2010

# ZigBee 2007-Pro 无线系统

# TI-CC2530

## 无线透明传输

锋硕电子科技有限公司
www.fuccesso.com.cn
2010-9-27

# 目录

第一章 功能描述 ..... 3
第二章 工程整体架构和选项设置 ..... 6
2.1 工程架构 ..... 6
2.2 工程选项设置 ..... 9
第三章 App 初始化和任务事件处理 ..... 12
3.1 App 初始化 ..... 12
3.2 App 任务事件处理函数 ..... 13
第四章 ZDO 初始化和任务事件处理 ..... 14
4.1 ZDO 初始化 ..... 14
4.2 ZDO 任务事件处理函数 ..... 14
第五章 协调器建立网络流程分析 ..... 16
5.1 协调器设备类型和初始状态 ..... 16
5.2 协调器建立网络流程 ..... 16
第六章 路由器加入网络流程分析 ..... 20
6.1 路由器设备类型和初始状态 ..... 20
6.2 路由器加入网络流程 ..... 20
第七章 终端设备加入网络流程分析 ..... 24
7.1 终端设备设备类型和初始状态 ..... 24
7.2 终端设备加入网络流程 ..... 24
第八章 绑定过程分析 ..... 28
8.1 终端设备绑定 ..... 28
8.2 匹配描述符绑定 ..... 29
第九章 发送数据 ..... 32
第十章 接收数据 ..... 33

# 第一章 功能描述

本工程目录为:

ZigBee2007FSCode\ZStack-CC2530-2.3.1-1.4.0\Projects\zstack\Samples\SerialP2P

在无线透明传输应用中，上位机(电脑或微控制器)与模块进行信息交换的数据格式，没有像指令字头、结束符等数据包信息，只有上位机串口有数据输出，模块就把串口的数据以无线方式编码发送。当接收模块接收到发射模块发送的无线数据信号后进行解码，把解码后的数据按发送端的格式从串口输出。也就是说模块对使用者是开发的、透明的。

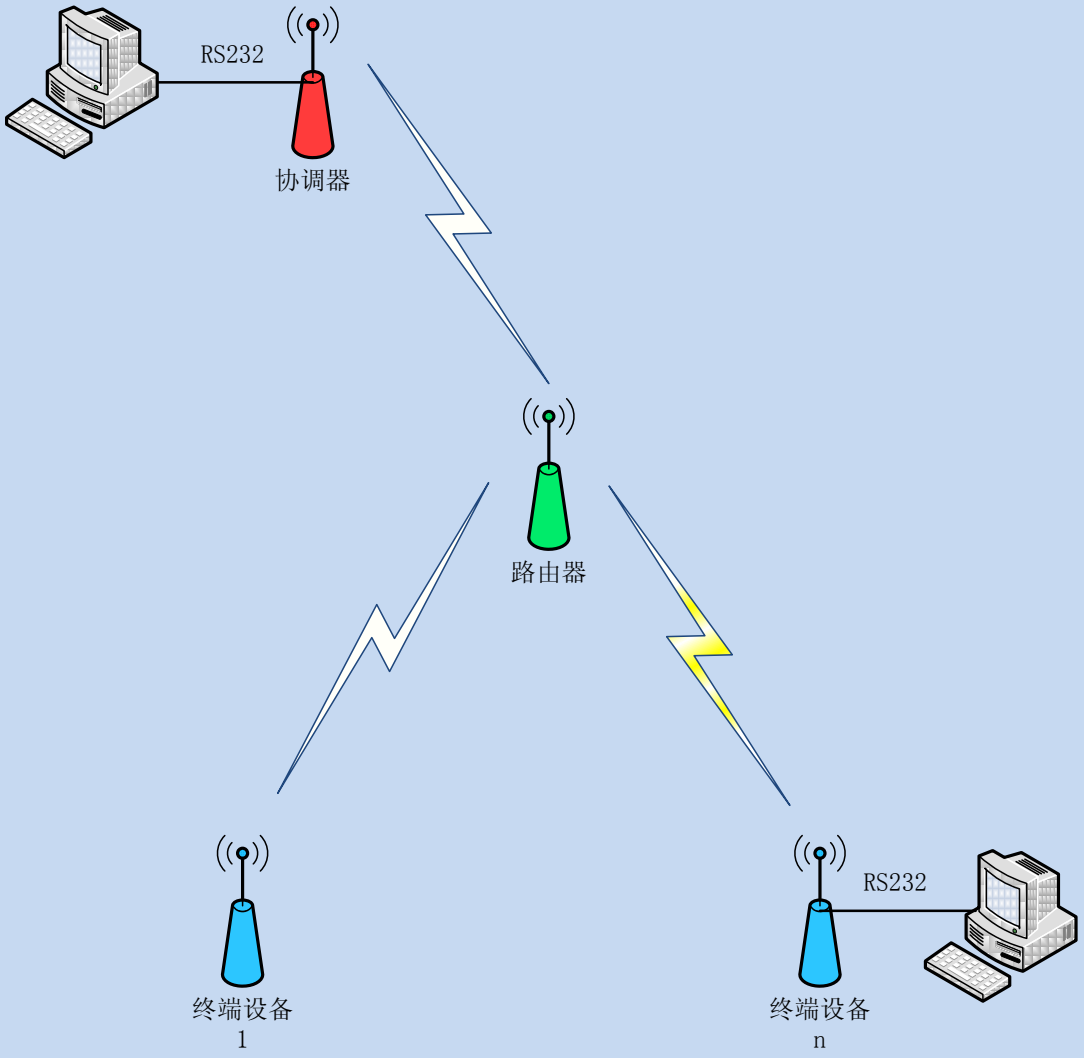


图 1.1 树型网络拓扑结构

其中，终端设备、路由器和协调器分别采用锋硕电子开发的协调器节点 CC2530+GPRS、

路由节点 CC2530+CC2591 和协调器节点 CC2530+GPRS：


（路由节点 CC2530+CC2591）

（协调器节点 CC2530+GPRS）

终端设备采用 2 种绑定方法：1）终端设备绑定；2）匹配描述符绑定。本例中，终端设备使用 CC2530 开发板，用于发送数据；路由器使用 CC2530+CC2591 大功率开发板，起到中继路由的作用；协调器使用 CC2530+GPRS 开发板，用于接收数据。

为实现上述目的，整个应用程序应该具备以下功能：

1) 协调器自动建立网络；

2) 路由器自动加入网络，并自动启动路由功能；

3) 终端设备自动加入网络；

4) 手动发起终端设备绑定；

5) 手动发起匹配描述符绑定

6) 建立绑定后，绑定设备之间可以相互通信。

# 第二章 工程整体架构和选项设置

## 2.1 工程架构

用户打开 SerialApp 工程后，会在 Workspace 区域看到不同的设备类型，不同的设备类型下均包含 App 文件夹，里面存放着各种应用实现的源文件。其中，OSAL_SerialApp.c 实现任务初始化和任务处理函数列表；SerialApp.c 包含应用层的任务初始化函数和应用层任务事件处理函数。

Z-Stack 的目录结构如图 2.1 所示：

1) App：应用层，存放应用程序。
2) HAL：硬件层，与硬件电路相关。
3) MAC：数据链路层。
4) MT：监控调试层，通过串口调试各层，与各层进行直接交互。
5) NWK：网络层。
6) OSAL：操作系统层。
7) Profile：协议栈配置文件（AF）。
8) Security：安全层。
9) Services：地址处理层。
10) Tools：工程配置。
11) ZDO：设备对象，调用 APS 子层和 NWK 层服务。
12) Zmac：MAC 层接口函数。
13) ZMain：整个工程的入口。
14) Output：输出文件（由 IAR 自动生成）。

对于协调器，在 Workspace 区域的下拉菜单中选择 CoordinatorEB-Pro，鼠标点击上方的“make 按钮”后，所有文件对应的红色“*”将消失，此时配置文件 f8wCoord.cfg 将被使用，而 f8wEndev.cfg 和 f8wRouter.cfg 不会使用。如图 2.1 所示：

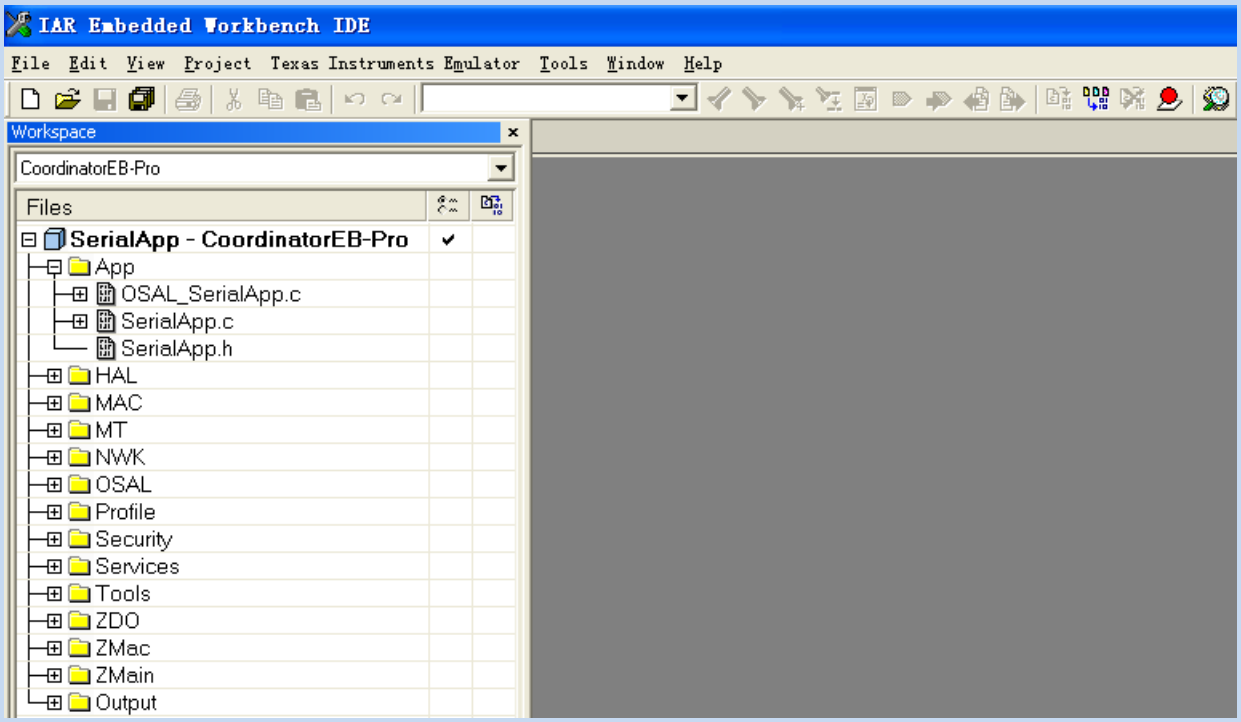


图 2.1 协调器工程架构

对于路由器，在 Workspace 区域的下拉菜单中选择 RouterEB-Pro，鼠标点击上方的“make 按钮”后，所有文件对应的红色“*”将消失，此时配置文件 f8wRouter.cfg 将被使用，而 f8wEndev.cfg 和 f8wCoord.cfg 不会使用。如图 2.2 所示：

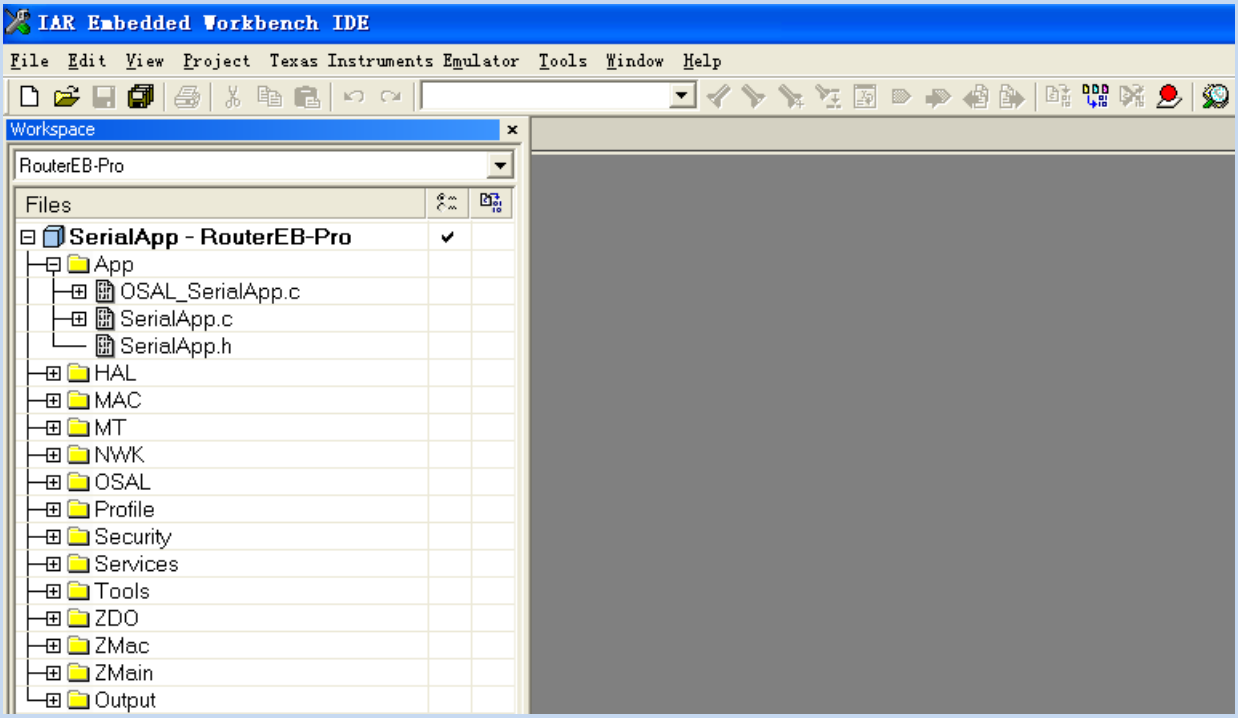


图 2.2 路由器工程架构

对于终端设备，在 Workspace 区域的下拉菜单中选择 EndDeviceEB-Pro，鼠标点击上方的“make 按钮”后，所有文件对应的红色“*”将消失，此时配置文件 f8wEndev.cfg 将被使用，而 f8wRouter.cfg 和 f8wCoord.cfg 不会使用。如下图所示：

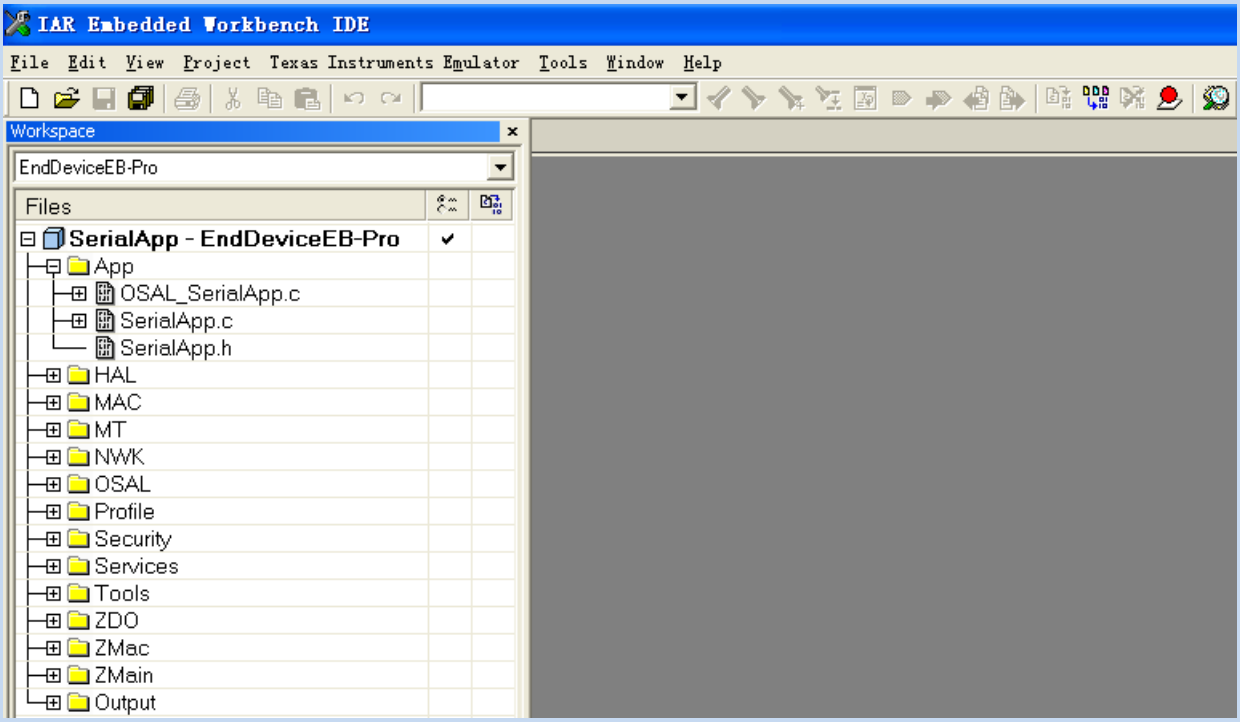


图 2.3 终端设备工程架构

## 2.2 工程选项设置

打开 SerialApp 工程后，欲进入到协调器的编译选项设置界面。选中工程名 SerialApp-CoordinatorEB-Pro，然后根据工程选项设置的路径：Project->Options->C/C++ Compiler->Preprocessor->Defined 。

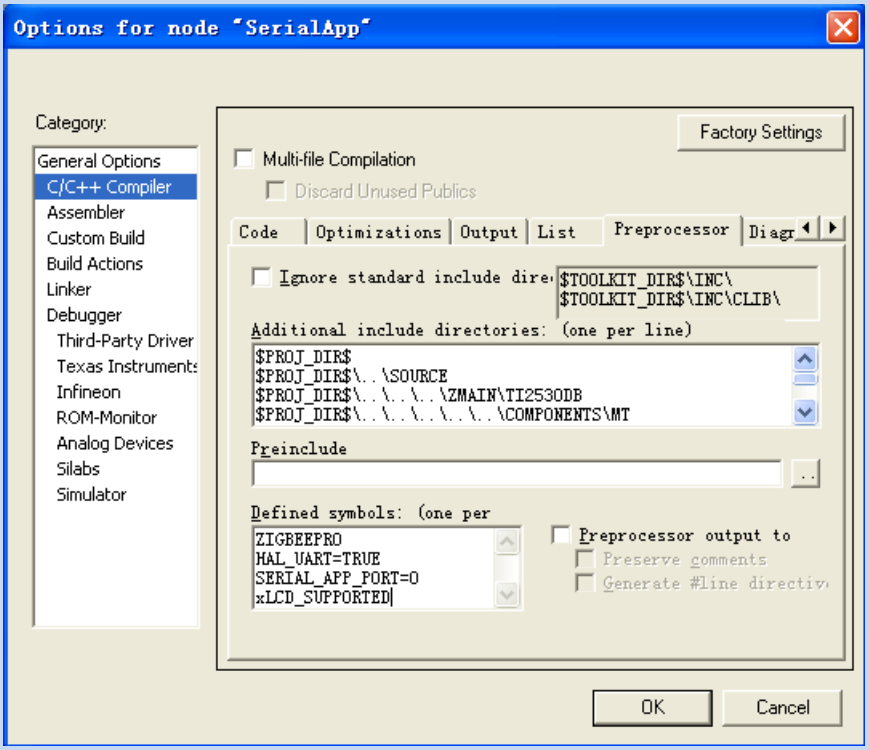


图 2.4 协调器 IAR 工程选项设置

要为工程选项添加一条编译选项，只需在 Defined symbols 框内添加一条新选项即可；要取消编译选项，只需在该编译选项的左侧添加“x”即可。

欲进入到协调器的编译选项设置界面。选中工程名 SerialApp-RouterEB-Pro，然后根据工程选项设置的路径：Project->Options->C/C++ Compiler->Preprocessor->Defined 。

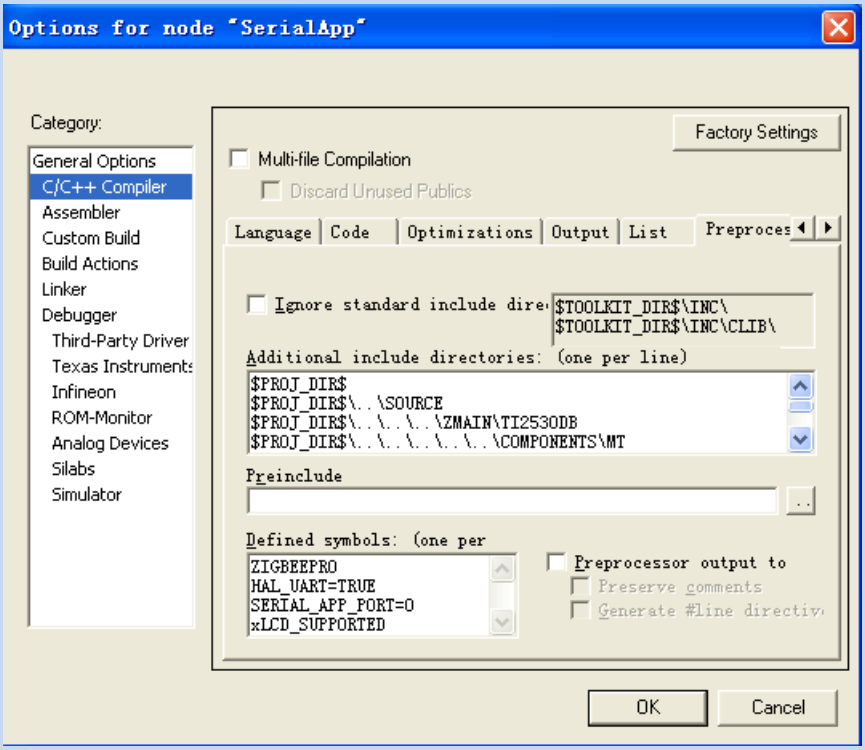


图 2.5 路由器 IAR 工程选项设置

欲进入到路由器的编译选项设置界面。选中工程名 SerialApp-EndDeviceEB-Pro，然后根据工程选项设置的路径：Project->Options->C/C++ Compiler->Preprocessor->Defined 。

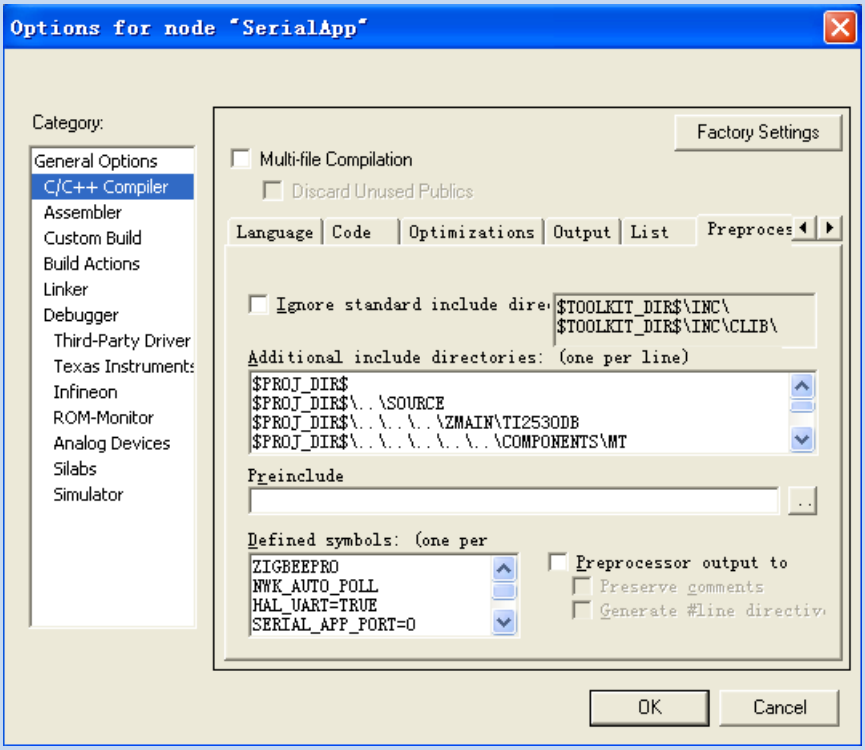


图 2.6 终端设备 IAR 工程选项设置

对于协调器、路由器和终端设备，分别打开 Tools->f8wCoord.cfg、 Tools->f8wRouter.cfg

和 Tools->f8wEndev.cfg 后，可以看到关于协调器、路由器和终端设备的配置信息。

综上，总结协调器、路由器和终端设备的工程选项设置如下表：

| 节点类型 | IAR 选项设置 | .cfg 配置文件 |
|---|---|---|
| 协调器 | ZIGBEEPRO<br>HAL_UART=TRUE<br>SERIAL_APP_PORT=0<br>xLCD_SUPPORTED | -DZDO_COORDINATOR<br>-DRTR_NWK |
| 路由器 | ZIGBEEPRO<br>HAL_UART=TRUE<br>SERIAL_APP_PORT=0<br>xLCD_SUPPORTED | -DRTR_NWK |
| 终端设备 | ZIGBEEPRO<br>NWK_AUTO_POLL<br>HAL_UART=TRUE<br>SERIAL_APP_PORT=0<br>xLCD_SUPPORTED<br>xPOWER_SAVING | 空 |

# 第三章 App 初始化和任务事件处理

## 3.1 App 初始化

SerialApp.c 中的 SerialApp_Init（）函数实现 App 初始化，主要完成以下几个方面的初始化工作：

1) 初始化任务 ID 号，其中 task_id 由操作系统初始化任务函数 osalInitTasks（）决定。

```
SerialApp_TaskID = task_id;
```

2) 接收序列。

```
SerialApp_RxSeq = 0xC3;
```

3) 初始化串口。设置波特率 38400，关闭流控制，接收最大字节 128，发送最大字节 128，使能串口中断，开启串口 0。

```
uartConfig.configured           = TRUE;
uartConfig.baudRate             = SERIAL_APP_BAUD;
uartConfig.flowControl          = FALSE;
uartConfig.flowControlThreshold = SERIAL_APP_THRESH;
uartConfig.rx.maxBufSize        = SERIAL_APP_RX_SZ;
uartConfig.tx.maxBufSize        = SERIAL_APP_TX_SZ;
uartConfig.idleTimeout          = SERIAL_APP_IDLE;
uartConfig.intEnable            = TRUE;
uartConfig.callBackFunc         = SerialApp_CallBack;
HalUARTOpen (SERIAL_APP_PORT, &uartConfig);
```

4) 在 AF 层注册该端点描述符。

```
afRegister( &SerialApp_epDesc );
```

5) 注册按键事件。

```
RegisterForKeys(SerialApp_TaskID );
```

6) 注册终端绑定响应消息和匹配描述符响应消息。

```
ZDO_RegisterForZDOMsg( SerialApp_TaskID, End_Device_Bind_rsp );
ZDO_RegisterForZDOMsg( SerialApp_TaskID, Match_Desc_rsp );
```

## 3.2 App 任务事件处理函数

应用层任务事件处理函数 SerialApp_ProcessEvent（）所有的事件，包含时间、消息和其他用户定义的事件。

| 事件 | | 处理函数 |
|---|---|---|
| 系统消息事件<br>SYS_EVENT_MSG | AF 信息输入<br>AF_INCOMING_MSG_CMD | SerialApp_MessageMSGCB（） |
| | 按键<br>KEY_CHANGE | SerialApp_HandleKeys（） |
| | ZDO 反馈消息<br>ZDO_CB_MSG | SerialApp_ProcessZDOMsgs（） |
| 发送数据事件<br>SERIALAPP_SEND_EVT | | SerialApp_Send() |
| 发送反馈信息事件<br>SERIALAPP_RESP_EVT | | SerialApp_Resp() |

# 第四章 ZDO 初始化和任务事件处理

## 4.1 ZDO 初始化

ZDApp.c 中的 ZDApp_Init（）函数实现 ZDO 初始化，主要完成以下几个方面的初始化工作：

1) 初始化任务 ID 号，其中 task_id 由操作系统初始化任务函数 osalInitTasks（）决定。

```
ZDAppTaskID = task_id;
```

2) 初始化网络地址，地址模式为 16 位，网络地址为无效。

```
ZDAppNwkAddr.addrMode = Addr16Bit;
ZDAppNwkAddr.addr.shortAddr = INVALID_NODE_ADDR;
```

3) 保存 64 位 IEEE 地址。

```
NLME_GetExtAddr();
```

4) 检测是否阻止自动启动。

```
ZDAppCheckForHoldKey();
```

5) 根据设备类型初始化网络服务

```
ZDO_Init();
```

6) 注册端点 0

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

7) ZDO 初始化设备

```
ZDOInitDevice();
```

8) 注册响应事件

```
ZDApp_RegisterCBs();
```

## 4.2 ZDO 任务事件处理函数

任务事件处理函数 ZDApp_event_loop（）包含消息、网络初始化、网络启动、路由启动等事件。

| 事件 | | 处理函数 |
|---|---|---|
| 系统消息事件<br>SYS_EVENT_MSG | AF 信息输入<br>AF_INCOMING_MSG_CMD | ZDP_IncomingData（） |
| | ZDO 反馈<br>AF_DATA_CONFIRM_CMD | ZDApp_ProcessMsgCBs（） |
| | AF 数据确认<br>AF_DATA_CONFIRM_CMD | 无 |
| | 网络发现确认<br>ZDO_NWK_DISC_CNF | NLME_JoinRequest（）或<br>NLME_ReJoinRequest（） |
| | 网络加入指示<br>ZDO_NWK_JOIN_IND | ZDApp_ProcessNetworkJoin（） |
| | 网络加入请求<br>ZDO_NWK_JOIN_REQ | ZDApp_NetworkInit() |
| 网络初始化<br>ZDO_NETWORK_INIT | | ZDO_StartDevice（） |
| 网络启动<br>ZDO_NETWORK_START | | ZDApp_NetworkStartEvt（） |
| 路由启动<br>ZDO_ROUTER_START | | osal_pwrmgr_device（） |
| 状态改变<br>ZDO_STATE_CHANGE_EVT | | ZDO_UpdateNwkStatus（） |
| 网络 NV 更新<br>ZDO_NWK_UPDATE_NV | | ZDApp_SaveNetworkStateEvt（） |
| 设备重新启动<br>ZDO_DEVICE_RESET | | SystemResetSoft（） |

# 第五章 协调器建立网络流程分析

## 5.1 协调器设备类型和初始状态

协调器的 IAR 工程配置选项中没有定义 BUILD_ALL_DEVICES，因此在 ZGlobals.h 文件中：

```
#define ZSTACK_DEVICE_BUILD                0x01
```

进一步有：

```
#define ZG_DEVICE_COORDINATOR_TYPE     1
```

从而：

```
#define DEVICE_LOGICAL_TYPE                0x00
```

由此，在 ZGlobals.c 文件中，可以得知设备逻辑类型为协调器：

```
zgDeviceLogicalType =0x00
```

协调器的 IAR 工程配置选项中没有定义阻止自动启动，因此在 ZDApp.c 文件中定义了设备初始状态和启动模式：

```
devState = DEV_INIT
devStartMode = MODE_HARD
```

## 5.2 协调器建立网络流程

详细的网络形成流程图如图 5.1 所示：

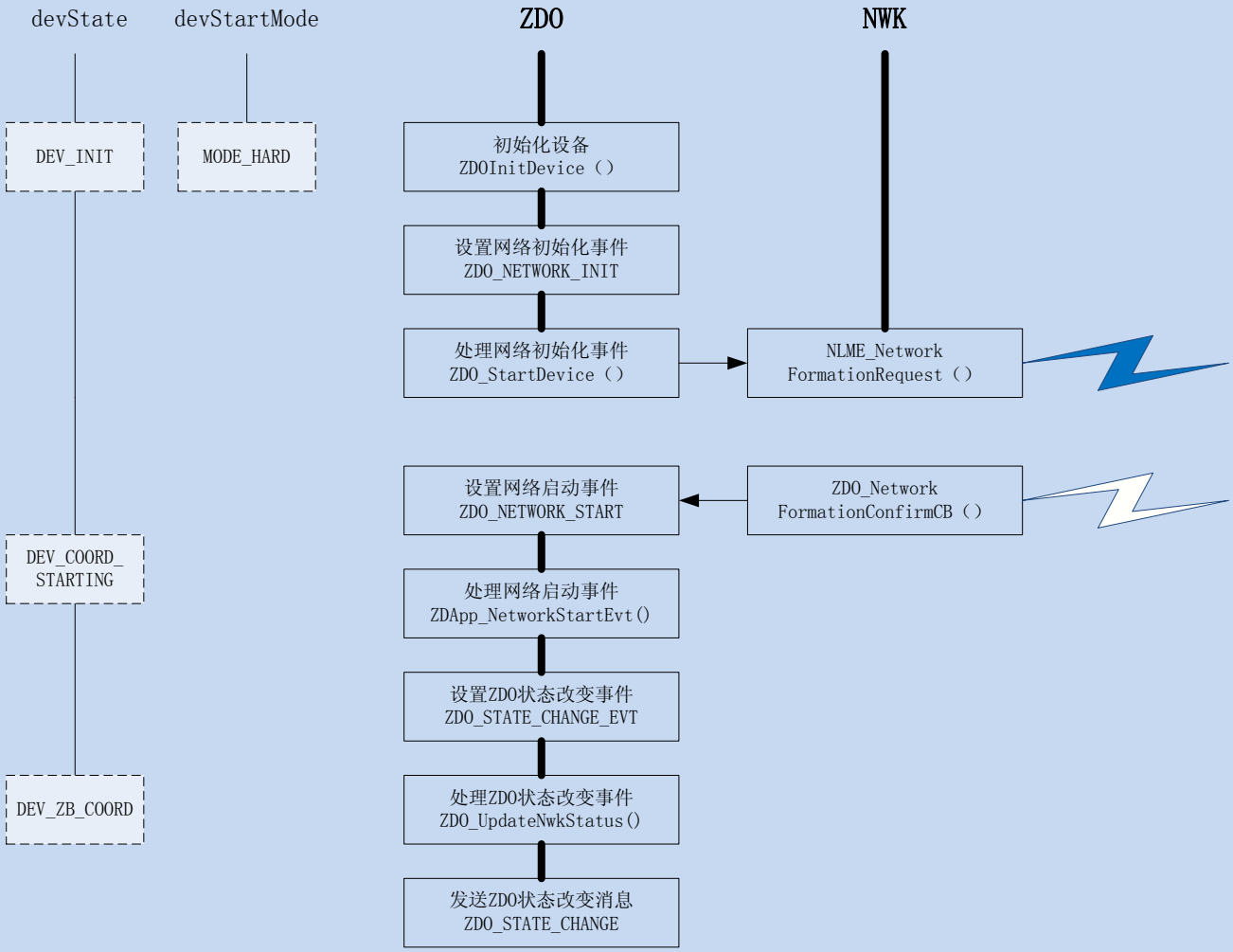


图 5.1 协调器网络形成流程分析

当协调器上电后，ZDO 层首先经历一系列的初始化工作，然后调用 ZDO 层的初始化设备函数：

```
ZDOInitDevice(0);
```

在该函数中设置了 NV 网络状态：

```
networkStateNV = ZDO_INITDEV_NEW_NETWORK_STATE;
```

最终触发网络初始化函数：

```
ZDApp_NetworkInit( extendedDelay );
```

设置网络初始化事件：

```
osal_set_event( ZDAppTaskID, ZDO_NETWORK_INIT );
```

ZDO 层的任务事件处理函数对网络初始化事件进行处理，即启动该设备：

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

此时将改变设备状态为协调器启动：

```
devState = DEV_COORD_STARTING;
```

并根据设备逻辑类型和启动模式调用 NWK 层网络形成请求函数：

```
NLME_NetworkFormationRequest( zgConfigPANID, zgApsUseExtendedPANID, zgDefaultChannelList, zgDefaultStartingScanDuration, beaconOrder, superframeOrder, false );
```

其中，个域网 ID 号和默认通道号在 f8wConfig.Cfg 中定义：

```
-DZDAPP_CONFIG_PAN_ID=0xFFFF
-DDEFAULT_CHANLIST=0x00000800  // 11 - 0x0B
```

外扩个域网 ID 号在 ZGlobals.c 中定义：

```
zgApsUseExtendedPANID[Z_EXTADDR_LEN] = {00,00,00,00,00,00,00,00};
```

当 NWK 层通过调用 MAC 和 PHY 层相关功能函数执行一些列网络形成动作后，NWK 层将接收到网络形成反馈，即：

```
ZDO_NetworkFormationConfirmCB（）
```

设置网络启动事件：

```
osal_set_event( ZDAppTaskID, ZDO_NETWORK_START );
```

ZDO 层任务事件处理函数将执行网络启动事件处理：

```
ZDApp_NetworkStartEvt();
```

此时将改变设备状态为协调器，并且保证电源供电：

```
devState = DEV_ZB_COORD;
osal_pwrmgr_device( PWRMGR_ALWAYS_ON );
```

而且设置 ZDO 状态改变事件：

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

ZDO 层任务事件处理函数将执行 ZDO 更新网络状态事件处理：

```
ZDO_UpdateNwkStatus( devState );
```

此时搜索端点列表，寻找曾经在应用层注册过的端点号，并且将 ZDO 状态改变消息发送给这些端点：

```
zdoSendStateChangeMsg(state, *(pItem->epDesc->task_id));
```

而且确定协调器（此时为协调器）的 16 位网络地址和 64 位 IEEE 地址：

```
NLME_GetShortAddr();
NLME_GetExtAddr();
```

当应用层接收到 ZDO 状态改变消息后，等待 5s 后启动周期性发送数据事件，时间间隔为 5s：

```
osal_start_timerEx( SerialApp_TaskID,SERIALAPP_SEND_MSG_EVT,SERIALAPP_SEND_MSG_TIMEOUT );
```

# 第六章 路由器加入网络流程分析

## 6.1 路由器设备类型和初始状态

路由器的 IAR 工程配置选项中没有定义 BUILD_ALL_DEVICES，因此在 ZGlobals.h 文件中：

```
#define ZSTACK_DEVICE_BUILD          0x02
```

进一步有：

```
#define ZG_BUILD_RTR_TYPE            0x02
```

从而：

```
#define DEVICE_LOGICAL_TYPE          0x01
```

由此，在 ZGlobals.c 文件中，可以得知路由器的设备逻辑类型为：

```
zgDeviceLogicalType =0x01
```

以及：

```
#define ZG_DEVICE_RTR_TYPE           1
#define ZG_DEVICE_JOINING_TYPE       1
```

路由器的 IAR 工程配置选项中没有定义阻止自定启动，因此在 ZDApp.c 文件中定义了设备初始状态和启动模式：

```
devState = DEV_INIT
devStartMode = MODE_JOIN
```

## 6.2 路由器加入网络流程

详细的加入网络流程图如图 6.1 所示：

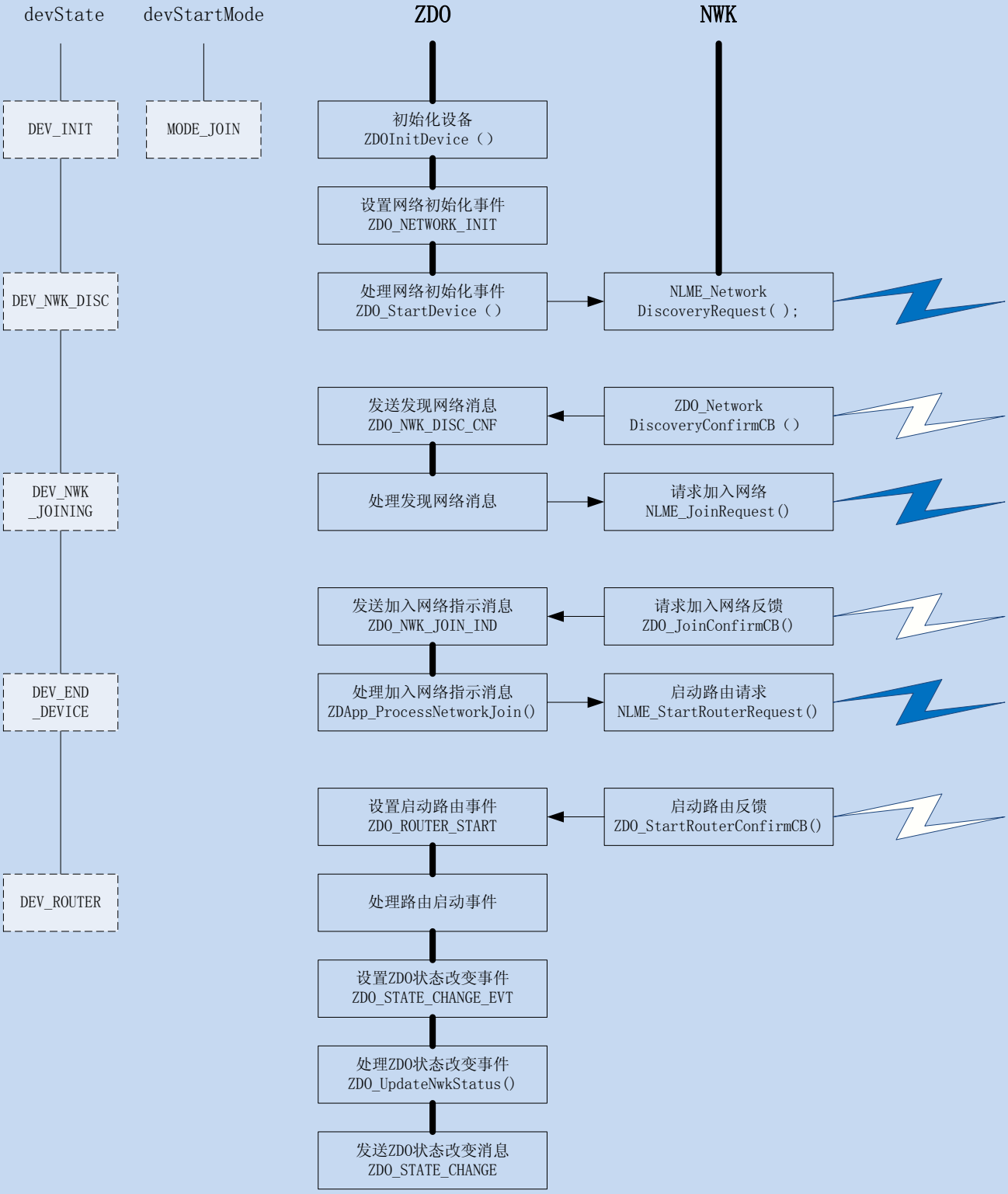


图 6.1 路由器加入网络流程分析

当路由器上电后，首先经历一系列的初始化工作，最终调用 ZDO 层的初始化设备函数：

```
ZDOInitDevice(zgStartDelay);
```

在该函数中设置了 NV 网络状态，并修改了当前设备状态：

```
networkStateNV = ZDO_INITDEV_NEW_NETWORK_STATE;
```

最终触发网络初始化函数：

```
ZDApp_NetworkInit( extendedDelay );
```

设置网络初始化事件：

```
osal_set_event( ZDAppTaskID, ZDO_NETWORK_INIT );
```

ZDO 层的任务事件处理函数对网络初始化事件进行处理，即启动该设备：

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

此时将改变设备状态为发现网络：

```
devState = DEV_NWK_DISC;
```

并根据设备逻辑类型和启动模式调用 NWK 层发现网络请求函数：

```
NLME_NetworkDiscoveryRequest( zgDefaultChannelList, zgDefaultStartingScanDuration );
```

其中，默认通道号在 f8wConfig.Cfg 中定义：

```
-DDEFAULT_CHANLIST=0x00000800   // 11 - 0x0B
```

当 NWK 层通过调用 MAC 和 PHY 层相关功能函数执行一些列发现网络动作后，NWK 层将接收到发现网络反馈，即：

```
ZDO_NetworkDiscoveryConfirmCB( uint8 ResultCount, networkDesc_t *NetworkList )
```

发送发现网络消息至 ZDO 层：

```
ZDApp_SendMsg( ZDAppTaskID, ZDO_NWK_DISC_CNF, sizeof(ZDO_NetworkDiscoveryCfm_t),
(uint8 *)&msg );
```

ZDO 层接收到该消息后，首先修改设备状态为正在加入网络：

```
devState = DEV_NWK_JOINING;
```

任务事件处理函数将执行请求加入网络事件：

```
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 );
```

当 NWK 层通过调用 MAC 和 PHY 层相关功能函数执行一些列请求加入网络动作后，NWK 层将接收到请求加入网络反馈，即：

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

发送加入网络指示消息至 ZDO 层。

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

ZDO 层接收到该消息后，任务事件处理函数将执行处理加入网络函数：

```
ZDApp_ProcessNetworkJoin();
```

修改设备状态为终端设备：

```
devState = DEV_END_DEVICE;
```

并且根据设备逻辑类型为路由器，发送请求启动路由：

```
NLME_StartRouterRequest( 0, 0, false );
```

当 NWK 层通过调用 MAC 和 PHY 层相关功能函数执行一些列请求启动路由动作后，NWK 层将接收到请求启动路由反馈，即

```
ZDO_StartRouterConfirmCB( ZStatus_t Status )
```

然后设置启动路由事件：

```
osal_set_event( ZDAppTaskID, ZDO_ROUTER_START );
```

ZDO 层任务事件处理函数处理路由启动事件，首先修改设备状态为路由器：

```
devState = DEV_ROUTER;
```

路由器采用电源供电：

```
osal_pwrmgr_device(PWRMGR_ALWAYS_ON );
```

设置 ZDO 状态改变事件：

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

ZDO 层任务事件处理函数将执行 ZDO 更新网络状态事件处理：

```
ZDO_UpdateNwkStatus( devState );
```

此时搜索端点列表，寻找曾经在应用层注册过的端点号，并且将 ZDO 状态改变消息发送给这些端点：

```
zdoSendStateChangeMsg(state, *(pItem->epDesc->task_id));
```

而且确定路由器的 16 位网络地址和 64 位 IEEE 地址：

```
NLME_GetShortAddr();
NLME_GetExtAddr();
```

当应用层接收到 ZDO 状态改变消息后，等待 5s 后启动周期性发送数据事件，时间间隔为 5s：

```
osal_start_timerEx( SerialApp_TaskID,SERIALAPP_SEND_MSG_EVT,SERIALAPP_SEND_MSG_TIMEOUT );
```

# 第七章 终端设备加入网络流程分析

## 7.1 终端设备设备类型和初始状态

终端设备的 IAR 工程配置选项中没有定义 BUILD_ALL_DEVICES，因此在 ZGlobals.h 文件中：

```
#define ZSTACK_DEVICE_BUILD                0x04
```

进一步有：

```
#define ZG_BUILD_ENDDEVICE_TYPE                0x04
```

从而：

```
#define DEVICE_LOGICAL_TYPE                0x02
```

由此，在 ZGlobals.c 文件中，可以得知终端设备的设备逻辑类型为：

```
zgDeviceLogicalType =0x02
```

以及：

```
#define ZG_DEVICE_ENDDEVICE_TYPE           1
#define ZG_DEVICE_JOINING_TYPE             1
```

终端设备的 IAR 工程配置选项中没有定义阻止自定启动，因此在 ZDApp.c 文件中定义了设备初始状态和启动模式：

```
devState = DEV_INIT
devStartMode = MODE_JOIN
```

## 7.2 终端设备加入网络流程

详细的加入网络流程图如图 7.1 所示：

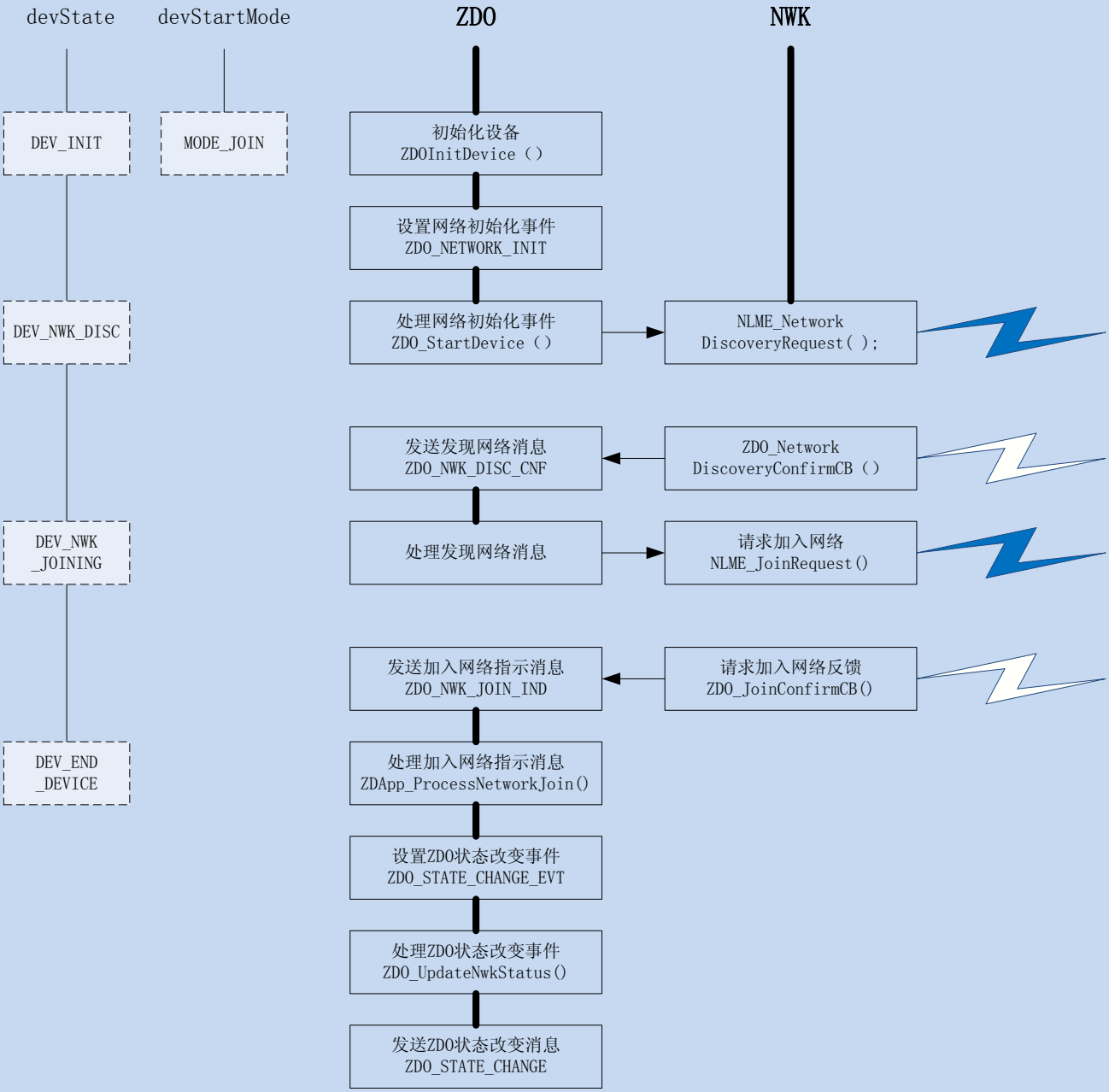


图 7.1 终端设备加入网络流程分析

当终端设备上电后，首先经历一系列的初始化工作，最终调用 ZDO 层的初始化设备函数：

```
ZDOInitDevice(zgStartDelay);
```

在该函数中设置了 NV 网络状态，并修改了当前设备状态：

```
networkStateNV = ZDO_INITDEV_NEW_NETWORK_STATE;
```

最终触发网络初始化函数：

```
ZDApp_NetworkInit( extendedDelay );
```

设置网络初始化事件：

```
osal_set_event( ZDAppTaskID, ZDO_NETWORK_INIT );
```

ZDO 层的任务事件处理函数对网络初始化事件进行处理，即启动该设备：

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

此时将改变设备状态为发现网络：

```
devState = DEV_NWK_DISC;
```

并根据设备逻辑类型和启动模式调用 NWK 层发现网络请求函数：

```
NLME_NetworkDiscoveryRequest( zgDefaultChannelList, zgDefaultStartingScanDuration );
```

其中，默认通道号在 f8wConfig.Cfg 中定义：

```
-DDEFAULT_CHANLIST=0x00000800   // 11 - 0x0B
```

当 NWK 层通过调用 MAC 和 PHY 层相关功能函数执行一些列发现网络动作后，NWK 层将接收到发现网络反馈，即：

```
ZDO_NetworkDiscoveryConfirmCB( uint8 ResultCount, networkDesc_t *NetworkList )
```

发送发现网络消息至 ZDO 层：

```
ZDApp_SendMsg( ZDAppTaskID, ZDO_NWK_DISC_CNF, sizeof(ZDO_NetworkDiscoveryCfm_t),
(uint8 *)&msg );
```

ZDO 层接收到该消息后，首先修改设备状态为正在加入网络：

```
devState = DEV_NWK_JOINING;
```

任务事件处理函数将执行请求加入网络事件：

```
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 );
```

当 NWK 层通过调用 MAC 和 PHY 层相关功能函数执行一些列请求加入网络动作后，NWK 层将接收到请求加入网络反馈，即：

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

发送加入网络指示消息至 ZDO 层。

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

ZDO 层接收到该消息后，任务事件处理函数将执行处理加入网络函数：

```
ZDApp_ProcessNetworkJoin();
```

修改设备状态为终端设备：

```
devState = DEV_END_DEVICE;
```

设置 ZDO 状态改变事件：

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

ZDO 层任务事件处理函数将执行 ZDO 更新网络状态事件处理：

```
ZDO_UpdateNwkStatus( devState );
```

此时搜索端点列表，寻找曾经在应用层注册过的端点号，并且将 ZDO 状态改变消息发送给这些端点：

```
zdoSendStateChangeMsg(state, *(pItem->epDesc->task_id));
```

而且确定终端设备的 16 位网络地址和 64 位 IEEE 地址：

```
NLME_GetShortAddr();
NLME_GetExtAddr();
```

当应用层接收到 ZDO 状态改变消息后，等待 5s 后启动周期性发送数据事件，时间间隔为 5s：

```
osal_start_timerEx( SerialApp_TaskID,SERIALAPP_SEND_MSG_EVT,SERIALAPP_SEND_MSG_TIMEOUT );
```

# 第八章 绑定过程分析

## 8.1 终端设备绑定

手动按下终端节点按键 S1， 向协调器节点发送绑定请求信息，绑定信息的目的地址为协调器节点：

```
dstAddr.addrMode = Addr16Bit;
dstAddr.addr.shortAddr = 0x0000; // Coordinator
```

通过调用以下函数发送绑定请求信息：

```
ZDP_EndDeviceBindReq( &dstAddr, NLME_GetShortAddr(), SerialApp_epDesc.endPoint,
SERIALAPP_PROFID,SERIALAPP_MAX_CLUSTERS, (cId_t *)SerialApp_ClusterList,
SERIALAPP_MAX_CLUSTERS, (cId_t *)SerialApp_ClusterList,FALSE );
```

该函数实际调用无线发送函数将绑定请求发送给协调器节点：

```
fillAndSend( &ZDP_TransID, dstAddr, End_Device_Bind_req, len )
```

对于协调器节点而言，其在 ZDApp 函数初始化时注册了终端绑定请求信息：

```
void ZDApp_RegisterCBs( void )
{
#if ZG_BUILD_COORDINATOR_TYPE
  ZDO_RegisterForZDOMsg( ZDAppTaskID, End_Device_Bind_req );
#endif
}
```

因此，协调器节点的 ZDApp 接收到外界输入的数据后，由于注册了 ZDO 反馈消息，即 ZDO_CB_MSG，ZDApp 层任务事件处理函数将进行处理：

```
ZDApp_ProcessMsgCBs( (zdoIncomingMsg_t *)msgPtr );
```

然后，根据 ClusterID（这里是 End_Device_Bind_req）选择相对应的匹配描述符处理函数，首先解析绑定请求信息：

```
ZDO_ParseEndDeviceBindReq( inMsg, &bindReq );
```

然后向发送绑定请求的节点发送绑定响应消息：

```
ZDO_MatchEndDeviceBind( &bindReq );
```

该函数首先设置目的地址：

```
dstAddr.addrMode = Addr16Bit;
```

```
dstAddr.addr.shortAddr = bindReq->srcAddr;
```

然后调用发送终端绑定响应函数：

```
ZDP_EndDeviceBindRsp( bindReq->TransSeq, &dstAddr, status, bindReq->SecurityUse )
```

由于终端节点在 SerialApp.c 中层注册过 End_Device_Bind_rsp 消息，因此当接收到协调器节点发来的绑定响应消息将交由 SerialApp 任务事件处理函数处理：

```
SerialApp_ProcessZDOMsgs( (zdoIncomingMsg_t *)MSGpkt );
```

该函数根据命令 ID 号做出相应处理：

```
switch ( inMsg->clusterID )
{
    case End_Device_Bind_rsp:
      {
        // Light LED
        HalLedSet( HAL_LED_4, HAL_LED_MODE_ON );
      }
}
```

## 8.2 匹配描述符绑定

手动按下终端节点按键 S2， 向所有节点广播发送绑定请求信息，绑定信息的目的地址为广播地址：

```
dstAddr.addr.shortAddr = NWK_BROADCAST_SHORTADDR;
```

调用匹配描述符请求函数广播发送绑定请求：

```
ZDP_MatchDescReq( &dstAddr, NWK_BROADCAST_SHORTADDR,
                  SERIALAPP_PROFID,
                  SERIALAPP_MAX_CLUSTERS, (cId_t *)SerialApp_ClusterList,
                  SERIALAPP_MAX_CLUSTERS, (cId_t *)SerialApp_ClusterList,
                  FALSE );
```

其中，地址模式 destination 为 16 位网络地址模式；16 位网络地址为广播地址；应用程序配置文件 AppProfId 为 0x0F04。 请求匹配描述符函数最后调用：

```
fillAndSend( &ZDP_TransID, dstAddr, Match_Desc_req, len );
```

其中，传输序号 ZDP_TransID 由 0 开始逐步递增；目的地址模式和地址 dstAddr 为 16 位网络地址模式和广播地址；命令 ID 为 Match_Desc_req；数据包长度 len 为：nwkAddr+ProfileID

+NumInClusters+NumOutClusters，单位为字节。

该函数最终通过调用无线发送数据包函数将匹配消息（Match_Desc_req）发送出去：

AF_DataRequest(&afAddr,&ZDApp_epDesc,clusterID,(uint16)(len+1), (uint8*)(ZDP_TmpBuf-1), transSeq, ZDP_TxOptions, AF_DEFAULT_RADIUS );

其中，目的地址 afAddr 为 16 位网络地址模式和广播地址；端点号为：端点 0， 命令 ID 号为：Match_Desc_req；发送选项为 ZDP_TXOptions，即 TXOption=0；跳数为 AF_DEFAULT_RADIUS，即 Radius=0x14。

并且将绑定标志位设置成绑定请求，即：sapi_bindInProgress= Match_Desc_req

协调器节点的 ZDApp 接收到外界输入的数据后，即 AF_INCOMING_MSG_CMD，ZDApp 层任务事件处理函数将进行处理：

ZDP_IncomingData( (afIncomingMSGPacket_t *)msgPtr );

然后，根据 ClusterID（这里是 Match_Desc_req）查找结构体数组函数，找到相对应的匹配描述符处理函数：

ZDO_ProcessMatchDescReq( zdoIncomingMsg_t *inMsg )

该函数最终将发送匹配描述符响应至请求匹配描述的发起者：

ZDP_MatchDescRsp( inMsg->TransSeq, &(inMsg->srcAddr), ZDP_SUCCESS, ZDAppNwkAddr.addr.shortAddr, epCnt, (uint8 *)ZDOBuildBuf, inMsg->SecurityUse )

该函数实质调用匹配描述符响应的 API 函数：

ZDP_EPRsp( uint16 MsgType, byte TransSeq, zAddrType_t *dstAddr,byte Status, uint16 nwkAddr, byte Count,uint8 *pEPList, byte SecurityEnable )

该函数最终调用：

FillAndSendTxOptions( &TransSeq, dstAddr, MsgType, len, txOptions )

其中 txOptions= AF_MSG_ACK_REQUEST。该函数最终仍然是调用无线发送数据包函数将绑定响应直接发送至请求绑定的节点（终端节点）：

AF_DataRequest( &afAddr, &ZDApp_epDesc, clusterID, (uint16)(len+1), (uint8*)(ZDP_TmpBuf-1), transSeq, ZDP_TxOptions, AF_DEFAULT_RADIUS );

终端节点的 ZDApp 接收到外界输入的数据后，即 AF_INCOMING_MSG_CMD，ZDApp 层任务事件处理函数将进行处理：

```
ZDP_IncomingData( (afIncomingMSGPacket_t *)msgPtr );
```

该函数将绑定响应消息发送至 SerialApp 层：

```
ZDO_SendMsgCBs( &inMsg );
```

即：

```
msgPtr->hdr.event = ZDO_CB_MSG;
osal_msg_send( pList->taskID, (uint8 *)msgPtr );
```

SerialApp 层接收到绑定响应消息后，交由 sapi 层任务事件处理函数处理：

```
SAPI_ProcessZDOMsgs( (zdoIncomingMsg_t *)pMsg );
```

由于已经找到绑定节点，此时将绑定节点的地址设为目的地址：

```
switch ( inMsg->clusterID )
{
  case Match_Desc_rsp:
    {
        SerialApp_DstAddr.addrMode = (afAddrMode_t)Addr16Bit;
        SerialApp_DstAddr.addr.shortAddr = pRsp->nwkAddr;

        HalLedSet( HAL_LED_4, HAL_LED_MODE_ON );
    }
}
```

# 第九章 发送数据

由于在 SerialApp.c 中初始化目的地址模式设定为绑定模式，因此实现数据通信之前必须进行源地址与目的地址绑定。在本例中，当电脑或微控制器通过串口发送数据给 CC2530 时，调用串口回调函数 SerialApp_CallBack（），然后通过调用以下函数向远程节点发送数据包：

```
SerialApp_Send();
```

该函数最终将调用无线发送数据函数：

```
AF_DataRequest(&SerialApp_TxAddr,
                (endPointDesc_t *)&SerialApp_epDesc,
                SERIALAPP_CLUSTERID1,
                SerialApp_TxLen+1, SerialApp_TxBuf,
                &SerialApp_MsgID, 0, AF_DEFAULT_RADIUS)
```

其中，目的地址为绑定地址。数据包的内容即为串口输出的内容。

# 第十章 接收数据

节点接收到外界输入的数据后，即 AF_INCOMING_MSG_CMD，应用层任务事件处理函数将进行处理：

```
SerialApp_MessageMSGCB( MSGpkt );
```

根据命令 ID 号 SERIALAPP _CLUSTERID1 可以将接收到的字符串打印在 LCD 显示屏上：

```
HalUARTWrite( SERIAL_APP_PORT, pkt->cmd.Data+1, (pkt->cmd.DataLength-1) )
```

然后计算数据包在空气中传输的延迟：

```
delay = (stat == OTA_SER_BUSY) ? SERIALAPP_NAK_DELAY : SERIALAPP_ACK_DELAY;
```

并且设置发送响应事件：

```
osal_set_event( SerialApp_TaskID, SERIALAPP_RESP_EVT );
```

交由应用层进行处理，应用层将会调用发送响应反馈函数：

```
SerialApp_Resp();
```

该函数实际是调用无线发送数据函数：

```
AF_DataRequest(&SerialApp_RxAddr,
                    (endPointDesc_t *)&SerialApp_epDesc,
                    SERIALAPP_CLUSTERID2,
                    SERIAL_APP_RSP_CNT, SerialApp_RspBuf,
                    &SerialApp_MsgID, 0, AF_DEFAULT_RADIUS)
```