

2010

ZigBee 2007 无线系统

TI-CC2530

远程数据采集

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	5
2.1	工程架构	5
2.2	工程选项设置	7
第三章	App 初始化和任务事件处理	11
3.1	App 初始化	11
3.2	App 任务事件处理函数	11
第四章	ZDO 初始化和任务事件处理	13
4.1	ZDO 初始化	13
4.2	ZDO 任务事件处理函数	13
第五章	采集节点建立网络流程分析	15
5.1	采集节点设备类型和初始状态	15
5.2	采集节点建立网络流程	15
第六章	传感节点加入网络流程分析	20
6.1	传感节点设备类型和初始状态	20
6.2	传感节点建立网络流程	20
第七章	采集节点与传感节点绑定分析	25
第八章	传感节点发送数据	30
第九章	采集节点接收数据	32

第一章 功能描述

本工程的目录为：

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

在远程数据采集中，传感节点（sensor node）将采集到的传感数据发送至远端的采集节点（collection node），以使用户进行离线的分析和处理。多个传感节点与唯一的采集节点组成星型网络拓扑结构。

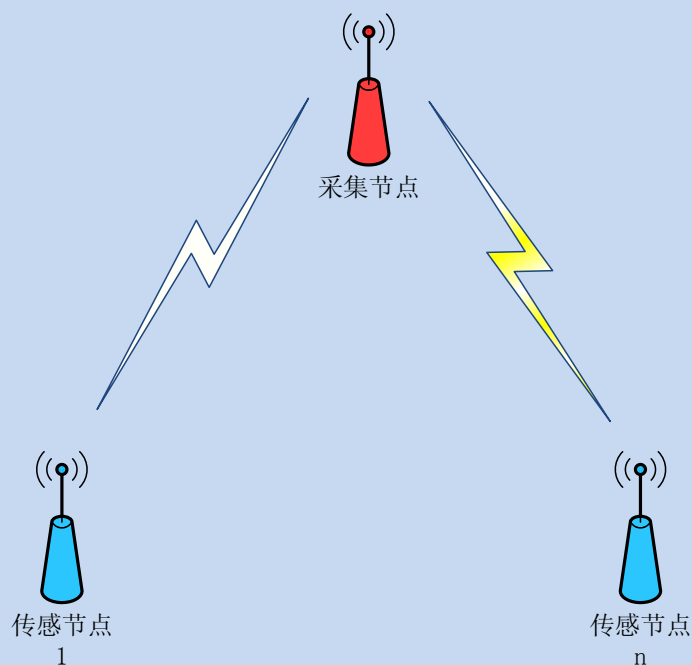
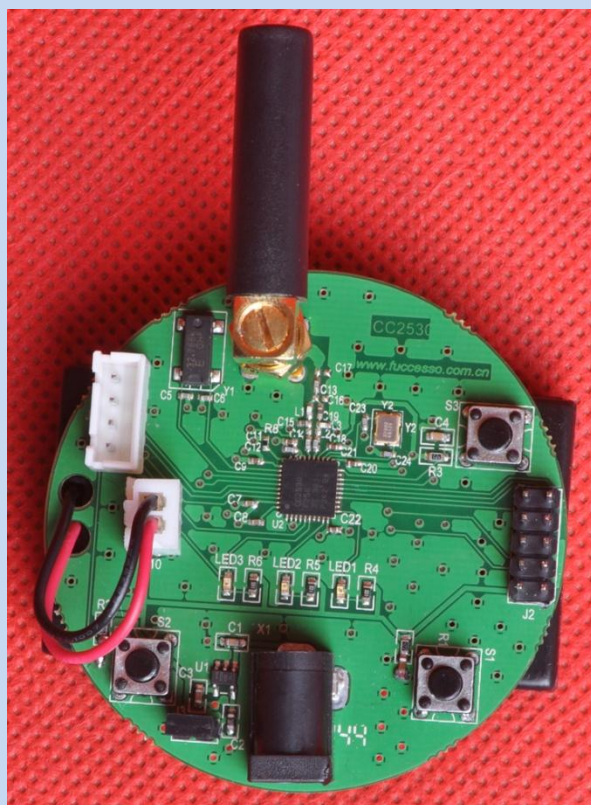


图 1.1 远程数据采集网络拓扑结构

本例中，传感节点使用锋硕电子开发的终端节点 CC2530，记录当前温度值和电压值：



采集节点使用协调器节点 CC2530+GPRS，使用串口终端显示并打印传感数据：



为实现上述目的，整个应用程序应该具备以下功能：

- 1) 采集节点自动建立网络（注：首次启动时，需按下按键 S1 配置自动启动）
- 2) 传感节点自动发现采集节点，而且一旦加入网络后，能够自动与采集节点建立绑定；
- 3) 传感节点能够周期性地向采集节点发送数据，并且用端到端确认（end-to-end ACK）的发送模式；
- 4) 如果传感节点没有收到采集节点的确认消息，它将解除与该采集节点的绑定，然后，重新发现采集节点并与其建立绑定。

第二章 工程整体架构和选项设置

2.1 工程架构

用户打开 SimpleApp 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。可以发现，每种设备类型都包含 sapi.c、sapi.h 和 SimpleApp.h 文件，即每种设备程序运行时的任务初始化函数、任务事件处理函数均是在 sapi.c 中实现。

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)。

对于采集节点, 在 Workspace 区域的下拉菜单中选择 SimpleCollectorEB, 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 SimpleCollector.c 是进行编译的文件, 而 SimpleController.c、SimpleSensor.c 和 SimpleSwitch.c 颜色呈灰色, 表示这 3 个文件对于采集节点而言不会使用。采集节点在整个网络中实现 ZigBee 协调器的功能, 因此配置文件 f8wCoord.cfg 将被使用, 而 f8wEndev.cfg 和 f8wRouter.cfg 不会使用。如图 2.1 所示:

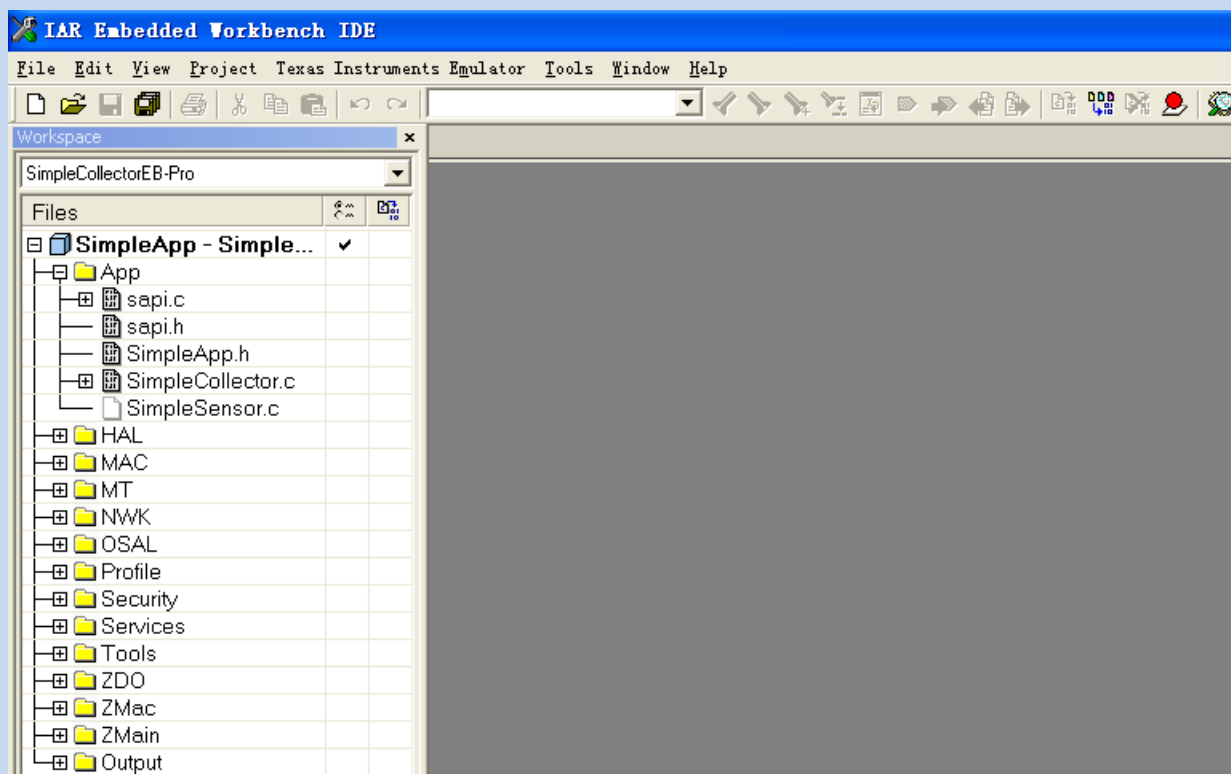


图 2.1 采集节点工程架构

对于传感节点,在 Workspace 区域的下拉菜单中选择 SimpleSensorEB,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 SimpleSensor.c 是进行编译的文件,而 SimpleCollector.c、SimpleController.c 和 SimpleSwitch.c 颜色呈灰色,表示这 3 个文件对于传感节点而言不会使用。如下图所示:

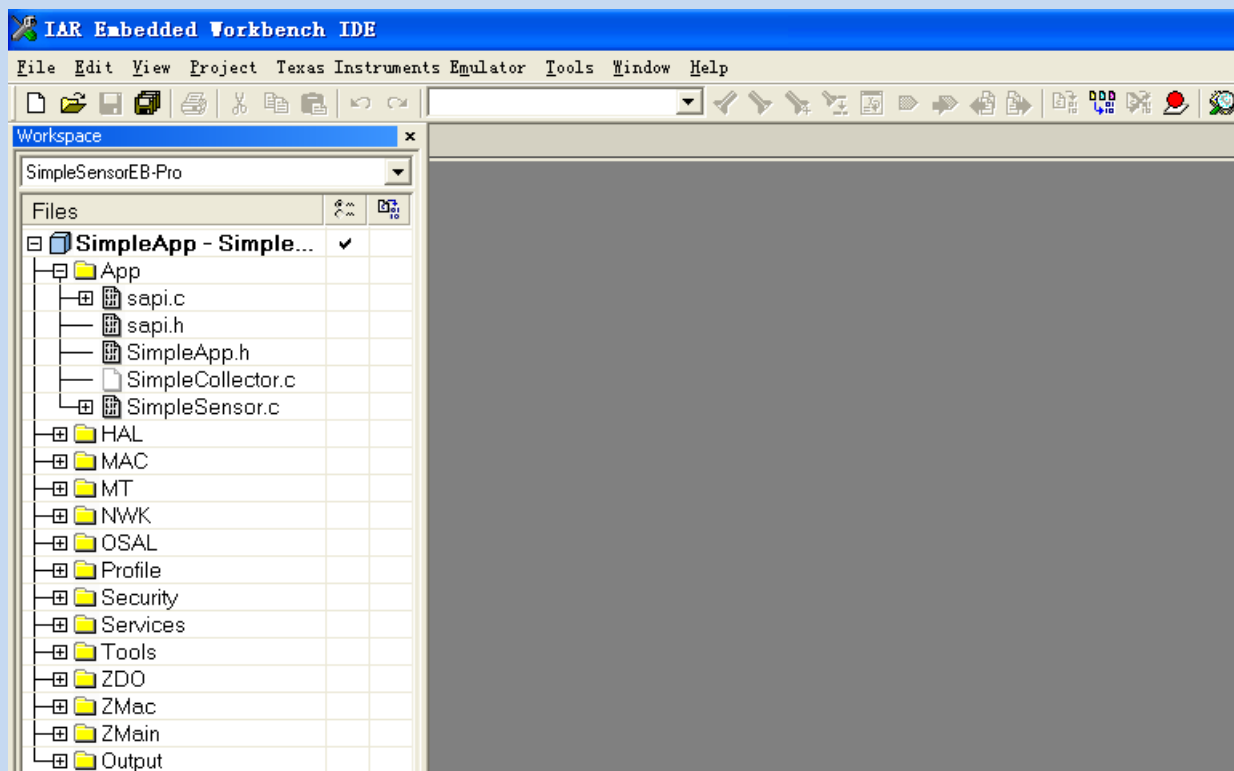


图 2.2 传感节点工程架构

2.2 工程选项设置

打开 SimpleApp 工程后,欲进入到采集节点的编译选项设置界面。选中工程名 SimpleApp-SimpleCollectorEB,然后根据工程选项设置的路径: Project->Options->C/C++ Compiler->Preprocessor->Defined。

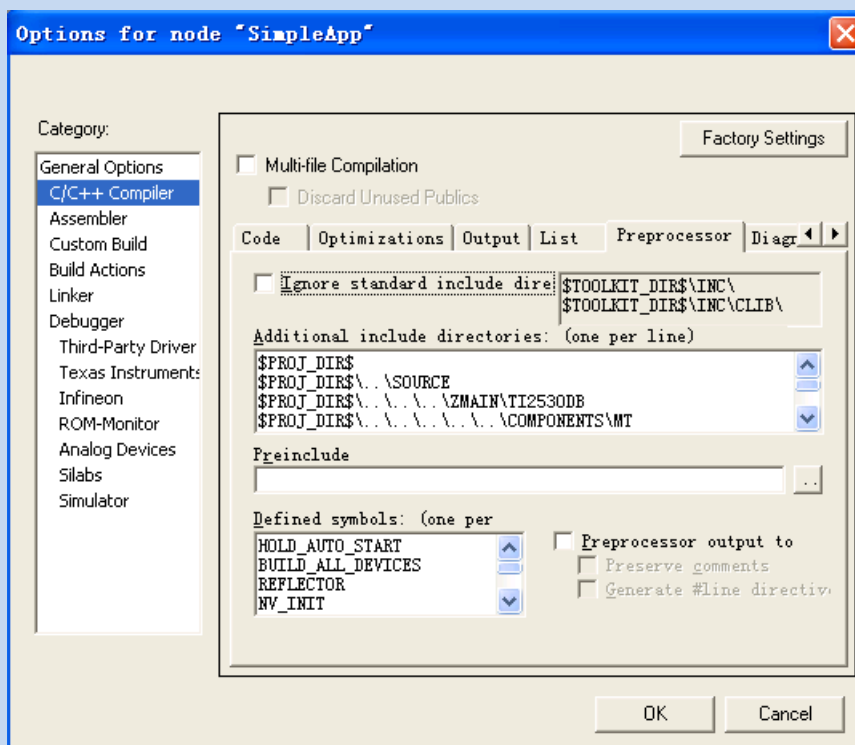


图 2.3 采集节点 IAR 工程选项设置

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。

欲进入到传感节点的编译选项设置界面。选中工程名 SimpleApp-SimpleSensorEB，然后根据工程选项设置的路径：Project->Options->C/C++ Compiler->Preprocessor->Defined。

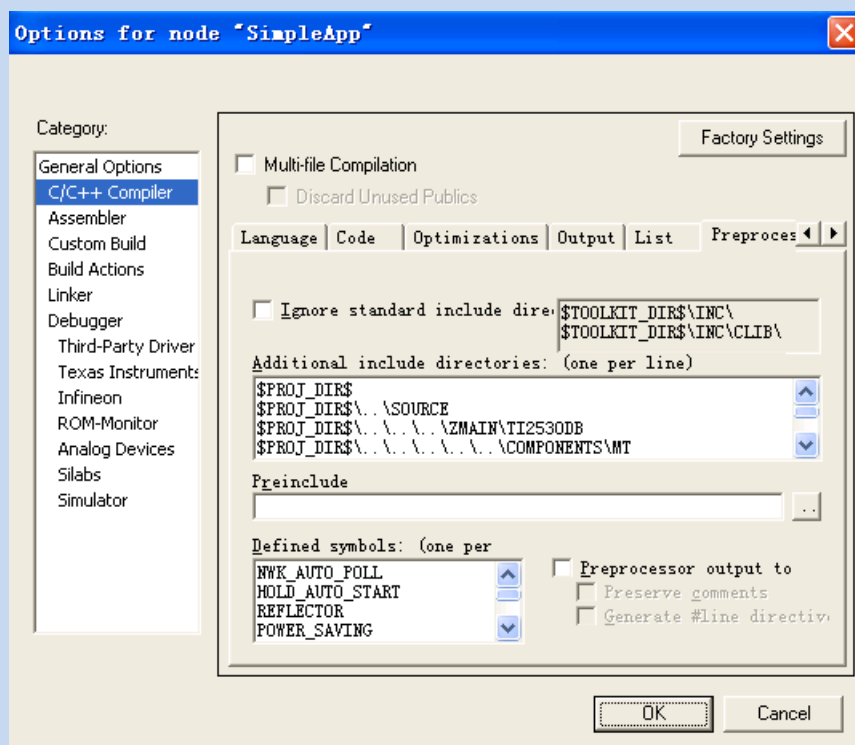


图 2.4 传感节点 IAR 工程选项设置

对于采集节点和传感节点，分别打开 Tools->f8wCoord.cfg 和 Tools->f8wEndev.cfg 后，可以看到关于采集节点传感节点的配置信息。

综上，总结采集节点和传感节点的工程选项设置如下表：

节点类型	IAR 选项设置	.cfg 配置文件
采集节点	ZIGBEEPRO HOLD_AUTO_START BUILD_ALL_DEVICES REFLECTOR NV_INIT xNV_RESTORE ZTOOL_P1 MT_TASK MT_SYS_FUNC xMT_SAPI_FUNC xMT_SAPI_CB_FUNC	-DZDO_COORDINATOR -DRTR_NWK
传感节点	ZIGBEEPRO NWK_AUTO_POLL HOLD_AUTO_START REFLECTOR POWER_SAVING NV_INIT	空

	xNV_RESTORE xZTOOL_P1 xMT_TASK xMT_SYS_FUNC xMT_SAPI_FUNC xMT_SAPI_CB_FUNC	
--	---	--

第三章 App 初始化和任务事件处理

3.1 App 初始化

Sapi.c 中的 SAPI_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。

```
sapi_TaskID = task_id;
```

- 2) 初始化绑定标志位，默认不允许绑定。

```
sapi_bindInProgress = 0xffff;
```

- 3) 初始化端点描述符，分别有：端点任务 ID 号、端点号、简单描述符和延时请求，端点描述符内部的简单描述符由应用程序决定。

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

- 4) 在 AF 层注册该端点描述符。

```
afRegister( &sapi_epDesc );
```

- 5) 关闭匹配描述符的响应。

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

- 6) 注册 2 个响应事件，即网络地址响应和匹配描述符响应。

```
ZDO_RegisterForZDOMsg( sapi_TaskID, NWK_addr_rsp );  
ZDO_RegisterForZDOMsg( sapi_TaskID, Match_Desc_rsp );
```

- 7) 注册按键事件。

```
RegisterForKeys( sapi_TaskID );
```

- 8) 设置进入事件，启动应用。

```
osal_set_event(task_id, ZB_ENTRY_EVENT);
```

3.2 App 任务事件处理函数

在 SAPI_Init () 函数的最后设置了进入事件 (ZB_ENTRY_EVENT)，这将出发任务的事

件处理函数 SAPI_ProcessEvent ()。该函数处理任务所有的事件，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 SYS_EVENT_MSG	ZDO 反馈 ZDO_CB_MSG	SAPI_ProcessZDOMsgs ()
	AF 数据确认 AF_DATA_CONFIRM_CMD	SAPI_SendDataConfirm ()
	AF 信息输入 AF_INCOMING_MSG_CMD	SAPI_ReceiveDataIndication ()
	ZDO 状态改变 ZDO_STATE_CHANGE	SAPI_StartConfirm ()
	ZDO 匹配描述符响应 ZDO_MATCH_DESC_RSP_SENT	SAPI_AllowBindConfirm ()
	按键 KEY_CHANGE	zb_HandleKeys ()
	发送数据确认 SAPICB_DATA_CNF	SAPI_SendDataConfirm ()
	绑定确认 SAPICB_BIND_CNF	SAPI_BindConfirm ()
	设备启动确认 SAPICB_START_CNF	SAPI_StartConfirm ()
允许绑定时间事件 ZB_ALLOW_BIND_TIMER		afSetMatch ()
绑定时间事件 ZB_BIND_TIMER		SAPI_BindConfirm ()
进入事件 ZB_ENTRY_EVENT		zb_StartRequest ()

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) 启动设备 (非阻止情况)

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

第五章 采集节点建立网络流程分析

5.1 采集节点设备类型和初始状态

采集节点的 IAR 工程配置选项中定义了 BUILD_ALL_DEVICES，因此在 ZGlobals.h 文件中：

```
#define ZSTACK_DEVICE_BUILD 0x07
```

进一步有：

```
#define ZG_BUILD_COORDINATOR_TYPE 0x01
```

从而：

```
#define DEVICE_LOGICAL_TYPE 0x00
```

由此，在 ZGlobals.c 文件中，可以得知采集节点的设备逻辑类型为协调器：

```
zgDeviceLogicalType = 0x00
```

以及：

```
#define ZG_DEVICE_COORDINATOR_TYPE 1  
#define ZG_DEVICE_RTR_TYPE 1
```

采集节点的 IAR 工程配置选项中定义了阻止自定启动，即 HOLD_AUTO_START，因此在 ZDApp.c 文件中定义了设备初始状态和启动模式：

```
devState = DEV_HOLD  
devStartMode = MODE_HARD
```

在采集节点的 NV 中，默认状态下没有设置启动模式，即

```
ZCD_NV_STARTUP_OPTION=0
```

因此，初次使用采集节点时不会自定启动该节点。

另外，对应采集节点的 SimpleCollector.c 定义了应用层的状态：

```
myAppState = APP_INIT
```

5.2 采集节点建立网络流程

当采集节点上电后，首先经历一系列的初始化工作，最终在 sapi 层设置进入事件，然后通

过任务事件处理函数对该事件进行处理，当读取 NV 的启动模式选项时，

```
zb_ReadConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );
```

判断为非自动启动，因此看到采集节点的 LED_2 闪烁，

```
HalLedBlink(HAL_LED_2, 0, 50, 500);
```

操作系统等待其他事件发生。

当按下按键 S1 后，由于在 sapi 层注册了按键事件，因此会发送 KEY_CHANGE 消息至 sapi 层，当收到 KEY_CHANGE 消息后，sapi 层的任务事件处理函数调用：

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

然后将设备逻辑类型（协调器）写入到 NV，并将自动启动模式写入到 NV：

```
zb_WriteConfiguration(ZCD_NV_LOGICAL_TYPE, sizeof(uint8), &logicalType);  
zb_WriteConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );
```

最后重新启动：

```
zb_SystemReset();
```

详细的网络形成流程图如图 5.1 所示：

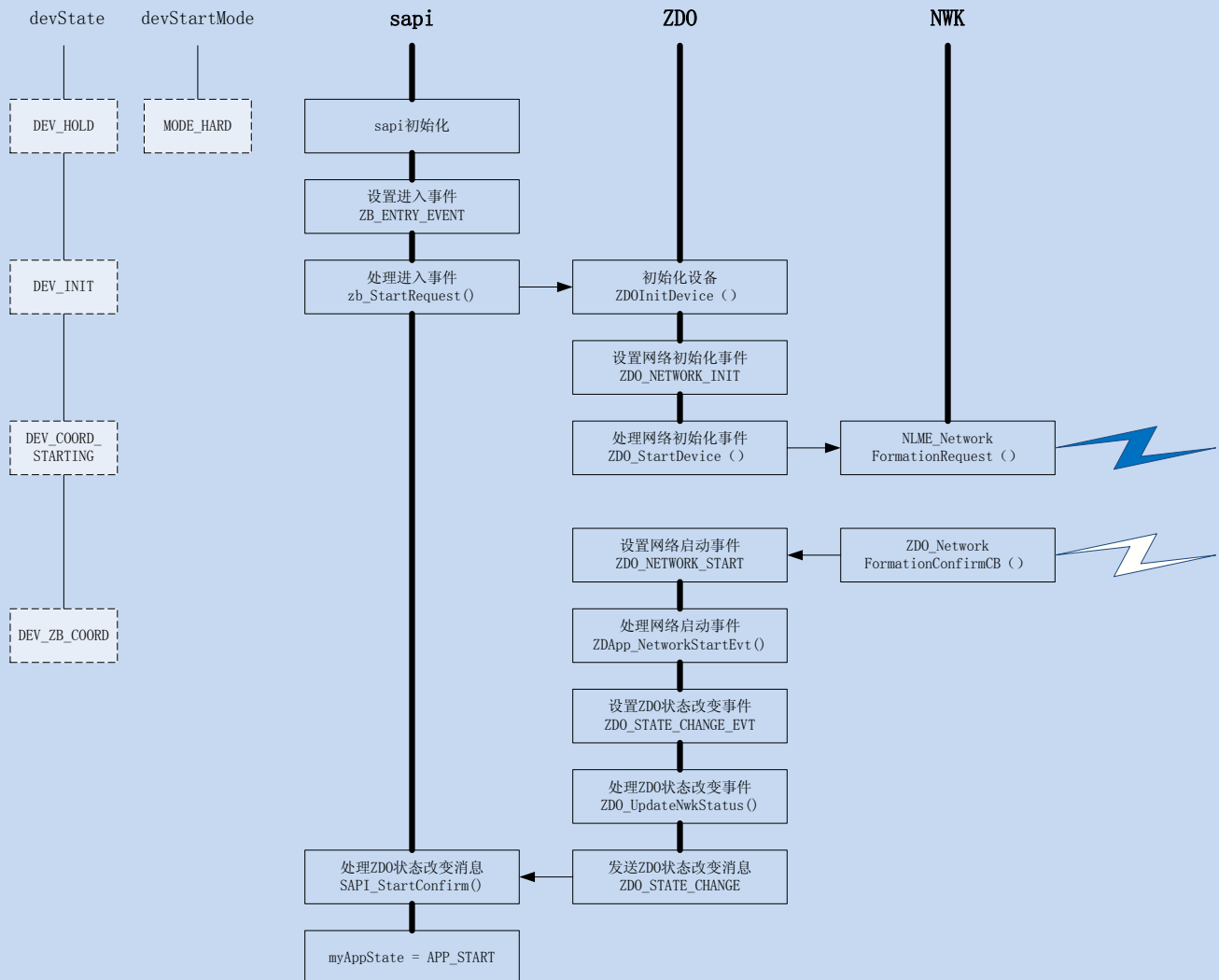


图 5.1 采集节点网络形成流程分析

重新启动后，依然进入通过任务事件处理函数对进入事件进行处理，当读取 NV 的启动模式选项时，判断为自动启动，然后调用：

```
zb_StartRequest();
```

紧接着调用 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;
devState = DEV_INIT;
```

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 sapi 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, *(pltem->epDesc->task_id));
```

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

```
NLME_GetShortAddr();
```

```
NLME_GetExtAddr();
```

当 sapi 层接收到 ZDO 状态改变消息后，sapi 层的任务事件处理函数将进行处理：

```
SAPI_StartConfirm( ZB_SUCCESS );
```

最终将改变设备的应用状态为启动状态：

```
myAppState = APP_START;
```

第六章 传感节点加入网络流程分析

6.1 传感节点设备类型和初始状态

传感节点的 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，可以得知传感节点的设备逻辑类型为终端设备：

```
zgDeviceLogicalType = 0x02
```

以及：

```
#define ZG_DEVICE_ENDDEVICE_TYPE 1
```

传感节点的 IAR 工程配置选项中定义了阻止自定启动，即 HOLD_AUTO_START，因此在 ZDApp.c 文件中定义了设备初始状态和启动模式：

```
devState = DEV_HOLD  
devStartMode = MODE_JOIN
```

在传感节点的 NV 中，默认状态下没有设置启动模式，即

```
ZCD_NV_STARTUP_OPTION=0
```

因此，初次使用传感节点时不会自定启动该节点。

另外，对应传感节点的 Sensor.c 定义了应用层的状态：

```
myAppState = APP_INIT
```

6.2 传感节点建立网络流程

当传感节点上电后，首先经历一系列的初始化工作，最终在 sapi 层设置进入事件，然后通过任务事件处理函数对该事件进行处理，当读取 NV 的启动模式选项时，

```
zb_ReadConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );
```

判断为非自动启动，因此看到传感节点的 LED_2 闪烁，

```
HalLedBlink(HAL_LED_2, 0, 50, 500);
```

操作系统等待其他事件发生。

当按下按键 S1 后，由于在 sapi 层注册了按键事件，因此会发送 KEY_CHANGE 消息至 sapi 层，当收到 KEY_CHANGE 消息后，sapi 层的任务事件处理函数调用：

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

然后将设备逻辑类型（终端设备）写入到 NV，并将自动启动模式写入到 NV：

```
zb_WriteConfiguration(ZCD_NV_LOGICAL_TYPE, sizeof(uint8), &logicalType);
```

```
zb_WriteConfiguration( ZCD_NV_STARTUP_OPTION, sizeof(uint8), &startOptions );
```

最后重新启动：

```
zb_SystemReset();
```

详细的加入网络流程图如图 6.1 所示：

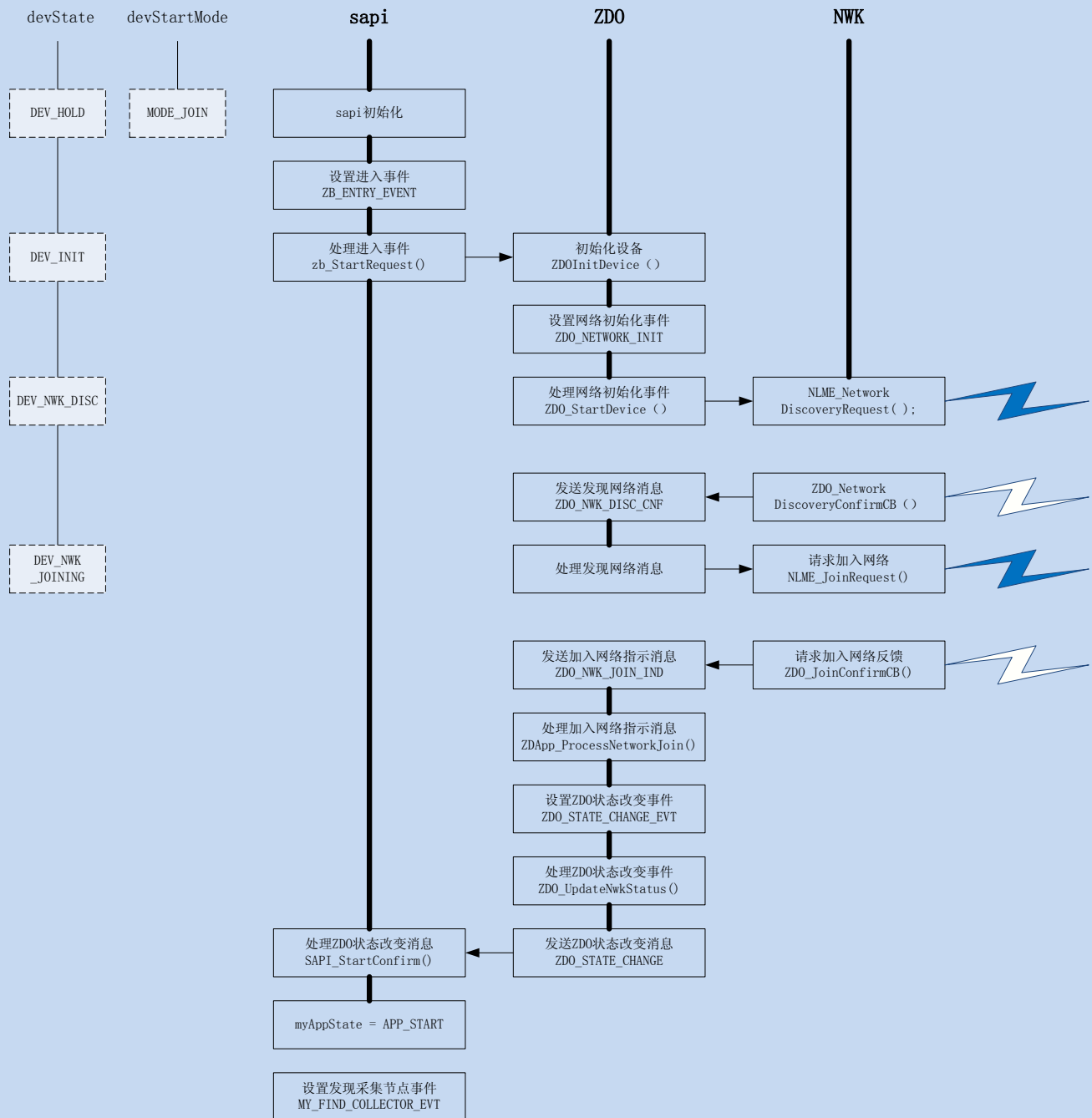


图 6.1 传感节点加入网络流程分析

重新启动后，依然进入通过任务事件处理函数对进入事件进行处理，当读取 NV 的启动模式选项时，判断为自动启动，然后调用：

`zb_StartRequest();`

紧接着调用 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;  
devState = DEV_INIT;
```

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设备发现网络:

```
devState = DEV_NWK_DISC;
```

并调用请求发现网络函数:

```
NLME_NetworkDiscoveryRequest( zgDefault_ChannelList, zgDefaultStarting_ScanDuration );
```

其中, 通道号为协调器所在通道。

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 )
```

其中, 网络参数列表为该网络详细的网络参数。并将其中的个域网 ID、逻辑通道、版本号和扩展个域网 ID 号组成网络发现确认消息发送至 ZDO 层:

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

此时将改变设备状态为正在加入网络:

```
devState = DEV_NWK_JOINING;
```

ZDO 层接收到该消息后, 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();
```

设置 ZDO 状态改变事件：

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

由于 IAR 设置了节点选项，所以采用电池供电：

```
osal_pwrmgr_device( PWRMGR_BATTERY );
```

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 sapi 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, *(pltem->epDesc->task_id));
```

而且确定传感节点（此时为终端设备）的 16 位网络地址和 64 位 IEEE 地址：

```
NLME_GetShortAddr();
```

```
NLME_GetExtAddr();
```

当 sapi 层接收到 ZDO 状态改变消息后，sapi 层的任务事件处理函数将进行处理：

```
SAPI_StartConfirm( ZB_SUCCESS );
```

最终将改变设备的应用状态为启动状态：

```
myAppState = APP_START;
```

并且设置经绑定延迟后发现采集节点事件：

```
osal_start_timerEx( sapi_TaskID, MY_FIND_COLLECTOR_EVT, myBindRetryDelay );
```


第七章 采集节点与传感节点绑定分析

采集节点在默认情况下（sapi 层初始化时）关闭了匹配描述符响应。当采集节点建立网络后，应用层状态：

```
myAppState = APP_START;
```

通过按下按键 S1 可以开启允许绑定功能：

```
zb_AllowBind( 0xFF );
```

允许绑定的实质即开启匹配描述符响应：

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

采集节点允许绑定的时间范围是 10s，即允许其他节点在 10s 内与它建立绑定关系。

如果需要取消绑定连接，则可以通过按下按键 S2，解除绑定：

```
zb_AllowBind( 0x00 );
```

解除绑定的实质即关闭匹配描述符响应：

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

绑定详细流程如图 7.1 所示：

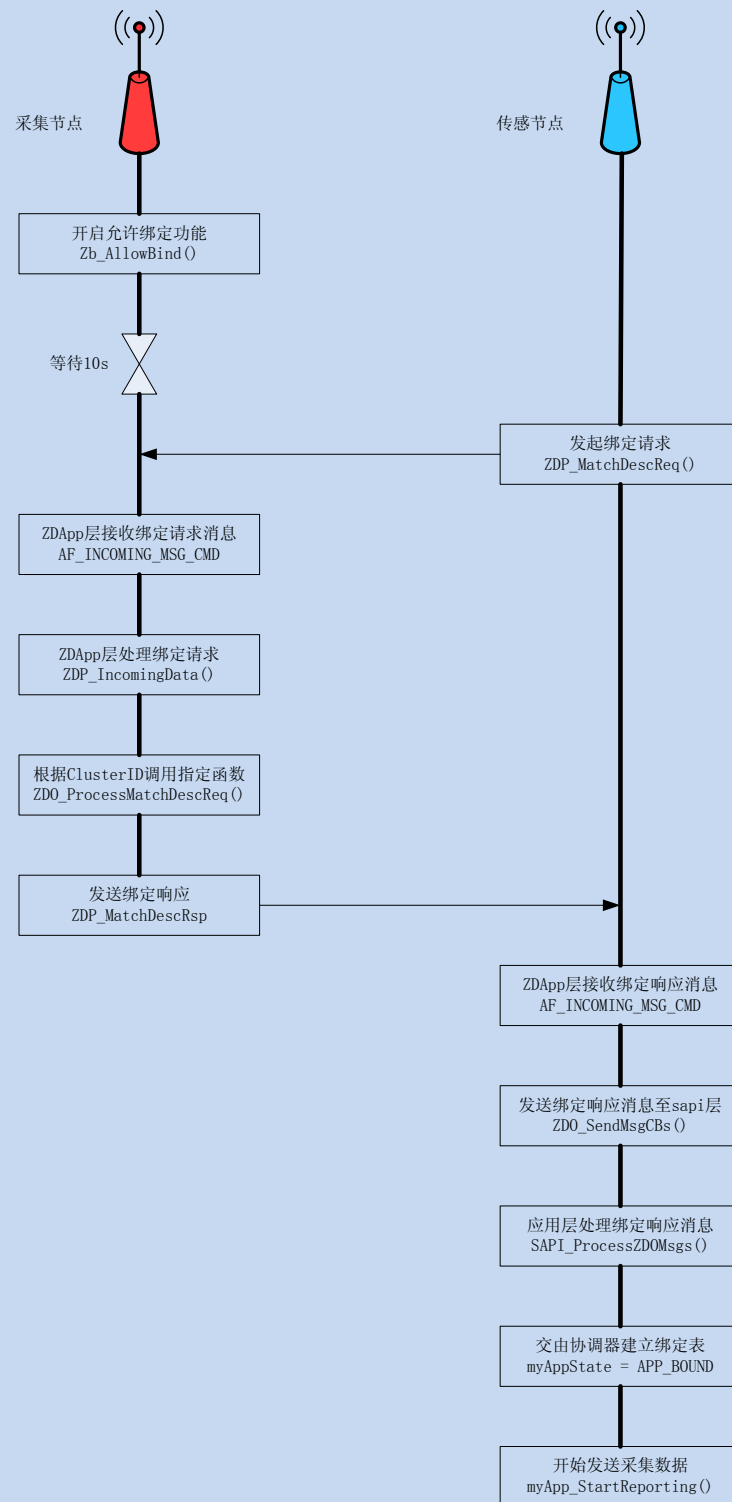


图 7.1 绑定过程分析

传感节点加入网络后，经过绑定延迟，设置了发现采集节点事件。传感节点的 sapi 层任务事件处理函数处理该事件，即发起与采集节点绑定：

```
zb_BindDevice( TRUE, SENSOR_REPORT_CMD_ID, (uint8 *)NULL);
```

其中，指定 64 位 IEEE 目的地址为 NULL。所以将设定目的地址模式为 16 位网络地址，而且

此地址为广播地址：

```
destination.addrMode = Addr16Bit;  
destination.addr.shortAddr = NWK_BROADCAST_SHORTADDR;
```

对于传感节点而言，它完成数据采集并将数据发送至采集节点，因此簇号为 SENSOR_REPORT_CMD_ID 的簇是输出簇；该簇对于采集节点而言是输入簇。当传感节点在输出簇列表中找到该簇后：

```
ZDO_AnyClusterMatches( 1, &commandId, sapi_epDesc.simpleDesc->AppNumOutClusters,  
sapi_epDesc.simpleDesc->pAppOutClusterList )
```

将尽量与一个处于允许绑定模式的设备进行匹配：

```
ZDP_MatchDescReq( &destination, NWK_BROADCAST_SHORTADDR,  
sapi_epDesc.simpleDesc->AppProfId, 1, &commandId, 0, (cld_t *)NULL, 0 );
```

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 0x0F10；输入簇数目为：1；输入簇为：SENSOR_REPORT_CMD_ID；输出簇数目为：0；输出簇为：NULL；

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_t)(len+1), (uint8_t*)(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

采集节点的 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( zdIncomingMsg_t *inMsg )
```

该函数最终将发送匹配描述符响应至传感节点：

```
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：

```
afStatus_t 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（传感节点）：

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

传感节点的 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 sapi 层：

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

即：

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

sapi 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( (zdIncomingMsg_t *)pMsg );
```

根据 inMsg->clusterID 为 Match_Desc_rsp，调用应用支持子层的 API 函数 APSME_BindRequest()，

由协调器建立绑定表。传感节点在建立绑定之后，调用绑定确认函数：

```
zb_BindConfirm( sapi_bindInProgress, ZB_SUCCESS );
```

修改应用层状态为绑定状态：

```
myAppState = APP_BOUND;
```

并开始采集和发送数据至采集节点：

```
myApp_StartReporting();
```

第八章 传感节点发送数据

传感节点设置了 2 种周期性数据采集事件，周期分别为 myTempReportPeriod=5s，myBatteryCheckPeriod=21s：

```
osal_start_timerEx( sapi_TaskID, MY_REPORT_TEMP_EVT, myTempReportPeriod );
osal_start_timerEx( sapi_TaskID, MY_REPORT_BATT_EVT, myBatteryCheckPeriod );
```

这 2 个事件均由 sapi 层任务事件处理函数处理。设定数据的第 1 个字节为数据类型，分别是：温度报告（TEMP_REPORT）和电压报告（BATTERY_REPORT）；数据的第 2 个字节为采集的数据，分别调用温度读取函数（myApp_ReadTemperature()）和电压读取函数（myApp_ReadBattery()）。最终都通过发送数据请求函数将数据发送至远端的采集节点：

```
zb_SendDataRequest( 0xFFFFE, SENSOR_REPORT_CMD_ID, 2, pData, 0, AF_ACK_REQUEST, 0 );
```

其中，命令 ID 号均为 SENSOR_REPORT_CMD_ID。

由于已经与采集节点建立绑定联系，所以目的地址设为 0xFFFFE。该函数最终将调用无线发送数据函数：

```
AF_DataRequest(&dstAddr, &sapi_epDesc, commandId, len, pData, &handle, txOptions, radius);
```

其中，txOptions= AF_ACK_REQUEST。

另外，读取温度和读取电压函数实质是读取 CC2530 内部的温度传感器和电压传感器所采集的数据，并根据数据手册将原始数据进行相应处理。

读取温度：

```
ADCCON3 = (HAL_ADC_REF_125V | HAL_ADC_DEC_512 | HAL_ADC_CHN_TEMP);
while ( !ADCIF );
value = ADCL;
value |= ((uint16) ADCH) << 8;
```

读取电压：

```
ADCCON3 = (HAL_ADC_REF_125V | HAL_ADC_DEC_128 | HAL_ADC_CHN_VDD3);
while ( !ADCIF );
value = ADCL;
value |= ((uint16) ADCH) << 8;
```


第九章 采集节点接收数据

采集节点接收到外界输入的数据后, 即 AF_INCOMING_MSG_CMD, sapi 层任务事件处理

函数将进行处理:

```
SAPI_ReceiveDataIndication( pMSGpkt->srcAddr.addr.shortAddr, pMSGpkt->clusterId,  
pMSGpkt->cmd.DataLength, pMSGpkt->cmd.Data);
```

根据命令 ID 号 SENSOR_REPORT_CMD_ID 和数据类型对温度报告和电压报告进行相应处理,

并且均通过串口将数据在 PC 上显示出来, 所调用的串口打印数据函数为:

```
debug_str( (uint8 *)buf );
```