FS_ZigBee 协议栈实验指导书

(ZigBee Evaluation Board V1.0)

公司名称: 锋硕电子科技有限公司

公司网址: www.fuccesso.com.cn

联系人: 王锋

邮箱: fuccesso@163.com

联系电话: 13912636180

日期: 2010/6/24

目录

第一章 基础概念

1.1 ZigBee 技术的概念

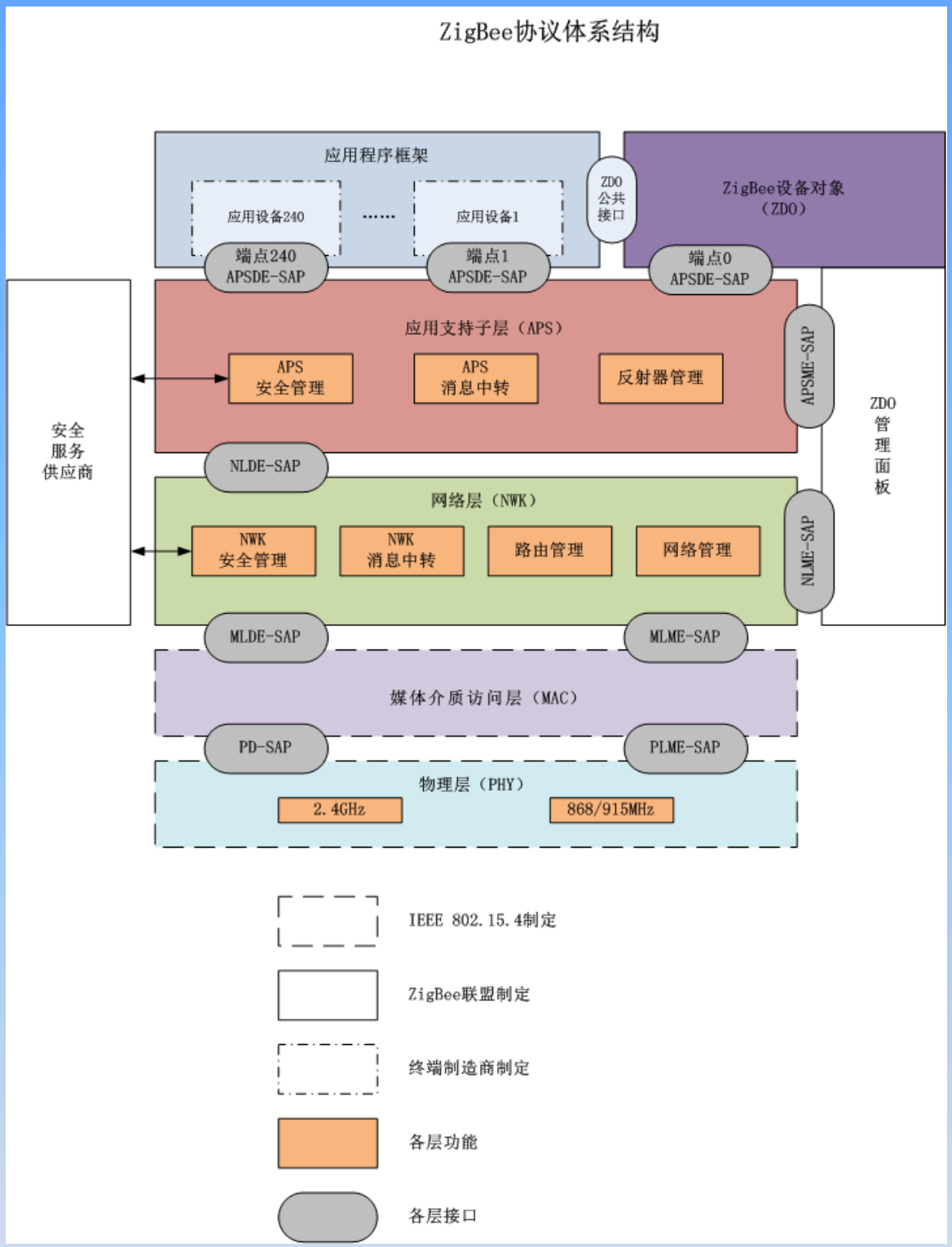
无线传感网络的无线通信技术可以采用 ZigBee 技术、蓝牙、Wi-Fi 和红外等技术。

ZigBee 技术是一种短距离、低复杂度、低功耗、低数据速率、低成本的双向无线通信技术或无线网络技术，是一组基于 IEEE802.15.4 无线标准研制开发的有关组网、安全和应用软件方面的通信技术。

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

1.2 ZigBee 协议的体系结构

ZigBee 的体系结构由称为层的各模块组成。每一层为其上层提供特定的服务：即由数据服务实体提供数据传输服务；管理实体提供所有的其他管理服务。每个服务实体通过相应的服务接入点(SAP)为其上层提供一个接口，每个服务接入点通过服务原语来完成所对应的功能。ZigBee 协议的体系结构如下图所示：



- **物理层（PHY）**

物理层定义了物理无线信道和 MAC 子层之间的接口，提供物理层数据服务和物理层管理服务。

物理层内容：

1)ZigBee 的激活；2)当前信道的能量检测；3)接收链路服务质量信息；4)ZigBee

信道接入方式；5)信道频率选择；6)数据传输和接收。

- **介质接入控制子层（MAC）**

MAC 层负责处理所有的物理无线信道访问，并产生网络信号、同步信号；支持 PAN 连接和分离，提供两个对等 MAC 实体之间可靠的链路。

MAC 层功能：

1）网络协调器产生信标；

2）与信标同步；

3）支持 PAN(个域网)链路的建立和断开；

4）为设备的安全性提供支持；

5）信道接入方式采用免冲突载波检测多址接入(CSMA-CA)机制；

6）处理和维护保护时隙(GTS)机制；

7）在两个对等的 MAC 实体之间提供一个可靠的通信链路。

- **网络层（NWK）**

ZigBee 协议栈的核心部分在网络层。网络层主要实现节点加入或离开网络、接收或抛弃其他节点、路由查找及传送数据等功能。

网络层功能：

1)网络发现；2)网络形成；3)允许设备连接；4)路由器初始化；5)设备同网络连接；6)直接将设备同网络连接；7)断开网络连接；8)重新复位设备；9)接收机同步；10)信息库维护。

- **应用层（APL）**

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

应用支持层的功能包括：维持绑定表、在绑定的设备之间传送消息。

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

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

- **应用程序框架（AF）：**

运行在 ZigBee 协议栈上的应用程序实际上就是厂商自定义的应用对象，并且遵循规范（profile）运行在端点 1~ 240 上。在 ZigBee 应用中，提供 2 种标准服务类型：键值对（KVP）或报文（MSG）

- **ZigBee 设备对象（ZDO）：**

远程设备通过 ZDO 请求描述符信息，接收到这些请求时，ZDO 会调用配置对象获取相应描述符值。另外，ZDO 提供绑定服务。

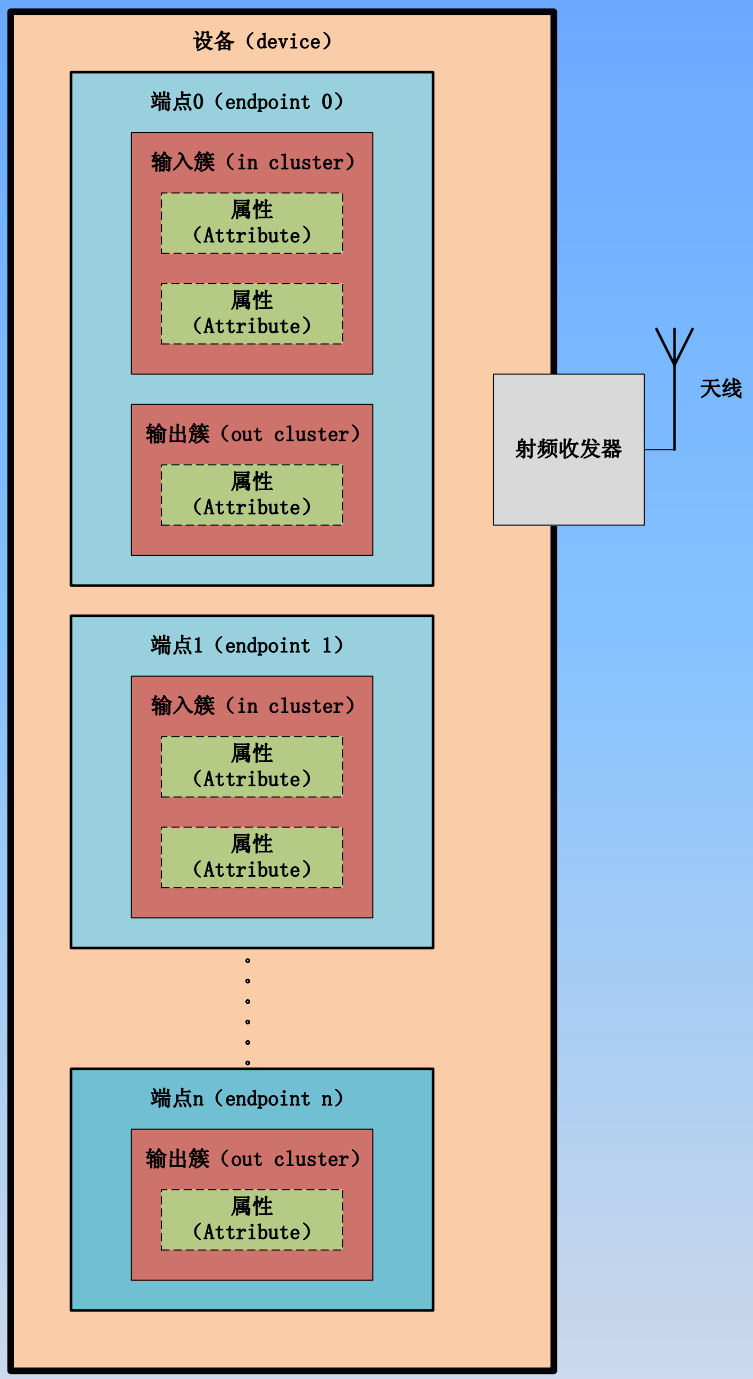
协议栈体系分层架构与协议栈代码文件夹对应表如下：

协议栈体系分层架构	协议栈代码文件夹
物理层（PHY）	硬件层目录（HAL）
介质接入控制子层（MAC）	链路层目录（MAC 和 Zmac）
网络层（NWK）	网络层目录（NWK）
应用支持层(APS)	网络层目录（NWK）
应用程序框架（AF）	配置文件目录（Profile）和应用程序（sapi）
ZigBee 设备对象（ZDO）	设备对象目录（ZDO）

1.3 ZigBee 协议术语

ZigBee 基本术语之间的关系图如下：

ZigBee协议术语关系



- **设备（device）**

一个节点（FFD/RFD）就是一个设备，对应一个无线单片机（CC2430）；一个设备有一个射频端，具有唯一的 IEEE 地址（64 位）和网络地址（16 位）。在协议栈中不同的设备有相应的配置文件：

协调器（全功能设备 FFD）配置文件：f8wCoord.cfg

路由器（全功能设备 FFD）配置文件：f8wRouter.cfg

终端设备（半功能设备 RFD）配置文件：f8wEndev.cfg

- **端点（endpoint）**

它是一个 8 位的字段，描述一个射频端所支持的不同应用。

端点 0x00：用于寻址设备配置文件，这是每个 ZigBee 设备必须使用的端点；

端点 0xff：用于寻址所有活动端点；

端点 0xf1~0xfe：保留；

端点 0x01~0xf0：共支持 240 个应用，即一个物理信道最多支持 240 个虚拟链路。

- **簇（cluster）**

多个属性的汇集形成了簇，簇是属性的集合，每个簇也拥有一个唯一的 ID。譬如，

FS_ZStack\ SimpleSwitch.c

```
const cId_t zb_OutCmdList[NUM_OUT_CMD_SWITCH] = //输出簇列表
{
  TOGGLE_LIGHT_CMD_ID                          //簇 ID, 1
};
```

FS_Zstack\SimpleController.c

```
const cId_t zb_InCmdList[NUM_IN_CMD_CONTROLLER] =    //输入簇列表
{
  TOGGLE_LIGHT_CMD_ID                          //簇 ID, 1
};
```

- **属性（attribute）**

设备之间通信的每一种数据像开关的状态或温度计值等皆可称为属性。每个属性可得到唯一的 ID，它们都用结构体来描述。

FS_Zstack\zcl.h

```
typedef struct
{
  uint16   attrId;              // Attribute ID
  uint8    dataType;            // Data Type - defined in AF.h
  uint8    accessControl;       // Read/write - bit field
  void     *dataPtr;            // Pointer to data field
} zclAttribute_t;


typedef struct
{
  uint16            clusterID;      // Real cluster ID
```

```
  zclAttribute_t      attr;
} zclAttrRec_t;
```

➢ 描述符（Descriptor）

一个设备(device)可以有 240 个端点（endpoint 1~endpoint 240），每一个端点必须有一个端点描述符 endPointDesc，端点描述符里包括一个简单描述符 SimpleDescriptionFormat，它们都用结构体来描述。

```
Typedef struct
{
  Byte endPoint;                          //端点号 1-240
  byte *task_id;                          //任务 ID 号
  SimpleDescriptionFormat_t *simpleDesc;  //简单描述符
  afNetworkLatencyReq_t   latencyReq;     //延时请求
} endPointDesc_t;                         //端点描述符
```

譬如：

FS_Zstack \sapi.c

对端点描述符进行初始化

```
  sapi_epDesc.endPoint = zb_SimpleDesc.EndPoint;
  sapi_epDesc.task_id = &sapi_TaskID;
  sapi_epDesc.simpleDesc = (SimpleDescriptionFormat_t *)&zb_SimpleDesc;
  sapi_epDesc.latencyReq = noLatencyReqs;
```

特别的，

FS_Zstack \ZDApp.c

端点 0 的端点描述符为：

```
endPointDesc_t ZDApp_epDesc =
{
  ZDO_EP,                                 //端点 0 ID,   0
  &ZDAppTaskID,
  (SimpleDescriptionFormat_t *)NULL,      // No Simple description for ZDO
  (afNetworkLatencyReq_t)0                // No Network Latency req
};
typedef struct
{
  byte        EndPoint;                   //端点号 1-240
  uint16      AppProfId;                  //支持的 Profile ID
  uint16      AppDeviceId;                //支持的设备 ID
  byte        AppDevVer:4;                //执行的设备描述的版本
  byte        Reserved:4;                 //保留
  byte        AppNumInClusters;           //终端支持的输入簇数目
```

```
  cId_t       *pAppInClusterList;     //指向输入 Cluster ID 列表的指针
  byte         AppNumOutClusters;     //终端支持的输出簇数目
  cId_t       *pAppOutClusterList;    //指向输出 Cluster ID 列表的指针
} SimpleDescriptionFormat_t;          //简单描述符
```

譬如开关设备（switch）和控制器设备（controller）均具有不同的简单描述符：

1）开关设备（switch）简单描述符：

FS_Zstack\SimpleSwitch.c

```
const SimpleDescriptionFormat_t zb_SimpleDesc =
{
  MY_ENDPOINT_ID,              // Endpoint, 0x02
  MY_PROFILE_ID,               // Profile ID, 0x0F10
  DEV_ID_SWITCH,               // Device ID, 1
  DEVICE_VERSION_SWITCH,       // Device Version, 1
  0,                           // Reserved
  NUM_IN_CMD_SWITCH,           // Number of Input Commands, 0
  (cId_t *) NULL,              // Input Command List
  NUM_OUT_CMD_SWITCH,          // Number of Output Commands, 1
  (cId_t *) zb_OutCmdList      // Output Command List
};
```

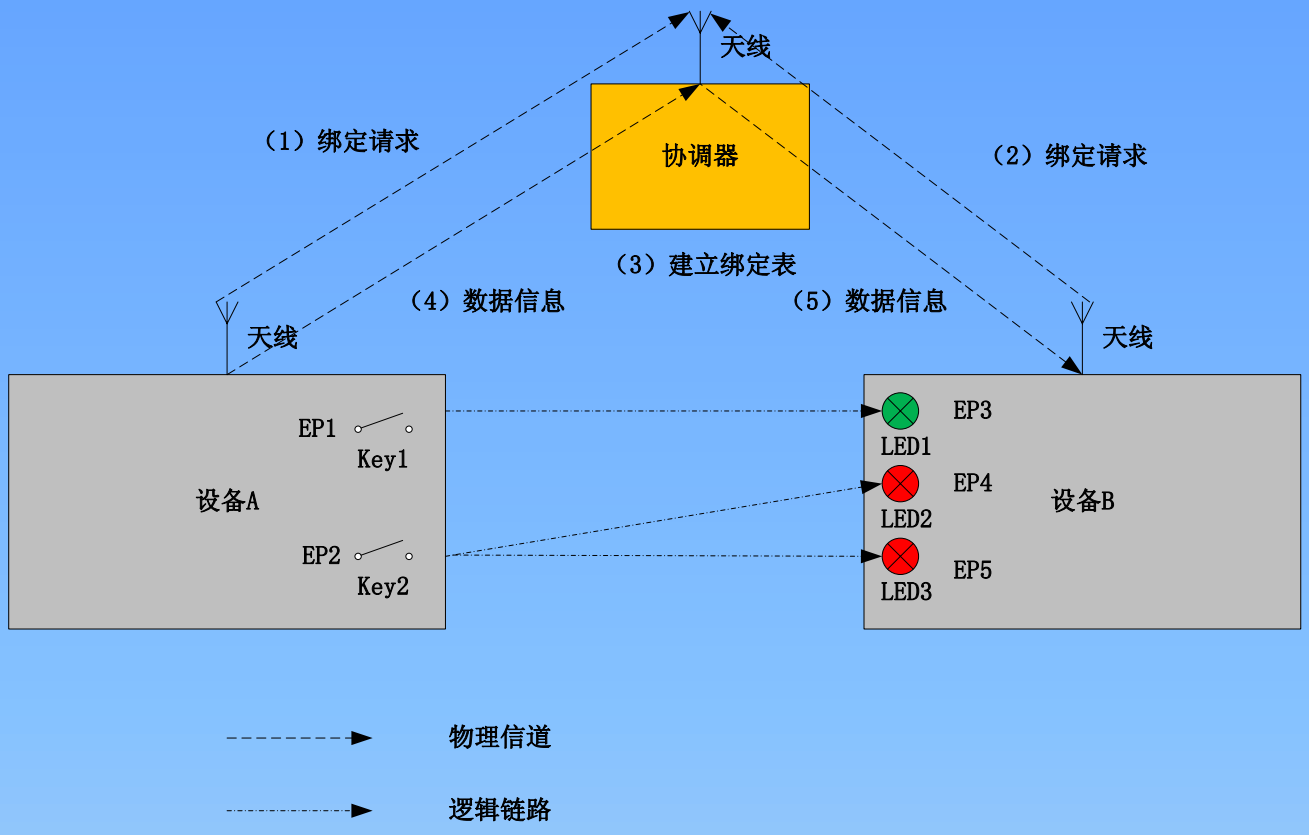
2）控制器设备（Controller）简单描述符：

FS_Zstack \SimpleController.c

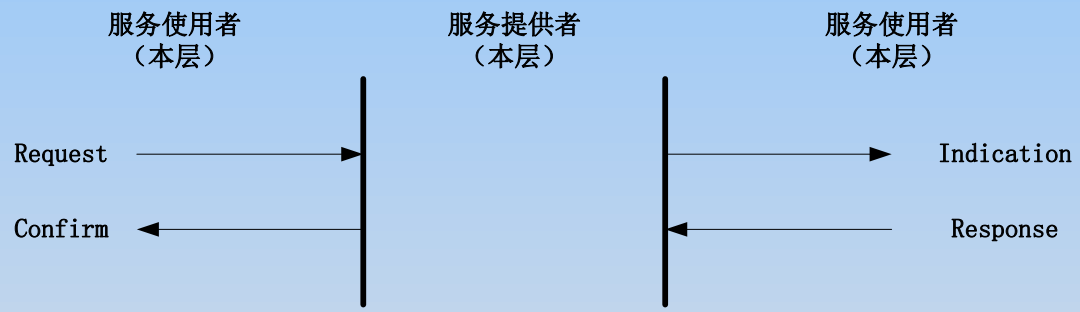
```
const SimpleDescriptionFormat_t zb_SimpleDesc =
{
  MY_ENDPOINT_ID,              // Endpoint, 0x02
  MY_PROFILE_ID,               // Profile ID, 0x0F10
  DEV_ID_CONTROLLER,           // Device ID, 2
  DEVICE_VERSION_CONTROLLER,   // Device Version, 1
  0,                           // Reserved
  NUM_IN_CMD_CONTROLLER,       // Number of Input Commands, 1
  (cId_t *) zb_InCmdList,      // Input Command List,
  NUM_OUT_CMD_CONTROLLER,      // Number of Output Commands, 0
  (cId_t *) NULL               // Output Command List,
};
```

- **绑定（binding）**

通过使用 ClusterID 为不同设备上的端点建立一个逻辑上的连接。绑定过程如下图：



➢ 分层协议标准



Request：上层向本层请求指定的服务；

Confirm：本层响应上层的请求；

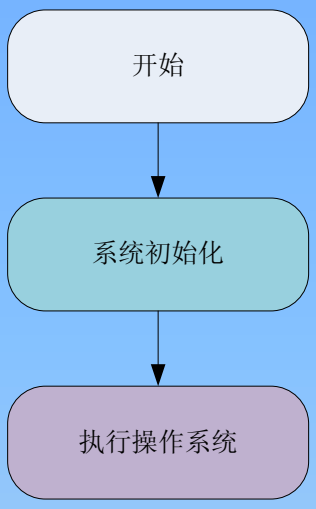
Indication：本层发送给上层用来指示本层的某一事件；

Response：上层响应本层的指示。

第二章 Z-Stack 体系架构

2.1 Z-Stack 软件架构

Z-Stack 由 main（）函数开始执行，main（）函数共做了 2 件事：一是系统初始化，另外一件是开始执行轮转查询式操作系统，如下图所示：



FS_Zstack\ZMain.c

```
ZSEG 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
  zmain_ram_init();               // 初始化堆栈
  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
  zmain_ext_addr();               // 确定 IEEE 地址
  zgInit();                       //初始化非易失变量
  ZMacInit();                     // 初始化 MAC 层
```

```
  osal_init_system();                  //初始化操作系统

  osal_int_enable( INTS_ALL );         //使能全部中断

  InitBoard( OB_READY );               //初始化按键

  zmain_dev_info();                    //显示设备信息

  osal_start_system();                 //执行操作系统

}
```

2.2 Z-Stack 操作系统初始化

OSAL.c

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

  osal_qHead = NULL;        // Initialize the message queue

  osalTimerInit();          // Initiaize the timers

  osal_pwrmgr_init();       // Initialize the Power Management System

  osalInitTasks();          //初始化系统的任务

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

  return ( ZSUCCESS );
}
```

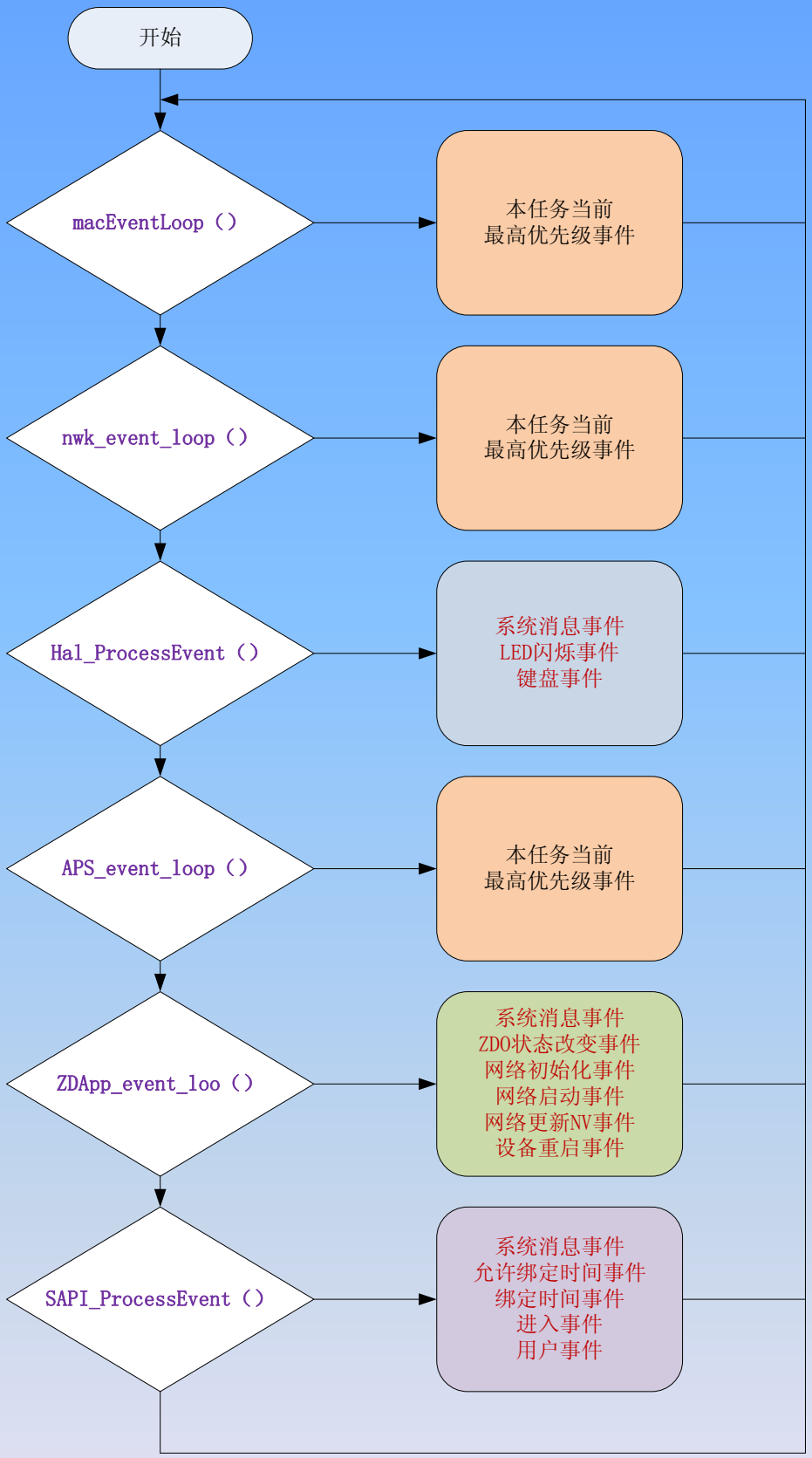
sapi.c

```
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
```

```
  APS_Init( taskID++ );              //APS_Init(3)，用户不需考虑
  ZDApp_Init( taskID++ );            //ZDApp_Init(4)，用户需考虑
  SAPI_Init( taskID );               //SAPI_Init(5)，用户需考虑
}
```

2.3 操作系统执行过程

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



详细执行过程如下：

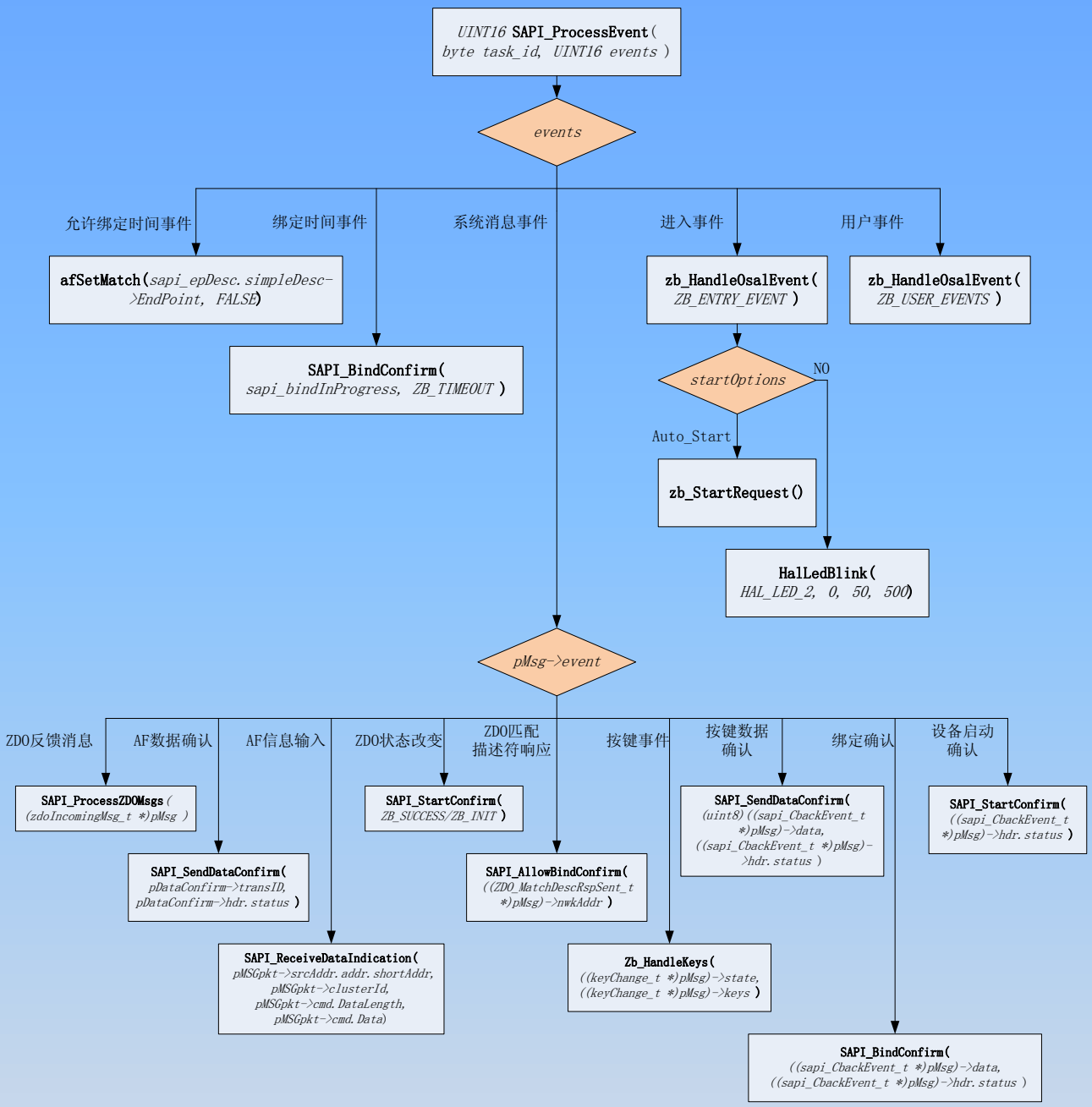
FS_Zstack\OSAL.c

```
void osal_start_system( void )
{
  for(;;)                                          // 死循环
  {
    do {
      if (tasksEvents[idx])
      {
        break;                                     // 得到待处理的最高优先级任务索引号 idx
      }
    } while (++idx < tasksCnt);

    if (idx < tasksCnt)
    {
      HAL_ENTER_CRITICAL_SECTION(intState);  //进入临界区
      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

      HAL_ENTER_CRITICAL_SECTION(intState);  //进入临界区
      tasksEvents[idx] |= events;            // 保存未处理的事件
      HAL_EXIT_CRITICAL_SECTION(intState);   //退出临界区
      }
    }
}
```

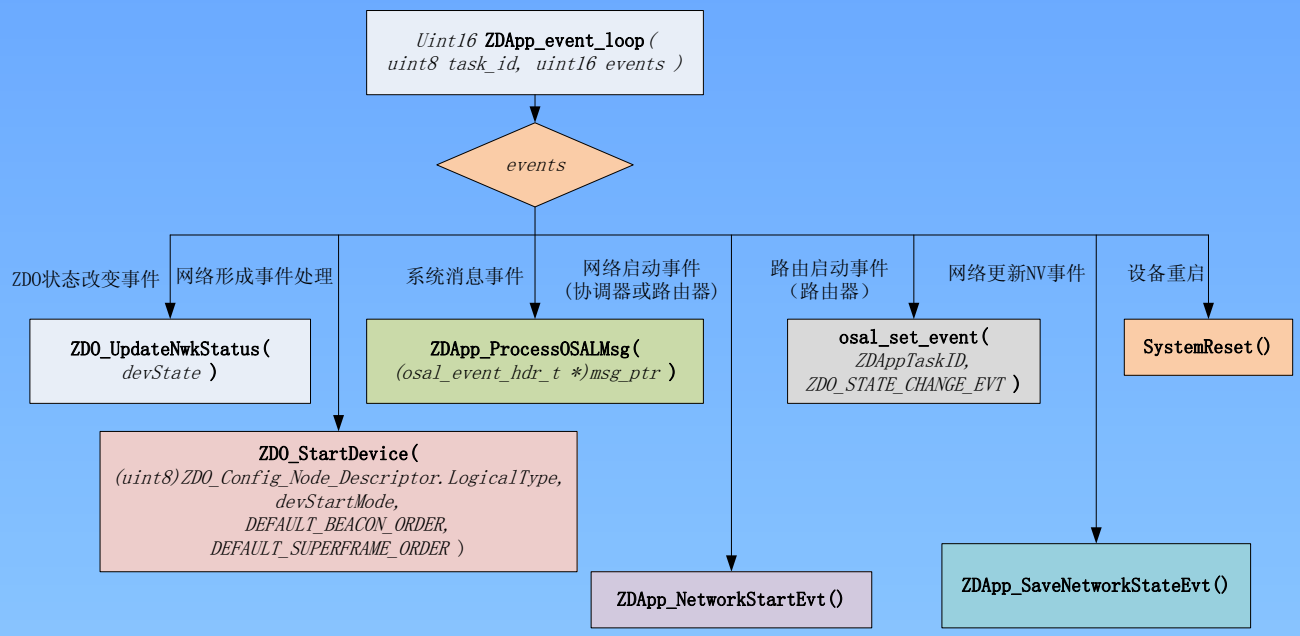
sapi.c

```
const pTaskEventHandlerFn tasksArr[] = {
  macEventLoop,                 //用户不需要考虑
  nwk_event_loop,               //用户不需要考虑
  Hal_ProcessEvent,             //用户可以考虑
  APS_event_loop,               //用户不需要考虑
  ZDApp_event_loop,             //用户可以考虑
  SAPI_ProcessEvent             //用户可以考虑
};
```

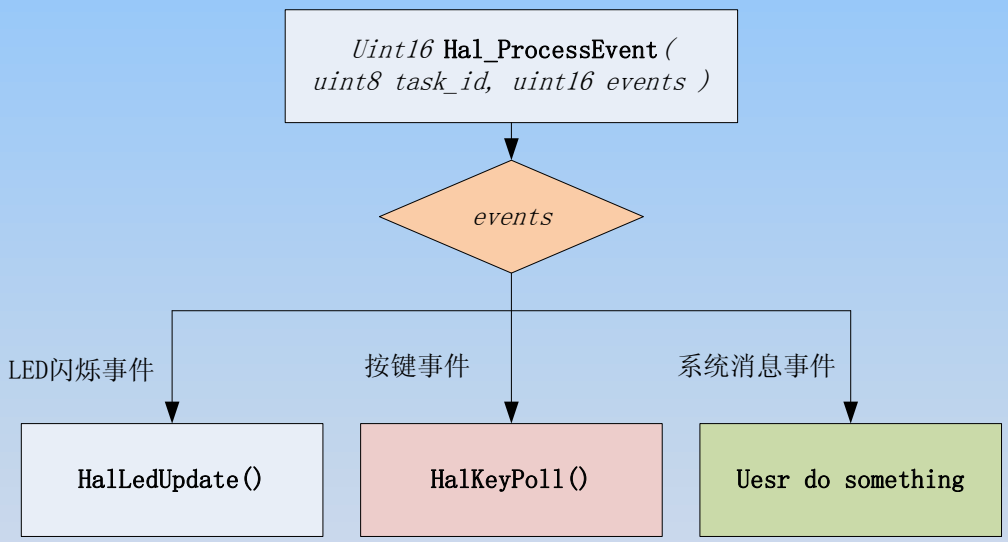
2.3.1 SAPI 任务事件处理函数



2.3.2 ZDApp 任务事件处理函数



2.3.3 Hal 任务事件处理函数



第三章 采集节点和传感节点通信分析

评估板 A 作为采集节点，上电启动后负责建立网络，并开启允许绑定功能；评估板 B 作为传感节点，上电启动后加入网络，并自动发起绑定请求，待采集节点建立绑定后，将采集到的温度和电压值发送到采集节点；评估板 A 的 LED5 闪烁表示温度数据正在发送，当评估板 B 接收到温度数据时，LED7 闪烁；评估板 A 的 LED6 闪烁表示电压数据正在发送，当评估板 B 接收到电压数据时，LED8 闪烁。评估板 A 连接电脑的串口，用户通过超级终端可以观察到温度和电压数据。

3.1 实验目的

- ✧ 掌握采集节点启动过程
- ✧ 了解采集节点建立网络过程
- ✧ 掌握传感节点启动过程
- ✧ 了解传感节点加入网络过程
- ✧ 掌握采集节点和传感节点绑定过程
- ✧ 掌握采集节点和传感节点数据传输过程

3.2 实验电路

电路参见：ZigBee Evaluation Board V1.0 原理图 LED 模块、按键、光敏电阻、电位器、射频模块。

采集节点和传感节点电路板设置如下表：

I/O 端口	跳线	采集节点	传感节点
P0_0	——	按键 S5	按键 S5
P0_1	——	按键 S6	按键 S6
P0_2	——	RXD0	RXD0
P0_3	——	TXD0	TXD0
P0_4	——	键盘 S2、S3、S4、S7	键盘 S2、S3、S4、S7

P0_5	SW13	——	——
P0_6	SW11	——	——
P0_7	SW12	——	——
P1_0	SW2	LED1/Motor(1)	LED1/Motor(1)
P1_1	SW3	LED2/Motor(2)	LED2/Motor(2)
P1_2	SW4	LED3/Motor(3)	LED3/Motor(3)
P1_3	SW5	LED4/Motor(4)	LED4/Motor(4)
P1_4	SW6	LED5	LED5
P1_5	SW7	LED6	LED6
P1_6	SW8	LED7	LED7
P1_7	SW9	LED8	LED8
P2_0	SW10	——	——
P2_1	——	DD	DD
P2_2	——	DC	DC

3.3 实验原理及代码

3.3.1 采集节点启动及建立网络

采集节点启动时首先进行任务初始化 osalInitTasks（），应用层的任务初始化工作有：分配任务 ID 号，设置绑定标志位（默认不允许绑定），初始化端点描述符，在 AF 层注册该端点描述符，默认关闭匹配描述符的响应，注册键盘事件，设置进入事件。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
void SAPI_Init( byte task_id )
{
  sapi_TaskID = task_id;                    //分配任务 ID，sapi_TaskID=5
  sapi_bindInProgress = 0xffff;             //绑定标志位，默认不允许绑定
  //初始化 sapi 层的端口描述符, SimpleCollector.c 中定义
  sapi_epDesc.endPoint = zb_SimpleDesc.EndPoint;
  sapi_epDesc.task_id = &sapi_TaskID;
```

```
  sapi_epDesc.simpleDesc = (SimpleDescriptionFormat_t *)&zb_SimpleDesc;
  sapi_epDesc.latencyReq = noLatencyReqs;

  //在 AF 层注册该端口描述符
  afRegister ( &sapi_epDesc );

  //默认关闭匹配描述符的响应
  afSetMatch(sapi_epDesc.simpleDesc->EndPoint, FALSE);

//sapi 层注册网络地址响应事件和匹配描述符响应事件，簇 ID=0x8000 和 簇 ID=0x8006
//在 sapi 层将使用函数 SAPI_ProcessZDOMsgs( (zdoIncomingMsg_t *)pMsg )
  ZDO_RegisterForZDOMsg( sapi_TaskID, NWK_addr_rsp );
  ZDO_RegisterForZDOMsg( sapi_TaskID, Match_Desc_rsp );

//注册（HAL）键盘事件
//在 sapi 层将使用函数 zb_HandleKeys( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys )
  RegisterForKeys( sapi_TaskID );

//为该任务设置事件-ZB_ENTRY_EVENT（0x1000）
  osal_set_event(task_id, ZB_ENTRY_EVENT);
}
```

采集节点首次启动时，NV 中的 StartOption 默认值为 0x0，因此不能建立网络，用户只看到 LED8 闪烁。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & ZB_ENTRY_EVENT )//0x1000，进入事件
  {
    //从设备的非易失性存储器读取配置信息
    zb_ReadConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );

    if ( startOptions & ZCD_STARTOPT_AUTO_START )//只有 NV 中的 startOptions=4 时，执行此项
    {
      zb_StartRequest();//重新启动设备后，建立网络
    }
    else//首次使用时，NV 中 startOptions=0，因此执行此项
    {
      HalLedBlink(HAL_LED_8, 8, 50, 500);
    }
    ……
  }
}
```

然后一直查询按键事件，当按下按键 S5 按下时，修改 StartOption 默认值为 0x4，并重启该协调器。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……
        case KEY_CHANGE://0xC0，按键事件
          zb_HandleKeys ( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys );
          break;
        ……
      }
    }
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleCollector.c

```
void zb_HandleKeys( uint8 shift, uint8 keys )
{
  if ( keys & HAL_KEY_SW_5 )
  {
    ……
    //将 logicalType 设置为协调器，并保存至 NV 中
    zb_ReadConfiguration( ZCD_NV_LOGICAL_TYPE, sizeof(uint8), &logicalType );
    logicalType = ZG_DEVICETYPE_COORDINATOR;
    zb_WriteConfiguration(ZCD_NV_LOGICAL_TYPE, sizeof(uint8), &logicalType);

    //修改 StartOptions 的值，并保存至 NV 中
    zb_ReadConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );
    startOptions = ZCD_STARTOPT_AUTO_START;
    zb_WriteConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );

    //重新启动协调器
    zb_SystemReset();
  }
  ……
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & ZB_ENTRY_EVENT )//0x1000，进入事件
  {
    zb_StartRequest();//重新启动设备，建立网络
    ……
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
void zb_StartRequest()
{
    //设备打开电源之后，由于未形成网络，经逻辑类型判断后，执行此处
    ZDOInitDevice(zgStartDelay);
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
uint8 ZDOInitDevice( uint16 startDelay )
{
    ……
    //开始形成网络
    ZDApp_NetworkInit ( extendedDelay );
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDApp_NetworkInit( uint16 delay )
{
    ……
    //ZDO_NETWORK_INIT 事件处理函数查看 ZDApp_event_loop()
    osal_start_timerEx ( ZDAppTaskID, ZDO_NETWORK_INIT, delay );
    ……
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  ……
  if ( events & ZDO_NETWORK_INIT )//网络初始化事件处理
  {
    // Initialize apps and start the network
    devState = DEV_INIT;
    //设备逻辑类型，启动模式，信标时间，超帧长度
    ZDO_StartDevice ( (uint8)ZDO_Config_Node_Descriptor.LogicalType, devStartMode,
                      DEFAULT_BEACON_ORDER, DEFAULT_SUPERFRAME_ORDER );
    HalLedSet (HAL_LED_1, HAL_LED_MODE_ON);
    ……
```

```
  }
……
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDObject.c

```
void ZDO_StartDevice( byte logicalType, devStartModes_t startMode, byte beaconOrder, byte superframeOrder )
{
  ……
  devState = DEV_COORD_STARTING;
  ret = NLME_NetworkFormationRequest( zgConfigPANID, zgDefaultChannelList,
    zgDefaultStartingScanDuration,beaconOrder,superframeOrder,false );
    //0xFFFF,0x00000800,5,0,0,false
    //在 f8wConfig.cfg 中 ZDAPP_CONFIG_PAN_ID=0xFFFF
    //在 f8wConfig.cfg 中 zgDefaultChannelList=0x00000800，即，通道 11
  ……
}
```

当网络层执行 NLME_NetworkFormationRequest（）建立网络后，将给予 ZDO 层反馈信息。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDO_NetworkFormationConfirmCB( ZStatus_t Status )
{
  HalLedSet ( HAL_LED_2, HAL_LED_MODE_ON );

    osal_set_event ( ZDAppTaskID, ZDO_NETWORK_START );
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
UINT16 ZDApp_event_loop ( byte task_id, UINT16 events )
{
……
  if ( events & ZDO_NETWORK_START )
  {
    ZDApp_NetworkStartEvt ();

    HalLedSet (HAL_LED_3, HAL_LED_MODE_ON);
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDApp_NetworkStartEvt( void )
{
    …….
    osal_pwrmgr_device( PWRMGR_ALWAYS_ON );
```

```
    osal_set_event( ZDAppTaskID, ZDO_STATE_CHANGE_EVT );
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
UINT16 ZDApp_event_loop ( byte task_id, UINT16 events )
{
  ……
  if ( events & ZDO_STATE_CHANGE_EVT )
  {
    ……
    ZDO_UpdateNwkStatus ( devState );
    HalLedSet (HAL_LED_4, HAL_LED_MODE_ON);
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDObject.c

```
void ZDO_UpdateNwkStatus( devStates_t state )
{
      ZDAppNwkAddr.addr.shortAddr = NLME_GetShortAddr();
      (void)NLME_GetExtAddr();    // Load the saveExtAddr pointer.

      msgPtr = (osal_event_hdr_t *)osal_msg_allocate( bufLen );

      if ( msgPtr )
      {
        msgPtr->event = ZDO_STATE_CHANGE; // Command ID
        msgPtr->status = (byte)state;

        //发送更新信息至注册过的应用端点
        osal_msg_send ( *(epDesc->epDesc->task_id), (byte *)msgPtr );
      }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
UINT16 SAPI_ProcessEvent ( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……

        case ZDO_STATE_CHANGE:
          SAPI_StartConfirm ( ZB_SUCCESS );
```

```
            HalLedSet (HAL_LED_6, HAL_LED_MODE_ON);
        ……
      }
    }
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
void SAPI_StartConfirm( uint8 status )
{
    ……
    HalLedSet (HAL_LED_5, HAL_LED_MODE_ON);
    zb_StartConfirm ( status );
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleCollector.c

```
void zb_StartConfirm( uint8 status )
{
  // If the device sucessfully started, change state to running
    myAppState = APP_START;
}
```

3.3.2 传感节点启动及加入网络

传感节点首次启动时，NV 中的 StartOption 默认值为 0x0，因此不能加入网络，用户只看到 LED8 闪烁。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
void SAPI_Init( byte task_id )
{
  sapi_TaskID = task_id;                          //分配任务 ID，sapi_TaskID=5
  sapi_bindInProgress = 0xffff;                   //绑定标志位，默认不允许绑定
  //初始化 sapi 层的端口描述符, SimpleSensorr.c 中定义
  sapi_epDesc.endPoint = zb_SimpleDesc.EndPoint;
  sapi_epDesc.task_id = &sapi_TaskID;
  sapi_epDesc.simpleDesc = (SimpleDescriptionFormat_t *)&zb_SimpleDesc;
  sapi_epDesc.latencyReq = noLatencyReqs;

  //在 AF 层注册该端口描述符
  afRegister ( &sapi_epDesc );

  //默认关闭匹配描述符的响应
  afSetMatch(sapi_epDesc.simpleDesc->EndPoint, FALSE);
```

```
  //注册响应事件，簇 ID=0x8000 和 簇 ID=0x8006
  ZDO_RegisterForZDOMsg( sapi_TaskID, NWK_addr_rsp );
  ZDO_RegisterForZDOMsg( sapi_TaskID, Match_Desc_rsp );

  //注册（HAL）键盘事件
  RegisterForKeys( sapi_TaskID );

  //为该任务设置事件-ZB_ENTRY_EVENT（0x1000）
  osal_set_event(task_id, ZB_ENTRY_EVENT);
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & ZB_ENTRY_EVENT )//0x1000，进入事件
  {
    //从设备的非易失性存储器读取配置信息
    zb_ReadConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );

    if ( startOptions & ZCD_STARTOPT_AUTO_START )//只有 NV 中的 startOptions=4 时，执行此项
    {
      zb_StartRequest();//重新启动设备后，建立网络
    }
    Else//首次使用时，NV 中 startOptions=0，因此执行此项
    {
      HalLedBlink(HAL_LED_8, 8, 50, 500);
    }
    ……
  }
}
```

然后一直查询按键事件，当按下按键 S5 按下时，修改 StartOption 默认值为 0x4，并重启该终端设备。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……
        case KEY_CHANGE://0xC0，按键事件
```

```
            zb_HandleKeys ( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys );
            break;
          ……
        }
      }
    }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleSensor.c

```
void zb_HandleKeys( uint8 shift, uint8 keys )
{
    if ( keys & HAL_KEY_SW_5 )
    {
        ……
        //将 logicalType 设置为终端设备，并保存至 NV 中
        logicalType = ZG_DEVICETYPE_ENDDEVICE;
        zb_WriteConfiguration(ZCD_NV_LOGICAL_TYPE, sizeof(uint8), &logicalType);

        //修改 StartOptions 的值，并保存至 NV 中
        zb_ReadConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );
        startOptions = ZCD_STARTOPT_AUTO_START;
        zb_WriteConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );

        //重新启动协调器
        zb_SystemReset();
    }
    ……
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & ZB_ENTRY_EVENT )//0x1000，进入事件
  {
      zb_StartRequest();//重新启动设备，建立网络
    ……
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
void zb_StartRequest()
{
    //设备打开电源之后，由于未形成网络，经逻辑类型判断后，执行此处
    ZDOInitDevice(zgStartDelay);
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
uint8 ZDOInitDevice( uint16 startDelay )
{
    ……
    //网络初始化
    ZDApp_NetworkInit ( extendedDelay );
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDApp_NetworkInit( uint16 delay )
{
    ……
    osal_start_timerEx ( ZDAppTaskID, ZDO_NETWORK_INIT, delay );
    ……
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  ……
  if ( events & ZDO_NETWORK_INIT )//网络初始化事件处理
  {
    // Initialize apps and start the network
    devState = DEV_INIT;
    //设备逻辑类型，启动模式，信标时间，超帧长度
    ZDO_StartDevice ( (uint8)ZDO_Config_Node_Descriptor.LogicalType, devStartMode,
                      DEFAULT_BEACON_ORDER, DEFAULT_SUPERFRAME_ORDER );
    HalLedSet (HAL_LED_1, HAL_LED_MODE_ON);
    ……
  }
……
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDObject.c

```
void ZDO_StartDevice( byte logicalType, devStartModes_t startMode, byte beaconOrder, byte
superframeOrder )
{
  ……
  devState = DEV_NWK_DISC;
  ret = NLME_NetworkDiscoveryRequest( zgDefaultChannelList, zgDefaultStartingScanDuration );
  // zgDefaultChannelList 与协调器形成网络的通道号匹配。
   ……
}
```

当发现有网络存在时，网络层将给予 ZDO 层发现网络反馈信息。然后由网络层发起加入网络请求，如加入网络成功，则网络层将给予 ZDO 层加入网络反馈

信息。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
ZStatus_t ZDO_NetworkDiscoveryConfirmCB( byte ResultCount, networkDesc_t *NetworkList )
{
      HalLedSet ( HAL_LED_2, HAL_LED_MODE_ON );
      ZDApp_SendMsg( ZDAppTaskID, ZDO_NWK_DISC_CNF,
                              sizeof(ZDO_NetworkDiscoveryCfm_t), (byte *)&msg );
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{
   switch ( msgPtr->event )
   {
      case ZDO_NWK_DISC_CNF:
      HalLedSet (HAL_LED_3,HAL_LED_MODE_ON);
      NLME_JoinRequest( ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->extendedPANID,
      BUILD_UINT16( ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->panIdLSB,
      ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->panIdMSB ),
      ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->logicalChannel,
      ZDO_Config_Node_Descriptor.CapabilityFlags )
   }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDO_JoinConfirmCB( uint16 PanId, ZStatus_t Status )
{
   HalLedSet (HAL_LED_5, HAL_LED_MODE_ON);
   ……
   ZDApp_SendMsg( ZDAppTaskID, ZDO_NWK_JOIN_IND, sizeof(osal_event_hdr_t), (byte*)NULL );
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{
   switch ( msgPtr->event )
   {
      ……
      case ZDO_NWK_JOIN_IND:
         ZDApp_ProcessNetworkJoin();
         break;
   }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDApp_ProcessNetworkJoin( void )
{
  ……
  osal_set_event( ZDAppTaskID, ZDO_STATE_CHANGE_EVT )
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  if ( events & ZDO_STATE_CHANGE_EVT )//ZDO 状态改变事件
  {
    ZDO_UpdateNwkStatus( devState );
    HalLedSet (HAL_LED_4, HAL_LED_MODE_ON);
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDObject.c

```
void ZDO_UpdateNwkStatus( devStates_t state )
{
        ……
        msgPtr->event = ZDO_STATE_CHANGE; // Command ID
        msgPtr->status = (byte)state;

        osal_msg_send( *(epDesc->epDesc->task_id), (byte *)msgPtr );
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
uint16 sapi_ProcessEvent( uint8 task_id, uint16 events )
{
        case ZDO_STATE_CHANGE:
          SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
          if ( (SampleApp_NwkState == DEV_ZB_COORD)
              || (SampleApp_NwkState == DEV_ROUTER)
              || (SampleApp_NwkState == DEV_END_DEVICE) )
          {
            SAPI_StartConfirm( ZB_SUCCESS );
            HalLedSet (HAL_LED_6, HAL_LED_MODE_ON);
          }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
void SAPI_StartConfirm( uint8 status )
{
    zb_StartConfirm( status );
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleSensor.c

```
void zb_StartConfirm( uint8 status )
{
  if ( status == ZB_SUCCESS )
  {
    myAppState = APP_START;
    //10ms 后进入绑定事件
    osal_start_timerEx( sapi_TaskID, MY_FIND_COLLECTOR_EVT, myBindRetryDelay );
  }
  else
  {
    osal_start_timerEx( sapi_TaskID, MY_START_EVT, myStartRetryDelay );
  }
}
```

3.3.3 采集节点允许绑定

在传感节点发送绑定请求之前，采集节点应做好准备工作，即，允许绑定。这里，通过按下采集节点的 S5 键进行允许绑定。当按下 S5 键后，所有 LED 熄灭，允许绑定完成。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……
        case KEY_CHANGE://0xC0，按键事件
          zb_HandleKeys ( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys );
          break;
        ……
      }
    }
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleCollctor.c

```
void zb_HandleKeys( uint8 shift, uint8 keys )
{
    if ( keys & HAL_KEY_SW_5 )
    {
```

```
        ……
        zb_AllowBind( 0xFF );
        HalLedSet( HAL_LED_ALL, HAL_LED_MODE_OFF );
        ……
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
void zb_AllowBind ( uint8 timeout )
{

    afSetMatch(sapi_epDesc.simpleDesc->EndPoint, TRUE);
}
```

3.3.4 传感节点发送绑定请求

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & ( ZB_USER_EVENTS ) )
  {
    zb_HandleOsalEvent( events );
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleSensor.c

```
void zb_HandleOsalEvent( uint16 event )
{
  ......
  if ( event & MY_FIND_COLLECTOR_EVT )//寻找并与一个采集节点建立绑定
  {
    // Find and bind to a collector device
    zb_BindDevice( TRUE, SENSOR_REPORT_CMD_ID, (uint8 *)NULL );
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapic

```
void zb_BindDevice ( uint8 create, uint16 commandId, uint8 *pDestination )
{
  uint8 ret = ZB_ALREADY_IN_PROGRESS;

  if ( create )//create=TRUE
  {
    if (sapi_bindInProgress == 0xffff)//函数SAPI_Init（）定义sapi_bindInProgress = 0xffff
    {
        ret = ZB_INVALID_PARAMETER;
        destination.addrMode = Addr16Bit;
        destination.addr.shortAddr = NWK_BROADCAST_SHORTADDR;//0xFFFF
```

```
        if ( ZDO_AnyClusterMatches( 1, &commandId, sapi_epDesc.simpleDesc->AppNumOutClusters,
                                                    sapi_epDesc.simpleDesc->pAppOutClusterList ) )
          {
            //匹配一个在允许绑定模式下的设备
            ret = ZDP_MatchDescReq( &destination, NWK_BROADCAST_SHORTADDR,
                sapi_epDesc.simpleDesc->AppProfId, 1, &commandId, 0, (cId_t *)NULL, 0 );
          }
      else if ( ZDO_AnyClusterMatches( 1, &commandId, sapi_epDesc.simpleDesc->AppNumInClusters,
                                                    sapi_epDesc.simpleDesc->pAppInClusterList ) )
          {
            ret = ZDP_MatchDescReq( &destination, NWK_BROADCAST_SHORTADDR,
                sapi_epDesc.simpleDesc->AppProfId, 0, (cId_t *)NULL, 1, &commandId, 0 );
          }

          if ( ret == ZB_SUCCESS )
          {
            HalLedSet (HAL_LED_ALL, HAL_LED_MODE_OFF);
            //设置一个时间，确保绑定完成
            osal_start_timerEx(sapi_TaskID, ZB_BIND_TIMER, AIB_MaxBindingTime);
            sapi_bindInProgress = commandId;//允许基于命令的绑定过程
          }
        }

      SAPI_SendCback( SAPICB_BIND_CNF, ret, commandId );
    }
}
```

3.3.5 采集节点处理绑定请求并发送绑定响应

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  uint8 *msg_ptr;

  if ( events & SYS_EVENT_MSG )//0x8000 系统消息事件
  {
    while ( (msg_ptr = osal_msg_receive( ZDAppTaskID )) )//接收任务号为 ZDAppTaskID 的命令信息
    {
      ZDApp_ProcessOSALMsg( (osal_event_hdr_t *)msg_ptr );//关于命令信息的事件处理函数
      .....
    }
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{
  switch ( msgPtr->event )
  {
    // Incoming ZDO Message
    case AF_INCOMING_MSG_CMD://AF 信息输入
      ZDP_IncomingData( (afIncomingMSGPacket_t *)msgPtr );
      break;
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDPofile.c

```
void ZDP_IncomingData( afIncomingMSGPacket_t *pData )
{
  uint8 x = 0;
  uint8 handled;
  zdoIncomingMsg_t inMsg;

  inMsg.srcAddr.addrMode = Addr16Bit;
  inMsg.srcAddr.addr.shortAddr = pData->srcAddr.addr.shortAddr;
  inMsg.wasBroadcast = pData->wasBroadcast;
  inMsg.clusterID = pData->clusterId;
  inMsg.SecurityUse = pData->SecurityUse;

  inMsg.asduLen = pData->cmd.DataLength-1;
  inMsg.asdu = pData->cmd.Data+1;
  inMsg.TransSeq = pData->cmd.Data[0];

  handled = ZDO_SendMsgCBs( &inMsg );

  while ( zdpMsgProcs[x].clusterID != 0xFFFF )
  {
    if ( zdpMsgProcs[x].clusterID == inMsg.clusterID )
    {
      zdpMsgProcs[x].pFn( &inMsg );
      return;
    }
    x++;
  }
.....
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDPofile.h

```
CONST zdpMsgProcItem_t zdpMsgProcs[] =
{
  { NWK_addr_req,              zdpProcessAddrReq },
```

```
  { IEEE_addr_req,              zdpProcessAddrReq },
  { Node_Desc_req,              ZDO_ProcessNodeDescReq },
  { Power_Desc_req,             ZDO_ProcessPowerDescReq },
  { Simple_Desc_req,            ZDO_ProcessSimpleDescReq },
  { Active_EP_req,              ZDO_ProcessActiveEPReq },
  { Match_Desc_req,             ZDO_ProcessMatchDescReq },
  ......
  {0xFFFF, NULL} // Last
};
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDObject.c

```
void ZDO_ProcessMatchDescReq( zdoIncomingMsg_t *inMsg )
{
    ......
    //ZDP_MatchDescRsp - Send an list of endpoint that match
    ZDP_MatchDescRsp(inMsg->TransSeq, &(inMsg->srcAddr), ZDP_SUCCESS,
                     ZDAppNwkAddr.addr.shortAddr, epCnt, (uint8 *)ZDOBuildBuf, inMsg->SecurityUse );
    return;
    ......
}
```

3.3.6 传感节点接收并处理绑定响应

由于 sapi 层在初始化时注册了匹配描述符的响应事件，因此在 sapi 层处理匹配描述符的响应事件。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000 系统消息事件
  {
    pMsg = (osal_event_hdr_t *) osal_msg_receive( task_id );
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        case ZDO_CB_MSG://0xD3，ZDO 的反馈消息
          SAPI_ProcessZDOMsgs( (zdoIncomingMsg_t *)pMsg );
          break;
      }
    }
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
void SAPI_ProcessZDOMsgs( zdoIncomingMsg_t *inMsg )
```

```
{
  switch ( inMsg->clusterID )
  {
    ......
    case Match_Desc_rsp://匹配消息的响应信息
      {
        HalLedSet( HAL_LED_8, HAL_LED_MODE_ON );
        if ( sapi_bindInProgress != 0xffff )
        {
          // Create a binding table entry 建立绑定表
          dstAddr.addrMode = Addr16Bit;
          dstAddr.addr.shortAddr = pRsp->nwkAddr;

          if ( APSME_BindRequest( sapi_epDesc.simpleDesc->EndPoint,
                                  sapi_bindInProgress, &dstAddr, pRsp->epList[0] ) == ZSuccess )
          {
            osal_stop_timerEx(sapi_TaskID,   ZB_BIND_TIMER);
            osal_start_timerEx( ZDAppTaskID, ZDO_NWK_UPDATE_NV, 250 );// xNV_RESTORE
            sapi_bindInProgress = 0xffff;

            //查找 IEEE 地址
            ZDP_IEEEAddrReq( pRsp->nwkAddr, ZDP_ADDR_REQTYPE_SINGLE, 0, 0 );

            //向应用程序发送绑定确认消息
            zb_BindConfirm( sapi_bindInProgress, ZB_SUCCESS );
          }
        }
      }
      break;
  }
}
```

建立绑定表时，只知道绑定节点（即采集节点）的 16 位网络地址和端点号，还需查找绑定节点的 64 位 IEEE 地址才可以与绑定节点建立通讯连接。因此，向绑定节点发送 ClusterID=IEEE_addr_req 的查找 IEEE 地址请求。当绑定节点接收到 IEEE 地址请求时，将进行处理并向传感节点发送 IEEE 地址响应。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  uint8 *msg_ptr;

  if ( events & SYS_EVENT_MSG )//0x8000 系统消息事件
  {
```

```
    while ( (msg_ptr = osal_msg_receive( ZDAppTaskID )) )//接收任务号为 ZDAppTaskID 的命令信息
    {
      ZDApp_ProcessOSALMsg( (osal_event_hdr_t *)msg_ptr );//关于命令信息的事件处理函数
      ......
    }
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{
  switch ( msgPtr->event )
  {
    // Incoming ZDO Message
    case AF_INCOMING_MSG_CMD://AF 信息输入
      ZDP_IncomingData( (afIncomingMSGPacket_t *)msgPtr );
      break;
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDPofile.c

```
void ZDP_IncomingData( afIncomingMSGPacket_t *pData )
{
  uint8 x = 0;
  uint8 handled;
  zdoIncomingMsg_t inMsg;

  inMsg.srcAddr.addrMode = Addr16Bit;
  inMsg.srcAddr.addr.shortAddr = pData->srcAddr.addr.shortAddr;
  inMsg.wasBroadcast = pData->wasBroadcast;
  inMsg.clusterID = pData->clusterId;
  inMsg.SecurityUse = pData->SecurityUse;

  inMsg.asduLen = pData->cmd.DataLength-1;
  inMsg.asdu = pData->cmd.Data+1;
  inMsg.TransSeq = pData->cmd.Data[0];

  handled = ZDO_SendMsgCBs( &inMsg );

  while ( zdpMsgProcs[x].clusterID != 0xFFFF )
  {
    if ( zdpMsgProcs[x].clusterID == inMsg.clusterID )
    {
      zdpMsgProcs[x].pFn( &inMsg );
      return;
    }
```

```
    x++;
  }
.....
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDPofile.h

```
CONST zdpMsgProcItem_t zdpMsgProcs[] =
{
  { NWK_addr_req,              zdpProcessAddrReq },
  { IEEE_addr_req,             zdpProcessAddrReq },
  { Node_Desc_req,             ZDO_ProcessNodeDescReq },
  { Power_Desc_req,            ZDO_ProcessPowerDescReq },
  { Simple_Desc_req,           ZDO_ProcessSimpleDescReq },
  { Active_EP_req,             ZDO_ProcessActiveEPReq },
  { Match_Desc_req,            ZDO_ProcessMatchDescReq },
  .....
  {0xFFFF, NULL} // Last
};
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDPofile.c

```
void zdpProcessAddrReq( zdoIncomingMsg_t *inMsg )
{
    aoi = BUILD_UINT16( inMsg->asdu[0], inMsg->asdu[1] );

    if ( aoi == ZDAppNwkAddr.addr.shortAddr )
    {
       ieee = saveExtAddr;
    }

  if ( (aoi != INVALID_NODE_ADDR) && (ieee != NULL) )
  {
    byte *pBuf = ZDP_TmpBuf;

    byte len = 1 + Z_EXTADDR_LEN + 2;

    byte stat = ((reqType == ZDP_ADDR_REQTYPE_SINGLE)
                 || (reqType == ZDP_ADDR_REQTYPE_EXTENDED)
              || ((reqType == ZDP_ADDR_REQTYPE_MEMBERSHIP)
            && (inMsg->clusterID == NWK_addr_req)) ) ? ZDP_SUCCESS : ZDP_INVALID_REQTYPE;

    *pBuf++ = stat;

    pBuf = osal_cpyExtAddr( pBuf, ieee );

    *pBuf++ = LO_UINT16( aoi );
    *pBuf++ = HI_UINT16( aoi );
```

```
    ZDP_TxOptions = AF_MSG_ACK_REQUEST;
    fillAndSend( &(inMsg->TransSeq), &(inMsg->srcAddr),
                 (cId_t)(inMsg->clusterID | ZDO_RESPONSE_BIT), len );
    ZDP_TxOptions = AF_TX_OPTIONS_NONE;
  }
}
```

由于在 sapi 层只注册了网络地址响应，而未注册 IEEE 地址响应，因此 sapi 层将不对 IEEE 地址响应作出处理，而是在 ZDO 层作出 IEEE 地址响应处理。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{

  switch ( msgPtr->event )
  {
      ……
      case ZDO_CB_MSG://ZDO 反馈信息
      ZDApp_ProcessMsgCBs( (zdoIncomingMsg_t *)msgPtr );
      break;
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\ZDApp.c

```
void ZDApp_ProcessMsgCBs( zdoIncomingMsg_t *inMsg )
{
  switch ( inMsg->clusterID )
  {
    case IEEE_addr_rsp:
      {
        HalLedSet( HAL_LED_7, HAL_LED_MODE_ON );
        ZDO_NwkIEEEAddrResp_t *pAddrRsp;
        pAddrRsp = ZDO_ParseAddrRsp( inMsg );
        if ( pAddrRsp )
        {
          if ( pAddrRsp->status == ZSuccess )
          {
            ZDO_UpdateAddrManager( pAddrRsp->nwkAddr, pAddrRsp->extAddr );
          }
          osal_mem_free( pAddrRsp );
        }
      }
      break;
  }
}
```

3.3.7 传感节点发送数据

当传感节点与采集节点建立绑定连接后，将自动进入数据采集并发送至采集节点。

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleSensor.c

```
void zb_BindConfirm( uint16 commandId, uint8 status )
{

  if ( ( status == ZB_SUCCESS ) && ( myAppState == APP_START ) )
  {
    myAppState = APP_BOUND;
    //开始向采集节点发送传感数据
    myApp_StartReporting();
  }
  else
  {
    //如果不能和一个采集节点建立绑定，重新搜索采集节点
    osal_start_timerEx( sapi_TaskID, MY_FIND_COLLECTOR_EVT, myBindRetryDelay );
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleSensor.c

```
void myApp_StartReporting( void )
{
    //5s 发送一次温度信息
    osal_start_timerEx( sapi_TaskID, MY_REPORT_TEMP_EVT, myTempReportPeriod );
    //21s 发送一次电压信息
    osal_start_timerEx( sapi_TaskID, MY_REPORT_BATT_EVT, myBatteryCheckPeriod );
 }
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & ( ZB_USER_EVENTS ) )
  {
    // User events are passed to the application
    zb_HandleOsalEvent( events );
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleSensor.c

```
void zb_HandleOsalEvent( uint16 event )
{
  .....
if ( event & MY_REPORT_TEMP_EVT )//读取并发送温度信息
  {
```

```
      // Read and report temperature value
      pData[0] = TEMP_REPORT;//0x01
      pData[1] =    myApp_ReadLight();
      zb_SendDataRequest( 0xFFFE, SENSOR_REPORT_CMD_ID, 2, pData, 0, AF_ACK_REQUEST,
0 );
      HalLedSet( HAL_LED_5, HAL_LED_MODE_TOGGLE );
      osal_start_timerEx( sapi_TaskID, MY_REPORT_TEMP_EVT, myTempReportPeriod );
    }

    if ( event & MY_REPORT_BATT_EVT )//读取并发送电压值
    {
      // Read battery value
      pData[0] = BATTERY_REPORT;
      pData[1] =    myApp_ReadBattery();
      zb_SendDataRequest( 0xFFFE, SENSOR_REPORT_CMD_ID, 2, pData, 0, AF_ACK_REQUEST,
0 );
      HalLedSet( HAL_LED_6, HAL_LED_MODE_TOGGLE );
      osal_start_timerEx( sapi_TaskID, MY_REPORT_BATT_EVT, myBatteryCheckPeriod );
    }
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleSensor.c

```
uint8 myApp_ReadLight( void )
{
    uint16 value;
    /* Clear ADC interrupt flag */
    ADCIF = 0;
    ADCCON3 = (HAL_ADC_REF_125V | HAL_ADC_DEC_512 | HAL_ADC_CHN_TEMP);
    /* Wait for the conversion to finish */
    while ( !ADCIF );
    /* Get the result */
    value = ADCL;
    value |= ((uint16) ADCH) << 8;
    /*
      * value ranges from 0 to 0x8000 indicating 0V and 1.25V
      * VOLTAGE_AT_TEMP_ZERO = 0.743 V = 19477
      * TEMP_COEFFICIENT = 0.0024 V/C = 62.9 /C
      * These parameters are typical values and need to be calibrated
      * See the datasheet for the appropriate chip for more details
      * also, the math below may not be very accurate
      */
#define VOLTAGE_AT_TEMP_ZERO          19477     // 0.743 V
#define TEMP_COEFFICIENT              62.9      // 0.0024 V/C
    // limit min temp to 0 C
    if ( value < VOLTAGE_AT_TEMP_ZERO )
      value = VOLTAGE_AT_TEMP_ZERO;
```

```
  value = value - VOLTAGE_AT_TEMP_ZERO;
  // limit max temp to 99 C
  if ( value > TEMP_COEFFICIENT * 99 )
    value = TEMP_COEFFICIENT * 99;

  return ( (uint8)(value/TEMP_COEFFICIENT) );
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleSensor.c

```
uint8 myApp_ReadBattery( void )
{
  uint16 value;

  /* Clear ADC interrupt flag */
  ADCIF = 0;
  ADCCON3 = (HAL_ADC_REF_125V | HAL_ADC_DEC_128 | HAL_ADC_CHN_VDD3);
  /* Wait for the conversion to finish */
  while ( !ADCIF );
  /* Get the result */
  value = ADCL;
  value |= ((uint16) ADCH) << 8;
  /*
    * value now contains measurement of Vdd/3
    * 0 indicates 0V and 32767 indicates 1.25V
    * voltage = (value*3*1.25)/32767 volts
    * we will multiply by this by 10 to allow units of 0.1 volts
    */
  value = value >> 6;     // divide first by 2^6
  value = value * 37.5;
  value = value >> 9;     // ...and later by 2^9...to prevent overflow during multiplication

  return value;
}
```

3.3.8 采集节点接收数据

当采集节点接收到数据后，将触发 SYS_EVENT_MSG 事件，并对 AF_INCOMING_MSG_CMD 信息做处理：

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
INT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{

  if ( events & SYS_EVENT_MSG )//0x8000 系统消息事件
```

```
  {
    pMsg = (osal_event_hdr_t *) osal_msg_receive( task_id );
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        case AF_INCOMING_MSG_CMD://0x1A，AF 信息输入
        pMSGpkt = (afIncomingMSGPacket_t *) pMsg;
        SAPI_ReceiveDataIndication( pMSGpkt->srcAddr.addr.shortAddr, pMSGpkt->clusterId,
        pMSGpkt->cmd.DataLength, pMSGpkt->cmd.Data);}
    }
  }
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\sapi.c

```
void SAPI_ReceiveDataIndication( uint16 source, uint16 command, uint16 len, uint8 *pData   )
{
    zb_ReceiveDataIndication( source, command, len, pData   );
}
```

FS_ZStack\Projects\zstack\Samples\Collector_Sensor\CC2430DB\SimpleCollector.c

```
void zb_ReceiveDataIndication( uint16 source, uint16 command, uint16 len, uint8 *pData   )
{
  if (command == SENSOR_REPORT_CMD_ID)
  {
    // Received report from a sensor
    sensorReading = pData[1];//从传感节点接收到数据

    // If tool available, write to serial port
    //如果设备允许，将数据写入串口
    tmpLen = (uint8)osal_strlen( (char*)strDevice );
    pBuf = osal_memcpy( buf, strDevice, tmpLen );
    _ltoa( source, pBuf, 16 );//把长整形数转换为字符串的函数
    pBuf += 4;
    *pBuf++ = ' ';

    if ( pData[0] == BATTERY_REPORT )
    {
      tmpLen = (uint8)osal_strlen( (char*)strBattery );
      pBuf = osal_memcpy( pBuf, strBattery, tmpLen );

      *pBuf++ = sensorReading + '0';              // convent to ascii
      *pBuf++ = ' ';
      HalLedSet( HAL_LED_7, HAL_LED_MODE_TOGGLE );
    }
    else
```

```
    {
        tmpLen = (uint8)osal_strlen( (char*)strTemp );
        pBuf = osal_memcpy( pBuf, strTemp, tmpLen );

        *pBuf++ = sensorReading + '0';          // convent to ascii
        *pBuf++ = ' ';
        HalLedSet( HAL_LED_8, HAL_LED_MODE_TOGGLE );
    }

    *pBuf++ = '\r';
    *pBuf++ = '\n';
    *pBuf = '\0';

#if defined( MT_TASK )
    debug_str( (uint8 *)buf );
#endif

  }
}
```

3.4 实验步骤及演示

步骤	操作	现象
1	串口线连接评估板 P1 与电脑	串口线连接
2	SW2~SW9 左方跳线，P1_0~P1_7 连接 LED1~LED8	LED 跳线连接
3	10 芯 HEADER 连接仿真器	准备下载程序
4	CON 连接 5V 适配器，按下 U10	上电，D2 点亮
5	仿真器 USB 线连接电脑	仿真器 LED 点亮
评估板 A		
6	通过 IAR 软件下载 SimpleCollectorEB 至评估板 A	评估板 A 实现采集节点功能
7	按下评估板 A 的 S1，重启	LED8 闪烁一段时间后熄灭
8	按下评估板 A 的 S5，建立网络请求	LED1 点亮
9	自动获得协议栈的建立网路反馈	LED2 点亮，设置 ZDO_NETWORK_START
10	网络自动启动	LED3 点亮，设置 ZDO_STATE_CHANGE_EVT 事件
11	启动 ZDO 状态改变	LED4 点亮，发送 ZDO_STATE_CHANGE 系统消息
12	SAPI 启动确认	LED5 点亮，设置 MY_START_EVT 事件
13	用户事件	等待按下 S5，允许绑定
评估板 B		
14	通过 IAR 软件下载 SimpleSensorEB 至评估板 B	评估板 B 实现传感节点功能
15	按下评估板 B 的 S1，重启	LED8 闪烁一段时间后熄灭
8	按下评估板 A 的 S5，发现网络请求	LED1 点亮，
9	自动获得协议栈的发现网路反馈	LED2 点亮，发送 ZDO_NWK_DISC_CNF 系统消息
10	自动发送加入网络请求	LED3 点亮，

11	自动获得协议栈的加入网路反馈	LED5 点亮，发送 ZDO_NWK_JOIN_IND 系统消息
12	启动 ZDO 状态改变	LED4 点亮，发送 ZDO_STATE_CHANGE 系统消息
13	SAPI 启动确认	LED6 点亮，设置 MY_FIND_COLLECTOR_EVT 事件
14	用户事件	开始发送绑定请求
评估板 A		
15	按下评估板 A 的 S5，允许绑定	所有 LED 熄灭
评估板 B		
16	自动发送匹配描述符请求	所有 LED 熄灭
17	处理匹配描述符的响应，发送绑定确认给 sapi	LED8 点亮
18	处理 IEEE 地址响应，发送地址响应确认给 ZDO	LED7 点亮
19	开始发送数据至评估板 A	LED5~LED6 闪烁
评估板 A		
20	接收来自评估板 B 的数据	LED7~LED8 闪烁，串口打印数据

注：PC 端使用串口打印采集数据，其串口设置参数如下：

波特率：38400

校验位：NONE

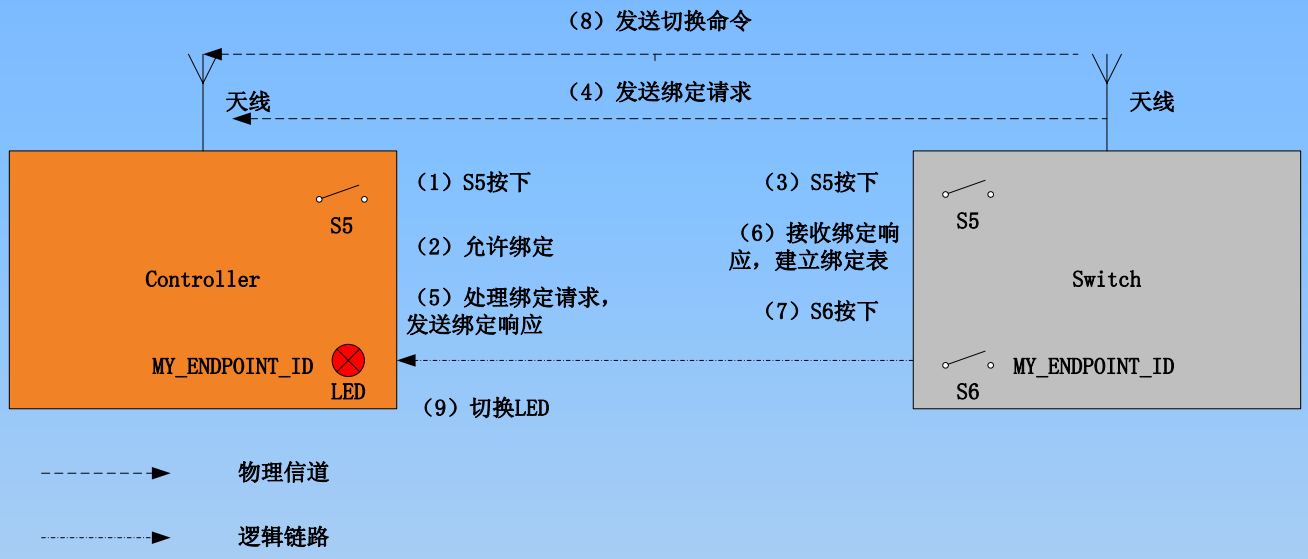
数据位：8

停止位：1

第四章 控制节点和开关节点通信分析

评估板 A 作为控制节点，上电启动后负责建立网络，并开启允许绑定功能；评估板 B 作为开关节点，上电启动后加入网络，并自动发起绑定请求，待控制节点建立绑定后，发送开关命令切换控制节点的 LED 亮灭。

控制节点（Controller）与开关节点（Switch）之间通信首先需要进行绑定操作，Controller 与 Switch 之间的绑定过程和数据传输过程如下图所示：



4.1 实验目的

- ✧ 掌握控制节点启动过程
- ✧ 了解控制节点建立网络过程
- ✧ 掌握开关节点启动过程
- ✧ 了解开关节点加入网络过程
- ✧ 掌握控制节点和开关节点绑定过程
- ✧ 掌握控制节点和开关节点数据传输过程

4.2 实验电路

电路参见：ZigBee Evaluation Board V1.0 原理图 LED 模块、按键、蜂鸣器。

控制节点和开关节点电路板设置如下表：

I/O 端口	跳线	控制节点	开关节点
P0_0	——	按键 S5	按键 S5
P0_1	——	按键 S6	按键 S6
P0_2	——	RXD0	RXD0
P0_3	——	TXD0	TXD0
P0_4	——	键盘 S2、S3、S4、S7	键盘 S2、S3、S4、S7
P0_5	SW13	——	——
P0_6	SW11	——	——
P0_7	SW12	——	——
P1_0	SW2	LED1/Motor(1)	LED1/Motor(1)
P1_1	SW3	LED2/Motor(2)	LED2/Motor(2)
P1_2	SW4	LED3/Motor(3)	LED3/Motor(3)
P1_3	SW5	LED4/Motor(4)	LED4/Motor(4)
P1_4	SW6	LED5	LED5
P1_5	SW7	LED6	LED6
P1_6	SW8	LED7	LED7
P1_7	SW9	LED8	LED8
P2_0	SW10	——	——
P2_1	——	DD	DD
P2_2	——	DC	DC

4.3 实验原理及代码

4.3.1 控制节点启动及建立网络

控制节点启动时首先进行任务初始化，主要工作有：分配任务 ID 号，设置绑定标志位（默认不允许绑定），初始化端点描述符，在 AF 层注册该端点描述符，默认关闭匹配描述符的响应，注册键盘事件，设置进入事件。

FS_ZStack\sapi.c

```
void SAPI_Init( byte task_id )
{
  sapi_TaskID = task_id;                          //分配任务 ID，sapi_TaskID=5
  sapi_bindInProgress = 0xffff;                   //绑定标志位，默认不允许绑定
  //初始化 sapi 层的端口描述符, SimpleCollector.c 中定义
  sapi_epDesc.endPoint = zb_SimpleDesc.EndPoint;
  sapi_epDesc.task_id = &sapi_TaskID;
  sapi_epDesc.simpleDesc = (SimpleDescriptionFormat_t *)&zb_SimpleDesc;
  sapi_epDesc.latencyReq = noLatencyReqs;

  //在 AF 层注册该端口描述符
  afRegister ( &sapi_epDesc );

  //默认关闭匹配描述符的响应
  afSetMatch(sapi_epDesc.simpleDesc->EndPoint, FALSE);

  //注册返回事件，簇 ID=0x8000 和 簇 ID=0x8006
  ZDO_RegisterForZDOMsg( sapi_TaskID, NWK_addr_rsp );
  ZDO_RegisterForZDOMsg( sapi_TaskID, Match_Desc_rsp );

  //注册（HAL）键盘事件
  RegisterForKeys( sapi_TaskID );

  //为该任务设置事件-ZB_ENTRY_EVENT（0x1000）
  osal_set_event(task_id, ZB_ENTRY_EVENT);
}
```

采集节点首次启动时，NV 中的 StartOption 默认值为 0x0，因此不能建立网络，用户只看到 LED8 闪烁。然后一直查询按键事件，当按下按键 S5 按下时，修改 StartOption 默认值为 0x4，并重启该协调器。直至最后，通过调用函数 NLME_NetworkDiscoveryRequest（）建立网络：

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & ZB_ENTRY_EVENT )//0x1000，进入事件
  {
    //从设备的非易失性存储器读取配置信息
    zb_ReadConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );

    if ( startOptions & ZCD_STARTOPT_AUTO_START )//只有 NV 中的 startOptions=4 时，执行此项
    {
```

```
        zb_StartRequest();//重新启动设备后，建立网络
      }
      else//首次使用时，NV 中 startOptions=0，因此执行此项
      {
        HalLedBlink(HAL_LED_8, 8, 50, 500);
      }
      ……
    }
  }
```

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……
        case KEY_CHANGE://0xC0，按键事件
          zb_HandleKeys ( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys );
          break;
        ……
      }
    }
  }
}
```

SimpleController.c

```
void zb_HandleKeys( uint8 shift, uint8 keys )
{
    if ( keys & HAL_KEY_SW_5 )
    {
        ……
        //将 logicalType 设置为协调器，并保存至 NV 中
         zb_ReadConfiguration( ZCD_NV_LOGICAL_TYPE, sizeof(uint8), &logicalType );
         logicalType = ZG_DEVICETYPE_COORDINATOR;
         zb_WriteConfiguration(ZCD_NV_LOGICAL_TYPE, sizeof(uint8), &logicalType);

        //修改 StartOptions 的值，并保存至 NV 中
        zb_ReadConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );
        startOptions = ZCD_STARTOPT_AUTO_START;
        zb_WriteConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );

        //重新启动协调器
```

```
            zb_SystemReset();
        }
        ……
}
```

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & ZB_ENTRY_EVENT )//0x1000，进入事件
  {
        zb_StartRequest();//重新启动设备，建立网络
      ……
  }
}
```

sapi.c

```
void zb_StartRequest()
{
    //设备打开电源之后，由于未形成网络，经逻辑类型判断后，执行此处
    ZDOInitDevice(zgStartDelay);
}
```

ZDApp.c

```
uint8 ZDOInitDevice( uint16 startDelay )
{
    ……
    //开始形成网络
    ZDApp_NetworkInit ( extendedDelay );
}
```

ZDApp.c

```
void ZDApp_NetworkInit( uint16 delay )
{
    ……
    osal_start_timerEx ( ZDAppTaskID, ZDO_NETWORK_INIT, delay );
    ……
}
```

ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  ……
  if ( events & ZDO_NETWORK_INIT )//网络初始化事件处理
  {
    // Initialize apps and start the network
    devState = DEV_INIT;
    //设备逻辑类型，启动模式，信标时间，超帧长度
```

```
    ZDO_StartDevice ( (uint8)ZDO_Config_Node_Descriptor.LogicalType, devStartMode,
                      DEFAULT_BEACON_ORDER, DEFAULT_SUPERFRAME_ORDER );
    ……
  }
……
}
```

ZDObject.c

```
void ZDO_StartDevice( byte logicalType, devStartModes_t startMode, byte beaconOrder, byte superframeOrder )
{
  ……
  devState = DEV_COORD_STARTING;
  ret = NLME_NetworkFormationRequest( zgConfigPANID, zgDefaultChannelList,
  zgDefaultStartingScanDuration,beaconOrder,superframeOrder,
false );//0xFFFF,0x00000800,5,0,0,false
  ……
}
```

当网络层执行 NLME_NetworkFormationRequest（）建立网络后，将给予 ZDO 层反馈信息：

ZDApp.c

```
void ZDO_NetworkFormationConfirmCB( ZStatus_t Status )
{
#if defined(ZDO_COORDINATOR)
  osal_set_event ( ZDAppTaskID, ZDO_NETWORK_START );
#endif  //ZDO_COORDINATOR
}
```

ZDApp.c

```
UINT16 ZDApp_event_loop ( byte task_id, UINT16 events )
{
……
#if defined (RTR_NWK)//f8wcoord.cfg 中预编译
  if ( events & ZDO_NETWORK_START )
  {
    ZDApp_NetworkStartEvt ();
  }
#endif  //RTR_NWK
}
```

ZDApp.c

```
void ZDApp_NetworkStartEvt( void )
{
    …….
```

```
    osal_pwrmgr_device( PWRMGR_ALWAYS_ON );
    osal_set_event( ZDAppTaskID, ZDO_STATE_CHANGE_EVT );
}
```

ZDApp.c

```
UINT16 ZDApp_event_loop ( byte task_id, UINT16 events )
{
  ……
  if ( events & ZDO_STATE_CHANGE_EVT )
  {
    ……
    ZDO_UpdateNwkStatus ( devState );
  }
}
```

ZDObject.c

```
void ZDO_UpdateNwkStatus( devStates_t state )
{
      msgPtr = (osal_event_hdr_t *)osal_msg_allocate( bufLen );

      if ( msgPtr )
      {
        msgPtr->event = ZDO_STATE_CHANGE; // Command ID
        msgPtr->status = (byte)state;

        osal_msg_send ( *(epDesc->epDesc->task_id), (byte *)msgPtr );
      }
}
```

sapi.c

```
UINT16 SAPI_ProcessEvent ( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……
        case ZDO_STATE_CHANGE:
            SAPI_StartConfirm ( ZB_SUCCESS );
        ……
      }
    }
  }
}
```

sapi.c

```
void SAPI_StartConfirm( uint8 status )
{
    ……
    zb_StartConfirm ( status );
}
```

SimpleController.c

```
void zb_StartConfirm( uint8 status )
{
  // If the device sucessfully started, change state to running
    myAppState = APP_START;
}
```

4.3.2 开关节点启动及加入网络

开关节点首次启动时，NV 中的 StartOption 默认值为 0x0，因此不能加入网络，用户只看到所有 LED 闪烁。然后一直查询按键事件，当按下按键 S5 按下时，修改 StartOption 默认值为 0x4，并重启该终端设备。直至最后，通过调用函数 NLME_NetworkFormationRequest（）加入网络：

FS_ZStack\sapi.c

```
void SAPI_Init( byte task_id )
{
  uint8 startOptions;

  sapi_TaskID = task_id;                                    //分配任务 ID，sapi_TaskID=5
  sapi_bindInProgress = 0xffff;                             //绑定标志位，默认不允许绑定
  //初始化 sapi 层的端口描述符, SimpleSensorr.c 中定义
  sapi_epDesc.endPoint = zb_SimpleDesc.EndPoint;
  sapi_epDesc.task_id = &sapi_TaskID;
  sapi_epDesc.simpleDesc = (SimpleDescriptionFormat_t *)&zb_SimpleDesc;
  sapi_epDesc.latencyReq = noLatencyReqs;

  //在 AF 层注册该端口描述符
  afRegister ( &sapi_epDesc );

  //默认关闭匹配描述符的响应
  afSetMatch(sapi_epDesc.simpleDesc->EndPoint, FALSE);

  //注册返回事件，簇 ID=0x8000 和 簇 ID=0x8006
  ZDO_RegisterForZDOMsg( sapi_TaskID, NWK_addr_rsp );
  ZDO_RegisterForZDOMsg( sapi_TaskID, Match_Desc_rsp );
```

```
  //注册（HAL）键盘事件
  RegisterForKeys( sapi_TaskID );

  //为该任务设置事件-ZB_ENTRY_EVENT（0x1000）
  osal_set_event(task_id, ZB_ENTRY_EVENT);
}
```

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & ZB_ENTRY_EVENT )//0x1000，进入事件
  {
    //从设备的非易失性存储器读取配置信息
    zb_ReadConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );

    if ( startOptions & ZCD_STARTOPT_AUTO_START )//只有NV 中的startOptions=4 时，执行此项
    {
      zb_StartRequest();//重新启动设备后，建立网络
    }
    Else//首次使用时，NV 中startOptions=0，因此执行此项
    {
      HalLedBlink(HAL_LED_8, 8, 50, 500);
    }
    ……
  }
}
```

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……
        case KEY_CHANGE://0xC0，按键事件
          zb_HandleKeys ( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys );
          break;
        ……
      }
    }
  }
```

```
}
```

SimpleSwitch.c

```
void zb_HandleKeys( uint8 shift, uint8 keys )
{
    if ( keys & HAL_KEY_SW_5 )
    {
        ……
        //将 logicalType 设置为终端设备，并保存至 NV 中
         logicalType = ZG_DEVICETYPE_ENDDEVICE;
         zb_WriteConfiguration(ZCD_NV_LOGICAL_TYPE, sizeof(uint8), &logicalType);

        //修改 StartOptions 的值，并保存至 NV 中
        zb_ReadConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );
        startOptions = ZCD_STARTOPT_AUTO_START;
        zb_WriteConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );

        //重新启动协调器
        zb_SystemReset();
    }
    ……
}
```

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & ZB_ENTRY_EVENT )//0x1000，进入事件
  {
      zb_StartRequest();//重新启动设备，建立网络
    ……
  }
}
```

sapi.c

```
void zb_StartRequest()
{
    //设备打开电源之后，由于未形成网络，经逻辑类型判断后，执行此处
    ZDOInitDevice(zgStartDelay);
}
```

ZDApp.c

```
uint8 ZDOInitDevice( uint16 startDelay )
{
    ……
    //网络初始化
    ZDApp_NetworkInit ( extendedDelay );
}
```

ZDApp.c

```
void ZDApp_NetworkInit( uint16 delay )
{
    ……
    osal_start_timerEx ( ZDAppTaskID, ZDO_NETWORK_INIT, delay );
    ……
}
```

ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  ……
  if ( events & ZDO_NETWORK_INIT )//网络初始化事件处理
  {
    // Initialize apps and start the network
    devState = DEV_INIT;
    //设备逻辑类型，启动模式，信标时间，超帧长度
    ZDO_StartDevice ( (uint8)ZDO_Config_Node_Descriptor.LogicalType, devStartMode,
                      DEFAULT_BEACON_ORDER, DEFAULT_SUPERFRAME_ORDER );
    ……
  }
……
}
```

ZDObject.c

```
void ZDO_StartDevice( byte logicalType, devStartModes_t startMode, byte beaconOrder, byte superframeOrder )
{
  ……
  devState = DEV_NWK_DISC;
  ret = NLME_NetworkDiscoveryRequest( zgDefaultChannelList, zgDefaultStartingScanDuration );
   ……
}
```

当发现有网络存在时，网络层将给予 ZDO 层发现网络反馈信息。然后由网络层发起加入网络请求，如加入网络成功，则网络层将给予 ZDO 层加入网络反馈信息。

ZDApp.c

```
ZStatus_t ZDO_NetworkDiscoveryConfirmCB( byte ResultCount,networkDesc_t *NetworkList )
{
  ZDApp_SendMsg( ZDAppTaskID, ZDO_NWK_DISC_CNF, sizeof(ZDO_NetworkDiscoveryCfm_t), (byte *)&msg );
}
```

ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{
  switch ( msgPtr->event )
  {
  case ZDO_NWK_DISC_CNF:
    NLME_JoinRequest( ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->extendedPANID,
    BUILD_UINT16( ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->panIdLSB,
    ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->panIdMSB ),
    ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->logicalChannel,
    ZDO_Config_Node_Descriptor.CapabilityFlags )
  }
}
```

ZDApp.c

```
void ZDO_JoinConfirmCB( uint16 PanId, ZStatus_t Status )
{
  ……
  ZDApp_SendMsg( ZDAppTaskID, ZDO_NWK_JOIN_IND, sizeof(osal_event_hdr_t), (byte*)NULL );
}
```

ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{
  switch ( msgPtr->event )
  {
    ……
    case ZDO_NWK_JOIN_IND:
      ZDApp_ProcessNetworkJoin();
      break;
  }
}
```

ZDApp.c

```
void ZDApp_ProcessNetworkJoin( void )
{
  ……
  osal_set_event( ZDAppTaskID, ZDO_STATE_CHANGE_EVT )
}
```

ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  if ( events & ZDO_STATE_CHANGE_EVT )//ZDO 状态改变事件
  {
    ZDO_UpdateNwkStatus( devState );
  }
```

```
}
```

ZDObject.c

```
void ZDO_UpdateNwkStatus( devStates_t state )
{
        ……
        msgPtr->event = ZDO_STATE_CHANGE; // Command ID
        msgPtr->status = (byte)state;
        osal_msg_send( *(epDesc->epDesc->task_id), (byte *)msgPtr );
}
```

sapi.c

```
uint16 sapi_ProcessEvent( uint8 task_id, uint16 events )
{
        case ZDO_STATE_CHANGE:
          SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
          if ( (SampleApp_NwkState == DEV_ZB_COORD)
               || (SampleApp_NwkState == DEV_ROUTER)
               || (SampleApp_NwkState == DEV_END_DEVICE) )
          {
            SAPI_StartConfirm( ZB_SUCCESS );}
}
```

sapi.c

```
void SAPI_StartConfirm( uint8 status )
{
    zb_StartConfirm( status );
}
```

SimpleSwitch.c

```
void zb_StartConfirm( uint8 status )
{
  if ( status == ZB_SUCCESS )
  {
    myAppState = APP_START;
  }
  else
  {
    osal_start_timerEx( sapi_TaskID, MY_START_EVT, myStartRetryDelay );
  }
}
```

4.3.3 控制节点允许绑定

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……
        case KEY_CHANGE://0xC0，按键事件
          zb_HandleKeys ( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys );
          break;
        ……
      }
    }
  }
}
```

SimpleController.c

```
void zb_HandleKeys( uint8 shift, uint8 keys )
{
    if ( keys & HAL_KEY_SW_5 )
    {
        ……
        zb_AllowBind ( myAllowBindTimeout );
        ……
}
```

sapi.c

```
void zb_AllowBind ( uint8 timeout )
{

    osal_stop_timerEx(sapi_TaskID, ZB_ALLOW_BIND_TIMER);

    afSetMatch(sapi_epDesc.simpleDesc->EndPoint, TRUE);
    ......
    osal_start_timerEx(sapi_TaskID, ZB_ALLOW_BIND_TIMER, timeout*1000);
}
```

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & ZB_ALLOW_BIND_TIMER )//0x4000，允许绑定时间事件
  {
    afSetMatch(sapi_epDesc.simpleDesc->EndPoint, FALSE);
  }
```

```
}
```

4.3.4 开关节点发送绑定请求

在采集节点和传感节点的通信过程中，传感节点在启动加入网络后自动发起绑定请求。在本章中，用户通过按下 S5 键发起绑定请求：

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……
        case KEY_CHANGE://0xC0，按键事件
          zb_HandleKeys ( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys );
          break;
        ……
      }
    }
  }
}
```

SimpleSwitch.c

```
void zb_HandleKeys( uint8 shift, uint8 keys )
{
    if ( keys & HAL_KEY_SW_5)
    {
        // Initiate a binding with null destination
        zb_BindDevice (TRUE, TOGGLE_LIGHT_CMD_ID, NULL); //TOGGLE_LIGHT_CMD_ID=1
    }
    ……
}
```

sapic

```
void zb_BindDevice ( uint8 create, uint16 commandId, uint8 *pDestination )
{
  uint8 ret = ZB_ALREADY_IN_PROGRESS;

  if ( create )//create=TRUE
  {
    if (sapi_bindInProgress == 0xffff)//函数 SAPI_Init（）定义 sapi_bindInProgress = 0xffff
```

```
    {
        ret = ZB_INVALID_PARAMETER;
        destination.addrMode = Addr16Bit;
        destination.addr.shortAddr = NWK_BROADCAST_SHORTADDR;
      if ( ZDO_AnyClusterMatches( 1, &commandId, sapi_epDesc.simpleDesc->AppNumOutClusters,
                                                  sapi_epDesc.simpleDesc->pAppOutClusterList ) )
        {
          //匹配一个在允许绑定模式下的设备
          ret = ZDP_MatchDescReq( &destination, NWK_BROADCAST_SHORTADDR,
              sapi_epDesc.simpleDesc->AppProfId, 1, &commandId, 0, (cId_t *)NULL, 0 );
        }
    else if ( ZDO_AnyClusterMatches( 1, &commandId, sapi_epDesc.simpleDesc->AppNumInClusters,
                                                  sapi_epDesc.simpleDesc->pAppInClusterList ) )
        {
          ret = ZDP_MatchDescReq( &destination, NWK_BROADCAST_SHORTADDR,
              sapi_epDesc.simpleDesc->AppProfId, 0, (cId_t *)NULL, 1, &commandId, 0 );
        }

        if ( ret == ZB_SUCCESS )
        {
          //设置一个时间，确保绑定完成
          osal_start_timerEx(sapi_TaskID, ZB_BIND_TIMER, AIB_MaxBindingTime);
          sapi_bindInProgress = commandId;//允许基于命令的绑定过程
        }
      }

    SAPI_SendCback( SAPICB_BIND_CNF, ret, commandId );
  }
}
```

4.3.5 控制节点处理绑定请求并发送绑定响应

ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  uint8 *msg_ptr;
  if ( events & SYS_EVENT_MSG )//0x8000 系统消息事件
  {
    while ( (msg_ptr = osal_msg_receive( ZDAppTaskID )) )//接收任务号为 ZDAppTaskID 的命令信息
    {
      ZDApp_ProcessOSALMsg( (osal_event_hdr_t *)msg_ptr );//关于命令信息的事件处理函数
      .....
    }
  }
```

```
}
```

ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{
  switch ( msgPtr->event )
  {
    // Incoming ZDO Message
    case AF_INCOMING_MSG_CMD://AF 信息输入
      ZDP_IncomingData( (afIncomingMSGPacket_t *)msgPtr );
      break;
  }
}
```

ZDPofile.c

```
void ZDP_IncomingData( afIncomingMSGPacket_t *pData )
{
  uint8 x = 0;
  uint8 handled;
  zdoIncomingMsg_t inMsg;

  inMsg.srcAddr.addrMode = Addr16Bit;
  inMsg.srcAddr.addr.shortAddr = pData->srcAddr.addr.shortAddr;
  inMsg.wasBroadcast = pData->wasBroadcast;
  inMsg.clusterID = pData->clusterId;
  inMsg.SecurityUse = pData->SecurityUse;

  inMsg.asduLen = pData->cmd.DataLength-1;
  inMsg.asdu = pData->cmd.Data+1;
  inMsg.TransSeq = pData->cmd.Data[0];

  handled = ZDO_SendMsgCBs( &inMsg );

  while ( zdpMsgProcs[x].clusterID != 0xFFFF )
  {
    if ( zdpMsgProcs[x].clusterID == inMsg.clusterID )
    {
      zdpMsgProcs[x].pFn( &inMsg );
      return;
    }
    x++;
  }
.....
}
```

ZDPofile.h

```
CONST zdpMsgProcItem_t zdpMsgProcs[] =
```

```
{
  { NWK_addr_req,              zdpProcessAddrReq },
  { IEEE_addr_req,             zdpProcessAddrReq },
  { Node_Desc_req,             ZDO_ProcessNodeDescReq },
  { Power_Desc_req,            ZDO_ProcessPowerDescReq },
  { Simple_Desc_req,           ZDO_ProcessSimpleDescReq },
  { Active_EP_req,             ZDO_ProcessActiveEPReq },
  { Match_Desc_req,            ZDO_ProcessMatchDescReq },
  ......
  {0xFFFF, NULL} // Last
};
```

ZDObject.c

```
void ZDO_ProcessMatchDescReq( zdoIncomingMsg_t *inMsg )
{
    ......
    ZDP_MatchDescRsp( inMsg->TransSeq, &(inMsg->srcAddr), ZDP_INVALID_REQTYPE,
                      ZDAppNwkAddr.addr.shortAddr, 0, NULL, inMsg->SecurityUse );
    return;
    ......
}
```

4.3.6 开关节点接收并处理绑定响应

由于 sapi 层在初始化时注册了匹配描述符的响应事件，因此在 sapi 层处理匹配描述符的响应事件。

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000 系统消息事件
  {
    pMsg = (osal_event_hdr_t *) osal_msg_receive( task_id );
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        case ZDO_CB_MSG://0xD3，ZDO 的反馈消息
          SAPI_ProcessZDOMsgs( (zdoIncomingMsg_t *)pMsg );
          break;
      }
    }
  }
}
```

sapi.c

```
void SAPI_ProcessZDOMsgs( zdoIncomingMsg_t *inMsg )
{
  switch ( inMsg->clusterID )
  {
    .....
    case Match_Desc_rsp://匹配消息的响应信息
      {

        if ( sapi_bindInProgress != 0xffff )
        {
          // Create a binding table entry 建立绑定表
          dstAddr.addrMode = Addr16Bit;
          dstAddr.addr.shortAddr = pRsp->nwkAddr;

          if ( APSME_BindRequest( sapi_epDesc.simpleDesc->EndPoint,
                                  sapi_bindInProgress, &dstAddr, pRsp->epList[0] ) == ZSuccess )
          {
            osal_stop_timerEx(sapi_TaskID,   ZB_BIND_TIMER);
            osal_start_timerEx( ZDAppTaskID, ZDO_NWK_UPDATE_NV, 250 );
            sapi_bindInProgress = 0xffff;

            //查找 IEEE 地址
            ZDP_IEEEAddrReq( pRsp->nwkAddr, ZDP_ADDR_REQTYPE_SINGLE, 0, 0 );

            //向应用程序发送绑定确认消息
            zb_BindConfirm( sapi_bindInProgress, ZB_SUCCESS );
          }
        }
      }
      break;
  }
}
```

SimpleSwitch.c

```
void zb_BindConfirm( uint16 commandId, uint8 status )
{

  if ( ( status == ZB_SUCCESS ) && ( myAppState == APP_START ) )
  {
    // Turn on LED 1
    HalLedSet( HAL_LED_1, HAL_LED_MODE_ON );
  }
}
```

4.3.7 开关节点发送切换命令

当建立绑定后，开关节点仍然查询按键事件，当按下按键 S6 后，发送切换命令：

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……
        case KEY_CHANGE://0xC0，按键事件
          zb_HandleKeys ( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys );
          break;
        ……
      }
    }
  }
}
```

SimpleSwitch.c

```
void zb_HandleKeys( uint8 shift, uint8 keys )
{
    if ( keys & HAL_KEY_SW_1)
    {
        // 发送切换命令
        zb_SendDataRequest ( 0xFFFE, TOGGLE_LIGHT_CMD_ID, 0,
                             (uint8 *)NULL, myAppSeqNumber, 0, 0 );}
        ……
    }
}
```

sapi.c

```
void zb_SendDataRequest ( uint16 destination, uint16 commandId, uint8 len,
                          uint8 *pData, uint8 handle, uint8 txOptions, uint8 radius )
{
    ……
    // Set the destination address
    dstAddr.addrMode = afAddrNotPresent;//绑定模式地址
    ……
    // Set the endpoint
```

```
    dstAddr.endPoint = sapi_epDesc.simpleDesc->EndPoint;//设定端点
    ……
    // Send the message
    status = AF_DataRequest(&dstAddr, &sapi_epDesc, commandId, len,
                            pData, &handle, txOptions, radius);//发送消息
}
```

4.3.8 控制节点接收切换命令

当 Controller 接收到数据时，会对其进行响应，即会触发 SYS_EVENT_MSG 事件，并调用 AF_INCOMING_MSG_CMD 的处理函数。

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……
        case AF_INCOMING_MSG_CMD://0x1A，AF 信息输入
          SAPI_ReceiveDataIndication ( pMSGpkt->srcAddr.addr.shortAddr, pMSGpkt->clusterId,
                                       pMSGpkt->cmd.DataLength,pMSGpkt->cmd.Data);
        break;
        ……
      }
    }
  }
}
```

sapi.c

```
void SAPI_ReceiveDataIndication( uint16 source, uint16 command, uint16 len, uint8 *pData   )
{
    ……
    zb_ReceiveDataIndication ( source, command, len, pData   );
}
```

SimpleController.c

```
void zb_ReceiveDataIndication( uint16 source, uint16 command, uint16 len, uint8 *pData   )
{
  if (command == TOGGLE_LIGHT_CMD_ID)
  {
    // Received application command to toggle the LED
```

```
      HalLedSet (HAL_LED_5, HAL_LED_MODE_TOGGLE);
      HalLedSet (HAL_LED_6, HAL_LED_MODE_TOGGLE);
      HalLedSet (HAL_LED_7, HAL_LED_MODE_TOGGLE);
      HalLedSet (HAL_LED_8, HAL_LED_MODE_TOGGLE);
    }
}
```

4.4 实验步骤及演示

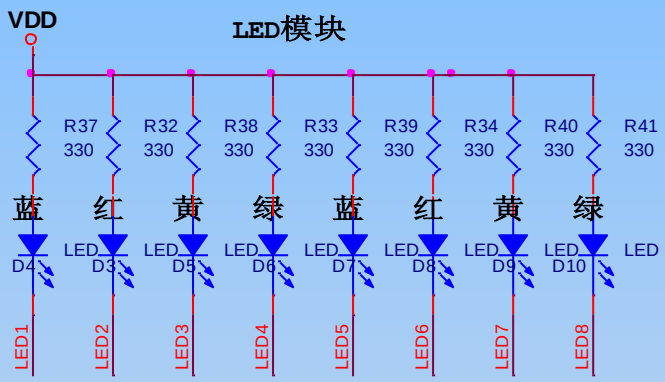
步骤	操作	现象
1	连接 J7	电机驱动芯片供电
2	SW2~SW9 左方跳线，P1_0~P1_7 连接 LED1~LED8	LED 跳线连接
3	SW10 上方跳线	连接蜂鸣器
4	CON 连接 5V 适配器，按下 U10	上电，D2 点亮
5	HEADER 连接仿真器，仿真器 USB 线连接电脑	仿真器 LED 点亮
评估板 A		
6	通过 IAR 软件下载 SimpleControllerEB 至评估板 A	评估板 A 实现控制节点功能
7	按下评估板 A 的 S1，重启	LED8 闪烁
8	按下评估板 A 的 S5，建立网络请求	LED1 点亮
9	自动获得协议栈的建立网路反馈	LED2 点亮，设置 ZDO_NETWORK_START
10	网络自动启动	LED3 点亮，设置 ZDO_STATE_CHANGE_EVT 事件
11	启动 ZDO 状态改变	LED4 点亮，发送 ZDO_STATE_CHANGE 系统消息
12	SAPI 启动确认	LED6 点亮，设置 MY_START_EVT 事件
评估板 B		
14	通过 IAR 软件下载 SimpleSensorEB 至评估板 B	评估板 B 实现开关节点功能
15	按下评估板 B 的 S1，重启	LED8 闪烁
8	按下评估板 A 的 S5，发现网络请求	LED1 点亮，
9	自动获得协议栈的发现网路反馈	LED2 点亮，发送 ZDO_NWK_DISC_CNF 系统消息
10	自动发送加入网络请求	LED3 点亮，
11	自动获得协议栈的加入网路反馈	LED5 点亮，发送 ZDO_NWK_JOIN_IND 系统消息
12	启动 ZDO 状态改变	LED4 点亮，发送 ZDO_STATE_CHANGE 系统消息
13	SAPI 启动确认	LED6 点亮，设置 MY_FIND_COLLECTOR_EVT 事件
评估板 A		
15	按下评估板 A 的 S5，允许绑定	所有 LED 熄灭
评估板 B		
16	按下评估板 B 的 S5，发送匹配描述符请求	所有 LED 熄灭
17	处理匹配描述符的响应	LED8 点亮
18	发送绑定确认给 sapi	LED7 点亮
18	按下 S1，发送切换至评估板 A	LED1 闪烁
评估板 A		
19	接收来自评估板 B 的切换	LED5—LED8 亮灭切换

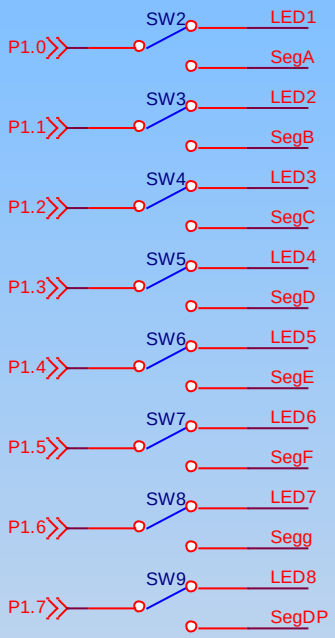
第五章 硬件电路和驱动

5.1 LED 显示

5.1.1 LED 硬件电路

开发板中 LED 模块共计有 8 个 LED 灯，如图所示。它们的阴极分别接到切换开关。由此，当选择使用 LED 模块时，LED1~LED7 分别对应连接 CC2430 的 P1.0~P1.7。I/O 口输出低电平时，LED 点亮；I/O 口输出高电平时，LED 熄灭。其中，330 欧姆电阻起到限流的作用。





5.1.2 LED 驱动配置

Hal_board_cfg.h

```
#define HAL_NUM_LEDS              8
#define HAL_LED_BLINK_DELAY()     st( { volatile uint32 i; for (i=0; i<0x5800; i++) { }; } )

/* 1 - Blue */
#define LED1_BV                   BV(0)
#define LED1_SBIT                 P1_0
#define LED1_DDR                  P1DIR
#define LED1_POLARITY             ACTIVE_LOW
```

```
/* 2 - Red */
#define LED2_BV           BV(1)
#define LED2_SBIT         P1_1
#define LED2_DDR          P1DIR
#define LED2_POLARITY     ACTIVE_LOW

/* 3 - Yellow */
#define LED3_BV           BV(2)
#define LED3_SBIT         P1_2
#define LED3_DDR          P1DIR
#define LED3_POLARITY     ACTIVE_LOW

/* 4 - Green */
#define LED4_BV           BV(3)
#define LED4_SBIT         P1_3
#define LED4_DDR          P1DIR
#define LED4_POLARITY     ACTIVE_LOW

/* 5 - Blue */
#define LED5_BV           BV(4)
#define LED5_SBIT         P1_4
#define LED5_DDR          P1DIR
#define LED5_POLARITY     ACTIVE_LOW

/* 6 - Red */
#define LED6_BV           BV(5)
#define LED6_SBIT         P1_5
#define LED6_DDR          P1DIR
#define LED6_POLARITY     ACTIVE_LOW

/* 7 - Yellow */
#define LED7_BV           BV(6)
#define LED7_SBIT         P1_6
#define LED7_DDR          P1DIR
#define LED7_POLARITY     ACTIVE_LOW

/* 8 - Green */
#define LED8_BV           BV(7)
#define LED8_SBIT         P1_7
#define LED8_DDR          P1DIR
#define LED8_POLARITY     ACTIVE_LOW
```

Hal_board_cfg.h

```
/* ---------- LED's ---------- */
```

```
#define HAL_TURN_OFF_LED1()       st( LED1_SBIT = LED1_POLARITY (0); )
#define HAL_TURN_OFF_LED2()       st( LED2_SBIT = LED2_POLARITY (0); )
#define HAL_TURN_OFF_LED3()       st( LED3_SBIT = LED3_POLARITY (0); )
#define HAL_TURN_OFF_LED4()       st( LED4_SBIT = LED4_POLARITY (0); )
#define HAL_TURN_OFF_LED5()       st( LED1_SBIT = LED5_POLARITY (0); )
#define HAL_TURN_OFF_LED6()       st( LED2_SBIT = LED6_POLARITY (0); )
#define HAL_TURN_OFF_LED7()       st( LED3_SBIT = LED7_POLARITY (0); )
#define HAL_TURN_OFF_LED8()       st( LED4_SBIT = LED8_POLARITY (0); )

#define HAL_TURN_ON_LED1()        st( LED1_SBIT = LED1_POLARITY (1); )
#define HAL_TURN_ON_LED2()        st( LED2_SBIT = LED2_POLARITY (1); )
#define HAL_TURN_ON_LED3()        st( LED3_SBIT = LED3_POLARITY (1); )
#define HAL_TURN_ON_LED4()        st( LED4_SBIT = LED4_POLARITY (1); )
#define HAL_TURN_ON_LED5()        st( LED1_SBIT = LED5_POLARITY (1); )
#define HAL_TURN_ON_LED6()        st( LED2_SBIT = LED6_POLARITY (1); )
#define HAL_TURN_ON_LED7()        st( LED3_SBIT = LED7_POLARITY (1); )
#define HAL_TURN_ON_LED8()        st( LED4_SBIT = LED8_POLARITY (1); )

#define HAL_TOGGLE_LED1()         st( if (LED1_SBIT) { LED1_SBIT = 0; } else { LED1_SBIT = 1;} )
#define HAL_TOGGLE_LED2()         st( if (LED2_SBIT) { LED2_SBIT = 0; } else { LED2_SBIT = 1;} )
#define HAL_TOGGLE_LED3()         st( if (LED3_SBIT) { LED3_SBIT = 0; } else { LED3_SBIT = 1;} )
#define HAL_TOGGLE_LED4()         st( if (LED4_SBIT) { LED4_SBIT = 0; } else { LED4_SBIT = 1;} )
#define HAL_TOGGLE_LED5()         st( if (LED5_SBIT) { LED5_SBIT = 0; } else { LED5_SBIT = 1;} )
#define HAL_TOGGLE_LED6()         st( if (LED6_SBIT) { LED6_SBIT = 0; } else { LED6_SBIT = 1;} )
#define HAL_TOGGLE_LED7()         st( if (LED7_SBIT) { LED7_SBIT = 0; } else { LED7_SBIT = 1;} )
#define HAL_TOGGLE_LED8()         st( if (LED8_SBIT) { LED8_SBIT = 0; } else { LED8_SBIT = 1;} )

#define HAL_STATE_LED1()          (LED1_POLARITY (LED1_SBIT))
#define HAL_STATE_LED2()          (LED2_POLARITY (LED2_SBIT))
#define HAL_STATE_LED3()          (LED3_POLARITY (LED3_SBIT))
#define HAL_STATE_LED4()          (LED4_POLARITY (LED4_SBIT))
#define HAL_STATE_LED5()          (LED5_POLARITY (LED5_SBIT))
#define HAL_STATE_LED6()          (LED6_POLARITY (LED6_SBIT))
#define HAL_STATE_LED7()          (LED7_POLARITY (LED7_SBIT))
#define HAL_STATE_LED8()          (LED8_POLARITY (LED8_SBIT))
```

Hal_LED.h

```
//定义 LED 的 bit 位
#define HAL_LED_1       0x01
#define HAL_LED_2       0x02
#define HAL_LED_3       0x04
```

```
#define HAL_LED_4         0x08
#define HAL_LED_5         0x10
#define HAL_LED_6         0x20
#define HAL_LED_7         0x40
#define HAL_LED_8         0x80
#define  HAL_LED_ALL     (HAL_LED_1 | HAL_LED_2 | HAL_LED_3 | HAL_LED_4| HAL_LED_5 | HAL_LED_6 | HAL_LED_7 | HAL_LED_8)

//定义 LED 工作模式
#define HAL_LED_MODE_OFF          0x00
#define HAL_LED_MODE_ON           0x01
#define HAL_LED_MODE_BLINK        0x02
#define HAL_LED_MODE_FLASH        0x04
#define HAL_LED_MODE_TOGGLE       0x08

//定义 LED 的个数
#define HAL_LED_DEFAULT_MAX_LEDS        8
```

5.1.3 LED 驱动分析

ZMain.c

```
ZSEG int main( void )
{
  ……
  // Initialze HAL drivers
  HalDriverInit();
  ……
}
```

Hal_drivers.c

```
void HalDriverInit (void)
{
  ……
  HalLedInit();
}
```

Hal_led.c

```
void HalLedInit (void)
{
  ……
  HalLedSet (HAL_LED_ALL, HAL_LED_MODE_OFF);
}
```

Hal_led.c

```
uint8 HalLedSet (uint8 leds, uint8 mode)
{
```

```
  uint8 led;
  HalLedControl_t *sts;

  switch (mode)
  {
    case HAL_LED_MODE_BLINK:
      /* Default blink, 1 time, D% duty cycle */
     HalLedBlink (leds, 1, HAL_LED_DEFAULT_DUTY_CYCLE, HAL_LED_DEFAULT_FLASH_TIME);
      break;
    case HAL_LED_MODE_FLASH:
      /* Default flash, N times, D% duty cycle */
      HalLedBlink (leds, HAL_LED_DEFAULT_FLASH_COUNT, HAL_LED_DEFAULT_DUTY_CYCLE,
HAL_LED_DEFAULT_FLASH_TIME);
      break;

    case HAL_LED_MODE_ON:
    case HAL_LED_MODE_OFF:
    case HAL_LED_MODE_TOGGLE:

      led = HAL_LED_1;
      leds &= HAL_LED_ALL;
      sts = HalLedStatusControl.HalLedControlTable;

      while (leds)
      {
        if (leds & led)
        {
          if (mode != HAL_LED_MODE_TOGGLE)
          {
            sts->mode = mode;   /* ON or OFF */
          }
          else
          {
            sts->mode ^= HAL_LED_MODE_ON;   /* Toggle */
          }
          HalLedOnOff (led, sts->mode);
          leds ^= led;
        }
        led <<= 1;
        sts++;
      }
      break;
```

```
      default:
        break;
    }

    return ( HalLedState );

}
```

Hal_led.c

```
void HalLedOnOff (uint8 leds, uint8 mode)
{
    if (leds & HAL_LED_1)
    {
      if (mode == HAL_LED_MODE_ON)
      {
        HAL_TURN_ON_LED1();
      }
      else
      {
        HAL_TURN_OFF_LED1();
      }
    }
    ……// HAL_LED_1~ HAL_LED_7 类似
    if (leds & HAL_LED_7)
    {
      if (mode == HAL_LED_MODE_ON)
      {
        HAL_TURN_ON_LED7();
      }
      else
      {
        HAL_TURN_OFF_LED7();
      }
    }

    /* Remember current state */
    if (mode)
    {
      HalLedState |= leds;
    }
    else
    {
      HalLedState &= ~leds;
    }
}
```

Hal_led.c

```
void HalLedBlink (uint8 leds, uint8 numBlinks, uint8 percent, uint16 period)
{

  if (leds && percent && period)
  {
    if (percent < 100)
    {
      led = HAL_LED_1;
      leds &= HAL_LED_ALL;
      sts = HalLedStatusControl.HalLedControlTable;

      while (leds)
      {
        if (leds & led)
        {
          /* Store the current state of the led before going to blinking */
          preBlinkState |= (led & HalLedState);

          sts->mode   = HAL_LED_MODE_OFF;              /* Stop previous blink */
          sts->time   = period;                        /* Time for one on/off cycle */
          sts->onPct = percent;                        /* % of cycle LED is on */
          sts->todo   = numBlinks;                     /* Number of blink cycles */
          if (!numBlinks) sts->mode |= HAL_LED_MODE_FLASH;   /* Continuous */
          sts->next = osal_GetSystemClock();           /* Start now */
          sts->mode |= HAL_LED_MODE_BLINK;             /* Enable blinking */
          leds ^= led;
        }
        led <<= 1;
        sts++;
      }
      osal_set_event (Hal_TaskID, HAL_LED_BLINK_EVENT);
    }
    else
    {
      HalLedSet (leds, HAL_LED_MODE_ON);                    /* >= 100%, turn on */
    }
  }
  else
  {
    HalLedSet (leds, HAL_LED_MODE_OFF);                     /* No on time, turn off */
  }
}
```

Hal_drivers.c

```
uint16 Hal_ProcessEvent( uint8 task_id, uint16 events )
{
  if ( events & HAL_LED_BLINK_EVENT )//LED 闪烁事件
  {
    HalLedUpdate();
  }
}
```

Hal_led.c

```
void HalLedUpdate (void)
{
  next = 0;
  led  = HAL_LED_1;
  leds = HAL_LED_ALL;
  sts = HalLedStatusControl.HalLedControlTable;

  /* Check if sleep is active or not */
  if (!HalLedStatusControl.sleepActive)
  {
    while (leds)
    {
      if (leds & led)
      {
        if (sts->mode & HAL_LED_MODE_BLINK)
        {
          time = osal_GetSystemClock();
          if (time >= sts->next)
          {
            if (sts->mode & HAL_LED_MODE_ON)
            {
              pct = 100 - sts->onPct;                    /* Percentage of cycle for off */
              sts->mode &= ~HAL_LED_MODE_ON;             /* Say it's not on */
              HalLedOnOff (led, HAL_LED_MODE_OFF);       /* Turn it off */

              if (!(sts->mode & HAL_LED_MODE_FLASH))
              {
                sts->todo--;                             /* Not continuous, reduce count */
                if (!sts->todo)
                {
                  sts->mode ^= HAL_LED_MODE_BLINK;   /* No more blinks */
                }
              }
            }
            else
            {
```

```
            pct = sts->onPct;                    /* Percentage of cycle for on */
            sts->mode |= HAL_LED_MODE_ON;        /* Say it's on */
            HalLedOnOff (led, HAL_LED_MODE_ON);  /* Turn it on */
          }

          if (sts->mode & HAL_LED_MODE_BLINK)
          {
            wait = (((uint32)pct * (uint32)sts->time) / 100);
            sts->next = time + wait;
          }
          else
          {
            /* no more blink, no more wait */
            wait = 0;
            /* After blinking, set the LED back to the state before it blinks */
          HalLedSet (led, ((preBlinkState & led)!=0)?HAL_LED_MODE_ON:HAL_LED_MODE_OFF);
            /* Clear the saved bit */
            preBlinkState &= ~led;
          }
        }
        else
        {
          wait = sts->next - time;  /* Time left */
        }

        if (!next || ( wait && (wait < next) ))
        {
          next = wait;
        }
      }
      leds ^= led;
    }
    led <<= 1;
    sts++;
  }

  if (next)
  {
    osal_start_timerEx(Hal_TaskID, HAL_LED_BLINK_EVENT, next);  /* Schedule event */
  }
 }
}
```

5.2 按键操作

5.2.1 按键硬件电路

开发板共有 6 个按键，其中 S5 定义为“OK”按键，连接 CC2430 的 P0.0 口；S6 定义为“Cancel”按键，连接 CC2430 的 P0.1 口；其余按键 S2、S3、S4、S7 利用对 P0.4 口采样测量电压的方法判断按键。电路原理图如下图所示：

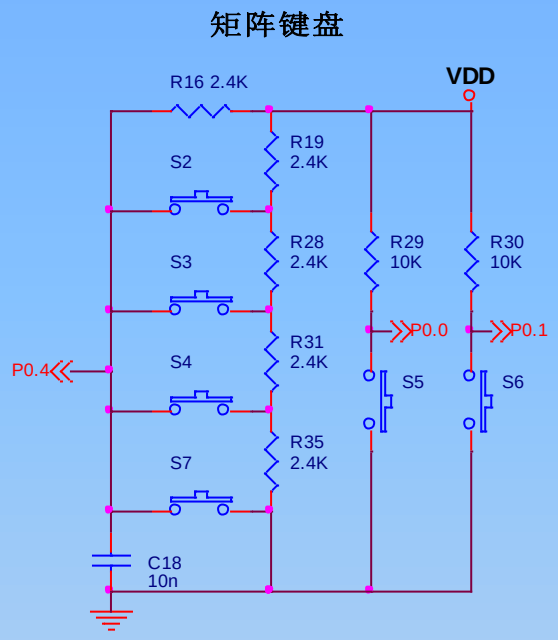


图 7.3 键盘电路原理图

在 Z-Stack 中对按键的处理有 2 种方式：查询方式和中断方式。按键 S2、S3、S4、S7 使用定时查询、AD 采样的方式处理键盘；按键 S5、S6 既可以采用电平高低判断，也可以用来产生中断，由中断服务程序进行事件处理。

Hal_key.h

```
#define HAL_KEY_SW_1 0x01     // Button S2 if available
#define HAL_KEY_SW_2 0x02     // Button S3 if available
#define HAL_KEY_SW_3 0x04     // Button S4 if available
#define HAL_KEY_SW_4 0x08     // Button S7 if available
#define HAL_KEY_SW_5 0x10     // Button S5 if available
#define HAL_KEY_SW_6 0x20     // Button S6 if available
```

```
#define HAL_KEY_UP          0x01    // Joystick up
#define HAL_KEY_DOWN        0x02    // Joystick down
#define HAL_KEY_LEFT        0x04    // Joystick left
#define HAL_KEY_RIGHT       0x08    // Joystick right
```

Hal_key.c

```
//定义与 key 有关的 bit 位
#define HAL_KEY_BIT0    0x01
#define HAL_KEY_BIT1    0x02
#define HAL_KEY_BIT2    0x04
#define HAL_KEY_BIT3    0x08
#define HAL_KEY_BIT4    0x10
#define HAL_KEY_BIT5    0x20
#define HAL_KEY_BIT6    0x40
#define HAL_KEY_BIT7    0x80
```

Hal_key.c

HAL_KEY_JOYSTICK_ENABLE

```
#define HAL_KEY_SW_5_ENABLE
#define HAL_KEY_SW_5_PORT           P0                  /* Port location of S5 */
#define HAL_KEY_SW_5_BIT            HAL_KEY_BIT0        /* Bit location of S5 */
#define HAL_KEY_SW_5_SEL            P0SEL               /* Port Select Register for S5 */
#define HAL_KEY_SW_5_DIR            P0DIR               /* Port Direction Register for S5 */
#define HAL_KEY_SW_5_INP            P0INP               /* Port Input Mode Register for S5 */
#define HAL_KEY_SW_5_IEN            IEN1                /* Interrupt Enable Register for S5 */
#define HAL_KEY_SW_5_IENBIT         HAL_KEY_BIT5        /* Interrupt Enable bit for S5 */
#define HAL_KEY_SW_5_EDGE           HAL_KEY_RISING_EDGE /* Type of interrupt for S5 */
#define HAL_KEY_SW_5_EDGEBIT        HAL_KEY_BIT0        /* EdgeType enable bit S5 */
#define HAL_KEY_SW_5_ICTL           PICTL               /* Port Interrupt Control for S5 */
#define HAL_KEY_SW_5_ICTLBIT        HAL_KEY_BIT3        /* Interrupt enable bit for S5 */
#define HAL_KEY_SW_5_PXIFG          P0IFG               /* Port Interrupt Flag for S5 */
/* P0 can only be enabled/disabled as group of high or low nibble */
#define HAL_KEY_POINT_HIGH_USED     HAL_KEY_SW_5_BIT

#define HAL_KEY_SW_6_ENABLE
#define HAL_KEY_SW_6_PORT           P0                  /* Port location of S6 */
#define HAL_KEY_SW_6_BIT            HAL_KEY_BIT1        /* Bit location of S6 */
#define HAL_KEY_SW_6_SEL            P0SEL               /* Port Select Register for S6 */
#define HAL_KEY_SW_6_DIR            P0DIR               /* Port Direction Register for S6 */
#define HAL_KEY_SW_6_INP            P0INP               /* Port Input Mode Register for S6 */
#define HAL_KEY_SW_6_IEN            IEN1                /* Interrupt Enable Register for S6 */
#define HAL_KEY_SW_6_IENBIT         HAL_KEY_BIT5        /* Interrupt Enable bit for S6 */
#define HAL_KEY_SW_6_EDGE           HAL_KEY_RISING_EDGE /* Type of interrupt for S6 */
```

```
#define HAL_KEY_SW_6_EDGEBIT      HAL_KEY_BIT0          /* EdgeType enable bit S6 */
#define HAL_KEY_SW_6_ICTL         PICTL                 /* Port Interrupt Control for S6 */
#define HAL_KEY_SW_6_ICTLBIT      HAL_KEY_BIT3          /* Interrupt enable bit for S6 */
#define HAL_KEY_SW_6_PXIFG        P0IFG                 /* Port Interrupt Flag for S6 */
/* P0 can only be enabled/disabled as group of high or low nibble */
#define HAL_KEY_P0INT_LOW_USED    HAL_KEY_SW_6_BIT
```

5.2.2 按键配置分析

ZMain.c

```
ZSEG int main( void )
{
  ……
  // Final board initialization
  InitBoard( OB_READY );
  ……
}
```

OnBoard.c

```
void InitBoard( byte level )
{
     ……
      /* Initialize Key stuff */
     OnboardKeyIntEnable = HAL_KEY_INTERRUPT_DISABLE;//禁止按键中断
     HalKeyConfig( OnboardKeyIntEnable, OnBoard_KeyCallback);
}
```

Hal_key.c

```
void HalKeyConfig (bool interruptEnable, halKeyCBack_t cback)
{
  if (Hal_KeyIntEnable)//中断使能
  {
     PICTL &= ~(HAL_KEY_SW_5_EDGEBIT);                  /* Set rising edge */
     HAL_KEY_SW_5_ICTL |= HAL_KEY_SW_5_ICTLBIT;          /* Set interrupt enable bit */
     HAL_KEY_SW_5_IEN |= HAL_KEY_SW_5_IENBIT;
     HAL_KEY_SW_5_PXIFG = ~(HAL_KEY_SW_5_BIT);          /* Clear any pending interrupts */

     PICTL &= ~(HAL_KEY_SW_6_EDGEBIT);                 /* Set rising or falling edge */
     HAL_KEY_SW_6_ICTL |= HAL_KEY_SW_6_ICTLBIT;          /* Set interrupt enable bit */
     HAL_KEY_SW_6_IEN |= HAL_KEY_SW_6_IENBIT;
     HAL_KEY_SW_6_PXIFG = ~(HAL_KEY_SW_6_BIT);          /* Clear any pending interrupts */

       osal_stop_timerEx( Hal_TaskID, HAL_KEY_EVENT);   /* Cancel polling if active */
  }
  else      /* Interrupts NOT enabled */
  {
```

```
    HAL_KEY_SW_6_ICTL &= ~(HAL_KEY_SW_6_ICTLBIT);       /* Clear interrupt enable bit */
    HAL_KEY_SW_6_IEN &= ~(HAL_KEY_SW_6_IENBIT);

    HAL_KEY_SW_5_ICTL &= ~(HAL_KEY_SW_5_ICTLBIT);       /* Clear interrupt enable bit */
    HAL_KEY_SW_5_IEN &= ~(HAL_KEY_SW_5_IENBIT);

    osal_start_timerEx (Hal_TaskID, HAL_KEY_EVENT, HAL_KEY_POLLING_VALUE);
  }
}
```

5.2.3 按键中断方式

Hal_key.c

```
HAL_ISR_FUNCTION( halKeyPort0Isr, P0INT_VECTOR )
{
  halProcessKeyInterrupt();
}
```

Hal_key.c

```
void halProcessKeyInterrupt (void)
{


  if (HAL_KEY_SW_6_PXIFG & HAL_KEY_SW_6_BIT)        /* Interrupt Flag has been set */
  {
    HAL_KEY_SW_6_PXIFG = ~(HAL_KEY_SW_6_BIT);       /* Clear Interrupt Flag */
    valid = TRUE;
  }
  if (HAL_KEY_SW_5_PXIFG & HAL_KEY_SW_5_BIT)        /* Interrupt Flag has been set */
  {
    HAL_KEY_SW_5_PXIFG = ~(HAL_KEY_SW_5_BIT);       /* Clear Interrupt Flag */
    valid = TRUE;
  }

  if (valid)
  {
    osal_start_timerEx (Hal_TaskID, HAL_KEY_EVENT, HAL_KEY_DEBOUNCE_VALUE);
  }
}
```

5.2.4 按键事件处理

Hal_drivers.c

```
uint16 Hal_ProcessEvent( uint8 task_id, uint16 events )
```

```
{

  if (events & HAL_KEY_EVENT)//按键事件
  {
    HalKeyPoll();
    /* if interrupt disabled, do next polling */
    if (!Hal_KeyIntEnable)
    {
      osal_start_timerEx( Hal_TaskID, HAL_KEY_EVENT, 100);
    }
  }
}
```

Hal_key.c

```
void HalKeyPoll (void)
{
  if (!(HAL_KEY_SW_6_PORT & HAL_KEY_SW_6_BIT))       /* Key is active low */
  {
    keys |= HAL_KEY_SW_6;
  }
  if (HAL_KEY_SW_5_PORT & HAL_KEY_SW_5_BIT)           /* Key is active high */
  {
    keys |= HAL_KEY_SW_5;
  }

  do
  {
    adc = HalAdcRead (HAL_KEY_JOY_CHN, HAL_ADC_RESOLUTION_8);
      if ((adc >= 90) && (adc <= 100))
      {
          ksave0 |= HAL_KEY_UP;
      }
      else if ((adc >= 75) && (adc <= 85))
      {
        ksave0 |= HAL_KEY_RIGHT;
      }
      else if ((adc >= 45) && (adc <= 55))
      {
        ksave0 |= HAL_KEY_LEFT;
      }
      else if (adc <= 10)
      {
        ksave0 |= HAL_KEY_DOWN;
      }
      else if ((adc >= 101) && (adc <= 115))
```

```
        {
        }
    } while (ksave0 != ksave1);

    keys |= ksave0;

    /* Exit if polling and no keys have changed */
    if (!Hal_KeyIntEnable)
    {
      if (keys == halKeySavedKeys)
      {
        return;
      }
      halKeySavedKeys = keys;        /* Store the current keys for comparation next time */
    }

    /* Invoke Callback if new keys were depressed */
    if (keys && (pHalKeyProcessFunction))
    {
      (pHalKeyProcessFunction) (keys, HAL_KEY_STATE_NORMAL);
    }
}
```

OnBoard.c

```
void OnBoard_KeyCallback ( uint8 keys, uint8 state )
{
    if ( OnBoard_SendKeys( keys, shift ) != ZSuccess )
    ……
}
```

OnBoard.c

```
byte OnBoard_SendKeys( byte keys, byte state )
{
      // Send the address to the task
      msgPtr = (keyChange_t *)osal_msg_allocate( sizeof(keyChange_t) );
      if ( msgPtr )
      {
        msgPtr->hdr.event = KEY_CHANGE;
        msgPtr->state = state;
        msgPtr->keys = keys;

        osal_msg_send( registeredKeysTaskID, (uint8 *)msgPtr );
      }
}
```

sapi.c

```
UINT16 SAPI_ProcessEvent( byte task_id, UINT16 events )
```

```
{
  if ( events & SYS_EVENT_MSG )//0x8000,系统消息事件
  {
    while ( pMsg )
    {
      switch ( pMsg->event )
      {
        ……
        case KEY_CHANGE://0xC0，按键事件
          zb_HandleKeys ( ((keyChange_t *)pMsg)->state, ((keyChange_t *)pMsg)->keys );
          break;
        ……
      }
    }
  }
}
```

第六章 树型网络通信过程

评估板 A 作为协调器，上电启动后自动建立网络，并且周期性地广播数据包；评估板 B 作为路由器，上电启动后加入网络，并自动发起路由请求，待处理完路由响应后，周期性地广播数据包；评估板 C 作为终端设备，上电启动后加入网络，并且周期性地广播数据包。

6.1 实验目的

- ✧ 掌握协调器启动过程
- ✧ 了解协调器建立网络过程
- ✧ 掌握路由器启动过程
- ✧ 了解路由器加入网络过程
- ✧ 掌握终端设备启动过程
- ✧ 了解终端设备加入网络过程
- ✧ 掌握协调器、路由器和终端设备广播数据包
- ✧ 掌握协调器、路由器和终端设备组播数据包

6.2 实验电路

电路参见：ZigBee Evaluation Board V1.0 原理图 LED 模块、按键、步进电机。

6.3 实验原理及代码

6.3.1 启动过程分析

无论是协调器还是路由器或是终端设备，其启动过程至网络初始步骤均是一样的，只是不同设备的配置文件在编译时有所区别：

✧ 协调器： F8wCoord.cfg

```
/* Coordinator Settings */
-DZDO_COORDINATOR                    // Coordinator Functions
```

✧ 路由器：F8wRouter.cfg

```
/* Router Settings */
-DRTR_NWK                                   // Router Functions
```

✧ 终端设备：F8wEndev.cfg

无

启动过程如下：

C:\Texas Instruments\FS_ZStack\Projects\zstack\Samples\SampleApp\CC2430DB\SampleApp\ZMain.c

```
ZSEG int main ( void )
{
  ……
  //初始化操作系统
  osal_init_system ();
  ……
}
```

OSAL.c

```
byte osal_init_system( void )
{
  ……
  //初始化系统的任务
  osalInitTasks ();
  ……
}
```

OSAL_SampleApp.c

```
void osalInitTasks( void )
{
  ……
  ZDApp_Init( taskID++ );
  SampleApp_Init( taskID );
}
```

ZDApp.c

```
void ZDApp_Init( byte task_id )
{
  ZDAppTaskID = task_id;

  // Initialize the ZDO global device short address storage
  ZDAppNwkAddr.addrMode = Addr16Bit;//地址模式为 16bit 短地址
  ZDAppNwkAddr.addr.shortAddr = INVALID_NODE_ADDR;//短地址=0xFFFE
  (void)NLME_GetExtAddr();   //API 函数，用于得到 64bit IEEE 地址

  // Initialize ZDO items and setup the device - type of device to create.
  ZDO_Init();

  afRegister( (endPointDesc_t *)&ZDApp_epDesc );//为设备注册端点 0 描述符
```

```
  // Start the device?
  if ( devState != DEV_HOLD ) // 无 HOLD_AUTO_START 此项预编译，所以 devState = DEV_INIT
  {
    ZDOInitDevice( 0 );
  }
  ZDApp_RegisterCBs();
}
```

ZDApp.c

```
uint8 ZDOInitDevice( uint16 startDelay )
{
  uint8 networkStateNV = ZDO_INITDEV_NEW_NETWORK_STATE;//0x01,初始化该设备新的网络状态

  devState = DEV_INIT;      // 无连接
  ZDAppDetermineDeviceType();//确定该设备的类型
  extendedDelay = (uint16)((NWK_START_DELAY + startDelay)
                + (osal_rand() & EXTENDED_JOINING_RANDOM_MASK));//加入网络的时延

  ZDApp_NetworkInit( extendedDelay );//开始形成网络
}
```

ZDApp.c

```
void ZDApp_NetworkInit( uint16 delay )
{
  if ( delay )
  {
    // Wait awhile before starting the device
    osal_start_timerEx( ZDAppTaskID, ZDO_NETWORK_INIT, delay );
  }
  else
  {
    osal_set_event( ZDAppTaskID, ZDO_NETWORK_INIT );
  }
}
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{

  if ( events & ZDO_NETWORK_INIT )//网络初始化事件处理
  {
    // Initialize apps and start the network
    devState = DEV_INIT;
    ZDO_StartDevice( (uint8)ZDO_Config_Node_Descriptor.LogicalType, devStartMode,
                     DEFAULT_BEACON_ORDER, DEFAULT_SUPERFRAME_ORDER );
    HalLedSet (HAL_LED_1, HAL_LED_MODE_ON);//点亮 LED1 表示网络初始化完成
  }
```

6.1.1 协调器建网

ZDObject.c

```
void ZDO_StartDevice( byte logicalType, devStartModes_t startMode, byte beaconOrder, byte superframeOrder )
{
    devState = DEV_COORD_STARTING;
    ret = NLME_NetworkFormationRequest( zgConfigPANID, zgDefaultChannelList,
          zgDefaultStartingScanDuration, beaconOrder,superframeOrder,false );
    ……
}
```

6.1.2 路由器入网

ZDObject.c

```
void ZDO_StartDevice( byte logicalType, devStartModes_t startMode, byte beaconOrder, byte superframeOrder )
{
     devState = DEV_NWK_DISC;
     ret = NLME_NetworkDiscoveryRequest( zgDefaultChannelList, zgDefaultStartingScanDuration );
     ……
}
```

6.1.3 终端设备入网

ZDObject.c

```
void ZDO_StartDevice( byte logicalType, devStartModes_t startMode, byte beaconOrder, byte superframeOrder )
{
     devState = DEV_NWK_DISC;
     ret = NLME_NetworkDiscoveryRequest( zgDefaultChannelList, zgDefaultStartingScanDuration );
     ……
}
```

6.1.4 应用层初始化

SampleApp.c

```
void SampleApp_Init( uint8 task_id )
{
  // Setup for the periodic message's destination address
  // Broadcast to everyone
```

```
  SampleApp_Periodic_DstAddr.addrMode = (afAddrMode_t)AddrBroadcast;
  SampleApp_Periodic_DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
  SampleApp_Periodic_DstAddr.addr.shortAddr = 0xFFFF;

  // Setup for the flash command's destination address - Group 1
  SampleApp_Flash_DstAddr.addrMode = (afAddrMode_t)afAddrGroup;
  SampleApp_Flash_DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
  SampleApp_Flash_DstAddr.addr.shortAddr = SAMPLEAPP_FLASH_GROUP;// shortAddr =0x0001

  // Fill out the endpoint description.
  SampleApp_epDesc.endPoint = SAMPLEAPP_ENDPOINT;
  SampleApp_epDesc.task_id = &SampleApp_TaskID;
  SampleApp_epDesc.simpleDesc
            = (SimpleDescriptionFormat_t *)&SampleApp_SimpleDesc;
  SampleApp_epDesc.latencyReq = noLatencyReqs;

  // Register the endpoint description with the AF
  afRegister( &SampleApp_epDesc );

  // Register for all key events - This app will handle all key events
  RegisterForKeys( SampleApp_TaskID );

  // By default, all devices start out in Group 1
  SampleApp_Group.ID = 0x0001;
  osal_memcpy( SampleApp_Group.name, "Group 1", 7  );
  aps_AddGroup( SAMPLEAPP_ENDPOINT, &SampleApp_Group );
}
```

6.3.2 协调器启动并周期性广播数据包

当网络层执行 NLME_NetworkFormationRequest（）建立网络后，将给予 ZDO 层反馈信息：

ZDApp.c

```
void ZDO_NetworkFormationConfirmCB( ZStatus_t Status )
{
#if defined(ZDO_COORDINATOR)
  HalLedSet ( HAL_LED_2, HAL_LED_MODE_ON );
  ……
  osal_set_event ( ZDAppTaskID, ZDO_NETWORK_START );
#endif
}
```

ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  ……
  if ( events & ZDO_NETWORK_START )//网络启动事件处理
  {
    ZDApp_NetworkStartEvt();
    HalLedSet (HAL_LED_3, HAL_LED_MODE_ON);
  }
}
```

ZDApp.c

```
void ZDApp_NetworkStartEvt( void )
{
    devState = DEV_ZB_COORD;
    ……
    osal_set_event( ZDAppTaskID, ZDO_STATE_CHANGE_EVT );
}
```

ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  ……
  if ( events & ZDO_STATE_CHANGE_EVT )//ZDO 状态改变事件
  {
    ZDO_UpdateNwkStatus( devState );
    HalLedSet (HAL_LED_4, HAL_LED_MODE_ON);
  }
}
```

ZDObject.c

```
void ZDO_UpdateNwkStatus( devStates_t state )
{
        ……
        msgPtr->event = ZDO_STATE_CHANGE; // Command ID
        msgPtr->status = (byte)state;

        osal_msg_send( *(epDesc->epDesc->task_id), (byte *)msgPtr );
}
```

SampleApp.c

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
        case ZDO_STATE_CHANGE:
          SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
          if ( (SampleApp_NwkState == DEV_ZB_COORD)
```

```
            || (SampleApp_NwkState == DEV_ROUTER)
            || (SampleApp_NwkState == DEV_END_DEVICE) )
        {
          // Start sending the periodic message in a regular interval.
          osal_start_timerEx( SampleApp_TaskID,
                              SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
                              SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
        }
}
```

SampleApp.c

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
    // Send the periodic message
    SampleApp_SendPeriodicMessage();

    HalLedSet (HAL_LED_5, HAL_LED_MODE_TOGGLE);

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

SampleApp.c

```
void SampleApp_SendPeriodicMessage( void )
{
    AF_DataRequest( &SampleApp_Periodic_DstAddr, &SampleApp_epDesc,
                    SAMPLEAPP_PERIODIC_CLUSTERID,
                    1,
                    (uint8*)&SampleAppPeriodicCounter,
                    &SampleApp_TransID,
                    AF_DISCV_ROUTE,
                    AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
}
```

6.3.3 路由器启动并周期性发送数据包

当网络层执行加入网络后，将给予 ZDO 层反馈信息：

ZDApp.c

```
ZStatus_t ZDO_NetworkDiscoveryConfirmCB( byte ResultCount,networkDesc_t *NetworkList )
{
  HalLedSet ( HAL_LED_8, HAL_LED_MODE_ON );
  ZDApp_SendMsg( ZDAppTaskID, ZDO_NWK_DISC_CNF,sizeof(ZDO_NetworkDiscoveryCfm_t),
  (byte *)&msg );
```

```
}
```

ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{
  switch ( msgPtr->event )
  {
  case ZDO_NWK_DISC_CNF:

  HalLedSet (HAL_LED_7, HAL_LED_MODE_ON);

  NLME_JoinRequest( ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->extendedPANID,
  BUILD_UINT16( ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->panIdLSB,
  ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->panIdMSB ),
  ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->logicalChannel,
  ZDO_Config_Node_Descriptor.CapabilityFlags )
  }
}
```

ZDApp.c

```
void ZDO_JoinConfirmCB( uint16 PanId, ZStatus_t Status )
{
  HalLedSet ( HAL_LED_6, HAL_LED_MODE_ON );
  ……
  ZDApp_SendMsg( ZDAppTaskID, ZDO_NWK_JOIN_IND, sizeof(osal_event_hdr_t), (byte*)NULL );
}
```

ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{
  switch ( msgPtr->event )
  {
    ……
    case ZDO_NWK_JOIN_IND:
      ZDApp_ProcessNetworkJoin();
      break;
  }
}
```

ZDApp.c

```
void ZDApp_ProcessNetworkJoin( void )
{
  if ( (devState == DEV_NWK_JOINING) ||
       ((devState == DEV_NWK_ORPHAN)   &&
        (ZDO_Config_Node_Descriptor.LogicalType == NODETYPE_ROUTER)) )
  {
    // Result of a Join attempt by this device.
```

```
    if ( nwkStatus == ZSuccess )
    {
      osal_set_event( ZDAppTaskID, ZDO_STATE_CHANGE_EVT );

      devState = DEV_END_DEVICE;
     ……
      NLME_StartRouterRequest( 0, 0, false )
    }
  }
}
```

ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
  ......
  if ( events & ZDO_STATE_CHANGE_EVT )//ZDO 状态改变事件
  {
    ZDO_UpdateNwkStatus( devState );

    HalLedSet (HAL_LED_4, HAL_LED_MODE_ON);

    // Return unprocessed events
    return (events ^ ZDO_STATE_CHANGE_EVT);
  }
```

ZDObject.c

```
void ZDO_UpdateNwkStatus( devStates_t state )
{

  ZDAppNwkAddr.addr.shortAddr = NLME_GetShortAddr();
  (void)NLME_GetExtAddr();    // Load the saveExtAddr pointer.

  while ( epDesc )
  {
    if ( epDesc->epDesc->endPoint != ZDO_EP )
    {
      msgPtr = (osal_event_hdr_t *)osal_msg_allocate( bufLen );
      if ( msgPtr )
      {
        msgPtr->event = ZDO_STATE_CHANGE; // Command ID
        msgPtr->status = (byte)state;

        osal_msg_send( *(epDesc->epDesc->task_id), (byte *)msgPtr );
      }
    }
```

```
      epDesc = epDesc->nextDesc;
    }
}
```

SampleApp.c

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
          case ZDO_STATE_CHANGE:
             SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
             if ( (SampleApp_NwkState == DEV_ZB_COORD)
                  || (SampleApp_NwkState == DEV_ROUTER)
                  || (SampleApp_NwkState == DEV_END_DEVICE) )
             {
               // Start sending the periodic message in a regular interval.
               osal_start_timerEx( SampleApp_TaskID,
                                   SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
                                   SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
             }
}
```

SampleApp.c

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
    // Send the periodic message
    SampleApp_SendPeriodicMessage();

    HalLedSet (HAL_LED_5, HAL_LED_MODE_TOGGLE);

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

SampleApp.c

```
void SampleApp_SendPeriodicMessage( void )
{
    AF_DataRequest( &SampleApp_Periodic_DstAddr, &SampleApp_epDesc,
                                SAMPLEAPP_PERIODIC_CLUSTERID,
                                1,
                                (uint8*)&SampleAppPeriodicCounter,
                                &SampleApp_TransID,
                                AF_DISCV_ROUTE,
                                AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
}

void ZDO_StartRouterConfirmCB( ZStatus_t Status )
```

```
{
   ……
   HalLedSet ( HAL_LED_3, HAL_LED_MODE_ON );
   osal_set_event( ZDAppTaskID, ZDO_ROUTER_START );
}
```

ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
   if ( events & ZDO_ROUTER_START )//路由启动事件处理
   {
      devState = DEV_ROUTER;
      osal_set_event( ZDAppTaskID, ZDO_STATE_CHANGE_EVT );
   }
}
```

ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
   if ( events & ZDO_STATE_CHANGE_EVT )//ZDO 状态改变事件
   {
      ZDO_UpdateNwkStatus( devState );
   }
}
```

ZDObject.c

```
void ZDO_UpdateNwkStatus( devStates_t state )
{
          ……
          msgPtr->event = ZDO_STATE_CHANGE; // Command ID
          msgPtr->status = (byte)state;

          osal_msg_send( *(epDesc->epDesc->task_id), (byte *)msgPtr );
}
```

SampleApp.c

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
          case ZDO_STATE_CHANGE:
             SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
             if ( (SampleApp_NwkState == DEV_ZB_COORD)
                  || (SampleApp_NwkState == DEV_ROUTER)
                  || (SampleApp_NwkState == DEV_END_DEVICE) )
             {
                // Start sending the periodic message in a regular interval.
                osal_start_timerEx( SampleApp_TaskID,
```

```
                                SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
                                SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
            }
}
```

SampleApp.c

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
    // Send the periodic message
    SampleApp_SendPeriodicMessage();

    HalLedSet (HAL_LED_5, HAL_LED_MODE_TOGGLE);

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

6.3.4 终端设备启动并周期性发送数据包

当网络层执行加入网络后，将给予 ZDO 层反馈信息：

ZDApp.c

```
ZStatus_t ZDO_NetworkDiscoveryConfirmCB( byte ResultCount,networkDesc_t *NetworkList )
{
  ZDApp_SendMsg( ZDAppTaskID, ZDO_NWK_DISC_CNF,sizeof(ZDO_NetworkDiscoveryCfm_t),
  (byte *)&msg );
}
```

ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{
  switch ( msgPtr->event )
  {
  case ZDO_NWK_DISC_CNF:
  NLME_JoinRequest( ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->extendedPANID,
  BUILD_UINT16( ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->panIdLSB,
  ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->panIdMSB ),
  ((ZDO_NetworkDiscoveryCfm_t *)msgPtr)->logicalChannel,
  ZDO_Config_Node_Descriptor.CapabilityFlags )
  }
}
```

ZDApp.c

```
void ZDO_JoinConfirmCB( uint16 PanId, ZStatus_t Status )
{
```

```
    ……
    ZDApp_SendMsg( ZDAppTaskID, ZDO_NWK_JOIN_IND, sizeof(osal_event_hdr_t), (byte*)NULL );
}
```

ZDApp.c

```
void ZDApp_ProcessOSALMsg( osal_event_hdr_t *msgPtr )
{
    switch ( msgPtr->event )
    {
      ……
      case ZDO_NWK_JOIN_IND:
        ZDApp_ProcessNetworkJoin();
        break;
    }
}
```

ZDApp.c

```
void ZDApp_ProcessNetworkJoin( void )
{
    ……
    osal_set_event( ZDAppTaskID, ZDO_STATE_CHANGE_EVT )
}
```

ZDApp.c

```
UINT16 ZDApp_event_loop( byte task_id, UINT16 events )
{
    if ( events & ZDO_STATE_CHANGE_EVT )//ZDO 状态改变事件
    {
      ZDO_UpdateNwkStatus( devState );
    }
}
```

ZDObject.c

```
void ZDO_UpdateNwkStatus( devStates_t state )
{
            ……
            msgPtr->event = ZDO_STATE_CHANGE; // Command ID
            msgPtr->status = (byte)state;

            osal_msg_send( *(epDesc->epDesc->task_id), (byte *)msgPtr );
}
```

SampleApp.c

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
            case ZDO_STATE_CHANGE:
              SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
```

```
        if ( (SampleApp_NwkState == DEV_ZB_COORD)
            || (SampleApp_NwkState == DEV_ROUTER)
            || (SampleApp_NwkState == DEV_END_DEVICE) )
        {
          // Start sending the periodic message in a regular interval.
          osal_start_timerEx( SampleApp_TaskID,
                              SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
                              SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
        }
}
```

SampleApp.c

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
    // Send the periodic message
    SampleApp_SendPeriodicMessage();

    HalLedSet (HAL_LED_5, HAL_LED_MODE_TOGGLE);

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

6.3.5 发送数据

SampleApp.c

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
  if ( events & SYS_EVENT_MSG )
  {
    while ( MSGpkt )
    {
      switch ( MSGpkt->hdr.event )
      {
        // Received when a key is pressed
      case KEY_CHANGE:
      SampleApp_HandleKeys( ((keyChange_t *)MSGpkt)->state, ((keyChange_t *)MSGpkt)->keys );
      break;
      }
    }
  }
}
```

SampleApp.c

```
void SampleApp_HandleKeys( uint8 shift, uint8 keys )
{
  if ( keys & HAL_KEY_SW_5 )
  {
    SampleApp_SendFlashMessage( SAMPLEAPP_FLASH_DURATION );
  }
}
```

SampleApp.c

```
void SampleApp_SendFlashMessage( uint16 flashTime )
{
  uint8 buffer[3];
  buffer[0] = (uint8)(SampleAppFlashCounter++);
  buffer[1] = LO_UINT16( flashTime );
  buffer[2] = HI_UINT16( flashTime );

  AF_DataRequest( &SampleApp_Flash_DstAddr, &SampleApp_epDesc,
                  SAMPLEAPP_FLASH_CLUSTERID,
                  3,
                  buffer,
                  &SampleApp_TransID,
                  AF_DISCV_ROUTE,
                  AF_DEFAULT_RADIUS )
}
```

6.3.6 接收数据

SampleApp.c

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{

  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 );
          break;
```

```
        }
      }
    }
}
```

SampleApp.c

```
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
  uint16 flashTime;

  switch ( pkt->clusterId )
  {
    case SAMPLEAPP_PERIODIC_CLUSTERID:
      break;

    case SAMPLEAPP_FLASH_CLUSTERID:
      flashTime = BUILD_UINT16(pkt->cmd.Data[1], pkt->cmd.Data[2] );
      HalLedBlink( HAL_LED_1, 4, 50, (flashTime / 4) );
      HalLedBlink( HAL_LED_2, 4, 50, (flashTime / 4) );
      break;
  }
}
```

6.4 实验步骤及演示

步骤	操作	现象
1	连接 J7	电机驱动芯片供电
2	SW2~SW9 左方跳线，P1_0~P1_7 连接 LED1~LED8	LED 跳线连接
3	连接电机 5 芯接头至 J4	连接步进电机
4	CON 连接 5V 适配器，按下 U10	上电，D2 点亮
5	HEADER 连接仿真器，仿真器 USB 线连接电脑	仿真器 LED 点亮
评估板 A		
6	通过 IAR 软件下载 CoordinatorEB 至评估板 A	评估板 A 实现协调器功能
7	自动发起建立网络请求	LED1 点亮
8	自动获得协议栈的建立网路反馈	LED2 点亮，设置 ZDO_NETWORK_START
9	网络自动启动	LED3 点亮，设置 ZDO_STATE_CHANGE_EVT 事件
10	启动 ZDO 状态改变	LED4 点亮，发送 ZDO_STATE_CHANGE 系统消息
11	广播数据包	LED5 闪烁，其他 LED 熄灭
评估板 B		
12	通过 IAR 软件下载 RouterEB 至评估板 B	评估板 B 实现路由器功能
13	自动发送发现网络请求	LED1 点亮，
14	自动获得协议栈的发现网路反馈	LED8 点亮，发送 ZDO_NWK_DISC_CNF 系统消息
15	自动发送加入网络请求	LED7 点亮，

16	自动获得协议栈的加入网路反馈	LED6 点亮，发送 ZDO_NWK_JOIN_IND 系统消息
17	启动 ZDO 状态改变	LED4 点亮，发送 ZDO_STATE_CHANGE 系统消息
18	自动获得路由器启动响应	LED3 点亮
19	广播数据包	LED5 闪烁，其他 LED 熄灭
评估板 C		
20	通过 IAR 软件下载 EndDeviceEB 至评估板 B	评估板 C 实现终端设备功能
21	自动发送发现网络请求	LED1 点亮，
22	自动获得协议栈的发现网路反馈	LED8 点亮，发送 ZDO_NWK_DISC_CNF 系统消息
23	自动发送加入网络请求	LED7 点亮，
24	自动获得协议栈的加入网路反馈	LED6 点亮，发送 ZDO_NWK_JOIN_IND 系统消息
25	启动 ZDO 状态改变	LED4 点亮，发送 ZDO_STATE_CHANGE 系统消息
26	广播数据包	LED5 闪烁，其他 LED 熄灭
评估板 A		
27	按下 S5， 发送组播数据包	LED8 闪烁
评估板 B		
28	路由器自动接收数据包	LED1 和 LED2 闪烁
评估板 C		
29	终端设备自动接收数据包	LED3 和 LED4 闪烁
30	按下 S6， 离开该组网络	LED7 点亮，LED3 和 LED4 停止闪烁