第 4 章　ZigBee 无线数据通信的设计与实现

本章学习目标

本章将在前几章知识学习的基础上，进行实际的无线数据通信实验。为了能快速实现 ZigBee 无线网络的数据收发，本章将首先对 ZigBee 协议栈数据通信中几个重要的函数和协议栈串口通信的实现方法进行讲解，最后通过几个具体的实例实现 ZigBee 无线组网的数据传输。通过本章的学习，具体要求读者掌握以下目标:

- 理解 ZigBee 协议栈的串口通信机制
- 掌握 ZigBee 协议栈应用层关键函数的原理和使用方法
- 掌握单播通信的实现方法
- 掌握串口透传的设计和实现方法

4.1　ZigBee 协议栈应用层关键函数解析

在 ZigBee 协议栈中已经实现了 ZigBee 协议，用户在进行应用程序开发时可以直接使用协议栈提供的 API 进行编程设计，在实际项目开发过程中完全不必关心 ZigBee 协议提供的具体细节，只需要关心一个最核心的问题，就是系统采集到的数据是从哪里来的，并且到哪里去了。

1. 具体步骤

举个例子，用户实现一个简单的无线数据通信时无非就是以下 3 个步骤。

（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。

因此，开发人员不需要关心协议栈具体是怎么实现的，甚至成千上万条函数代码每条是什么意思，只需要知道协议栈中提供的函数能实现什么功能，会调用相应的函数来实现自己所需要的功能即可。

既然协议栈已经做了很多的工作，用户只需要在应用层上实现自己的应用就可以了。这也是用户进行二次开发时用得最多的地方。下面对协议栈应用层中的几个关键函数进行讲解。

2. 协议栈主要函数说明

（1）任务的初始化函数。

分析任务需要完成的功能，需要对发送函数使用终端配置进行注册，对组网参数的设定、

对发送模式、接收地址的设定等。程序清单 4.1 是协议栈中任务的初始化代码。

程序清单 4.1

```
/*************************************************
*函数名：SampleApp_Init
*功能描述：初始化应用任务的函数，它是在任务初始化列表中被调用
*参数：task_id-OASL 分配的任务 ID，这个 ID 将用于发送消息和设定 OS 时间
*返回：无
*/
void SampleApp_Init( uint8 task_id )
{
  SampleApp_TaskID = task_id;              //分配任务的 ID
  SampleApp_NwkState = DEV_INIT;           //网络类型
  SampleApp_TransID = 0;

/*通过硬件来实现设备类型的选择，该功能必须要 BUILD_ALL_DEVICE 打开后才能使用*/
 #if defined ( BUILD_ALL_DEVICES )
  if ( readCoordinatorJumper() )
    zgDeviceLogicalType = ZG_DEVICETYPE_COORDINATOR;
  else
    zgDeviceLogicalType = ZG_DEVICETYPE_ROUTER;
#endif                  // BUILD_ALL_DEVICES

#if defined ( HOLD_AUTO_START )
  // HOLD_AUTO_START is a compile option that will surpress ZDApp
  // from starting the device and wait for the application to
  // start the device
  ZDOInitDevice(0);
#endif

  //定义发送的数据为广播方式（网络中所有节点都能收到）
  SampleApp_Periodic_DstAddr.addrMode = (afAddrMode_t)AddrBroadcast;       //广播方式
  SampleApp_Periodic_DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
  SampleApp_Periodic_DstAddr.addr.shortAddr = 0xFFFF;                      //网络地址

  //定义发送数据的方式为组播发送
  SampleApp_Flash_DstAddr.addrMode = (afAddrMode_t)afAddrGroup;            //组播方式
  SampleApp_Flash_DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
  SampleApp_Flash_DstAddr.addr.shortAddr = SAMPLEAPP_FLASH_GROUP;    //组地址

  //终端节点描述
  SampleApp_epDesc.endPoint = SAMPLEAPP_ENDPOINT;
  SampleApp_epDesc.task_id = &SampleApp_TaskID;
  SampleApp_epDesc.simpleDesc
            = (SimpleDescriptionFormat_t *)&SampleApp_SimpleDesc;
  SampleApp_epDesc.latencyReq = noLatencyReqs;

  //对终端节点描述注册
  afRegister( &SampleApp_epDesc );
```

```
  //注册所有按键任务
  RegisterForKeys( SampleApp_TaskID );
  //默认所有的设备在组 1 中
  SampleApp_Group.ID = 0x0001;                                    //组 ID 号
  osal_memcpy( SampleApp_Group.name, "Group 1", 7    );
  aps_AddGroup( SAMPLEAPP_ENDPOINT, &SampleApp_Group );           //设备加入组

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

（2）任务处理函数。

任务处理函数包括对按键事件的处理、对接收消息的处理和设备类型的处理等，如程序清单 4.2 所示。

程序清单 4.2

```
/*************************************************
*函数名：SampleApp_ProcessEvent
*功能描述：应用任务处理，这个函数是处理任务中的所有事件
*参数：task_id（OS 分配的 ID）events（事件处理）
*返回：无
*/
uint16 SampleApp_ProcessEvent( uint8 task_id, uint16 events )
{
  afIncomingMSGPacket_t *MSGpkt;
  (void)task_id;                                  // Intentionally unreferenced parameter
  if ( events & SYS_EVENT_MSG )
  {
    MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );
    while ( MSGpkt )
    {
      switch ( MSGpkt->hdr.event )
      {
        //当一个按键触发时，返回按键事件
        case KEY_CHANGE:
          SampleApp_HandleKeys( ((keyChange_t *)MSGpkt)->state, ((keyChange_t *)MSGpkt)->keys );
          break;
        //当收到消息后，返回接收消息事件
        case AF_INCOMING_MSG_CMD:
          SampleApp_MessageMSGCB( MSGpkt );
          break;
        //当网络状态发生变化时，返回状态事件
        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) )
          {
            //开始周期发送函数，启动发送数据事件
            osal_start_timerEx( SampleApp_TaskID,
```

```
                                        SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
                                        SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
                }
                else
                {
                  //设备不在网络中
                }
                break;
              default:
                break;
            }
            //释放内存
            osal_msg_deallocate( (uint8 *)MSGpkt );
            //下一个任务
            MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );
          }
          //返回未处理的事件
          return (events ^ SYS_EVENT_MSG);
        }
          //发送一个信息出去
        if ( events & SAMPLEAPP_SEND_PERIODIC_MSG_EVT )
        {
          //周期发送一个数据
          SampleApp_SendPeriodicMessage();
          //下次发送数据的事件设置
          osal_start_timerEx(SampleApp_TaskID, SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
              (SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT + (osal_rand() & 0x00FF)) );
          //返回未加工事件
          return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);
        }
        //丢弃未加工事件
        return 0;
      }
```

（3）发送函数。

在 SampleApp 中提供了两个发送函数，这两个函数提供了广播发送和组播发送两种发送方式，SampleApp_SendPeriodicMessage 和 SampleApp_SendFlashMessage 函数就是实现发送的这两个函数。发送函数通过调用 AF_DataRequest 函数实现数据的发送，在 AF_DataRequest 函数中需要进行参数的设置。各参数如下：

```
*dstAddr:
*srcEP:
cID:
Len:
*buf:
*transID :
Options:
Radius: AF_DEFAULT_RADIUS
```

在初始化函数 SampleApp_Init 中，已经定义了*dstAddr 和*srcEP 的参数，所以在发送函数中不需要设置，直接调用就可以。其他参数可以根据自己的实际情况设置。发送函数如程序清单 4.3 和 4.4 所示。

程序清单 4.3

```
/*************************************************
*函数名：SampleApp_SendPeriodicMessage
*功能描述：数据广播发送
*参数：无
*返回：无
*/
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
  {
    //发送失败
  }
}
```

程序清单 4.4

```
/*************************************************
*函数名：SampleApp_SendFlashMessage
*功能描述：数据组播发送
*参数：flashTime——发送时间，以 ms 为单位
*返回：无
*/
void SampleApp_SendFlashMessage( uint16 flashTime )
{
  uint8 buffer[3];                          //发送数据 buffer
  buffer[0] = (uint8)(SampleAppFlashCounter++);  //发送的次数
  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
    {
      //发送数据错误
    }
}
```

（4）接收处理函数。

接收处理函数的功能是将接收到的数据进行解析，并根据不同的 ClusterID 实现不同的功能。实现代码如程序清单 4.5 所示。

程序清单 4.5

```
/*************************************************
*函数名：SampleApp_MessageMSGCB
*功能描述：数据接收处理
*参数：afIncomingMSGPacket_t
*返回：无
*/
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
  uint16 flashTime;
  switch ( pkt->clusterId )                          //取得接收到的 clusterId 值
  {
    case SAMPLEAPP_PERIODIC_CLUSTERID:               //如果是周期性广播数据
      break;
    case SAMPLEAPP_FLASH_CLUSTERID:                  //如果是组播数据
      flashTime = BUILD_UINT16(pkt->cmd.Data[1], pkt->cmd.Data[2] );
      HalLedBlink( HAL_LED_4, 4, 50, (flashTime / 4) );
      break;
  }
}
```

通过协议栈中应用层这些关键函数的调用，用户可以很容易地通过二次开发设计出自己的应用程序。在接下来的系统设计中，将使用以上函数实现 ZigBee 无线网络的数据收发。

4.2 ZigBee 协议栈串口通信功能的实现

4.2.1 串行通信简介

串行通信是将数据字节分成一位一位的形式在一条传输线上逐个传送。串口按位（bit）发送和接收字节。尽管比按字节（byte）的并行通信慢，但是串口可以在使用一根线发送数据的同时用另一根线接收数据。它很简单并且能够实现远距离通信，如图 4-1 所示。

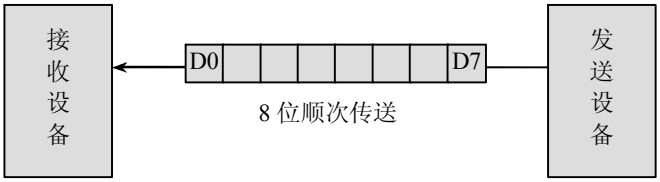

图 4-1　串行数据传输

串行通信的特点：传输线少，长距离传送时成本低，且可以利用电话网等现成的设备，但数据的传送控制比并行通信复杂。

串行通信的传输方向如下。

1. 单工

单工是指数据传输仅能沿一个方向，不能实现反向传输，如图 4-2 所示。

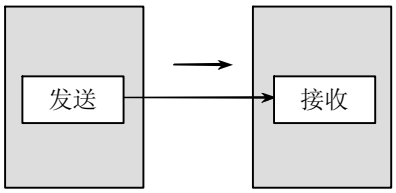

图 4-2　单工通信

2. 半双工

半双工是指数据传输可以沿两个方向，但需要分时进行，如图 4-3 所示。

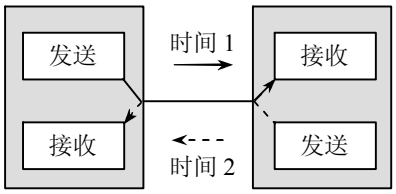

图 4-3　半双工通信

3. 全双工

全双工是指数据可以同时进行双向传输，如图 4-4 所示。

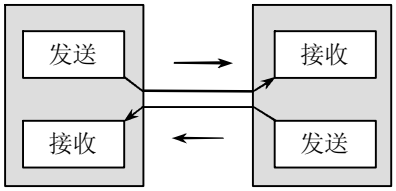

图 4-4　全双工通信

4.2.2 串行数据传输

串行数据传输是以字符（构成的帧）为单位进行的传输，字符与字符之间的间隙（时间间隔）是任意的，但每个字符中的各位是以固定的时间传送的，即字符之间不一定有“位间隔”的整数倍的关系，但同一字符内的各位之间的距离均为“位间隔”的整数倍。串行数据传输的数据格式如图 4-5 所示。

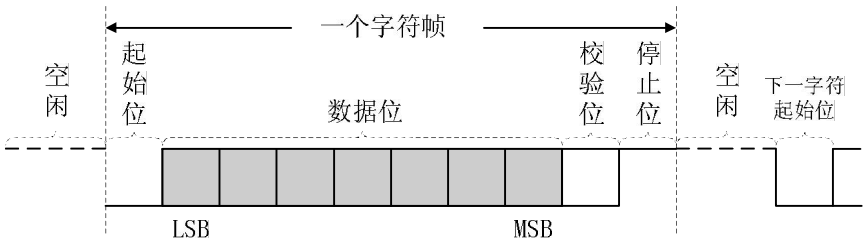

图 4-5　串行数据传输的数据格式

典型地，串口用于 ASCII 码字符的传输，通信使用 3 根线完成：地线、发送线、接收线。由于串口通信是异步的，端口能够在一根线上发送数据同时在另一根线上接收数据。其他线用于握手，但不是必须的。串口通信最重要的参数是波特率、数据位、停止位和奇偶校验。

1. 波特率

这是一个衡量通信速度的参数，它表示每秒钟传送的位的个数。波特率是每秒钟传输二进制代码的位数，单位是：位/秒（bps）。如每秒钟传送 240 个字符，而每个字符格式包含 10 位（1 个起始位、1 个停止位、8 个数据位），这时的波特率为：

$$10\text{ 位}\times 240\text{ 个/秒} = 2400\text{ bps}$$

2. 数据位

这是衡量通信中实际数据位的参数。当计算机发送一个数据帧，每个数据帧是指一个字节，包括开始/停止位，数据位和奇偶校验位，实际的数据不一定是 8 位的，也可能是 5、6、7 位，具体设置取决于想传送的信息。

3. 停止位

用于表示单个包的最后一位，典型的值为 1、1.5 和 2 位。由于数据是在传输线上定时的，并且每一个设备有其自己的时钟，很可能在通信中两台设备间出现了小小的不同步。因此停止位不仅仅是表示传输的结束，还提供计算机校正时钟同步的机会。

4. 奇偶校验位

它是在串口通信中一种简单的检错方式，常用的检错方式有偶校验、奇校验。当然没有校验位也是可以的。对于偶和奇校验的情况，串口会设置校验位（数据位后面的一位），用一个值确保传输的数据有偶个或者奇个位。例如，如果数据是 011，那么对于偶校验，校验位为 0。如果是奇校验，校验位为 1。

4.2.3 ZigBee 协议栈串口功能的应用实现

串口是 ZigBee 开发板和用户电脑交互的一种工具，协调器将传感器节点采集的数据通过串口发送给上位机或者协调器接收上位机通过串口发送过来的命令。正确地使用串口对学习 ZigBee 无线传感网络有极大的促进作用。

1. 使用串口的基本步骤

（1）初始化串口，包括波特率、串口号等。

（2）向缓冲区发送数据或者从接收缓冲区读取数据。

上述方法是 CC2530 单片机串口裸机编程的常用方法，但是由于 ZigBee 协议栈的存在，使得串口的使用更加简单和方便，因为在 ZigBee 协议栈中已经对串口初始化所需要的函数进行了实现，这里只需要传递几个参数就可以使用串口。此外，ZigBee 协议栈还实现了串口的读取和写入函数。

因此，用户在使用串口时，只需要掌握 ZigBee 协议栈提供的串口操作相关的三个函数即可，它们为：

```
uint8 HalUARTOpen(uint8 port,halUARTCfg_t *config);
uint16 HalUARTRead(uint8 port,uint8 *buf,uint16 len);
uint16 HalUARTWrite(uint8 port,uint8 *buf, uint16 len );
```

这三个函数分别实现打开串口、读取串口数据和写串口数据的功能。下面通过两个具体的例子来实现协议栈读写串口的功能。ZigBee 协调器与 PC 上位机之间的串口通信结构如图 4-6 所示。

2. 协议栈往上位机串口写数据功能的实现

具体要求的功能是在 ZigBee 协调器模块上电后通过协议栈自动向串口发送“UART is ok”字符串，若在上位机串口调试助手上能收到这些字符，说明下位机与上位机之间的串口通信成功。协议栈串口通信的初始化流程如图 4-7 所示。

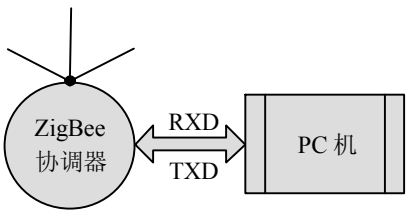

图 4-6 串口通信结构图

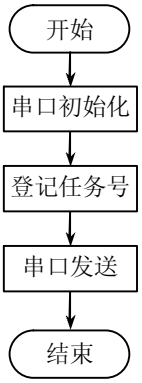

图 4-7 串口通信初始化流程

为了实现上电后，向串口调试助手输出“UART is ok”，需要对 SampleApp.c 做如下修改（新增加的部分以加粗字体显示），如程序清单 4.6 所示。

程序清单 4.6

```
/* HAL */
#include "hal_lcd.h"
#include "hal_led.h"
#include "hal_key.h"
#include "MT_UART.h"
```

MT 层是 Z-Stack 协议栈中的一个调试层，找到 MT_UART.h 文件之后，可以从中找到 MT_UartInit()函数。这里可以直接使用该函数进行串口初始化，以此简化了操作流程。

接下来找到 void SampleApp_Init(uint8 task_id)应用程序初始化函数，在其中添加串口初始化“MT_UartInit();”语句，并向把串口事件通过 task_id 登记在 SampleApp_Init()里面，同时向串口输出“UART is ok”字符，如程序清单 4.7 所示。

程序清单 4.7

```
void SampleApp_Init( uint8 task_id )
{
  SampleApp_TaskID = task_id;
  SampleApp_NwkState = DEV_INIT;
  SampleApp_TransID = 0;

 /***********串口初始化************/
  MT_UartInit();                          //串口初始化
  MT_UartRegisterTaskID(task_id);         //登记事件任务号
  HalUARTWrite(0,"UART is ok\n",11);
```

ZigBee 协议栈中对串口的配置是使用一个结构体来实现的，该结构体为 halUARTCfg_t，在此不必关心该结构体的具体定义形式，只需要对其功能有个了解，该结构体将串口初始化有关的参数集合在一起，例如波特率、是否打开串口、是否使用流控等，这里只需要将各个参数初始化即可。

进入 MT_UartInit()函数，修改串口初始化配置，需要修改的地方有以下几点：

（1）“uartConfig.baudRate = MT_UART_DEFAULT_BAUDRATE;”语句是配置波特率，继续进入 MT_UART_DEFAULT_BAUDRATE 的定义，可以看到：

#define MT_UART_DEFAULT_BAUDRATE HAL_UART_BR_38400

默认的波特率是 38400bps，现在修改成 115200bps，修改如下：

#define MT_UART_DEFAULT_BAUDRATE HAL_UART_BR_115200

（2）“uartConfig.flowControl = MT_UART_DEFAULT_OVERFLOW;”语句是配置流控的，进入定义可以看到：

#define MT_UART_DEFAULT_OVERFLOW TRUE

默认是打开串口流控的，这里串口是只连接 RX/TX 两根线的，所以要关闭流控，修改如下：

#define MT_UART_DEFAULT_OVERFLOW FALSE

此外，由于协议栈串口发送采用了 MT 层定义的串口发送格式，使得一些不需要的调试

信息也在串口通信时出现，需要在预编译时将其去除，在 IAR 环境中，具体的操作方法如下：

1）在 SampleApp-CoordinatorEB-Pro 工程上右击，在弹出的菜单中选择 Options 选项，如图 4-8 所示。

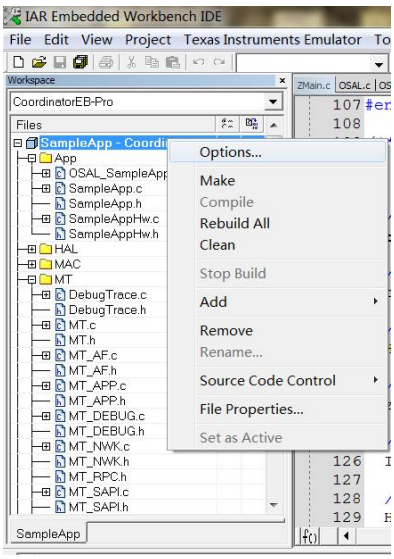

图 4-8　选择 Options 菜单

2）在弹出的 Options for node "SampleApp"对话框中，选择 C/C++ Compiler 选项，在对话框右边选择 Preprocessor 选项卡，然后在 Defined symbols 列表框中将 MT 和 LCD 相关的内容注释掉，最后单击 OK 按钮即可，如图 4-9 所示。

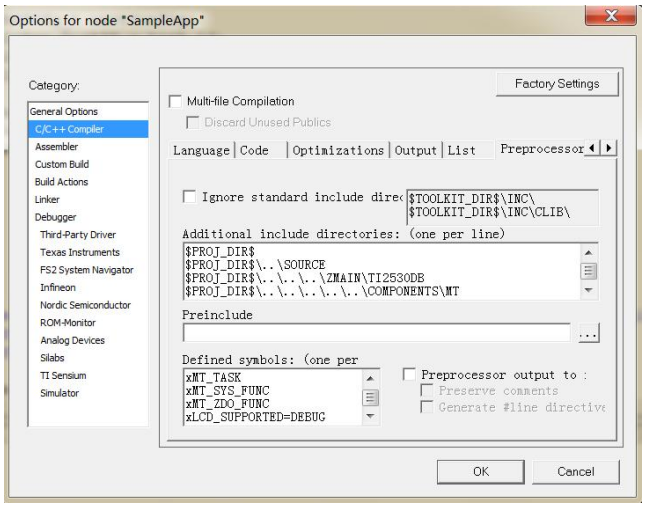

图 4-9　注释相关语句

经过这些修改，现在串口已经可以发送信息了。连接 CC DEBUGGER 和 USB 转串口线，选择 CoordinatorEB-Pro，单击下载并调试。全速运行，可以看到协调器重新上电后上位机串口调试助手出现如图 4-10 所示的信息。

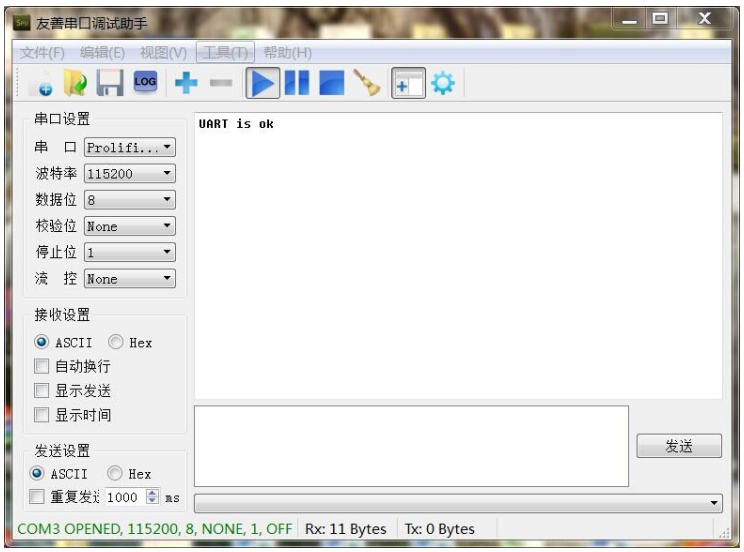

图 4-10　上电后提示 UART is ok

3．协议栈通过串口读取上位机数据

具体要求为上位机能将特定含义的数据发送给下位机 ZigBee 协调器模块，ZigBee 协调器模块收到数据后进行回显。本例中将使用协议栈串口回调函数（非事件处理）来实现。

接下来仍然是对 SampleApp.c 进行修改，修改后的内容如程序清单 4.8 所示（新增部分加粗字体显示）。

程序清单 4.8

```
#include "OSAL.h"
#include "ZGlobals.h"
#include "AF.h"
#include "aps_groups.h"
#include "ZDApp.h"
#include "SampleApp.h"
#include "SampleAppHw.h"
#include "OnBoard.h"

/* HAL */
#include "hal_lcd.h"
#include "hal_led.h"
#include "hal_key.h"

// This list should be filled with Application specific Cluster IDs
```

```
const cId_t SampleApp_ClusterList[SAMPLEAPP_MAX_CLUSTERS] =
{
  SAMPLEAPP_PERIODIC_CLUSTERID,
  SAMPLEAPP_FLASH_CLUSTERID
};

const SimpleDescriptionFormat_t SampleApp_SimpleDesc =
{
  SAMPLEAPP_ENDPOINT,              //  int Endpoint;
  SAMPLEAPP_PROFID,                //  uint16 AppProfId[2];
  SAMPLEAPP_DEVICEID,              //  uint16 AppDeviceId[2];
  SAMPLEAPP_DEVICE_VERSION,        //  int AppDevVer:4;
  SAMPLEAPP_FLAGS,                 //  int AppFlags:4;
  SAMPLEAPP_MAX_CLUSTERS,          //  uint8 AppNumInClusters;
  (cId_t *)SampleApp_ClusterList,  //  uint8 *pAppInClusterList;
  SAMPLEAPP_MAX_CLUSTERS,          //  uint8 AppNumInClusters;
  (cId_t *)SampleApp_ClusterList   //  uint8 *pAppInClusterList;
};

endPointDesc_t SampleApp_epDesc;
uint8 SampleApp_TaskID;
devStates_t SampleApp_NwkState;
uint8 SampleApp_TransID;           // This is the unique message ID (counter)

afAddrType_t SampleApp_Periodic_DstAddr;
afAddrType_t SampleApp_Flash_DstAddr;
aps_Group_t SampleApp_Group;
uint8 SampleAppPeriodicCounter = 0;
uint8 SampleAppFlashCounter = 0;

unsigned char uartbuf[128];              //添加字符串（数组）变量

void SampleApp_HandleKeys( uint8 shift, uint8 keys );
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pckt );
void SampleApp_SendPeriodicMessage( void );
void SampleApp_SendFlashMessage( uint16 flashTime );
static void rxCB(uint8 port, uint8 event);     //定义一个回调函数
void SampleApp_Init( uint8 task_id )
{
        halUARTCfg_t uartConfig;
          ...
          uartConfig.configured = TRUE;
          uartConfig.baudRate = HAL_UART_BR_115200;
          uartConfig.flowControl = FALSE;
          uartConfig.callBackFunc = rxCB;
          HalUARTOpen(0, &uartConfig);

}
```

```
static void rxCB(uint8 port, uint8 event)
      {
            HalUARTRead(0, uartbuf, 16);                    //读取串口缓冲区数据
//判断收到的字符串是否是 www.chuangjian.com
 if(osal_memcmp(uartbuf,"www.chuangjian.com",18))
                  {
                     HalUARTWrite(0, uartbuf,18);           //将读到的数据写到串口中
                   }
      }
```

rxCB()函数完成了读取缓冲区中的数据，该函数是一个回调函数，回调函数就是通过函数指针（函数地址）调用的函数。如果把函数的指针（即函数的地址）作为参数传递给另一个函数，当通过这个指针调用它所指向的函数时，称为函数的回调。

将程序编译后下载到 ZigBee 协调器中，设置好串口调试助手，在输入栏输入一串字符“www.chuangjian.com”，单击“发送”按钮，可以看到在接收区显示“www.chuangjian.com”，说明串口收发正常，如图 4-11 所示。

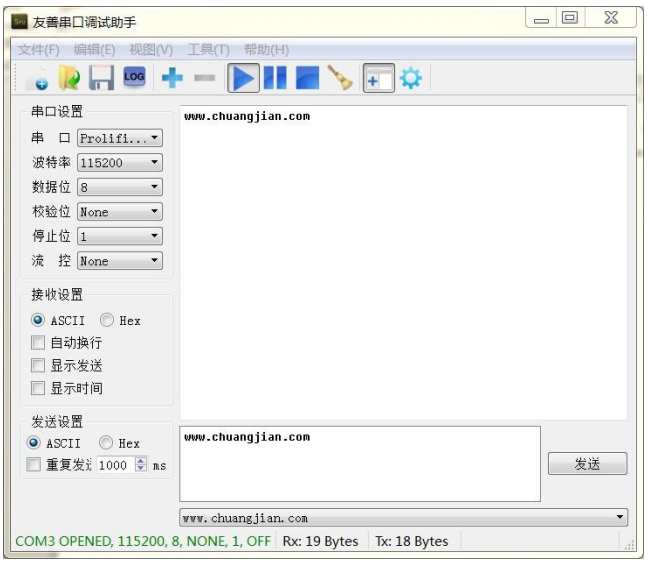

图 4-11　协议栈读取上位机串口发送的数据

4.3　ZigBee 无线数据通信的实现

上一节我们已经完成了串口的初始化设置并实现了基本的串口收发功能，本节将在上节的基础上完成一个最简单的无线数据传输实验。本实验的实验原理如图 4-12 所示，即协调器建立网络后，能和终端节点通过这一网络进行数据的传输。

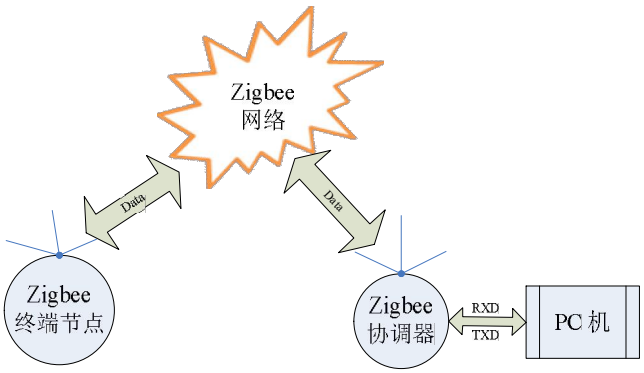

图 4-12　数据通信实验原理图

下面我们首先来测试一下无线传输的实验效果，再去分析其中的原理。在 SampleApp.c 中，找到消息处理函数 void SampleApp_MessageMSGCB(afIncomingMSGPacket_t *pkt)，在 case SAMPLEAPP_PERIODIC_CLUSTERID 下面加入 HalUARTWrite(0,"received data\n",14)语句，如图 4-13 所示。

```
396  * @param   none
397  *
398  * @return  none
399  */
400 void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
401 {
402   uint16 flashTime;
403
404   switch ( pkt->clusterId )
405   {
406     case SAMPLEAPP_PERIODIC_CLUSTERID:
407       HalUARTWrite(0,"received data\n",14);
408       break;
```

图 4-13　在消息处理函数中加入一行语句

选择 CoodinatorEB-pro 和 EndDeviceEB-pro，分别下载到协调器（作为协调器串口跟电脑连接）和终端节点（作为终端设备无线发送数据给协调器）开发板上。给两个模块上电，打开串口调试助手，可以看到大约 5s 后会收到“received data”的内容，如图 4-14 所示。

4.3.1　实验原理解析

整个协议栈只添加了一行代码，就完成了无线数据传输的实验。实际上我们是使用了 SampleApp 上一个广播的例子修改而来的。整个数据传输过程可以分为数据发送部分和数据接收部分，但是前提是必须有相应的任务事件触发。发送部分的工作流程如图 4-15 所示。

1. 登记事件、设计编号和发送时间

本实验中的事件包括了网络状态改变事件、定时器事件和数据接收事件。打开 SampleApp.c 文件，找到 SampleApp 事件处理函数 SampleApp_ProcessEvent，如程序清单 4.9 所示。

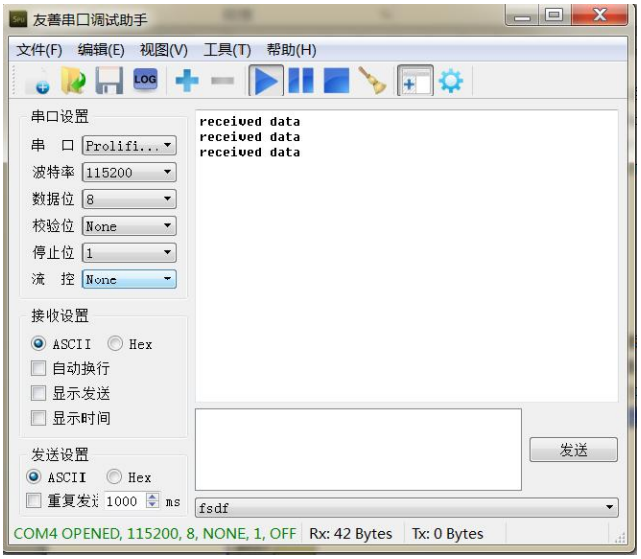

图 4-14　收到 received data 的内容

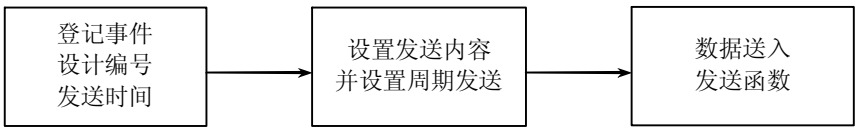

图 4-15　发送部分的工作流程

程序清单 4.9

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

……
……
        // Received when a messages is received (OTA) for this endpoint
        case AF_INCOMING_MSG_CMD:
          SampleApp_MessageMSGCB( MSGpkt );
          break;
        // Received whenever the device changes state in the network
        case ZDO_STATE_CHANGE:
          SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
          if ( (SampleApp_NwkState == DEV_ZB_COORD)
              || (SampleApp_NwkState == DEV_ROUTER)
              || (SampleApp_NwkState == DEV_END_DEVICE) )
          {
            // Start sending the periodic message in a regular interval
            osal_start_timerEx( SampleApp_TaskID,
                                SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
                                SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
          }
```

```
            else
            {
              // Device is no longer in the network
            }
            break;
          default:
            break;
        ......
    ......
```

其中 case ZDO_STATE_CHANGE 是 ZigBee 设备对象网络状态改变的事件，下面的 if 语句表明，当协调器、路由器和终端中的任何一个节点网络状态发生改变时，都会触发该事件，也就是说一旦协调器建立网络，或者路由器或终端节点加入网络，即组网成功后，该事件就被触发了。本实验中正是终端节点加入协调器的网络之后进入了该 Case 事件中。

接下来的 osal_start_timerEx()函数是重点，该函数可以当做协议栈下的定时器来使用，不过它的使用需要带 3 个参数，第一个就是使用该函数的 TaskID，函数开头定义了 SampleApp_TaskID = task_id，也就是使用 SampleApp 初始化的任务 ID 号。第二个就是当定时结束时向协议栈发出的消息，即事件的编号，第三个参数是定时的时长。后两个参数在 SampleApp.h 中的定义为：

```
#define SAMPLEAPP_SEND_PERIODIC_MSG_EVT       0x0001
#define SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT     5000    // Every 5 seconds
```

通过分析这段代码，我们得出了这样的结论：ZigBee 节点状态发生改变时（通常为加入或者离开网络），节点会执行一个定时为 5s 的定时函数，5s 之后会将 SAMPLEAPP_SEND_PERIODIC_MSG_EVT 事件编号发送出去。

登记好事件后，如果网络状态没有改变，就不会再次进入这个函数了，所以这个相当于初始化，只执行 1 次。

2. 设置发送内容

接下来就是实现周期性的数据发送，在同一个函数下面可以找到如程序清单 4.10 所示的代码。

程序清单 4.10

```
// Send a message out - This event is generated by a timer
// setup in SampleApp_Init()
if ( events & SAMPLEAPP_SEND_PERIODIC_MSG_EVT )
{
  // Send the periodic message
  SampleApp_SendPeriodicMessage();

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

  // return unprocessed events
  return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);
}
```

首先，判断 SAMPLEAPP_SEND_PERIODIC_MSG_EVT(0x0001)有没有发生，如果有就执行下面的 SampleApp_SendPeriodicMessage()函数。这个函数就是发送无线消息的函数，而第二个就是上面已经分析过的定时函数，这个定时函数放在这里就形成了一个循环定时，每次定时完之后都会进入该函数再次定时，从而实现了周期性的定时发送。

SampleApp_SendPeriodicMessage()是周期性的广播发送函数，是用户编写需要发送内容的地方，具体的内容用户在这个函数中进行一些修改就行。在 SampleApp.c 中找到 SampleApp_SendPeriodicMessage()函数代码，如程序清单 4.11 所示。

程序清单 4.11

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

在此函数中，最核心的就是无线数据发送函数 afStatus_t AF_DataRequest()，用户调用该函数即可实现数据的无线发送。当然，函数中带了很多的参数，用户需要将每个参数的含义理解后，才能达到熟练应用该函数进行无线通信的目的。这里读者需要重点掌握其中的 4 个参数。该函数的原型如下：

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

（1）afAddrType_t *dstAddr 为发送模式，本实验中使用的是 SampleApp_Periodic_DstAddr，配置为广播模式。

（2）uint16 cID 为发送标号，本实验中使用 SAMPLEAPP_PERIODIC_CLUSTERID 标号，可以去定义查看它定义的标号为 1，它的作用是和接收方约定好一个标识。这个标号在接收部分也会提及，即协调器收到这个标号，如果是 1，就证明是由周期性广播方式发送过来的。

（3）uint16 len 为数据长度，这个很好理解，我们从字面上就可以猜出它是标明下一个参数*buf 的长度的，本实验中直接填写了 1，标明发送一个字节的数据。

（4）uint8 *buf 为数据指针，这个就是我们要发送数据的指针，通常事先拼凑好数据放入

一个 buf 中通过指针来传递。本实验中是一个 0。

通过该函数，数据便会以指定的方式发送，本实验中该函数的效果为广播了一个标识为 SAMPLEAPP_PERIODIC_CLUSTERID，长度为 1，内容为字符 0 的一个无线包。我们也可以自行修改一下，比如改为长度为 4，内容为 2014 的一个数组，修改后的代码如程序清单 4.12 所示。

程序清单 4.12

```
void SampleApp_SendPeriodicMessage( void )
{
  uint8 data[4] ={'2', '0', '1', '4'};
  if ( AF_DataRequest( &SampleApp_Periodic_DstAddr, &SampleApp_epDesc,
                       SAMPLEAPP_PERIODIC_CLUSTERID,
                       4,
                       data,
                       &SampleApp_TransID,
                       AF_DISCV_ROUTE,
                       AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
  else
  {
    // Error occurred in request to send
  }
}
```

至此，发送部分代码修改完成，上电后 ZigBee 模块会以 5s 为周期来广播发送数据 2014。

3. 接收数据

接收部分需要完成 2 个任务：

（1）读取接收到的数据。

（2）把数据通过串口发送给 PC 机。

接收函数位于 SampleApp.c 文件中 SampleApp_ProcessEvent 函数下的 case AF_INCOMING_MSG_CMD 事件中。该事件与 ZDO_State_Change 事件为并列等级事件，该事件的作用是每当 OSAL 层接收到无线消息时，会触发该事件，并调用消息处理函数。相关代码如程序清单 4.13 所示。

程序清单 4.13

```
// Received when a message is received (OTA) for this endpoint
        case AF_INCOMING_MSG_CMD:
        SampleApp_MessageMSGCB( MSGpkt );
        break;
```

SampleApp_MessageMSGCB(MSGpkt)就是将接收到的数据包进行处理的函数，所有的操作都被封装到了 SampleApp_MessageMSGCB(MSGpkt)函数中。进入 SampleApp_MessageMSGCB(MSGpkt)函数，也就是第一个实验开头插入代码的函数位置。代码如程序清单 4.14 所示。

程序清单 4.14

```
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
 switch ( pkt->clusterId )
  {
    case SAMPLEAPP_PERIODIC_CLUSTERID:
      HalUARTWrite(0,"received data\n",14);
      break;
……
}
```

在这里可以看到 SAMPLEAPP_PERIODIC_CLUSTERID 标识，因为接收函数中的标识和发送函数的标识一致，确认是周期性广播数据，所以便执行了该 case 语句的内容将“received data”字符通过串口显示出来了。

但是，通过分析我们也发现，这并不是由终端节点发送过来的真实数据，只是在接收到数据时才执行了这一行代码。那么真实收到的数据在什么地方呢？在接收处理函数中可以看到有个 afIncomingMSGPacket_t 类型的参数，进入类型定义如程序清单 4.15 所示。

程序清单 4.15

```
typedef struct
{
  osal_event_hdr_t hdr;
  uint16 groupId;
  uint16 clusterId;
  afAddrType_t srcAddr;
  uint16 macDestAddr;
  uint8 endPoint;
  uint8 wasBroadcast;
  uint8 LinkQuality;
  uint8 correlation;
  int8   rssi;
  uint8 SecurityUse;
  uint32 timestamp;
  afMSGCommandFormat_t cmd;     /* Application Data */
} afIncomingMSGPacket_t;
```

这里最重要的就是cmd成员变量，根据TI Zstack协议栈的注释，这个afMSGCommandFormat_t类型的变量里就存放着应用层收到的数据，afMSGCommandFormat_t 结构体的定义如程序清单 4.16 所示。

程序清单 4.16

```
typedef struct
{
  byte    TransSeqNumber;     //存储的发送序列号
  uint16 DataLength;          //存储的发送数据的长度信息
  byte   *Data;               //存储的接收数据缓冲区的指针
} afMSGCommandFormat_t;
```

接收到的数据就存放在结构体 afMSGCommandFormat_t 中的这三个变量中，数据接收后存放在一个缓冲区中，*Data 参数存储了指向该缓冲区的指针，&pkt->cmd.Data 就是存放接收数据的首地址。所以如果要把改好的发射端发送的 2014 字符显示出来，只需要将 Data 指针首地址开始的 4 个字符取出来送到串口即可。代码如程序清单 4.17 所示。

程序清单 4.17

```
switch ( pkt->clusterId )
{
  case SAMPLEAPP_PERIODIC_CLUSTERID:
      HalUARTWrite(0,"received data\n",14);
      HalUARTWrite(0, &pkt->cmd.Data[0],4);      //串口打印收到的数据
      HalUARTWrite(0,"\n",1);                    //回车换行
    break;
}
```

将修改后的程序分别下载到协调器和终端节点开发板上。给两个模块上电，打开串口调试助手，可以看到此时协调器大约每隔 5s 会收到“2014”字符串，而“2014”正是网络中传输的真正数据，如图 4-16 所示。

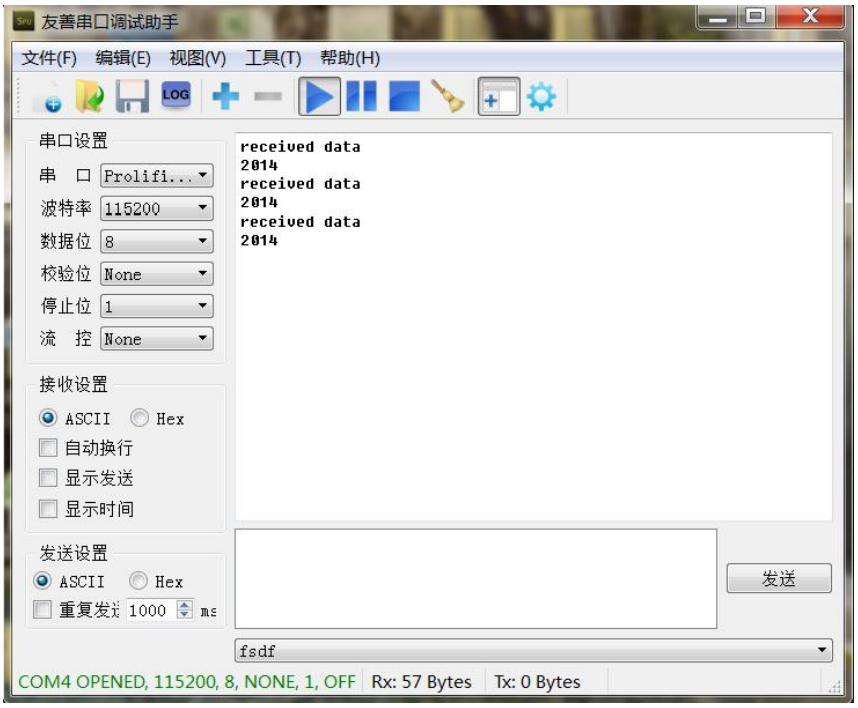

图 4-16　接收到 2014 字符串

至此，已经实现了基本的无线数据广播通信的功能，总流程如图 4-17 所示。

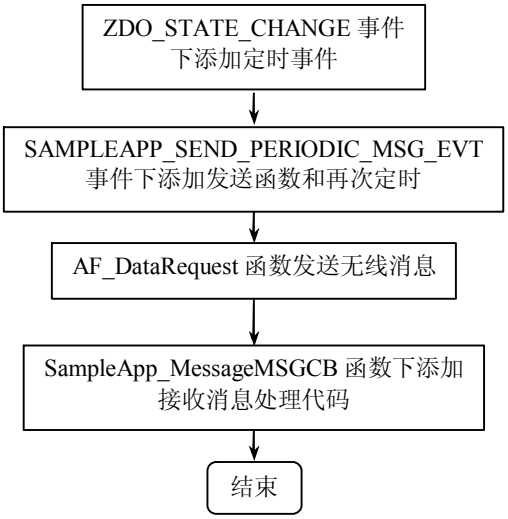

图 4-17　无线数据广播通信的实现流程

4.3.2　ZigBee 单播通信的实现

ZigBee 协议栈将数据通信过程高度抽象，使用一个函数完成数据的通信，以不同的参数来选择数据的发送方式（广播、组播还是单播）。经过前面的无线数据通信实验，可以看出 ZigBee 协议栈默认使用的是周期性广播的发送方式。

单播描述的是网络中 2 个节点相互通信的过程，即点和点之间的通信。使用单播可以使终端设备有针对性地发送数据给指定设备，不像广播和组播可能会造成数据冗余。

在上一个实验中已经介绍了无线发送函数 afStatus_t AF_DataRequest()的第一个参数 afAddrType_t *dstAddr 决定了数据通信的方式，它是一个指向 afAddrType_t 类型结构体的指针，该结构体的定义如程序清单 4.18 所示。

程序清单 4.18

```
typedef struct
{
  union
  {
    uint16 shortAddr;
    ZLongAddr_t extAddr;
  } addr;
  afAddrMode_t addrMode;
  byte endPoint;
  uint16 panId;    // used for the INTER_PAN feature
} afAddrType_t;
```

其中 addrMode 参数是一个 afAddrMode_t 类型的变量，afAddrMode_t 类型的定义如程序清单 4.19 所示。

程序清单 4.19

```
typedef enum
{
  afAddrNotPresent = AddrNotPresent,
  afAddr16Bit = Addr16Bit,
  afAddr64Bit = Addr64Bit,
  afAddrGroup = AddrGroup,
  afAddrBroadcast = AddrBroadcast
} afAddrMode_t;
```

可见，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。

这里使用到的 AddrBroadcast、AddrGroup、Addr64Bit 在 ZigBee 协议栈里面的定义如程序清单 4.20 所示。

程序清单 4.20

```
enum
{
  AddrNotPresent = 0,
  AddrGroup = 1,
  Addr16Bit = 2,
  Addr64Bit = 3,
  AddrBroadcast = 15
};
```

这里向读者详细介绍了 AF_DataRequest()函数的第一个参数，因为该参数决定了以哪种方式发送数据。单播通信具体实现的步骤和方法如下：

首先，需要在 SampleApp.c 中的发送函数 void SampleApp_SendPeriodicMessage(void)中定义一个 afAddrType_t 类型的变量。

```
afAddrType_t    SampleApp_ Unicast _DstAddr;   //自己的单播通信定义
```

然后，将 addrMode 参数设置为 Addr16Bit，并将发送地址设定为协调器的地址 0x0000。代码如程序清单 4.21 所示。

程序清单 4.21

```
SampleApp_ Unicast _DstAddr .addrMode=(afAddrMode_t)Addr16bit;        //点播
SampleApp_ Unicast _DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
SampleApp_ Unicast _DstAddr.addr.shortAddr = 0x0000;                  //发给协调器
```

最后，调用 AF_DataRequest()函数进行数据的无线发送即可。

```
AF_DataRequest( & SampleApp_ Unicast _DstAddr,……)
```

若要在上一实验基础上实现无线单播方式发送数据“2014”给协调器，无线发送函数的

代码修改如程序清单 4.22 所示。

程序清单 4.22

```
void SampleApp_SendPeriodicMessage ( void )
{
  uint8 data[4]={'2','0','1','4'};
  afAddrType_t   SampleApp_ Unicast _DstAddr;
  SampleApp_ Unicast _DstAddr .addrMode=(afAddrMode_t)Addr16Bit;
  SampleApp_ Unicast _DstAddr.endPoint = SAMPLEAPP_ENDPOINT;
  SampleApp_ Unicast _DstAddr.addr.shortAddr = 0x0000;
  if ( AF_DataRequest( & SampleApp_ Unicast _DstAddr,
                       &SampleApp_epDesc,
                       SAMPLEAPP_PERIODIC_CLUSTERID,
                       4,
                       data,
                       &SampleApp_TransID,
                       AF_DISCV_ROUTE,
                       AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
}
```

由于协调器不能对自己单播，需要将网络状态改变事件中协调器的状态变化注释掉，具体实现如图 4-18 所示。

```
281          case AF_INCOMING_MSG_CMD:
282            SampleApp_MessageMSGCB( MSGpkt );
283            break;
284
285          // Received whenever the device changes state in the network
286          case ZDO_STATE_CHANGE:
287            SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
288            if ( //(SampleApp_NwkState == DEV_ZB_COORD) 协调器不能给自己单播
289                 (SampleApp_NwkState == DEV_ROUTER)
290              || (SampleApp_NwkState == DEV_END_DEVICE) )
291            {
292              // Start sending the periodic message in a regular interval.
293             // osal_start_timerEx( SampleApp_TaskID,
294                    //           SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
295                          //     SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
296            }
297            else
298            {
299              // Device is no longer in the network
300            }
301            break;
302
```

图 4-18　注释协调器网络状态改变事件

至此，我们已经实现了 ZigBee 单播的数据通信方式，将程序代码分别下载到协调器、路由器和终端节点三个模块中，并分别连接到 PC 机的串口调试助手上，可以发现只有协调器能够收到数据，路由器和终端节点无法收到数据，如图 4-19 至图 4-21 所示。组播通信方式的实现方法类似，感兴趣的读者可以参考单播方式自行修改协议栈代码实现。

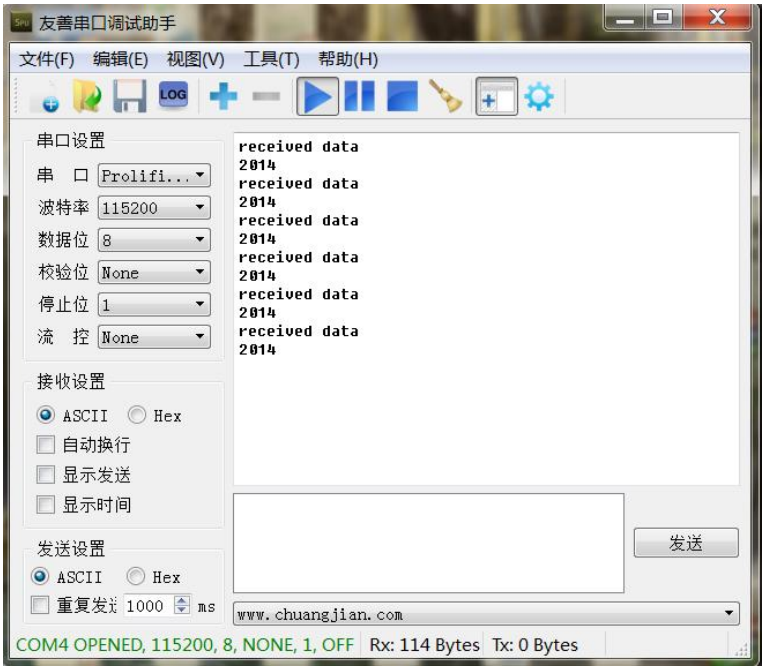

图 4-19　协调器数据接收界面

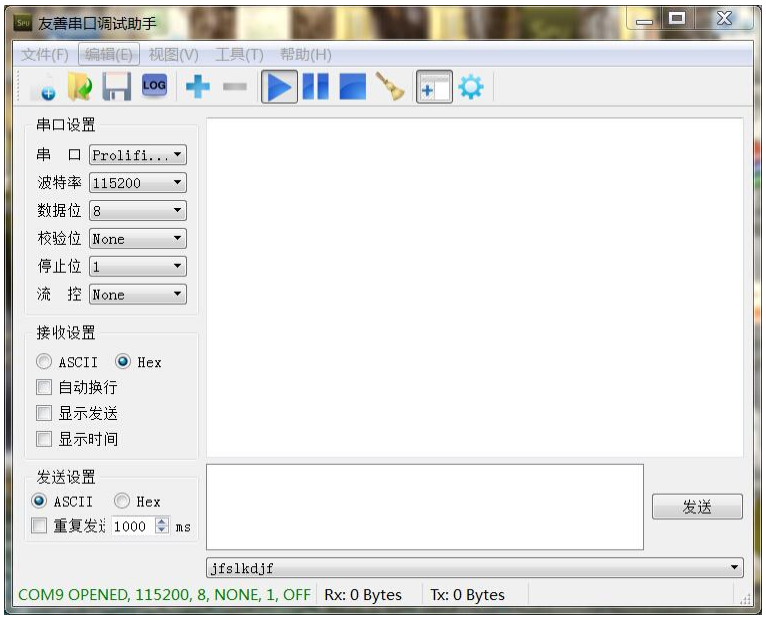

图 4-20　路由器无法接收到数据

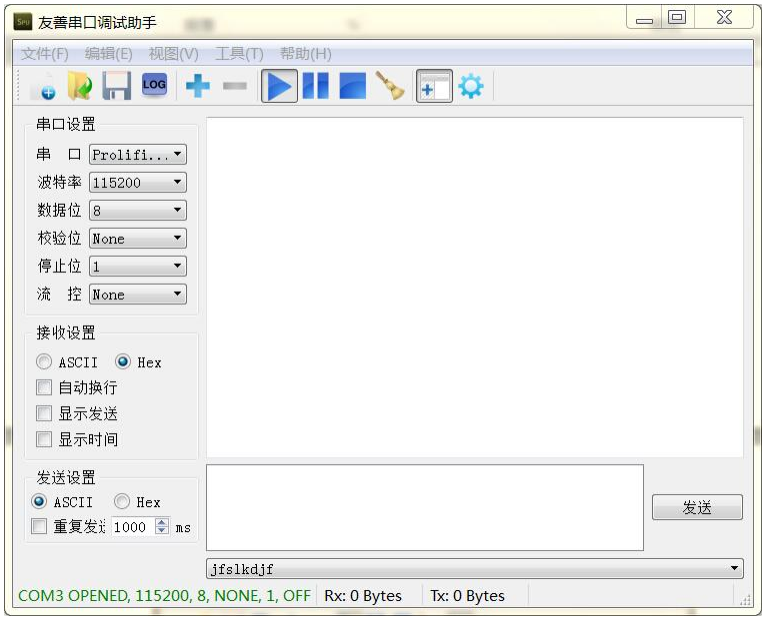

图 4-21　终端节点无法接收到数据

4.3.3　ZigBee 串口无线透传功能的实现

在 ZigBee 协议栈中串口的数据接收有特定的格式，使用时较为麻烦。本节内容将在前面串口初始化配置的基础上，实现串口的透明传输，所谓串口透传就是不管所传输的数据内容、数据的协议形式，只是把需要传输的内容当成一组二进制数据完美地传输到接收端，不对要传的数据做任何处理。透明传输不用关心下层协议的传输，比如你要寄信，只需要写地址交给邮局就行了，然后对方就能收到你的信，但是中途经过多少车站、邮递员是谁，你根本不知道，所以对于你来说邮递的过程是透明的。

在 MT 层的 MT_UART.c 文件中找到 MT_UartProcessZToolData()函数，这里存放的就是 UART 串口接收数据的代码，下面直接进行修改从而实现串口数据的透明接收，并使得发往上层的串口数据包简化为“数据长度+数据”的格式。代码如程序清单 4.23 所示。

程序清单 4.23

```
void MT_UartProcessZToolData ( uint8 port, uint8 event )
{
  uint8 flag=0,j,k=0;                    //flag 用于判断有没有收到串口数据，k 记录数据长度
  uint8 buf[128];                        //定义串口缓冲区大小，这里使用默认最大的 128
   (void)event;                          // Intentionally unreferenced parameter
   while (Hal_UART_RxBufLen(port))       //检测串口数据是否接收完成
   {
    HalUARTRead (port,&buf[k], 1);       //读取串口数据放到缓冲区 buf 中
    k++;                                 //记录数据字符个数
    flag=1;                              //接收标志变为 1，已经从串口接收到信息
```

```
            }
            if(flag==1)            //若已经接收完一帧数据
            {
      /* Allocate memory for the data */
              //分配内存空间：分别为结构体内容、数据内容和 1 个记录长度的数据
              pMsg = (mtOSALSerialData_t *)osal_msg_allocate( sizeof
                      ( mtOSALSerialData_t )+k+1);
               //事件号使用 SPI_INCOMING_ZTOOL_PORT
              pMsg->hdr.event = SPI_INCOMING_ZTOOL_PORT;
              pMsg->msg = (uint8*)(pMsg+1);       //把数据定位到结构体数据部分
              pMsg->msg [0]= k;                   //给上层的第一个数据是长度 k
             for(j=0;j<k;j++)                     //从第二个开始记录串口接收到的数据
             pMsg->msg [j+1]= buf[j];
         /*将包含该消息的指针发送到 App_TaskID 任务 ID 中，使得在下一次轮询时能够检测到此状态的变化，执行相应的
后续操作*/
             osal_msg_send( App_TaskID, (byte *)pMsg );
             osal_msg_deallocate ( (uint8 *)pMsg );          //释放内存
          }
       }
```

MT_UartProcessZToolData()函数经过简化后，主要流程如图 4-22 所示。

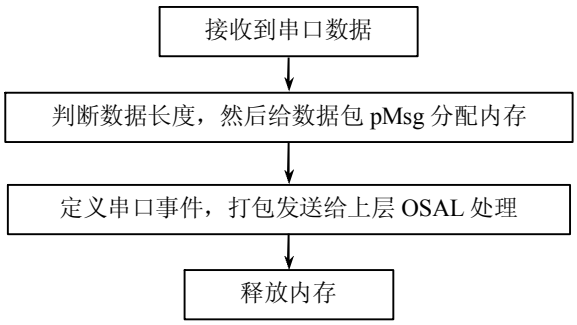

图 4-22　简化后的 MT_UartProcessZToolData 函数工作流程

现在我们已经实现了 ZigBee 协调器向 PC 机发送数据的功能和通过串口透传实现 PC 机向 ZigBee 协调器发送数据的功能，串口的无线透传功能要求协调器接收 PC 机串口发送的相关指令数据，并发送给终端节点，终端节点收到数据后判断是否驱动执行相应机构动作，以实现上位机无线控制下位机节点的功能。这就是 ZigBee 网络无线串口透传的功能。

下面将在本地串口透传的基础上将上位机发送的指令数据通过无线串口发送函数广播到所有的终端节点。

在 SampleApp.c 文件中找到任务处理函数 uint16 SampleApp_ProcessEvent(uint8 task_id, uint16 events)，在里面添加串口透传事件，如程序清单 4.24 所示。

程序清单 4.24

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

```
    afIncomingMSGPacket_t *MSGpkt;
    (void)task_id;                          // Intentionally unreferenced parameter

    if ( events & SYS_EVENT_MSG )
    {
      MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );
      while ( MSGpkt )
      {
        switch ( MSGpkt->hdr.event )        //读取事件
        {
          case SPI_INCOMING_ZTOOL_PORT:                          //串口透传事件
          SampleApp_SerialTRANS ((mtOSALSerialData_t *)MSGpkt);  //无线串口发送
          break;

          case KEY_CHANGE:
          SampleApp_HandleKeys( ((keyChange_t *)MSGpkt)->state, ((keyChange_t *)MSGpkt)->keys );
              break;
        ……
    }
```

SPI_INCOMING_ZTOOL_PORT 为串口事件号，串口收到数据后由 MT_UART 层传递过来的数据，在由 MT_UART 层直接传递到应用层，数据包的格式为数据长度（datalen）+数据（data）。

接下来就是处理接收到的数据包，这里要把它再原封不动地发送出去，串口事件发送函数如程序清单 4.25 所示。

程序清单 4.25

```
void SampleApp_SerialTRANS (mtOSALSerialData_t *cmdMsg)
{
 uint8 k,len,*str=NULL;
 str=cmdMsg->msg;
 len=*str;                  //msg 里的第 1 个字节代表后面的数据长度
  //无线串口数据广播发送
  if ( AF_DataRequest( &SampleApp_Periodic_DstAddr,
&SampleApp_epDesc,
                       SAMPLEAPP_SERIAL_CLUSTERID,
                       len+1,                  //发送数据包长度
                       str,                    //数据内容
                       &SampleApp_TransID,
                       AF_DISCV_ROUTE,
                       AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
  }
  else
  {
    //发送请求错误
  }

}
```

SAMPLEAPP_SERIAL_CLUSTERID 是定义的无线串口接收判别标志，我们在 SampleApp.h 头文件中加入该标志的定义，如程序清单 4.26 所示。

程序清单 4.26

```
#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_SERIAL_CLUSTERID 3
```

至此，上位机通过串口发送的数据已经被原封不动地发送出去了，接下来是无线串口数据的接收，在 SampleApp_MessageMSGCB()消息处理函数中增加无线串口透传接收代码，如程序清单 4.27 所示。

程序清单 4.27

```
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
  uint16 flashTime;
  uint8 k,len;
  switch ( pkt->clusterId )
  {
      case SAMPLEAPP_SERIAL_CLUSTERID:        //如果是无线串口透传数据
        len=pkt->cmd.Data[0];                 //第一个接收到的是数据的长度
        for(k=0;k<len;k++)
        {
HalUARTWrite(0,&pkt->cmd.Data[k+1],1);        //往串口写接收到的数据
}
        HalUARTWrite(0,“\n”,1);               //回车换行
        break;
    }
}
```

SAMPLEAPP_SERIAL_CLUSTERID 是发送端定义的无线串口传输标志，若在消息处理函数中接收到此标志，说明本次接收到的数据是无线串口透传数据，接下来就可以取出送到串口进行显示了。

将协议栈程序在串口初始化的基础上做以上修改后，分别下载到协调器和终端节点开发板中，并分别连接到 PC 机串口调试助手上，在协调器发送窗口内输入“Hello world!”，单击“发送”按钮；在终端节点串口调试助手内出现了“Hello world!”，如图 4-23 和图 4-24 所示，表明无线串口透传数据收发成功。

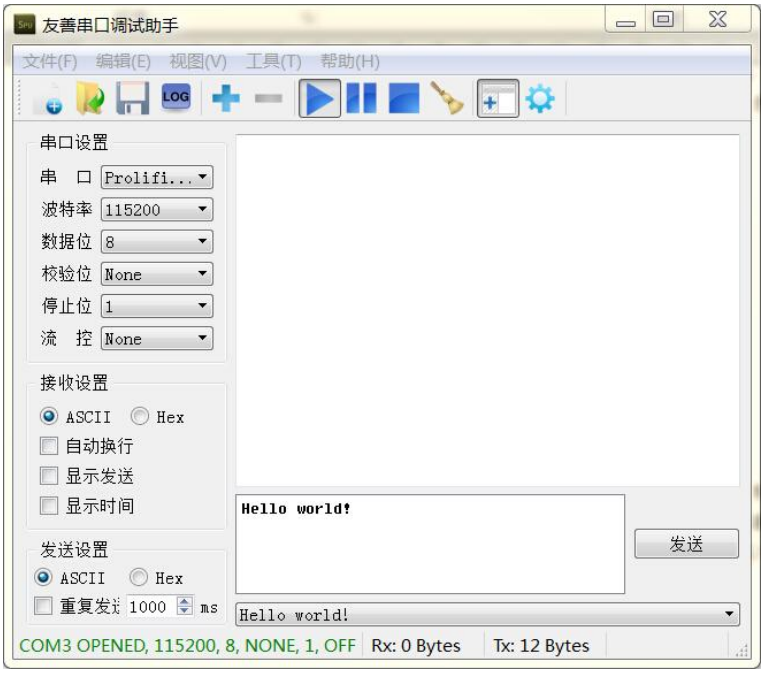

图 4-23　协调器通过串口无线发送“Hello world!”数据

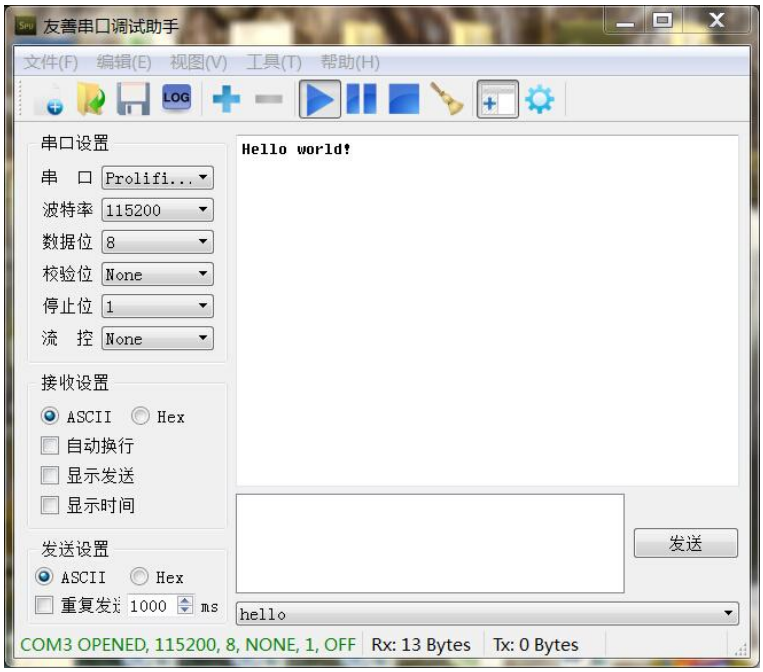

图 4-24　终端节点接收到协调器通过无线透传发送过来的数据

本章小结

本章通过几个典型的实验完成了几种基本的 ZigBee 无线数据通信，特别是广播和单播通信。此外，还重点介绍了在 ZigBee 通信中协议栈串口数据的收发实现方法。本章中所介绍的 ZigBee 数据通信的实现方法将在接下来的章节中反复涉及，希望读者能够深入理解并动手实践，以便能够快速上手后面的实战项目。