# ZStack函数说明

# Zmain文件夹

Zmain.c文件包含了main主程序，需要注意的是osal\_init\_system()函数【初始化操作系统，包括了任务初始化】和osal\_start\_system()函数【执行操作系统，进入主循环】。

# OSAL文件夹

该文件夹中有许多文件，其中OSAL.c文件中包含了osal\_init\_system()函数和osal\_start\_system()函数。

# SampleApp文件夹

该文件夹主要包括OSAL\_SampleApp.c、SampleApp.c、SampleApp.h、SampleAppHw.c、 SampleAppHw.h。后两个函数主要通过读取跳线来确定设备是否为协调器，这里不涉及这个功能，所以不做介绍。

### 2.1 OSAL\_SampleApp.c文件

该文件主要包括了osalInitTasks()函数和taskArr[]数组。

#### osalInitTasks()函数

**调用路径：**

该函数由1. Zmain文件夹中的osal\_init\_system()函数调用，主要用于任务初始化。

路径：main.c -> osal\_init\_system() -> osalInitTasks().

**主要功能：**

主要功能是为操作系统的所有任务分配空间，并对各个任务进行初始化，包括MAC层、NWK层、HAL硬件抽象层(不属于Zigbee协议栈) 、APS层、和用户自定义层任务的初始化，其中用户自定义层任务初始化函数为SampleApp\_Init()函数。

#### taskArr[]数组

**调用路径：**

该数组在1.Zmain文件夹中的osal\_start\_system()函数使用，主要用于轮询调用任务处理函数。

路径：main.c -> osal\_start\_system() -> taskArr[]。

**主要功能：**

taskArr[]数组中存放了任务处理回调函数，在此数组中的事件处理函数与osalInitTasks()函数中的任务初始化函数是一一对应的(按顺序排列)。其中需要注意的是用户自定义的应用层任务处理函数SampleApp\_ProcessEvent()。

### SampleApp.c文件

该文件主要包括了SampleApp\_Init()函数、SampleApp\_ProcessEvent()函数、SampleApp\_HandleKeys()函数、SampleApp\_MessageMSGCB()函数、SampleApp\_SendPeriodicMessage()函数和SampleApp\_SendFlashMessage()函数。

#### SampleApp\_Init()函数

**调用路径：**

该函数由osalInitTasks()函数调用。

路径：main.c -> osal\_init\_system() -> osalInitTasks() –> SampleApp\_Init()。

**主要功能：**

该函数主要是实现用户自定义任务的初始化，包括任务ID、网络状态、传输序列号、数据发送地址、端点等信息初始化。包括广播、组播和单播的设置。

#### SampleApp\_ProcessEvent()函数

**调用路径：**

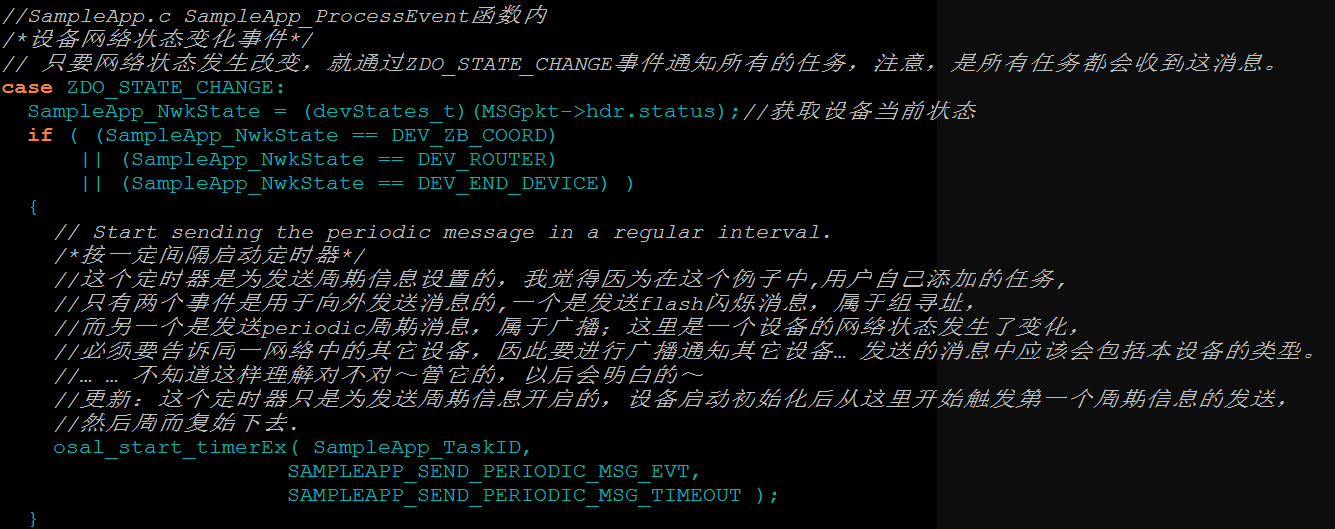
该函数由taskArr[]数组中定义，由osal\_start\_system()函数在轮询任务中调用。

路径：main.c -> osal\_start\_system() -> SampleApp\_ProcessEvent()。

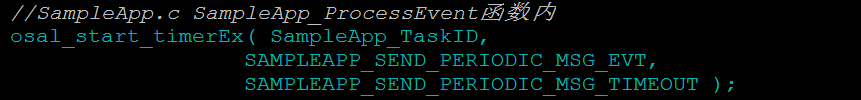
**主要功能：**

实现具体事件处理，包括接收数据和发送数据。在接收数据上，主要是接收系统消息事件，包括按键事件、接收消息事件、网络状态改变事件和定时事件等。按键处理事件主要调用SampleApp\_HandleKeys()函数，接收消息事件主要调用SampleApp\_MessageMSGCB()函数，网络状态改变事件中需要定时广播告诉所有的设备网络状态的变化。在发送数据上,主要是周期性发送数据，利用SampleApp\_SendPeriodicMessage()函数周期性发送数据，可以使传感器定时采集和上传数据。**周期性发送的数据的设备和时间在网络状态改变事件中处理。**

**注：周期性发送的数据的设备和时间在网络状态改变事件中处理。**

****

**If条件中可以定义需要定时发送数据的设备类型，例如协调器、路由器和终端设备。一般来说，协调器是不需要定时发送数据的，往往终端设备是需要定时采集传感数据进行发送给协调器。这里可以在if语句中自定义需要周期发送的设备类型。**

****

**这个就是设置定时发送的函数，第一个是自定义任务的ID，第二个就是周期性发送数据的事件编号，查看定义是0x01,而第三个参数就是设置周期性发送数据的时间间隔，这里是5000，也就是5s定期发送数据。那么发送数据在哪里执行呢？**

****

**定时发送数据时，先比对事件编号，然后利用数据发送函数SampleApp\_SendPeriodicMessage()函数发送数据，需要注意的是发送完以后发送完以后又开启了一个5s发送数据的周期性函数。这里可以做的工作就是周期性发送传感器采集的数据。**

#### SampleApp\_HandleKeys()函数

主要用于按键事件处理。

#### SampleApp\_MessageMSGCB()函数

**调用路径：**

该函数由SampleApp\_ProcessEvent()函数调用。

路径：main.c -> osal\_start\_system() -> SampleApp\_ProcessEvent() –> SampleApp\_MessageMSGCB()。

**主要功能：**

用于实现数据的接收，通过判断簇ID来执行广播或组播的接收信息。需要注意的是afIncomingMSGPacket\_t类型的结构体和其中的afMSGCommandFormat\_t类型结构体。这在6.3数据的接收中讲解。

#### SampleApp\_SendPeriodicMessage()函数

**调用路径：**

该函数SampleApp\_ProcessEvent()函数调用时用作周期性发送数据，当然也可以在其他地方灵活调用，例如接收数据SampleApp\_MessageMSGCB()函数用于发送反馈数据等。

路径：main.c -> osal\_start\_system() -> SampleApp\_ProcessEvent() –> SampleApp\_SendPeriodicMessage()。

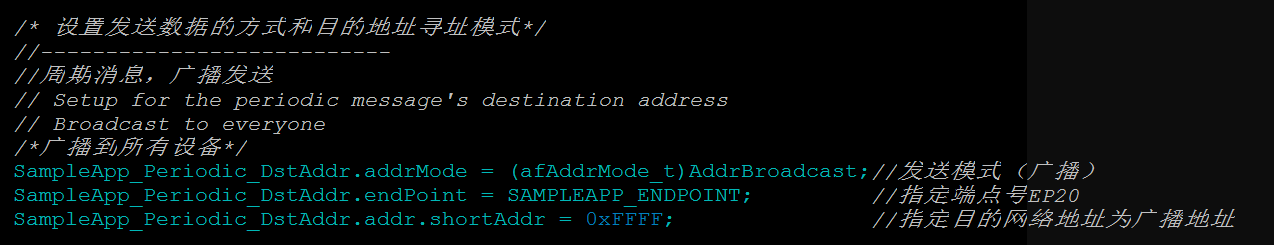
main.c -> osal\_start\_system() -> SampleApp\_MessageMSGCB() –> SampleApp\_SendPeriodicMessage()。

还可以有其他调用路径。

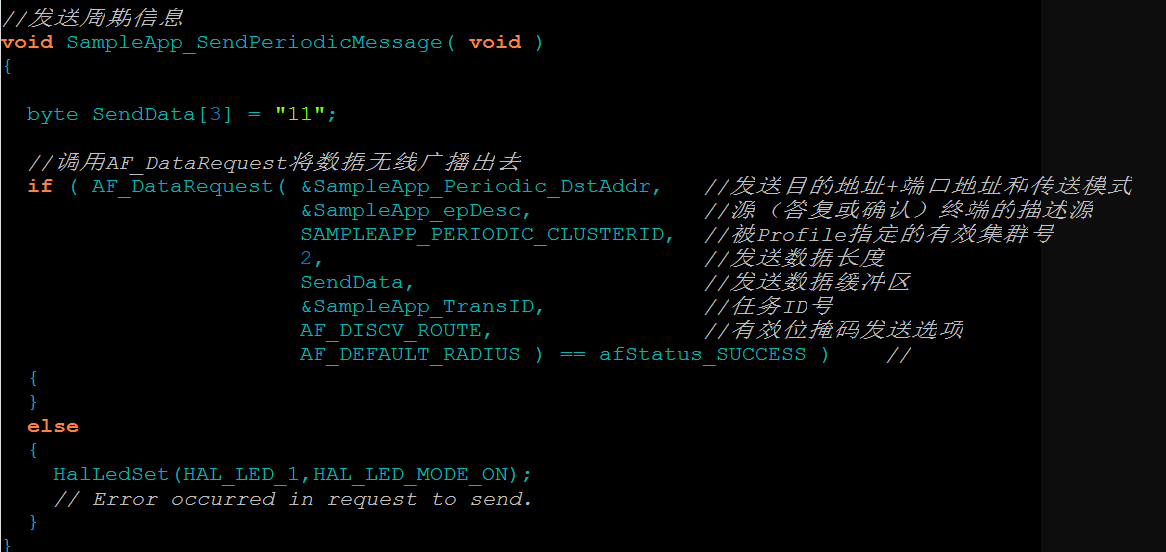
**主要功能：**

可以用于周期性发送数据或者单次发送数据。需要注意的是，这个函数用于广播发送数据，发送的数据任何设备都可以接收到。

注：查看SampleApp\_Init()函数中的定义



再看SampleApp\_SendPeriodicMessage()函数



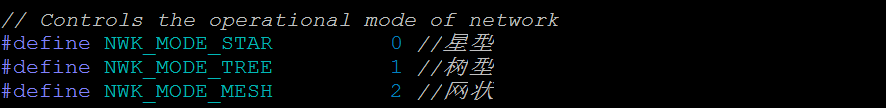
AF\_DataRequest的第一个参数就是广播地址参数。第三个参数就是在接收函数SampleApp\_MessageMSGCB()用来判断数据是广播的还是组播的等等。

#### SampleApp\_SendFlashMessage()函数

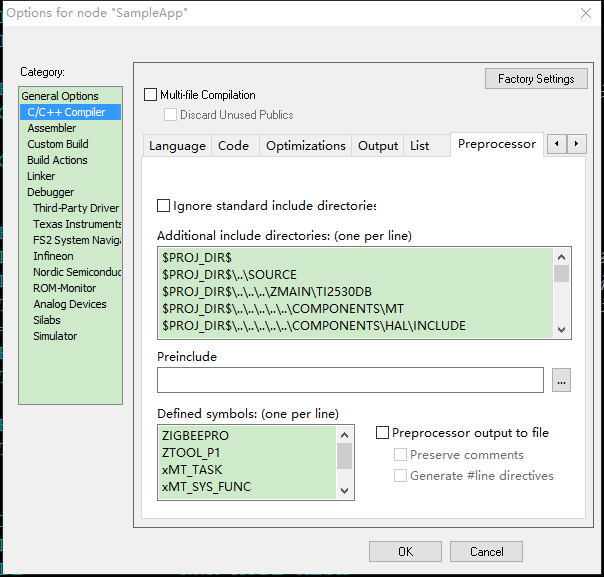
该函数只要用于组播发送数据。

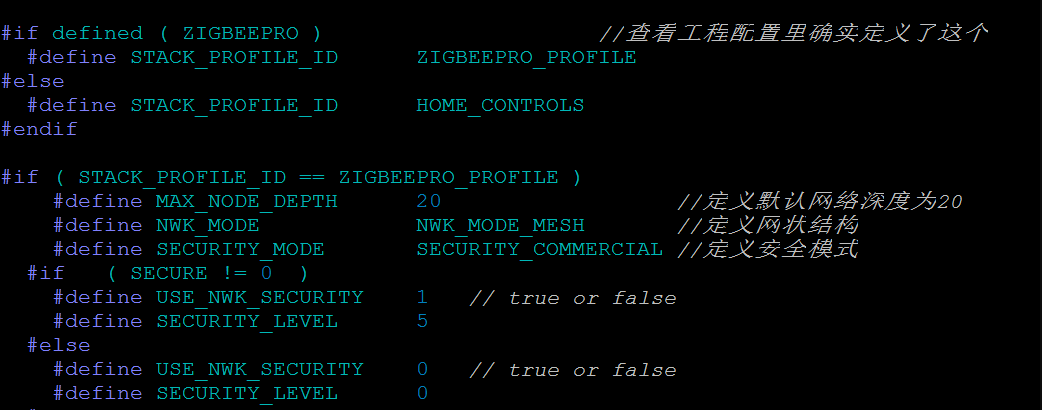
# NWK文件夹(这个如果要配置比较难，暂时不深入研究)

该文件夹中的nwk\_globals.h定义了网络拓扑结构类型

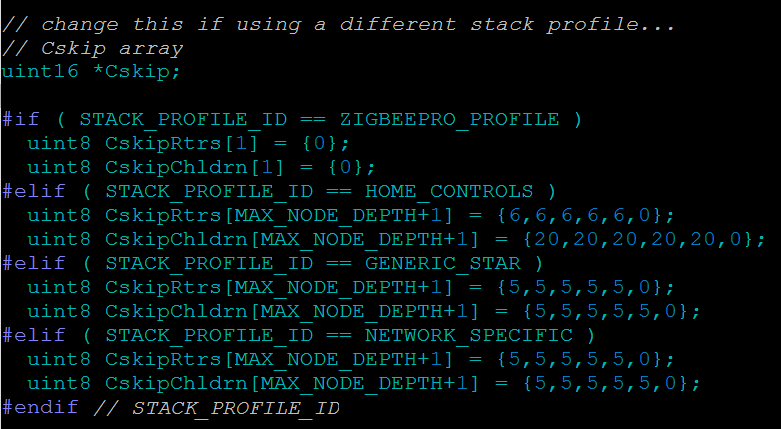


查看配置项宏定义了ZIGBEEPRO



  
这里如何把网络类型设置为星型结构？需要深入研究。

节点地址的分配和深度解析，看nwk\_globals.c



CskipChldrn[]中的每一个元素表示协调器或路由器可以连接的子节点的最大个数，这里还不是很清楚CskipChldrn[1] = {0}为什么是这样？原因是这里默认设置为网状网络结构，该结构的网络地址由协调器随机分配，所以设置默认值即可。但是在树状网络中，就不能这样了，要按照一定的算法分配地址。

# HAL文件夹

主要包含三个文件夹，Common、Include和Target文件夹。

### Common文件夹

Common文件夹中主要的文件是hal\_drivers.c。

#### hal\_drivers.c文件

该文件主要包括硬件初始化Hal\_Init()函数、硬件抽象层驱动初始化HalDriverInit()函数和硬件抽象层事件处理函数Hal\_ProcessEvent()。

Hal\_Init()函数：为硬件抽象层注册任务ID。在osalInitTasks()函数中被调用。

HalDriverInit()函数: 包括了定时器、ADC、DMA、Flash、AES、LED、UART、KEY和SPI等的初始化。在main函数中被调用。

Hal\_ProcessEvent()函数: 任务处理函数，当然是在taskArr[]数组中被定义。主要实现硬件抽象层的各种事件处理，比如系统消息事件、LED闪烁事件、按键事件和睡眠模式事件等。可以注意比较该文件中的程序和SampleApp\_ProcessEvent()事件处理函数的关系。

### Include文件夹

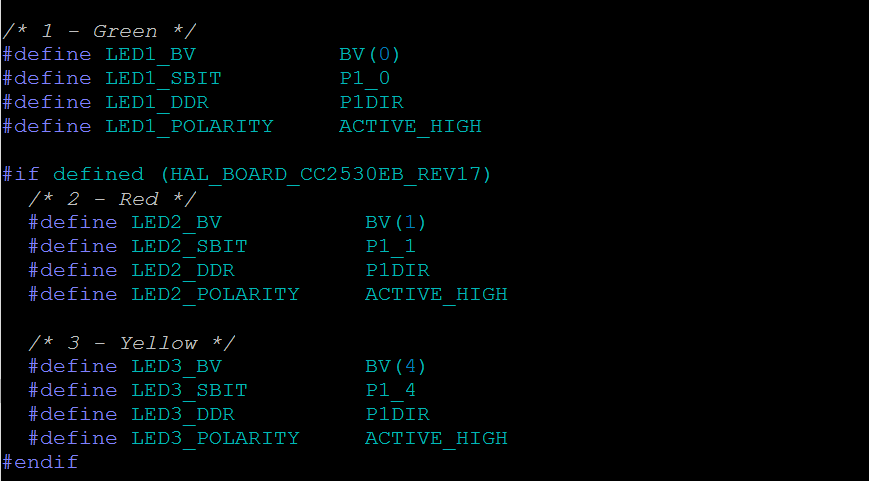
### Target文件夹

该文件夹主要包括CC2530EB文件夹（Config、Drivers、Includes）。

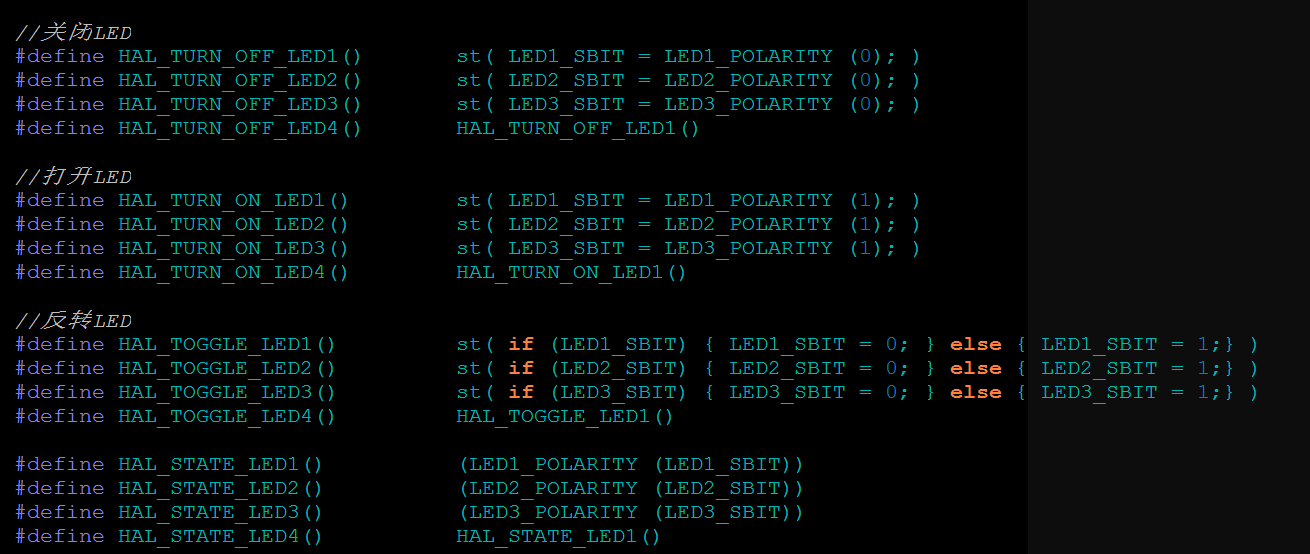
#### Config文件夹

该文件夹中有一个hal\_board\_cfg.h文件。

hal\_board\_cfg.h：为硬件资源LED等进行配置，在官方的协议栈中定义了3个LED，分别是P1\_0，P1\_1和P1\_4引脚。

****

对LED等进行控制

****

#### Drivers文件夹

定义了硬件资源的驱动函数文件，包括LED、ADC、KEY、定时器、串口、DMA以及Flash等。

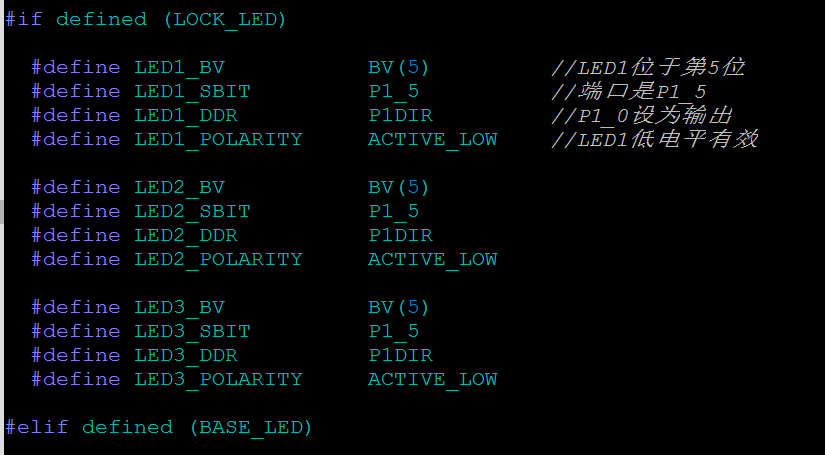
### LED驱动修改

#### 门锁LED驱动修改

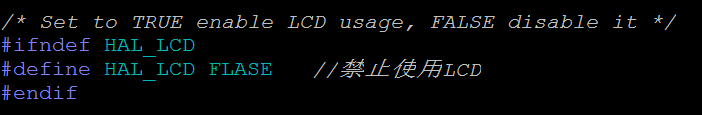
详细的查看Zigbee技术于实训教程-基于CC2530的无线传感网技术 P191

在预编译选项中加入LOCK\_LED，更改hal\_board\_cfg.h中的LED宏定义

需要注意的是Z-Stack默认有3个LED，门锁LED只有一个，为了方便起见，设置为3个一样。



需要注意可能改配置与其他的驱动端口有冲突，可以屏蔽其他的驱动端口，仍然在hal\_board\_cfg.h中



可以禁止LCD的端口。

### 按键驱动修改

按键采用两种方式采集按键数据：轮询和中断方式！Z-Stack默认使用轮询方式。CC2530定义了P0\_1独立按键和P2\_0摇杆按键（摇杆按键无法使用中断方式，采用AD值）。

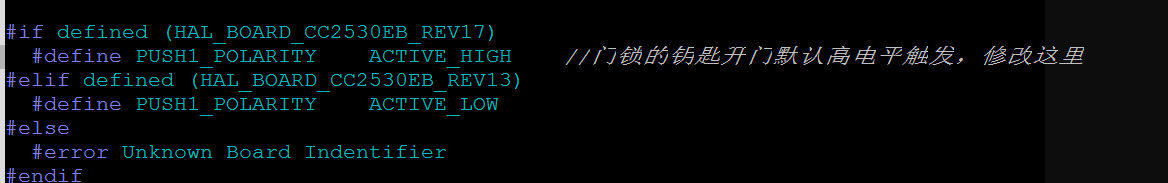
按键的消息应用层触发原理查看<http://bbs.eeworld.com.cn/thread-455980-1-1.html>

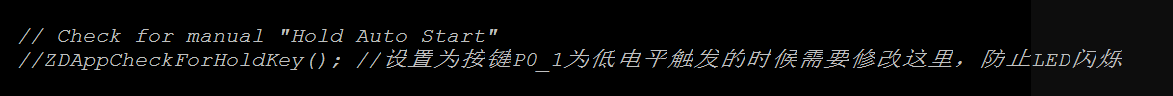
<http://www.cnblogs.com/perfect2014/p/4122706.html>（这个也不错）

最详细的查看Zigbee技术于实训教程-基于CC2530的无线传感网技术 P195

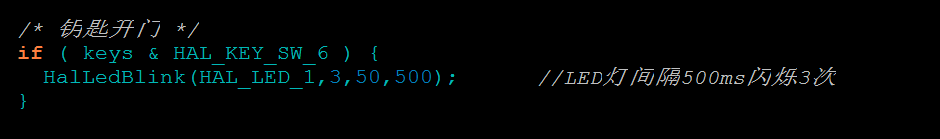
#### 按键轮询方式

查看hal\_board\_cfg.h，修改



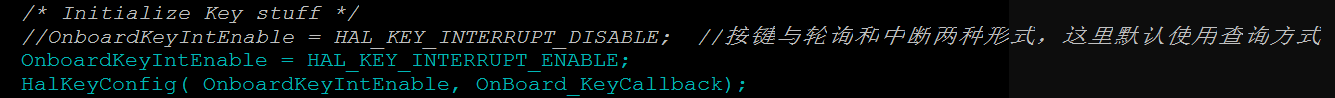
另外需要注意的是，要屏蔽ZDAPP.c中的Hold Auto Start状态的闪烁LED代码（判断按键是否按下,按键按下则进入Hold Auto Start状态）  


修改应用层按键事件的函数



#### 按键中断方式

只需要修改OnBoard.c即可中的InitBoard函数即可



Z-Stack分配了独立按键和摇杆按键，门锁不需要摇杆按键，这里去掉摇杆按键！

# ZStack应用

# 数据传输

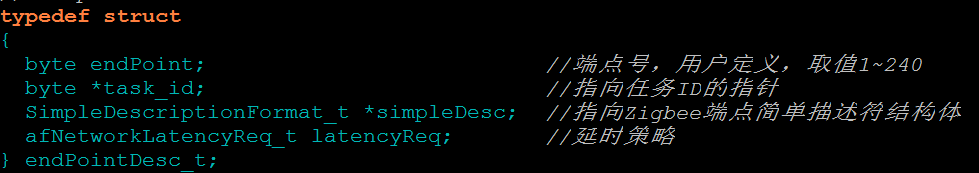
### 端点

在Z-Stack中最多可以实现256个端点，端点0被ZDO使用，端点255被用于广播，端点1~240倍应用层分配，用于数据的发送接收以及绑定。端点241~254为预留扩展端点。

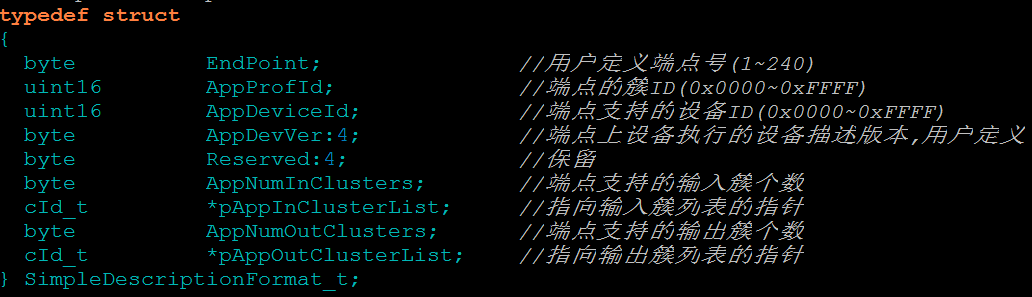
端点1~240的作用：在数据发送和接收时使用了端点的概念，在数据发送方和接收方使用的端点必须一致，否则数据发送不成功。在节点绑定过程中，必须注册一个或多个端点来进行绑定表的建立。

端点的意思就是在一个设备中，最多有240个应用对象！相同端点的应用对象可以互相通信，不同端点的应用对象不能相互通信。

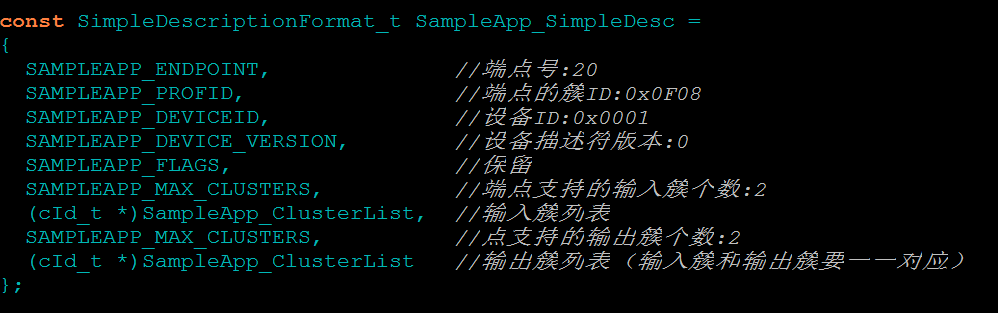
端点的实现由端点描述符来实现，查看AF.h



Zigbee端点简单描述符(AF.h)

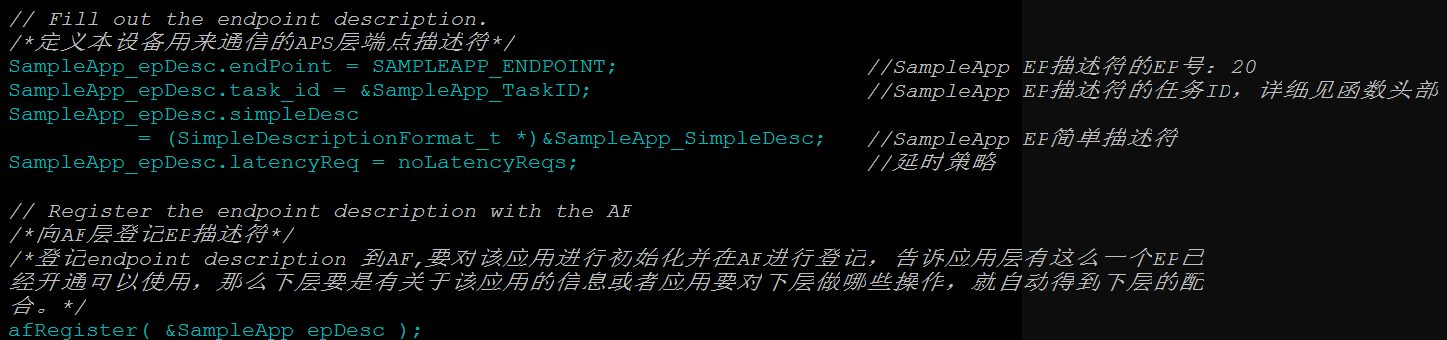


Ti官方的SampleApp.c中的默认Zigbee端点简单描述符



需要注意的是输入簇和输出簇要一一对应，即发送方的输出簇要和接收方的输入簇一致。在TI官方的SampleApp中发送端和接收端的输入/输出簇列表定义如下(SampleApp.c)：  


端点定义了以后需要在Z-Stack应用层注册端点，端点注册了以后才能使用Zigbee协议栈进行数据的发送和接收，详见SampleApp\_Init()函数



### 数据的发送

#### AF\_DataRequest()发送函数

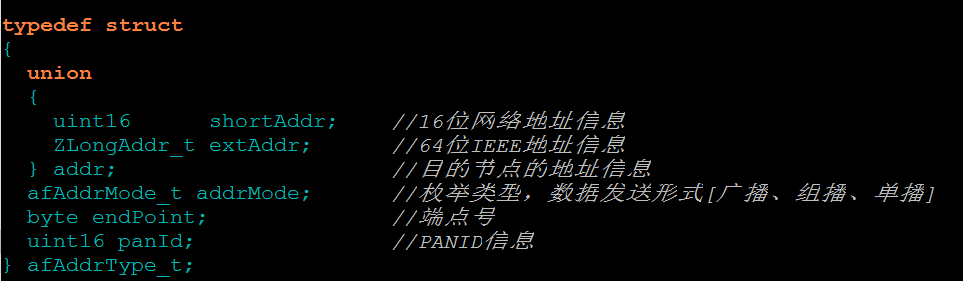
数据的发送函数详见SampleApp.c中的SampleApp\_SendPeriodicMessage()函数和SampleApp\_SendFlashMessage()函数。这两个函数都调用了AF\_DataRequest()函数进行发送。例如广播发送函数



查看第一个参数目的节点的地址信息的定义(SampleApp.c)

afAddrType\_t SampleApp\_Periodic\_DstAddr; //这是广播的目的节点

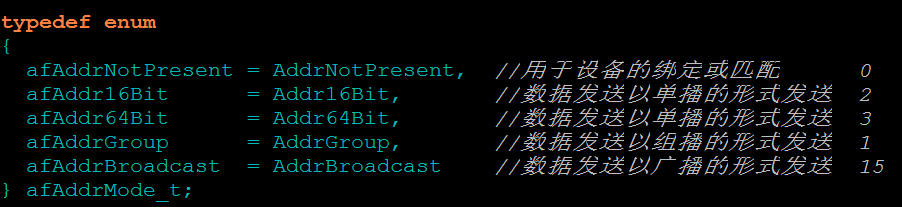
afAddrType\_t结构体如下



这里说明一下，使用64位物理地址和16位网络地址都可以发送数据，[64位](https://www.baidu.com/s?wd=64%E4%BD%8D&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH0LPWu-nj61nWKBujRs0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErH6Yn10kP1nzP1nsPW01rjb4)的地址只能用于单跳寻址，路由的寻址只能用[短地址](https://www.baidu.com/s?wd=%E7%9F%AD%E5%9C%B0%E5%9D%80&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuH0LPWu-nj61nWKBujRs0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErH6Yn10kP1nzP1nsPW01rjb4)。

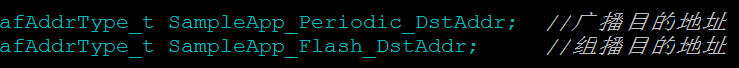
但是使用16位网络地址可以采用路由算法降低功耗，当然对于没有路由的设备是无所谓的，但是也最好使用16位网络地址，发送数据少。其次如果是需要协调器对某个终端设备点对点通信？

查看afAddrMode\_t枚举类型



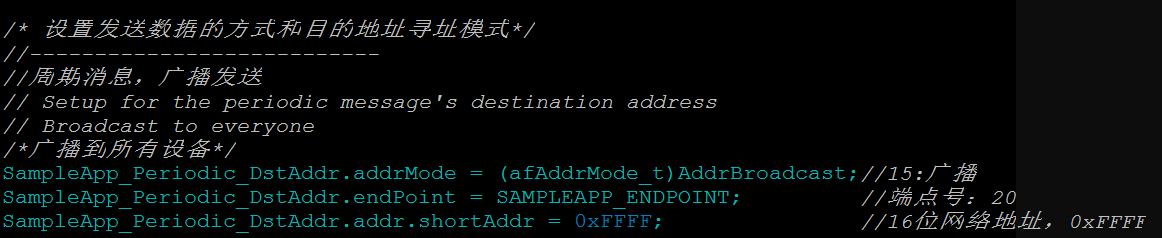
#### 地址信息的设置

6.2.1详细介绍了afAddrType\_t结构体，在SampleApp.c中定义了两个目的地址



##### 广播地址的设置(SampleApp.c SampleInit())

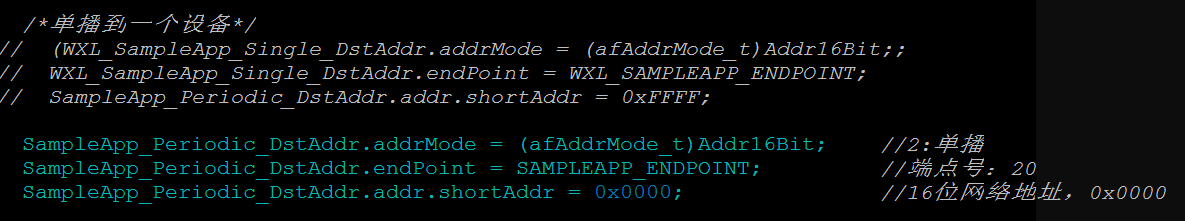
通过变量afAddrType\_t SampleApp\_Periodic\_DstAddr进行设置

  
数据广播的地址afAddrType\_t中的addr取值有4种形式

|  |  |
| --- | --- |
| 地址 | 说明 |
| 0xFFFF | 广播模式地址的默认值，广播至网络中所有设备，对于睡眠的设备，数据包将保留在父节点，当节点被激活后主动到此节点的父节点查询此数据包是否丢失。 |
| 0xFFFD | 传送至网络中所有打开接收功能的空闲节点。 |
| 0xFFFC | 发送至网络中的所有路由器，包括协调器。 |
| 0xFFFE | 应用层不指定目标设备，通过协议栈读取绑定表来获得目标设备的短地址。 |

##### 6.2.2.2 单播地址的设置(SampleApp.c SampleInit())

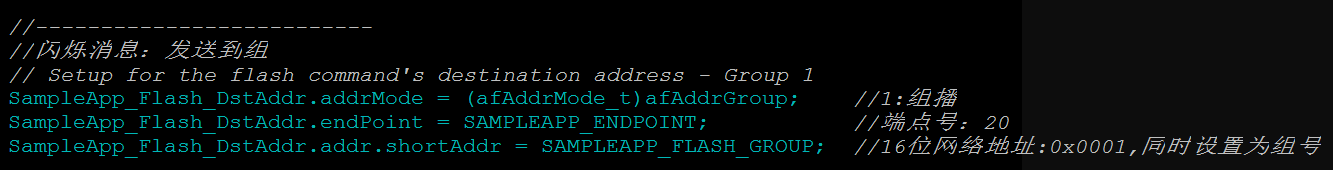
单播和广播一样，通过SampleApp\_Periodic\_DstAddr(也可以重新开辟一个变量，这里方便起见使用了组播的变量进行设置)进行设置。



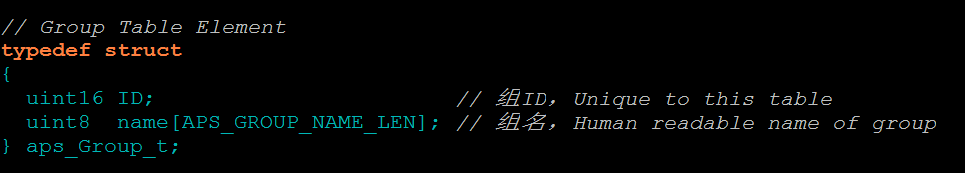
设备的目的地址设置为16位固定的网络地址，例如向协调器发送信息，协调器的网络地址为固定值0x0000,因此单播的地址就是0x0000(除了协调器的地址是固定的，其他设备的网络地址都是随机分配的，所以只能通过函数获取，有专门的获取地址函数，发送时协议会发送网络地址？然后才能根据网络地址反馈？)。还有就是多个协调器的情况下，如何组网？？？分布在不同的楼层？？？信道设置不一样？？？

##### 6.2.2.3 组播地址的设置(SampleApp.c SampleInit())

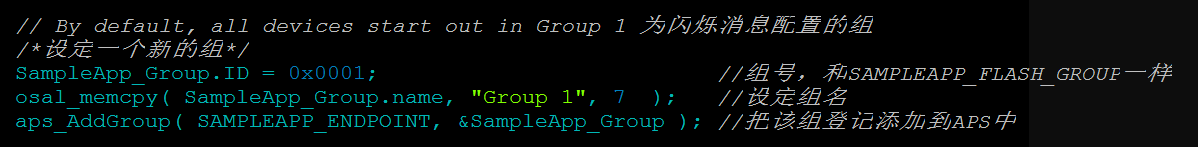
通过变量afAddrType\_t SampleApp\_Flash\_DstAddr进行设置



数组的组播需要设置组ID,组名，在aps\_groups.h文件中定义结构体



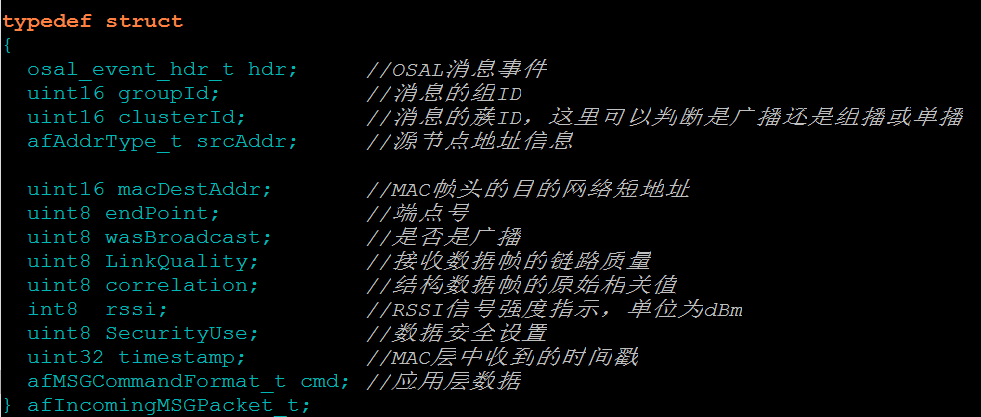
组ID的设置范围为0x0000~0xFFFF，设置完成后需要将设置信息注册到应用支持子层APS中，具体设置在SampleApp.c的SampleApp\_Init()中



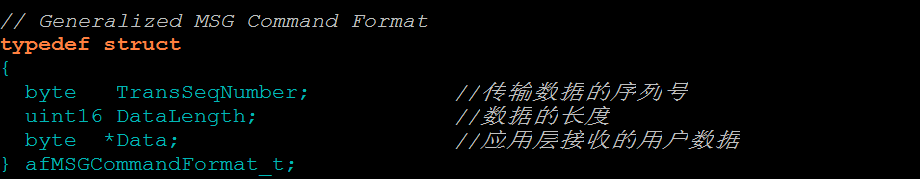
### 数据的接收

查看3.2.2 SampleApp\_ProcessEvent()函数，定义的第一个变量是一个接收消息的结构体afIncomingMSGPacket\_t \*MSGpkt; //接收到的消息，消息包

查看这个结构体(AF.h)



afAddrType\_t这个结构体可以查看6.2.1发送函数中的介绍，可以知道发送的时候发送了目的节点的地址信息，接收的时候同样可以知道源节点的地址信息。查看最后一个应用层数据结构体 afMSGCommandFormat\_t cmd; //应用层数据，仍然在AF.h



在应用层数据的接收通过osal\_msg\_receive()函数(OSAL.c)实现，数据接收完成之后在3.2.4节接收函数SampleApp\_MessageMSGCB()中进行处理，主要有两步

1. 判断事件类型，包括按键事件、AF层接收消息事件（消息到来响应事件）、网络状态改变事件等。
2. 如果接收到的消息是某一个事件类型，将调用相应事件的处理函数，如果是AF层接收消息事件，还会调用SampleApp\_MessageMSGCB()函数执行，并判断数据的簇ID是广播、组播还是单播。

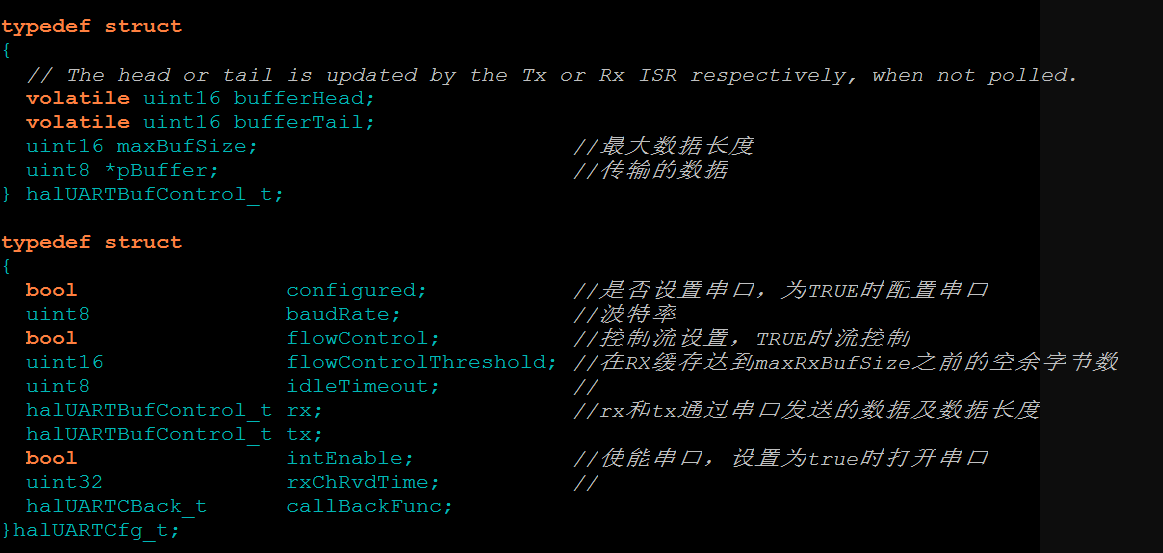
# 串口应用

串口的主要作用当然是给PC发送数据，所以一般来说是协调器会用到的功能。

串口的相关配置在HAL层的hal\_uart.h和hal\_uart.c中。具体路径

HAL -> Target -> CC2530EB -> Drivers -> hal\_uart.c

在hal\_usrt\_h中查看结构体

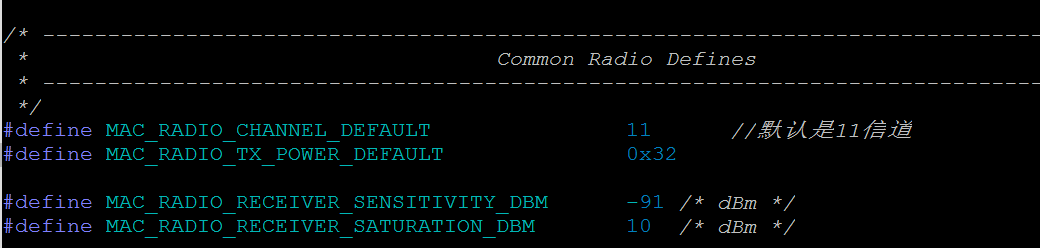


具体的应用查看MT层MT\_USRT.c函数。

# 绑定和匹配（暂时不研究）

# 信道的设定

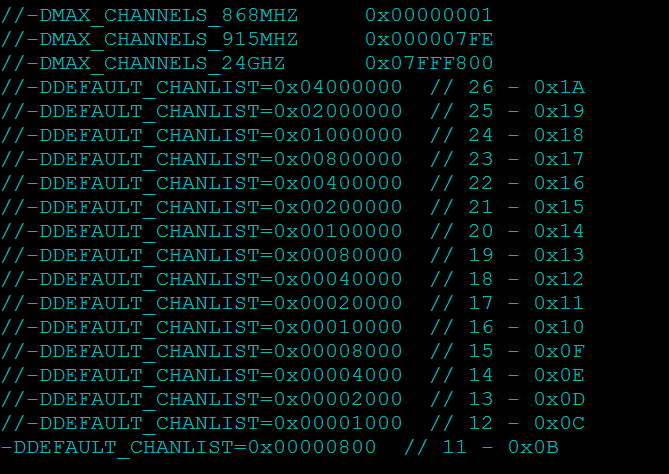
查看 MAC -> Low Level -> mac\_radio\_defs.h



也可以通过设置信道设置

以下此方法设置成功！

更具体的设置可以查看Zigbee技术与实训教程 P186. 查看Tools -> f8Config.cfg文件 注释11开启25，避免与WI-FI冲突



# 协调器网络的建立

查看Zigbee2007/PRO协议栈实验与实践 P92/94

这里还说明了协调器的16位网络地址是0x0000.

### 网络的形成

查看ZDApp\_event\_loop任务下的ZDO\_StartDevice()函数，对于协调器创建网络，对于终端和路由器加入网络。该工作是通过操作系统管理完成的，我们不需要考虑。

# 设置操作系统的事件

触发事件有三种情况：1.外部中断 2.定时器 3.对设备进行轮询

外部中断和定时器不需要进行干预，但是例如串口这样的设备，需要使用Hal\_ProcessPoll()进行轮询。

Zigbee技术实训教程 P183

Osal\_set\_event()设置任务事件的函数

例如中断的触发事件

查看Zigbee2007/PRO协议栈实验与实践 P219/216

查看Zigbee技术原理与实战 P234对任务和事件的理解

一个任务在处理过程中是不可以被中断的，必须前一个任务被处理完毕后才能够执行任务切换，每个任务都只有调用osal\_set\_event()函数后才能够得到处理。

### 操作系统流程图.

查看Zigbee2007/PRO协议栈实验与实践 P254

### Z-Stack执行流程

查看Zigbee技术原理与实战P214

### Z-Stack重要术语关系

查看Zigbee技术原理与实战P219 包括节点、端点、设备以及群集等之间的关系，这里可以结合ZigBee无线网络与收发器P85。并说明了端点和TCP/IP的端口的相似性。

### Z-Stack获取父节点地址和自身地址

查看Zigbee技术原理与实战 P230

### Packet Sniffer入网过程分析

查看Zigbee技术原理与实战 P253

#### 协调器上电创建网络



协调器创建网络的过程

在15信道首先发送了一条信标请求帧，用来判断周围的环境情况，为创建网络做准备。

当协调器创建网络成功后，就会发送一个网络命令帧，该帧数据每隔15S发送一次，可以在nwk\_globals.h中配置，直到有终端节点加入，否则一直轮循15S发送一次。其中广播地址为0xFFFF，协调器的网络地址固定为0x0000，PAN ID可以查看Tools -> f8wConfig.cfg中的配置（和信道配置在同一个配置文件中）,可以看出Z-Stack中默认的PAN ID为0xFFF1，路由器和重点设备可以加入网络用这个PAN ID，当然如果要建立不同的PAN网络，可以修改这个配置。

/\* Define the default PAN ID.

\*

\* Setting this to a value other than 0xFFFF causes

\* ZDO\_COORD to use this value as its PAN ID and

\* Routers and end devices to join PAN with this ID

\*/

-DZDAPP\_CONFIG\_PAN\_ID=0xFFF1

在数据帧中的网络建立状态信息(NWK Link Status)中可以看出，没有路由器或者终端设备连接时Count为0，且List\_status\_list也为空。

#### 终端设备上电寻求网络



同样是15信道，终端节点上电后，首先向周围环境执行信道能量扫描，选择一个能量比较合适的信道开始搜寻网络，在Z-Stack里，信道能量是有一个阈值的，当信道能量低于这个门限会被认为不存在网络，信道选择以后，终端设备会周期性的向周围发送信标请求数据包(Beacon request)来搜寻协调器的回复。

router / coordinator 不存在时，或者因为信号强度过低而链路断开时，从 sniffer 嗅探器里可观察到，end-device 频繁发送 [beacon](http://www.yidianzixun.com/m/channel/keyword/beacon?display=beacon&word_id=beacon&type=token) 导致传感节点的电池电量（往往传感节点都是电池供电）被消耗殆尽。何解？

聪明的你一定知道 f8wConfig.cfg 文件的存在。这其中，涵盖了一众 Zigbee 协议栈的配置信息。

# file: \Projects\zstack\Tools\CC2530DB\f8wConfig.cfg

-DBEACON\_REQUEST\_DELAY=3000 # from default 100 –> 3000

-DBEACON\_REQ\_DELAY\_MASK=0x0FFF # from default 0x00FF –> 0x0FFF

其中 –D 前缀表示预处理器的 #define 宏定义。而上述两个宏定义，前者，表示 beacon request 之间的延时 delay，后者，表示延时的掩码 mask（引入随机性），即真正的延时是 delay + rand & mask。

可以通过简单的计算得知，起初 delay 默认是 100 毫秒，mask 默认是 0x00FF，延时的上下限分别是 [100, 355]；修改成 3000 毫秒和 0x0FFF 掩码后，上下限则分别变成了 [3000, 7095]。取平均值 228ms 和 5048ms，可见平均延时增加了 20 倍左右。如果一直处于 beacon request 搜网过程里，原本能持续搜索 5 天的电池容量，一下子可以持续支撑 3 个月。很简单，对吧！

在只有一群 end-device 和一个 coordinator 的稳定运行网络里，更换 coordinator 后，节点无法再次入网。如何破？

根据假设，网络里只有一群 end-device 和 coordinator，并且已经稳定运行ing。突然你的小猫小狗跑过来，把协调器从桌上扔到了地上，不幸翘腿，以至于你不得不更换协调器。问题来了：节点依然在发送 beacon request，新的协调器也一直在回应 superframe（dev.Cap 也是 1，即还有剩余的 end-device capacity、允许节点入网），可节点就是没有 association request，更别提入网了…… 你感到很沮丧，把小猫小狗批评了一顿，尽管它们不懂你在说什么。

要解决这个问题，有两种全然不同的方法。

a) 第一种，简单粗暴，软件重启。

聪明的你肯定知道，sample application 应用的 SampleApp\_ProcessEvent 函数中，有一个 ZDO\_STATE\_CHANGE 的系统消息，是 ZDO (Zigbee Device Object) 层发来的消息称「哥状态有变，兄弟们请根据新状态自行作出处理」。

啥情况下 SampleApp\_ProcessEvent 会收到来自 ZDO 的 STATE\_CHANGE 消息呢？

在上述问题情况下，由于 end-device 不知 coordinator 已经翘腿，所以一旦传感器检测到某种信息，它依然傻兮兮的给协调器发过去；自然，它无法收到来自协调器的 MAC layer ACK 即确认响应（confirmation acknowledgement）；从 sniffer 上观察，你会发现收不到响应的 end-device 多次重传数据包。

多次重传数据包都失败（都没有收到 MAC layer ACK）后，end-device 就会认为已经同父节点（parent-node）失去了联系，遂敦促 ZDO 发送状态变更（先前是 DEV\_END\_DEVICE 状态，如今和父节点失联，成为了孤儿节点状态，即 DEV\_NWK\_ORPHAN）。

一旦 SampleApp\_ProcessEvent 收到状态变更为孤儿，软件重启开始新一轮搜网入网过程即可。

b) 第一种方法过于简单粗暴，以至于你无法完成一些更完善的操作，以应对更复杂的现实环境。

不妨再假设，你的协调器被小猫小狗扔到地上后，并没有翘腿，而仅仅是电池脱落（暂时休克）。正在厕所里拉屎的你，听到响声后，虽心有余，而力不足，无法立刻冲出厕所，拯救暂时休克的协调器，只能眼睁睁看着 end-device 给协调器发数据而收不到 MAC layer ACK、从而自认为变成了孤儿……

根据第一种方法，end-device 的传感器或许检测到了某个重要的信息，而一旦重启，除非你将数据写入了 NV 即 non-volatile 区域，否则，信息将丢失……

其实，end-device 只需要再等两分钟，等你走出厕所，给协调器安上电池后，它就可以直接通过 orphan notification（而不是 beacon request），以更少的空中交互次数、更低的能量消耗完成再次入网…… 如何做到这一点？

聪明的你肯定可以搜索到 devStartModes\_t 枚举类型，这个枚举状态，决定了节点入网的默认行为。

如，MODE\_RESUME 对应于孤儿节点状态，即试图通过 orphan notification 入网；而 MODE\_REJOIN 和 MODE\_JOIN 虽然都是发送 beacon request 入网，但 REJOIN 希望看到 superframe.extended\_PAN\_ID 字段（扩展 PAN ID），等于其变成孤儿状态之前的网络的 extended PAN ID。

节点同父节点失联后，MAC / NWK 层自然是最先得知此消息，为了尽快告知其它层的兄弟们，MAC / NWK 通过 ZDO 层的回调函数 ZDO\_SyncIndicationCB，告知说「咱们已经同父节点失联，哥们请负责通知其他兄弟」。此函数遂发送 ZDO\_NWK\_JOIN\_REQ 消息给 ZDApp，通知说「我们已经失联，请尽快处理重新入网事宜」。

接下来的事情，聪明的你应该可以通过阅读 Z-Stack 网络部分的源码自行搞定了。如果想要把 Zigbee primitive 即原语弄得很清楚，可以参考Zigbee 2007 Specification，戳这里可以参考下俺之前阅读源码时的一些摘要。

整体来讲，理解清楚 request confirmation indication 的含义，应该就可以比较顺利的理解网络层代码了。比如，假设现在是 MODE\_JOIN 入网模式，ZDO\_StartDevice 里的 NLME\_NetworkDiscoveryRequest 会请求「发送 beacon 获取周围的父节点们」，对于回应的 superframe(s) 数据帧，会通过 ZDO\_beaconNotifyIndCB 做处理，而 Network Discovery 的结果，则会通过 ZDO\_NetworkDiscoveryConfirmCB 来反馈，等等。这篇博客里，也在理论层面上，描述了整体的入网过程。

从上面的回答可以知道，如果终端设备断网，默认会以100ms向周围发送信标请求数据包(Beacon request)来搜寻协调器的回复。这样频繁的发送对于设备的电池消耗非常大，可以通过修改f8wConfig.cfg文件中的

/\* Minimum number of milliseconds to delay between each beacon request

\* in a joining cycle.默认100

\*/

-DBEACON\_REQUEST\_DELAY=100

/\* Mask for the random beacon request delay. This value is masked with

\* the return from osal\_rand() to get a random delay time for

\* each joining cycle. This random value is added to DBEACON\_REQUEST\_DELAY.

\* For example, a value of 0x00FF will be a beacon request delay of 0 to 255

\* milliseconds.默认0xFF

\*/

-DBEACON\_REQ\_DELAY\_MASK=0x00FF

来延长信标帧的发送，修改参数进行尝试

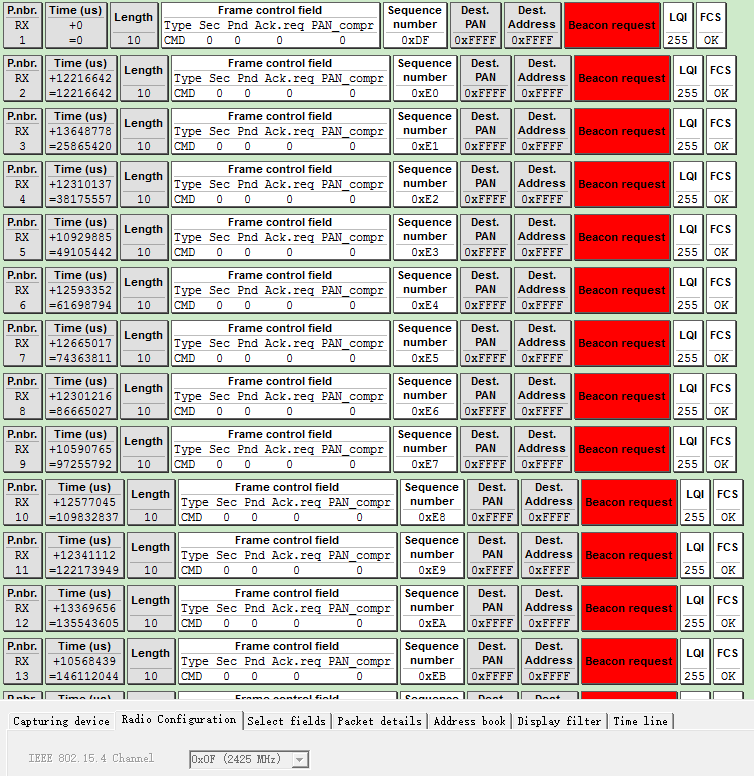


默认的信标请求的时间间隔大致为600~700ms发送一次。

-DBEACON\_REQUEST\_DELAY=10000

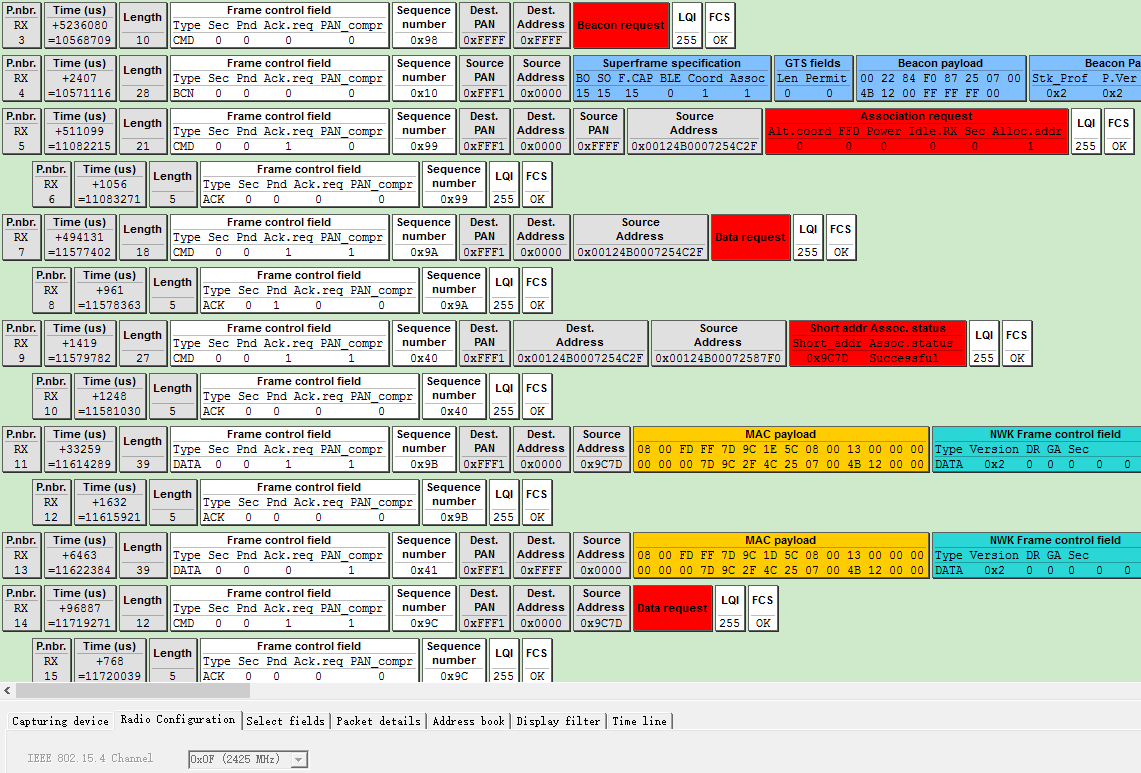
-DBEACON\_REQ\_DELAY\_MASK=0x0FFF

延时大致为10000ms~14095ms



明显从Time(us)一栏中可以发现信标帧的时间间隔大致为12s~14s左右。

#### 关联过程



1．信标请求帧

2．协调器响应一个包含IEEE 64位地址的超帧，主要目的是将MAC地址交给路由器，以便后续通信。

3. 终端节点接收到超帧之后，将协调器的IEEE地址保存起来，并利用这个地址向协调器发送一个关联请求(Association Request)的命令包，以寻求加入网络。

4. 收到协调器MAC层的关联请求确认帧（ACK）之后，终端节点紧接着发送数据请求（Data Request）命令包来获取协调器为其分配的16位网络短地址。

5. 协调器MAC层又发送了对于这个网络短地址请求帧的ACK表明收到了终端节点发过来的帧，并为终端节点分配一个网络范围内唯一的16位网络短地址（从图中可以看出分配的网络短地址为0x9C7D），然后发送包含该网络短地址的关联响应包。由于发送网络短地址之前协调器并不知道目标设备的网络地址，所以这个包是通过MAC地址发送的。终端节点接收到发送的网络地址之后将自己的网络地址配置为0x9C7D,就可以利用这个短地址和协调器进行网络层的通信了。此时，终端节点成功加入网络了。

6. 终端设备收到了协调器发送的网络地址数据帧时，MAC层自动回复ACK,表明收到了数据。

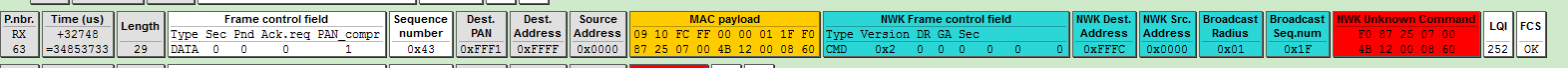
7. 这一帧数据是入网宣告，通过这个帧广播入网信息给网络里所有节点。

8. 入网宣告帧发送以后，协调器进行了转发。

9. 入网稳定后，终端会给协调器发送数据请求帧(Data request)。主要是告诉协调器网络连接正常。

10.协调器在收到终端发来的帧，硬件会自动回复的ACK。

11. 后面还会出现这一帧，这一帧数据就是15s发一次

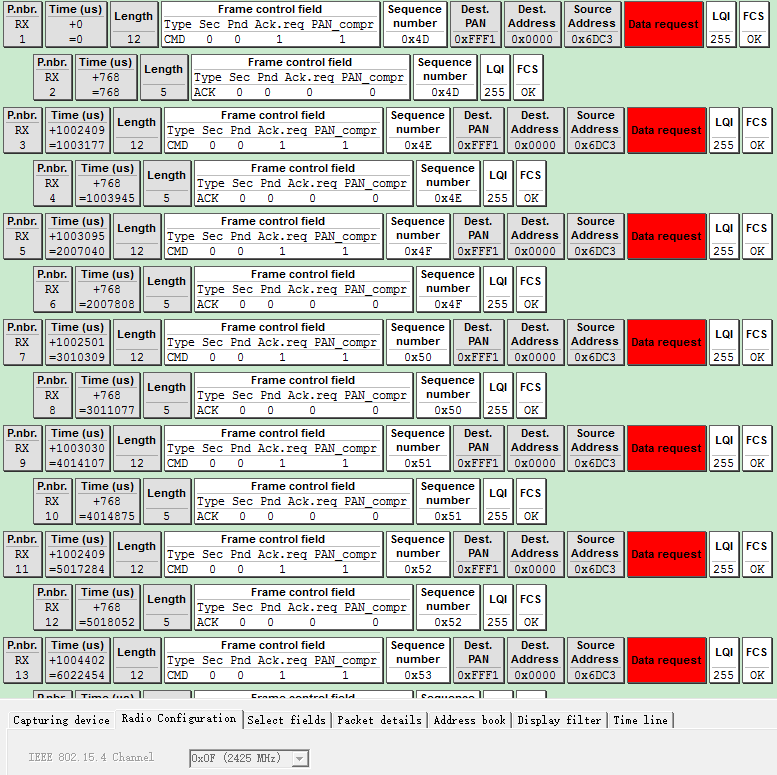


和最开始的协调器上电有点类似



只不过协调器后面的NWK Link Status变成了NWK Unkown Command（可能路由器能显示但是终端设备显示不了）。

#### 终端节点不停的请求数据过程



终端节点入网成功后，会不断的给父节点（协调器）发送请求数据帧，告诉父节点当前我处于非休眠状态，协调器在收到终端发来的帧，硬件会自动回复ACK。但是明显发送的周期大概在1s左右，这样不停的发送电池的耗电量迅速上升。

可以修改配置，具体可以查看12.1 休眠中的配置！

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* The following are for End Devices only

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

-DRFD\_RCVC\_ALWAYS\_ON=FALSE

/\* The number of milliseconds to wait between data request polls to the coordinator. \*/

-DPOLL\_RATE=1000 //修改这个参数就是Date Request的请求周期

/\* This is used after receiving a data indication to poll immediately

\* for queued messages...in milliseconds.

\*/

-DQUEUED\_POLL\_RATE=100

/\* This is used after receiving a data confirmation to poll immediately

\* for response messages...in milliseconds

\*/

-DRESPONSE\_POLL\_RATE=100

/\* This is used as an alternate response poll rate only for rejoin request.

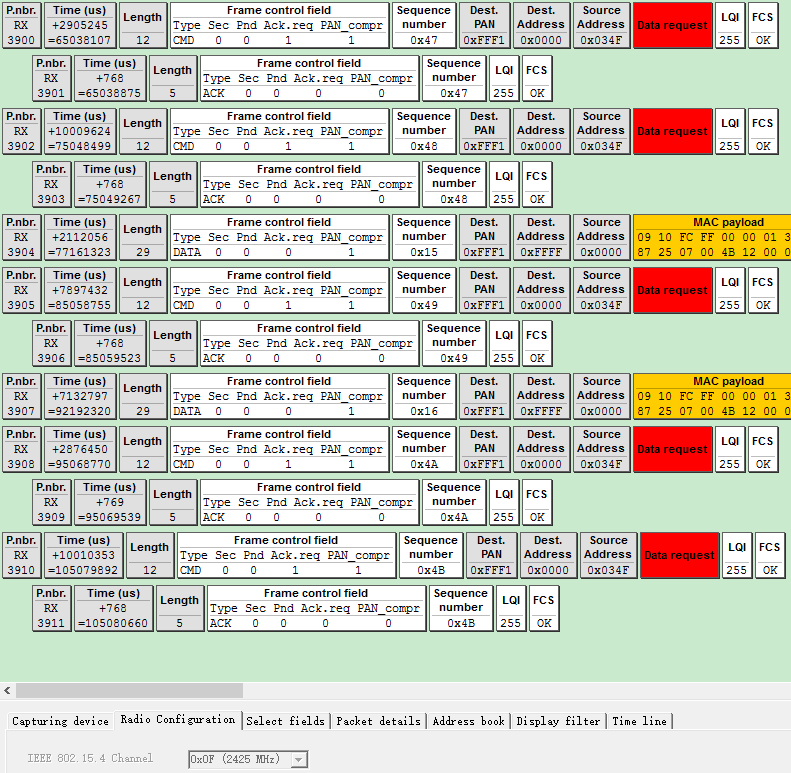
\* This rate is determined by the response time of the parent that the device

\* is trying to join.

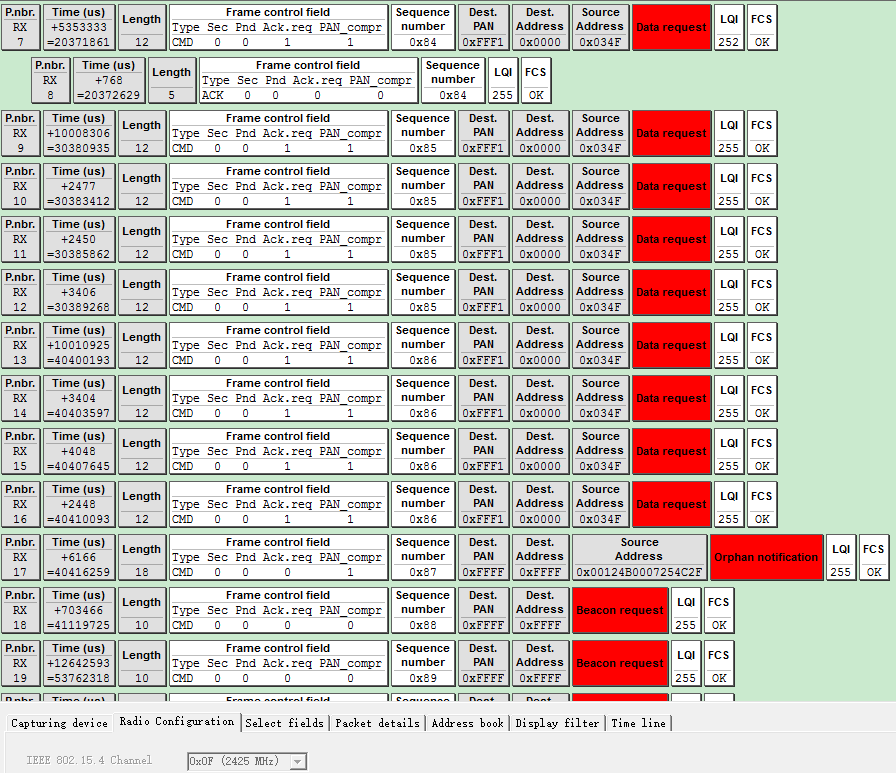
\*/

-DREJOIN\_POLL\_RATE=440

可以测试一下修改成10s，测试成功！

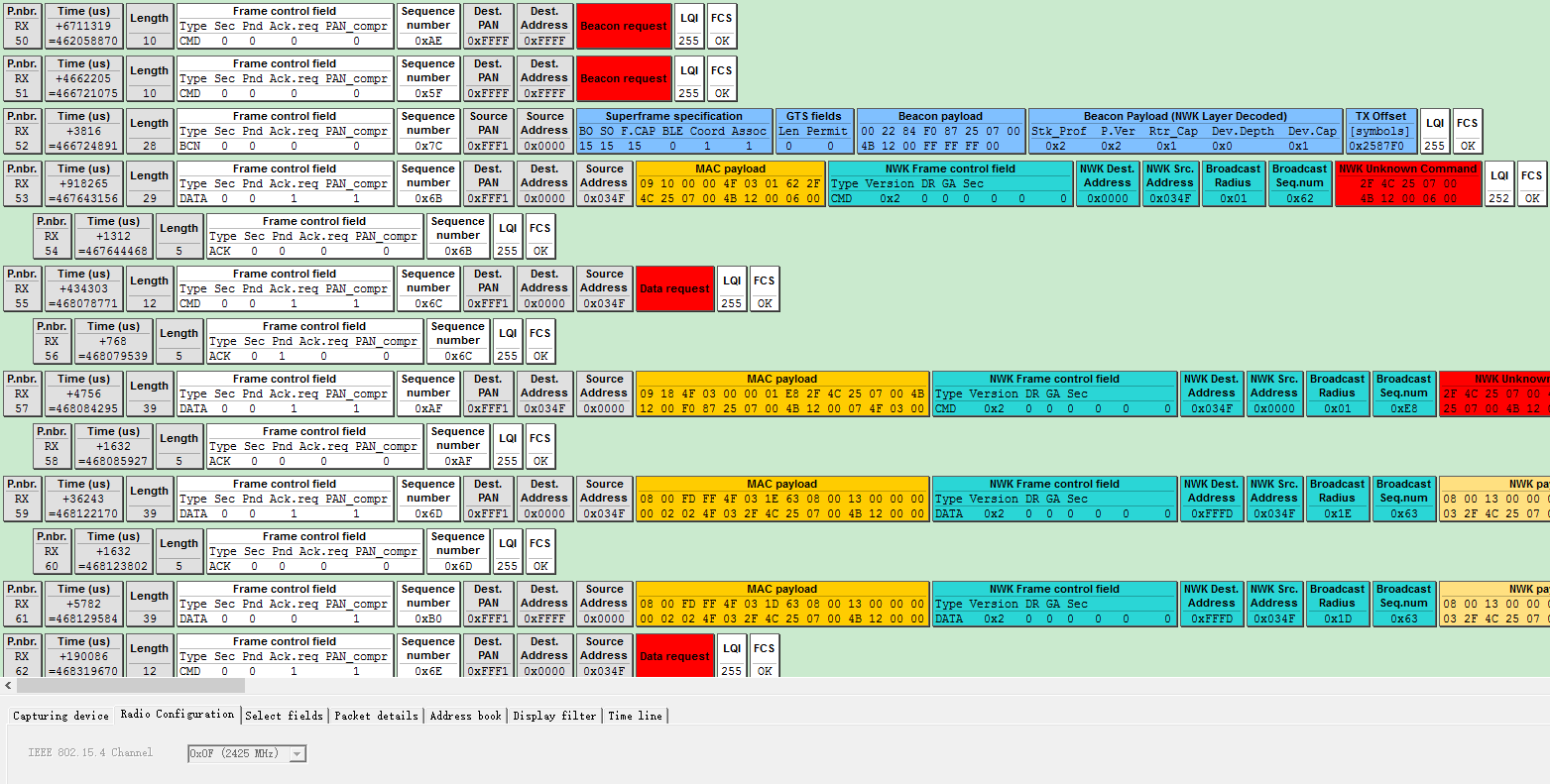


#### 协调器如果断电



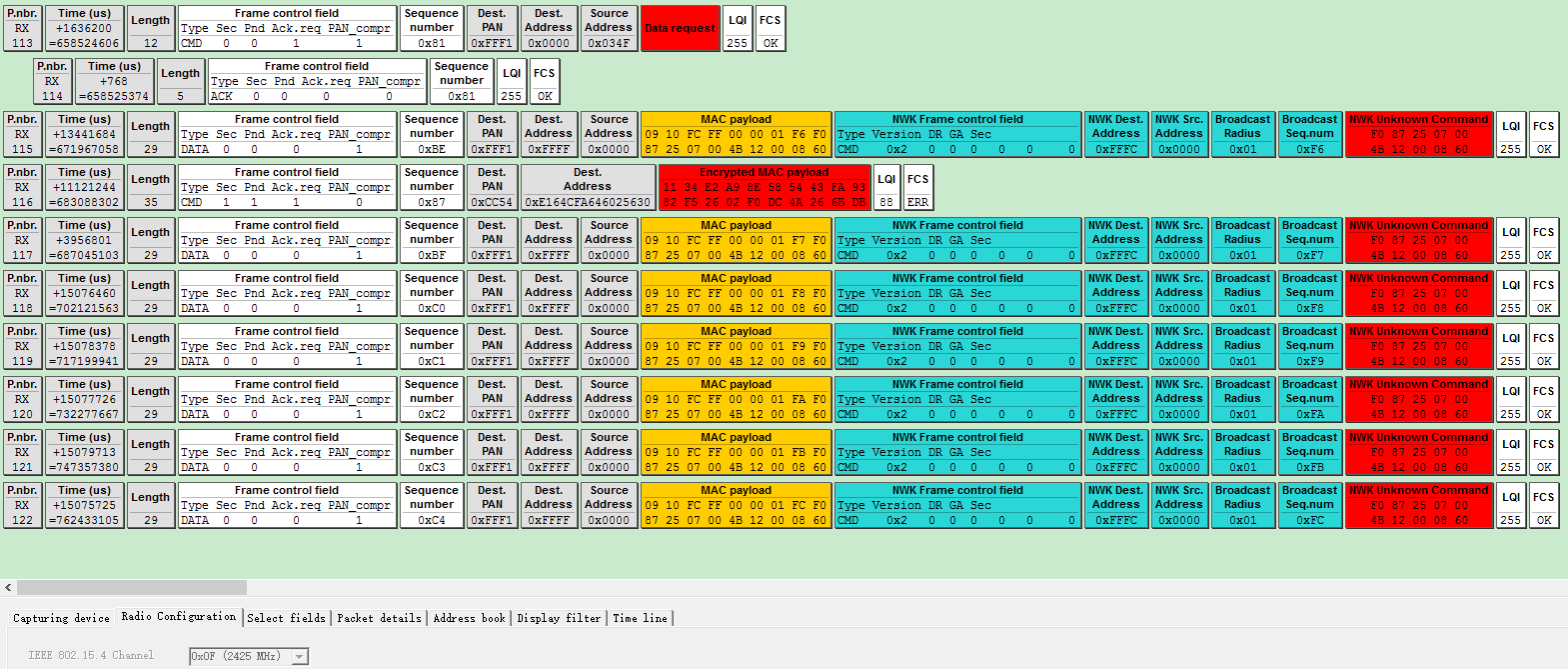
协调器断电时，终端设备将收不到Date request请求之后协调器MAC层自动回复的ACK,在之后的一段时间内终端设备会快速发送多次Date request请求，在多次请求未收到ACK后，终端设备将被判定为孤儿设备，此时终端设备又会开启信标请求帧请求加入网络。该帧的请求周期前面已经设置过了。

协调器再次上电后, 可以直接通过 orphan notification（而不是 beacon request），以更少的空中交互次数、更低的能量消耗完成再次入网

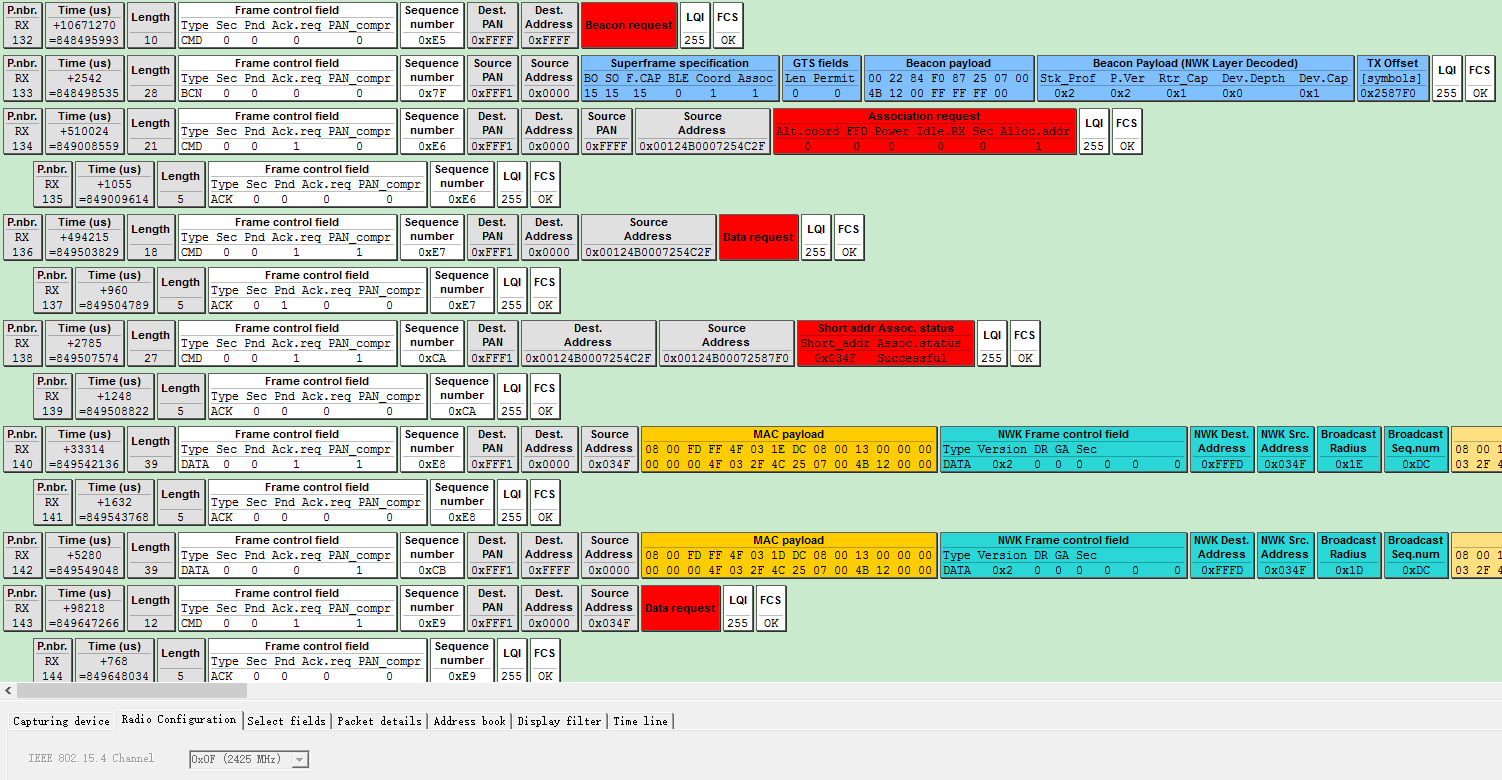


#### 终端设备如果断电

断电！



如果给终端设备再次上电连接



需要注意的网络短地址

没有变化，并没有重新分配！

#### 如果两个协调器同时上电

第二个协调器会分配一个和第一个协调器不一样的PAN ID地址。其实就是在PAN ID 0xFFF1的基础上又加了1.

#### 如果基站坏了，又坏了一个基站

终端设备一直连不上网，此时重新上电终端节点才可以……这种情况下唯一的解决方案就是重启终端的程序，需要一个程序的Reset.

### 数据测试过程

查看Zigbee技术原理与实战 P297

### Zigbee网络多信道调度

查看Zigbee技术原理与实战 P316 建立多个PAN网络！

# 低功耗

### PM2休眠

默认情况下电源管理设置为正常工作模式，如果要设置轻度休眠，设置为PM2模式：

第一步：添加编译选项：POWER\_SAVING

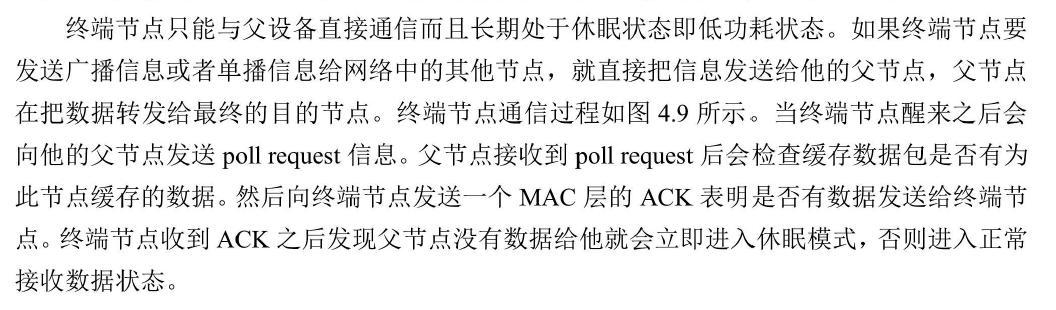
说明：默认情况下，Z-stack里终端设备的电源管理是关闭的。添加编译选项POWER\_SAVING以开启此功能。

第二步：在f8wConfig.cfg里设置：-DRFD\_RCVC\_ALWAYS\_ON = FALSE

说明：如果设置为TRUE会导致MAC层在休眠时不关闭，从而会阻止设备进入休眠模式。

第三步：设置DPOLL\_RATE、DQUEUED\_POLL\_RATE、DRESPONSE\_POLL\_RATE以及DREJOIN\_POLL\_RATE参数。例如要使终端设备进入深度休眠，则需要设置三个参数都为0。

说明：低功耗的设备如何接收协调器的数据呢？

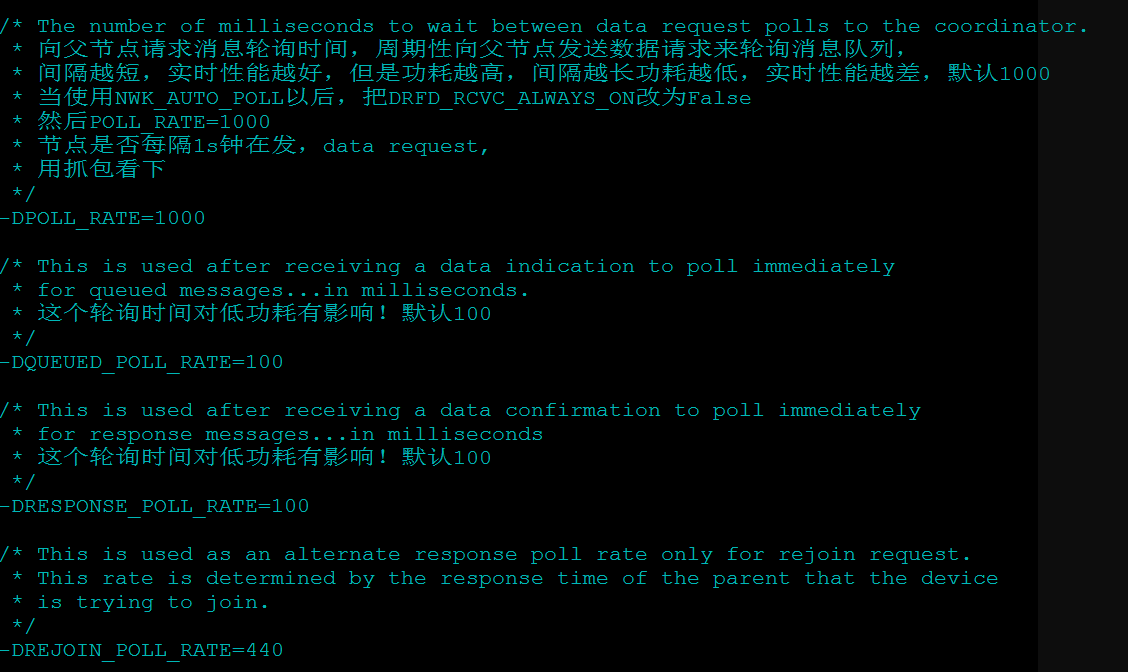


以上说明有问题，poll request其实是data request,发送的时间间隔就是DPOLL\_RATE设置的时间。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* The following are for End Devices only

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

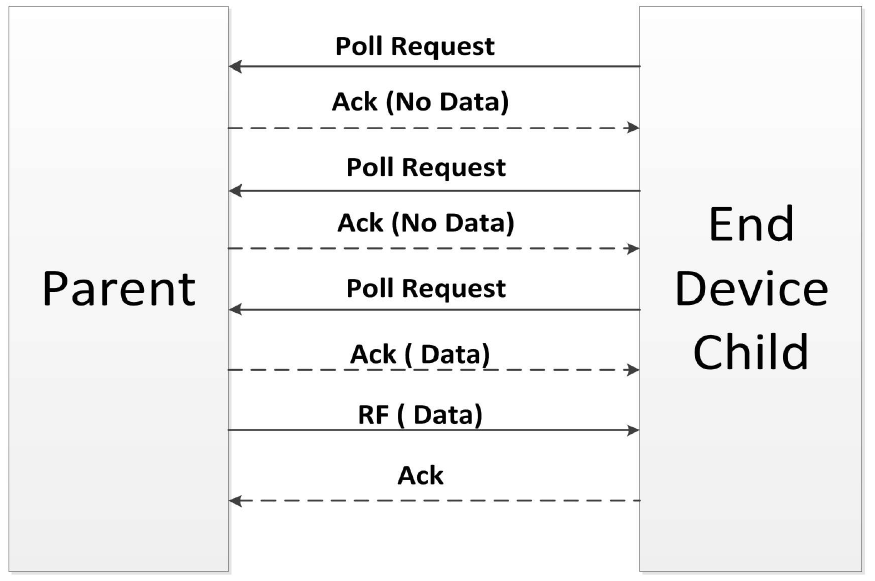


三个参数说明：

DPOLL\_RATE：终端设备向协调器发送的数据请求的时间间隔，默认设置为1s。(查看编译选项的时候我们发现有NWK\_AUTO\_POLL选项，该项设置就是为了使终端设备能够自动从父节点获取数据)。

DQUEUED\_POLL\_RATE：在收到数据指示后，就会向父节点请求消息。**这个应该是终端每隔一段时间向父节点发送数据。**

DRESPONSE\_POLL\_RATE：在收到数据确认指示后，就会向父节点请求响应消息。**这个时间是每隔一段时间向父节点发送ACK消息**



### 休眠时间的取值

休眠时间是Z-Stack协议栈自行计算的，取决于OSAL定时器下一次溢出的时间timeout,下一次溢出的时间一般是取所有打开的定时器中最小的时间作为睡眠时间，例如DPOLL\_RATE数据请求轮询时间间隔设置为1s,设定的睡眠时间为6s,那么1s就是睡眠时间。

### 门锁低功耗设计

1. 关门的时候立马让电机回转，减少程序的运行时间，降低功耗
2. 减少射频的发射功率
3. 电机的PWM占空比随着电池电量改变，缓慢启动电机，慢慢启动

# 信标和非新标网络

详细见Zigbee无线网络与收发器 P10

# 发射功率的修改

首先查看MAC -> Low Level –> Common -> mac\_radio.c -> macRadioInit()

MAC\_INTERNAL\_API void macRadioInit(void)

{

/\* variable initialization for this module \*/

reqChannel = MAC\_RADIO\_CHANNEL\_DEFAULT;

macPhyChannel = MAC\_RADIO\_CHANNEL\_DEFAULT;

reqTxPower = MAC\_RADIO\_TX\_POWER\_DEFAULT;

macPhyTxPower = MAC\_RADIO\_TX\_POWER\_DEFAULT;

}

其中reqTxPower和macPhyTxPower设置了发射功率的初始值，

#define MAC\_RADIO\_TX\_POWER\_DEFAULT 0x32

查看0x32是多少功率，在MAC -> Low Level –> System -> mac\_radio\_defs.c中

const uint8 CODE macRadioDefsTxPwrBare[] =

{

3, /\* tramsmit power level of the first entry \*/

(uint8)(int8)-22, /\* transmit power level of the last entry \*/

/\* 3 dBm \*/ 0xF5, /\* characterized as 4.5 dBm in datasheet \*/

/\* 2 dBm \*/ 0xE5, /\* characterized as 2.5 dBm in datasheet \*/

/\* 1 dBm \*/ 0xD5, /\* characterized as 1 dBm in datasheet \*/

/\* 0 dBm \*/ 0xD5, /\* characterized as 1 dBm in datasheet \*/

/\* -1 dBm \*/ 0xC5, /\* characterized as -0.5 dBm in datasheet \*/

/\* -2 dBm \*/ 0xB5, /\* characterized as -1.5 dBm in datasheet \*/

/\* -3 dBm \*/ 0xA5, /\* characterized as -3 dBm in datasheet \*/

/\* -4 dBm \*/ 0x95, /\* characterized as -4 dBm in datasheet \*/

/\* -5 dBm \*/ 0x95,

/\* -6 dBm \*/ 0x85, /\* characterized as -6 dBm in datasheet \*/

/\* -7 dBm \*/ 0x85,

/\* -8 dBm \*/ 0x75, /\* characterized as -8 dBm in datasheet \*/

/\* -9 dBm \*/ 0x75,

/\* -10 dBm \*/ 0x65, /\* characterized as -10 dBm in datasheet \*/

/\* -11 dBm \*/ 0x65,

/\* -12 dBm \*/ 0x55, /\* characterized as -12 dBm in datasheet \*/

/\* -13 dBm \*/ 0x55,

/\* -14 dBm \*/ 0x45, /\* characterized as -14 dBm in datasheet \*/

/\* -15 dBm \*/ 0x45,

/\* -16 dBm \*/ 0x35, /\* characterized as -16 dBm in datasheet \*/

/\* -17 dBm \*/ 0x35,

/\* -18 dBm \*/ 0x25, /\* characterized as -18 dBm in datasheet \*/

/\* -19 dBm \*/ 0x25,

/\* -20 dBm \*/ 0x15, /\* characterized as -20 dBm in datasheet \*/

/\* -21 dBm \*/ 0x15,

/\* -22 dBm \*/ 0x05 /\* characterized as -22 dBm in datasheet \*/

};

所以这里默认的TI官方设置在-18dbm ~ -17dbm之间

其中reqTxPower和macPhyTxPower设置了发射功率的初始值，

#define MAC\_RADIO\_TX\_POWER\_DEFAULT 0x32

修改这个参数即可！

# RSSI

RSSI=-(81-(LQI\*91)/255) //LQI是链路质量

LQI的取值是0x00~0xff，0x00表示质量最差

RSSI的正常范围可以是：【－93，－113】，超过这个范围，则可视为RSSI异常

RSSI异常的原因分类：

RSSI异常分3种情况，分别是过低、过高、主分级差值过大等，常见的引起RSSI异常原因有：工程质量问题、外界干扰、参数设置错误、设备故障和终端问题等。如下表所示：

RSSI异常情况 现象 可能原因

RSSI过低 主（分）集长时间RSSI低于-113dBm左右，不随负荷增加而改变 工程质量问题(包括从天馈到TRX的各个接头接触不好)、硬件故障(如天馈、TRX、CDU、功放故障)等。

RSSI过高 主（分）集长时间RSSI高于-93dBm或在一定时间内高于-93dBm 工程质量问题(跳线接头制作不规范，跳线损坏等)，接头进水、过高的话务量导致Abis或FMR资源不够、参数设置问题（登记及接入消息设置不合理）、外部干扰。

RSSI主分集差异过大 主（分）集两者间RSSI长时间相差6dB以上或出现RSSI主分集对比告警、TRM主（分）集接收告警 工程质量问题(跳线单级连接不好等)，天馈驻波、分集旁路开关设置错误、外部干扰

各个厂家要求有细微的所不同，一般要求不高于-95DBM

基站厂家 基站RSSI 正常范围主分集 RSSI 差值正常范围

中兴 -115dBm～-95dBm ±10dB

华为 -112dBm～-93dBm ±6dB

上海贝尔 抬升0dB～15dB ±10dB

摩托罗拉 -100dBm～-110dBm 城区5dB；乡村10dB

北电 -115dBm～-105dBm ±9dB

三星-85dBm～-100dBm ±10dB