第一百三十四节：“应用层半双工”双机串口通讯的程序框架。

【134.1　应用层的“半双工”和“全双工”。】

应用层的“半双工”。主机与从机在程序应用层采用“一问一答”的查询模式，主机是主动方，从机是被动方，主机问一句从机答一句，“聊天对话“的氛围很无趣很呆板。从机没有发言权，当从机想主动给主机发送一些数据时就“憋得慌”。半双工适用于大多数单向通讯的场合。

应用层的“全双工”。主机与从机在程序应用层可以实现任意双向的通讯，这时从机也可变为主机，主机也可变为从机，就像两个人平时聊天，无所谓谁是从机谁是主机，也无所谓非要对方对我每句话都要应答附和（只要对方能听得清我讲什么就可以），“聊天对话”的氛围很生动很活泼。全双工适用于通讯更复杂的场合。

本节从“半双工“开始讲，让初学者先熟悉双机通讯的基本程序框架，下一节再讲“全双工“。

【134.2　双机通讯的三类核心函数。】

双机通讯在程序框架层面有三类核心的涵数，它们分别是：通讯过程的控制涵数，发送的队列驱动涵数，接收数据后的处理涵数。

“通讯过程的控制涵数”的数量可以不止 1 个，每一个通讯事件都对应一个独立的“通讯过程的控制涵数”，根据通讯事件的数量，一个系统往往有 N 个“通讯过程的控制涵数”。顾名思义，它负责过程的控制，无论什么项目，凡是过程控制我都首选 switch 语句。此函数是属于上层应用的函数，它的基础底层是“发送的队列驱动涵数”和“接收数据后的处理涵数”这两个函数。

“发送的队列驱动涵数”在系统中只有 1 个“发送的队列驱动涵数”，负责“通讯管道的占用”的分配，负责数据的具体发送。当同时存在很多“待发送”的请求指令时，此函数会根据“if ,else if...”的优先级，像队列一样安排各指令发送的先后顺序，确保各指令不会发生冲突。此函数属于底层的驱动函数。

“接收数据后的处理涵数”在系统中只有 1 个，负责处理当前接收到的数据，它既属于“底层函数”也属于“应用层函数”，二者成分皆有。

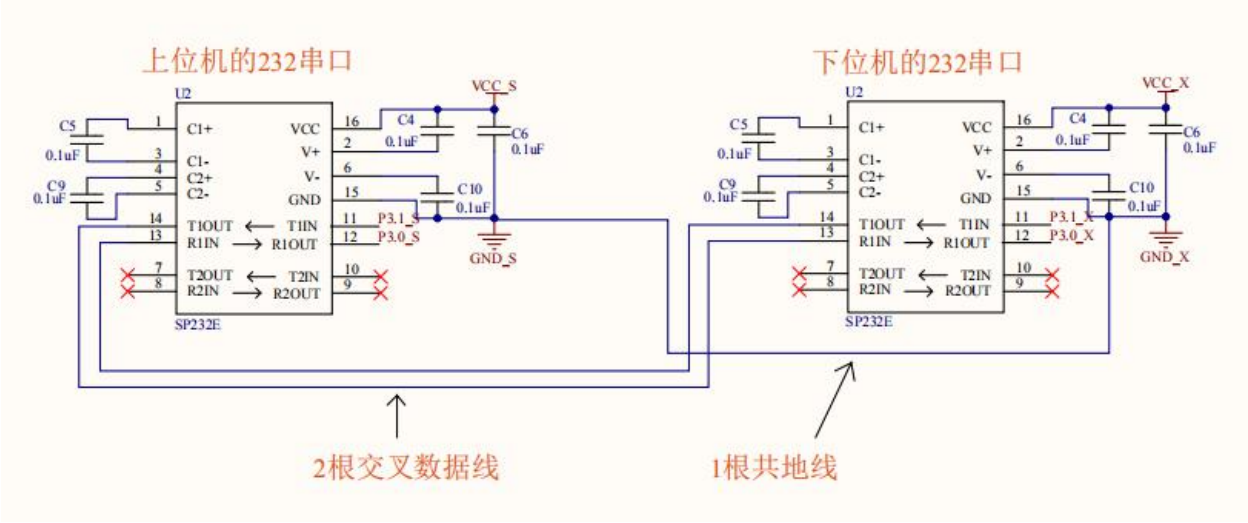
我们一旦深刻地领悟了这三类函数各自的分工与关联方式，将来应付再复杂的通讯系统都会脉络清析，游刃有余。

【134.3　例程的功能需求。】

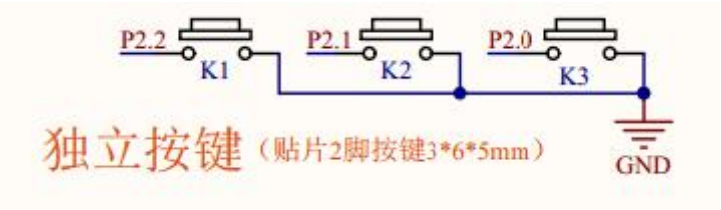
上位机与下位机都有一个一模一样的 57 个字节的大数组。在上位机端按下独立按键 K1 后，上位机开始与下位机建立通讯，上位机的目的是读取下位机的那 57 个字节的大数组，分批读取，每批读取 10 个字节，最后一批读取的是余下的 7 个字节。读取完毕后，上位机把读取到的大数组与自己的大数组进行对比：如果相等，表示通讯正确，蜂鸣器“长鸣”一声；如果不相等，表示通讯错误，蜂鸣器“短鸣”一声。在通讯过程中，如果出现通信异常（比如因为接收超时或者接收某批次数据错误而导致重发的次数超过最大限制的次数）也表示通讯错误，蜂鸣器也会发出“短鸣”一声的提示。

【134.4　例程的电路图。】

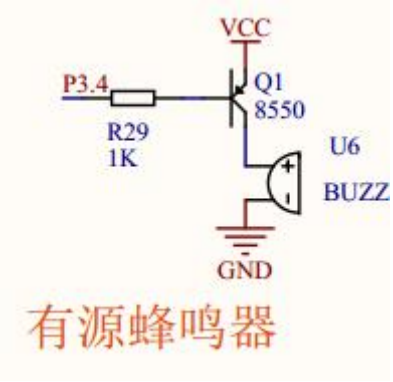
两个单片机进行 232 串口通讯，一共需要 3 根线：1 根作为共地线，其它 2 根是交叉的收发数据线（上位机的“接收线”连接下位机的“发送线”，上位机的“发送线”连接下位机的“接收线”），如下图所示：



上图 134.4.1　　双机通讯的 232 串口接线图



上图 134.4.2　上位机的独立按键



上图 134.4.3　上位机的有源蜂鸣器

【134.5　例程的通讯协议。】

（一）通讯参数。波特率 9600，校验位 NONE（无），数据位 8，停止位 1。

（二）上位机读取下位机的数组容量的大小的指令。

（1）上位机发送十六进制的数据：EB 01 00 00 00 07 ED。

EB 是数据头。

01 是指令类型，01 代表请求下位机返回大数组的容量大小。

00 00 00 07 代表整个指令的数据长度。
ED 是前面所有字节数据的异或结果，用来作为校验数据。

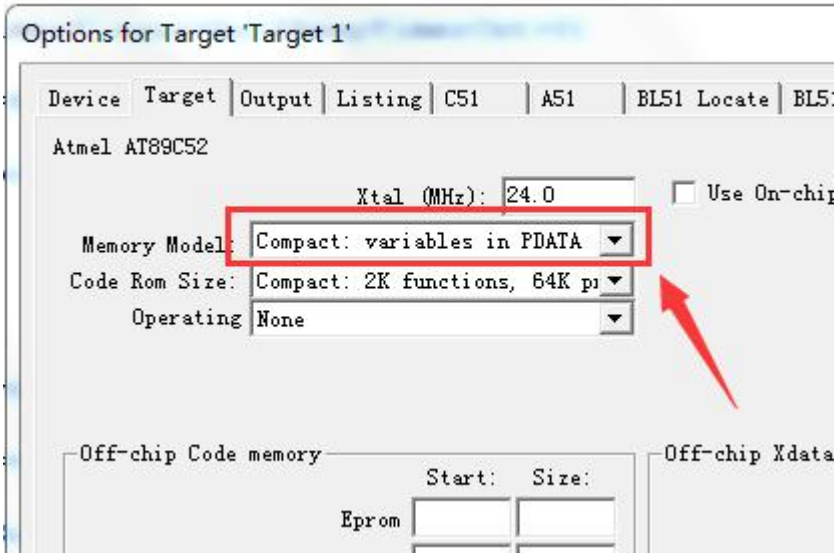
（2）下位机返回十六进制的数据：EB 01 00 00 00 0C XX XX XX XX ZZ。
EB 是数据头。
01 是指令类型，01 代表返回大数组的容量大小。
00 00 00 0B 代表整个指令的数据长度
XX XX XX XX 代表大数组的容量大小
ZZ 是前面所有字节数据的异或结果，用来作为校验数据。

（三）上位机读取下位机的大数组的分段数据的指令。
（1）上位机发送十六进制的数据：EB 02 00 00 00 0F RR RR RR RR YY YY YY YY ZZ
EB 是数据头
02 是指令类型，02 代表请求下位机返回当前分段的数据。
00 00 00 0F 代表整个指令的数据长度
RR RR RR RR 代表请求下位机返回的数据的“请求起始地址”
YY YY YY YY 代表请求下位机从“请求起始地址”一次返回的数据长度
ZZ 是前面所有字节数据的异或结果，用来作为校验数据。

（2）下位机返回十六进制的数据：EB 02 TT TT TT TT RR RR RR RR YY YY YY YY HH ...HH ZZ
EB 是数据头
02 是指令类型，02 代表返回大数组当前分段的数据
TT TT TT TT 代表整个指令的数据长度
RR RR RR RR 代表下位机返回数据时的“请求起始地址”
YY YY YY YY 代表下位机从“请求起始地址”一次返回的数据长度
HH ...HH 代表中间有效的数据内容
ZZ 是前面所有字节数据的异或结果，用来作为校验数据。

【134.6　解决本节例程编译不过去的方法。】

因为本节用到的全局变量比较多，如果有初学者在编译的时候出现“error C249: 'DATA': SEGMENT TOO LARGE”的提示，请按下图的窗口提示来设置一下编译的环境。



上图 134.5.1　设置编译的环境

【134.7　例程的上位机程序。】

```
#include "REG52.H"

#define RECE_TIME_OUT    2000  //通讯过程中字节之间的超时时间 2000ms
#define REC_BUFFER_SIZE  30    //常规控制类数组的长度
#define KEY_FILTER_TIME  25    //按键滤波的“稳定时间”

void usart(void);  //串口接收的中断函数
void T0_time();    //定时器的中断函数

void BigBufferUsart(void);  //读取下位机大数组的“通讯过程的控制涵数”。三大核心函数之一
void QueueSend(void);       //发送的队列驱动涵数。三大核心函数之一
void ReceDataHandle(void);  //接收数据后的处理涵数。三大核心函数之一

void UsartTask(void);    //串口收发的任务函数，放在主函数内

unsigned char CalculateXor(const unsigned char *pCu8Buffer, //异或的算法函数
                           unsigned long u32BufferSize);

//比较两个数组的是否相等。返回 1 代表相等，返回 0 代表不相等
//u32BufferSize 是参与对比的数组的大小
unsigned char CmpTwoBufferIsSame(const unsigned char *pCu8Buffer_1,
                                 const unsigned char *pCu8Buffer_2,
                                 unsigned long u32BufferSize);

void UsartSendByteData(unsigned char u8SendData); //发送一个字节的底层驱动函数
```

```
//发送带协议的函数
void UsartSendMessage(const unsigned char *pCu8SendMessage,unsigned long u32SendMaxSize);

void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);
void KeyScan(void);
void KeyTask(void);

sbit P3_4=P3^4;         //蜂鸣器的驱动输出口
sbit KEY_INPUT1=P2^2;  //K1 按键识别的输入口。

//下面表格数组的数据与下位机的表格数据一模一样，目的用来检测接收到的数据是否正确
code unsigned char Cu8TestTable[]=
{
0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,
0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,
0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2A,
0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,
0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A,
0x51,0x52,0x53,0x54,0x55,0x56,0x57
};

unsigned char Gu8ReceTable[57]; //从下位机接收到的表格数据的数组

//把一些针对某个特定事件的全局变量放在一个结构体内，可以让全局变量的分类更加清晰
struct StructBigBufferUsart     //控制读取大数组的通讯过程的结构体
{
unsigned char u8Status; //通讯过程的状态 0 为初始状态 1 为通讯成功 2 为通讯失败
unsigned char u8ReSendCnt; //重发计数器
unsigned char u8Step;   //通讯过程的步骤
unsigned char u8Start;  //通讯过程的启动
unsigned long u32NeedSendSize;    //一共需要发送的全部数据量
unsigned long u32AlreadySendSize; //实际已经发送的数据量
unsigned long u32CurrentAddr; //当前批次需要发送的起始地址
unsigned long u32CurrentSize; //当前批次从起始地址开始发送的数据量
unsigned char u8QueueSendTrig;//队列驱动函数的发送的启动
unsigned char u8QueueSendBuffer[30]; //队列驱动函数的发送指令的数组
unsigned char u8QueueStatus; //队列驱动函数的通讯状态 0 为初始状态 1 为通讯成功 2 为通讯失败
```

```
};

unsigned char Gu8QueueReceUpdate=0;  //1 代表“队列发送数据后，收到了新的数据”

struct StructBigBufferUsart  GtBigBufferUsart;//此结构体变量专门用来控制读取大数组的通讯事件

volatile unsigned char vGu8BigBufferUsartTimerFlag=0;  //过程控制的超时定时器
volatile unsigned int vGu16BigBufferUsartTimerCnt=0;

volatile unsigned char vGu8QueueSendTimerFlag=0;  //队列发送的超时定时器
volatile unsigned int vGu16QueueSendTimerCnt=0;

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8KeySec=0;

unsigned char Gu8SendByteFinish=0; //发送一个字节完成的标志

unsigned char Gu8ReceBuffer[REC_BUFFER_SIZE]; //常规控制类的小内存
unsigned char *pGu8ReceBuffer;  //用来切换接收内存的“中转指针”

unsigned long Gu32ReceCntMax=REC_BUFFER_SIZE;  //最大缓存
unsigned long Gu32ReceCnt=0;  //接收缓存数组的下标
unsigned char Gu8ReceStep=0;  //接收中断函数里的步骤变量
unsigned char Gu8ReceFeedDog=1; //“喂狗”的操作变量。
unsigned char Gu8ReceType=0; //接收的数据类型
unsigned char Gu8Rece_Xor=0; //接收的异或
unsigned long Gu32ReceDataLength=0;  //接收的数据长度
unsigned char Gu8FinishFlag=0;  //是否已接收完成一串数据的标志
unsigned long *pu32Data; //用于数据转换的指针
volatile unsigned char vGu8ReceTimeOutFlag=0;//通讯过程中字节之间的超时定时器的开关
volatile unsigned int vGu16ReceTimeOutCnt=0; //通讯过程中字节之间的超时定时器，“喂狗”的对象

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        UsartTask();   //串口收发的任务函数
        KeyTask();
    }
```

```
}

void KeyTask(void)    //按键任务函数，放在主函数内
{
    if(0==vGu8KeySec)
    {
        return; //按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8KeySec) //根据不同的按键触发序号执行对应的代码
    {
       case 1:     //1 号按键。K1 的独立按键
            //GtBigBufferUsart.u8Start 在开机初始化函数里必须初始化为 0!这一步很关键!
            if(0==GtBigBufferUsart.u8Start) //只有在还没有启动的情况下，才能启动
            {
                GtBigBufferUsart.u8Status=0; //通讯过程的状态 0 为初始状态
                GtBigBufferUsart.u8Step=0;   //通讯过程的步骤 0 为从当前开始的步骤
                GtBigBufferUsart.u8Start=1;  //通讯过程的启动
            }
            vGu8KeySec=0;  //响应按键服务处理程序后，按键编号必须清零，避免一致触发
            break;
    }
}

/* 注释一：
*  每一个通讯事件都对应的一个独立的“通讯过程的控制涵数”,一个系统中有多少个通讯事件，就存在
*  多少个“通讯过程的控制涵数”。该函数负责某个通讯事件从开始到结束的整个过程。比如本节项目，
*  在通讯过程中，如果发现接收到的数据错误，则继续启动重发的机制。当发现接收到的累加字节数等于
*  预期想要接收的数量时，则结束这个通讯的事件。
*/

void BigBufferUsart(void)  //读取下位机大数组的“通讯过程的控制涵数”
{
    static const unsigned char SCu8ReSendCntMax=3; //重发的次数
    static unsigned long *pSu32Data; //用于数据与数组转换的指针

    switch(GtBigBufferUsart.u8Step)  //过程控制，我首选 switch 语句!
    {
        case 0:
             if(1==GtBigBufferUsart.u8Start) //通讯过程的启动
             {
                 //根据实际项目需要，在此第 0 步骤里可以添加一些初始化相关的数据
                  GtBigBufferUsart.u8ReSendCnt=0; //重发计数器清零
```

```
                GtBigBufferUsart.u8Step=1;   //切换到下一步
            }
            break;

//-----------先发送“读取下位机的数组容量的大小的指令”---------------------
//-----------EB 01 00 00 00 07 ED                    ---------------------
        case 1:
            GtBigBufferUsart.u8QueueSendBuffer[0]=0xeb; //数据头
            GtBigBufferUsart.u8QueueSendBuffer[1]=0x01; //数据类型 读取数组容量大小
            pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2];
            *pSu32Data=7; //数据长度  本条指令的数据总长是 7 个字节
    //异或算法的函数
    GtBigBufferUsart.u8QueueSendBuffer[6]=CalculateXor(GtBigBufferUsart.u8QueueSendBuffer,
                                                       6) ;  //最后一个字节不纳入计算

            //队列驱动函数的状态 0 为初始状态 1 为通讯成功 2 为通讯失败
            GtBigBufferUsart.u8QueueStatus=0; //队列驱动函数的通讯状态
            GtBigBufferUsart.u8QueueSendTrig=1;//队列驱动函数的发送的启动



            vGu8BigBufferUsartTimerFlag=0;
             vGu16BigBufferUsartTimerCnt=2000;
            vGu8BigBufferUsartTimerFlag=1;   //过程控制的超时定时器的启动

            GtBigBufferUsart.u8Step=2;   //切换到下一步
            break;

        case 2:  //发送之后，等待下位机返回的数据的状态
            if(1==GtBigBufferUsart.u8QueueStatus) //当前批次的接收到的数据成功
            {
                GtBigBufferUsart.u8ReSendCnt=0; //重发计数器清零
                GtBigBufferUsart.u32AlreadySendSize=0; //实际已经发送的数据量清零
                GtBigBufferUsart.u32CurrentAddr=0;  //当前批次需要发送的起始地址
                GtBigBufferUsart.u32CurrentSize=10; //从当前批次起始地址开始发送的数据量
                GtBigBufferUsart.u8Step=3;   //切换到下一步
            }
            else if(2==GtBigBufferUsart.u8QueueStatus) //当前批次的接收到的数据失败
            {
                GtBigBufferUsart.u8ReSendCnt++;
                if(GtBigBufferUsart.u8ReSendCnt>=SCu8ReSendCntMax) //大于最大的重发次数
                {
                    GtBigBufferUsart.u8Step=0;
                    GtBigBufferUsart.u8Start=0; //结束当前的过程通讯
                    GtBigBufferUsart.u8Status=2; //对外宣布“通讯失败”
```

```
                    vGu8BeepTimerFlag=0;
                    vGu16BeepTimerCnt=30;   //让蜂鸣器“短鸣”一声
                    vGu8BeepTimerFlag=1;
                }
                else
                {
                    GtBigBufferUsart.u8Step=1;   //返回上一步，重发当前段的数据
                }
            }
            else if(0==vGu16BigBufferUsartTimerCnt) //当前批次在等待接收返回数据时，超时
            {
                GtBigBufferUsart.u8ReSendCnt++;
                if(GtBigBufferUsart.u8ReSendCnt>=SCu8ReSendCntMax) //大于最大的重发次数
                {
                    GtBigBufferUsart.u8Step=0;
                    GtBigBufferUsart.u8Start=0; //结束当前的过程通讯
                    GtBigBufferUsart.u8Status=2; //对外宣布“通讯失败”

                    vGu8BeepTimerFlag=0;
                    vGu16BeepTimerCnt=30;   //让蜂鸣器“短鸣”一声
                    vGu8BeepTimerFlag=1;
                }
                else
                {
                    GtBigBufferUsart.u8Step=1;   //返回上一步，重发当前段的数据
                }
            }
            break;

//-----------接着发送“读取下位机的大数组的分段数据的指令”----------------------
//-----------EB 02 00 00 00 0F RR RR RR RR YY YY YY YY ZZ  ----------------------
        case 3:
            GtBigBufferUsart.u8QueueSendBuffer[0]=0xeb; //数据头
            GtBigBufferUsart.u8QueueSendBuffer[1]=0x02; //数据类型 读取分段数据
            pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2];
            *pSu32Data=15; //数据长度  本条指令的数据总长是 15 个字节

            pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2+4];
            *pSu32Data=GtBigBufferUsart.u32CurrentAddr; //当前批次需要发送的起始地址

            pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2+4+4];
            *pSu32Data=GtBigBufferUsart.u32CurrentSize; //从当前批次起始地址发送的数据量

   //异或算法的函数
```

```
GtBigBufferUsart.u8QueueSendBuffer[14]=CalculateXor(GtBigBufferUsart.u8QueueSendBuffer,
                                                    14);  //最后一个字节不纳入计算

        //队列驱动函数的状态 0 为初始状态 1 为通讯成功 2 为通讯失败
        GtBigBufferUsart.u8QueueStatus=0; //队列驱动函数的通讯状态
        GtBigBufferUsart.u8QueueSendTrig=1;//队列驱动函数的发送的启动

        vGu8BigBufferUsartTimerFlag=0;
         vGu16BigBufferUsartTimerCnt=2000;
        vGu8BigBufferUsartTimerFlag=1;   //过程控制的超时定时器的启动

        GtBigBufferUsart.u8Step=4;   //切换到下一步
        break;

    case 4: //发送之后，等待下位机返回的数据的状态
        if(1==GtBigBufferUsart.u8QueueStatus) //当前批次的接收到的数据成功
        {
            //更新累加当前实际已经发送的字节数
            GtBigBufferUsart.u32AlreadySendSize=GtBigBufferUsart.u32AlreadySendSize+
                                                GtBigBufferUsart.u32CurrentSize;

            //更新下一步起始的发送地址
            GtBigBufferUsart.u32CurrentAddr=GtBigBufferUsart.u32CurrentAddr+
                                            GtBigBufferUsart.u32CurrentSize;

            //更新下一步从起始地址开始发送的字节数
            if((GtBigBufferUsart.u32CurrentAddr+GtBigBufferUsart.u32CurrentSize)>
                GtBigBufferUsart.u32NeedSendSize) //最后一段数据的临界点的判断
            {
                GtBigBufferUsart.u32CurrentSize=GtBigBufferUsart.u32NeedSendSize-
                                                GtBigBufferUsart.u32CurrentAddr;
            }
            else
            {
                GtBigBufferUsart.u32CurrentSize=10;
            }

            //判断是否已经把整个大数组的 57 个字节都已经接收完毕。如果已经接收完毕，则
            //结束当前通信；如果还没结束，则继续请求下位机发送下一段新数据。
            if(GtBigBufferUsart.u32AlreadySendSize>=GtBigBufferUsart.u32NeedSendSize)
            {
                GtBigBufferUsart.u8Step=0;
                GtBigBufferUsart.u8Start=0; //结束当前的过程通讯
```

```
            if(1==CmpTwoBufferIsSame(Cu8TestTable,     //如果接收的数据与存储的相等
                                     Gu8ReceTable,
                                     57))
            {
                vGu8BeepTimerFlag=0;
                vGu16BeepTimerCnt=1000;   //让蜂鸣器“长鸣”一声
                vGu8BeepTimerFlag=1;
                GtBigBufferUsart.u8Status=1; //对外宣布“通讯成功”
            }
            else
            {
                vGu8BeepTimerFlag=0;
                vGu16BeepTimerCnt=30;   //让蜂鸣器“短鸣”一声
                vGu8BeepTimerFlag=1;
                GtBigBufferUsart.u8Status=2; //对外宣布“通讯失败”
            }

        }
        else
        {
            GtBigBufferUsart.u8ReSendCnt=0; //重发计数器清零
            GtBigBufferUsart.u8Step=3;   //返回上一步，继续发下一段的新数据
        }

    }
    else if(2==GtBigBufferUsart.u8QueueStatus) //当前批次的接收到的数据失败
    {
        GtBigBufferUsart.u8ReSendCnt++;
        if(GtBigBufferUsart.u8ReSendCnt>=SCu8ReSendCntMax) //大于最大的重发次数
        {
            GtBigBufferUsart.u8Step=0;
            GtBigBufferUsart.u8Start=0; //结束当前的过程通讯
            GtBigBufferUsart.u8Status=2; //对外宣布“通讯失败”

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=30;   //让蜂鸣器“短鸣”一声
            vGu8BeepTimerFlag=1;
        }
        else
        {
            GtBigBufferUsart.u8Step=3;   //返回上一步，重发当前段的数据
        }
    }
    else if(0==vGu16BigBufferUsartTimerCnt) //当前批次在等待接收返回数据时，超时
```

```
            {
                GtBigBufferUsart.u8ReSendCnt++;
                if(GtBigBufferUsart.u8ReSendCnt>=SCu8ReSendCntMax) //大于最大的重发次数
                {
                    GtBigBufferUsart.u8Step=0;
                    GtBigBufferUsart.u8Start=0; //结束当前的过程通讯
                    GtBigBufferUsart.u8Status=2; //对外宣布“通讯失败”

                    vGu8BeepTimerFlag=0;
                    vGu16BeepTimerCnt=30;   //让蜂鸣器“短鸣”一声
                    vGu8BeepTimerFlag=1;
                }
                else
                {
                    GtBigBufferUsart.u8Step=3;   //返回上一步，重发当前段的数据
                }
            }
            break;
    }

}

/* 注释二：
*  整个项目中只有一个“发送的队列驱动涵数”，负责“通讯管道的占用”的分配，负责数据的具体发
*  送。当同时存在很多“待发送”的请求指令时，此函数会根据“if ,else if...”的优先级，像队列一
*  样安排各指令发送的先后顺序，确保各指令不会发生冲突。
*/

void QueueSend(void)  //发送的队列驱动涵数
{
    static unsigned char Su8Step=0;

    switch(Su8Step)
    {
       case 0:  //分派即将要发送的任务
            if(1==GtBigBufferUsart.u8QueueSendTrig)
            {
                GtBigBufferUsart.u8QueueSendTrig=0;  //及时清零。驱动层，不管结果，只发一次。

                Gu8QueueReceUpdate=0; //接收应答数据的状态恢复初始值

                //发送带指令的数据
                UsartSendMessage((const unsigned char *)&GtBigBufferUsart.u8QueueSendBuffer[0],
                                 30);
```

```
                vGu8QueueSendTimerFlag=0;
                vGu16QueueSendTimerCnt=2000;
                vGu8QueueSendTimerFlag=1;  //队列发送的超时定时器
                Su8Step=1;
            }
    //      else if(...)  //当有其它发送的指令时，可以在此处继续添加判断，越往下优先级越低
    //      else if(...)  //当有其它发送的指令时，可以在此处继续添加判断，越往下优先级越低

            break;

       case 1: //发送之后，等待下位机的应答。驱动层，只管有没有应答，不管应答对不对。
            if(1==Gu8QueueReceUpdate) //如果“接收数据后的处理涵数”接收到应答数据
            {
                Su8Step=0;  //返回上一步继续处理其它“待发送的指令”
            }

            if(0==vGu16QueueSendTimerCnt) //发送指令之后，等待应答超时
            {
                Su8Step=0;  //返回上一步继续处理其它“待发送的指令”
            }

            break;
    }
}



/* 注释三：
*  整个项目中只有一个“接收数据后的处理涵数”，负责即时处理当前接收到的数据。
*/

void ReceDataHandle(void)  //接收数据后的处理涵数
{
    static unsigned long *pSu32Data; //数据转换的指针
    static unsigned long i;
    static unsigned char Su8Rece_Xor=0;  //计算的“异或”

    static unsigned long Su32CurrentAddr; //读取的起始地址
    static unsigned long Su32CurrentSize; //读取的发送的数据量

    if(1==Gu8ReceFeedDog) //每被“喂一次狗”，就及时更新一次“超时检测的定时器”的初值
    {
        Gu8ReceFeedDog=0;
```

```
    vGu8ReceTimeOutFlag=0;
    vGu16ReceTimeOutCnt=RECE_TIME_OUT;//更新一次“超时检测的定时器”的初值
    vGu8ReceTimeOutFlag=1;
}
else if(Gu8ReceStep>0&&0==vGu16ReceTimeOutCnt) //超时，并且步骤不在接头暗号的步骤
{
    Gu8ReceStep=0; //串口接收数据的中断函数及时切换回接头暗号的步骤
}

if(1==Gu8FinishFlag)  //1 代表已经接收完毕一串新的数据，需要马上去处理
{
    switch(Gu8ReceType)  //接收到的数据类型
    {
        case 0x01:   //读取下位机的数组容量的大小
             Gu8QueueReceUpdate=1;  //告诉“队列驱动函数”收到了新的应答数据

             Gu8Rece_Xor=Gu8ReceBuffer[Gu32ReceDataLength-1];  //提取接收到的“异或”
             Su8Rece_Xor=CalculateXor(Gu8ReceBuffer,Gu32ReceDataLength-1); //计算“异或”

             if(Gu32ReceDataLength>=11&& //接收到的数据长度必须大于或者等于 11 个字节
               Su8Rece_Xor==Gu8Rece_Xor) //验证“异或”，“计算的”与“接收的”是否一致
             {
                pSu32Data=(unsigned long *)&Gu8ReceBuffer[6]; //数据转换。
                GtBigBufferUsart.u32NeedSendSize=*pSu32Data;  //提取将要接收数组的大小
                GtBigBufferUsart.u8QueueStatus=1; //告诉“过程控制函数”，当前通讯成功
             }
             else
             {
                GtBigBufferUsart.u8QueueStatus=2; //告诉“过程控制函数”，当前通讯失败
             }

             break;

        case 0x02:   //读取下位机的分段数据
             Gu8QueueReceUpdate=1;  //告诉“队列驱动函数”收到了新的应答数据

             Gu8Rece_Xor=Gu8ReceBuffer[Gu32ReceDataLength-1];  //提取接收到的“异或”
             Su8Rece_Xor=CalculateXor(Gu8ReceBuffer,Gu32ReceDataLength-1); //计算“异或”

             pSu32Data=(unsigned long *)&Gu8ReceBuffer[6]; //数据转换。
             Su32CurrentAddr=*pSu32Data;  //读取的起始地址

             pSu32Data=(unsigned long *)&Gu8ReceBuffer[6+4]; //数据转换。
             Su32CurrentSize=*pSu32Data;  //读取的发送的数据量
```

```
                if(Gu32ReceDataLength>=11&& //接收到的数据长度必须大于或者等于 11 个字节
                Su8Rece_Xor==Gu8Rece_Xor&& //验证“异或”，“计算的”与“接收的”是否一致
    Su32CurrentAddr==GtBigBufferUsart.u32CurrentAddr&& //验证“地址”，相当于验证“动态密匙”
    Su32CurrentSize==GtBigBufferUsart.u32CurrentSize) //验证“地址”，相当于验证“动态密匙”
                {
                    for(i=0;i<Su32CurrentSize;i++)
                    {
                          //及时把接收到的数据存储到 Gu8ReceTable 数组
                         Gu8ReceTable[Su32CurrentAddr+i]=Gu8ReceBuffer[6+4+4+i];
                    }

                    GtBigBufferUsart.u8QueueStatus=1; //告诉“过程控制函数”，当前通讯成功
                }
                else
                {
                    GtBigBufferUsart.u8QueueStatus=2; //告诉“过程控制函数”，当前通讯失败
                }

                break;
        }

        Gu8FinishFlag=0;  //上面处理完数据再清零标志，为下一次接收新的数据做准备
    }
}

void UsartTask(void)    //串口收发的任务函数，放在主函数内
{
    BigBufferUsart();  //读取下位机大数组的“通讯过程的控制涵数”
    QueueSend();       //发送的队列驱动涵数
    ReceDataHandle();  //接收数据后的处理涵数
}

void usart(void) interrupt 4   //串口接发的中断函数，中断号为 4
{
   if(1==RI)  //接收完一个字节后引起的中断
   {
        RI = 0; //及时清零，避免一直无缘无故的进入中断。

       if(0==Gu8FinishFlag)  //1 代表已经完成接收了一串新数据，并且禁止接收其它新的数据
       {
          Gu8ReceFeedDog=1; //每接收到一个字节的数据，此标志就置 1 及时更新定时器的值。
          switch(Gu8ReceStep)
          {
```

```
case 0:     //“前部分的”数据头。接头暗号的步骤。
     Gu8ReceBuffer[0]=SBUF; //直接读取刚接收完的一个字节的数据。
     if(0xeb==Gu8ReceBuffer[0])  //等于数据头 0xeb，接头暗号吻合。
     {
        Gu32ReceCnt=1; //接收缓存的下标
        Gu8ReceStep=1;  //切换到下一个步骤，接收其它有效的数据
     }
     break;

case 1:     //“前部分的”数据类型和长度
     Gu8ReceBuffer[Gu32ReceCnt]=SBUF; //直接读取刚接收完的一个字节的数据。
     Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备
     if(Gu32ReceCnt>=6)  //前 6 个数据。接收完了“数据类型”和“数据长度”。
     {
          Gu8ReceType=Gu8ReceBuffer[1];  //提取“数据类型”
           //以下的数据转换，在第 62 节讲解过的指针法
          pu32Data=(unsigned long *)&Gu8ReceBuffer[2]; //数据转换
          Gu32ReceDataLength=*pu32Data; //提取“数据长度”
          if(Gu32ReceCnt>=Gu32ReceDataLength) //靠“数据长度”来判断是否完成
          {
              Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
              Gu8ReceStep=0;   //及时切换回接头暗号的步骤
          }
          else   //如果还没结束，继续切换到下一个步骤，接收“有效数据”
          {
              //本节只用到一个接收数组，把指针关联到 Gu8ReceBuffer 本身的数组
               pGu8ReceBuffer=(unsigned char *)&Gu8ReceBuffer[6];
               Gu32ReceCntMax=REC_BUFFER_SIZE;  //最大缓存

               Gu8ReceStep=2;   //切换到下一个步骤
          }
     }
     break;
case 2:     //“后部分的”数据
     pGu8ReceBuffer[Gu32ReceCnt-6]=SBUF; //这里的指针就是各种不同内存的化身!!!
     Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备

     //靠“数据长度”来判断是否完成。也不允许超过数组的最大缓存的长度
     if(Gu32ReceCnt>=Gu32ReceDataLength||Gu32ReceCnt>=Gu32ReceCntMax)
     {
        Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
        Gu8ReceStep=0;   //及时切换回接头暗号的步骤
     }
     break;
```

```
            }
        }
    }
    else  //发送数据引起的中断
    {
        TI = 0;  //及时清除发送中断的标志，避免一直无缘无故的进入中断。
        Gu8SendByteFinish=1; //从 0 变成 1 通知主函数已经发送完一个字节的数据了。
    }
}

void UsartSendByteData(unsigned char u8SendData) //发送一个字节的底层驱动函数
{
        static unsigned int Su16TimeOutDelay;  //超时处理的延时计时器

        Gu8SendByteFinish=0;  //在发送以字节之前，必须先把此全局变量的标志清零。
        SBUF =u8SendData; //依靠寄存器 SBUF 作为载体发送一个字节的数据
        Su16TimeOutDelay=0xffff;  //超时处理的延时计时器装载一个相对合理的计时初始值
        while(Su16TimeOutDelay>0)  //超时处理
        {
            if(1==Gu8SendByteFinish)
            {
                break;  //如果 Gu8SendByteFinish 为 1，则发送一个字节完成，退出当前循环等待。
            }
            Su16TimeOutDelay--;  //超时计时器不断递减
        }

        //Delay();//在实际应用中，当连续发送一堆数据时如果发现丢失数据，可以尝试在此增加延时
}

//发送带协议的函数
void UsartSendMessage(const unsigned char *pCu8SendMessage,unsigned long u32SendMaxSize)
{
        static unsigned long i;
        static unsigned long *pSu32;
        static unsigned long u32SendSize;

        pSu32=(const unsigned long *)&pCu8SendMessage[2];
        u32SendSize=*pSu32;  //从带协议的数组中提取整包数组的有效发送长度

        if(u32SendSize>u32SendMaxSize) //如果“有效发送长度”大于“最大限制的长度”,数据异常
        {
           return;  //数据异常，直接退出当前函数，预防数组越界
        }
```

```
        for(i=0;i<u32SendSize;i++) //u32SendSize 为发送的数据长度
        {
           UsartSendByteData(pCu8SendMessage[i]); //基于“发送单字节的最小接口函数”来实现的
        }
}

unsigned char CalculateXor(const unsigned char *pCu8Buffer, //此处加 const 代表数组“只读”
                           unsigned long u32BufferSize)  //参与计算的数组的大小
{
    unsigned long i;
    unsigned char Su8Rece_Xor;
    Su8Rece_Xor=pCu8Buffer[0]; //提取数据串第“i=0”个数据作为异或的原始数据
    for(i=1;i<u32BufferSize;i++) //注意，这里是从第“i=1”个数据开始
    {
        Su8Rece_Xor=Su8Rece_Xor^pCu8Buffer[i];   //计算“异或”
    }
    return Su8Rece_Xor;  //返回运算后的异或的计算结果
}

//比较两个数组的是否相等。返回 1 代表相等，返回 0 代表不相等
unsigned char CmpTwoBufferIsSame(const unsigned char *pCu8Buffer_1,
                                 const unsigned char *pCu8Buffer_2,
                                 unsigned long u32BufferSize)  //参与对比的数组的大小
{
    unsigned long i;
    for(i=0;i<u32BufferSize;i++)
    {
       if(pCu8Buffer_1[i]!=pCu8Buffer_2[i])
       {
          return 0; //只要有一个不相等，则返回 0 并且退出当前函数
        }
    }
    return 1; //相等
}


void KeyScan(void)  //此函数放在定时中断里每 1ms 扫描一次
{
   static unsigned char Su8KeyLock1; //1 号按键的自锁
   static unsigned int  Su16KeyCnt1; //1 号按键的计时器

   //1 号按键
   if(0!=KEY_INPUT1)//IO 是高电平，说明按键没有被按下，这时要及时清零一些标志位
   {
```

```
      Su8KeyLock1=0; //按键解锁
      Su16KeyCnt1=0; //按键去抖动延时计数器清零，此行非常巧妙，是全场的亮点。
   }
   else if(0==Su8KeyLock1)//有按键按下，且是第一次被按下。
   {
      Su16KeyCnt1++; //累加定时中断次数
      if(Su16KeyCnt1>=KEY_FILTER_TIME) //滤波的“稳定时间”KEY_FILTER_TIME，长度是 25ms。
      {
         Su8KeyLock1=1;  //按键的自锁,避免一直触发
         vGu8KeySec=1;    //触发 1 号键
      }
   }
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();

    if(1==vGu8BigBufferUsartTimerFlag&&vGu16BigBufferUsartTimerCnt>0) //过程控制的超时定时器
    {
         vGu16BigBufferUsartTimerCnt--;
    }

    if(1==vGu8QueueSendTimerFlag&&vGu16QueueSendTimerCnt>0) //队列发送的超时定时器
    {
         vGu16QueueSendTimerCnt--;
    }

    if(1==vGu8ReceTimeOutFlag&&vGu16ReceTimeOutCnt>0) //通讯过程中字节之间的超时定时器
    {
         vGu16ReceTimeOutCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}



void SystemInitial(void)
{
    unsigned char u8_TMOD_Temp=0;

    //以下是定时器 0 的中断的配置
```

```
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;

    //以下是串口接收中断的配置
    //串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
    u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
    TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
    TMOD=TMOD|u8_TMOD_Temp;  //把高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
    TH1=256-(11059200L/12/32/9600);  //波特率为 9600。11059200 代表晶振 11.0592MHz，
    TL1=256-(11059200L/12/32/9600);  //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。
    TR1=1;  //开启定时器 1

    SM0=0;
    SM1=1;  //SM0 与 SM1 的设置：选择 10 位异步通讯，波特率根据定时器 1 可变
    REN=1;  //允许串口接收数据

    //为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，
    //这个时候，串口中断可以打断任何其他的中断服务函数实现嵌套，
    IP =0x10;  //把串口中断设置为最高优先级，必须的。

    ES=1;          //允许串口中断
    EA=1;          //允许总中断
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{
    GtBigBufferUsart.u8Start=0;  //通讯过程的启动变量必须初始化为 0!这一步很关键!
}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
```

```
{
    P3_4=1;
}

void VoiceScan(void)
{

      static unsigned char Su8Lock=0;

      if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
      {
          if(0==Su8Lock)
          {
                Su8Lock=1;
                BeepOpen();
           }
          else
          {

                vGu16BeepTimerCnt--;

                if(0==vGu16BeepTimerCnt)
                {
                    Su8Lock=0;
                    BeepClose();
                }

          }
      }
}
```

【134.8　例程的下位机程序。】

下位机作为从机应答上位机的指令，程序相对简化了很多。不需要“通讯过程的控制涵数”，直接在“接收数据后的处理涵数”里启动“发送的队列驱动涵数”来发送应答的数据即可。发送应答数据后，也不用等待上位机的应答数据。

```
#include "REG52.H"

#define RECE_TIME_OUT    2000  //通讯过程中字节之间的超时时间 2000ms
#define REC_BUFFER_SIZE  30    //常规控制类数组的长度

void usart(void);  //串口接收的中断函数
```

```
void T0_time();    //定时器的中断函数

void QueueSend(void);        //发送的队列驱动涵数
void ReceDataHandle(void);  //接收数据后的处理涵数

void UsartTask(void);    //串口收发的任务函数，放在主函数内

unsigned char CalculateXor(const unsigned char *pCu8Buffer, //异或的算法的函数
                           unsigned long u32BufferSize);

void UsartSendByteData(unsigned char u8SendData); //发送一个字节的底层驱动函数

//发送带协议的函数
void UsartSendMessage(const unsigned char *pCu8SendMessage,unsigned long u32SendMaxSize);

void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

//下面表格数组的数据与上位机的表格数据一模一样，目的用来让上位机检测接收到的数据是否正确
code unsigned char Cu8TestTable[]=
{
0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,
0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,
0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2A,
0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,
0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A,
0x51,0x52,0x53,0x54,0x55,0x56,0x57
};

//把一些针对某个特定事件的全局变量放在一个结构体内，可以让全局变量的分类更加清晰
struct StructBigBufferUsart     //应答读取大数组的通讯过程的结构体
{
unsigned char u8QueueSendTrig;//队列驱动函数的发送的启动
unsigned char u8QueueSendBuffer[30]; //队列驱动函数的发送指令的数组
unsigned char u8QueueStatus; //队列驱动函数的通讯状态 0 为初始状态 1 为通讯成功 2 为通讯失败
};

unsigned char Gu8QueueReceUpdate=0;  //1 代表“队列发送数据后，收到了新的数据”

struct StructBigBufferUsart  GtBigBufferUsart;//此结构体变量专门用来应答读取大数组的通讯事件

volatile unsigned char vGu8QueueSendTimerFlag=0;  //队列发送的超时定时器
volatile unsigned int vGu16QueueSendTimerCnt=0;
```

```
unsigned char Gu8SendByteFinish=0; //发送一个字节完成的标志

unsigned char Gu8ReceBuffer[REC_BUFFER_SIZE]; //常规控制类的小内存
unsigned char *pGu8ReceBuffer;  //用来切换接收内存的“中转指针”

unsigned long Gu32ReceCntMax=REC_BUFFER_SIZE;  //最大缓存
unsigned long Gu32ReceCnt=0;  //接收缓存数组的下标
unsigned char Gu8ReceStep=0;  //接收中断函数里的步骤变量
unsigned char Gu8ReceFeedDog=1; //“喂狗”的操作变量。
unsigned char Gu8ReceType=0; //接收的数据类型
unsigned char Gu8Rece_Xor=0; //接收的异或
unsigned long Gu32ReceDataLength=0;  //接收的数据长度
unsigned char Gu8FinishFlag=0;  //是否已接收完成一串数据的标志
unsigned long *pu32Data; //用于数据转换的指针
volatile unsigned char vGu8ReceTimeOutFlag=0;//通讯过程中字节之间的超时定时器的开关
volatile unsigned int vGu16ReceTimeOutCnt=0; //通讯过程中字节之间的超时定时器，“喂狗”的对象

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
        UsartTask();   //串口收发的任务函数
    }
}

/* 注释一：
*  整个项目中只有一个“发送的队列驱动涵数”，负责“通讯管道的占用”的分配，负责数据的具体发
*  送。当同时存在很多“待发送”的请求指令时，此函数会根据“if ,else if...”的优先级，像队列一
*  样安排各指令发送的先后顺序，确保各指令不会发生冲突。
*/

void QueueSend(void)  //发送的队列驱动涵数
{
    static unsigned char Su8Step=0;

    switch(Su8Step)
    {
       case 0:  //分派即将要发送的任务
            if(1==GtBigBufferUsart.u8QueueSendTrig)
            {
```

```
                GtBigBufferUsart.u8QueueSendTrig=0;  //及时清零。驱动层，不管结果，只发一次。

                Gu8QueueReceUpdate=0; //接收应答数据的状态恢复初始值

                //发送带指令的数据
                UsartSendMessage((const unsigned char *)&GtBigBufferUsart.u8QueueSendBuffer[0],
                                 30);
                //注意，这里是从机应答主机的数据，不需要等待返回的数据，因此不需要切换 Su8Step
            }
    //      else if(...)  //当有其它发送的指令时，可以在此处继续添加判断，越往下优先级越低
    //      else if(...)  //当有其它发送的指令时，可以在此处继续添加判断，越往下优先级越低

            break;

       case 1: //发送之后，等待下位机的应答。驱动层，只管有没有应答，不管应答对不对。
            if(1==Gu8QueueReceUpdate) //如果“接收数据后的处理涵数”接收到应答数据
            {
                Su8Step=0;  //返回上一步继续处理其它“待发送的指令”
            }

            if(0==vGu16QueueSendTimerCnt) //发送指令之后，等待应答超时
            {
                Su8Step=0;  //返回上一步继续处理其它“待发送的指令”
            }

            break;
    }
}



/* 注释二：
*  整个项目中只有一个“接收数据后的处理涵数”，负责即时处理当前接收到的数据。
*/

void ReceDataHandle(void)  //接收数据后的处理涵数
{
    static unsigned long *pSu32Data; //数据转换的指针
    static unsigned long i;
    static unsigned char Su8Rece_Xor=0;  //计算的“异或”

    static unsigned long Su32CurrentAddr; //读取的起始地址
    static unsigned long Su32CurrentSize; //读取的发送的数据量

    if(1==Gu8ReceFeedDog) //每被“喂一次狗”，就及时更新一次“超时检测的定时器”的初值
```

```
{
    Gu8ReceFeedDog=0;

    vGu8ReceTimeOutFlag=0;
    vGu16ReceTimeOutCnt=RECE_TIME_OUT;//更新一次“超时检测的定时器”的初值
    vGu8ReceTimeOutFlag=1;
}
else if(Gu8ReceStep>0&&0==vGu16ReceTimeOutCnt) //超时，并且步骤不在接头暗号的步骤
{
    Gu8ReceStep=0; //串口接收数据的中断函数及时切换回接头暗号的步骤
}

if(1==Gu8FinishFlag)  //1 代表已经接收完毕一串新的数据，需要马上去处理
{
    switch(Gu8ReceType)  //接收到的数据类型
    {
        case 0x01:   //返回下位机的数组容量的大小

            Gu8Rece_Xor=Gu8ReceBuffer[Gu32ReceDataLength-1];  //提取接收到的“异或”
            Su8Rece_Xor=CalculateXor(Gu8ReceBuffer,Gu32ReceDataLength-1); //计算“异或”

            if(Su8Rece_Xor!=Gu8Rece_Xor) //验证“异或”，如果不相等，退出当前 switch
            {
                break;    //退出当前 switch
            }

            GtBigBufferUsart.u8QueueSendBuffer[0]=0xeb; //数据头
            GtBigBufferUsart.u8QueueSendBuffer[1]=0x01; //数据类型 返回数组容量的大小
            pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2];
            *pSu32Data=11; //数据长度  本条指令的数据总长是 11 个字节

            //提取数组容量的大小
            pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2+4];
            *pSu32Data=sizeof(Cu8TestTable);//相当于*pSu32Data=57;sizeof 请参考第 69 节

  //异或算法的函数
  GtBigBufferUsart.u8QueueSendBuffer[10]=CalculateXor(GtBigBufferUsart.u8QueueSendBuffer,
                                                      10);  //最后一个字节不纳入计算

            //队列驱动函数的状态 0 为初始状态 1 为通讯成功 2 为通讯失败
            GtBigBufferUsart.u8QueueStatus=0; //队列驱动函数的通讯状态
            GtBigBufferUsart.u8QueueSendTrig=1;//队列驱动函数的发送的启动

            Gu8QueueReceUpdate=1;  //告诉“队列驱动函数”此发送指令无需等待上位机的应答
```

```
            break;

        case 0x02:   //返回下位机的分段数据

            Gu8Rece_Xor=Gu8ReceBuffer[Gu32ReceDataLength-1];  //提取接收到的“异或”
            Su8Rece_Xor=CalculateXor(Gu8ReceBuffer,Gu32ReceDataLength-1); //计算“异或”

            if(Su8Rece_Xor!=Gu8Rece_Xor) //验证“异或”，如果不相等，退出当前 switch
            {
                break;    //退出当前 switch
            }

            pSu32Data=(unsigned long *)&Gu8ReceBuffer[6]; //数据转换。
             Su32CurrentAddr=*pSu32Data;  //读取的起始地址

            pSu32Data=(unsigned long *)&Gu8ReceBuffer[6+4]; //数据转换。
             Su32CurrentSize=*pSu32Data;  //读取的发送的数据量

            GtBigBufferUsart.u8QueueSendBuffer[0]=0xeb; //数据头
            GtBigBufferUsart.u8QueueSendBuffer[1]=0x02; //数据类型 返回分段数据

            pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2];
            *pSu32Data=6+4+4+Su32CurrentSize+1; //数据总长度

            pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2+4];
            *pSu32Data=Su32CurrentAddr; //返回接收到的起始地址

            pSu32Data=(unsigned long *)&GtBigBufferUsart.u8QueueSendBuffer[2+4+4];
            *pSu32Data=Su32CurrentSize; //返回接收到的当前批次的数据量

            for(i=0;i<Su32CurrentSize;i++)
            {
                  //装载即将要发送的分段数据
               GtBigBufferUsart.u8QueueSendBuffer[6+4+4+i]=Cu8TestTable[Su32CurrentAddr+i];
            }

            //异或算法的函数
            GtBigBufferUsart.u8QueueSendBuffer[6+4+4+Su32CurrentSize]=
            CalculateXor(GtBigBufferUsart.u8QueueSendBuffer, 6+4+4+Su32CurrentSize);

            //队列驱动函数的状态 0 为初始状态 1 为通讯成功 2 为通讯失败
            GtBigBufferUsart.u8QueueStatus=0; //队列驱动函数的通讯状态
            GtBigBufferUsart.u8QueueSendTrig=1;//队列驱动函数的发送的启动
```

```
                Gu8QueueReceUpdate=1;  //告诉“队列驱动函数”此发送指令无需等待上位机的应答
                break;
        }

        Gu8FinishFlag=0;  //上面处理完数据再清零标志，为下一次接收新的数据做准备
    }
}

void UsartTask(void)    //串口收发的任务函数，放在主函数内
{
    QueueSend();       //发送的队列驱动涵数
    ReceDataHandle();  //接收数据后的处理涵数
}

void usart(void) interrupt 4   //串口接发的中断函数，中断号为 4
{
   if(1==RI)  //接收完一个字节后引起的中断
   {
        RI = 0; //及时清零，避免一直无缘无故的进入中断。

       if(0==Gu8FinishFlag)  //1 代表已经完成接收了一串新数据，并且禁止接收其它新的数据
       {
          Gu8ReceFeedDog=1; //每接收到一个字节的数据，此标志就置 1 及时更新定时器的值。
          switch(Gu8ReceStep)
          {
              case 0:     //“前部分的”数据头。接头暗号的步骤。
                   Gu8ReceBuffer[0]=SBUF; //直接读取刚接收完的一个字节的数据。
                   if(0xeb==Gu8ReceBuffer[0])  //等于数据头 0xeb，接头暗号吻合。
                   {
                      Gu32ReceCnt=1; //接收缓存的下标
                      Gu8ReceStep=1;  //切换到下一个步骤，接收其它有效的数据
                   }
                   break;

              case 1:     //“前部分的”数据类型和长度
                   Gu8ReceBuffer[Gu32ReceCnt]=SBUF; //直接读取刚接收完的一个字节的数据。
                   Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备
                   if(Gu32ReceCnt>=6)  //前 6 个数据。接收完了“数据类型”和“数据长度”。
                   {
                        Gu8ReceType=Gu8ReceBuffer[1];  //提取“数据类型”
                         //以下的数据转换，在第 62 节讲解过的指针法
                        pu32Data=(unsigned long *)&Gu8ReceBuffer[2]; //数据转换
                        Gu32ReceDataLength=*pu32Data; //提取“数据长度”
                        if(Gu32ReceCnt>=Gu32ReceDataLength) //靠“数据长度”来判断是否完成
```

```
                        {
                            Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
                            Gu8ReceStep=0;   //及时切换回接头暗号的步骤
                        }
                        else   //如果还没结束，继续切换到下一个步骤，接收“有效数据”
                        {
                           //本节只用到一个接收数组，把指针关联到 Gu8ReceBuffer 本身的数组
                            pGu8ReceBuffer=(unsigned char *)&Gu8ReceBuffer[6];
                            Gu32ReceCntMax=REC_BUFFER_SIZE;  //最大缓存

                            Gu8ReceStep=2;   //切换到下一个步骤
                        }
                    }
                    break;
            case 2:     //“后部分的”数据
                    pGu8ReceBuffer[Gu32ReceCnt-6]=SBUF; //这里的指针就是各种不同内存的化身!!!
                    Gu32ReceCnt++; //每接收一个字节，数组下标都自加 1，为接收下一个数据做准备

                    //靠“数据长度”来判断是否完成。也不允许超过数组的最大缓存的长度
                    if(Gu32ReceCnt>=Gu32ReceDataLength||Gu32ReceCnt>=Gu32ReceCntMax)
                    {
                       Gu8FinishFlag=1; //接收完成标志“置 1”，通知主函数处理。
                       Gu8ReceStep=0;   //及时切换回接头暗号的步骤
                    }
                    break;
          }
       }
   }
   else  //发送数据引起的中断
   {
        TI = 0;  //及时清除发送中断的标志，避免一直无缘无故的进入中断。
        Gu8SendByteFinish=1; //从 0 变成 1 通知主函数已经发送完一个字节的数据了。
   }
}

void UsartSendByteData(unsigned char u8SendData) //发送一个字节的底层驱动函数
{
        static unsigned int Su16TimeOutDelay;  //超时处理的延时计时器

        Gu8SendByteFinish=0;  //在发送以字节之前，必须先把此全局变量的标志清零。
        SBUF =u8SendData; //依靠寄存器 SBUF 作为载体发送一个字节的数据
        Su16TimeOutDelay=0xffff;  //超时处理的延时计时器装载一个相对合理的计时初始值
        while(Su16TimeOutDelay>0)  //超时处理
        {
```

```
            if(1==Gu8SendByteFinish)
            {
                break;  //如果 Gu8SendByteFinish 为 1，则发送一个字节完成，退出当前循环等待。
            }
            Su16TimeOutDelay--;  //超时计时器不断递减
        }

        //Delay();//在实际应用中，当连续发送一堆数据时如果发现丢失数据，可以尝试在此增加延时
}

//发送带协议的函数
void UsartSendMessage(const unsigned char *pCu8SendMessage,unsigned long u32SendMaxSize)
{
        static unsigned long i;
        static unsigned long *pSu32;
        static unsigned long u32SendSize;

        pSu32=(const unsigned long *)&pCu8SendMessage[2];
        u32SendSize=*pSu32;  //从带协议的数组中提取整包数组的有效发送长度

        if(u32SendSize>u32SendMaxSize) //如果“有效发送长度”大于“最大限制的长度”,数据异常
        {
           return;  //数据异常，直接退出当前函数，预防数组越界
        }

        for(i=0;i<u32SendSize;i++) //u32SendSize 为发送的数据长度
        {
           UsartSendByteData(pCu8SendMessage[i]); //基于“发送单字节的最小接口函数”来实现的
        }
}

unsigned char CalculateXor(const unsigned char *pCu8Buffer, //此处加 const 代表数组“只读”
                           unsigned long u32BufferSize)  //参与计算的数组的大小
{
    unsigned long i;
    unsigned char Su8Rece_Xor;
    Su8Rece_Xor=pCu8Buffer[0]; //提取数据串第“i=0”个数据作为异或的原始数据
    for(i=1;i<u32BufferSize;i++) //注意，这里是从第“i=1”个数据开始
    {
        Su8Rece_Xor=Su8Rece_Xor^pCu8Buffer[i];   //计算“异或”
    }
    return Su8Rece_Xor;  //返回运算后的异或的计算结果
}
```

```
void T0_time() interrupt 1
{

    if(1==vGu8QueueSendTimerFlag&&vGu16QueueSendTimerCnt>0) //队列发送的超时定时器
    {
         vGu16QueueSendTimerCnt--;
    }

    if(1==vGu8ReceTimeOutFlag&&vGu16ReceTimeOutCnt>0) //通讯过程中字节之间的超时定时器
    {
         vGu16ReceTimeOutCnt--;
    }

    TH0=0xfc;
    TL0=0x66;
}


void SystemInitial(void)
{
    unsigned char u8_TMOD_Temp=0;

    //以下是定时器 0 的中断的配置
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;

    //以下是串口接收中断的配置
    //串口的波特率与内置的定时器 1 直接相关，因此配置此定时器 1 就等效于配置波特率。
    u8_TMOD_Temp=0x20; //即将把定时器 1 设置为：工作方式 2，初值自动重装的 8 位定时器。
    TMOD=TMOD&0x0f; //此寄存器低 4 位是跟定时器 0 相关，高 4 位是跟定时器 1 相关。先清零定时器 1。
    TMOD=TMOD|u8_TMOD_Temp;  //把高 4 位的定时器 1 填入 0x2，低 4 位的定时器 0 保持不变。
    TH1=256-(11059200L/12/32/9600);  //波特率为 9600。11059200 代表晶振 11.0592MHz，
    TL1=256-(11059200L/12/32/9600);  //L 代表 long 的长类型数据。根据芯片手册提供的计算公式。
    TR1=1;  //开启定时器 1

    SM0=0;
    SM1=1;  //SM0 与 SM1 的设置：选择 10 位异步通讯，波特率根据定时器 1 可变
    REN=1;  //允许串口接收数据

    //为了保证串口中断接收的数据不丢失，必须设置 IP = 0x10，相当于把串口中断设置为最高优先级，
```

```
    //这个时候，串口中断可以打断任何其他的中断服务函数实现嵌套，
    IP =0x10;  //把串口中断设置为最高优先级，必须的。

    ES=1;          //允许串口中断
    EA=1;          //允许总中断
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{

}
```