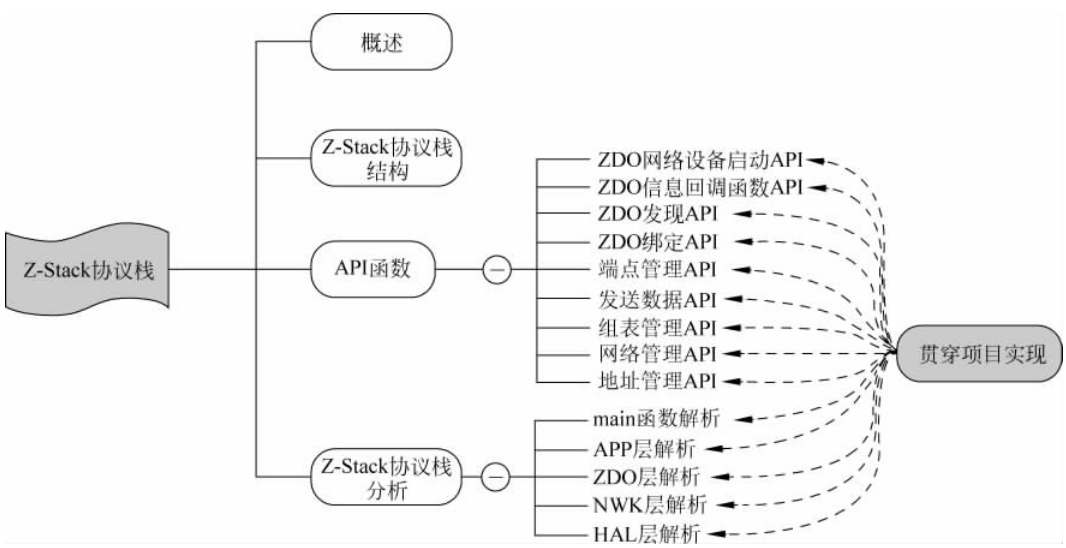
第3章 Z-Stack协议栈

任务驱动

【任务 3-1】 Z-Stack 协议栈下载、调试及组网。

学习导航 / 课程定位



本章目标

知　识　点	Listen(听)	Know(懂)	Do(做)	Revise(复习)	Master(精通)
Z-Stack 协议栈结构	★	★	★		
ZDO 网络设备启动 API	★	★	★		
ZDO 信息回调函数 API	★	★	★		
ZDO 发现 API	★	★	★		
ZDO 绑定 API	★	★	★	★	★
端点管理 API	★	★	★	★	★

续表

知 识 点	Listen(听)	Know(懂)	Do(做)	Revise(复习)	Master(精通)
发送数据 API	★	★	★	★	★
组表管理 API	★	★	★	★	
网络管理 API	★	★			
地址管理 API	★	★	★	★	
Z-Stack 协议栈分析	★	★	★	★	★

3.1 概述

Z-Stack 协议栈是德州仪器(TI)公司开发的完全符合 ZigBee 标准的解决方案,可以实现 ZigBee 的各种应用。Z-Stack 协议栈可以从 TI 的官方网站 www.ti.com 下载。

本书使用 ZStack-CC2530-2.2.2-1.3.0,从网站上下载软件安装工具之后,可以进行软件安装,如图 3-1 所示。

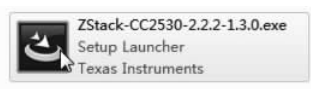


图 3-1 Z-Stack 协议栈安装软件

双击安装软件进行安装,安装界面如图 3-2 所示。

图 3-2 安装界面

随后会进入 Welcome to the ZStack-2.2.2 Installer 界面,单击 Next 按钮进行下一步安装,如图 3-3 所示。

在安装过程中需要接受 Z-Stack 安装协议,如图 3-4 所示。

单击 Finish 按钮完成安装,如图 3-5 所示。

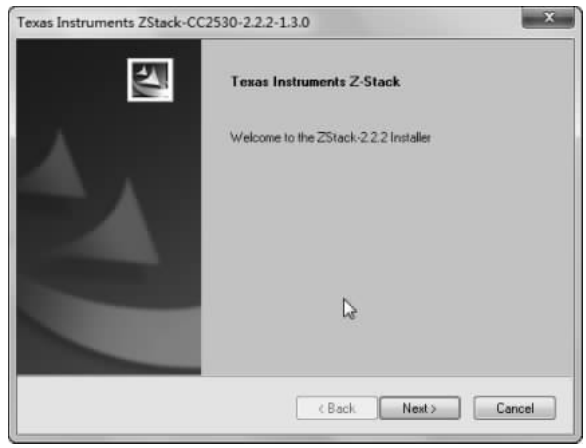


图 3-3　欢迎安装界面

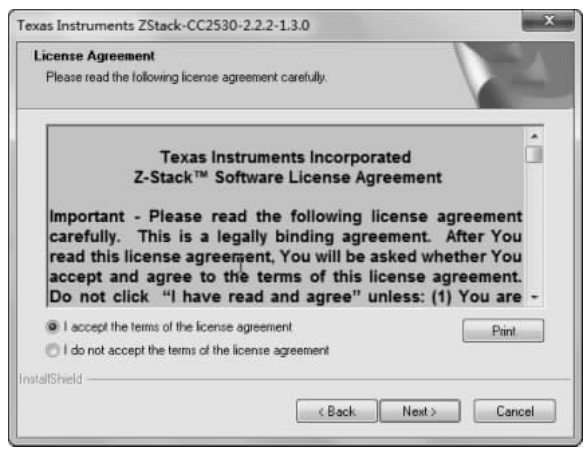


图 3-4　接受安装协议

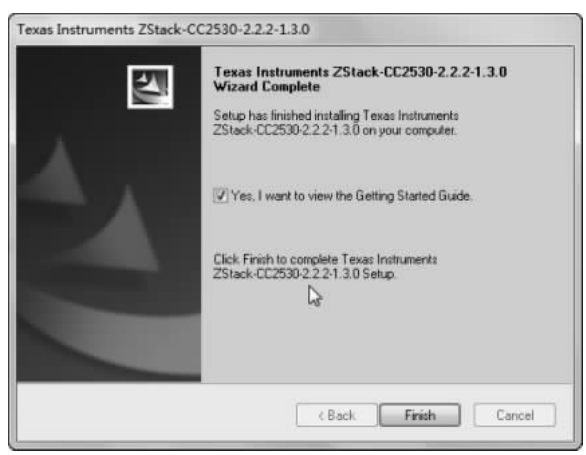


图 3-5　安装完成

本章将讲解 Z-Stack 协议栈架构、API 函数、协议栈各层分析以及运行机制。

3.2 Z-Stack 协议栈结构

Z-Stack 协议栈安装完成之后，打开 C:\Texas Instruments\ZStack-CC2530-2.2.2-1.3.0，可以看到 ZStack-CC2530-2.2.2-1.3.0 目录结构，如图 3-6 所示。

图 3-6 Z-Stack 协议栈文件目录

ZStack-CC2530-2.2.2-1.3.0 版本主要有 5 个文件：Components、Documents、Projects、Tools 和 Getting Started Guide-CC2530.pdf。

(1) Components 文件夹主要实现 Z-Stack 协议栈的各个功能部件。其内部包含了 7 个文件夹，如图 3-7 所示。

图 3-7 Components 文件夹目录

- hal 文件夹为硬件平台的抽象层，包含了硬件接口函数以及硬件驱动；
- mac 文件夹包含了实现 IEEE 802.15.4 标准协议所需代码的头文件，由于 Z-Stack 协议为半开源的协议栈，部分没有给出具体的源代码，而是以库文件的形式存在；
- mt 文件夹包含了 Z-Tools 调试功能所需要的文件；
- osal 文件夹包含了操作系统抽象层所需要的文件；
- services 文件夹包括 Z-Stack 提供的寻址服务和数据服务所需要的文件；
- stack 文件是 Components 文件夹最核心的部分，是 ZigBee 协议栈的具体实现部分，其下包含了应用框架 AF、网络层 NWK、简单应用接口 SAPI、安全层 SEC、系统文件 SYS、ZigBee 簇库 ZCL 和 ZigBee 设备对象 ZDO；
- zmac 文件夹包含了 Z-Stack 的 MAC 导出层文件。

(2) Documents 文件夹包含了对整个协议栈进行说明的文档信息，可以将它们作为参考手册根据需要阅读。

(3) Projects 文件夹包含了用于 Z-Stack 功能演示的各个项目实例，可供用户学习。

(4) Tools 文件夹包含了 Z-Stack 的一些配置文件，包括协调器、路由器、终端节点的配置文件。

(5) Getting Started Guide CC2530.pdf 文件是协议栈安装或卸载说明文档。

3.3 API 函数

API 函数的主要功能是 Z-Stack 协议栈内部的服务提供接口。Z-Stack 提供的 API 函数包括 ZDO 设备对象、应用框架 AF、应用支持子层和网络层。

- ZDO 设备对象 API 包括 ZDO 网络设备启动 API、ZDO 信息回调函数 API、ZDO 发现 API 和 ZDO 绑定 API;
- 应用框架 AF 层 API 包括端点管理 API 和发送数据 API;
- 应用支持子层 API 包括组表管理 API;
- 网络层 API 包括网络管理 API 和地址管理 API。

3.3.1 ZDO 网络设备启动 API

ZDOInitDevice()为 ZDO 网络设备启动 API 函数,在 ZDO 的 ZDApp.c 文件的 ZDApp_Init()函数中调用,其函数描述如下所示。

【函数 3-1】 ZDOInitDevice()

```
uint8 ZDOInitDevice(uint16 startDelay);
```

参数描述:

- startDelay——设备启动延时(ms)。

返回值:

- 网络状态为恢复——ZDO_INITDEV_RESTOPED_NETWORK_STATE;
- 网络状态为初始化——ZDO_INITDEV_NEW_NETWORK_START;
- 网络状态为未启动——ZDO_INITDEV_LEAVE_NOT_STARTED。

3.3.2 ZDO 信息回调函数 API

ZDO 信息回调函数 API 为 ZDO_RegisterForZDOMsg()和 ZDO_RemoveRegistedCB()。

1. ZDO_RegisterForZDOMsg()

ZDO_RegisterForZDOMsg()函数主要功能为注册请求或响应消息。用户调用该函数可以将接收到的无线传输信息复制到操作系统抽象层(OSAL 层)的某个任务。该任务接收到消息后可以自行解析此消息或通过 ZDO 解析函数解析此消息。

【函数 3-2】 ZDO_RegisterForZDOMsg()

```
ZStaus_t ZDO_RegisterForZDOMsg(uint8 taskID,uint16 clusterID);
```

参数描述:

- taskID——任务 ID,该任务发送 OSAL 消息;

- clusterID——簇 ID，在 ZDProfile.h 定义相关的簇。

返回值：

- ZStaus_t——返回值的类型或状态。

2. ZDO_RemoveRegistedCB()

ZDO_RemoveRegistedCB()函数主要功能为取消请求无线传输的消息。

【函数 3-3】 ZDO_RemoveRegistedCB()

```
ZStatus_t ZDO_RemoveRegistedCB(uint8 taskID,uint16 clusterID);
```

参数描述：

- taskID——任务 ID，此任务 ID 必须与 ZDO_RegisterForZDOMsg()所注册的任务 ID 相同；
- clusterID——簇 ID，该簇 ID 必须与 ZDO_RegisterForZDOMsg()所使用的簇 ID 相同。

返回值：

- ZStaus_t——返回值的类型或状态。

3.3.3 ZDO 发现 API

ZDO 发现 API 包含建立、发送 ZDO 设备以及服务的请求发现和响应。其 API 函数和 ZigBee 设备规范(ZigBee Device Profile Command，ZDP)命令如表 3-1 所示。

表 3-1 ZDO 发现 API

API 函数	ZDP 命令
ZDP_NwkAddrReq()	NWK_addr_req
ZDP_NWKAddrRsp()	NWK_addr_rsp
ZDP_IEEEAddrReq()	IEEE_addr_req
ZDP_IEEEAddrRsp()	IEEE_addr_rsp
ZDP_NodeDescReq()	Node_Desc_req
ZDP_NodeDescRsp()	Node_Desc_rsp
ZDP_PowerDescReq()	Power_Desc_req
ZDP_PowerDescRsp()	Power_Desc_rsp
ZDP_SimpleDescReq()	Simple_Desc_req
ZDP_SimpleDescRsp()	Simple_Desc_rsp
ZDP_ComplexDescReq()	Complex_Desc_req
ZDP_ActiveEPIFReq()	Active_EP_req
ZDP_ActiveEPIFRsp()	Active_EP_rsp
ZDP_MatchDescReq()	Match_Desc_req
ZDP_MatchDescRsp()	Match_Desc_rsp
ZDP_UserDescSet()	User_Desc_set
ZDP_UserDescConf()	User_Desc_conf
ZDP_UserDescReq()	User_Desc_req

续表

API 函数	ZDP 命令
ZDP_UserDescRsp()	User_Desc_rsp
ZDP_DeviceAnnce()	Device_annce
ZDP_ServerDiscReq()	System_Server_Discovery_req
ZDP_ServerDiscRsp()	System_Server_Discovery_rsp

1. ZDP_NwkAddrReq()

ZDP_NwkAddrReq()函数的主要功能是根据一个已知节点的 IEEE 地址访问远程节点的 16 位地址，并生成一个消息，并且此消息将作为一个广播消息发送至网络中的所有节点。

【函数 3-4】 ZDP_NwkAddrReq()

```
afStatus_t ZDP_NwkAddrReq(byte * IEEEAddress,
                          byte ReqType,
                          byte StartIndex,
                          byte SecuritySuite);
```

参数描述：

- IEEEAddress——远程节点的 IEEE 地址；
- ReqRype——返回的地址类型，有两种情况：一是只返回节点的短地址和扩展地址，即 ZDP_NWKADDR_REQTYPE_SINGLE；二是返回节点的短地址和扩展地址以及所有相关节点的短地址，即 ZDP_NWKADDR_REQTYPE_EXTENDED；
- StartIndex——响应节点的信息；
- SecuritSuite——安全要求。

返回值：

- afStatus_t——返回值的类型或状态。

2. ZDP_NWKAddrRsp()

ZDP_NWKAddrRsp()函数调用 ZDP_ADDRRsp()，用于建立和发送 16 位网络地址响应。

【函数 3-5】 ZDP_NWKAddrRsp()

```
AfStatus_t ZDP_NWKAddrRsp(byte TranSeq,
                          zAddrType_t * dstAddr,
                          byte Status,
                          byte * IEEEAddrRemoteDev,
                          byte ReqType,
                          uint16 nwkAddr,
                          byte NumAssocDev,
                          byte StartIndex,
                          uint16 * NWKAddrAssocDevList,
                          byte SecuritySuite);
```

参数描述：

- TranSeq——传输序列号；
- dstAddr——目的地址；
- Status——状态描述，有3种形式：发送或建立失败、发送或建立成功、未找到设备；
- IEEEAddrRemoteDev——目的节点的6位IEEE地址；
- ReqType——请求的类型；
- nwkAddr——目的节点的16位网络地址；
- NumAssocDev——与目的节点关联的节点数目；
- StartIndex——目的节点的响应信息；
- NWKAddrAssocDevList——与目的节点关联的节点的16位网络地址列表；
- SecuritySuite——安全要求。

返回值：

- ZStatus_t——返回值的类型或状态。

3. ZDP_IEEEAddrReq()

ZDP_IEEEAddrReq()函数主要功能是根据已知节点的16位网络地址访问64位IEEE地址的节点并生成一个消息，并将此消息直接单播发送至该远程节点。

【函数3-6】 ZDP_IEEEAddrReq()

```
afStatus_t ZDP_IEEEAddrReq(uint16 shaortAddr,
                           byte ReqType,
                           byte StartIndex,
                           byte SecuritySuite);
```

参数描述：

- ShortAddr——目的节点的16位网络地址；
- ReqType——返回的地址类型，有两种情况：一是只返回节点的短地址和扩展地址，即ZDP_NWKADDR_REQTYPE_SINGLE；二是返回节点的短地址、扩展地址以及所有相关节点的短地址，即ZDP_NWKADDR_REQTYPE_EXTENDED；
- StartIndex——与目的节点关联节点的16位网络地址列表；
- SecuritySuite——安全要求。

返回值：

- ZStatus_t——返回值的类型或状态。

4. ZDP_IEEEAddraRsp()

ZDP_IEEEAddrRsp()函数调用ZDP_ADDRRsp()这个宏定义，用于建立和发送IEEE地址响应。

【函数 3-7】 ZDP_IEEEAddrRsp()

```
afStatus_t ZDP_IEEEAddrRsp(byte TranSeq,
                           zAddrType * dstAddr,
                           byte Status,
                           byte * IEEEAddrRemoteDev,
                           byte ReqType,
                           uint16 nwkAddr,
                           byte NumAssocDev,
                           byte StartIndex,
                           uint16 * NWKAddrAssocDevList,
                           byte SecuritySuite);
```

参数描述：

- TranSeq——传输序列号；
- dstAddr——目的地址；
- Status——状态描述，有 3 种形式：发送或建立失败、发送或建立成功、未找到设备；
- IEEEAddrRemoteDev——目的节点的 6 位 IEEE 地址；
- ReqType——请求的类型；
- nwkAddr——目的节点的 16 位网络地址；
- NumAssocDev——与目的节点关联的节点数目；
- StartIndex——目的节点的响应信息；
- NWKAddrAssocDevList——与目的节点关联的节点的 16 位网络地址列表；
- SecuritySuite——安全要求。

返回值：

- ZStatus_t——返回值的类型或状态。

5. ZDP_NodeDescReq()

ZDP_NodeDescReq()函数主要功能是建立和发送一个节点描述符(Node Descriptor)请求到已知网络地址的远程节点。

【函数 3-8】 ZDP_NodeDescReq()

```
afStatus ZDP_NodeDescReq(zAddrType * dstAddr,
                         uint16 NWKAddrOfInterest,
                         byte SecuritySuite);
```

参数描述：

- dstAddr——目的地址；
- NWKAddrOfInterest——远程节点的 16 位网络地址；
- SecuritySuite——安全要求。

返回值：

- ZStatus_t——返回值的类型或状态。

6. ZDP_NodeDescMsg()

ZDP_NodeDescMsg()函数主要功能是响应节点描述符的请求。

【函数 3-9】 ZDP_NodeDescMsg()

```
afStatus_t ZDP_NodeDescMsg(byte TransSeq,
                           zAddrType_t * dstAddr,
                           byte Status,
                           uint16 nwkAddr,
                           NodeDescriptorFormat_t * pNodeDesc,
                           byte SecuritySuite);
```

参数描述：

- TranSeq——传输序号；
- dstAddr——目的地址；
- Status——状态描述，有 2 种形式：发送或建立失败或未找到设备；
- nwkAddr——已明确的远程节点的 16 位网络地址；
- pNodeDesc——节点描述符；
- SecuritySuite——安全要求。

返回值：

- ZStatus_t——返回值的类型或状态。

7. ZDP_PowerDescReq()

ZDP_PowerDescReq()函数主要功能是建立和发送一个电源描述符(Power Descriptor)请求到已明确网络地址的远程节点。

【函数 3-10】 ZDP_PowerDescReq()

```
afStatus_t ZDP_PowerDescReq(zAddrType * dstAddr,
                            uint16 NWKAddrOfInterest,
                            byte SecuritySuite);
```

参数描述：

- dstAddr——目的地址；
- NWKAddrOfInterest——远程节点的 16 位网络地址；
- SecuritySuite——安全要求。

返回值：

- ZStatus_t——返回值的类型或状态。

8. ZDP_PowerDescMsg()

ZDP_PowerDescMsg()函数主要功能为响应电源描述符的请求。

【函数 3-11】 ZDP_PowerDescMsg()

```
afStatus_t ZDP_PowerDescMsg(byte TransSeq,
                            zAddrType_t * dstAddr,
                            byte Status,
                            uint16 nwkAddr,
                            NodePowerDescriptorFormat_t * pPowerDesc,
                            byte SecuritySuite);
```

参数描述：

- TransSeq——传输序列号；
- dstAddr——目的地址；
- Status——状态描述，有 2 种形式：发送或建立失败或未找到设备；
- nwkAddr——已明确的远程节点的 16 位网络地址；
- pPowerDesc——电源描述符；
- SecuritySuite——安全要求。

返回值：

- ZStatus_t——返回值的类型或状态。

9. ZDP_SimpleDescReq()

ZDP_SimpleDescReq()函数主要功能是建立和发送一个简单描述符(Simple Descriptor)请求至已明确网络地址的远程节点。

【函数 3-12】 ZDP_SimpleDescReq()

```
afStatus_t ZDP_SimpleDescReq(zAddrType_t * dstAddr,
                             uint16 nwkAddr,
                             byte endpoint,
                             byte SecurityEable);
```

参数描述：

- dstAddr——目的地址；
- nwkAddr——远程节点的 16 位网络地址；
- endpoint——端点号；
- SecuritySuite——安全要求。

返回值：

- ZStatus_t——返回值的类型或状态。

10. ZDP_SimpleDescRsp()

ZDP_SimpleDescRsp()函数主要功能是响应简单描述符请求。

【函数 3-13】 ZDP_SimpleDescReq()

```
afStatus_t ZDP_SimpleDescRsp(byte TranSeq,
                             zAddrType_t * dstAddr,
                             byte Status,
                             SimpleDescriptionFormat_t * pSimpleDesc,
                             byte SecuritySuite);
```

参数描述：

- TranSeq——传输的序列号；
- dstAddr——目的地址；
- Status——状态描述，其取值形式有 3 种类型，如表 3-2 所示。
- pSimpleDesc——指向简单描述符的指针；
- SecuritySuite——安全选项。

表 3-2 Status 取值

意　义	值
SUCCESS	0
INVALID_EP	1
NOT_ACTIVE	2

返回值：

- ZStatus_t——返回值的类型或状态。

11. ZDP_ComplexDescReq()

ZDP_ComplexDescReq()函数主要功能是建立或发送复杂描述符请求。

【函数 3-14】 ZDP_ComplexDescReq()

```
afStatus_t ZDP_ComplexDescReq(zAddrType_t * dstAddr,
                              uint16 nwkAddr,
                              byte SecuritySuite);
```

参数描述：

- dstAddr——目的地址；
- nwkAddr——已知节点的 16 位网络地址；
- SecuritySuite——安全选项。

返回值：

- ZStatus_t——返回值的类型或状态。

12. ZDP_ActiveEPIFReq()

ZDP_ActiveEPIFReq()函数主要功能是建立并向已明确网络地址的远程节点发送一个活动端点请求。

【函数 3-15】 ZDP_ActiveEPIFReq()

```
afStatus_t ZDP_ActiveEPIFReq(zAddrType_t *dstAddr,
                             uint16 NWKAddrOfInterest,
                             byte SecuritySuite);
```

参数描述：

- dstAddr——目的地址；
- NWKAddrOfInterest——远程节点的 16 位网络地址；
- SecuritySuite——安全选项。

返回值：

- ZStatus_t——返回值的类型或状态。

13. ZDP_ActiveEPIFRsp()

ZDP_ActiveEPIFRsp()函数调用宏定义 ZDP_EPIFRsp()，主要功能是响应活动端点的请求。

【函数 3-16】 ZDP_ActiveEPIFRsp()

```
afStatus_t ZDP_ActiveEPIFResp(byte TranSeq,
                              zAddrType_t *dstAddr,
                              byte Status,
                              uint16 nwkAddr,
                              byte Count,
                              byte *pEPIntfList,
                              byte SecuritySuite);
```

参数描述：

- TranSeq——传输序列号；
- dstAddr——目的地址；
- Status——状态描述，其取值形式有 2 种类型，如表 3-3 所示。
- nwkAddr——已知节点的 16 位网络地址；
- Count——活动端点个数；
- pEPIntfList——活动端点列表；
- SecuritySuite——安全选项。

表 3-3 Status 取值

意　义	值
ZDP_SUCCESS	0
ZDP_DEVICE_NOT_FOUND	1

返回值：

- ZStatus_t——返回值的类型或状态。

14. ZDP_MatchDescReq()

ZDP_MatchDescReq()函数主要功能是建立并发送一个匹配描述符请求，查询与本地节点的输入或输出簇列表相匹配的远程节点。

【函数 3-17】 ZDP_MatchDescReq()

```
afStatus_f ZDP_MatchDescReq(zAddrType_t * dstAddr,
                            uint16 nwkAddr,
                            uint16 ProfileID,
                            byte NumInCluster,
                            byte * InClusterList,
                            byte NumOutCluster,
                            byte * OutClusterList,
                            byte SecuritySuite);
```

参数描述：

- dstAddr——目的地址；
- nwkAddr——已知节点的 16 位网络地址；
- ProfileID——应用规范 ID；
- NumInCluster——输入簇个数；
- InClusterList——输入簇列表；
- NumOutCluster——输出簇个数；
- OutClusterList——输出簇列表；
- SecuritySuite——安全选项。

返回值：

- ZStatus_t——返回值的类型或状态。

15. ZDP_MatchDescRsp()

ZDP_MatchDescRsp()函数主要功能是响应匹配描述符请求。

【函数 3-18】 ZDP_MatchDescRsp()

```
afStatus_t ZDP_MatchDescRsp(byte TranSeq,
                            zAddrType_t * dstAddr,
                            byte Status,
                            uint16 nwkAddr,
                            byte Count,
                            byte * pEPIntfList,
                            byte SecuritySuite);
```

参数描述：

- TranSeq——传输序列号；

- dstAddr——目的节点地址；
- Status——状态描述，其取值形式有 2 种类型，如表 3-4 所示。
- nwkAddr——已知节点的 16 位网络地址；
- Count——活动端点个数；
- pEPIntfList——活动端点列表；
- SecuritySuite——安全选项。

表 3-4　Status 取值

意　义	值
ZDP_SUCCESS	0
ZDP_DEVICE_NOT_FOUND	1

返回值：

- ZStatus_t——状态。

16. ZDP_DeviceAnnce()

ZDP_DeviceAnnce()函数为 ZigBee 终端节点建立和发送一个 End_Device_Annce()命令，通知网络中的其他 ZigBee 节点，此命令包含终端设备的 16 位网络地址和 64 位 IEEE 地址以及地址容量。当接收方接收到 End_Device_Annce()命令时，将检查 IEEE 地址并更新相应的网络地址。

【函数 3-19】　ZDP_Device_Annce()

```
afStatus_t ZDP_Device_Annce(uint16 nwkAddr,
                            byte * IEEEAddr,
                            byte capabilities,
                            byte SecurityEable);
```

参数描述：

- nwkAddr——本地节点的 16 位网络地址；
- IEEEAddr——本地节点的 64 位 IEEE 地址；
- capabilities——本地节点的容量；
- SecurityEable——安全选项。

返回值：

- ZStatus_t——返回值的类型或状态。

3.3.4　ZDO 绑定 API

绑定是指两个节点在应用层上建立起来的一种逻辑链路。在同一个节点上可以建立多个绑定服务，分别对应不同种类的数据包。

ZDO 绑定 API 主要功能是建立和发送 ZDO 绑定请求和响应。以下介绍的 ZDO 绑定 API 函数的所有绑定表需建立在 ZigBee 协调器中，因此只有协调器可以接收绑定请求。绑定 API 函数列表如表 3-5 所示。

表 3-5 ZDO 绑定 API

ZDO 绑定 API	ZDO 绑定服务命令
ZDP_EndDeviceBindReq()	End_Device_Bind_req
ZDP_EndDeviceBindRsp()	End_Device_Bind_rsp
ZDP_BindReq()	Bind_req
ZDP_BindRsp()	Bind_rsp
ZDP_UnbindReq()	Unbind_req
ZDP_UnbindRsp()	Unbind_rsp

1. ZDP_EndDeviceBindReq()

ZDP_EndDeviceBindReq()函数的主要功能是建立和发送一个终端设备手动绑定请求。在手动绑定建立之后,可以间接发送信息至协调器,协调器将发送此消息至绑定节点。

【函数 3-20】 ZDP_EndDeviceBindReq()

```
afStatus_t ZDP_EndDeviceReq(zAddrType_t * dstAddr,
                            uint16 LocalCoordinator,
                            byte ep,
                            uint16 ProfileID,
                            byte NumInClusters,
                            byte * InClusterList,
                            byte NumOutClusters,
                            byte * OutClusterList,
                            byte SecuritySuite);
```

参数描述:

- dstAddr——目的地址;
- LocalCoordinator——已知协调器的 16 位网络地址;
- ep——端点;
- ProfileID——应用规范 ID;
- NumInClusters——输入簇个数;
- InClustersList——输入簇列表;
- NumOutClusters——输出簇个数;
- OutClustersList——输出簇列表;
- SecuritySuite——安全选项。

返回值:

- ZStatus_t——返回值的类型或状态。

2. ZDP_EndDeviceBindRsp()

ZDP_EndDeviceBindRsp()函数直接调用宏定义 ZDP_SendData(),主要功能是响应终端设备绑定请求。

【函数 3-21】 ZDP_EndDeviceBindRsp()

```
afStatus_t ZDP_EndDeviceBindRsp(byte TranSeq,
                                zAddrType_t * dstAddr,
                                byte Status,
                                byte SecurityEable);
```

参数描述：

- TranSeq——传输序列号；
- dstAddr——目的地址；
- Status——状态描述，其取值形式有 4 种类型，如表 3-6 所示。
- SecurityEable——安全选项。

表 3-6 Status 取值

意　义	值
SUCCESS	0
NOT_SUPPORT	1
TIMEOUT	2
NO_MATCH	3

返回值：

- ZStatus_t——返回值的类型或状态。

3. ZDP_BindReq()

ZDP_BindReq()函数主要功能是建立和发送一个绑定请求。

【函数 3-22】 ZDP_BindReq()

```
afStatus_t ZDP_BindReq(zAddrType_t * dstAddr,
                       byte * SourceAddr,
                       byte srcEP,
                       byte ClusterID,
                       byte * DestinationAddr,
                       byte DstEP,
                       byte SecuritySuite);
```

参数描述：

- dstAddr——目的地址；
- SourceAddr——本地节点 64 位 IEEE 地址；
- srcEP——本地节点的端点；
- ClusterID——绑定簇的 ID 号；
- DestinationAddr——远程节点的 64 位 IEEE 地址；
- DstEP——远程节点的端点；
- SecuritySuite——安全选项。

返回值：

- ZStatus_t——返回值的类型或状态。

4. ZDP_BindRsp()

ZDP_BindRsp()函数主要功能是响应绑定请求。

【函数 3-23】 ZDP_BindRsp()

```
afStatus_t ZDP_BindRsp(byte TranSeq,
                       zAddrType_t * dstAddr,
                       byte Status,
                       byte SecurityEable);
```

参数描述：

- TranSeq——传输序列号；
- dstAddr——目的地址；
- Status——状态描述，其取值形式有 3 种类型，如表 3-7 所示。
- SecurityEable——安全选项。

表 3-7 ZDP_BindRsp()——Status 取值

意　义	值
SUCCESS	0
NOT_SUPPORT	1
TABLE_FULL	2

返回值：

- ZStatus_t——返回值的类型或状态。

5. ZDP_UnbindReq()

ZDP_UnbindReq()函数的功能是建立和发送一个解除绑定的请求，通知协调器解除设备的绑定。

【函数 3-24】 ZDP_UnbindReq()

```
afStatus_t ZDP_UnbindReq(zAddrType_t * dstAddr,
                         byte * SourceAddr,
                         byte SrcEP,
                         byte ClusterID,
                         byte * DestinationAddr,
                         byte DstEP,
                         byte SecuritySuite);
```

参数描述：

- dstAddr——目的地址；
- SourceAddr——本地节点 64 位 IEEE 地址；

- SrcEP——本地节点的端点；
- ClusterID——绑定簇的 ID 号；
- DestinationAddr——远程节点的 64 位 IEEE 地址；
- DstEP——远程节点的端点；
- SecuritySuite——安全选项。

返回值：

- ZStatus_t——返回值的类型或状态。

6. ZDP_UnbindRsp()

ZDP_UnbindRsp()函数的主要功能是响应解除绑定请求。

【函数 3-25】 ZDP_UnbindRsp()

```
afStatus_t ZDP_UnbindRsp(byte Transeq,
                         zAddrType_t *dstAddr,
                         byte Status,
                         byte SecurityEable);
```

参数描述：

- TranSeq——传输序列号；
- dstAddr——目的地址；
- Status——状态描述，其取值形式有 3 种类型，如表 3-8 所示。
- SecurityEable——安全选项。

表 3-8 ZDP_UnbindRsp——Status 取值

意　义	值
SUCCESS	0
NOT_SUPPORT	1
NO_ENTRY	2

返回值：

- afStatus_t——返回值的类型或状态。

3.3.5 端点管理 API

在 ZigBee 网络中每个设备都是一个节点，每个节点都具有唯一的 64 位 IEEE 地址和一个 16 位网络地址。在向某一个指定节点发送信息时，网络中的发送节点发送数据时必须指定目标节点的短地址，数据才能被接收，即使是广播通信方式，也有一个 0xffff 的短地址。每个节点有 241 个端点，其中端点 0 被 ZDO 层使用，端点 1～240 由应用程序分配使用。在 ZigBee 网络中，应用程序必须登记注册一个或多个端点，才能接收和发送数据。以下将详细讲解端点管理 API 函数。

1. afRegister()

afRegister()函数主要功能是为节点登记注册一个新的端点。每个端点在注册时，应用

程序将调用此函数。

【函数 3-26】 afRegister()

```
afStatus_t afRegister(endPointDesc_t * epDesc);
```

参数描述：

- epDesc——端点描述符。

返回值：

- afStatus_t——返回值的类型或状态。

2. afFindEndPointDesc()

afFindEndPointDesc()函数主要功能是从一个端点查找相应的端点描述符。

【函数 3-27】 afFindEndPointDesc()

```
endPointDesc_t * afFindEndPointDesc(byte endPoint);
```

参数描述：

- endPoint——端点号。

返回值：

- endPointDesc_t * ——端点描述符。

3. afFindSimpleDesc()

afFindSimpleDesc()函数用来从一个端点查找相应的简单描述符。

【函数 3-28】 afFindSimpleDesc()

```
Byte afFindSimpleDesc(SimpleDescriptionFormat_t ** ppDesc,byte EP);
```

参数描述

- ppDesc——简单描述符；
- EP——端点号。

返回值：

- 简单描述符内存地址。

4. afGetMatch()

afGetMatch()函数主要功能是响应 ZDO 匹配描述符请求，可以使用此函数获得 ZDO 匹配描述符响应信息的设置。

【函数 3-29】 afGetMatch()

```
uint8 afGetMatch(uint8 ep);
```

参数描述：

- ep——获得的 ZDO 匹配描述符响应信息的端点号。

返回值：

- true——允许响应；
- false——不允许响应或未找到。

5. afSetMatch()

afSetMatch()函数主要功能是允许响应 ZDO 匹配描述符请求，根据此函数改变状态。

【函数 3-30】 afSetMatch()

```
uint8 afSetMatch(uint8 ep,uint8 action);
```

参数描述：

- ep——改变 ZDO 匹配描述符响应信息的端点号；
- action——响应行为描述，具体有两种形式，即 true 为允许响应，false 为不允许响应。

返回值：

- true——成功；
- false——未找到匹配值。

3.3.6 发送数据 API

发送数据 API 为 AF_DataRequest()，主要功能是发送数据。

【函数 3-31】 AF_DataRequest()

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

参数描述：

- dstAddr——目的地址；
- srcEP——源端点；
- cID——簇 ID；
- len——发送数据字节长度；
- buf——准备发送的数据；
- transID——传输序列号；
- option——信息发送选项，其取值如表 3-9 所示。
- radius——最大跳数。

表 3-9 option 取值

意 义	值
AF_ACK_REQUEST	0x10
AF_DISCV_ROUTE	0x20
AF_EN_SECURITY	0x40
AF_SKIP_ROUTING	0x80

返回值：

- afStatus_t——返回值的类型或状态。

3.3.7 组表管理 API

组表是一个分配在 RAM 中的一个链表，随着组表个数的增加，所分配的堆栈也随之增长。表的最大尺寸由定义在 Z-Stack 协议栈的配置文件 f8wConfig.cfg 获得。以下内容讲解组表管理 API。

1. aps_AddGroup()

aps_AddGroup()函数的主要功能是在端点中增加一个组至组表中。

【函数 3-32】 aps_AddGroup()

```
ZStatus_t aps_AddGroup(uint8 endpoint,aps_Group_t * group);
```

参数描述：

- endpoint——新增组的端点；
- group——增加至组表的组号和组名。

返回值：两种返回值。

- 成功——ZSuccuss；
- 失败——有 3 种情况：添加的条目重复即 ZApsDuplicateEntry、表满即 ZApsTableFull、系统内存错误即 ZMemError。

2. aps_RemoveGroup()

aps_RemoveGroup()函数的主要功能是从组表中删除一个组。

【函数 3-33】 aps_RemoveGroup()

```
ZStatus_t apsRemoveGroup(uint8 endpoint,uint16 groupID);
```

参数描述：

- endpoint——删除组的端点；
- groupID——删除组的 ID 号。

返回值：两种返回值。

- 成功——true；

- 失败——false。

3. aps_FindGroup()

aps_FindGroup()函数主要功能是根据端点和组号从组表中查找到一个组。

【函数 3-34】 aps_FindGroup()

```
aps_Group_t * aps_FindGroup(uint8 endpoint,uint16 groupID);
```

参数描述：

- endpoint——查找组的端点；
- groupID——查找组的 ID 号。

返回值：两种返回值。

- 找到——指向组的指针；
- 失败——NULL。

3.3.8 网络管理 API

网络管理 API 主要有组建网络请求 API、发现网络请求 API、加入网络请求 API、重新加入网络请求 API、孤立节点连接父节点请求 API 和启动路由请求 API 函数。

1. NLME_NetworkFormationRequest()

NLME_NetworkFormationRequest()函数主要功能是请求组建一个新网络并允许自身成为该网络的 ZigBee 协调器。这个行为的结果返回到 ZDO_NetworkFormationConfirmCB()中。

【函数 3-35】 NLME_NetworkFormationRequest()

```
ZStatus_t NLME_NetworkFormationRequest( uint16 PanId,
                                        uint8 * ExtendedPANID,
                                        uint32 ScanChannels,
                                        byte ScanDuration,
                                        byte BeaconOrder,
                                        byte SuperframeOrder,
                                        byte BatteryLifeExtension );
```

参数描述：

- PanId——个域网 ID 号，有效值：0～0xFFFE，如果为 0xFFFF，则网络层将选择 PANID 供网络使用；
- ExtendedPANID——扩展的个域网 ID 号；
- ScanChannels——扫描的频道，11～26；
- ScanDuration——扫描时间；
- BeaconOrder——该值为 BEACON_ORDER_NO_BEACONS；
- SuperframeOrder——该值为 BEACON_ORDER_NO_BEACONS；

- BatteryLifeExtension——电池寿命延长模式。

返回值：

- ZStatus_t——返回请求的结果成功或失败。

2. NLME_NetworkDiscoveryRequest()

NLME_NetworkDiscoveryRequest()函数主要功能请求网络层寻找相邻的路由器。这个函数应该在加入并执行网络扫描之前调用。这个行为的结果返回到 ZDO_NetworkDiscoveryConfirmCB()中，返回结果包含发现的路由个数和网络描述列表。

【函数 3-36】 NLME_NetworkDiscoveryRequest()

```
ZStatus_t NLME_NetworkDiscoveryRequest( uint32 ScanChannels, byte ScanDuration );
```

参数描述：

- ScanChannels——扫描的频道，11～26；
- ScanDuration——扫描时间。

返回值：

- ZStatus_t——状态。

3. NLME_JoinRequest()

NLME_JoinRequest()函数请求节点将自己加入到一个网络中。这个行为的结果返回到 ZDO_JoinConfirmCB()中。

【函数 3-37】 NLME_JoinRequest()

```
ZStatus_t NLME_JoinRequest( uint8 *ExtendedPANID,
                            uint16 PanId,
                            byte Channel,
                            byte CapabilityInfo);
```

参数描述：

- ExtendedPANID——试图加入的网络的扩展 PAN ID 号；
- PanId——试图加入的网络的 PAN ID 号；
- Channel——频道号，11～26；
- CapabilityInfo——加入网络设备的能力。

返回值：

- ZStatus_t——请求成功和失败的状态。

4. NLME_ReJoinRequest()

NLME_ReJoinRequest()函数请求节点重新加入到一个已经加入过的网络中。这个行为的结果返回到 ZDO_JoinConfirmCB()中。

【函数 3-38】 NLME_ReJoinRequest()

```
ZStatus_t NLME_ReJoinRequest( uint8 *ExtendedPANID, byte Channel);
```

参数描述：

- ExtendedPANID——试图加入的网络的扩展 PAN ID 号；
- Channel——频道号，11～26。

返回值：

- ZStatus_t——状态。

5. NLME_OrphanJoinRequest()

NLME_OrphanJoinRequest()函数主要功能是请求孤立节点连接到一个父节点。这个行为的结果返回到 ZDO_JoinConfirmCB()中。

【函数 3-39】 NLME_OrphanJoinRequest()

```
ZStatus_t NLME_OrphanJoinRequest( uint32 ScanChannels, byte ScanDuration);
```

参数描述：

- ScanChannels——频道号，11～26；
- ScanDuration——扫描时间。

返回值：

- ZStatus_t——请求成功和失败的状态。

6. NLME_StartRouterRequest()

NLME_StartRouterRequest()函数主要功能是以路由器身份启动，这个行为的结果返回到 ZDO_StartRouterConfirmCB()中。

【函数 3-40】 NLME_StartRouterRequest()

```
ZStatus_t NLME_StartRouterRequest( byte BeaconOrder,
                                   byte SuperframeOrder,
                                   byte BatteryLifeExtension);
```

参数描述：

- BeaconOrder——该值为 BEACON_ORDER_NO_BEACONS；
- SuperframeOrder——该值为 BEACON_ORDER_NO_BEACONS；
- BatteryLifeExtension——有两种类型，分别是 TRUE 或者 FALSE，TRUE 为电池供电，FALSE 为电源供电。

返回值：

- ZStatus_t——返回值的类型或状态。

3.3.9 地址管理 API

地址管理的主要功能是查询和存储在本地设备的远程地址，主要 API 函数有：获得节

点自身 64 位 IEEE 地址 API 函数、获得节点自身 16 位网络地址 API 函数、获得父节点 64 位 IEEE 地址 API 函数、判断地址是否为有效的广播地址 API 函数和处理有效的广播地址 API 函数。

1. NLME_GetExtAddr()

NLME_GetExtAddr()函数主要功能是得到节点自身的 64 位 IEEE 地址。

【函数 3-41】 NLME_GetExtAddr()

```
byte * NLME_GetExtAddr( void );
```

返回值：

- 指向 64 位 IEEE 地址的指针。

2. NLME_GetShortAddr()

NLME_GetShortAddr()函数主要功能是得到节点自身的 16 位网络地址。

【函数 3-42】 NLME_GetShortAddr()

```
byte * NLME_GetShortAddr( void );
```

返回值：

- 16 位网络地址。

3. NLME_GetCoordShortAddr()

NLME_GetCoordShortAddr()函数主要功能是得到父节点的 16 位网络地址。

【函数 3-43】 NLME_GetCoordShortAddr()

```
byte * NLME_GetCoordShortAddr( void );
```

返回值：

- 指向父节点 64 位 IEEE 地址的指针。

4. NLME_IsAddressBroadcast()

NLME_IsAddressBroadcast()函数主要功能是评估提供的地址是否是一个有效的广播地址。

【函数 3-44】 NLME_IsAddressBroadcast()

```
addr_filter_t NLME_IsAddressBroadcast(uint16 shortAddress);
```

参数描述：

- shortAddress——被测试的 16 位网络地址。

返回值：

- ADDR_NOT_BCAST；

- ADDR_BCAST_FOR_ME；
- ADDR_BCAST_NOT_ME。

5. NLME_SetBroadCastFilter()

NLME_SetBroadCastFilter()函数主要功能是设置掩码，用于处理有效的广播地址。

【函数 3-45】 NLME_SetBroadCastFilter()

```
void NLME_SetBroadCastFilter(byte capabilities);
```

参数描述：

- capabilities——用于确定该设备处理的广播信息类型。

返回值：

- 空。

3.4 Z-Stack 协议栈分析

Z-Stack 协议栈符合 ZigBee 协议，由物理层、MAC 层、网络层和应用层组成。由于 Z-Stack 协议栈是一个半开源的协议栈，MAC 层和网络层的部分源代码是非开源的，因此本书中的开源部分主要包括 main 函数、APP 层、ZDO 层、NWK 层和 HAL 层。

3.4.1 main 函数解析

安装完成协议栈之后，打开 TI 官方的案例 SampleApp，打开目录如下：Texas Instruments→ZStack-CC2530-2.2.2-1.3.0→Projects→zstack→Samples→SampleApp→CC2530DB→SampleApp.eww。打开工程后在 ZMain 文件夹中找到 ZMain.c 函数，如图 3-8 所示。

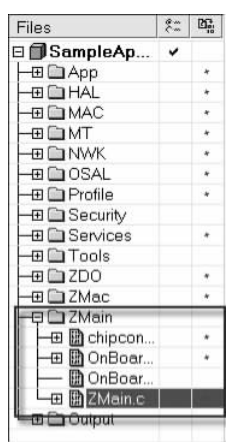


图 3-8 ZMain.c 文件

在主函数中对硬件设备、MAC 层、网络层等做相应的初始化，初始化完成之后运行操作系统，如代码 3-1 所示。

【代码 3-1】 main()

```
int main( void )
{
    /* 关闭所有中断 */
    osal_int_disable( INTS_ALL );
    /* 初始化硬件设备 */
    HAL_BOARD_INIT();
    /* 电源检测 */
    zmain_vdd_check();
    /* 初始化 I/O */
    InitBoard( OB_COLD );
    /* 初始化硬件抽象层 HAL 驱动 */
    HalDriverInit();
    /* 初始化 NV */
    osal_nv_init( NULL );
    /* 初始化 MAC */
    ZMacInit();
    /* 确定 64 位 IEEE 地址 */
    zmain_ext_addr();
    /* 初始化 NV 向量 */
    zgInit();
#ifndef NONWK
    /* AF 层初始化/
    afInit();
#endif
    /* 初始化任务 */
    osal_init_system();
    /* 开启中断 */
    osal_int_enable( INTS_ALL );
    /* 硬件 I/O 初始化完毕 */
    InitBoard( OB_READY );
    // Display information about this device
    zmain_dev_info();
    /*  如果定义了 LCD,初始化 LCD */
#ifdef LCD_SUPPORTED
    zmain_lcd_init();
#endif
#ifdef WDT_IN_PM1
    /*  如果定义了看门狗,看门狗使能 */
    WatchDogEnable( WDTIMX );
#endif
    /* 操作系统运行 */
    osal_start_system();
    return 0;
}
```

其中 osal_int_disable()函数主要功能是禁用所有中断，中断被禁用后，与该中断相关的服务例程将不再被调用。此函数在 OSAL.c 文件中实现，如代码 3-2 所示。

【代码 3-2】 osal_int_disable()

```
uint8 osal_int_disable( uint8 interrupt_id )
{
    /* 判断 ID 是否为中断 ID */
    if ( interrupt_id == INTS_ALL )
    {
        /* 关掉所有中断 */
        HAL_DISABLE_INTERRUPTS();
        /* 中断关闭成功,返回 SUCCESS */
        return ( SUCCESS );
    }
    else
    {
        /* 如果 ID 与 INST_ALL 不同,返回值为 INVALID_INTERRUPT_ID */
        return ( INVALID_INTERRUPT_ID );
    }
}
```

3.4.2 APP 层解析

Z-Stack 协议栈的 APP 层主要功能是实现用户定义的事件，APP 层由 5 个文件组成，分别是 OSAL_SampleApp.c 文件、SampleApp.c 文件、SampleApp.h 文件、SampleAppHw.c 文件和 SampleAppHw.h 文件。其中 SampleAppHw.c 文件和 SampleAppHw.h 文件主要功能是通过读取跳线来确定设备是否为协调器，由于本书中不涉及此项功能，在此不详细介绍。其 Z-Stack 协议栈的 APP 层如图 3-9 所示。

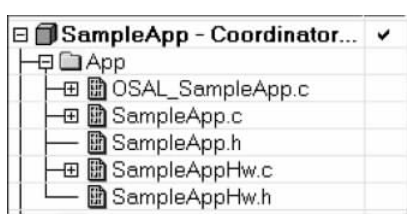


图 3-9 Z-Stack 协议栈 APP 层

1. OSAL_SampleApp.c

OSAL_SampleApp.c 文件中主要实现任务初始化以及事件处理函数，其中 osalInitTask()函数主要负责任务初始化；tasksArr[]数组中给出了相应任务的处理函数。

osalInitTasks()函数主要功能是为任务分配空间，另外调用了任务初始化函数，如代码 3-3 所示。

【代码 3-3】 osalInitTasks()

```
void osalInitTasks( void )
{
    uint8 taskID = 0;
    /* 为任务分配空间 */
    tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
    /* 任务初始化 */
    osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));
    /* MAC 层任务初始化 */
    macTaskInit( taskID++ );
    /* 网络层任务初始化 */
    nwk_init( taskID++ );
    /* 硬件抽象层任务初始化 */
    Hal_Init( taskID++ );
    /* 如果定义了 MT_TASK,MT 层任务初始化 */
#if defined( MT_TASK )
    MT_TaskInit( taskID++ );
#endif
    /* APS 层任务初始化 */
    APS_Init( taskID++ );
    /* 如果定义了 ZIGBEE_FRAGMENTATION,APSF 任务初始化 */
#if defined ( ZIGBEE_FRAGMENTATION )
    APSF_Init( taskID++ );
#endif
    /* ZDApp 任务初始化 */
    ZDApp_Init( taskID++ );
    /* ZDNwkMgr 初始化 */
#if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
    ZDNwkMgr_Init( taskID++ );
#endif
    /* SampleApp 初始化 */
    SampleApp_Init( taskID );
}
```

tasksArr 中存放了事件处理回调函数,在此数组中的事件处理函数与 osalInitTasks() 中的任务初始化函数是一一对应的,如代码 3-4 所示。

【代码 3-4】 tasksArr[]

```
const pTaskEventHandlerFn tasksArr[] = {
    /* MAC 事件处理函数 */
    macEventLoop,
    /* 网络层事件处理函数 */
    nwk_event_loop,
    /* 硬件抽象层事件处理函数 */
    Hal_ProcessEvent,
#if defined( MT_TASK )
    /* MT 层事件处理函数 */
```

```
    MT_ProcessEvent,
#endif
    /* APS 层事件处理函数 */
    APS_event_loop,
#if defined ( ZIGBEE_FRAGMENTATION )
    /* APSF 层事件处理函数 */
    APSF_ProcessEvent,
#endif
    /* ZDApp 层事件处理函数 */
    ZDApp_event_loop,
#if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
    /* 网络层事件处理函数 */
    ZDNwkMgr_event_loop,
#endif
    /* 应用层事件处理函数(用户自定义) */
    SampleApp_ProcessEvent
};
```

2. SampleApp.c

SamplApp.c 文件主要有两个功能：对应用层用户定义的任务进行初始化；调用事件处理函数。此文件中的函数有用户任务初始化函数 SampleApp_Init()、任务处理函数 SampleApp_ProcessEvent()、按键处理事件 SampleApp_HandleKeys()、数据发送函数 SampleApp_SendFlashMessage()、数据发送函数 SampleApp_SendPeriodicMessage()和数据接收函数 SampleApp_MessageMSGCB()。

SampleApp_Init()函数主要功能是实现用户自定义任务的初始化，包括任务 ID、网络状态、传输序列号、数据发送地址、端点等信息初始化，如代码 3-5 所示。

【代码 3-5】 SampleApp_Init()

```
void SampleApp_Init( uint8 task_id )
{
    /* 任务 ID */
    SampleApp_TaskID = task_id;
    /* 网络状态初始化为 DEV_INIT */
    SampleApp_NwkState = DEV_INIT;
    /* 传输序列号初始化为 0 */
    SampleApp_TransID = 0;
#if defined ( BUILD_ALL_DEVICES )
    if ( readCoordinatorJumper() )
        zgDeviceLogicalType = ZG_DEVICETYPE_COORDINATOR;
    else
        zgDeviceLogicalType = ZG_DEVICETYPE_ROUTER;
#endif

#if defined ( HOLD_AUTO_START )
    ZDOInitDevice(0);
```

```
#endif
    /* SampleApp_Periodic_DstAddr 地址模式初始化为广播地址 */
    SampleApp_Periodic_DstAddr.addrMode = (afAddrMode_t)AddrBroadcast;
    /* SampleApp_Periodic_DstAddr 端点初始化 */
    SampleApp_Periodic_DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
    /* SampleApp_Periodic_DstAddr 短地址为 0xFFFF,即 */
    SampleApp_Periodic_DstAddr.addr.shortAddr = 0xFFFF;

    SampleApp_Flash_DstAddr.addrMode = (afAddrMode_t)afAddrGroup;
    SampleApp_Flash_DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
    SampleApp_Flash_DstAddr.addr.shortAddr = SAMPLEAPP_FLASH_GROUP;

    SampleApp_epDesc.endPoint = SAMPLEAPP_ENDPOINT;
    SampleApp_epDesc.task_id = &SampleApp_TaskID;
    SampleApp_epDesc.simpleDesc
                          = (SimpleDescriptionFormat_t *)&SampleApp_SimpleDesc;
    SampleApp_epDesc.latencyReq = noLatencyReqs;
    afRegister( &SampleApp_epDesc );

    RegisterForKeys( SampleApp_TaskID );

    SampleApp_Group.ID = 0x0001;
    osal_memcpy( SampleApp_Group.name, "Group 1", 7 );
    aps_AddGroup( SAMPLEAPP_ENDPOINT, &SampleApp_Group );

#if defined ( LCD_SUPPORTED )
    HalLcdWriteString( "SampleApp", HAL_LCD_LINE_1 );
#endif
}
```

SampleApp_ProcessEvent()函数主要功能是实现具体事件处理,比如系统消息事件、按键事件、接收信息事件、网络状态改变事件和定时事件,如代码 3-6 所示。

【代码 3-6】 SampleApp_ProcessEvent()

```
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
  afIncomingMSGPacket_t *MSGpkt;
  (void)task_id;
  if ( events & SYS_EVENT_MSG )
  {
    MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );
    while ( MSGpkt )
    {
      switch ( MSGpkt->hdr.event )
      {
        case KEY_CHANGE:
           SampleApp_HandleKeys( ((keyChange_t *)MSGpkt)->state, ((keyChange_t *)
MSGpkt)->keys );
```

```
            break;

          case AF_INCOMING_MSG_CMD:
            SampleApp_MessageMSGCB( MSGpkt );
            break;

          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) )
            {
              osal_start_timerEx( SampleApp_TaskID,
                                  SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
                                  SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
            }
            else
            {
            }
            break;
          default:
            break;
        }
        osal_msg_deallocate( (uint8 *)MSGpkt );
        MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );
      }
      return (events ^ SYS_EVENT_MSG);
    }
    if ( events & SAMPLEAPP_SEND_PERIODIC_MSG_EVT )
    {
      SampleApp_SendPeriodicMessage();
      osal_start_timerEx( SampleApp_TaskID, SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
          (SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT + (osal_rand() & 0x00FF)) );
      return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);
    }
    return 0;
  }
```

SampleApp_HandleKeys()函数主要实现按键事件的处理，如果按键按下将触发相应的事件，如代码 3-7 所示。

【代码 3-7】 SampleApp_HandleKeys()

```
void SampleApp_HandleKeys( uint8 shift, uint8 keys )
{
  (void)shift;

  if ( keys & HAL_KEY_SW_1 )
  {
```

```
    SampleApp_SendFlashMessage( SAMPLEAPP_FLASH_DURATION );
  }

  if ( keys & HAL_KEY_SW_2 )
  {
    aps_Group_t * grp;
    grp = aps_FindGroup( SAMPLEAPP_ENDPOINT, SAMPLEAPP_FLASH_GROUP );
    if ( grp )
    {
    aps_RemoveGroup( SAMPLEAPP_ENDPOINT, SAMPLEAPP_FLASH_GROUP );
    }
    else
    {
      aps_AddGroup( SAMPLEAPP_ENDPOINT, &SampleApp_Group );
    }
  }
}
```

SampleApp_MessageMSGCB()函数主要实现数据接收功能，数据的接收通过判断簇ID来与发送端发送的数据进行匹配，如代码3-8所示。

【代码3-8】 SampleApp_MessageMSGCB()

```
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_4, 4, 50, (flashTime / 4) );
      break;
  }
}
```

SampleApp_SendPeriodicMessage()函数主要功能是实现周期性数据的发送，通过调用AF_DataRequest()函数进行数据的发送，如代码3-9所示。

【代码3-9】 SampleApp_SendPeriodicMessage()

```
void SampleApp_SendPeriodicMessage( void )
{
  if ( AF_DataRequest( &SampleApp_Periodic_DstAddr, &SampleApp_epDesc,
                       SAMPLEAPP_PERIODIC_CLUSTERID,
                       1,
                       (uint8 * )&SampleAppPeriodicCounter,
```

```
                        &SampleApp_TransID,
                        AF_DISCV_ROUTE,
                        AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
    {
    }
    else
    {
    }
}
```

SampleApp_SendFlashMessage()函数调用 AF_DataRequest()函数发送 LED 闪烁时间事件命令,如代码 3-10 所示。

【代码 3-10】 SampleApp_SendFlashMessage()

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

  if ( AF_DataRequest( &SampleApp_Flash_DstAddr, &SampleApp_epDesc,
                       SAMPLEAPP_FLASH_CLUSTERID,
                       3,
                       buffer,
                       &SampleApp_TransID,
                       AF_DISCV_ROUTE,
                       AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
  else
  {
  }
}
```

3. SampleApp.h

在 SampleApp.h 文件中定义了端点描述符、端点简单描述符、应用层用户自定义事件、定时事件的定时时间、组寻址 ID,如代码 3-11 所示。

【代码 3-11】 SampleApp.h

```
#define SAMPLEAPP_ENDPOINT          20

#define SAMPLEAPP_PROFID            0x0F08
#define SAMPLEAPP_DEVICEID          0x0001
#define SAMPLEAPP_DEVICE_VERSION    0
```

```
#define SAMPLEAPP_FLAGS                         0

#define SAMPLEAPP_MAX_CLUSTERS                  2
#define SAMPLEAPP_PERIODIC_CLUSTERID            1
#define SAMPLEAPP_FLASH_CLUSTERID               2

#define SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT     5000

#define SAMPLEAPP_SEND_PERIODIC_MSG_EVT         0x0001

#define SAMPLEAPP_FLASH_GROUP                   0x0001

#define SAMPLEAPP_FLASH_DURATION                1000
extern void SampleApp_Init( uint8 task_id );
extern UINT16 SampleApp_ProcessEvent( uint8 task_id, uint16 events );
```

3.4.3 ZDO层解析

ZDO层主要负责网络设备的建立和启动，在Z-Stack协议中ZDO层目录下包括ZDApp.c文件、ZDApp.h文件、ZDConfig.c文件、ZDConfig.h文件等。ZDO层目录如图3-10所示。

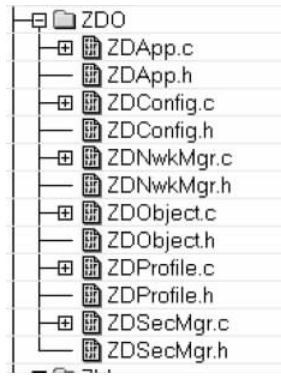


图3-10 ZDO层目录

其中网络的建立和设备的启动通过ZDApp_Init()函数实现。ZDApp_Init()函数在ZDApp.c文件中定义，主要实现以下功能：

- 初始化ZDO网络设备短地址；
- 获得64位IEEE地址信息；
- ZDO层初始化；
- 网络设备启动。

ZDApp_Init()函数具体实现如代码3-12所示。

【代码 3-12】 ZDApp_Init()

```
void ZDApp_Init( uint8 task_id )
{
  ZDAppTaskID = task_id;

  ZDAppNwkAddr.addrMode = Addr16Bit;
  ZDAppNwkAddr.addr.shortAddr = INVALID_NODE_ADDR;
  (void)NLME_GetExtAddr();
  ZDAppCheckForHoldKey();

  ZDO_Init();
  afRegister( (endPointDesc_t *)&ZDApp_epDesc );

#if defined( ZDO_USERDESC_RESPONSE )
  ZDApp_InitUserDesc();
#endif

  if ( devState != DEV_HOLD )
  {
    ZDOInitDevice( 0 );
  }
  else
  {
    HalLedBlink ( HAL_LED_4, 0, 50, 500 );
  }

  ZDApp_RegisterCBs();
}
```

3.4.4 NWK 层解析

网络层主要负责的功能包括网络拓扑结构、网络参数的设定及节点地址分配，NWK 层目录具体文件如图 3-11 所示。

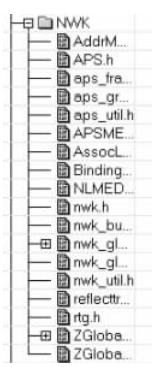


图 3-11 NWK 层目录

1. 网络拓扑结构

ZigBee网络拓扑结构有3种形式：星型、树型和网状网络。在Z-Stack协议栈中定义了这3种类型的网络，具体定义在nwk_globals.h文件中，如代码3-13所示。

【代码3-13】 nwk_globals.h

```
/* 网络拓扑结构的定义 */
/* 星型网络 */
#define NWK_MODE_STAR    0
/* 树型网络 */
#define NWK_MODE_TREE    1
/* 网状网络 */
#define NWK_MODE_MESH    2
```

2. 网络参数设置

在nwk_globals.h文件中定义了协议栈模式参数，当参数设置不同时，网络深度和网络类型不同；另外还定义了信道的频段。具体定义如代码3-14所示。

【代码3-14】 nwk_globals.h

```
/* 协议栈模式参数 */
#define NETWORK_SPECIFIC      0
#define HOME_CONTROLS         1
#define ZIGBEEPRO_PROFILE     2
#define GENERIC_STAR          3
#define GENERIC_TREE          4
/* 信道设置 */
#define MAX_CHANNELS_868MHZ   0x00000001
#define MAX_CHANNELS_915MHZ   0x000007FE
#define MAX_CHANNELS_24GHZ    0x07FFF800
/* 如果定义 ZIGBEEPRO,那么协议栈模式参数为 ZIGBEEPRO_PROFILE,否则为 HOME_CONTROLS */
#if defined ( ZIGBEEPRO )
    #define STACK_PROFILE_ID  ZIGBEEPRO_PROFILE
#else
    #define STACK_PROFILE_ID  HOME_CONTROLS
#endif
    /* 如果 STACK_PROFILE_ID 为 ZIGBEEPRO_PROFILE,网络深度为 20,网络模式为网状网络,安全
       模式为 SECURITY_COMMERCIAL */
#if ( STACK_PROFILE_ID == ZIGBEEPRO_PROFILE )
    #define MAX_NODE_DEPTH    20
    #define NWK_MODE          NWK_MODE_MESH
    #define SECURITY_MODE     SECURITY_COMMERCIAL
    #if ( SECURE != 0 )
    #define USE_NWK_SECURITY  1     // true or false
    #define SECURITY_LEVEL    5
  #else
```

```
    #define USE_NWK_SECURITY                0     // true or false
    #define SECURITY_LEVEL                  0
  #endif
    /* 如果 STACK_PROFILE_ID 为 HOME_CONTROLS,网络深度为 5,网络模式为网状网络,
    安全模式为 SECURITY_COMMERCIAL */
#elif ( STACK_PROFILE_ID == HOME_CONTROLS )
    #define MAX_NODE_DEPTH                  5
    #define NWK_MODE                        NWK_MODE_MESH
    #define SECURITY_MODE                   SECURITY_COMMERCIAL
  #if ( SECURE != 0 )
    #define USE_NWK_SECURITY                1
    #define SECURITY_LEVEL                  5
  #else
    #define USE_NWK_SECURITY                0
    #define SECURITY_LEVEL                  0
  #endif
    /* 如果 STACK_PROFILE_ID 为 GENERIC_STAR,网络深度为 5,网络模式为星型网络,
    安全模式为 SECURITY_COMMERCIAL */
#elif ( STACK_PROFILE_ID == GENERIC_STAR )
    #define MAX_NODE_DEPTH                  5
    #define NWK_MODE                        NWK_MODE_STAR
    #define SECURITY_MODE                   SECURITY_RESIDENTIAL
  #if ( SECURE != 0 )
    #define USE_NWK_SECURITY                1
    #define SECURITY_LEVEL                  5
  #else
    #define USE_NWK_SECURITY                0
    #define SECURITY_LEVEL                  0
#endif
    /* 如果 STACK_PROFILE_ID 为 NETWORK_SPECIFIC,网络深度为 5,网络模式为网状网络,
    安全模式为 SECURITY_RESIDENTIAL */
#elif ( STACK_PROFILE_ID == NETWORK_SPECIFIC )
    #define MAX_NODE_DEPTH                  5
    #define NWK_MODE                        NWK_MODE_MESH
    #define SECURITY_MODE                   SECURITY_RESIDENTIAL
    #if ( SECURE != 0 )
    #define USE_NWK_SECURITY                1
    #define SECURITY_LEVEL                  5
#else
    #define USE_NWK_SECURITY                0
    #define SECURITY_LEVEL                  0
#endif
#endif
```

3. 节点地址的分配

在 nwk_globals.c 文件中定义了两个数组：CskipChldrn[]和 CskipRtrs[]。分别表示不同含义：

- CskipChldrn[]数组中的每一个元素分别表示一个协调器或一个路由器可以连接的

子节点的最大个数；

- CskipRtrs[]数组中的每一个元素分别表示一个协调器或一个路由器可以连接具有路由功能的节点的最大个数。

【示例 3-1】 节点个数配置

```
CskipChldrn[MAX_NODE_DEPTH + 1] = {5,5,5,5,5,0};
CskipRtrs[MAX_NODE_DEPTH + 1] = {5,5,5,5,5,0};
```

其中 MAX_NODE_DEPTH 为路由深度，根据 CskipChldrn[]元素的个数，可以计算出 MAX_NODE_DEPTH 为 5。因此该示例中的 CskipChldrn 路由深度为 5 级，其中第 0 个元素的取值为 5，表示路由深度为 0 时，即协调器下面连接的子节点的最大子节点个数为 5。以此类推，第 5 级路由节点下面连接的子节点个数为 0，即不能连接子节点。

此示例中的 CskipRtrs[]中第 0 个元素的取值为 5，表示路由深度为 0 时，即协调器下面连接的子节点的最大路由器个数为 5。以此类推，第 5 级路由节点下面连接的路由器个数为 0，即不能连接路由器。

CskipChldrn[]和 CskipRtrs[]两个参数的设置会影响 16 位网络地址的分配，尤其是在树型网络中。16 位网络地址的分配由协调器来完成，在网状网络中，网路地址分配由协调器随机的分配；在树型网络中，网络地址的分配遵循了一定的算法。

在网状网络中，网络地址的配置是自动分配的，只需要将 CskipChldrn[]和 CskipRtrs[]设置为默认值即可。具体实现如代码 3-15 所示。

【代码 3-15】 **nwk_globals. c**

```
#if ( STACK_PROFILE_ID == ZIGBEEPRO_PROFILE )
  uint8 CskipRtrs[1] = {0};
  uint8 CskipChldrn[1] = {0};
#elif ( STACK_PROFILE_ID == HOME_CONTROLS )
  uint8 CskipRtrs[MAX_NODE_DEPTH + 1] = {6,6,6,6,6,0};
  uint8 CskipChldrn[MAX_NODE_DEPTH + 1] = {20,20,20,20,20,0};
#elif ( STACK_PROFILE_ID == GENERIC_STAR )
  uint8 CskipRtrs[MAX_NODE_DEPTH + 1] = {5,5,5,5,5,0};
  uint8 CskipChldrn[MAX_NODE_DEPTH + 1] = {5,5,5,5,5,0};
#elif ( STACK_PROFILE_ID == NETWORK_SPECIFIC )
  uint8 CskipRtrs[MAX_NODE_DEPTH + 1] = {5,5,5,5,5,0};
  uint8 CskipChldrn[MAX_NODE_DEPTH + 1] = {5,5,5,5,5,0};
```

树型网络的网络地址通过以下算法进行设定，首先定义路由深度为 d，最大路由深度为 L（最大路由深度在 Z-Stack 协议栈中使用 MAX_NODE_DEPTH 表示），其次 CsKipChldrn[]中元素的个数定义为 C，CskipRtrs[]中元素的个数定义为 R。那么深度为 d 的网络地址偏移量 Cskip 的计算公式如式(3-1)所示。

$$\mathrm{Cskip}(d)=\begin{cases}\dfrac{1+C-R-C\times R^{L-d-1}}{1-R} & R\neq 1\\ 1+C\times(L-d-1) & R=1\end{cases} \tag{3-1}$$

如果 $L=6$，$C=20$，$R=6$，按照式(3-1)计算得出 $d=1$ 的网络地址偏移量 Cskip(1)为0x143D。由于协调器网络地址为0x0000，协调器下第一个路由器的网络地址为0x0001，第二个路由器的网络地址为0x0001+0x143D，即0x143E。

3.4.5 HAL层解析

HAL文件中包含了Common、Include和Target文件，在这3个文件中为硬件资源提供了相应的驱动，如图3-12所示。

图3-12 HAL层目录

其中：

- Common→hal_drivers.c文件中主要实现了硬件初始化函数Hal_Init()、硬件抽象层驱动初始化HalDriverInit ()函数和硬件抽象层事件处理函数Hal_ProcessEvent()；
- 在Include文件中主要包含了硬件资源的定义与函数声明；
- Target→hal_board_cfg.h文件中为LED等硬件资源进行配置；
- Target→Drivers文件中包含了常用的硬件资源的实现函数。

以下内容阐述了各个函数的具体实现。

Hal_Init()函数主要为硬件抽象层注册任务ID，如代码3-16所示。

【代码3-16】 Hal_Init()

```
void Hal_Init( uint8 task_id )
{
    /* 注册任务 ID */
    Hal_TaskID = task_id;
}
```

HalDriverInit()函数主要实现硬件资源的初始化，包括定时器的初始化、ADC初始化、DMA初始化、Flash初始化、AES初始化、LCD初始化、LED初始化、UART初始化、按键初始化和SPI初始化，如代码3-17所示。

【代码3-17】 HalDriverInit()

```
void HalDriverInit (void)
{
    /* 定时器初始化 */
#if (defined HAL_TIMER) && (HAL_TIMER == TRUE)
    HalTimerInit();
#endif
    /* ADC 初始化 */
#if (defined HAL_ADC) && (HAL_ADC == TRUE)
    HalAdcInit();
```

```
#endif
    /* DMA 初始化 */
#if (defined HAL_DMA) && (HAL_DMA == TRUE)
    HalDmaInit();
#endif
    /* Flash 初始化 */
#if (defined HAL_FLASH) && (HAL_FLASH == TRUE)
    HalFlashInit();
#endif
    /* AES 初始化 */
#if (defined HAL_AES) && (HAL_AES == TRUE)
    HalAesInit();
#endif
    /* LCD 初始化 */
#if (defined HAL_LCD) && (HAL_LCD == TRUE)
    HalLcdInit();
#endif
    /* LED 初始化 */
#if (defined HAL_LED) && (HAL_LED == TRUE)
    HalLedInit();
#endif
    /* UART 初始化 */
#if (defined HAL_UART) && (HAL_UART == TRUE)
    HalUARTInit();
#endif
    /* KEY 按键初始化 */
#if (defined HAL_KEY) && (HAL_KEY == TRUE)
    HalKeyInit();
#endif
    /* SPI 初始化 */
#if (defined HAL_SPI) && (HAL_SPI == TRUE)
    HalSpiInit();
#endif
}
```

Hal_ProcessEvent()函数由 APP 层 OSAL_SampleApp.c 文件调用，主要实现硬件抽象层的各种事件处理，比如系统消息事件、LED 闪烁事件、按键事件和睡眠模式事件，如代码 3-18 所示。

【代码 3-18】 Hal_ProcessEvent()

```
uint16 Hal_ProcessEvent( uint8 task_id, uint16 events )
{
    uint8 *msgPtr;
    (void)task_id; // Intentionally unreferenced parameter
    /* 系统消息事件 */
    if ( events & SYS_EVENT_MSG )
    {
```

```
            msgPtr = osal_msg_receive(Hal_TaskID);
            while (msgPtr)
            {
                osal_msg_deallocate( msgPtr );
                msgPtr = osal_msg_receive( Hal_TaskID );
            }
            return events ^ SYS_EVENT_MSG;
        }
        /* LED 闪烁事件 */
        if ( events & HAL_LED_BLINK_EVENT )
        {
    #if (defined (BLINK_LEDS)) && (HAL_LED == TRUE)
            HalLedUpdate();
    #endif /* BLINK_LEDS && HAL_LED */
            return events ^ HAL_LED_BLINK_EVENT;
        }
        /* 按键事件 */
        if (events & HAL_KEY_EVENT)
        {
    #if (defined HAL_KEY) && (HAL_KEY == TRUE)
            /* Check for keys */
            HalKeyPoll();
            if (! Hal_KeyIntEnable)
            {
                osal_start_timerEx( Hal_TaskID, HAL_KEY_EVENT, 100);
            }
    #endif
            return events ^ HAL_KEY_EVENT;
        }
        /* 睡眠模式 */
    #ifdef POWER_SAVING
        if ( events & HAL_SLEEP_TIMER_EVENT )
        {
            halRestoreSleepLevel();
            return events ^ HAL_SLEEP_TIMER_EVENT;
        }
    #endif
    return 0;
    }
```

Target→hal_board_cfg.h 文件中为硬件资源 LED 等进行配置，在官方的协议栈中定义了 3 个 LED，分别接 CC2530 的 P1_0、P1_1 和 P1_4 引脚，如代码 3-19 所示。

【代码 3-19】 hal_board_cfg.h

```
    /* LED1 配置 */
    #define LED1_BV       BV(0)
    #define LED1_SBIT     P1_0
    #define LED1_DDR      P1DIR
```

```
#define LED1_POLARITY              ACTIVE_HIGH

#ifdef HAL_BOARD_CC2530EB_REV17
  /* LED2 配置 */
  #define LED2_BV                  BV(1)
  #define LED2_SBIT                P1_1
  #define LED2_DDR                 P1DIR
  #define LED2_POLARITY            ACTIVE_HIGH

  /* LED3 配置 */
  #define LED3_BV                  BV(4)
  #define LED3_SBIT                P1_4
  #define LED3_DDR                 P1DIR
  #define LED3_POLARITY            ACTIVE_HIGH
#endif
```

对 LED 进行引脚配置完成之后，可以控制 LED 开关状态，如代码 3-20 所示。

【代码 3-20】 hal_board_cfg.h

```
#if defined (HAL_BOARD_CC2530EB_REV17) && !defined (HAL_PA_LNA) && !defined (HAL_PA_LNA_
CC2590)
    /* 打开 LED */
  #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() HAL_TURN_OFF_LED1()
    /* 关闭 LED */
  #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()  HAL_TURN_ON_LED1()
    /* LED 状态改变 */
  #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() HAL_TOGGLE_LED1()

  #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()    HAL_STATE_LED1()
#endif
```

在 Drives 文件中定义了硬件资源的驱动函数文件，即 API 函数，包括 LED、ADC、KEY、LCD、定时器、串口、DMA 以及 Flash 等。

- LED：在 hal_led.c 文件中实现，为 LED 提供驱动函数；
- ADC：在 hal_adc.c 文件中实现，为 ADC 提供驱动函数；
- KEY：在 hal_key.c 文件中实现，为按键提供驱动函数；

- LCD：在 hal_LCD.c 文件中实现，为 LCD 提供驱动函数；
- 定时器：在 hal_timer.c 文件中实现，为定时器提供驱动函数；
- 串口：在 hal_uart.c 文件中实现，为串口提供驱动函数；
- DMA：在 hal_dma.c 文件中实现，为 DMA 提供驱动函数；
- Flash：在 hal_flash.c 文件中实现，为 Flash 提供驱动函数。

以 LED 为例，在 hal_led.c 中提供了两个 API 函数，在 APP 应用层可以直接调用此函数来控制 LED。

【示例 3-2】 LED 函数

```
HalLedSet(uint8 leds,uint8 mode);
HalLedBlink(uint8 leds,uint8 numBlink,uint8 percend,uint16 period);
```

其中 HalLedSet()函数主要功能是控制 LED 的亮灭状态。

【函数 3-46】 HalLedSet()

```
voidHalLedSet(uint8 leds,uint8 mode);
```

参数描述：

- leds——LED 的名称，在协议栈中如果定义了 4 个 LED，那么 leds 的参数取值有 4 种，分别是：HAL_LED_1；HAL_LED_2；HAL_LED_3；HAL_LED_4。
- mode——LED 的状态，可以有 3 种形式，分别是：HAL_LED_MODE_ON——打开 LED；HAL_LED_MODE_OFF——关闭 LED；HAL_LED_MODE_TOGGLE——LED 状态改变。

返回值：无。

HalLedBlink()函数主要功能是实现 LED 的闪烁。

【函数 3-47】 HalLedBlink()

```
voidHalLedBlink(uint8 leds,uint8 numBlink,uint8 percend,uint16 period);
```

参数描述：

- leds——LED 的名称，即 HAL_LED_1、HAL_LED_2、HAL_LED_3、HAL_LED_4。
- numBlink——LED 闪烁的次数；
- percend——LED 打开状态与关闭状态所占用时间的占空比，比如打开和关闭状态所有的时间比例为 1∶1，那么占空比的计算方式为 100/2，即占空比的取值为 50；
- period——指 LED 闪烁一个周期所需要的时间，以 ms 为单位。

3.5 贯穿项目实现

下面介绍实现贯穿项目 3-1 Z-Stack 协议栈下载、调试及组网。使用 IAR 打开 SampleApp 工程，打开目录为 Texas Instruments\ZStack-CC2530-2.2.2-1.3.0\Projects\zstack\Samples\SampleApp\CC2530DB，在 CC2530DB 文件夹中有一个 SampleApp.eww

的文件，使用 IAR 打开此文件，然后连接硬件设备。在 WorkSpace 选项选择 CoodinatorEB 或者 CoodinatorEB-Pro，本任务中采用 CoodinatorEB 选项，如图 3-13 所示。

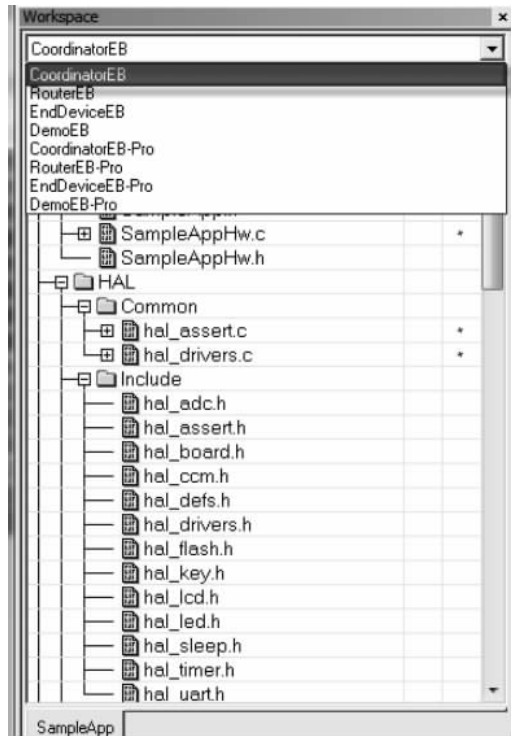


图 3-13 选择协调器设备

单击工具栏中的按钮，将程序下载至设备中。下载完成之后如图 3-14 所示。程序下载成功之后，自动会跳入程序的入口函数 main()。

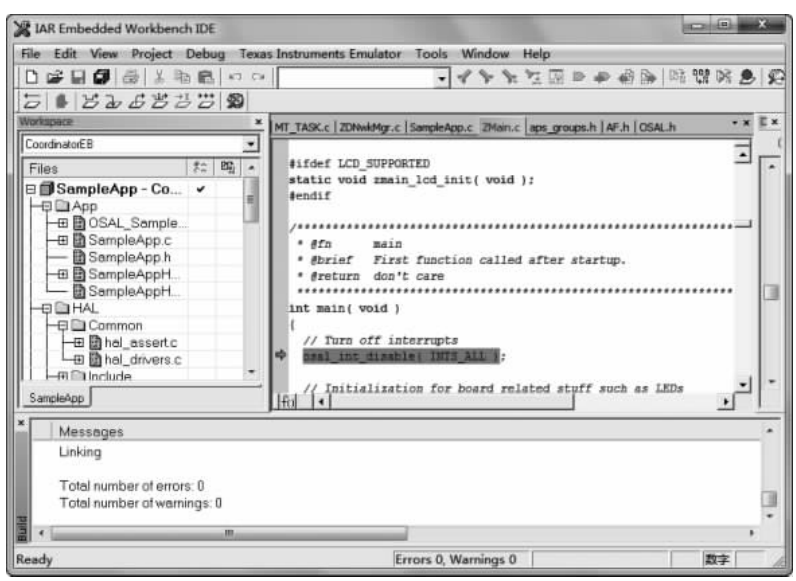


图 3-14 程序下载成功

程序下载完成之后，进行断点调试，在 APP → SampleApp. c → SampleApp _ ProcessEvent()函数中的网络状态改变事件处断点调试，单击 按钮运行程序，程序运行至断点处停止，如图 3-15 所示。此时继续运行程序，将执行 LED1～LED4 同时闪烁 4 次。

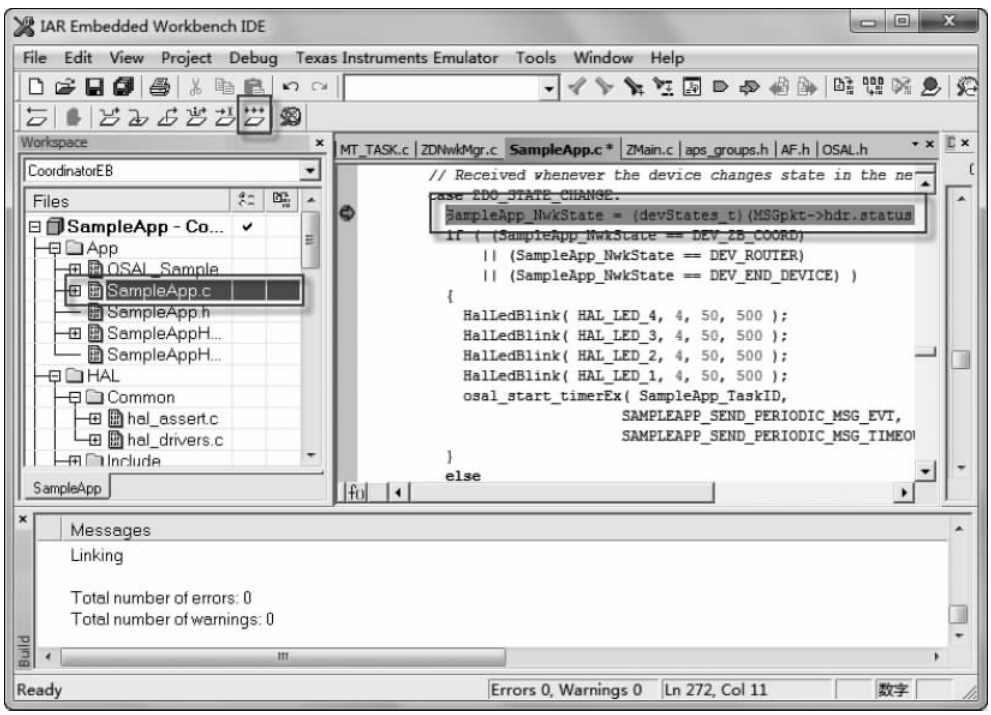


图 3-15　断点调试

本章总结

小结

- Z-Stack 协议栈是 TI 公司开发的完全符合 ZigBee 标准的解决方案，可以实现 ZigBee 的各种应用。
- Z-Stack 协议栈可以从 TI 的官方网站下载。
- API 函数的主要功能是 Z-Stack 协议栈内部的服务提供接口。Z-Stack 提供的 API 函数包括 ZDO 设备对象、应用框架 AF、应用支持子层 APS 和网络层。
- ZDO 设备对象 API 包括 ZDO 网络设备启动 API、ZDO 信息回调函数 API、ZDO 发现 API 和 ZDO 绑定 API。
- 应用框架 AF 层 API 包括端点管理 API 和发送数据 API。
- 应用支持子层 API 包括组表管理 API。
- 网络层 API 包括网络管理 API 和地址管理 API。

Q&A

问题：对于 Z-Stack 协议栈，用户开发在哪一层？

回答：对于 Z-Stack 协议栈，用户应用程序开发在应用层，应用层包括用户任务初始化和事件处理函数的调用。在用户任务初始化函数中对用户任务 ID、网络状态信息、发送地址信息、端点信息等进行初始化；在事件处理函数中对接收数据事件、按键事件和定时事件进行处理。

本章练习

习题

1. Z-Stack 提供的 API 函数包括_____、_____、_____和_____。
2. ZDO 设备对象 API 包括_____、_____、_____和_____。
3. 下列 API 函数中是端点管理 API 函数的是_____。

   A. afRegister()

   B. NLME_NetworkFormationRequest()

   C. ZDP_NwkAddrReq()

   D. ZDP_NodeDescReq()
4. 编写 OSAL_SampleApp.c 文件中任务初始化函数。