**Zigbee网络特点的总结（网上的总结）**

1. 节点的入网

1.1原语的分析

NLME-JOIN.request   
{   
ExtendedPANId,   
RejoinNetwork,   
ScanChannels,   
ScanDuration,   
CapabilityInformation,   
SecurityEnable   
}

Join入网过程。首先发起Network Discovery，返回所有应答的节点的信息。在发现的结果中找出符合要求（这里的要求是一些最基本的条件，详见ZigBee协议规范3.6.1.4.1.1节，下同）的父节点，向它发送入网请求。父节点分配16位网络地址。

Rejoin入网过程。发起Network Discovery，在应答的节点中挑选出和自己的ExtendedPANID相同的节点，在这些节点中找出符合要求的父节点，发送入网请求，并且使用自己已拥有的16位网络地址（若没有，则随机生成一个）。

Orphan Join过程。发起Orphan Scan，寻找邻居表中保存有本设备IEEE地址的父节点，在返回结果中找出符合要求的父节点，发送入网请求。父节点返回邻居表中保存的16位网络地址。

1.2 Z-stack分析(TI)

1.2.1 扫描

ZDO\_StartDevice，它是设备入网流程的入口。

ZDApp\_NetworkInit函数也是设备入网过程的触发

终端的入网第一步就是调用了这两个函数NLME\_NetworkDiscoveryRequest、   
NLME\_OrphanJoinRequest（放到第3步再看），而Join和Rejoin方式的这一部分是完全相同的。

1.2.2扫描结果

ZDO\_NetworkDiscoveryConfirmCB()回调函数(返回结果)中发现，就做了一件事，就是向ZDApp\_event\_loop发送ZDO\_NWK\_DISC\_CNF事件

对于扫描结果的处理是这样一个流程：首先需进行至少NUM\_DISC\_ATTEMPTS次扫描，每次都调用ZDApp\_NetworkInit进行重新扫描，如果找到了合格的父节点（pChosenNwk = ZDApp\_NwkDescListProcessing()) != NULL），就依照MODE\_JOIN或 MODE\_REJOIN 分别调用NLME\_JoinRequest或NLME\_ReJoinRequest向目标父节点发送请求。由于后者的请求中要附带自己的PANID和ShortAddress，所以要事先检查和处理。

1.2.3加入父节点

ZDO\_JoinConfirmCB()一样只做了一件事，就是向ZDApp\_event\_loop发送事件ZDO\_NWK\_JOIN\_IND。至此终端就完成了入网的全部流程，如果被父节点接受，那么入网成功；如果失败，则重新开始入网流程

1.2.4路由器入网

Orphan Join，而根据源代码的逻辑，如果是路由器（NODETYPE\_ROUTER）执行Orphan Join。对于终端来说，都只能执行一次Orphan Join，与宏定义MAX\_RESUME\_RETRY毫无关系。

1.2.5 终端判断是否入网

终端通过定时发送MAC地址请求或发送数据的结果来判断协议器是否还在网络中.如果连续数次mac地址请求失败或发送数据失败,则认为协调器已经不在网络。检测次数通过f8wConfig.cfg中的宏MAX\_POLL\_FAILURE\_RETRIES来定,默认值为2.当检测到协调器不在网络时,app会收到ZDO\_STATE\_CHANGE事件,终端网络状态变为DEV\_NWK\_ORPHAN。

1、extern devStates\_t devState;（在ZDApp.h中声明）

typedef enum

{

DEV\_HOLD, // Initialized - not started automatically

DEV\_INIT, // Initialized - not connected to anything

DEV\_NWK\_DISC, // Discovering PAN's to join

DEV\_NWK\_JOINING, // Joining a PAN

DEV\_NWK\_REJOIN, // ReJoining a PAN, only for end devices

DEV\_END\_DEVICE\_UNAUTH, // Joined but not yet authenticated by trust //center

DEV\_END\_DEVICE, // Started as device after authentication

DEV\_ROUTER, // Device joined, authenticated and is a router

DEV\_COORD\_STARTING, // Started as Zigbee Coordinator

DEV\_ZB\_COORD, // Started as Zigbee Coordinator

DEV\_NWK\_ORPHAN // Device has lost information about its parent..

} devStates\_t;

参考：http://www.deyisupport.com/question\_answer/wireless\_connectivity/zigbee/f/104/t/104629.aspx

1.2.6参考的东西

<http://www.deyisupport.com/question_answer/wireless_connectivity/zigbee/f/104/t/104629.aspx>

1. 路由器网络管理

2.1关联表(Z-stack中关联表AssociationDevList )

Z-stack中关联表AssociationDevList :

主要存储与该节点有父子关系的相关节点信息，表中信息的 IEEE 地址是唯一的.

每个设备加入它的父节点时会在表中添加一个记 录（一个设备掉电重启后 sniffer中显示两个短地址的一个为掉电前短地址另一个为新获取的段地址）

但是子节点断电离开网络时不会删除该条记录，如果有多个子节点，频繁更换父节点会造成 AssociatedDevList 表满或溢出，使其它子节点无法加入该父节点，因此需要及时的删除一些已离开的子节点记录。

2.2 结构体的解释-AssociationDevList

Typedef struct

{

Uint16 shortAddr;//关联设备的短地址

Uint16 addriddx;//条目索引号，其实为0

Byte nodeRelation;//与该节点的关系

Bype devStatus; //设备状态

Byte assocCnt; //连接次数，相同IEEE地址连接时会加1

Byte age； //连续未接受到link status的次数(终端设备始终增加)

linkInfo\_t linkinfo //

}associated\_devices\_t;

全功能节点：

类型associated\_devices\_t中有个age的域。当关联表的维护者收到一条来自该短地址的Link Status消息，就会重置该域。全能节点（路由器）默认15秒发送15秒发送一次Link Status，那么父节点就可以通过age来判断它的全能子节点是否还连接在网络中

半功能设备：

如果是终端是不会定时发送Link Status的。通过age域的判断方法就是不可用了。可能终端会睡眠。

2.3 删掉关联表中的东西

根据AssociationDevList 条目中的age值来判断 路由节点是否脱离网络。若脱离网络，则先调用AddrMgrEntryGet( ) 获取丢失节点的短地址 和 扩展地址。然后调用NLME\_LeaveReq()/AssocRemove()对表中的条目信息进行清除操作。

2.3.1具体CODE

//Remove a device from the list. Uses the extended address.

extern byte AssocRemove( byte \*extAddr );

//能够根据 索引号 读取关联表、绑定表等表中的短地址和扩展地址；

extern uint8 AddrMgrEntryGet( AddrMgrEntry\_t\* entry );

//关联表的查看

//可以直接从AssociatedDevList[NWK\_MAX\_DEVICES]数组中查看关联表

For(uint8 x=0;x<NWK\_MAX\_DEVICES;)

{

Byte nr=AssociatedDevList[x].nodeRelation;

….

}

//关联表记录删除

//关联表删除的方法有很多种，可以用AssocList.h文件中的函数。也可以从

//NV中直接删除。下面对AssocList.h文件中的函数部分举例说明

AddrMgrEntry\_t addrEntry;

NLME\_LeaveReq\_t req;

Uint8 I=Z\_EXTADDR\_LEN+1;

//Set up device info

addrEntry.user=ADDRMGR\_USER\_DEFAULT;

addrEntry.index=index;

if(AddrMgrEntryGet(&addrEntry))

{

For(i=0;i<Z\_EXTADDR\_LEN;I++)

{

If(addrEntry.extAddr!=0){break}

}

}

If(i<Z\_extaddr\_len)

{

//Remove device

Req.extAddr=addeEntry.extAddr;

Req.removeChildren=TRUE;

Req.rejoin=TRUE;

Req.silent=FALSE;

NLME\_LeaveReq(&req);

}

Else

{

AssocRemove(addrEntry.extAddr);

ZDA pp\_NVUpdate();

}

2.3.2 问题点

有时使用NLME\_LeaveReq()会出现无效的情况，更换为AssocRemove()则没问题 （原因未知）

对脱离网络的 终端设备相关条目删除 需要定时 的通报ED在网络中，可参考提供的链接进行尝

2.3.3自己实现

// //地址管理表

// AddrMgrEntry\_t addrEntry;

// uint8 i=0;

// uint8 NodeCnt =0;

// uint8 a[1],b[2],c[2];

//

// // Set up device info

// addrEntry.user = ADDRMGR\_USER\_DEFAULT;

// for(int j=0; j<NWK\_MAX\_DEVICES; j++)

// {

// addrEntry.index = j;

// if (AddrMgrEntryGet( &addrEntry ))

// {

// NodeCnt++;

// a[0]=NodeCnt;

// b[0]=addrEntry.index/256;

// b[1]=addrEntry.index%256;

// c[0]=addrEntry.nwkAddr/256;

// c[1]=addrEntry.nwkAddr%256;

// BuffOledShow[0]=a[0]/16;

// BuffOledShow[1]=a[0]%16;

// BuffOledShow[0]=Hex\_To\_ASCII(BuffOledShow[0]);

// BuffOledShow[1]=Hex\_To\_ASCII(BuffOledShow[1]);

// BuffOledShow[3]='\0';

// HalUARTWrite(0,"N:", 2);

// HalUARTWrite(0,BuffOledShow,2);

// HalUARTWrite(0, "\r\n" ,2);

// BuffOledShow[0]=b[0]/16;

// BuffOledShow[1]=b[0]%16;

// BuffOledShow[2]=b[1]/16;

// BuffOledShow[3]=b[1]%16;

// BuffOledShow[0]=Hex\_To\_ASCII(BuffOledShow[0]);

// BuffOledShow[1]=Hex\_To\_ASCII(BuffOledShow[1]);

// BuffOledShow[2]=Hex\_To\_ASCII(BuffOledShow[2]);

// BuffOledShow[3]=Hex\_To\_ASCII(BuffOledShow[3]);

// BuffOledShow[4]='\0';

// HalUARTWrite(0,"I:", 2);

// HalUARTWrite(0, BuffOledShow,4);

// HalUARTWrite(0, "\r\n" ,2);

// BuffOledShow[0]=c[0]/16;

// BuffOledShow[1]=c[0]%16;

// BuffOledShow[2]=c[1]/16;

// BuffOledShow[3]=c[1]%16;

// BuffOledShow[0]=Hex\_To\_ASCII(BuffOledShow[0]);

// BuffOledShow[1]=Hex\_To\_ASCII(BuffOledShow[1]);

// BuffOledShow[2]=Hex\_To\_ASCII(BuffOledShow[2]);

// BuffOledShow[3]=Hex\_To\_ASCII(BuffOledShow[3]);

// BuffOledShow[4]='\0';

// HalUARTWrite(0,"A:", 2);

// HalUARTWrite(0,BuffOledShow,4);

// HalUARTWrite(0, "\r\n" ,2);

// }

// }

//

//关联表2

uint8 e[2];

for (uint8 x=0;x<NWK\_MAX\_DEVICES;x++)

{

if(AssociatedDevList[x].shortAddr!=0xffff)

{

// uint16 sa = AssociatedDevList[x].shortAddr;

// e[0]=sa/256;

// e[1]=sa%256;

// BuffOledShow[0]=e[0]/16;

// BuffOledShow[1]=e[0]%16;

// BuffOledShow[2]=e[1]/16;

// BuffOledShow[3]=e[1]%16;

// BuffOledShow[0]=Hex\_To\_ASCII(BuffOledShow[0]);

// BuffOledShow[1]=Hex\_To\_ASCII(BuffOledShow[1]);

// BuffOledShow[2]=Hex\_To\_ASCII(BuffOledShow[2]);

// BuffOledShow[3]=Hex\_To\_ASCII(BuffOledShow[3]);

// BuffOledShow[4]='\0';

// HalUARTWrite(0,"sa:", 3);

// HalUARTWrite(0, BuffOledShow,4);

// HalUARTWrite(0, "\r\n" ,2);

uint16 ai= AssociatedDevList[x].addrIdx;

// e[0]=ai/256;

e[1]=ai%256;

// BuffOledShow[0]=e[0]/16;

// BuffOledShow[1]=e[0]%16;

BuffOledShow[0]=e[1]/16;

BuffOledShow[1]=e[1]%16;

// BuffOledShow[0]=Hex\_To\_ASCII(BuffOledShow[0]);

// BuffOledShow[1]=Hex\_To\_ASCII(BuffOledShow[1]);

BuffOledShow[0]=Hex\_To\_ASCII(BuffOledShow[0]);

BuffOledShow[1]=Hex\_To\_ASCII(BuffOledShow[1]);

// BuffOledShow[4]='\0';

// HalUARTWrite(0,"ai:", 3);

HalUARTWrite(0, BuffOledShow,2);

// HalUARTWrite(0, "\r\n" ,2);

// Delay1ms(200);

//

// uint8 nr = AssociatedDevList[x].nodeRelation;

// BuffOledShow[0]=nr/16;

// BuffOledShow[1]=nr%16;

// BuffOledShow[0]=Hex\_To\_ASCII(BuffOledShow[0]);

// BuffOledShow[1]=Hex\_To\_ASCII(BuffOledShow[1]);

// BuffOledShow[3]='\0';

// HalUARTWrite(0,"nr:", 3);

// HalUARTWrite(0,BuffOledShow,2);

// HalUARTWrite(0, "\r\n" ,2);

//

// uint8 ds = AssociatedDevList[x].devStatus;

// BuffOledShow[0]=ds/16;

// BuffOledShow[1]=ds%16;

// BuffOledShow[0]=Hex\_To\_ASCII(BuffOledShow[0]);

// BuffOledShow[1]=Hex\_To\_ASCII(BuffOledShow[1]);

// BuffOledShow[3]='\0';

// HalUARTWrite(0,"ds:", 3);

// HalUARTWrite(0,BuffOledShow,2);

// HalUARTWrite(0, "\r\n" ,2);

//

// uint8 ac = AssociatedDevList[x].assocCnt;

// BuffOledShow[0]=ac/16;

// BuffOledShow[1]=ac%16;

// BuffOledShow[0]=Hex\_To\_ASCII(BuffOledShow[0]);

// BuffOledShow[1]=Hex\_To\_ASCII(BuffOledShow[1]);

// BuffOledShow[3]='\0';

// HalUARTWrite(0,"ac:", 3);

// HalUARTWrite(0,BuffOledShow,2);

// HalUARTWrite(0, "\r\n" ,2);

//

// uint8 ag = AssociatedDevList[x].age;

// BuffOledShow[0]=ag/16;

// BuffOledShow[1]=ag%16;

// BuffOledShow[0]=Hex\_To\_ASCII(BuffOledShow[0]);

// BuffOledShow[1]=Hex\_To\_ASCII(BuffOledShow[1]);

// BuffOledShow[3]='\0';

// HalUARTWrite(0,"ds:", 3);

// HalUARTWrite(0,BuffOledShow,2);

// HalUARTWrite(0, "\r\n" ,2);

}

}

参考http://blog.csdn.net/amd123linux/article/details/12190895

2.3.4参考

<http://bbs.feibit.com/thread-3841-1-1.html>

<http://e2e.ti.com/support/low_power_rf/f/158/p/105319/676645.aspx>

<http://blog.csdn.net/mahaichao540174345/article/details/7973785>

2.4 路由表，邻居表和关联表

2.4.1 关联表和邻居表的简单介绍：

关联表是用来保存父子设备的

邻居表用来保存邻居设备

一个路由器最大的数目还是有内存空间决定的：

定义在nwk\_globals.h。#define NWK\_MAX\_DEVICE\_LIST 20 // Maximum number of devices in the。

2.4.2 打印路由表和邻居表

邻居表和路由表在Z-stack已经定义，我们可以从网络属性库中利用z-stack提供的API来打印信息。

在（NLMEDE.h）中定义：

Zstatus\_t NLME\_GetRequest(ZNwkAttributes\_t NIBAttribute,uint16 index,void \*Value);

参数描述：NIBAttribute-支持的属性：

nwkCapablityinfo 网络容量

nwkNumNeighbor TableEntries 邻居表数目

nwkNeighbor Table 邻居表

nwkNumRoutingTableEntries 路由表数目

nwkRoutingTable 路由表

具体code参考：

http://blog.csdn.net/mahaichao540174345/article/details/7973785

2.4.3 邻居表和路由表的Z-stack定义

邻居表：

Typedef struct

{

Uint16 neighborAddress;//相邻地址

Uint16 panid; //所属的PAN网络ID

linkInfo\_t linkInfo; //连接信息（包括发送/接受和安全针计数）

}neighborEntry\_t;

Typedef struct

{

Uint8 txCounter //Counter of transmission success/failures

Uint8 txCost; //Average of sending rssi values if link status is enabled.

//i.e NWK\_LINK\_STATUS\_PERIOD is define as non zero

Uint8 rxLqi; //average of received rssi values

//needs to be converted to link cost(1\_7)before used

Uint8 inKeySeqNum;//security key sequence number

Uint32 inFtmCntr; //security frame counter…

Uint16 txFailure; //higher values indicate more failures

}linkInfo\_t;

路由表定义格式

Typedef struct

{

Uint16 dstAddress;//目的地址

Uint16 nextHopAddress;//单挑地址

Byte expiryTime; //有效时间

Byte status; //状态

}rtgEntry\_t;

3使用NV\_INIT和NV\_RESTORE对网络影响

NV\_INIT保存在nv的内容主要是网络或者节点的配置参数，在节点运行中这些参数很少变化或者几乎不会有变化。NV\_INIT选项保存的所有参数可以在结构体zgItemTable中查看到。

NV\_RESTORE选项保存的内容主要是节点运行过程中的状态量，借助这个特性，节点在掉电或者复位之后不需要无线电通讯就可以恢复到之前的网络连接状态。NV\_RESTORE保存的内容主要有网络层[数据库](http://lib.csdn.net/base/mysql)NIB、设备关联表、绑定表、路由表等；

但是由于添加这两个选项，终端会继续尝试加入之前的网络而导致很难加入新的zigbee网络

3.1 再次入网的方法

方法一：

zgWriteStartupOptions(ZG\_STARTUP\_SET, ZCD\_STARTOPT\_DEFAULT\_NETWORK\_STATE);

SystemReset();

方法二：

NIB\_init();

NLME\_UpdateNV(NWK\_NV\_NIB\_ENABLE);

SystemReset();

方法三：

\_NIB.nwkDevAddress = INVALID\_NODE\_ADDR;

\_NIB.nwkLogicalChannel = 0;

\_NIB.nwkCoordAddress = INVALID\_NODE\_ADDR;

osal\_memset( \_NIB.nwkCoordExtAddress, 0, Z\_EXTADDR\_LEN );

\_NIB.nwkPanId = INVALID\_NODE\_ADDR;

NLME\_UpdateNV(NWK\_NV\_NIB\_ENABLE);

SystemReset();

方法四：

NLME\_InitNV();

NLME\_SetDefaultNV(); //设置默认NV条目

SystemReset();

4.Z-stack 2.6.x的新功能介绍

4.1 child aging 来解决终端的问题：

在Z-Stack 如果使能zgChildAgingEnable功能。

那么End\_Device 在入网成功以后，会通过Device Announce把Timeout相关的信息发送给父设备。

ZDApp\_AnnounceNewAddress(void)里的NLME\_SendEndDevTimeoutReq.

这个函数传递2个参数给父设备，也就是说不同节点可以配置不同的timeout。

Uint8 zgEndDeviceTimeoutValue=END\_DEV\_TIMEOUT\_VALUE;

Uint8 zgEndDeviceConfiguration=END\_DEV\_CONFIGURATION;

父设备上面，收到这个Req以后，会调用下面这个函数

NLME\_processEndDeviceTimeoutReq(NLDE\_FrameFormat\_t \*ff)

这个函数处理的过程，主要把收到的zgEndDeviceTimeoutValue,写道AssociateList。

Typedef struct

{

Uint16 shortAddr;

Uint16 addridx

Byte nodeRelation

Byte devStatus;

Byte assoccCnt;

Byte age;

Linkinfo\_t linklnfo;

Aging\_end\_device\_t endDev;

Uint32 timeoutCounter;

Bool keepaliveRcv;

}associated\_devices\_t;

在父设备会定期的去更新这个表里面的，周期性调用AssocChildAging,更新的周期是1s，开一个1s的time事件。每隔1s就会把Associate\_List减一。

同样只要在底层收到节点的Date request或者其他应用数据。立刻把值回复到原值。调用AssocChildTableUpdateTimeout。

如果timecounter减1等于0了，就把节点放到一个NotmyChildlist。

//Not my child structure

Typedef struct

{

Uint16 shortAddr;

Uint16 timeoutCounter;

}nwkNotMyChild\_t;

如果收到data req，就会把节点从NotmyChildlist重新移到Associatelist里面。

如果在NotmyChildList里面的counter的时候，就彻底删掉NwkNotMyChildSendLeave。

判断节点是否在线，只要判断AssociateList还是否存在就可以知道了

AssocGetWithShort（uint16 shortAddr）

协议栈里面的关联表的定义

Associated\_devices\_t AssociatedDevList[NWK\_MAX\_DEVICES];

nwkNotMyChild\_t notMyChildList[MAX\_NOT\_MYCHILD\_DEVICES];

4.2启动child的功能和注意点

4.2.1启动Child

使能ZIGBEE\_CHILD\_AGING,将zgChilAgingEnable设置为True。

关于Child Aging的timeout定义在协议栈nwk\_gloables.c文件中，如下：

CONST uint16 gKEEPALIVE\_TIMEOUT\_MAX = KEEPALIVE\_TIMEOUT\_MAX; //在协议栈运行过程中，被初始化以后就不会变化了，用来表示节点最大的Age时间可以是60分钟

CONST uint16 gTIMEOUT\_COUNTER\_MAX = TIMEOUT\_COUNTER\_MAX；//在协议栈运行过程中是动态变化的，协议栈中有一个1分钟的循环timer event，当timer event触发的时候，会去检测在这个在一分钟是否有收到Data request，或者其他数据，如果没有的话就会把timeoutCounter减1，如果连续60分钟没有收到的话就把这个节点放到另外一个not child 的list里面去了，在中途只要收到一条来自这个节点的数据，这个counter立刻被Reset成初始化60，这个值就是通过endDevKaTimeout获得的

4.2.2 一些特点

4.2.2.1zgChildAgingEnable ：

在1.2.2里面。如果使能了zgChildAgingEnable功能

那么EndDevice 在入网成功以后。会通过Device Announce把Timeout相关信息发给父设备

uint8 zgEndDeviceTimeoutValue = END\_DEV\_TIMEOUT\_VALUE;  
uint8 zgEndDeviceConfiguration = END\_DEV\_CONFIGURATION;

在父设备上面，收到这个req以后，会调用到下面这个函数

NLME\_ProcessEndDevTimeoutReq( NLDE\_FrameFormat\_t \*ff )

这个函数处理的过程，主要是把收到的zgEndDeviceTimeoutValue，写到AssociateList里面去

4.2.2.2zgRouterOffAssocCleanup：

In case a ZigBee Router gets off network for a long period of time, its children will try to join an alternative parent.

When the router is back online, the children will still appear in its child table, preventing proper routing of egress traffic to them.

In order to avoid this, it is recommended that routers prone to get off and on the network will have zgRouterOffAssocCleanup flag set to TRUE (mapped to NV item: ZCD\_NV\_ROUTER\_OFF\_ASSOC\_CLEANUP):

uint8 cleanupChildTable = TRUE;zgSetItem( ZCD\_NV\_ROUTER\_OFF\_ASSOC\_CLEANUP, sizeof(cleanupChildTable),&cleanupChildTable );

When enabled, deprecated end-device entries will be removed from the child table if traffic received from them was routed by another parent.

4.3 Z-Stack Home 1.2.1 和Z-Stack Home 1.2.2的差别

在Z-Stack Home 1.2.1 里面对Child Aging是一样的

父设备对于end device来说，他的time的时间是在父设备上实现固定。

// Child table Management timeout values  
CONST uint16 gKEEPALIVE\_TIMEOUT\_MAX = KEEPALIVE\_TIMEOUT\_MAX;  
CONST uint16 gTIMEOUT\_COUNTER\_MAX = TIMEOUT\_COUNTER\_MAX;

在1.2.2需要子设备在入网以后告诉父设备自己的poll rate是多少。

void ZDApp\_AnnounceNewAddress( void )  
{  
#if defined ( ZIGBEEPRO )  
// Turn off data request hold  
APSME\_HoldDataRequests( 0 );  
#endif

ZDP\_DeviceAnnce( NLME\_GetShortAddr(), NLME\_GetExtAddr(),  
ZDO\_Config\_Node\_Descriptor.CapabilityFlags, 0 );

#if defined ( ZIGBEEPRO )  
// Setup the timeout  
APSME\_HoldDataRequests( ZDAPP\_HOLD\_DATA\_REQUESTS\_TIMEOUT );  
#endif

if ( ZSTACK\_END\_DEVICE\_BUILD )  
{  
if ( zgChildAgingEnable == TRUE )  
{  
uint8 coordExtAddr[Z\_EXTADDR\_LEN];

// Send the message to parent  
NLME\_GetCoordExtAddr( coordExtAddr );  
**NLME\_SendEndDevTimeoutReq( NLME\_GetCoordShortAddr(), coordExtAddr,**  
**zgEndDeviceTimeoutValue,**  
**zgEndDeviceConfiguration );**  
}  
}  
}

父设备在收到这个消息以后，就会处理，把该节点的timeout时间修改过来的

void NLME\_ProcessEndDevTimeoutReq( NLDE\_FrameFormat\_t \*ff ）

5.自己的项目遇到的特性：

5.1 关联表和地址表的关系：

5.1.1协调器的关联表和地址表的关系

协调器的地址表比关联表多5个。且地址表的个数不会随节点的缺失而改变。会发生地址表满了，节点入不了网。

5.1.2路由器的关联表和地址表的关系

路由器的地址表和关联表一样多。地址表会随节点改变而改变。

5.2 函数zgWriteStartupOptions

这个函数会清掉协调器，路由器，终端的原来保存网络信息的NV。使节点使新的一样。

nwk\_Status: