第 3 章 CHAPTER 3 物联网网络层

3.1 ZStack 协议栈

3.1.1 ZStack 协议栈简介

ZigBee 是基于 IEEE 802.15.4 标准的低功耗局域网协议。该协议的物理层(PHY)和介质访问层(MAC)由 IEEE 802.15.4 标准定义；网络层(NWK)和应用层(APP)则由 ZigBee 联盟定义。

ZStack 是 TI 公司提供的一套符合 ZigBee 协议标准的协议栈。用户可以使用其提供的程序框架和 API 函数进行应用项目的开发。该协议栈经过了 ZigBee 联盟的认可，并且被全球很多企业作为商业级协议栈。实际上，ZStack 只是一个半开源的协议栈，其中的 MAC 层和 ZMAC 层并没有全部开源，但用户可以使用其提供的 API 来调用相关的库函数。ZigBee 协议结构和 ZStack 协议栈结构对比如表 3-1 所示。

表 3-1 ZigBee 协议结构和 ZStack 协议栈结构对比

ZigBee 协议结构	ZStack 协议栈结构
APP 层	APP 层、OSAL
ZDO 层、APS 层	ZDO 层
AF 层	Profile
NWK 层	NWK 层
MAC 层	ZMAC 层、MAC 层
PHY 层	HAL 层、MAC 层
安全服务提供商	Security 与 Services

简单来说，ZigBee 是一个符合国际标准的协议，而 ZStack 则是实现该协议的具体代码。如果前者是一幅建筑图纸，那么后者就是按照图纸修建的建筑物。所以，学习基于 CC2530 芯片的 ZigBee 无线组网技术，实际上就是学习 ZStack 协议栈的结构和运行机理，并且在其基础上进行项目开发。

图 3-1 展示了 ZigBee 无线网络协议层的架构图。ZigBee 的协议分为两部分，IEEE 802.15.4 定义了 PHY(物理层)和 MAC(介质控制访问层)技术规范；ZigBee 盟定义了 NWK(网络层)、APS(应用程序支持子层)、APL(应用层)技术规范。ZigBee 协议栈就是将各个层定义的协议都集合在一起，以函数的形式实现，并给用户提供 API(应用层)，用户可以直接调用。

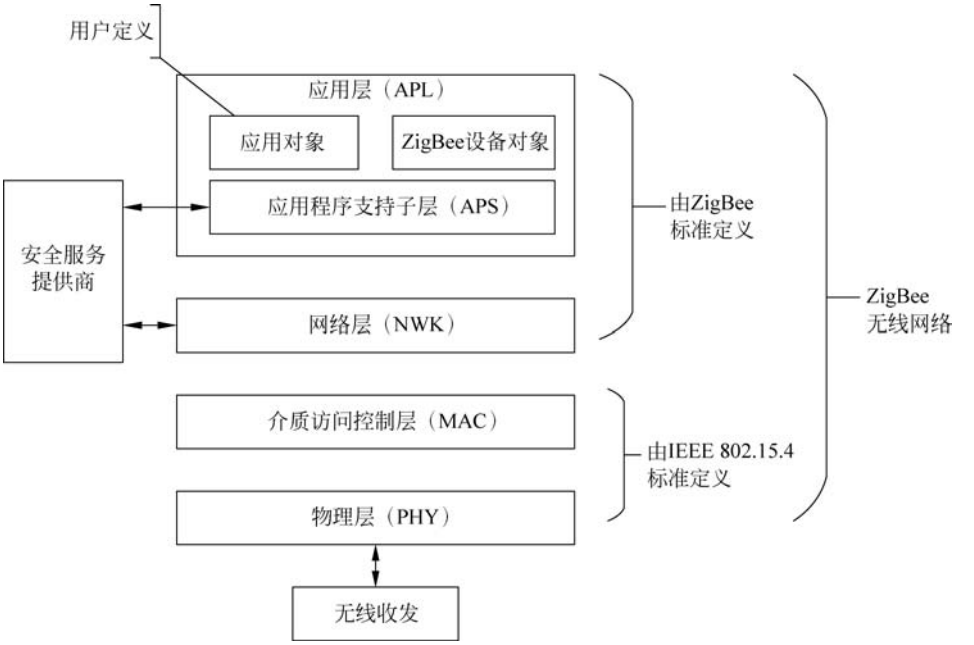


图 3-1 ZigBee 无线网络协议层架构图

在开发一个应用时，协议较底下的层与应用相互独立，它们可以从第三方获得，因此需要做的就只是在应用层进行相应的改动。介绍到这里，大家应该清楚协议和协议栈的关系了吧，是不是会想着怎么样才能用协议栈来开发自己的项目呢？技术总是在不断地发展，我们可以用 ZigBee 厂商提供的协议栈软件来方便地使用 ZigBee 协议栈(注意：不同厂商提供的协议栈是有区别的，此处介绍 TI 推出的 ZigBee 2007 协议栈也称 ZStack)。ZStack 是挪威半导体公司 Chipcon(已经被 TI 公司收购)推出其 CC2430 开发平台时，推出的一款业界领先的商业级协议栈软件。由于这个协议栈软件的出现，用户可以很容易地开发出具体的应用程序，也就是大家说的掌握 10 个函数就能使用 ZigBee 通信的原因。它使用瑞典公司 IAR 开发的 IAREmbedded Workbench for MCS-51 作为它的集成开发环境。Chipcon 公司为自己设计的 ZStack 协议栈中提供了一个名为操作系统抽象层 OSAL 的协议栈调度程序。对于用户来说，除了能够看到这个调度程序外，其他任何协议栈操作的具体实现细节都被封装在库代码中。用户在进行具体的应用开发时只能够通过调用 API 接口来进行，而无权知道 ZigBee 协议栈实现的具体细节，也没必要知道。

图 3-2 是 TI 公司的基于 ZigBee 2007 的协议栈 ZStack-CC2530-2.3.0，所有文件目录如红色框所示。可以把它看作一个庞大的工程，或者是一个小型的操作系统，采用任务轮询的方法运行。

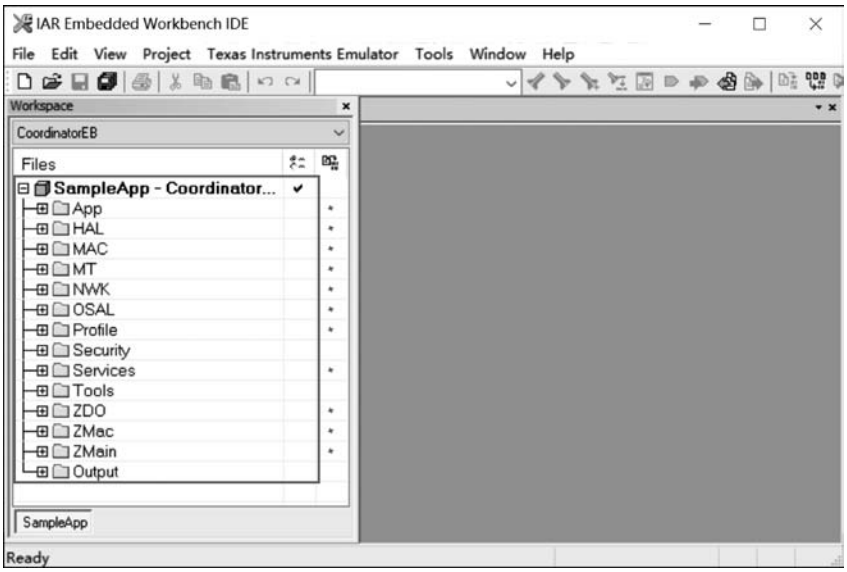


图 3-2　ZStack-CC2530-2.3.0 文件目录结构

其中，各个文件夹的功能如下。

- App：应用层。
- HAL：硬件驱动；优先级为 3，编程时需要考虑。
- MAC：数据链路层，没有源码，只有头文件和必要的 c 文件，优先级为 1。
- MT(Measure and test)：测量与测试，可查看源码，编写程序时可参考；可有可无，优先级为 4。
- NWK：网络层，没有源码；优先级为 2。
- OSAL：操作系统抽象层。
- Profile：应用层规则。
- Security：安全。
- Services：字母转数字等工具、帮助类函数。
- Tools：编译时用到的工具。
- ZDO(ZigBee device object)：ZigBee 开发板上的无线模块管理；优先级为 7，执行 ZDApp_init()函数后，如果是协调器将建立网络，如果是终端设备将加入网络。
- ZMac：通过该层提供的接口调用 MAC 层的函数。

ZigBee 协议栈已经实现了 ZigBee 协议，用户可以使用协议栈提供的 API 进行应用程序的开发，在开发过程中完全不必关心 ZigBee 协议的具体实现细节，要关心的问

题是：应用层的数据是使用哪些函数通过什么方式发送或者接收数据的。因此，最重要的是要学会使用 ZigBee 协议栈。

用户实现一个简单的无线数据通信时的一般步骤如下。

(1) 组网：调用协议栈的组网函数、加入网络函数，实现网络的建立与节点的加入。

(2) 发送：发送节点调用协议栈的无线数据发送函数，实现无线数据发送。

(3) 接收：接收节点调用协议栈的无线数据接收函数，实现无线数据接收。

无线数据通信的步骤非常简单，但无线通信过程是基于 ZStack 协议栈的，因此，要将简单的无线通信步骤加入 ZStack 软件架构中，才能利用 ZStack 提供的 API 完成无线通信功能。这类似于：QQ 是一款简单易用的软件，使用 QQ 的聊天步骤非常简单，只要输入消息并发送就可以了。但用户需要把 QQ 安装于一个操作系统上，如 PC 的 Windows 操作系统，或手机的 Android 操作系统，才能够打开 QQ 开始使用它的聊天功能。

在嵌入式软件的编程中，不会提供像 Windows 或 Android 那样简单易用的操作系统，例如 TI 提供的 ZStack 协议栈，用比较精简抽象的方式提供了一个类似于操作系统的平台，让程序可以在操作系统的简单管理下基于事件机制运行，并具备类似于应用程序一样的初始化、服务、销毁的生命周期。接下来的教程里面会详细地讨论 ZigBee 协议栈架构中每个层所包含的内容和功能及 ZStack 的软件架构。

3.1.2 协议栈的工作原理

CC2530 集成了增强型的 8051 内核，因此在 CC2530 的片上编程实践中，都遵循 51 单片机的编程方式。在这个内核中进行组网通信时，如果用以前基础实验的方法来写程序，逐步查阅芯片手册，了解相关寄存器，按手册进行参数设置等，工作量将非常巨大。TI 公司作为 ZigBee 的生产商，为用户搭建一个小型的操作系统(本质也是大型的程序)，名为 ZStack。ZStack 中已经对底层和网络层的内容进行了封装处理，将复杂部分屏蔽掉。让用户通过 API 函数就可以轻易搭建 ZigBee 网络，并实现无线数据通信。

在 51 单片机的编程方式下，微处理器只能处理一种任务，如 LED 点灯任务，如要使得 LED 灯闪烁，则要利用单片机的定时器资源；如要采用定时器中断方式让 LED 灯闪烁，则还需要利用单片机的中断资源。如果要在让 LED 灯闪烁的同时，处理串口收发，程序逻辑将会更加复杂。在操作系统的管理下，多个任务则可以有条不紊地执行，类似于在 Windows 操作系统中，安装 QQ 软件后，再安装 Office 软件，这些软件可以由 Windows 操作系统分配资源，并发运行。例如 LED 灯闪烁和串口收发这两个功能，在 ZStack 协议栈这个小型操作系统中，可以作为两个任务被管理起来，分配资源有序执行。各个要执行的任务在 ZStack 协议栈中被描述为任务事件队列(*tasksEvents)，如图 3-3 所示。

ZStack 在初始化后，会不断地查询任务事件队列，调用它所管理的定时器、中断等

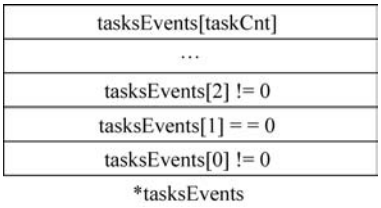


图 3-3　ZStack 协议栈的任务事件队列

资源，周而复始地计时，或根据中断向量调用中断服务程序，使任务队列中的任务得以顺利执行。这个方式称为任务轮询，如图 3-4 所示。

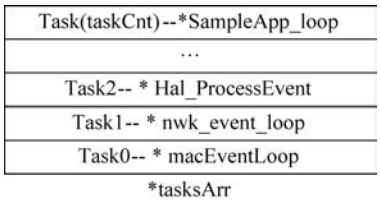


图 3-4　ZStack 的任务轮询

ZStack 调动 CC2530 芯片的定时器、中断等资源的过程已经被封装在协议栈的底层，因此，基于 ZStack 协议栈进行编程无须再考虑 CC2530 芯片的定时器、中断等寄存器配置，只关注如何将要处理的任务加入协议栈任务队列，接受操作系统轮询处理即可，这大大提升了编程效率。

ZStack 协议栈中操作系统的任务队列及轮询管理机制叫作 OSAL（Operating System Abstraction Layer，操作系统抽象层），它支持多任务运行，并不是一个传统意义上的操作系统，但是实现了部分类似操作系统的功能。它模拟 OS（操作系统）的一些方法为广大编程者提供一种编写 MCU 程序的方法。当有一个事件发生时，OSAL 负责将此事件分配给能够处理此事件的任务，然后此任务判断事件的类型，调用相应的事件处理程序进行处理。

打开协议栈文件夹 Texas Instruments\Projects\zstack，里面包含 TI 公司的例程和工具，再打开如图 3-5 所示的 Samples 文件夹。

图 3-5　ZStack 的 Sample 文件夹

Samples 文件夹里面有 3 个工程样例：GenericApp、SampleApp 和 SimpleApp。在此选择 SampleApp 对协议栈的工作流程进行讲解。打开\SampleApp\CC2530DB 下的工程文件 SampleApp.eww，留意左边的工程目录，暂时只需要关注 ZMain 文件夹和 App 文件夹，如图 3-6 所示。

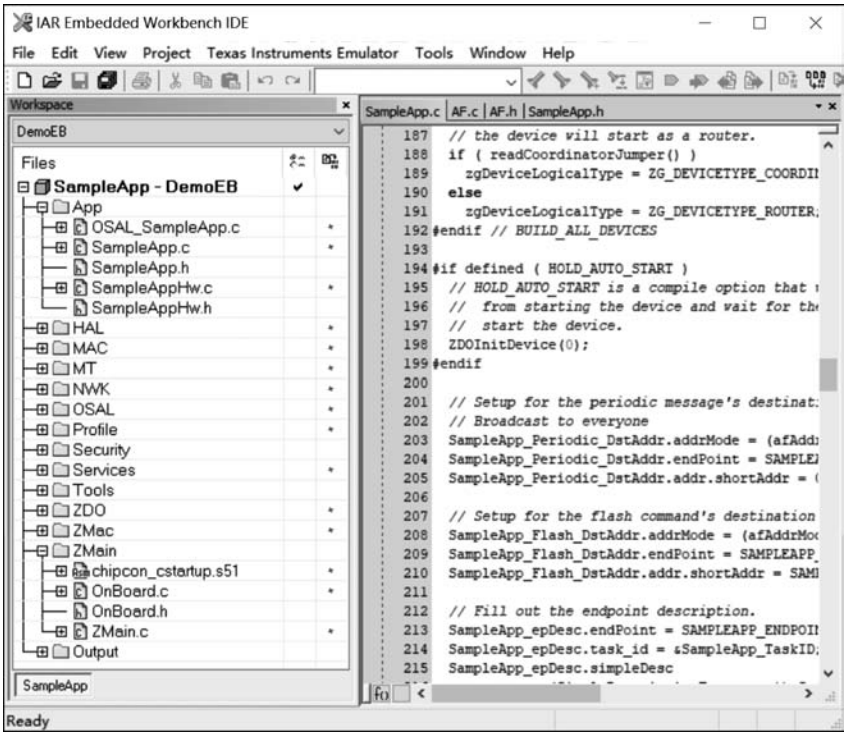


图 3-6　SampleApp 工程文件夹

任何程序都有入口，一般是 main()函数。ZStack 工程的入口在 ZMain.c 文件中，入口函数是 int main(void)函数，代码如下。

```
/*************************************************************************
 * @fn      main
 * @brief  First function called after startup.
 * @returndon't care
 */
int main( void )
{
//Turn off interrupts
osal_int_disable( INTS_ALL );                    //关闭所有中断
//Initialization for board related stuff such as LEDs
HAL_BOARD_INIT();                                //初始化系统时钟
//Make sure supply voltage is high enough to run
                  zmain_vdd_check();             //检查芯片电压是否正常
```

```
//Initialize board I/O
InitBoard( OB_COLD );                    //初始化 I/O,LED、Timer 等
//Initialze HAL drivers
HalDriverInit();                         //初始化芯片各硬件模块
//Initialize NV System
osal_nv_init( NULL );                    //初始化 Flash 存储器
//Initialize the MAC
ZmacInit();                              //初始化 MAC 层
//Determine the extended address
zmain_ext_addr();                        //确定 IEEE64 位地址
//Initialize basic NV items
zgInit();                                //初始化非易失变量
#ifndefNONWK
//Since the AF isn't a task, call it's initialization routine
afInit();
#endif
//Initialize the operating system
osal_init_system();                      //初始化操作系统
//Allow interrupts
osal_int_enable( INTS_ALL );             //使能全部中断
//Final board initialization
InitBoard( OB_READY );                   //初始化按键
//Display information about this device
zmain_dev_info();                        //显示设备信息
/* Display the device info on the LCD */
#ifdef LCD_SUPPORTED
zmain_lcd_init();
#endif
#ifdefWDT_IN_PM1
/* If WDT is used, this is a good place to enable it. */
WatchDogEnable( WDTIMX );
#endif
osal_start_system();                     //No Return from here 执行操作系统,进去后不会返回
return 0;                                //Shouldn't get here.
}
```

在上面的代码中,顺序执行了一系列初始化操作,包括硬件、网络层、任务等的初始化,然后就可以初始化 OSAL 和启动 OSAL 了。初始化操作系统的函数是 osal_init_system(),启动运行操作系统的函数是 osal_start_system(),关于函数的实现细节,在 IAR 编程环境中,可以在函数名上右击→Go to definition of,便可进入函数的定义部分,如图 3-7 所示。

首先,观察 osal_init_system()系统初始化函数,进入函数。发现里面有 6 个初始化函数,在用户层面上编写 OSAL 初始化任务时,只需关注 osalInitTasks()函数,如图 3-8 所示。

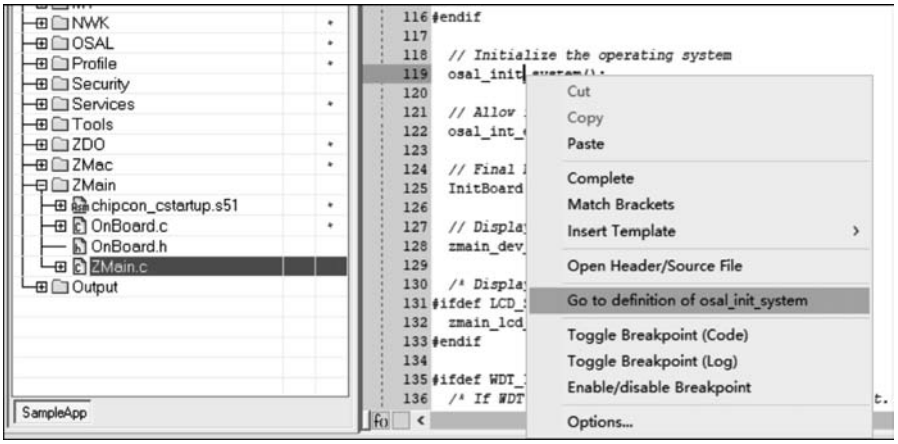
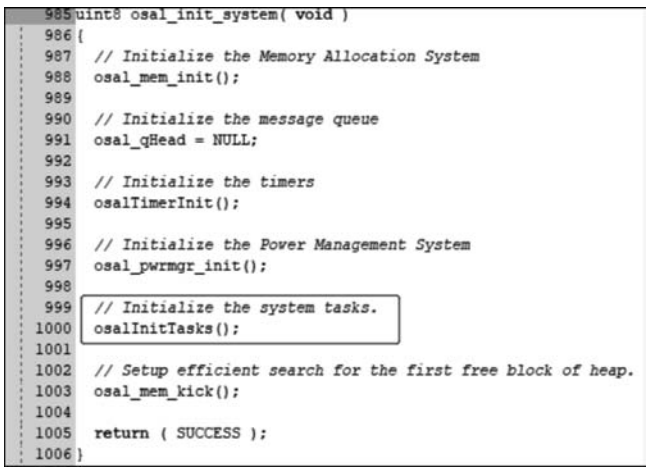


图 3-7 查看函数定义



```
985 uint8 osal_init_system( void )
986 {
987   // Initialize the Memory Allocation System
988   osal_mem_init();
989
990   // Initialize the message queue
991   osal_qHead = NULL;
992
993   // Initialize the timers
994   osalTimerInit();
995
996   // Initialize the Power Management System
997   osal_pwrmgr_init();
998
999   // Initialize the system tasks.
1000  osalInitTasks();
1001
1002  // Setup efficient search for the first free block of heap.
1003  osal_mem_kick();
1004
1005  return ( SUCCESS );
1006 }
```

图 3-8 osalInitTasks()函数的调用

下面继续通过 Go to definition…功能进入 osalInitTasks()函数的定义部分,代码如下。

```
void osalInitTasks( void )
{
    uint8 taskID = 0;
//分配内存,返回指向缓冲区的指针
    tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
//设置所分配的内存空间单元值为 0
    osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));
//任务优先级由高向低依次排列,高优先级对应 taskID 的值反而小
    macTaskInit(taskID++);            //macTaskInit(0),用户不需要考虑
    nwk_init(taskID++);               //nwk_init(1),用户不需要考虑
```

```
    Hal_Init(taskID++);                          //Hal_Init(2),用户需要考虑
    #if defined( MT_TASK )

    MT_TaskInit(taskID ++);
    #endif
    APS_Init(taskID++);                          //APS_Init(3),用户不需要考虑
    #if defined ( ZIGBEE_FRAGMENTATION )
    APSF_Init(taskID ++);
    #endif
    ZDApp_Init(taskID++);                        //ZDApp_Init(4),用户需要考虑
    #if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
    ZDNwkMgr_Init(taskID ++);
    #endif
    SampleApp_Init(taskID);                      //SampleApp_Init _Init(5),用户需要考虑
}
```

关于代码中的 taskID,可以理解为任务队列中的任务唯一编号。例如 LED 灯的闪烁、在网络中数据的收发、在串口中数据的收发,在 OSAL 的管理下,都称为任务。所需执行的任务存在于任务队列中。osal_init_system()中,要对 taskID 进行初始化,每完成一个任务的初始化,执行 taskID++。在 osal_init_system()函数的语句后写了一些注释,有些需要用户考虑,有些不需要用户考虑。这里的用户指的是使用 ZStack 协议栈的编程者。注释为不需要考虑的环节,已经由 ZStack 协议栈封装完成,而用户根据自己的硬件平台进行的其他设置,要在需要考虑的语句中修改完成。

下面分析操作系统启动运行的函数 osal_start_system(),这是任务系统轮询的主要函数。它会查找发生的事件,然后调用相应的事件执行函数。如果没有事件登记要发生,就进入睡眠模式。这个函数是永远不会返回的,用 go to definition 的方法进入该函数,代码如下所示。

```
/**************************************************************************
**
 * @fn     osal_start_system
 * @brief *
 * This function is the main loop function of the task system. It
 * will look through all task events and call the task_event_processor()
 * function for the task with the event. If there are no events (for
 * all tasks), this function puts the processor into Sleep.
 * This Function doesn't return.
 * @param void
 * @return none
 ***************************************************/
void osal_start_system( void )
{
#if !defined ( ZBIT ) && !defined ( UBIT )
for(;;)                                  //Forever Loop
```

```
    #endif
    {
        uint8 idx = 0;
        osalTimeUpdate();          //这里是在扫描哪个事件被触发了,然后置相应的标志位
        Hal_ProcessPoll();         //This replaces MT_SerialPoll() and osal_check_timer().
        Do
        {
            if (tasksEvents[idx]) //Task is highest priority that is ready.
            {
                break;             //得到待处理的最高优先级任务索引号 idx
            }
        } while (++idx < tasksCnt);

        if (idx < tasksCnt)
        {
            uint16 events;
            halIntState_t intState;
            HAL_ENTER_CRITICAL_SECTION(intState);      //进入临界区保护
            events = tasksEvents[idx];                 //提取需要处理的任务中的事件
            tasksEvents[idx] = 0;                      //Clear the Events for this task.清
                                                         除本次任务的事件
            HAL_EXIT_CRITICAL_SECTION(intState);       //退出临界区
            events = (tasksArr[idx])( idx, events );   //通过指针调用任务处理函数
            HAL_ENTER_CRITICAL_SECTION(intState);      //进入临界区
            tasksEvents[idx] |= events;                //Add back unprocessed events to the
                                                         current task.保存未处理的事件
        HAL_EXIT_CRITICAL_SECTION(intState);           //退出临界区
        }
        #if defined( POWER_SAVING )
        else // Complete pass through all task events with no activity?
        {
            osal_pwrmgr_powerconserve(); // Put the processor/system into sleep
        }
        #endif
        }
    }
```

那么,在 OSAL 运行过程中,是如何知道在初始化函数 osal_init_system()中将哪些任务加入了队列,又如何对队列中的任务进行轮询处理呢? 下面关注 osal_run_system()函数中的语句:

```
events = tasksEvents[idx];
```

通过 Go to definition…进入 tasksEvents[idx]数组定义,可以发现定义了该数组之后,紧接着又定义了 osalInitTasks(void)函数,如图 3-9 所示。

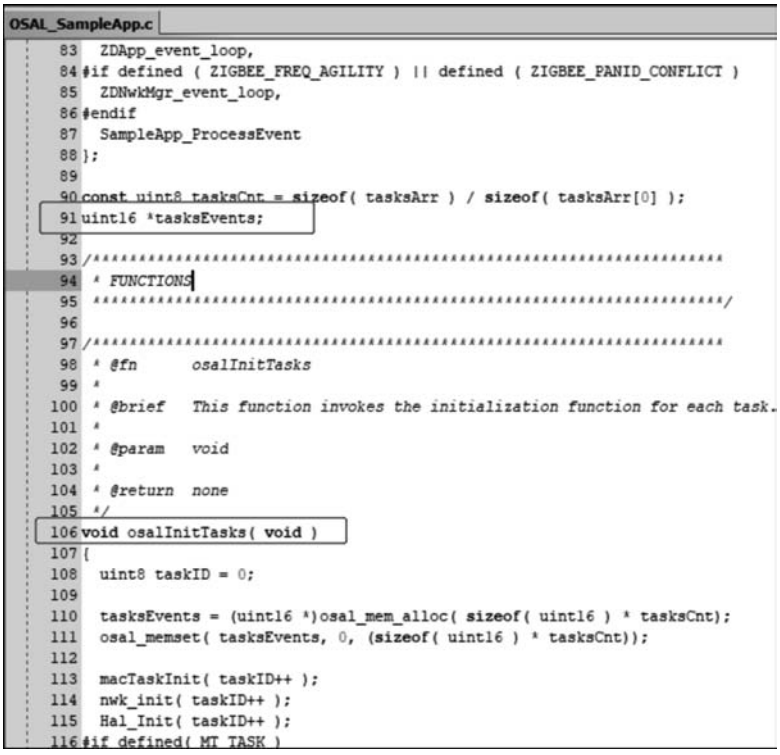


图 3-9　osalInitTasks()函数的定义

osalInitTasks()函数的代码如下。

```
void osalInitTasks( void )
{
  uint8 taskID = 0;

  tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
  osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));

  macTaskInit( taskID++);
  nwk_init( taskID++);
  Hal_Init( taskID++);
#if defined( MT_TASK )
  MT_TaskInit( taskID++);
#endif
  APS_Init( taskID++);
#if defined ( ZIGBEE_FRAGMENTATION )
  APSF_Init( taskID++);
#endif
  ZDApp_Init( taskID++);
#if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
```

```
  ZDNwkMgr_Init( taskID++);
#endif
  SampleApp_Init( taskID );
}
```

观察 osalInitTasks()函数结构，可以发现，函数对任务进行了判断，如果是网络层、硬件层、网络设备层的任务，则依次执行，并 taskID++，继续轮询任务队列。当这些必备任务都处理后，处理应用层（位于 ZStack 协议栈 APP 层）的任务。例如，样例工程 SampleApp，对应的任务就是 SampleApp 初始化任务。至此，通过 taskID 将 ZStack 协议栈的 OSAL 启动过程与 SampleApp 进行了关联。相当于 SampleApp 应用程序被安装到操作系统 OSAL 中，分配了 taskID，即任务号。当 OSAL 初始化并启动后，轮询发现应用程序 SampleApp 已经被安装好，则处理其任务请求。通过类似的原理，在下面的组网、广播、组播和点播实践中，均以 SampleApp 为基础程序框架，加以修改后完成组网、广播、组播和点播等应用任务。

3.2 基于 ZStack 协议栈组建无线网络

3.2.1 ZStack 的 SampleApp 应用分析

通过对 ZStack 协议栈工作原理的学习，我们知道 ZStack 协议栈是一个基于任务轮询方式的操作系统，其任务调度和资源分配由操作系统抽象层 OSAL 管理。

可以理解为：ZStack 协议栈 = OSAL 操作系统 + CC2530 硬件模块 + AF 无线网络应用。总体来看，ZStack 协议栈只做了两件事情：首先进行系统的初始化，然后启动 OSAL 操作系统。在任务轮询过程中，系统将会不断查询每个任务是否有事件发生，如果有事件发生，就执行相应的事件处理函数，如果没有事件发生，则查询下一个任务。

深入理解 OSAL 的调度机制和工作机理，是灵活应用 ZStack 协议栈进行 ZigBee 无线应用开发的重要基础。深入地理解 OSAL 操作系统的关键是要理解任务初始化函数 osalInitTasks()、任务标识符 taskID、任务事件数组 taskEvents[]和任务事件处理函数指针数组 tasksArr[]之间的对应关系以及它们在 OSAL 运行过程中的执行情况。

下面通过 ZStack 协议栈的自带例程 SampleApp，学习 ZStack 的编程原理。读者可以从 ZStack 的安装目录下获取 SampleApp 工程代码：C:\Texas Instruments\ZStack-CC2530-2.5.1a\Projects\zstack\Samples\SampleApp\CC2530DB。用 IAR 打开文件夹中的 SampleApp.eww 工程文件，其目录树结构如图 3-10 所示。

在样例工程 SampleApp 的工程目录树中，找到 Profile 目录下的 AF.c 文件。在

图 3-10　SampleApp 目录树结构

这个文件中有一个重要的函数：afStatus_t afRegister(endPointDesc_t * epDesc)。我们可以通过 ZStack API. pdf 中查询到 afRegister 的函数说明。该函数的作用是把一个设备注册成一个 ZigBee 网络中的新节点，并且不允许重复注册，以保证网络中节点的唯一性。

下面对该函数的参数加以分析。

该函数的参数 endPointDesc_t 是一个结构体，用于描述节点信息，如端点的任务序列号、网络延时请求、端点简单描述信息。其代码如下所示。

```
typedef struct
{
  uint8 endPoint;
  uint8 * task_id;                  //Pointer to location of the Application task ID.
  SimpleDescriptionFormat_t * simpleDesc;
  afNetworkLatencyReq_t latencyReq;
} endPointDesc_t;
```

其中，端点简单描述信息用结构体 SimpleDescriptionFormat_t 进行结构定义，代码如下所示。

```
typedef struct
{
  uint8         EndPoint;
  uint16        AppProfId;
  uint16        7 AppDeviceId;
  uint8         AppDevVer:4;
  uint8         Reserved:4;           //AF_V1_SUPPORT uses for AppFlags:4.
  uint8         AppNumInClusters;
  cId_t         * pAppInClusterList;
  uint8         AppNumOutClusters;
  cId_t         * pAppOutClusterList;
} SimpleDescriptionFormat_t;
```

- EndPoint　端点号可以设置1～240之间的整数(0被系统保留,不能使用)端点号定义的是该节点设备的子地址,用于接收数据。
- AppProfId　该字段定义应用描述ID,称为应用描述符。应用描述符是一组统一的消息,消息格式和处理方法,允许开发者建立一个可以共同使用的分布式应用程序,这些应用描述符是利用驻扎在独立设备中的应用实体来实现的。这些应用允许应用程序发送命令、请求数据和处理命令的请求。ZigBee联盟定义了一系列应用描述符,字段值从ZigBee联盟获取。
- AppDeviceId　该字段成为应用设备ID,每一个ZigBee节点对应唯一的应用设备ID,也由ZigBee联盟进行了预定义。
- AppDevVer　应用设备版本号,由十六进制数表示。例如0x00,指的是Version 1.0。
- Reserved　在编程中一般不涉及。
- AppNumInClusters　簇标识符可用来区分不同的簇,簇标识符关联着从设备流出和向设备流入的数据。在特殊的应用profiles范围内,簇标识符是唯一的。
- pAppInClusterList　节点的输入数据流的簇ID列表。
- AppNumOutClusters　8位,表示应用的输出数据流的簇。
- pAppOutClusterList　节点的输出数据流的簇ID列表。

下面分析afRegister()函数的返回值类型。其返回值类型afStatus_t也是一个结构体类型,如果组网成功,则返回ZSuccess;如果组网失败,则返回Error。所以,在后续的基于ZStack的组网实验中,将根据这个返回值,判断组网状态。

afRegister()函数由ZStack提供,在编写应用程序时,调用该函数,即可完成设备在ZigBee网络中的注册。

下面来分析SampleApp工程中的应用层。读者可以在工程目录树的App文件夹下打开SampleApp.c,后续的大部分编程都集中在应用层,即App文件夹下。

SampleApp.c中有以下3个全局变量。

- devStates_t SampleApp_NwkState;
- uint8 SampleApp_TransID;　// This is the unique message ID (counter)
- afAddrType_t SampleApp_Periodic_DstAddr;

分别用于表示网络状态、任务ID号和应用周期性发送数据时的目的地址。

这3个全局变量在void SampleApp_Init(uint8 task_id)和uint16 SampleApp_ProcessEvent(uint8 task_id, uint16 events)这两个函数中均有涉及。

SampleApp_Init()函数和SampleApp_ProcessEvent()函数是基于ZStack的网络编程中涉及最多的函数,程序中的组网逻辑和网络数据传输逻辑都要对这两个函数进行修改。

在SampleApp_Init()函数中,进行网络注册,即调用afRegister()函数,并根据afRegister()函数的返回值,判断组网结果和设备在网络中的状态(协调器、路由器,还是终端节点)。

SampleApp_ProcessEvent()函数主要用于ZStack网络中的事件处理。在基于51

单片机的编程中，程序 2-4 是查询式的按键检测，CC2530 的 main()函数只能在死循环中不断查询检测按键状态。程序 2-5 是外部中断式的按键检测，当有按键按下时，触发中断服务处理子程序，响应按键按下的状态，控制 LED 灯亮灭。因此，主程序可以完成 LED 灯控制等其他逻辑，外部按键按下时，从主程序跳转到中断服务程序执行中断响应。这些程序均利用 CC2530 的芯片资源完成，无法基于事件处理更为复杂的情况；无法综合处理无线网络的组建和感知、控制数据的发送和接收。本章的编程基于 ZStack 协议栈来完成，协议栈编程者提供了一个简单的操作系统抽象层，即 OSAL。SampleApp_ProcessEvent 则是 OSAL 提供的事件处理机制。在事件处理机制中，按键按下、网络中有消息发来等，都作为事件处理。当事件发生时，才调用事件处理程序；如果没有事件发生，则执行 OSAL 中的主逻辑，即保持网络的组网状态。

3.2.2　组建无线网络

本章的例程主要集中于物联网网络层的编程实践，基于 ZStack 协议栈，ZStack 的帮助文档在 ZStack 安装目录下：C:\Texas Instruments\ZStack-CC2530-2.5.1a\Documents；文件名为 ZStack API.pdf。请读者在学习过程中参照帮助文档理解相关知识点，掌握例程原理。

下面介绍如何在 CC2530 的硬件模块的 OSAL 操作系统中，进行任务事件的添加和无线网络组建。

【实验 3-1】 基于 ZStack 协议栈组建无线网络。

实验目的：CC2530 模块 OSAL 任务事件和 AF 无线网络应用。

软硬件环境：ZigBee 开发板(两块)、仿真器、IAR 集成开发环境。

本实验基于前面讲解的案例：SampleApp 改写，完成无线网络的组建，一个 ZigBee 开发板作为协调器，另一个开发板作为终端节点，两个节点上电后自组 ZigBee 网络。本章例程涉及的文件较多，因此在讲解中只给出有改动的代码或核心代码，完整工程代码见本书配套电子资源。

(1) 在本书配套电子资源中，打开 S2 工程，编写文件 s2.c 和 s2.h，并将 s2.c 和 s2.h 加入工程 App 文件夹中。

(2) 仿照 SampleApp.c，在 s2App.c 中定义自己的 3 个全局变量。

```
devStates_t S2App_NwkState;
uint8 S2App_TransID;
afAddrType_t S2App_Periodic_DstAddr;          //周期性发送消息的目标地址
```

在 S2App_Init()函数中，对 3 个变量进行赋值。

```
S2App_NwkState = DEV_INIT;
S2App_TransID = 0;
S2App_Periodic_DstAddr.addrMode = (afAddrMode_t)Addr16Bit;
```

```
  S2App_Periodic_DstAddr.endPoint = S2APP_ENDPOINT;      //在 s2App.h 中定义宏:S2APP
                                                           _ENDPOINT
  S2App_Periodic_DstAddr.addr.shortAddr = 0x0;
```

(3) 在 s2App.h 中定义宏。

```
#define S2APP_ENDPOINT 30
```

(4) 端点描述符和简单描述符的定义和赋值。

在 SampleApp_Init()继续添加如下代码。

```
//端点描述符
  S2App_epDesc.endPoint = S2APP_ENDPOINT;        //在 s2App.h 中定义过的宏
  S2App_epDesc.task_id = &S2AppTaskID;           //S2AppTaskID:s2App.c 中的全局变量
  S2App_epDesc.latencyReq = noLatencyReqs;       //参照 SampleApp.c,不用改动
  //simpleDesc 简单描述符
  S2App_epDesc.simpleDesc
            = (SimpleDescriptionFormat_t *)&S2App_SimpleDesc;
```

(5) 端点描述符和简单描述符的全局变量声明。

在 s2App.c 中声明全局变量。

```
endPointDesc_t S2App_epDesc;
```

在 s2App.c 中声明全局结构体变量 S2App_SimpleDesc 并赋值。

```
SimpleDescriptionFormat_t S2APP_SimpleDesc =
{
  S2APP_ENDPOINT,
  S2APP_PROFILE_ID,
  S2APP_DEVICE_ID,
  S2APP_DEVICE_VER,
  0,
  S2APP_CLUSTER_NUM,
  (cId_t *)S2APP_ClusterIDS,
  S2APP_CLUSTER_NUM,
  (cId_t *)S2APP_ClusterIDS
};
```

(6) 在 s2App.h 中添加宏定义。

```
#define S2APP_PROFILE_ID 0xF09
#define S2APP_DEVICE_ID 0x0
#define S2APP_DEVICE_VER 0x0
#define S2APP_CLUSTER_NUM 2
```

在 s2App.c 中定义全局变量数组 S2APP_CLUSTER_IDS 并赋值。

```
cId_t S2APP_CLUSTER_IDS[S2APP_CLUSTER_NUM] =
{
  S2APP_LEDOFF_CLUSTER_ID,
  S2APP_LEDON_CLUSTER_ID

};
```

在 s2App.h 中定义 Cluster 的宏。

```
#define S2APP_LEDOFF_CLUSTER_ID 0x0
#define S2APP_LEDON_CLUSTER_ID 0x1
```

(7) 注册端点。

在 s2App.c 中,找到初始化函数 SampleApp_Init(),在该函数内调用 ZStack 协议栈提供的 afRegister()函数,注册端点。

```
afRegister( &SampleApp_epDesc );     //application framework 注册 ZStack API.pdf 48 页
```

(8) 网络事件处理。

在 s2App.c 的 uint16 SampleApp_ProcessEvent(uint8 task_id, uint16 events)函数中,修改 switch 分支结构,加入代码中圈释部分的语句。

```
uint16 S2App_ProcessEvent( uint8 task_id, uint16 events ){
  afIncomingMSGPacket_t *MSGpkt;
  (void)task_id;      // Intentionally unreferenced parameter 先使用 task_id,避免出现警
                         告错误

  if ( events & SYS_EVENT_MSG )
  {
    MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( S2AppTaskID );
    while ( MSGpkt )
    {
      switch ( MSGpkt->hdr.event )
      {/*加入网络事件处理判断分支*/
      case ZDO_STATE_CHANGE:
        S2App_NwkState = (devStates_t)(MSGpkt->hdr.status);
        if(S2App_NwkState == DEV_ZB_COORD){          //协调器被建立
          HalUARTWrite(0,"COOR",4);
        }
        else if(S2App_NwkState == DEV_ROUTER){       //路由器
          HalUARTWrite(0,"ROU",3);
        }
        else if(S2App_NwkState == DEV_END_DEVICE){   //终端节点
          HalUARTWrite(0,"END",3);
        }
```

```
        else{                          //还没有建立网络
          HalUARTWrite(0,"ERR",3);
        }
        break; /* 网络事件处理分支结束 */
        default:
          break;
      }
```

(9) 调试运行。

① 在工程名上右击,在弹出的快捷菜单中选择 Options 选项,进入工程参数设置对话框,如图 3-11 所示; 在 C/C++Compiler 选项中,选取 Preprocessor 选项卡,在 Defined symbols 设置中填写 ZTOOL_P1,如有其他默认设置,在此处全部删掉。

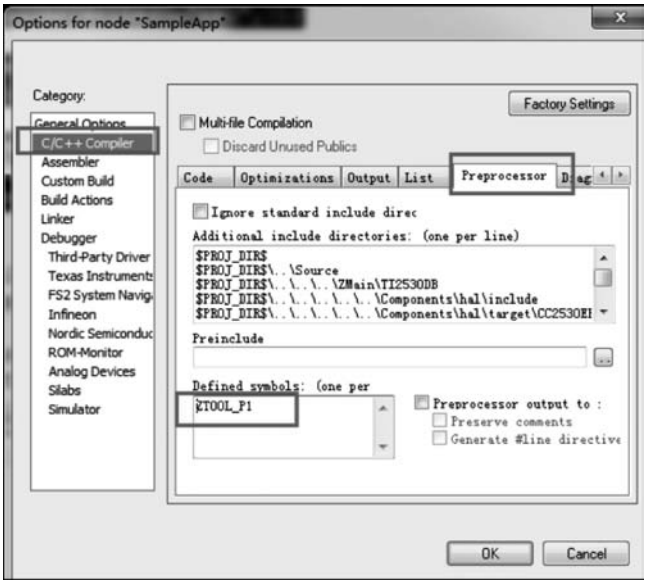


图 3-11 组网工程的调试运行设置

② 删掉工程目录中的 SampleApp.c 和 SampleApp.h 文件。

③ 在 OSAL_SampleApp.c 中,删掉关于 SampleApp 的 TaskID 和 events 数组中的内容,添加本节新建例程 S2App 的任务事件定义和初始化,如图 3-12 和图 3-13 所示。

```
126  ZDNwkMgr_Init( taskID++ );
127 #endif
128  //SampleApp_Init( taskID++ );
129  S2App_Init(taskID);
130 }
```

图 3-12 S2App 的初始化

④ 每一个无线网络由 PANID 进行唯一标识,为避免多组学生进行组网实验时

```
84   ZDApp_event_loop,
85 #if defined ( ZIGBEE_FREQ_AGILITY ) || d
86   ZDNwkMgr_event_loop,
87 #endif
88   S2App_ProcessEvent
89 };
```

图 3-13　S2App 的事件处理

PANID 冲突，建议修改 PANID，如用四位学号等信息加以区分，如图 3-14 所示。

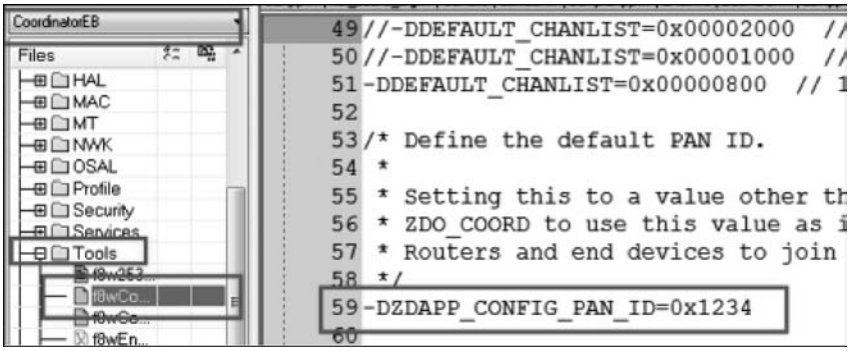


图 3-14　修改 PANID

⑤ 分别编译为协调器和终端节点，并烧写到两个 ZigBee 开发板上，运行，通过串口查看：协调器运行时串口输出字符串 COOR，终端节点运行时串口输出字符串 END。

注意：

本章中的代码通常涉及多个文件，还有一些数据结构的宏定义位于 ZStack 协议栈中，因此，编写代码时，注意逐步编写调试，导入相应的头文件。导入头文件时可在 IAR 中在相关变量代码处右击，采用 goto definition 的方式，查看变量在哪个头文件中，并导入，如：

```
#include "AF.h"   #include "ZDApp.h"
```

编写过程中，一些较长的变量名、宏名称等，尽量复制粘贴，避免低级错误。运行终端节点时，协调器也要运行，以保证终端节点能加入协调器组建的网络中协调器节点上电即可。

3.3　基于 ZStack 协议栈的无线通信

ZigBee 的通信方式主要有点播、组播和广播三种。点播，顾名思义就是点对点通信，也就是两个设备之间的通信，不允许有第三个设备收到信息；组播，就是把网络中的节点分组，每一个组员发出的信息只有相同组号的组员才能收到；广播，就是一个设备上发出的信息所有设备都能接收到，这也是 ZigBee 通信的基本方式。

3.3.1 点播(点对点通信)

点播描述的是网络中两个节点相互通信的过程。确定通信对象的就是节点的16bit 短地址。下面在 SampleApp 例程完通过简单的修改完成点播实验。

【实验 3-2】 ZigBee 网络中的点播。

为了简化大家理解。数据发送和接收的内容按照 3.2.2 小节的基本例程 SampleApp 逐步修改,完成点播功能。

打开 SampleApp 的 Profile 文件夹下的 AF.h 文件,找到如下代码。

```
typedef enum
{
    afAddrNotPresent = AddrNotPresent,
    afAddr16bit      = Addr16bit,
    afAddr64bit      = Addr64bit,
    afAddrGroup      = AddrGroup,
    afAddrBroadcast  = AddrBroadcast
} afAddrMode_t;
```

该类型是一个枚举类型:

当 addrMode= Addr16bit 时,对应点播方式;

当 addrMode= AddrGroup 时,对应组播方式;

当 addrMode= AddrBroadcast 时,对应广播方式。

按照以往的步骤,打开 SampleApp.c 文件可发现已经存在如下代码。

```
afAddrType_t SampleApp_Periodic_DstAddr;
afAddrType_t SampleApp_Flash_DstAddr;
```

这分别是组播和广播的定义。按照组播和广播的定义格式来添加点播定义如下。

```
afAddrType_t  Point_To_Point_DstAddr;        //点对点通信定义
```

对 Point_To_Point_DstAddr 的相关参数进行配置,找到下面的位置,参考 SampleApp_Periodic_DstAddr 和 SampleApp_Flash_DstAddr 进行点播的配置,加入如下代码。

```
// 点对点通信定义
Point_To_Point_DstAddr.addrMode = (afAddrMode_t)Addr16bit;        //点播
Point_To_Point_DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
Point_To_Point_DstAddr.addr.shortAddr = 0x0000;                   //发给协调器
```

第三行代码指明了点播的发送对象的 16 位短地址是 0x0000,也就是协调器的地

址。由此可知，本实验任务是节点和协调器之间的点对点通信。

继续添加自定义的点对点发送函数，在 SampleAPP.c 最后加入下面代码。

```
void SampleApp_SendPointToPointMessage( void )
{
  uint8 data[10] = {0,1,2,3,4,5,6,7,8,9};
  if ( AF_DataRequest( &Point_To_Point_DstAddr,
                       &SampleApp_epDesc,
                       SAMPLEAPP_POINT_TO_POINT_CLUSTERID,
                       10,
                       data,
                       &SampleApp_TransID,
                       AF_DISCV_ROUTE,
                       AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
else
  {
    // Error occurred in request to send.
  }
}
```

还需要在 SampleAPP.c 文件开头添加头函数声明。

```
void SampleApp_SendPointToPointMessage( void );
```

其中，Point_To_Point_DstAddr 之前已经定义，在 SampleApp.h 中加入 SAMPLEAPP_POINT_TO_POINT_CLUSTERID 的定义，代码如下。

```
#define SAMPLEAPP_POINT_TO_POINT_CLUSTERID  3              //传输编号
```

接下来，为了测试程序，需要把 SampleApp.c 文件中的 SampleApp_SendPeriodicMessage() 函数替换成刚刚建立的点对点发送函数 SampleApp_SendPointToPointMessage()，这样就能实现周期性点播发送数据了。

在接收方面，进行如下修改：接收 ID 在改成刚定义的 SAMPLEAPP_POINT_TO_POINT_CLUSTERID。由于协调器不允许给自己点播，故周期性点播初始化时，协调器不能初始化。

实验结果：将修改后的程序分别以协调器、路由器、终端的方式下载到 3 个节点设备中，连接串口。可以看到只有协调器在一个周期内收到信息。也就是说，路由器和终端均与地址为 0x00(协调器)的设备通信，不与其他设备通信。实现点对点传输，如图 3-15 所示。

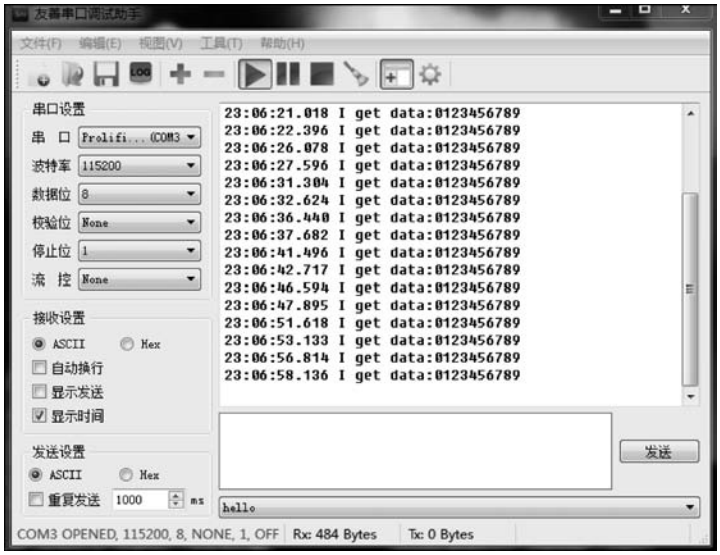


图 3-15 点播运行结果

3.3.2 组播

组播描述的是网络中所有节点设备被分组后组内相互通信的过程。确定通信对象的是节点的组号。下面在 SampleApp 例程完通过简单的修改完成组播实验。数据发送和接收的内容依然按照 3.2.1 小节的格式。修改流程与点播相似。

【实验 3-3】 ZigBee 网络中的组播。

(1) 关注 SampleApp.c 中两项内容。

• 组播 afAddrType_t 的类型变量

```
afAddrType_t SampleApp_Flash_DstAddr;        //组播
```

• 组播内容的结构体

```
aps_Group_t SampleApp_Group;                 //分组内容
```

(2) 组播参数的配置。

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

(3) 已经定义的组信息代码,将 ID 修改成组号相对应,方便以后扩展分组需要。

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

在 SampleApp.h 里面可以看到组号为 0x0001。

```
// Group ID for Flash Command
#define SAMPLEAPP_FLASH_GROUP       0x0001
```

(4) 在 SampleAPP.c 最后面添加自己的组播发送函数,代码如下。

```
void SampleApp_SendGroupMessage( void )
{
  uint8 data[10] = {'0','1','2','3','4','5','6','7','8','9'};     //自定义数据
if ( AF_DataRequest( & SampleApp_Flash_DstAddr,
                        &SampleApp_epDesc,
                        SAMPLEAPP_FLASH_CLUSTERID,
                        10,
                        data,
                        &SampleApp_TransID,
                        AF_DISCV_ROUTE,
                        AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )

  else
  {
    // Error occurred in request to send.
  }
}
```

(5) 添加函数后要在 SampleApp.c 函数声明里加入如下代码。

```
void SampleApp_SendGroupMessage(void);     //组播通信发送函数定义. 否则编译将报错
```

SAMPLEAPP_FLASH_CLUSTERID 的定义如下所示。

```
#define SAMPLEAPP_FLASH_CLUSTERID    2
```

(6) 为了测试程序,把 SampleApp.c 文件中的 SampleApp_SendPeriodicMessage()函数替换成刚刚建立的组播发送函数 SampleApp_SendGroupMessage(),这样就能实现周期性组播发送数据了。

(7) 在接收方面,进行如下修改:组播接收函数改成我们自己来获取数据。

```
case SAMPLEAPP_FLASH_CLUSTERID:
HalUARTWrite(0,"RouterDeviceReceived!",21);              //用于提示有数据
```

```
HalUARTWrite(0, &pkt->cmd.Data[0],10);                    //打印收到数据
HalUARTWrite(0,"\n",1);                                   //回车换行,便于观察
```

实验结果：将修改后的程序分别以一个协调器、两个路由器的方式下载到 3 个设备，把协调器和路由器组号 1 设置成 0x0001，路由器设备 2 组号设成 0x0002，如图 3-16 所示。连接串口，可以观察到只有 0x0001 的两个设备相互发送信息。（注意：终端设备不参与组播信息收发）。

```
// Application Events (OSAL) - These are bit weighted definitions.
#define SAMPLEAPP_SEND_PERIODIC_MSG_EVT       0x0001

// Group ID for Flash Command
#define SAMPLEAPP_FLASH_GROUP                 0x0002  //0x0001

// Flash Command Duration - in milliseconds
#define SAMPLEAPP_FLASH_DURATION              1000

/*********************************************************************
```

图 3-16　组播路由器组号设置

知识扩展：

终端设备不参与组播信息收发，原因是 SampleApp 例程中终端设备默认采用睡眠中断的工作方式，射频不是一直工作，假如下载组播例程到终端，会发现不能正常接收组播信息。如果确实需要使用终端设备参与组播，则可以参考以下方法：

在 ZigBee 协议规范中规定，睡眠中断不接收组播信息，如果一定想要接收组播信息，只有将终端的接收机一直打开，这样就可以接收了。具体做法为：将 f8config.cfg 配置文件中的-RFD_RCVC_ALWAYS_ON＝FALSE 改为-RFD_RCVC_ALWAYS_ON＝TRUE 就可以了。

每隔 5s，串口会显示两个字符串“RouterDeviceReceived!”，同时开发板 A 和开发板 B 的 LED 每隔 5s 点亮一次，开发板 C 的 LED 灯始终处于熄灭状态。实验测试结果如图 3-17 所示。

图 3-17　组播运行结果

3.3.3　广播

广播就是任何一个节点设备发出广播数据，网络中的任何设备都能接收到。有了点播和组播的实验基础，广播的实验进行起来就得心应手了。广播的定义都是协议栈预先定义好的。因此，直接运用即可。

【实验 3-4】 ZigBee 网络中的广播。

(1) 在协议栈 SampleApp 中找到广播参数的配置，代码如下。

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

0xFFFF 是广播地址。协议栈广播地址主要有 3 种类型，具体的定义如下。

- 0xFFFF　数据包将被传送到网络上的所有设备，包括睡眠中的设备。对于睡眠中的设备，数据包将被保留在其父亲节点直到查询到它，或者消息超时。
- 0xFFFD　数据包将被传送到网络上的所有在空闲时打开接收的设备(RXONWHENIDLE)，也就是说，除了睡眠中的所有设备。
- 0xFFFC　数据包发送给所有的路由器，包括协调器。

(2) 在广播实验中，使用默认的 0xFFFF。

在 SampleApp.c 中找到自带的周期性发送函数，修改代码如下。

```
void SampleApp_SendPeriodicMessage( void )
{
uint8 data[10] = {'0','1','2','3','4','5','6','7','8','9'};              //自定义
//数据
if ( AF_DataRequest( &SampleApp_Periodic_DstAddr,
&SampleApp_epDesc,
                        SAMPLEAPP_PERIODIC_CLUSTERID,
                        10,
                        data,
                        &SampleApp_TransID,
                        AF_DISCV_ROUTE,
                        AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
}
  else
  {
    // Error occurred in request to send.
  }
}
```

(3) 在 SampleApp.h 中增加广播传输编号的宏定义如下。

```
#define SAMPLEAPP_PERIODIC_CLUSTERID   1          //广播传输编号
```

(4) 测试程序,按照原来代码保留函数 SampleApp_SendGroupMessage(),这样就能实现周期性广播发送数据了。

在接收方面,在 SampleApp_MessageMSGCB()函数中,默认接收 ID 就是刚定义的周期性广播发送 ID。

```
case SAMPLEAPP_PERIODIC_CLUSTERID:
```

因此,当接收到广播消息后,会调用 SampleApp_MessageMSGCB(),即消息回调函数,在消息回调函数中进入此 case 分支,在该 case 分支中可以编写接收到广播消息后的响应程序。例如,将接收到的广播消息发送至串口,在上位机串口调试助手中显示。

实验结果:将修改后的程序分别以协调器、路由器、终端的方式下载到 3 个设备,可以看到各个设备都在广播发送信息,同时也接收广播信息,如图 3-18 所示。

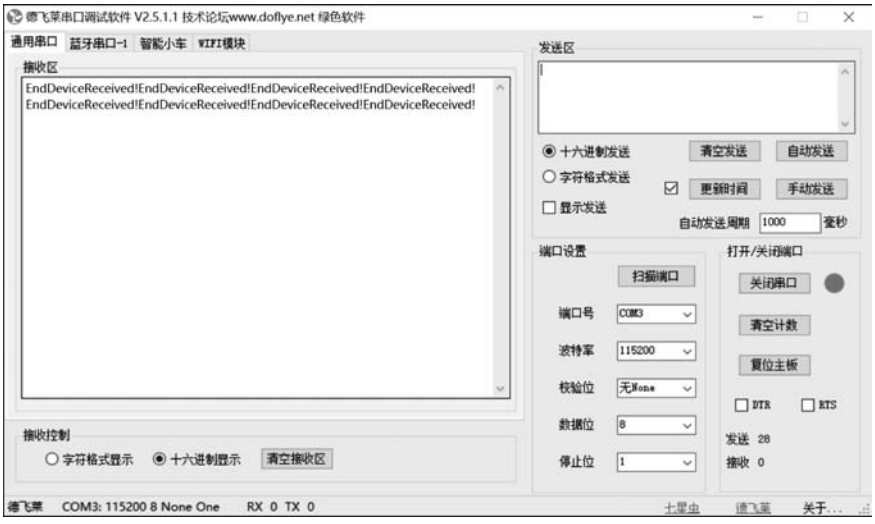


图 3-18 广播运行结果

3.4 RFID 刷卡及无线传输

射频识别(Radio Frequency Identification,RFID),其原理为阅读器与标签之间进行非接触式的数据通信,达到识别目标的目的。RFID 的应用非常广泛,典型应用有动物晶片、汽车晶片防盗器、门禁管制、停车场管制、生产线自动化、物料管理。

完整的 RFID 系统由读卡器(Reader)、电子标签(Tag)和数据管理系统三部分组成。

本节例程基于 3.2.2 小节中组建好的 ZigBee 网络。网络中包含一个终端节点和一个协调器。读卡器通过串口透传的方式与终端节点相连,读卡信息交给终端节点,终端节点通过无线传输的方式将卡号发送给协调器。协调器将收到的卡片信息通过串口发送给上位机,上位机在串口助手中查看卡号。

本节例程所使用的读卡器型号为 MD9291 读写模块,其外形如图 3-19 所示,其使用手册详见本书电子资源。

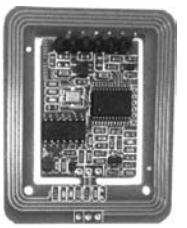


图 3-19　读卡器 MD9291 读写模块外观

本节例程采用的电子标签,即卡片,制式为 1443A 类型,其卡片信息可有配套的读卡器读取。后续的数据处理过程,即对卡片信息数据的处理,本节例程中由串口调试助手的卡号数据显示替代。在学习了物联网应用层的服务器、移动端开发后,可以处理卡号,如验证卡号登录、通过卡号计费等。

作为 RFID 读卡器的电子标签,即卡片,是没有电池的。在读卡器中的天线会向外辐射电磁波,如果卡接近电磁波磁场范围内,就会产生 RC 振荡,如果和 RC 电路的发射频率相同,卡的 RC 电路上就会产生电流,电流存储在电容中。在卡和读卡器通信时,电容就作为卡内芯片的电源。

在读卡器工作过程中,会以查询的方式不断地发送寻卡命令。当卡片靠近读卡器时,卡内芯片工作后会收到寻卡命令,给读卡器一个回应,该回应是卡内预存的卡号。由于可能会有多个卡片同时靠近读卡器,读卡器收到卡号后,还要进行防碰撞和选卡工作。在防碰撞和选卡之后,就可以对卡片内的存储控件进行读写操作了。在对卡片存储区进行读写操作之前,还要给卡片发送暂停指令,避免卡片不断地给读卡器回应。

如果在编程过程中逐一实现上述的寻卡、防碰撞、选卡、暂停等读卡器动作,程序将非常冗长。通过查阅读卡器使用手册可知,在编程过程中,只要使用卡片激活命令,即可对上述过程进行类似于批处理的整体操作。

通过手册中可知,读卡器发送如表 3-2 所示的数据帧,即可执行卡片激活命令,开始读卡动作。

表 3-2　主机发送数据帧

SOF	Length	ADD	CMD	Data	BCC
86	06	00	2C	26 00	8A

当有卡片靠近读卡器时,会显示模块正确返回数据帧,如表 3-3 所示。

表 3-3　模块正确返回数据帧

SOF	Length		ADD	CMD	Data	BCC
86	0D		00	2C	04 00 08 00 04 4C 51 E4 7B	2D

其中，4C 51 E4 7B 即为 RFID 卡号。

当没有卡片靠近时，会收到模块错误返回数据帧，如表 3-4 所示。

表 3-4 模块错误返回数据帧

SOF	Length	ADD	$\overline{\text{CMD}}$	Data	BCC
86	05	00	D3	83	D3

通过对命令格式的分析，我们可知，数据帧的帧头是 0x86，数据帧的内容都是 ASCII 码形式。无论是否有卡片靠近，执行卡片激活命令后，都会返回结果数据帧（正确或错误）。下面的例程中，将采用不断查询的方式进行读卡，没有卡靠近读卡器时，一直显示模块错误返回的十六进制字符信息。

【实验 3-5】 RFID 刷卡及无线传输。

实验目的：完成 RFID 功能模块的单例测试，终端节点从串口收到 RFID 读卡器读到的卡号后，通过 ZigBee 网络向协调器发送；协调器收到后，通过串口发送给 PC 机，由串口调试助手观察结果。

软硬件环境：pl2303USB 转串口模块；读卡器模块，RFID 卡片；杜邦线。一端插在读卡器模块的 wakeup 引脚上，一端接 3.3V 电源。两块 ZigBee 开发板，一个作为终端节点，接读卡器，一个作为协调器，接 pl2303，如图 3-20 和图 3-21 所示。

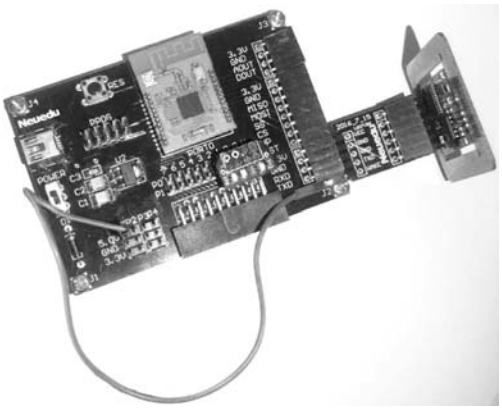


图 3-20 终端节点＋读卡器

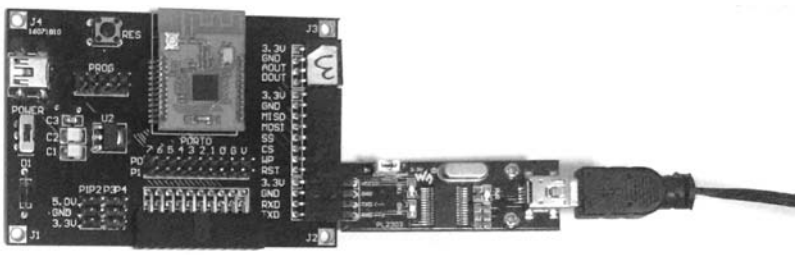


图 3-21 协调器节点＋串口 PL2303

软件：before 文件夹中：空的工程：s2.rar

后续改进：协调器解析收到字符串中的卡号信息(4 字节，参考手册)，并将卡号信息向云平台或网关等设备发送。

主要编程思路是终端节点接收串口传来的 RFID 数据，并向协调器发送，串口传来 RFID 数据，会调用串口回调函数，在串口回调函数中向协调器发送数据。协调器只负责接收数据并在串口打印输出查看。

(1) 实验前软件准备

修改工程 panid：打开工程目录树中的 Tools 文件夹，找到文件 f8wConfig.cfg，修改自定义的 PANID，以免多组实验时造成网络冲突。

```
-DZDAPP_CONFIG_PAN_ID=0x6665
```

修改波特率：由于读卡器的波特率是 9600b/s，因此，打开工程目录树中的 ZMain 文件夹下的 OnBoard.c，修改波特率。

```
uartConfig.baudRate             = HAL_UART_BR_9600;
```

(2) 编写程序。

① 终端节点从串口接收 RFID 数据并发送。

a) 该例程分终端节点和协调器两部分编写。终端节点要通过读卡器接收卡号，周期性向协调器发送数据。因此，首先找到终端节点事件的起始代码，位于 S2App_ProcessEvent 事件处理函数中。在该函数中，可以看到 switch case ZDO_STATE_CHANGE：这样的分支结构，在此分支结构下的语句：

```
else if(S2App_NwkState == DEV_END_DEVICE)
{
    osal_start_timerEx(S2AppTaskID,S2APP_END_DEVICE_PERIODIC_MSG_EVT,1000);
}
```

即为终端节点周期新发送数据的起始位置。通过该分支，可以找到关于 S2APP_END_DEVICE_PERIODIC_MSG_EVT 事件的判断处理代码，即可在代码块中添加读卡函数。

```
if(events & S2APP_END_DEVICE_PERIODIC_MSG_EVT)
  {

    S2APP_ReadCard();
    osal_start_timerEx(S2AppTaskID,S2APP_END_DEVICE_PERIODIC_MSG_EVT,1000);

    return (events ^SymbolYCp S2APP_END_DEVICE_PERIODIC_MSG_EVT);
  }
```

b）完成读卡函数 S2APP_ReadCard()，代码如下所示。

```
void S2APP_ReadCard(void){
  //设置卡片激活命令
  uint8 RfidCmd[ ] = {0x86,0x06,0x0,0x2C,0x26,0x00,0x8A};
  //将卡片激活命令从终端节点的串口发送给读卡器模块
  HalUARTWrite(0,RfidCmd,sizeof(RfidCmd)/sizeof(RfidCmd[0]));
}
```

c）S2APP_ReadCard()函数的原型声明如下。

```
void S2App_Init( uint8 task_id );
uint16 S2App_ProcessEvent( uint8 task_id, uint16 events );
void UartCallback(uint8 port,uint8 event);
void S2APP_EndSendPeriodicMsg(uint8 * nfpBuf,uint8 len);
void S2App_MessageMSGB(afIncomingMSGPacket_t *pkt);
void S2APP_ReadCard(void);                              //读卡函数的原型声明
```

d）串口回调函数，当终端节点的串口收到 RFID 读卡器读到的数据即向协调器发送数据。

```
uint8 rxBuf[128];
void UartCallback(uint8 port,uint8 event)              //串口回调函数
{//只要终端节点的串口收到了读卡器的数据，就向协调器发送
  //(调用终端节点发消息函数)
   uint16 rxCount;
  if(event&HAL_UART_RX_TIMEOUT)
  {
    osal_memset(rxBuf,0,20);                           //缓冲区清零
    rxCount = Hal_UART_RxBufLen(0);                    //读出缓冲区长度
    //读串口，把读卡器的数据通过串口读入到终端节点中
    HalUARTRead(0,rxBuf,rxCount);                      //读入到 rxBuf 数组中
    //把读到的读卡器内容由终端节点向协调器发送
    S2APP_EndSendPeriodicMsg(rxBuf,rxCount);
  }
}
```

e）实现函数 S2APP_EndSendPeriodicMsg(rxBuf,rxCount)。

写到这里，大家可以回顾一下 ZStack API 中提供的两个重要函数：afRegister()用于网络注册，AF_DataRequest()用于网络数据无线传输。在本函数中，即调用了 AF_DataRequest()函数，将读卡器数据传送给协调器。

```
void S2APP_EndSendPeriodicMsg(uint8 * nfcBuf,uint8 len){    //终端节点发消息
  afStatus_t ret;                                           //zstackAPI
  ret = AF_DataRequest(&S2APP_Periodic_DstAddr,
```

```
                        &S2App_epDesc,
                        S2APP_NFC_CLUSTER_ID,
                        len,
                        nfcBuf,
                        &S2App_TransID,
                        AF_DISCV_ROUTE,
                        AF_DEFAULT_RADIUS);
}
```

完成函数的定义后，将 S2APP_EndSendPeriodicMsg(rxBuf,rxCount)函数原型声明添加至 s2App.c 文件的函数声明代码区域即可，代码如下。

```
void S2App_Init( uint8 task_id );
uint16 S2App_ProcessEvent( uint8 task_id, uint16 events );
void UartCallback(uint8 port,uint8 event);
void S2APP_EndSendPeriodicMsg(uint8 * nfpBuf,uint8 len);     //添加函数原型声明
void S2App_MessageMSGB(afIncomingMSGPacket_t * pkt);
void S2APP_ReadCard(void);
```

② 协调器接收消息，并向串口打印输出。

a) AF_INCOMING_MSG_CMD 分支：协调器接收到的是终端节点发来的 RFID 消息，在 AF_INCOMING_MSG_CMD 分支中，代码如下。

```
case AF_INCOMING_MSG_CMD://协调器收到消息的事件
        S2App_MessageMSGB(MSGpkt);          //协调器收消息
```

因此，依据该分支实现函数 void S2App_MessageMSGB(afIncomingMSGPacket_t * pkt)，代码如下。

```
//协调器收消息
void S2App_MessageMSGB(afIncomingMSGPacket_t * pkt)
{//将收到的消息向串口 PL2303 打印输出
   uint8 DeviceType;
  DeviceType = pkt->clusterId;
  if(DeviceType == S2APP_NFC_CLUSTER_ID){
    HalUARTWrite(0,pkt->cmd.Data,pkt->cmd.DataLength);
  }
}
```

添加 void S2App_MessageMSGB(afIncomingMSGPacket_t * pkt)原型声明。

```
void S2App_Init( uint8 task_id );
uint16 S2App_ProcessEvent( uint8 task_id, uint16 events );
void UartCallback(uint8 port,uint8 event);
void S2APP_EndSendPeriodicMsg(uint8 * nfpBuf,uint8 len);
```

```
void S2App_MessageMSGB(afIncomingMSGPacket_t * pkt);      //协调器收到消息后向串口打
                                                            印输出的函数原型声明
void S2APP_ReadCard(void);
```

(3) 调试运行。

串口调试助手：波特率设为 9600b/s，设置为十六进制显示(十六进制显示的内容便于和 NFC 用户手册中的数据帧格式对比分析)。

将带有读卡器的 ZigBee 模块烧写为终端节点，将连接 pl2303 的 ZigBee 模块烧写为协调器；用串口助手查看协调器的串口输出，可观察到，读卡器读到的卡片信息，通过串口发送给终端节点，终端节点将卡片信息无线传输至协调器。最终，卡片信息通过协调器串口输出至上位机显示。

3.5 习题

1. 如果在 ZigBee 网络中实现点对点的通信，需要使用_____地址模式；在 ZigBee 网络中协调器需要网络中的每个设备都收到数据使用_____模式。

2. 下列哪个选项不是 CC2530 数据帧的基本结构组成部分？(　　)

A. 同步头　　B. 需要传输的数据

C. 帧尾　　D. 数据类型符

3. 中国使用的 ZigBee 工作的频段是_____，定义了_____信道。

4. 在 ZigBee 协议架构中哪一组是属于 IEEE 802.15.4 标准定义的？(　　)

A. 物理层和 MAC 层　　B. 网络层和 MAC 层

C. 物理层和网络层　　D. 应用层和 MAC 层

5. 阐述 ZigBee 技术的特点：

6. 下列(　　)不属于 ZigBee 的拓扑结构。

A. 星形　　B. 树形

C. 网状　　D. 总线型

7. OSAL 提供的信息管理 API 函数有_____、_____、_____和_____。

8. MAC 层提供_____和_____，并负责数据成帧。

9. ZigBee 网络结构分为 4 层，从下至上分别为_____、_____、_____和_____。

10. ZDO 提供了 ZigBee 设备管理功能包括_____、_____、_____、_____和_____等服务。

11. 以下(　　)项用于存储被识别物体的标识信息？

A. 天线　　B. 电子标签

C. 读写器　　D. 计算机

12. RFID 属于物联网的(　　)层。

A. 应用　　B. 网络

C. 业务　　D. 感知

13. 射频识别技术主要是基于(　　)和(　　)方式进行信息传输的。

A. 声波　　B. 电场

C. 双绞线　　D. 磁场

14. 超高频 RFID 卡的作用距离(　　)。

A. 小于 10cm　　B. 1～20cm

C. 3～8m　　D. 大于 10m

15. RFID 卡的读取方式(　　)。

A. CCD 或光束扫描　　B. 电磁转换

C. 无线通信　　D. 电擦除、写入

16. RFID 卡(　　)可分为只读(R/O)标签、读写(R/W)标签和 CPU 标签。

A. 按供电方式分　　B. 按工作频率分

C. 按通信方式分　　D. 按标签芯片分

17. (多选题) RFID 标签的分类按通信方式分包括(　　)。

A. 主动式标签(TTF)　　B. 被动式标签(RTF)

C. 有源(Active)标签　　D. 无源(Passive)标签

18. (多选题) RFID 标签的分类按工作频率分有(　　)。

A. 低频(LF)标签　　B. 高频(HF)标签

C. 超高频(UHF)标签　　D. 微波(uW)标签

19. (多选题)RFID 的技术特点有(　　)。

A. 非接触式,中远距离工作　　B. 大批量、由读写器快速自动读取

C. 信息量大、可以细分单品　　D. 芯片存储,可多次读取

20. (多选题) RFID 标签的分类按供电方式分有(　　)。

A. 高频标签　　B. 低频标签

C. 有源(Active)标签　　D. 无源(Passive)标签