更多嵌入式 Linux 学习资料，请关注：一口 Linux 回复关键字:1024

个人微信


公众号

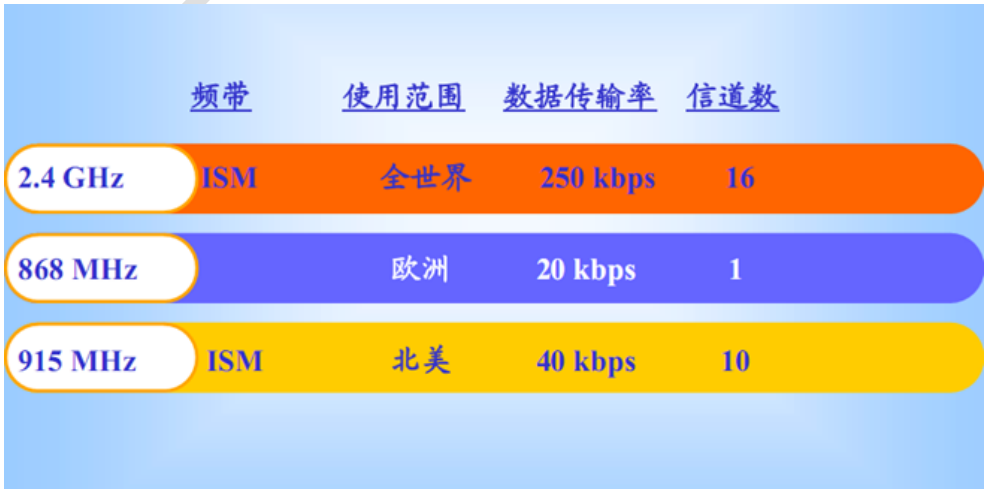
## 一、Zigbee 概述

ZigBee 是一种开放式的基于 IEEE 802.15.4 协定的无线个人局域网(Wireless Personal Area Networks)标准。

IEEE 802.15.4 定义了物理层和媒体接入控制层，而 ZigBee 则定义了更高层如网路层及应用层等。 ZigBee 技术是一种近距离、低复杂度、低功耗、低速率、低成本的双向无线通讯技术。

早期也被称为 “HomeRF Lite”、“RF- EasyLink”或“fireFly”无线电技术，目前统称为 ZigBee 技术。

ZigBee 可工作在 2.14GHz(全球流行)、868MHz(欧洲流行) 和 915 MHz(美国流行)3 个频段上,分别具有最高 250kbit/s、20kbit/s 和 40kbit/s 的传输速率,它的传输距离在 10-75m 的范围内, 但可以继续增加 。



# 1. ZigBee 技术发展历程

1. ZigBee 的前身是 1998 年由 INTEL、IBM 等产业巨头发起的“HomeRFLite”技术。
2. 2000 年 12 月成立了工作小组起草 IEEE 802.15.4 标准
3. 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。
4. 2004 年 12 月 ZigBee1.0 标准(又称为 ZigBee2004)敲定，这使得 ZigBee 有了自己的发展基本标准。
5. 2005 年 9 月公布 ZigBee1.0 标准并提供下载。在这一年里，华为技术有限公司和 IBM 公司加入了 ZigBee 联盟。但是基于该版本的应用很少，与后面的版本也不兼容。
6. 2006 年 12 月进行标准修订，推出 ZigBee1.1 版(又称为 ZigBee2006)。该协议虽然命名为 ZigBee1.1，但是与 ZigBee1.0 版是不兼容的。
7. 2007 年 10 月完成再次修订(称为 ZigBee2007/PRO)。能够兼容之前的 ZigBee2006 版本，并且加入了 ZibgeePRO 部分，此时 ZigBee 联盟更加专注于以下三个方面：

- 1)、家庭自动化(Home Automation；HA)；
- 2)、建筑/商业大楼自动化(Building Automation；BA)；
- 3)、先进抄表基础建设(Advanced Meter Infrastructure；AMI)；

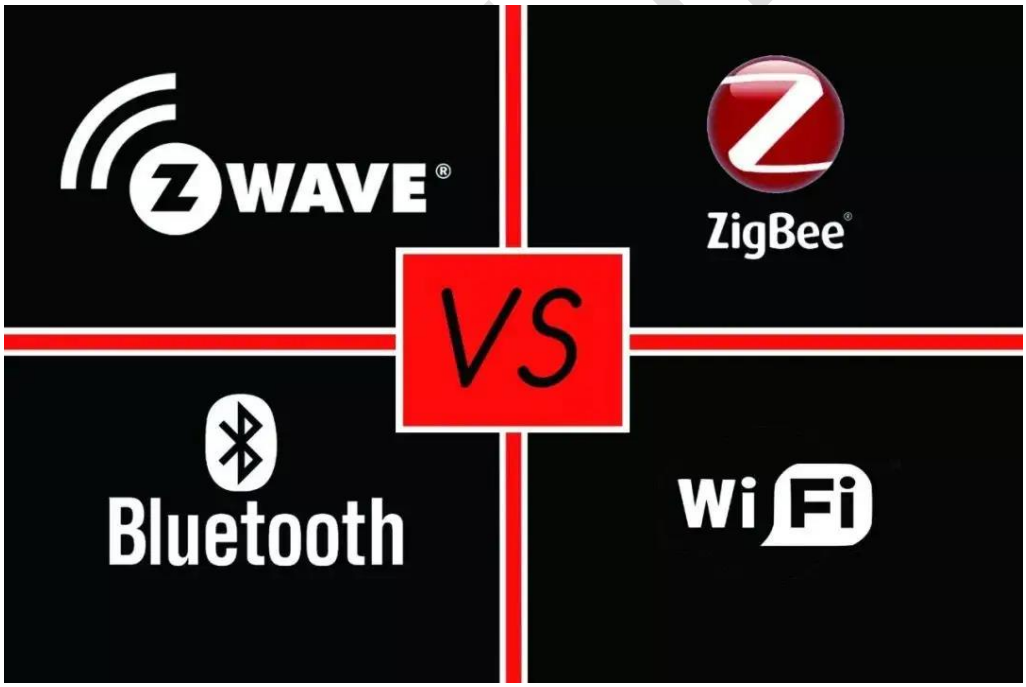

# 2. ZigBee 具体如下技术特点：

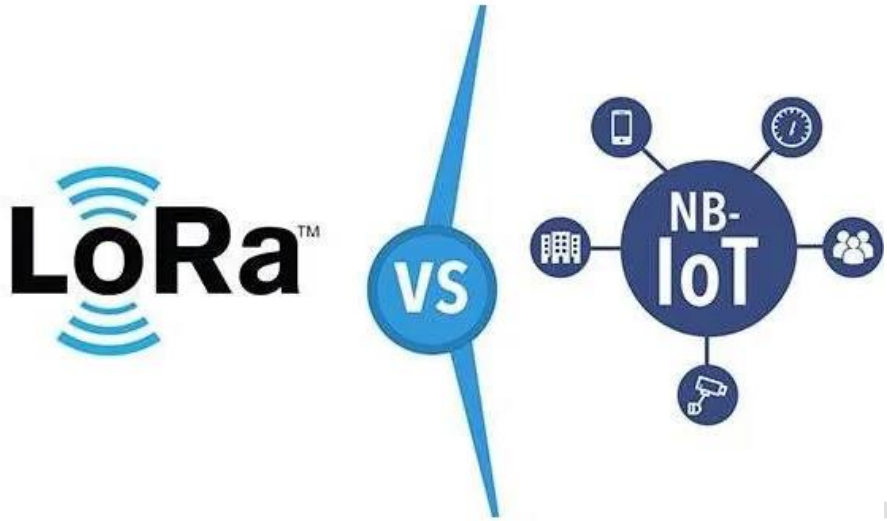
1. 低功耗 由于 ZigBee 的传输速率低,发射功率仅为 1mW,而且采用了休眠模式,功耗低，因此 ZigBee 设备非常省电。据估算,ZigBee 设备仅靠两节 5 号电池就可以维持长达 6 个月到 2 年左右的使用时间。

2. 低成本 由于 ZigBee 模块的复杂度不高，ZigBee 协议免专利费，再加之使用的频段无需付费，所以它的成本较低。
3. 时延短 通信时延和从休眠状态激活的时延都非常短,典型的搜索设备时延 30ms,休眠激活的时延是 15ms，活动设备信道接入的时延为 15ms。
4. 网络容量大 一个星型结构的 ZigBee 网络最多可以容纳 254 个从设备和一个主设备，一个区域内可以同时存在最多 100 个 ZigBee 网络，而且网络组成灵活。网状结构的 ZigBee 网络中可有 65000 多个节点。
5. 可靠 采取了碰撞避免策略,同时为需要固定带宽的通信业务预留了专用时隙,避开了发送数据的竞争和冲突。MAC 层采用了完全确认的数据传输模式，每个发送的数据包都必须等待接收方的确认信息。如果传输过程中出现问题可进行重发。
6. 安全 ZigBee 提供了基于循环冗余校验(CRC)的数据包完整性检查功能,支持鉴权和认证，采用了 AES-128 的加密算法,各个应用可以灵活确定其安全属性。

## 3. 其他无线通信协议

除了 zigbee 之外，其他常用的无线通信协议有:WAVE 、Bluetooth、WiFi、LoRa、NBIoT 等。






# 二、zigbee 设备类型及组网

## 1. 设备类型

在 ZigBee 无线传感器网络中有三种设备类型：

1. 协调器
2. 路由器
3. 终端节点

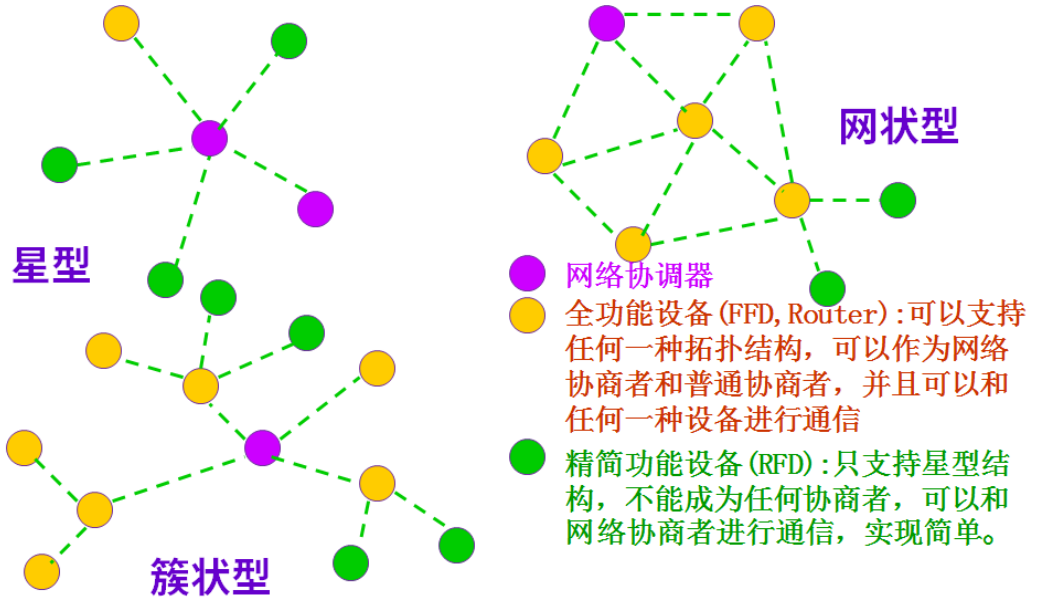
**ZigBee 协调器（Coordinator）** 它包含所有的网络信息，是 3 种设备中最复杂的，存储容量大、计算能力最强。它主要用于发送网络信标、建立一个网络、管理网络节点、存储网络节点信息、寻找一对节点间的路由信息并且不断的接收信息。一旦网络建立完成,这个协调器的作用就像路由器节点。

**ZigBee 路由器（Router）** 它执行的功能包括允许其它设备加入这个网络，跳跃路由，辅助子树下电池供电终端的通信。通常,路由器全时间处在活动状态,因此为主供电。但是在树状拓扑中,允许路由器操作周期运行,因此这个情况下允许路由器电池供电。

**ZigBee 终端设备（End-device）** 一个终端设备对于维护这个网络设备没有具体的责任,所以它可以睡眠和唤配，看它自己的选择。因此它能作为电池供电节点。

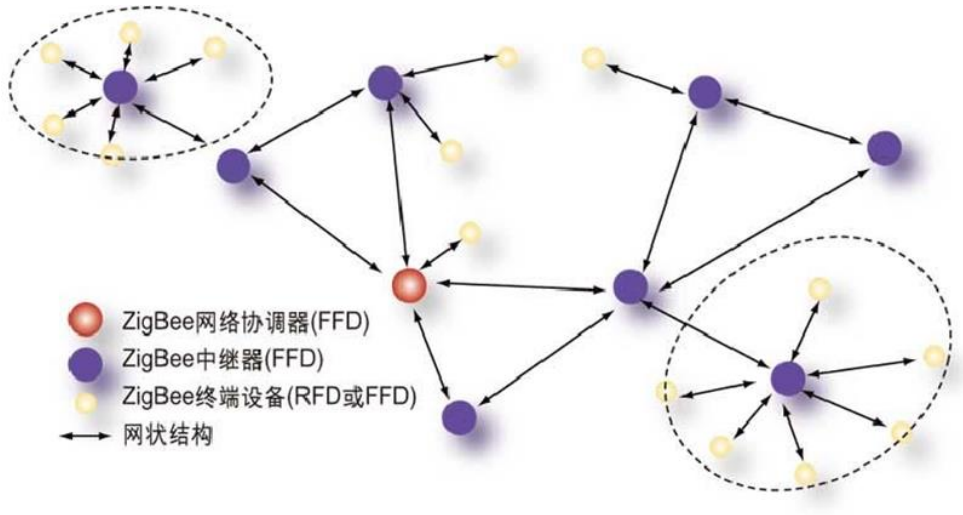
## 2. ZigBee 网络拓扑

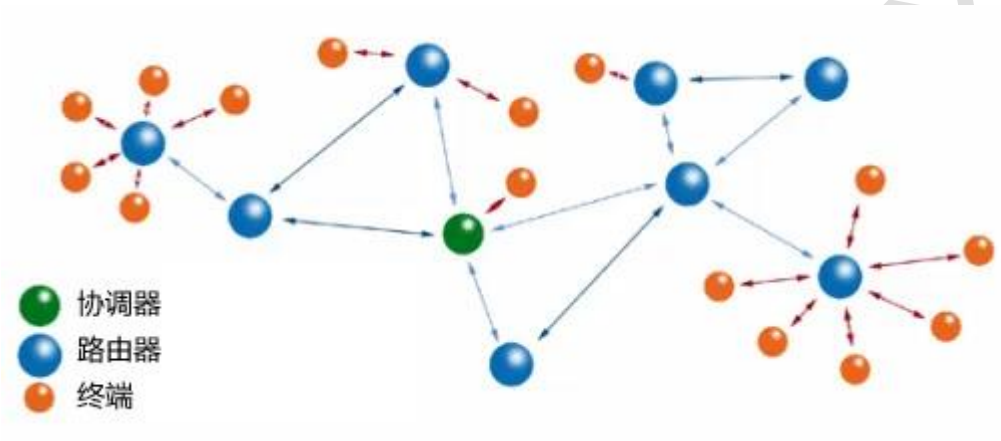
ZigBee 支持三种自组织无线网络类型，即星型结构、网状结构(Mesh)和簇状结构(ClusterTree)，特别是网状结构，具有很强的网络健壮性和系统可靠性。




## 3. ZigBee 网状（MESH）网络

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。




# 三、ZigBee 网络的建立

## 1.协调器建立一个新网络的流程

- 1） 检测协调器 节点必须具备两个条件，一是这个节点具有 ZigBee 协调器功能，二是这个节点没有加入到其它网络中。任何不满足这两个条件的节点发起建立一个新网络的进程都会被网络层管理实体终止，
- 2）信道扫描 信道扫描包括能量扫描和主动扫描两个过程。
- 3） 配置网络参数 网络层管理实体将为新网络选择一个 PAN 描述符，必须满足 PAN 描述符小于或等于 0x3fff，不等于 0xffff，并且在所选信道内是唯一的 PAN 描述符
- 4） 运行新网络
- 5） 允许设备加入网络 只有 ZigBee 协调器或路由器才能通过 NLME_PERMIT_JOINING.request 原语来设置节点处于允许设备加入网络的状态。

## 2. 节点加入网络

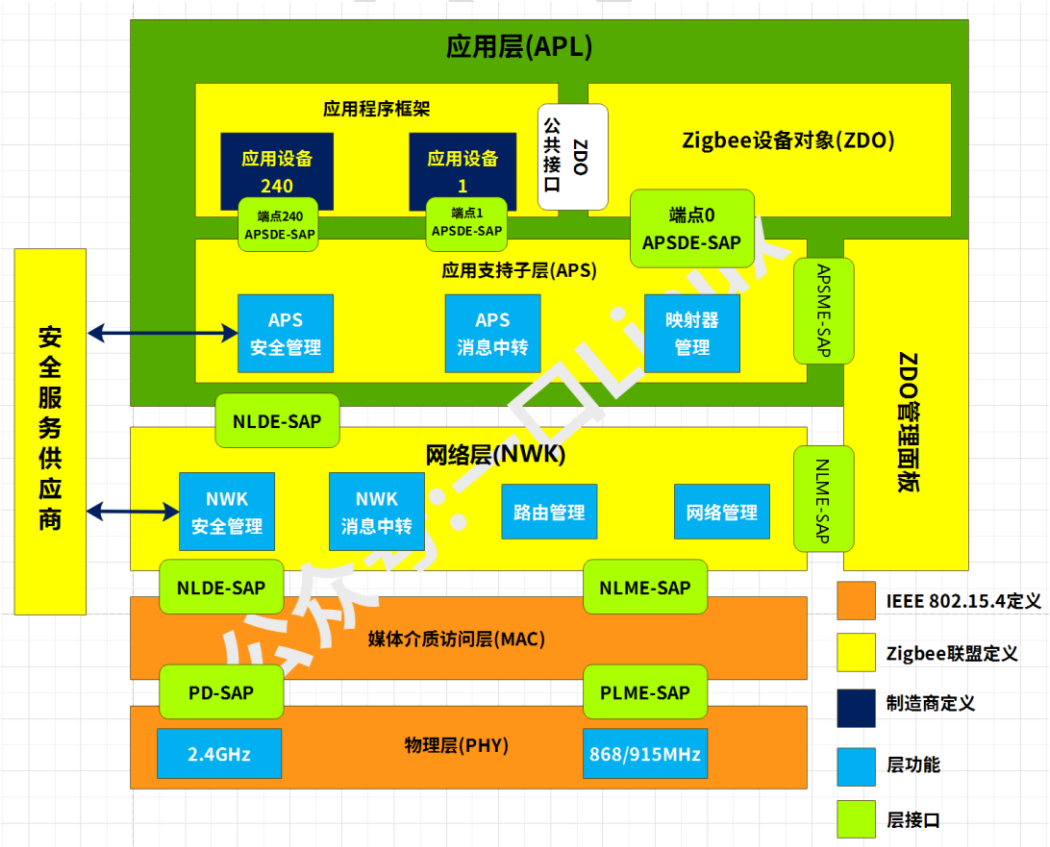
- 1） 通过 MAC 层关联加入网络

1、子节点发起信道扫描
2、 子节点存储各 PAN 信息
3、 子节点选择 PAN
4、 子节点选择父节点
5、 子节点请求 MAC 关联
6、 父节点响应 MAC 关联
7、 子节点响应连接成功
8、 父节点响应连接成功

- 2） 通过与先前指定父节点连接加入网络

# 四、zigbee 协议栈安装

ZigBee 的协议分为两部分， IEEE802.15.4 定义了物理层和 MAC 层技术规范， ZigBee 联盟定义了网络层、安全层和应用层技术规范， ZigBee 协议栈就是将各个层定义的协议都集合在一起，以函数的形式实现，并给用户提供一些应用层 API,供用户调用。



ZigBee 协议栈具有很多版本，不同厂商提供的 ZigBee 协议栈有一定的区别。
注意：虽然协议是统一的，但是协议的具体实现形式是变化的，即不同厂商提供的协议栈是有区别的，
例如： 函数名称和参数列表可能有区别，用户在选择协议栈以后，需要学习具体的例子，查看厂商提供的 Demo 演示程序和说明文档来学习各个函数的使用方式，进而快速的使用协议栈进行应用程序的开发工作。

本文选用的是 TI 推出的 ZigBee2007 协议栈进行讲解。

ZigBee 2007 协议栈 ZStack-CC2530-2.5.1a 要安装以后才能使用，下面讲解安装步骤。
协议栈安装文件目录:

```
物联网实训项目所有资料\环境工具\ZStack-CC2530-2.5.1a\ZStack-CC2530-2.5.1a.exe
```

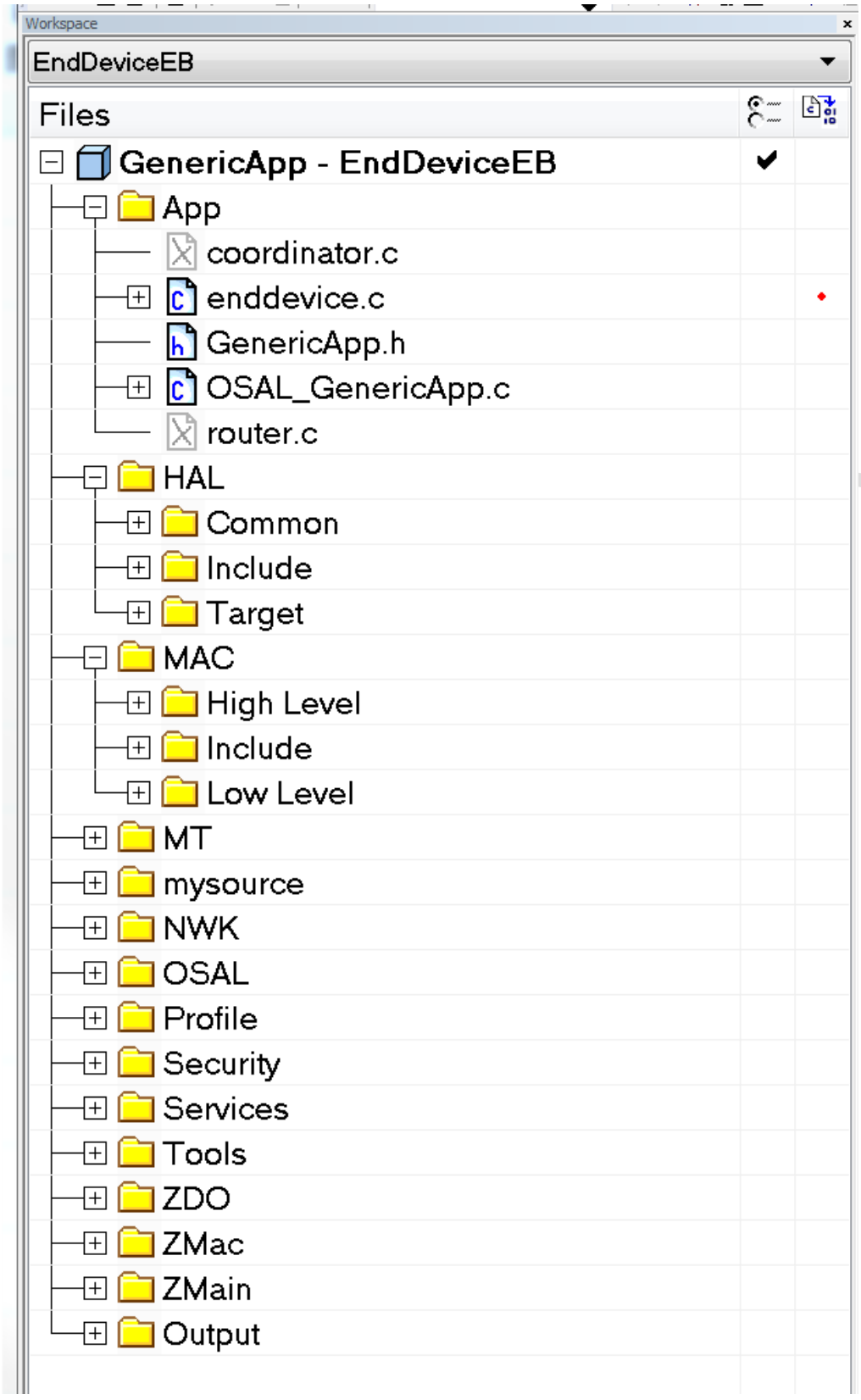
双击 ZStack-CC2530-2.5.1a.exe,即可进行协议栈的安装，默认是安装到 c 盘。
然后在路径：

```
C:\Texas Instruments\ZStack-CC2530-2.5.1a\Projects\zstack\Samples\GenericApp\CC2530DB
```

下找到 GenericApp.eww 打开该工程。
不过一口君已经给出的实例都已经包括了协议栈的代码，只需要点击任意一实例，找到如下目录:

```
ZStack-CC2530-2.5.1a\Projects\zstack\Samples\GenericApp\CC2530DB
```

点击 GenericApp.eww 打开该工程。




## 五、zigbee 协议栈使用

使用 ZigBee 协议栈进行开发的基本思路可以概括为如下三点：

- 1、用户对于 ZigBee 无线网络的开发就简化为应用层的 c 语言程序开发，用户不需要深入研究复杂的 ZigBee 协议栈；
- 2、 ZigBee 无线传感器网络中数据采集，只需用户在应用层加入传感器的读取函数即可；
- 3、如果考虑节能，可以根据数据采集周期进行定时，定时时间到就唤醒 ZigBee 的终端节点，终端节点唤醒后，自动采集传感器数据，然后将数据发送给路由器或者直接发给协调器。

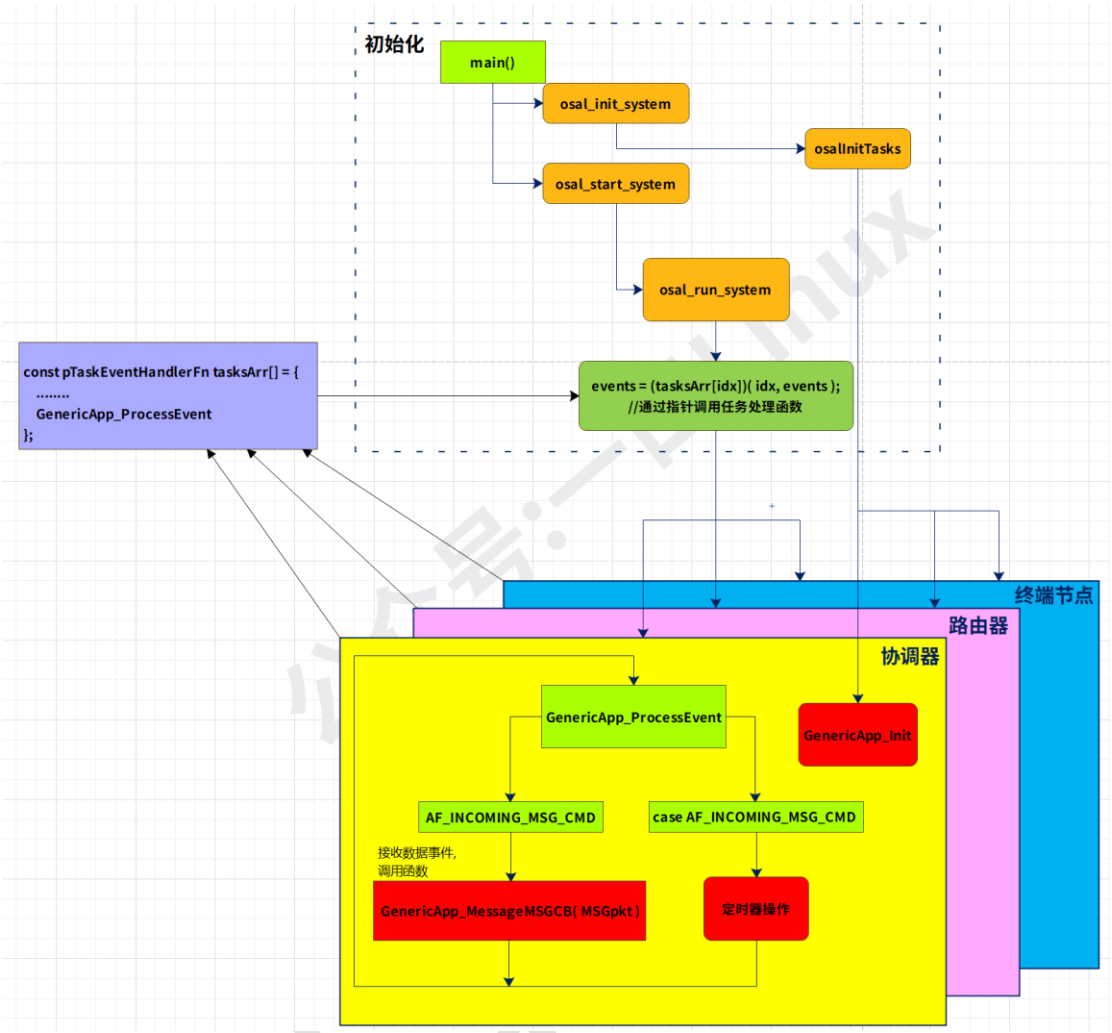
既然 ZigBee 协议栈已经实现了 ZigBee 协议，那么用户就可以使用协议栈提供的 API 进行应用程序的开发，在开发过程中完全不必关心 ZigBee 协议的具体实现细节，只需关心一个核心的问题：应用程序数据从哪里来到哪里去。
下面举一个例子，当用户应用程序需要进行数据通信时，需要按照如下步骤实现：

- （1）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）发送设备调用协议栈提供的无线数据发送函数，实现数据的发送；
- （3）接收设备调用协议栈提供的无线数据接收函数，实现数据的正确接收。

因此，使用协议栈进行应用程序开发时，开发者不需要关心协议栈是具体怎么实现的（例如，每个函数是怎么实现的，每条函数代码是什么意思等），只需要知道协议栈提供的函数实现什么样的功能，会调用相应的函数来实现自己的应用需求即可。
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初始化工作初始化好了。
这就类似于学习 TCP/IP 网络编程时，使用的 socket 编程，用户只需要调用 socket 相应的数据发送、接收 API 函数即可，而不必关心具体的网卡驱动的具体实现细节。

## 六、zigbee 协议栈重要的函数

zigbee 协议栈初始化重要的函数调用关系如下：




下面详细讲解这些函数功能。

## 1. main()

该函数是所有函数入口 ，

```
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
 zgInit(); //初始化非易失变量
```

```
#ifndef NONWK
  // 因为AF 不是任务，所以调用它的初始化例程
 afInit();
 #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
    //如果使用WDT，这是启用它的好地方。
 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; //无返回值
}
```

main 函数先执行初始化工作，包括硬件、网络层、任务等初始化。 然后进行 osal_start_system();操作系统，进去后无法返回。 在这里有两个重要的函数

- 初始化操作系统 osal_init_system();
- 运行操作系统 osal_start_system();

## 系统初始化函数，osal_init_system();

我们先看 osal_init_system();

```
uint8 osal_init_system( void )
{
  // Initialize the Memory Allocation System
  osal_mem_init();

  // Initialize the message queue
  osal_qHead = NULL;

  // Initialize the timers
```

```
  osalTimerInit();

  // Initialize the Power Management System
  osal_pwrmgr_init();

  // Initialize the system tasks.
  osalInitTasks();

  // Setup efficient search for the first free block of heap.
  osal_mem_kick();

  return ( SUCCESS );
}
```

包含 6 个初始化函数，在这里我们只看 osal_InitTasks();任务初始化函数。

## 2、初始化操作系统 osalInitTasks();任务初始化函数

函数所在文件目录:

```
Projects\zstack\Samples\GenericApp\Source\OSAL_GenericApp.c
```

函数代码如下：

```
void osalInitTasks( void )
{
 uint8 taskID = 0;
  // 分配内存，返回指向缓冲区的指针
 tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
  // 设置所分配的内存空间单元值为 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) ，用户不需考虑
 nwk_init( taskID++ ); //nwk_init(1)，用户不需考虑
 Hal_Init( taskID++ ); //Hal_Init(2) ，用户需考虑
 #if defined( MT_TASK ) //如果定义 MT_TASK 则调用 MT_TaskInit （）
 MT_TaskInit( taskID++ );
 #endif
 APS_Init( taskID++ ); //APS_Init(3) ，用户不需考虑
 #if defined ( ZIGBEE_FRAGMENTATION )
 APSF_Init( taskID++ );
 #endif
 ZDApp_Init( taskID++ ); //ZDApp_Init(4) ，用户需考虑
 #if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
 ZDNwkMgr_Init( taskID++ );
 #endif
```

```
  //用户创建的任务
 GenericApp_Init( taskID ); // SampleApp_Init _Init(5)，用户需考虑。重要!
}
```

- 函数对 taskID 进行初始化，每初始化一个，taskID++。
- 有些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。
- 需要考虑的用户可以根据自己的硬件平台进行其他设置，而写着不需考虑的也是不能修改的。TI 公司协议栈已完成。
- GenericApp_Init()是我们应用协议栈的必要函数，用户通常在这里初始化自己的东西。

## 3、 osal_start_system()

运行操作系统

```
void osal_start_system( void )
{
#if !defined ( ZBIT ) && !defined ( UBIT )
  for(;;)  // Forever Loop
#endif
  {
    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]) // T 务是准备好的最高优先级
    {
      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 将未处理的事件返回到当前任务。
    HAL_EXIT_CRITICAL_SECTION(intState); // 退出临界区
  }
  #if defined( POWER_SAVING )
  else // 完全通过所有没有活动的任务事件?
  {
    osal_pwrmgr_powerconserve(); // 使处理器/系统进入休眠状态
  }
  #endif
 #if defined (configUSE_PREEMPTION) && (configUSE_PREEMPTION == 0)
  {
    osal_task_yield();
  }
}
```

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

进 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 tasksArr。 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。

该结构体定义:

```
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 )
  ZDNwkMgr_event_loop,
#endif
  GenericApp_ProcessEvent
};
```

这是初始化的所有任务函数入口。

协调器和终端节点都初始化了函数 GenericApp_ProcessEvent()。

## 4、GenericApp_Init()用户应用任务初始化函数

协调器、路由器、终端节点均定义了该函数，
设备类型是由 ZigBee 协议栈不同的编译选项来选择的，
根据设备的类型不同会编译不同的文件，
下面以协调器代码为例讲解：

```
Projects\zstack\Samples\GenericApp\Source\coordinator.c

void GenericApp_Init( uint8 task_id )
{
 GenericApp_TaskID = task_id;//osal 分配的任务 ID 随着用户添加任务的增多而改变
  GenericApp_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**。 */
  GenericApp_TransID = 0; //消息发送 ID（多消息时有顺序之分）
  //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
 GenericApp_DstAddr.addrMode = (afAddrMode_t)AddrBroadcast; //广播
  GenericApp_DstAddr.endPoint = GENERICAPP_ENDPOINT; //指定端点号
  GenericApp_DstAddr.addr.shortAddr = 0xFFFF;//指定目的网络地址为广播地址

  //定义本设备用来通信的 APS 层端点描述符
  GenericApp_epDesc.endPoint = GENERICAPP_ENDPOINT;
  GenericApp_epDesc.task_id = &GenericApp_TaskID; //SampleApp 描述符的任务 ID
  GenericApp_epDesc.simpleDesc  = (SimpleDescriptionFormat_t*)&GenericApp_SimpleDes
c;
  //SampleApp 简单描述符
  GenericApp_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 );

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

```
#endif
  ZDO_RegisterForZDOMsg( GenericApp_TaskID, End_Device_Bind_rsp );
  ZDO_RegisterForZDOMsg( GenericApp_TaskID, Match_Desc_rsp );

#if defined( IAR_ARMCM3_LM )
  // Register this task with RTOS task initiator
  RTOS_RegisterApp( task_id, GENERICAPP_RTOS_MSG_EVT );
#endif
}
```

## 5. GenericApp_ProcessEvent()

前面分析的函数 osal_start_system()中，会初始化数组 tasksArr[]定义的所有数组，其中有一个最重要的函数 GenericApp_ProcessEvent()，该函数时用户应用任务的事件处理函数,在协调器、路由节点和终端节点中都有定义。

```
uint16 GenericApp_ProcessEvent( uint8 task_id, uint16 events )
{
 afIncomingMSGPacket_t *MSGpkt;
 (void)task_id;  // 有意未引用的参数
  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 GenericApp 的消息，以 SampleApp_TaskID 标记
  MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( GenericApp_TaskID);
    while ( MSGpkt )
    {
      switch ( MSGpkt->hdr.event )
      {
        case ZDO_CB_MSG:
            GenericApp_ProcessZDOMsgs( (zdoIncomingMsg_t *)MSGpkt );
          break;
      // 当按下密钥时收到
      case KEY_CHANGE: //按键事件
        SampleApp_HandleKeys( ((keyChange_t *)MSGpkt)->state,((keyChange_t*)MSGpkt)->
keys );
        break;  // 在接收到此端点的消息(OTA)时收到
      case AF_INCOMING_MSG_CMD: //接收数据事件,调用函数 AF_DataRequest()接收数据
        GenericApp_MessageMSGCB( MSGpkt ); //调用回调函数对收到的数据进行处理
        break;// 每当设备在网络中更改状态时接收。
      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 ( (GenericApp_NwkState == DEV_ZB_COORD)
              || (GenericApp_NwkState == DEV_ROUTER)
              || (GenericApp_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 );
      }
      break;
     default:
      break;
    }
   // 释放内存 //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);
 }
 // 发送消息-此事件由计时器生成。
 // (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();
   // 在正常时间内再次发送消息的设置(+一点抖动)
  osal_start_timerEx(SampleApp_TaskID,SAMPLEAPP_SEND_PERIODIC_MSG_EVT,(SAMPLEAPP_SE
ND_PERIODIC_MSG_TIMEOUT + (osal_rand() & 0x00FF)) );
   // return unprocessed events 返回未处理的事件
   return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);
 }
 // 丢弃未知事件
 return 0;
}
```

其中需要用户特别关注函数 GenericApp_MessageMSGCB()，

## 6. GenericApp_MessageMSGCB()接收数据函数

协调器和终端节点接收网络中无线数据后最红都会调用到该函数， 对于用户来说，我们只需要把自己的数据处理逻辑填充到该函数中即可。
下面仍然以协调器为例：

```
//接收数据，参数为接收到的数据
void GenericApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
 uint16 flashTime;
  byte buf[3];
  switch ( pkt->clusterId ) //设备 ID
  {
    case GENERICAPP_CLUSTERID: //收到广播数据
     ........
      break;
    case GENERICAPP_CLUSTERID1:
    case GENERICAPP_CLUSTERID2:
    case GENERICAPP_CLUSTERID3:
    case GENERICAPP_CLUSTERID4:
    HalUARTWrite(0, pkt->cmd.Data,24);
    break;
  }
}
```

其中 pkt->clusterId 是从其他无线设备接收到的 ID，在发送函数 AF_DataRequest ()中填充，
可以自己定义。

```
#define  GENERICAPP_CLUSTERID              0
#define GENERICAPP_CLUSTERID1              1   //节点 1
#define GENERICAPP_CLUSTERID2              2  //节点 2
#define GENERICAPP_CLUSTERID3              3
#define GENERICAPP_CLUSTERID4              4
```

## 8. AF_DataRequest 发送函数

要向网络中发送数据，调用该函数即可。

```
afStatus_t AF_DataRequest( afAddrType_t *dstAddr, endPointDesc_t *srcEP,
                           uint16 cID, uint16 len, uint8 *buf, uint8 *transID,
                           uint8 options, uint8 radius )
```

| 变量 | 含义 |
| --- | --- |
| afAddrType_t *dstAddr | 发送目的地址＋端点地址和传送模式 |
| endPointDesc_t *srcEP | 源(答复或确认)终端的描述（比如操作系统中任务 ID 等）源 EP |
| uint16 cID | 被 Profile 指定的有效的集群号 |
| uint16 len | 发送数据长度 |
| uint8 *buf | 发送数据缓冲区 |
| uint8 *transID | 任务 ID 号 |
| uint8 options | 有效位掩码的发送选项 |
| uint8 radius | 传送跳数，通常设置为 AF_DEFAULT_RADIUS |

其中 cID 对应的就是前一节函数 GenericApp_MessageMSGCB()接收到的数据 `pkt->clusterId`。

## 9.如何给串口注册回调函数

作为协调器，需要通过串口从上位机读取数据，可以通过以下方法注册回调函数：

```
halUARTCfg_t uartConfig;
 uartConfig.configured = TRUE;
 uartConfig.baudRate = HAL_UART_BR_115200;
 uartConfig.flowControl = FALSE;
 uartConfig.flowControlThreshold = 1;
 uartConfig.rx.maxBufSize = 255;
 uartConfig.tx.maxBufSize = 255;
 uartConfig.idleTimeout = 1;
 uartConfig.intEnable = TRUE;
 uartConfig.callBackFunc = rxCB;
 HalUARTOpen (HAL_UART_PORT_0, &uartConfig);
```

结构体变量 uartConfig 描述了串口的信息：

```
波特率：115200
流控：无
收发缓冲区：255
回调函数:rxCB
```

如下代码，功能：

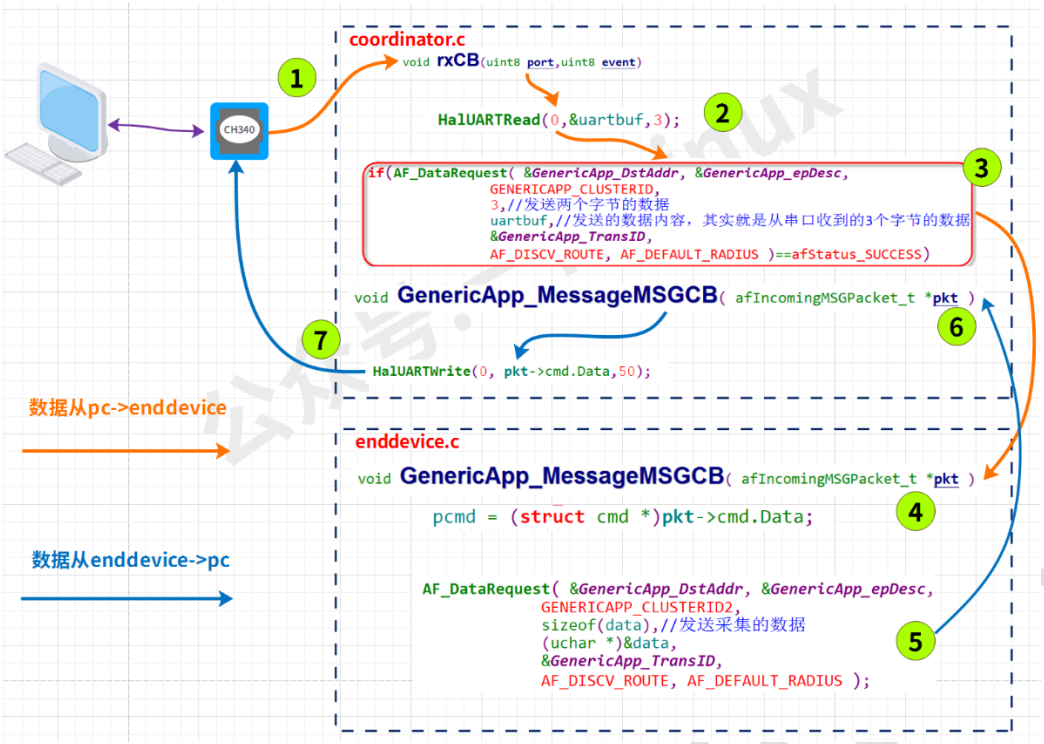
```
通过 UART 从上位机读取一个字节数据，并广播给其他无线节点的。

static void rxCB(uint8 port,uint8 event)
{
 uint8 uartbuf[4] = {0};
 uint8 cmd;
 HalUARTRead(0,&cmd,3);//从串口读取 3 个字节的数据到 cmd 中
 sprintf(uartbuf,"cmd:%x",cmd);
 //显示到 lcd 屏幕
 HalLcdWriteString(uartbuf,HAL_LCD_LINE_4);
 if(AF_DataRequest( &GenericApp_DstAddr, &GenericApp_epDesc,
                    GENERICAPP_CLUSTERID,
                    3,//发送两个字节的数据
                    &cmd,//发送的数据内容，其实就是从串口收到的 3 个字节的数据
                    &GenericApp_TransID,
                    AF_DISCV_ROUTE, AF_DEFAULT_RADIUS )==afStatus_SUCCESS)
 {
 }
}
```

初始化函数我们可以放到函数 GenericApp_Init()中初始化。

## 七、上位机与无线节点通信流程

综上：上位发送数据到无线节点，无线节点再上传数据给上位机流程如下：




1. 上位机通过串口软件发送数据给协调器；
2. 触发协调器串口回调函数 rxCB()，通过 HalUARTRead()从串口提取出数据；
3. 协调器通过 AF_DataRequest()将数据发送给网络中其他设备，通过 id:GENERICAPP_CLUSTERID，表明改数据是协调器发送；
4. 触发终端节点回调函数 GenericApp_MessageMSGCB()，协调器 id 存放在 pkt->clusterId 中，发送的数据存放在 pkt->cmd.data 中；
5. 终端节点可以采集传感器数据，发送数据在缓冲区 data 中，然后通过函数 AF_DataRequest()发送数据，填充 id:GENERICAPP_CLUSTERID2，表示数据是节点 2 发送；
6. 触发协调器的回调函数 GenericApp_MessageMSGCB()；
7. 协调器提取出终端节点发送的数据 pkt->cmd.data，通过 HalUARTWrite()发送给上位机。

其中协调器与终端节点的自动组网、数据收发均由协议栈完成。

## 八、如何添加定时器功能

我们希望终端节点可以定时上传传感器数据，这就需要定时器操作。

定时器启动函数

```
uint8 osal_start_timerEx( uint8 taskID, uint16 event_id, uint16 timeout_value )
taskID          : 启动定时器的任务 ID
```

```
event_id      : 需要通知的事件
timeout_value : 超时时间
```

本项目调用参数

```
osal_start_timerEx( GenericApp_TaskID,
                    GENERICAPP_SEND_MSG_EVT,
                    GENERICAPP_SEND_MSG_TIMEOUT );
```

思路如下：

1. ZDO 层更改网络状态后就启动定时器
2. 主任务处理函数 GenericApp_ProcessEvent()中，增加超时事件处理逻辑，上传传感器数据
3. 通过函数 osal_start_timerEx()重新启动定时器