_Init( uint8 task_id )
80 {
81     //该任务的ID由OSAL分布
82     SampleApp_TaskID = task_id;
83     //开启一个定时器任务,1s后本任务会触发SAMPLEAPP_LED_BLINK_EVT事件
84     osal_start_timerEx( SampleApp_TaskID, SAMPLEAPP_LED_BLINK_EVT, 1000 );
85 }
```

● 任务的事件处理函数

```
100 uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
101 {
102     //if表达式成立代表SAMPLEAPP_LED_BLINK_EVT事件发生
103     if ( events & SAMPLEAPP_LED_BLINK_EVT )
104     {
105         HalLedSet( HAL_LED_1, HAL_LED_MODE_TOGGLE );
106         HalLedSet( HAL_LED_2, HAL_LED_MODE_TOGGLE );
107         HalLedSet( HAL_LED_3, HAL_LED_MODE_TOGGLE );
108         //重复触发SAMPLEAPP_LED_BLINK_EVT事件
109         osal_start_timerEx( SampleApp_TaskID, SAMPLEAPP_LED_BLINK_EVT, 1000 );
110         return (events ^ SAMPLEAPP_LED_BLINK_EVT);
111     }
112
113     // Discard unknown events
114     return 0;
115 }
```

定时器实现

当我们需要实现周期性的操作时，我们往往会想到利用定时器来实现，协议栈中提供了一些定时器操作，我们可以根据OSAL_Timers.h中提供的函数接口来选择我们需要的函数：

```

SampleApp | OSAL_Timers.h | SampleApp.h | hal_led.c | hal_led.h | OSAL.c | OSAL.h | ZMain.c | OSAL_Timers.c
72
73 /*
74  * Initialization for the OSAL Timer System.
75  */
76 extern void osalTimerInit( void );
77
78 /*
79  * Set a Timer
80  */
81 extern uint8 osal_start_timerEx( uint8 task_id, uint16 event_id, uint16 timeout_value );
82
83 /*
84  * Set a timer that reloads itself.
85  */
86 extern uint8 osal_start_reload_timer( uint8 taskID, uint16 event_id, uint16 timeout_value );
87
88 /*
89  * Stop a Timer
90  */
91 extern uint8 osal_stop_timerEx( uint8 task_id, uint16 event_id );
92
93 /*
94  * Get the tick count of a Timer.
95  */
96 extern uint16 osal_get_timeoutEx( uint8 task_id, uint16 event_id );
97
98 /*
99  * Simulated Timer Interrupt Service Routine
100  */
101
102 extern void osal_timer_ISR( void );
103
104 /*
105  * Adjust timer tables
106  */
107 extern void osal_adjust_timers( void );
108

```

下面介绍一个常用的定时器事件产生函数：

uint8 osal_start_timerEx(uint8 task_id, uint16 event_id, uint16 timeout_value);

- **uint8 task_id**---该参数表明定时时间到达后，哪个任务对齐做出响应。
- **uint16 event_id**---该参数是一个事件ID，定时时间到达后，该事件发生，因此需要添加一个新的事件，该事件发生则表明定时时间到，因此可以在该事件的事件处理函数中实现数据发送；
- **uint16 timeout_value**---定时时间由**timeout_value**（以毫秒为单位）参数确定。

ZigBee—串口实验

在OSAL中实现串口功能

串口是开发板和用户电脑交互的一种工具，正确地使用串口对于 ZigBee 无线网络的学习具有较大的促进作用，使用串口的基本步骤：

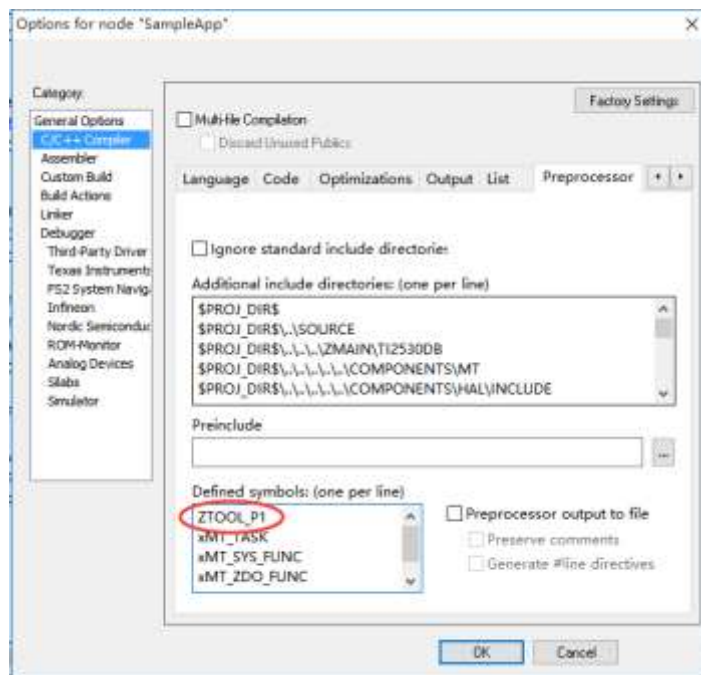
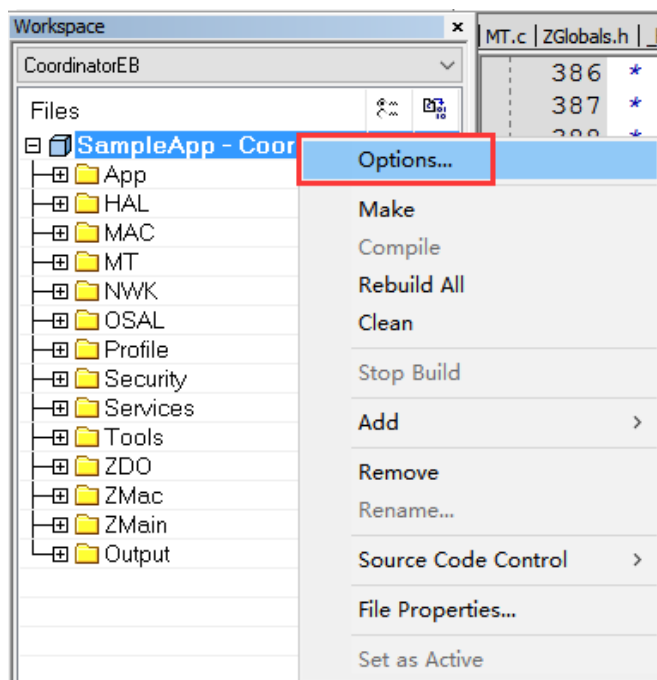
- (一) 初始化串口，包括设置波特率、中断等；
- (二) 向发送缓冲区发送数据或者从接收缓冲区读取数据。

上述方法是使用串口的常用方法，但是由于协议栈的存在，使得串口的使用略有不同，在 ZigBee 协议栈中已经对串口初始化所需要的函数进行了实现，用户只需要传递几个参数就可以使用串口，此外，ZigBee 协议栈还实现了串口的读取和写入函数。需要注意的是，在 ZigBee 协议栈中，TI 采用的方法是将串口和 DMA 结合起来使用，这样可以降低 CPU 的负担。

- ◆ 串口发送
- ◆ 串口接收

串口发送

- ▶ 使用串口前我们需要做的初始化工作:
加入一个工程预编译选项 `ZTOOL_P1`, 代表我们使用的串口是 0



```

90 /*****
91  * @fn      MT_UartInit
92  *
93  * @brief   Initialize MT with UART support
94  *
95  * @param   None
96  *
97  * @return  None
98  *****/
99 void MT_UartInit ()
100 {
101     halUARTCfg_t uartConfig;
102
103     /* Initialize APP ID */
104     App_TaskID = 0;
105
106     /* UART Configuration */
107     uartConfig.configured = TRUE;
108     uartConfig.baudRate = MT_UART_DEFAULT_BAUDRATE;
109     uartConfig.flowControl = MT_UART_DEFAULT_OVERFLOW;
110     uartConfig.flowControlThreshold = MT_UART_DEFAULT_THRESHOLD;
111     uartConfig.rx.maxBufSize = MT_UART_DEFAULT_MAX_RX_BUFF;
112     uartConfig.tx.maxBufSize = MT_UART_DEFAULT_MAX_TX_BUFF;
113     uartConfig.idleTimeout = MT_UART_DEFAULT_IDLE_TIMEOUT;
114     uartConfig.intEnable = TRUE;
115     #if defined (ZTOOL_P1) || defined (ZTOOL_P2)
116     uartConfig.callBackFunc = MT_UartProcessZToolData;
117     #elif defined (ZAPP_P1) || defined (ZAPP_P2)
118     uartConfig.callBackFunc = MT_UartProcessZAppData;
119     #else
120     uartConfig.callBackFunc = NULL;
121     #endif

```

对波特率和流控做出修改。

修改波特率、流控
等串口参数

```

70 #if !defined( MT_UART_DEFAULT_OVERFLOW )
71 // #define MT_UART_DEFAULT_OVERFLOW TRUE  关流控
72 #define MT_UART_DEFAULT_OVERFLOW FALSE
73 #endif
74
75 #if !defined MT_UART_DEFAULT_BAUDRATE  波特率:115200
76 // #define MT_UART_DEFAULT_BAUDRATE HAL_UART_BR_38400
77 #define MT_UART_DEFAULT_BAUDRATE HAL_UART_BR_115200
78 #endif

```

任务初始化 SampleApp_Init()函数中添加如下代码：

```
void SampleApp_Init( uint8 task_id )
```

```
{
```

```
SampleApp_TaskID = task_id;
```

```
/*初始化串口，用 MT 层提供的 API*/
```

```
MT_UartInit();
```

```
/*发送一句话，代表串口可用*/
```

```
HalUARTWrite(0, "uart0 is OK!\n", strlen("uart0 is  
OK!\n"));
```

```
}
```

到这里我们串口发送功能的代码已经实现了，我们将程序编译并下载至任意开发板，并将其通过 usb 连接至 PC，打开 PC 端的串口助手并按图中软件的配置进行调整，最后复位开发板，串口软件会出现“uart0 is OK!”：



串口接收

串口接收数据的大致流程：

操作系统会在接收到串口数据后适时地设置串口事件从而去调用串口接收回调函数 `MT_UartProcessZToolData()`，该函数会读取 DMA 中已经获取到的数据，会将串口接收到的数据打包成一个消息，消息 ID 是 `CMD_SERIAL_MSG`，消息发给谁呢？答案是：当任务使用 `MT_UartRegisterTaskID()` 函数注册了自己的 `task_id`，串口消息就会发送给该任务。

总结一下，我们需要做以下三步来实现 OSAL 中的串口接收：

◆ 应用层任务初始化函数中使用 `MT_UartRegisterTaskID()` 注册 `task_id`；

```
void SampleApp_Init( uint8 task_id )
{
    SampleApp_TaskID = task_id;
    /*初始化串口，用 MT 层提供的 API*/
    MT_UartInit();
    /*登记任务号码*/
    MT_UartRegisterTaskID(task_id);
    /*发送一句话，代表串口可用*/
    HalUARTWrite(0, "uart0 is OK!\n", strlen("uart0 is OK!\n"));
}
```

◆ 重写串口接收回调函数 MT_UartProcessZToolData () ；

```
void MT_UartProcessZToolData ( uint8 port, uint8 event )
{
    uint8 i = 0;
    uint8 rx_buf[MT_UART_DEFAULT_MAX_RX_BUFF], rx_len = 0;
    (void)event;
    while (Hal_UART_RxBufLen(port)) {
        HalUARTRead(port, &rx_buf[rx_len], 1);
        rx_len++;
    }
    if (rx_len != 0) {
        pMsg = (mtOSALSerialData_t *)osal_msg_allocate( sizeof
        ( mtOSALSerialData_t ) + rx_len + 1);
        pMsg->hdr.event = CMD_SERIAL_MSG;
        pMsg->msg = (uint8*)(pMsg + 1);
        pMsg->msg[0] = rx_len;
        for (i = 0; i < rx_len; i++)
            pMsg->msg[i + 1] = rx_buf[i];
        osal_msg_send( App_TaskID, (byte *)pMsg );
    }
}
```



事件处理函数中添加对串口事件的处理。

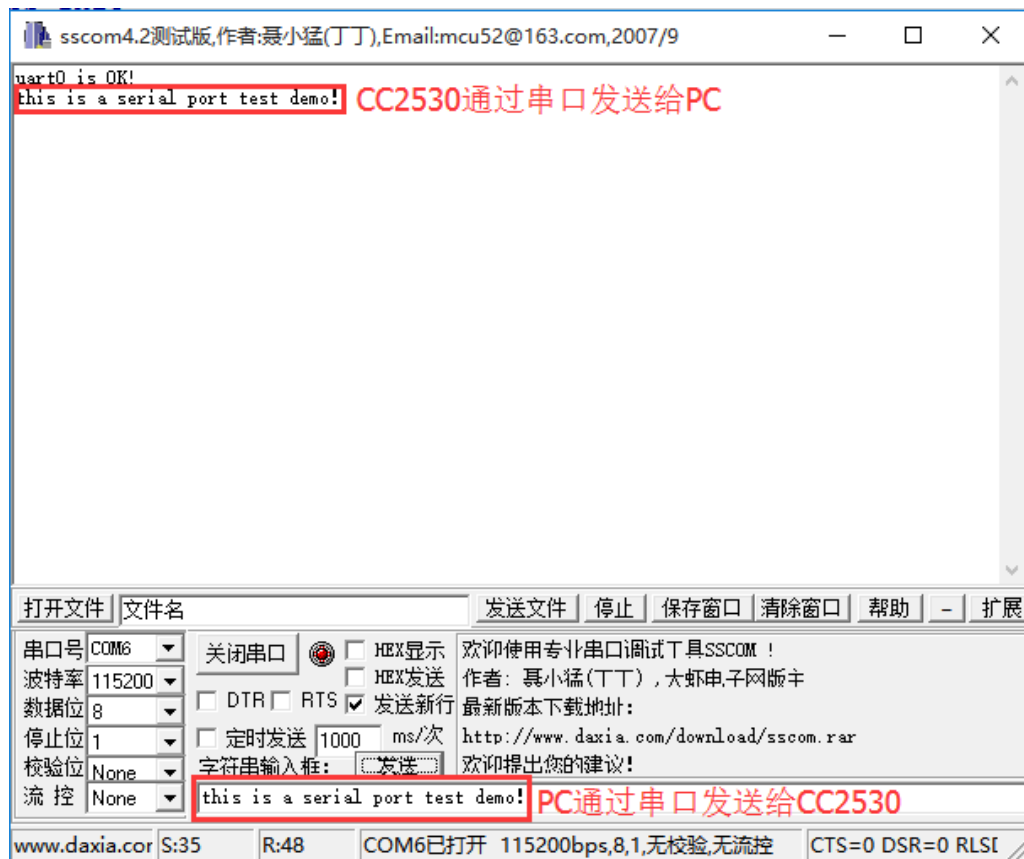
```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
    afIncomingMSGPacket_t *MSGpkt;
    (void)task_id; // Intentionally unreferenced parameter
    if ( events & SYS_EVENT_MSG )
    {
        MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive(
            SampleApp_TaskID );
        while ( MSGpkt )
        {
            switch ( MSGpkt->hdr.event )
            {
            case CMD_SERIAL_MSG:
                SerialData_Analysis(((mtOSALSerialData_t *)MSGpkt)->msg);
                break;
            default:
                ;
            }
        }
    }
}
```

```
// Release the memory
osal_msg_deallocate( (uint8 *)MSGpkt );
// Next - if one is available
MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );
}
// return unprocessed events
return (events ^ SYS_EVENT_MSG);
}
// Discard unknown events
return 0;
}
```

事件处理函数代码格式比较固定，只需熟悉即可。

实验现象

到这里我们就实现了在协议栈中串口接收的功能，我们将程序编译下载至我们的开发板，用 USB 连接至 PC，打开串口助手并正确配置波特率等选项。这时我们在串口中输入什么，就回显什么，下面是实验结果：

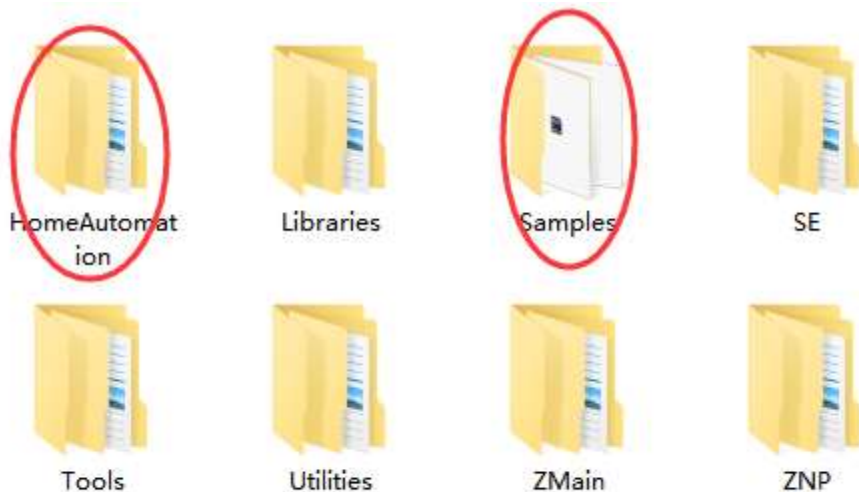


ZigBee—无线透传

高效利用协议栈

相信大家这时对 ZigBee 协议的基本内容都不理解，而且心里充满了疑问。像协议栈到底是用来干什么的，我们为什么要用协议栈，数据到底是怎么在空中传输的……诸如此类问题。

但是TI官方提供了种种例程,我们只需要掌握部分例程中的程序框架及流程就可以快速的进行ZigBee实际项目的开发。这就是使用协议栈进行程序开发的便利之处，所以我们学习协议栈就应该关注我们如何用协议栈提供的接口实现我们的功能，而不是上来就去考虑 ZigBee 协议的具体实现细节。



实现无线透传

透传就是透明传输的简称。那么什么是透明传输呢？顾名思义，透明传输就是指在传输过程中，对外界完全透明，不需要关系传输过程以及传输协议，最终目的是要把传输的内容原封不动的传递给被接收端，发送和接收的内容完全一致。这就相当于把信息直接扔给你想要传输的人，只需要扔（也就是传输）这一个步骤，不需要其他的内容安排。

程序需要实现的功能：

- 1、ZigBee 模块接收到从 PC 机发来的消息，然后无线发送出去
- 2、ZigBee 模块接收到其他 ZigBee 模块发来的消息，然后发送给 PC 机

本次实验我们需要掌握以下重点：

- ◆ 端点描述符
- ◆ 简单描述符
- ◆ 如何发送数据
- ◆ 如何接收数据



我们重点分析四个函数：

- 任务的初始化函数
- 任务的事件处理函数
- 无线消息接收回调函数
- 无线消息发送函数

我们来总结下发送和接收数据的流程：

发送数据：

填充并注册端点描述符 → 配置发送模式及目的地址 → AF_DataRequest() 发送数据

接收数据：

填充并注册端点描述符 → 处理系统事件 SYS_EVENT_MSG 中的 AF_INCOMING_MSG_CMD 消息 → 获得消息包中的无线数据

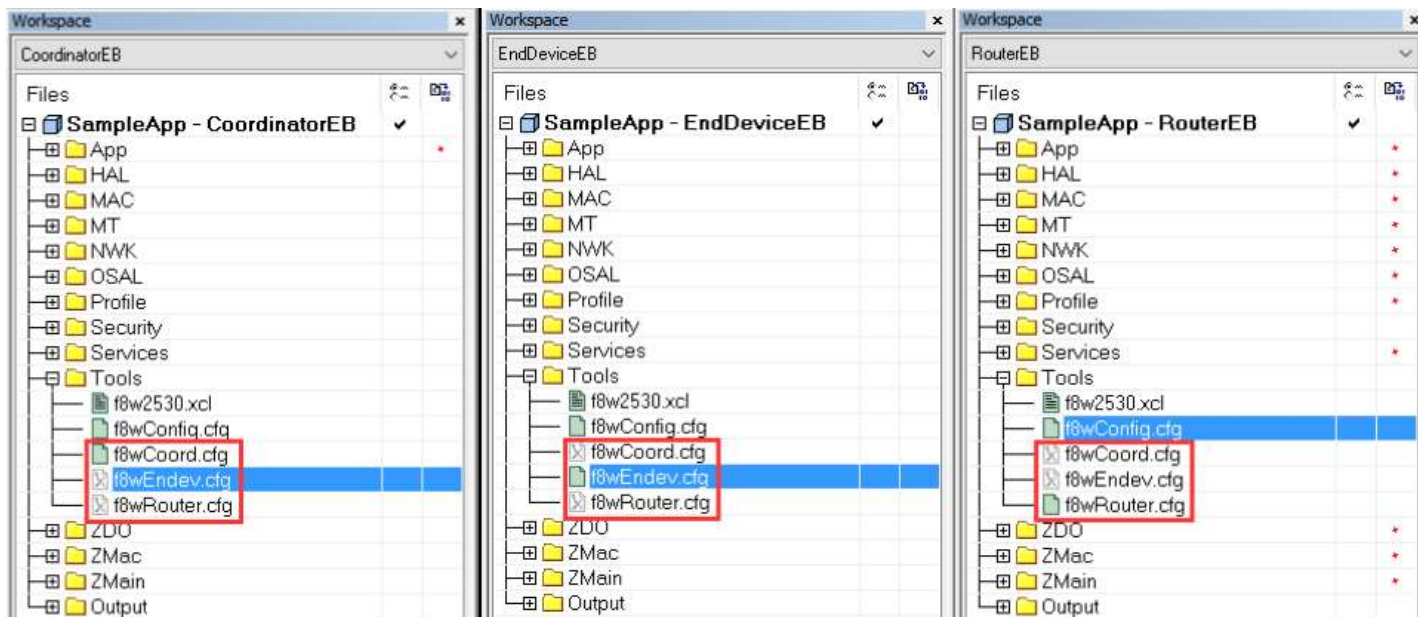
协调器还是终端

在使用IAR编译器进行项目开发的时候，我们经常会建立不同的工程模板，不同的工程模板共用了相同的代码，却可以实现不同的功能。这是如何实现的呢？

利用预编译选项、宏、是否编译特定的文件等方法。

我们知道协调器（coordinator）和终端节点（enddevice）是通过不同的工程区分的，但我们发现两个工程包含的文件又大致相同。那协议栈中如何判断设备的类型并进行相应的操作呢？

这里就是利用不同的工程模板包含特定的文件实现，



f8wCoord.cfg、f8wRouter.cfg文件中分别有如下配置：

Coordinator Settings: -DZDO_COORDINATOR -DRTR_NWK

```
ZDApp.s51 | ZDObject.h | f8wCoord.cfg | f8wRouter.cfg | ZDConfig.c | SampleApp.h | OSAL_SampleApp.c | MT_UART.c | mt_uart.h | sapi.h |
20 /* Coordinator Settings */
21 -DZDO_COORDINATOR                // Coordinator Functions
22 -DRTR_NWK                        // Router Functions
23
24 /* Optional Settings */
25 -DBLINK_LEDS                     // LED Blink Functions
26
27
```

Router Settings: -DRTR_NWK

```
ZDApp.s51 | ZDObject.h | f8wRouter.cfg | ZDConfig.c | SampleApp.h | OSAL_SampleApp.c | MT_UART.c | mt_uart.h | sapi.h | ZMain.c |
20 /* Router Settings */
21 -DRTR_NWK                        // Router Functions
22
23 /* Optional Settings */
24 -DBLINK_LEDS                     // LED Blink Functions
25
```

ZDO_COORDINATOR、RTR_NWK是否被定义决定了ZG_BUILD_COORDINATOR_TYPE、ZG_BUILD_RTR_TYPE、ZG_BUILD_ENDDEVICE_TYPE的值。

```
OSAL.c | OSAL_PwrMgr.c | ZGlobals.h | ZDSecMgr.s51 | ZGlobals.c | ZDApp.s51 | ZDObject.h | ZDConfig.c | SampleApp.h | OSAL_SampleApp.c | MT_UART.c | mt_uart.h | sapi.h | ZMain.c | AF.h | AddrMgr.h | mac_api.h | SampleApp.c | f
59 // Setup to work with the existing (old) compile flags
60 #if !defined ( ZSTACK_DEVICE_BUILD )
61     #if defined ( ZDO_COORDINATOR )
62         #define ZSTACK_DEVICE_BUILD (DEVICE_BUILD_COORDINATOR)
63     #elif defined ( RTR_NWK )
64         #define ZSTACK_DEVICE_BUILD (DEVICE_BUILD_ROUTER)
65     #else
66         #define ZSTACK_DEVICE_BUILD (DEVICE_BUILD_ENDDEVICE)
67     #endif
68 #endif
69
70 // Use the following to macros to make device type decisions
71 #define ZG_BUILD_COORDINATOR_TYPE (ZSTACK_DEVICE_BUILD & DEVICE_BUILD_COORDINATOR)
72 #define ZG_BUILD_RTR_TYPE (ZSTACK_DEVICE_BUILD & (DEVICE_BUILD_COORDINATOR | DEVICE_BUILD_ROUTER))
73 #define ZG_BUILD_ENDDEVICE_TYPE (ZSTACK_DEVICE_BUILD & DEVICE_BUILD_ENDDEVICE)
```

```
OSAL.c | OSAL_PwrMgr.c | ZGlobals.h | ZDSecMgr.s51 | ZGlobals.c | ZDApp.s51 | ZDObject.h | ZDConfig.c | SampleApp.h | OSAL_SampleApp.c
129 // Default Device Logical Type
130 #if !defined ( DEVICE_LOGICAL_TYPE )
131     #if ( ZG_BUILD_COORDINATOR_TYPE )
132         // If capable, default to coordinator
133         #define DEVICE_LOGICAL_TYPE ZG_DEVICETYPE_COORDINATOR
134     #elif ( ZG_BUILD_RTR_TYPE )
135         #define DEVICE_LOGICAL_TYPE ZG_DEVICETYPE_ROUTER
136     #elif ( ZG_BUILD_ENDDEVICE_TYPE )
137         // Must be an end device
138         #define DEVICE_LOGICAL_TYPE ZG_DEVICETYPE_ENDDEVICE
139     #else
140         #error ZSTACK_DEVICE_BUILD must be defined as something!
141     #endif
142 #endif
```

DEVICE_LOGI
CAL_TYPE就是
设备类型

DEVICE_LOGICAL_TYPE被赋值给全局变量 zgDeviceLogicalType，这个变量位于 ZGlobals.c 文件中，他是协议栈中的一个全局变量，我们用户不需要修改。协议栈可以根据这个变量判断设备是协调器、路由、还是终端设备来发送不同的内容。

```
OSAL.c | OSAL_PwrMgr.c | ZOSecMgr.s51 | ZGlobals.c | ZApp.s51 | ZObject.h | ZConfig.c | SampleApp.h | OSAL_SampleApp.c | MT_UART.c | mt_uart.h | sapi.h | ZMain.c | AF.h | AddrMgr.h | mac_api.h | SampleApp.c

190
191 // If true, preConfigKey should be configured on all devices on the network
192 // If false, it is configured only on the coordinator and sent to other
193 // devices upon joining.
194 uint8 zgPreConfigKeys = FALSE; // TRUE;
195
196 // If true, defaultTCLinkKey should be configured on all devices on the
197 // network. If false, individual trust center link key between each device and
198 // the trust center should be manually configured via MT_WRITE_NV
199 uint8 zgUseDefaultTCLK = TRUE; // FALSE
200
201
202 /*****
203  * ZDO GLOBAL VARIABLES
204  */
205
206 // Configured PAN ID
207 uint16 zgConfigPANID = ZDAPP_CONFIG_PAN_ID;
208
209 // Device Logical Type
210 uint8 zgDeviceLogicalType = DEVICE_LOGICAL_TYPE;
211
212 // Startup Delay
213 uint8 zgStartDelay = START_DELAY;
214
215 #if !defined MT_TASK
216 // Flag to use verbose (i.e. "cc2480-style") direct MT callbacks in ZDProfile.c, ZDP_IncomingData().
217 uint8 zgZdoDirectCB = FALSE;
218 #endif
219
220 /*****
```

终端描述符

端点（Endpoint）是协议栈应用层入口，它通常为节点的一个通信部件或设备（如各种类型的传感器、开关、LED 灯等）也是用户定义的应用对象驻留的地方。端点类似于在计算机 TCP/IP 网络通信中，为方便计算机中不同的进程进行通信，在传输层设置的端口（Port）概念。端点和 IEEE64 位扩展地址、16 位网络短地址一样，是 ZigBee 无线通信的一个重要地址参数，能够为多个应用对象提供逻辑子通道以进行通信。每个 ZigBee 设备支持多达 240 个端点，也就是说，每个设备最多可以定义 240 个应用对象。端点 0 必须在 ZigBee 设备中具有，它分配给 ZigBee 设备对象（ZD0）使用，端点 255 是端点广播地址，端点 241~254 保留，为以后扩展使用。端点描述符用来描述一个端点，Z-Stack 中的 AF.h 中有关于端点描述符的定义如下：

byte endpoint: 端点号。

byte *task_id: 指向任务 ID。

SimpleDescriptionFormat_t *simpleDesc: 简单描述符。

afNetworkLatencyReq_t latencyReq: 延迟，我们采用默认值初始化即可。

```
272 // Endpoint Table - this table is the device description
273 // or application registration.
274 // There will be one entry in this table for every
275 // endpoint defined.
276 typedef struct
277 {
278     byte endPoint;
279     byte *task_id; // Pointer to location of the Application task ID.
280     SimpleDescriptionFormat_t *simpleDesc;
281     afNetworkLatencyReq_t latencyReq;
282 } endPointDesc_t;
```

端点描述符需要在应用层 `SampleApp.c` 文件中的 `SampleApp_Init()` 函数中进行端点填充，并向 AF 层注册端点描述符。第一次对话实验中填充端点描述符的过程如下，该部分代码比较固定。

第 189 行：我们需要手动指定端口号，程序中 `SAMPLEAPP_ENDPOINT` 的宏值为 20，可以使用的端口号范围：1~240，用户可根据需要进行修改。

第 190 行：指定任务 ID，这里我们指向的是当前任务的 ID。

第 191~192 行：指向简单描述符。我们下边会分析简单描述符的定义。

第 193 行：采用默认值 `noLatencyReqs` 即可。

第 195 行：用 `afRegister()` 函数向 AF 层注册端点描述符。

```
188 // Fill out the endpoint description.
189 SampleApp_epDesc.endPoint = SAMPLEAPP_ENDPOINT;
190 SampleApp_epDesc.task_id = &SampleApp_TaskID;
191 SampleApp_epDesc.simpleDesc
192     = (SimpleDescriptionFormat_t *)&SampleApp_SimpleDesc;
193 SampleApp_epDesc.latencyReq = noLatencyReqs;
194 // Register the endpoint description with the AF
195 afRegister( &SampleApp_epDesc );
```

简单描述符

我们再来介绍下简单描述符 SimpleDescriptionFormat_t 的定义：

byte EndPoint: 端口号。

uint16 AppProfId: 应用规范 ID。

uint16 AppDeviceId: 应用设备 ID。

byte AppDevVer: 应用版本设备号（这里使用到了位域）。

byte Reserved: 保留（这里使用到了位域）。

byte AppNumInClusters: 输入簇的个数

cId_t *pAppInClusterList: 输入簇的列表

byte AppNumOutClusters: 输出簇的个数

cId_t *pAppOutClusterList: 输出簇的列表

```
178 // Simple Description Format Structure
179 typedef struct
180 {
181     byte        EndPoint;
182     uint16      AppProfId;
183     uint16      AppDeviceId;
184     byte        AppDevVer:4;
185     byte        Reserved:4;           // AF_V1_SUPPORT uses for AppFlags:4.
186     byte        AppNumInClusters;
187     cId_t       *pAppInClusterList;
188     byte        AppNumOutClusters;
189     cId_t       *pAppOutClusterList;
190 } SimpleDescriptionFormat_t;
```

实验代码中有一个SimpleDescriptionFormat_t类型的SampleApp_Simple Desc 常量，原型如下

```

99 const SimpleDescriptionFormat_t SampleApp_SimpleDesc =
100 {
101     SAMPLEAPP_ENDPOINT,           // int Endpoint;
102     SAMPLEAPP_PROFID,             // uint16 AppProfId[2];
103     SAMPLEAPP_DEVICEID,          // uint16 AppDeviceId[2];
104     SAMPLEAPP_DEVICE_VERSION,    // int AppDevVer:4;
105     SAMPLEAPP_FLAGS,             // int AppFlags:4;
106     SAMPLEAPP_MAX_CLUSTERS,       // uint8 AppNumInClusters;
107     (cId_t *)SampleApp_ClusterList, // uint8 *pAppInClusterList;
108     SAMPLEAPP_MAX_CLUSTERS,       // uint8 AppNumInClusters;
109     (cId_t *)SampleApp_ClusterList // uint8 *pAppInClusterList;
110 };

57 // These constants are only for example and should be changed to the
58 // device's needs
59 #define SAMPLEAPP_ENDPOINT          20
60 #define SAMPLEAPP_PROFID            0x0F08
61 #define SAMPLEAPP_DEVICEID          0x0001
62 #define SAMPLEAPP_DEVICE_VERSION    0
63 #define SAMPLEAPP_FLAGS              0
64 #define SAMPLEAPP_MAX_CLUSTERS      2
65 #define SAMPLEAPP_PERIODIC_CLUSTERID 1
    
```

发送数据

在 ZigBee 协议栈中进行数据发送可以调用 AF_DataRequest()函数实现，该函数会调用协议栈里面与硬件相关的函数最终将数据发送出去，这里面涉及对射频模块得操作，这部分协议栈已经实现了，我们不需要自己写代码实现，只需要掌握 AF_DataRequest()函数的使用方法即可。我们来看下该函数的参数：

afStatus_t AF_DataRequest(

afAddrType_t *dstAddr, //包含目的节点的网络地址以及发送数据的格式，如广播、组播或单播。

endPointDesc_t *srcEP, //某一个节点上不同的端口（endpoint），我们可以理解为 TCP/IP 中的端口。

uint16cID, //该参数主要是为了区分不同的命令，对应简单描述符中的输出群集

uint16 len, //该参数标识了发送数据的长度。

uint8 *buf, //该参数是指向发送缓冲区的指针。

uint8 *transID, //该参数是指向发送序号的指针。可以用来检测丢包率，由OSAL维护。

uint8 options, //取默认值 AF_DISCV_ROUTE 即可。

uint8 radius //取默认值 AF_DEFAULT_RADIUS 即可。

)

发送模式及目的地址

ZigBee协议栈将数据通信过程高度抽象，使用一个函数完成数据的发送，以不同的参数来选择数据发送方式（广播、组播还是单播）。在AF_Data Request函数中，第一个参数是一个指向afAddrType_t类型的结构体的指针，该结构体的定义如下：typedef struct

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

注意加粗字体部分的addrMode，该参数是一个afAddrMode_t类型的变量，afAddrMode_t类型的定义如下：




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



```
enum
{
    AddrNotPresent=0,
    AddrGroup=1,
    Addr16Bit=2,
    Addr64Bit=3,
    AddrBroadcast=15
}
```

当addrMode= AddrBroadcast时，就对应的广播方式发送数据；

当addrMode= AddrGroup时，就对应的组播方式发送数据；

当addrMode= Addr16Bit时，就对应的单播方式发送数据。

接收数据

当设备接收到无线数据后，操作系统会将该数据封装成一个消息然后放入消息队列中，同时设置系统事件SYS_EVENT_MSG，应用层的事件处理函数 SampleApp_ProcessEvent 会处理这个事件，他首先会将消息从消息队列中取出，根据消息的 ID 判断该消息是否就是接收到的数据，标识接收到新数据的消息 ID 是 AF_INCOMING_MSG_CMD，其中 AF_INCOMING_MSG_CMD 的值是 0x1a，这是在 ZigBee 协议栈中定义好的，用户不可更改。

File	Line	Code	Comment
mt_uart.h	323	#define MT_SYS_APP_MSG	0x23 // Raw data from an MT Sys message
sapi.h	324	#define MT_SYS_APP_RSP_MSG	0x24 // Raw data output for an MT Sys message
ZMain.c	325		
AF.h	326	#define AF_DATA_CONFIRM_CMD	0xFD // Data confirmation
ZComDef.h	327	#define AF_INCOMING_MSG_CMD	0x1A // Incoming MSG type message
AddrMgr.h	328	#define AF_INCOMING_KVP_CMD	0x1B // Incoming KVP type message
mac_api.h	329	#define AF_INCOMING_GRP_KVP_CMD	0x1C // Incoming Group KVP type message
ZDApp.c			
MT_APP.c			
ZGlobals.s51			
SampleApp.s51			
OnBoard.h			
ZMain.s51			
f8wConfig.cfg			
ZDApp.h			
OSAL.c			

具体的事件处理函数代码如下所示，这部分代码比较固定，我们只需要熟悉这种事件处理结构即可。

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
    afIncomingMSGPacket_t *MSGpkt;
    (void)task_id; // Intentionally unreferenced parameter
    if ( events & SYS_EVENT_MSG )
    {
        MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );
        while ( MSGpkt )
        {
            switch ( MSGpkt->hdr.event )
            {
                // Received when a messages is received (OTA) for this endpoint
                case AF_INCOMING_MSG_CMD:
                    SampleApp_MessageMSGCB( MSGpkt ); //我们去看下该函数做了什么?
                    break;
                default:
                    break;
            }
        }
        // Release the memory
        osal_msg_deallocate( (uint8 *)MSGpkt );
        // Next - if one is available
        MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );
    }
    // return unprocessed events
    return (events ^ SYS_EVENT_MSG);
}
// Discard unknown events
return 0;
}
```

```
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
    switch ( pkt->clusterId )
    {
        case SAMPLEAPP_PERIODIC_CLUSTERID:
            HalUARTWrite (0, (pkt->cmd).Data, ((pkt->cmd).DataLength));
            break;
    }
}
```

思考两个问题：

为什么要根据SAMPLEAPP_PERIODIC_CLUSTERID 才能得到正确的数据呢？

在 ZigBee 网络中进行通信时需要注意的，两者如果想进行通信，需要具有相同的 Cluster ID，而且一个选择“输入”，另一个选择“输出”（在简单描述符中填充）。

收到的数据在哪里？

在 pkt 指向的那个结构体里，我们去看下 afIncomingMSGPacket_t 的真面目了。

接收数据的存放位置

afIncomingMSGPacket_t 结构体的成员变量很多，不宜去强制记忆这些成员变量，我们需要做的就是根据 TI 给出的注释来尝试使用这些参数。在此我们只使用了 cmd 成员变量，这是一个 afMSGCommandFormat_t 类型的变量。

```

247 typedef struct
248 {
249     osal_event_hdr_t hdr;           /* OSAL Message header */
250     uint16 groupId;                 /* Message's group ID - 0 if not set */
251     uint16 clusterId;               /* Message's cluster ID */
252     afAddrType_t srcAddr;           /* Source Address, if endpoint is STUBAPS_INTER_PAN_EP,
253                                     it's an InterPAN message */
254     uint16 macDestAddr;              /* MAC header destination short address */
255     uint8 endPoint;                 /* destination endpoint */
256     uint8 wasBroadcast;              /* TRUE if network destination was a broadcast address */
257     uint8 LinkQuality;              /* The link quality of the received data frame */
258     uint8 correlation;              /* The raw correlation value of the received data frame */
259     int8 rssi;                      /* The received RF power in units dBm */
260     uint8 SecurityUse;               /* deprecated */
261     uint32 timestamp;               /* receipt timestamp from MAC */
262     afMSGCommandFormat_t cmd;        /* Application Data */
263 } afIncomingMSGPacket_t;
    
```

我们来看一下 afMSGCommandFormat_t 的定义,该结构体有三个成员变量:

byte TransSeqNumber: 用于存储发送序列号;

uint16 DataLength: 用于存储接收到的数据长度;

byte *Data: 数据接收后放在一个缓冲区, 该参数就是指向缓冲区的指针;

```
206 // Generalized MSG Command Format
207 typedef struct
208 {
209     byte    TransSeqNumber;
210     uint16  DataLength;           // Number of bytes in TransData
211     byte    *Data;
212 } afMSGCommandFormat_t;
```

ZigBee—SensorDemo

SensorDemo实现树状网

TI 提供的 SensorDemo 例程非常不错，配合 Zig Bee Sensor Monitor 上位机软件还可以观察 ZigBee 的树形网络拓扑结构。我们需要做以下工作：

- ◆ 查看使用文档
- ◆ 安装 ZigBee Sensor Monitor
- ◆ 下载SensorDemo源码



CC2530ZDK_Sensor_Demo_Users_Guide.pdf: 文档中描述了例程的基本功能、如何复原实验、简易的代码分析等。

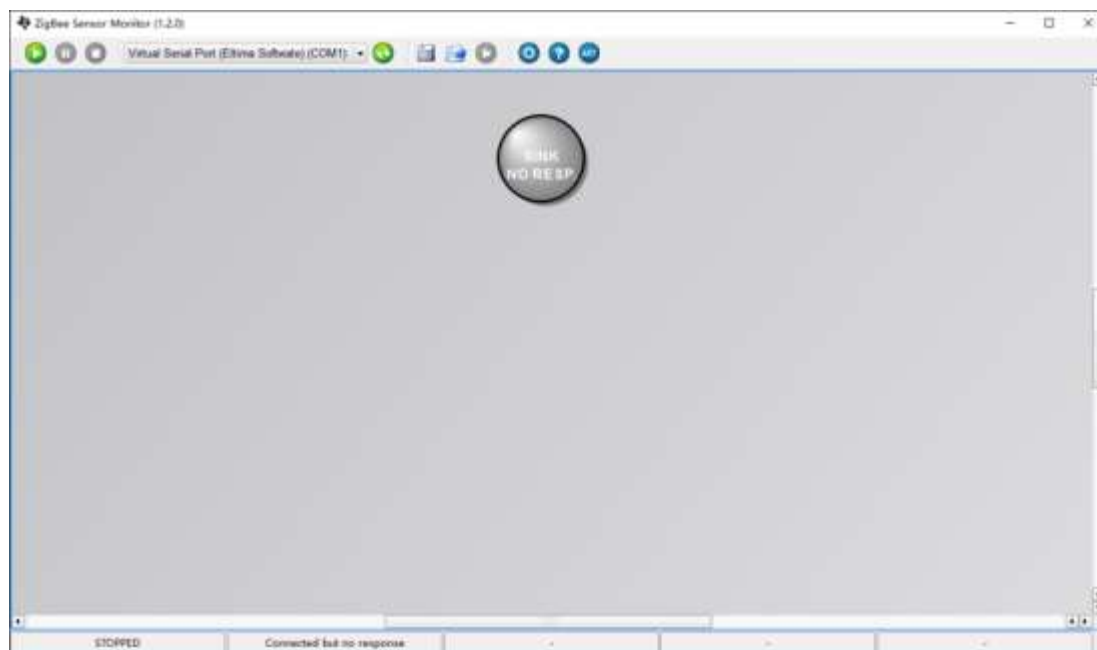
ZSensorMonitor User's Guide.pdf: 文档中描述了软件的安装及使用方法。我们根据文档安装Z-Sensor Monitor软件即可。



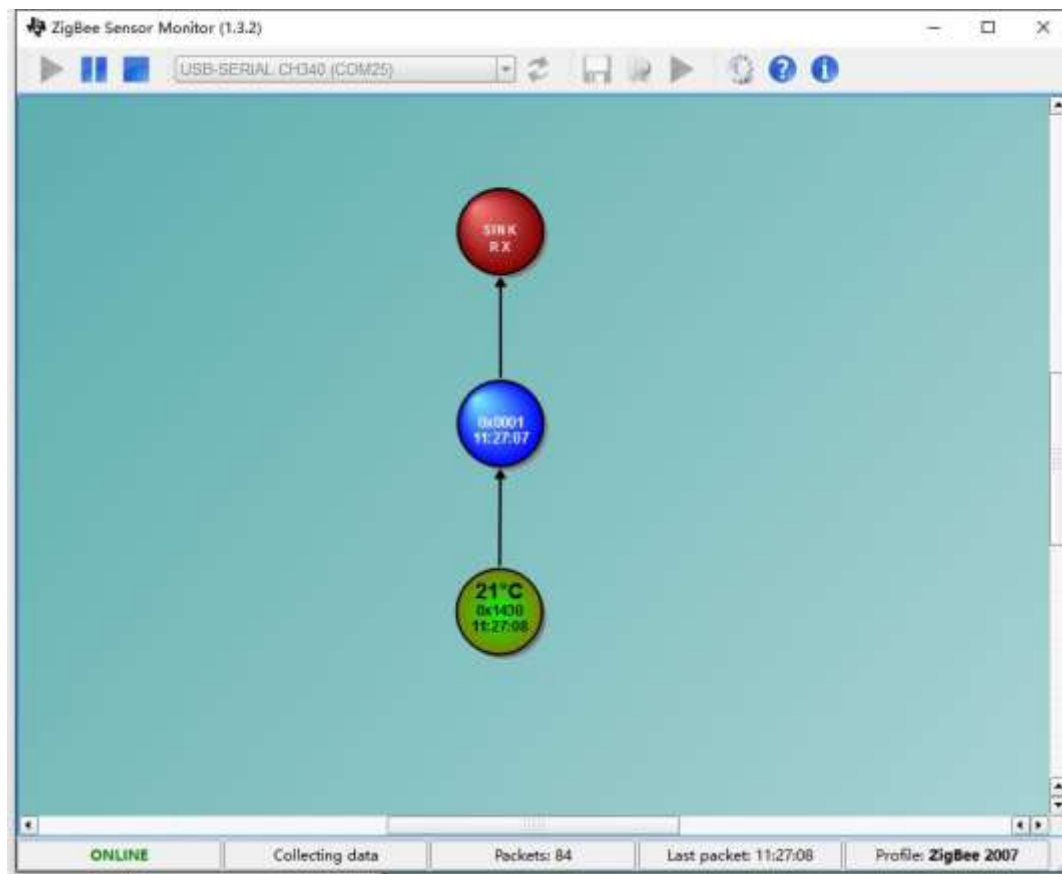
CC2530ZDK_Sensor_Demo_Users_Guide.pdf



ZSensorMonitor User's Guide.pdf



我们可以从Z-Sensor Monitor软件直观的看到当前的网络拓扑结构。



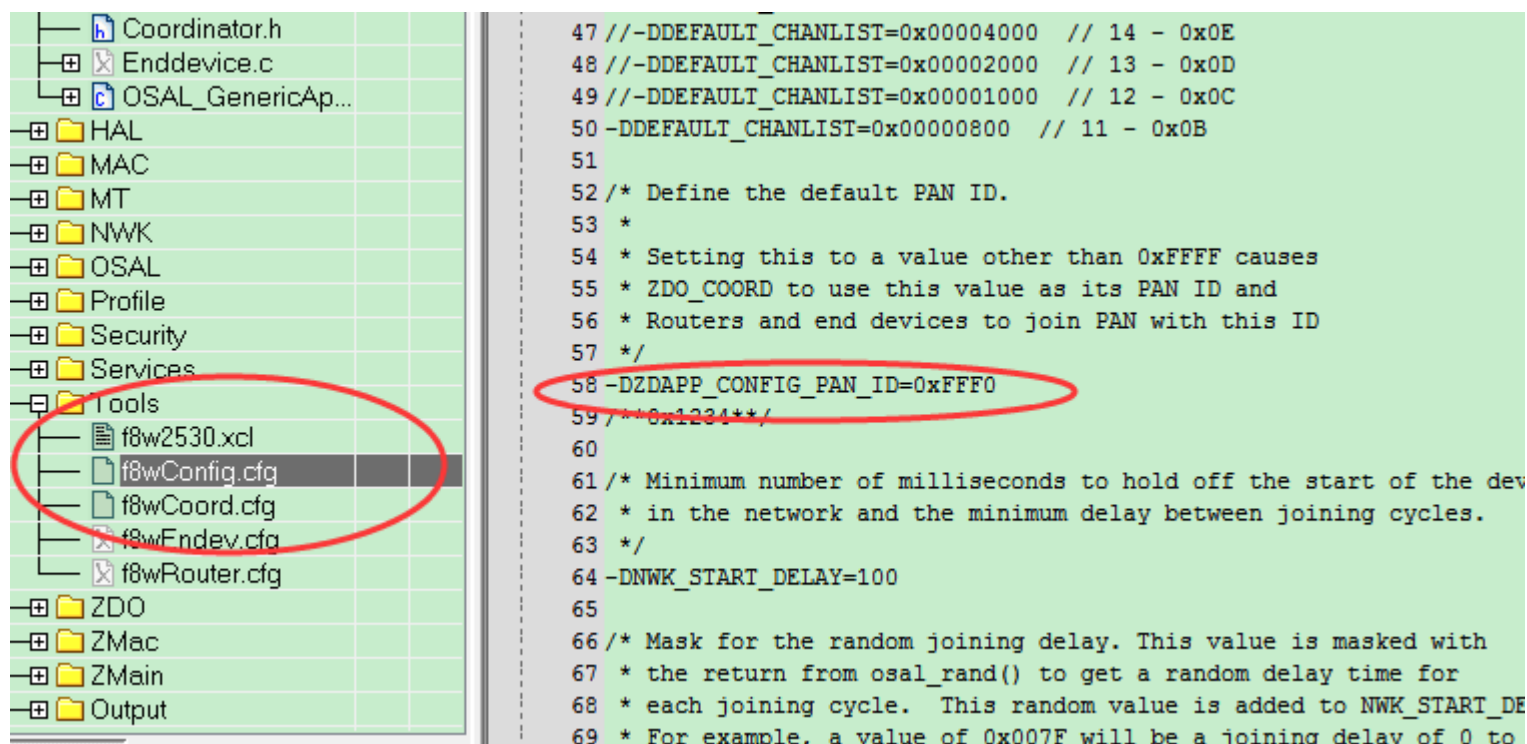
ZigBee—PanID

PanID: 区分不同的网络

ZigBee 协议使用一个 16 位的个域网标志符 (PAN ID) 来标识一个网络。ZStack 允许用两种方式配置 PAN ID, 当 ZDAPP_CONFIG_PAN_ID (在协议栈Tools文件夹下f8wConfig.cfg文件中) 的值不设置为 0xFFFF 时, 那么设备建立或加入网络的 PAN ID 由 ZDAPP_CONFIG_PAN_ID 指定; 如果设置ZDAPP_CONFIG_PAN_ID 为 0xFFFF, 那么协调器设备就将建立一个“最优”的网络 (PANID)。PANID的范围为: 0~0x3FFF.



修改PAN ID 的位置如下图所示：



```
47 //-DDEFAULT_CHANLIST=0x00004000 // 14 - 0x0E
48 //-DDEFAULT_CHANLIST=0x00002000 // 13 - 0x0D
49 //-DDEFAULT_CHANLIST=0x00001000 // 12 - 0x0C
50 -DDEFAULT_CHANLIST=0x00000800 // 11 - 0x0B
51
52 /* Define the default PAN ID.
53 *
54 * Setting this to a value other than 0xFFFF causes
55 * ZDO_COORD to use this value as its PAN ID and
56 * Routers and end devices to join PAN with this ID
57 */
58 -DZDAPP_CONFIG_PAN_ID=0xFFFF0
59 /*0x1234*/
60
61 /* Minimum number of milliseconds to hold off the start of the dev
62 * in the network and the minimum delay between joining cycles.
63 */
64 -DNWK_START_DELAY=100
65
66 /* Mask for the random joining delay. This value is masked with
67 * the return from osal_rand() to get a random delay time for
68 * each joining cycle. This random value is added to NWK_START_DE
69 * For example, a value of 0x007F will be a joining delay of 0 to
```

谢谢！

