

怎样用串口发送结构体-简单协议的封包和解包

先说解决方案，细节和实现代码都放在正文

下位机：把结构体拆分成8位的整型数据，加上数据包头和包尾，然后按顺序单个单个地发出；
上位机：把串口里的数据读取出来，找到包头，按顺序装填到结构体中，然后使用结构体引用数据；

一、串口通信

串口通讯(Serial Communication)是一种设备间非常常用的串行通讯方式

相信浏览本文的朋友都已经使用过串口通信协议在各机器之间传递信息。这种通讯方式只需要四只引脚就能在短距离内实现全双工的通信，非常方便。目前许多通用芯片（如STM32）还提供了硬件支持，并给定了数据收发的接口函数。

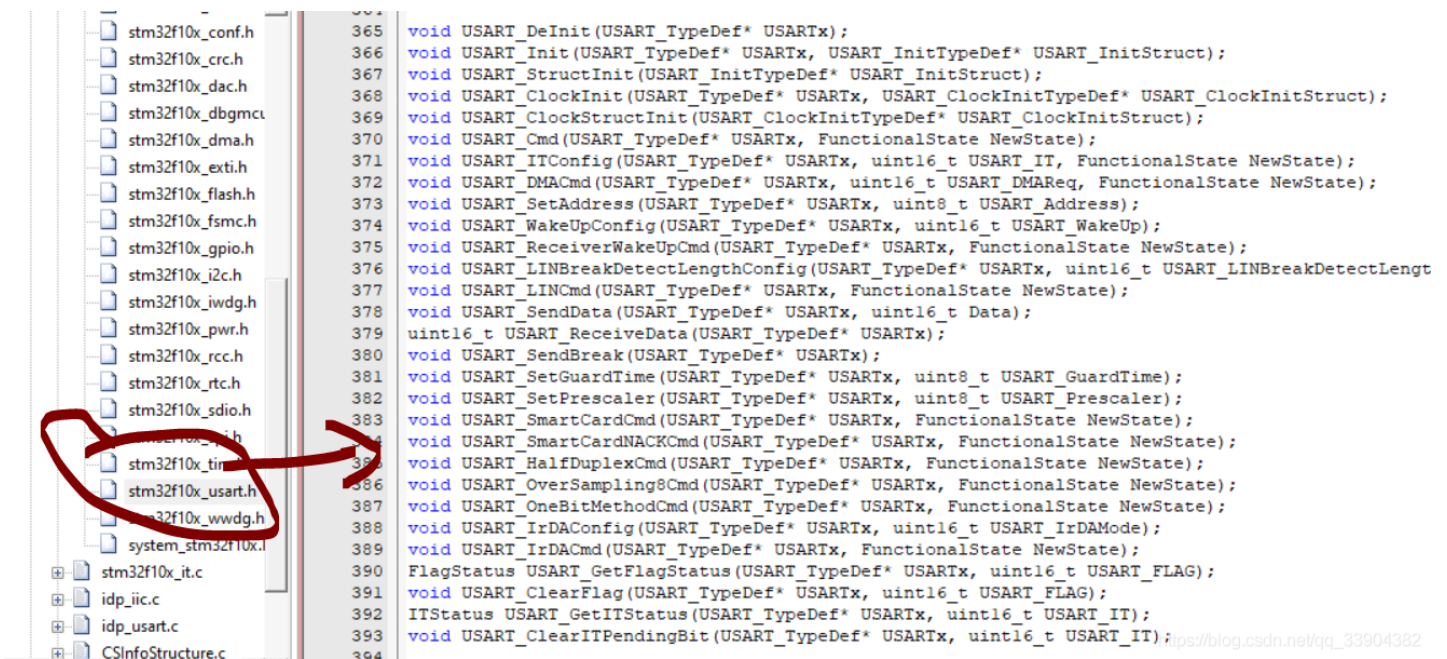
- 上电/断电复位(POR/PDR)、可编程电压监测器(PVD)
- 4~16MHz晶体振荡器
- 内嵌经出厂调校的8MHz的RC振荡器
- 内嵌带校准的40kHz的RC振荡器
- 带校准功能的32kHz RTC振荡器
- 低功耗
 - 睡眠、停机和待机模式
 - V_{BAT}为RTC和后备寄存器供电
- 3个12位模数转换器，1μs转换时间(多达21个输入通道)
- 系统时间基准：4~16MHz晶体振荡器
- 2个16位基本定时器用于驱动DAC
- 多达13个通信接口
 - 多达2个I²C接口(支持SMBus/PMBus)
 - 多达5个USART接口(支持ISO7816, LIN, IrDA接口和调制解调控制)
 - 多达3个SPI接口(18Mbit/秒)，2个可复用为I²S接口
 - CAN接口(2.0B 主动)
 - USB 2.0全速接口
 - SDIO接口
- CRC计算单元，96位的芯片唯一代码

以stm32为例，我们可以通过一些特定的函数使用芯片上的USART外设，不需要操心协议的电平规定就能够进行数据的收发，相关的函数在STM32的参考手册中列出：

Table 705. USART 库函数

函数名	描述
USART_DeInit	将外设 USARTx 寄存器重设为缺省值
USART_Init	根据 USART_InitStruct 中指定的参数初始化外设 USARTx 寄存器
USART_StructInit	把 USART_InitStruct 中的每一个参数按缺省值填入
USART_Cmd	使能或者失能 USART 外设
USART_ITConfig	使能或者失能指定的 USART 中断
USART_DMACmd	使能或者失能指定 USART 的 DMA 请求
USART_SetAddress	设置 USART 节点的地址
USART_WakeUpConfig	选择 USART 的唤醒方式
USART_ReceiverWakeUpCmd	检查 USART 是否处于静默模式
USART_LINBreakDetectLengthConfig	设置 USART LIN 中断检测长度
USART_LINCmd	使能或者失能 USARTx 的 LIN 模式
USART_SendData	通过外设 USARTx 发送单个数据

在固件库中也可以找到：



```
365 void USART_DeInit(USART_TypeDef* USARTx);
366 void USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct);
367 void USART_StructInit(USART_InitTypeDef* USART_InitStruct);
368 void USART_ClockInit(USART_TypeDef* USARTx, USART_ClockInitTypeDef* USART_ClockInitStruct);
369 void USART_ClockStructInit(USART_ClockInitTypeDef* USART_ClockInitStruct);
370 void USART_Cmd(USART_TypeDef* USARTx, FunctionalState NewState);
371 void USART_ITConfig(USART_TypeDef* USARTx, USART_TypeDef* USARTx, uint16_t USART_IT, FunctionalState NewState);
372 void USART_DMACmd(USART_TypeDef* USARTx, uint16_t USART_DMAREq, FunctionalState NewState);
373 void USART_SetAddress(USART_TypeDef* USARTx, uint8_t USART_Address);
374 void USART_WakeUpConfig(USART_TypeDef* USARTx, uint16_t USART_WakeUp);
375 void USART_ReceiverWakeUpCmd(USART_TypeDef* USARTx, FunctionalState NewState);
376 void USART_LINBreakDetectLengthConfig(USART_TypeDef* USARTx, uint16_t USART_LINBreakDetectLength);
377 void USART_LINCmd(USART_TypeDef* USARTx, FunctionalState NewState);
378 void USART_SendData(USART_TypeDef* USARTx, uint16_t Data);
379 uint16_t USART_ReceiveData(USART_TypeDef* USARTx);
380 void USART_SendBreak(USART_TypeDef* USARTx);
381 void USART_SetGuardTime(USART_TypeDef* USARTx, uint8_t USART_GuardTime);
382 void USART_SetPrescaler(USART_TypeDef* USARTx, uint8_t USART_Prescaler);
383 void USART_SmartCardCmd(USART_TypeDef* USARTx, FunctionalState NewState);
384 void USART_SmartCardNACKCmd(USART_TypeDef* USARTx, FunctionalState NewState);
385 void USART_HalfDuplexCmd(USART_TypeDef* USARTx, FunctionalState NewState);
386 void USART_OverSampling8Cmd(USART_TypeDef* USARTx, FunctionalState NewState);
387 void USART_OneBitMethodCmd(USART_TypeDef* USARTx, FunctionalState NewState);
388 void USART_IrDAModeConfig(USART_TypeDef* USARTx, uint16_t USART_IrDAMode);
389 void USART_IrDACmd(USART_TypeDef* USARTx, FunctionalState NewState);
390 FlagStatus USART_GetFlagStatus(USART_TypeDef* USARTx, uint16_t USART_FLAG);
391 void USART_ClearFlag(USART_TypeDef* USARTx, uint16_t USART_FLAG);
392 ITStatus USART_GetITStatus(USART_TypeDef* USARTx, uint16_t USART_IT);
393 void USART_ClearITPendingBit(USART_TypeDef* USARTx, uint16_t USART_IT);
```

大致看上去十分方便，但细看就会发现一个十分重要的问题，譬如发送数据的这个函数USART_SendData()：

```
/**
 * @brief Transmits single data through the USARTx peripheral.
 * @param USARTx: Select the USART or the UART peripheral.
 * This parameter can be one of the following values:
 * USART1, USART2, USART3, UART4 or UART5.
 * @param Data: the data to transmit.
 * @retval None
 */
void USART_SendData(USART_TypeDef* USARTx, uint16_t Data)
{
    /* Check the parameters */
    assert_param(IS_USART_ALL_PERIPH(USARTx));
    assert_param(IS_USART_DATA(Data));

    /* Transmit Data */
    USARTx->DR = (Data & (uint16_t)0x01FF);
}
```

他只给你提供了单个数据的发送方式，而且数据类型限制位16位无符号整型（uint16_t），这就引出了我们在文章开头提到的拆分并封包发送的必要性。下面我们先介绍一下发送的结构体的样子，然后再说拆分封包的问题。

二、定义要发送的结构体

首先明确发送的结构体是什么样子的

```
/**
 * @part 通信数据结构
 */

/* 加速度信息结构体-XYZ三分量 */
typedef struct CSModuleInfo_ACC{
    float _acc_X;
    float _acc_Y;
    float _acc_Z;
}CSInfo_Acc;

/* 经纬度信息结构体-经纬两分量 */
typedef struct CSModuleInfo_LL{
    float _latitude;
    float _longitude;
}CSInfo_LL;

/* 测控站信息结构体 */
typedef struct CSInfoStrcutre CSInfoS;
typedef struct CSInfoStrcutre{
    /* 核心温度 MCU温度 */
    float _temp_O_MCU;
    /* 气温 */
    float _temp_env;
    /* 气压 */
    float _gp;
    /* 加速度 */
    CSInfo_Acc _acc;
    /* 经纬度 */
    CSInfo_LL _ll;
} * ptrCSInfo;
```

为了保证本文能符合大伙的需求，咱搞一个结构体嵌套，并且把数据类型都定义浮点数。意在说明我们这种传输结构体的方式不受结构体类型的限制，也不受浮点数的存储方式的限制，请放心学习使用。

注：代码中/* 测控站信息结构体 */部分的ptrCSInfo是这个大结构体的指针类型，CSInfoS是这个结构体的别名，这种写法是C语言的语法规则所允许的，不用感到奇怪。

三、下位机封包发送

封包发送的过程可以用下面的代码实现：

```
/**
 * @brief 将数据打包并发送到上位机
 * @param
 * ptrInfoStructure 指向一个装填好信息的infoStructure的指针
 * @retval 无
 */

void CSInfo_PackAndSend(ptrCSInfo ptrInfoStructure)
{
    uint8_t infoArray[32]={0};
    uint8_t infoPackage[38]={0};
    CSInfo_2Array_uint8(ptrInfoStructure, infoArray);
    CSInfo_Pack(infoPackage, infoArray, sizeof(infoArray));
    CSInfo_SendPack(infoPackage, sizeof(infoPackage));
}
```

向这个函数传入一个装有数据的结构体的指针ptrInfoStructure，依次调用CSInfo_2Array_uint8、CSInfo_Pack、CSInfo_SendPack这三个自定义函数，即可通过串口将结构体发送出去。

这三个函数分别对应着拆分结构体、按照协议/规则封包和发送数据三个过程，具体说明和代码如下：

1、拆分

文章开头我们已经说了，先要把结构体拆分成8位无符号整型(uint8_t)的数据：

```
/**
 * @brief 将数据段（CSInfoStructure）重组为uint8类型的数组
 * @param
 * infoSeg 指向一个CSInfoStructure的指针
 * infoArray 由数据段重组的uint8类型数组
 * @retval 无
 */
void CSInfo_2Array_uint8(ptrCSInfo infoSeg, uint8_t* infoArray)
{
    int ptr=0; uint8_t
    *infoElem=(uint8_t*)infoSeg;
    for(ptr=0; ptr<sizeof(CSInfoS); ptr++){
        infoArray[ptr] = (*(infoElem+ptr));
    }
}
```

}

传入一个结构体的指针，并传入一个对应大小(`uint8_t`)类型的数组，用来装结构体拆分出来的元素。

那么数组需要多大呢？我们知道8位(**bit**)就是一个字节(**Byte**)，所以这个数组理论上只需要和这个结构体的字节数一样大就可以了！也就是：

```
sizeof(CSInfoS)
```

的返回值。这里我们也可以口算一下，结构体中总共有8个`float`类型的数据，也就是 $8 \times 32\text{bit} = 8 \times 4\text{Byte} = 32\text{Byte}$ 。结构体的大小也就是32字节，所以可以拆分成32个`uint8_t`类型的元素，数组大小也就需要32。

注意：传入的数组需要足够的大小，不要整个空指针或者不够大的数组进去。当然，你也可以返回一个数组，但我喜欢这种隐式返回的风格。

2、封包

选定一组特定的数据作为数据包的头部，选定另一组特定的数据作为数据包的尾部，方便我们在上位机接收数据后找到每一组数据的开始和结尾。

这里我们选定：

0x80 0x81 0x82

作为数据包的头部，同时选定：

0x82 0x81 0x80

作为数据包的尾部。所以我们向上位机发送的单个数据包都是如下形式的：

```
/**
 * @part 通信协议
 * @Protocol
 * -----
 * 头 | 信息 | 尾
 * -----
 * 0x80|0x81|0x82| CSInfoStrcutre |0x82|0x81|0x80
 * -----
 * 3Byte | 32Byte | 3Byte
 * -----
 */
```

上面`|CSInfoStrcutre|`的位置就是我们在上一步获得的`uint8_t`类型的数组`infoArray`，内容是`CSInfoStrcutre`中的数据。

封包的过程如下面的代码所示：

```
/**
 * @brief 按协议打包
 * @param
 * package 打包结果，按协议结果为3+32+3=38字节 （38*8bit）
 * infoArray 由数据段重组的uint8类型数组 | 结果
 * infoSize 数据段的大小---占用内存字节数（协议规定为32Byte）
 * @retval 无
 */
void CSInfo_Pack(uint8_t* infopackage,uint8_t* infoArray,uint8_t infoSize)
{
    uint8_t ptr=0;
    infopackage[0] = HEAD1;
    infopackage[1] = HEAD2;
    infopackage[2] = HEAD3;

    /* 将信息封入数据包中 */
    for(;ptr<infoSize;ptr++){
        infopackage[ptr+3] = infoArray[ptr];
    }

    infopackage[ptr+3] = TAIL1;
    infopackage[ptr+4] = TAIL2;
    infopackage[ptr+5] = TAIL3;
}
```

3、发送

接着，我们将把这个玩意儿(`infopackage`)通过串口发送出去：

```
/**
 * @brief 将数据包发送到上位机
 * @param
 * infoPackage 数据包
 * packSize 数据包的大小---占用内存字节数（协议规定为38Byte）
 * @retval 无
 */
void CSInfo_SendPack(uint8_t* infoPackage,uint8_t packSize)
{
    int ptr=0;
    for(ptr=0;ptr<packSize;ptr++){
        USART_SendByte(infoPackage[ptr]);
    }
}
```

注意，为了方便使用，这里我们用到了一个名为`USART_SendByte`的自定义函数，其定义如下：

```
/**
 * @brief 通过USART通道向上位机发送一个字节（8bit）的数据
 * @param byte 要发送的8位数据
 * @retval 无
 */
void USART_SendByte(uint8_t byte)
{
    /* 发送一个字节数据到串口 */
    USART_SendData(DEBUG_USARTx,byte);
    /* 等待发送完毕 */
    while (USART_GetFlagStatus(DEBUG_USARTx, USART_FLAG_TXE) == RESET);
}
```

到这里，我们就了解完了下位机打包发送的部分，接下来我们转到上位机视角，看一下咋个接收数据，咋个解析数据，也就是咋个把数据又装回一个结构体里，方便我们引用。

四、上位机接收数据并解包

回顾一下文章开头，我们说上位机的这部分工作的流程是这样的：

- 1、把串口里的数据读取出来
- 2、找到包头，
- 3、把数据包中对应数据的部分按顺序装填到结构体中

大致流程如下面的代码所示：

```
/* 读取数据 */
uint8_t packages[INFOSIZE*3]={0};
int numHasRead = readInfoFromSerialport(packages);
/* 解析数据 */
uint8_t infoArray[INFOSIZE];
/* 提取数据包 */
bool readable = CSInfo_GetInfoArrayInpackages(infoArray,packages,numHasRead);
/* 解包 */
if(readable)
    CSInfo_InfoArray2CSInfoS(infoArray,this->_ptrCSInfo);
```

也即是依次调用readInfoFromSerialport、CSInfo_GetInfoArrayInpackages、CSInfo_InfoArray2CSInfoS在这三个函数，从串口缓冲区的一堆数据里找到一个完整的数据包并把它装填到结构体里。

下面详细介绍这三个自定义函数：

1、读取数据

你可以用你所知的任何方法从串口的缓冲区读取出来，只要你能把它们放到一个方便后续的解包操作访问的地方。

这里我使用Qt开发的上位机界面，故而也顺带使用Qt提供的serialport类中的方法来读取，具体可以参考Qt的帮助文档，这里只做简要说明：

```
/**
 * @brief 把当前serialport缓冲区的数据全部读取到一个uint8类型的数组中
 * @param packages 从串口读取到的包含数据包的数据
 * @retval numHasRead 从缓冲区读取到的字节数
 */
int readInfoFromSerialport(uint8_t* packages)
{
    int numHasRead(0);
    /* 没有可用的串口设备则中止读取操作 退出函数 */
    if(QSerialPortInfo::availablePorts().isEmpty())
        return 0;
    _port = new QSerialPort(QSerialPortInfo::availablePorts()[0]);
    _port->setPort(QSerialPortInfo::availablePorts()[0]);
    if(!_port->open(QIODevice::ReadWrite)){
        goto next;
    }else{
        _port->setParity(QSerialPort::NoParity);
        _port->setBaudRate(QSerialPort::Baud115200);
        _port->setDataBits(QSerialPort::Data8);
        _port->setStopBits(QSerialPort::OneStop);
        _port->setFlowControl(QSerialPort::NoFlowControl);
        /* 开始从serialport读取数据 */
        /* 读取串口缓冲区所有的数据到CSInfo的缓冲区infoArray */
        _port->waitForReadyRead();
        QByteArray dataArray = _port->read(200);
        numHasRead = dataArray.size();
        if (INFOSIZE*3<numHasRead){
            for(int i=0;i<INFOSIZE*3;i++){
                *(packages+i) = dataArray[i];
            }
        }
    }
    next:;
    delete _port;
    return numHasRead;
}
```

上述代码首先获取了一个serialport类的对象_port，然后通过一系列的setxxx函数配置了必要的参数。接着调用readAll把串口中所有的数据读取到dataArray（readAll()的返回值就是一个QByteArray类型的容器），然后把大小等同于三个infoStructure的数据放到packages中，预备进行下一步的解包操作。

注意，之所以要读取三个基数，是为了保证至少包含一个完整的数据包。

2、找到一个完整的数据包

前面提到了，我们设定每个数据包的头部是0x80|0x81|0x82,而数据包的尾部则反过来。根据这个特征：

```
/**
 * @part 通信协议
 * @Protocol
 * -----
 * 头 | 信息 | 尾
 * -----
 * 0x80|0x81|0x82| CSInfoStrcutre |0x82|0x81|0x80
 * -----
 * 3Byte | 32Byte | 3Byte
 * -----
 */
```

我们可以先在上一步获得的packages中找到一个数据包的头部，以确定一个数据包的开始位置：

```
/**
 * @brief 在串口读取到的数据中提取出一个数据包的数据段（CSInfoStructure对应的部分）
 * 转存到infoArray中，供后续解析为CSInfoStructure.
 * @param infoArray 转存CSInfo的数组
 * packages 从串口读取到的包含数据包的数据
 * sizepackages 从串口读取到的字节数（packages的大小）
 * @retval 无
 */
bool CSInfo_GetInfoArrayInpackages(uint8_t* infoArray,uint8_t* packages,int sizepackages)
{
    }
```

```

int ptr;bool readable(true);
if(sizepackages<INFOSIZE*3){
    readable = false;
    return readable;
}
for(ptr=0;ptr<INFOSIZE*3;ptr++){
    // or: for(ptr=0;ptr<sizepackages-3;ptr++){ */
    if (packages[ptr]==HEAD1)&&(packages[ptr+1]==HEAD2)&&(packages[ptr+2]==HEAD3))
        break;
}
ptr += 3;
for(int i=0;i<INFOSIZE;i++){
    infoArray[i] = packages[ptr+i];
}
return readable;
}

```

通过调用这个函数，我们把packages中的一个完整的数据包的InfoStructure部分放到了infoArray中。接着看第三步，我们将把这个结构体的数据写入一个结构体中，真正还原它在下位机中的样子：

3、解析数据

直接把结构体当成一个数组，把数据依次填写进去就ok了

```

/**
 * @brief  把有一个数据段的数组解析为一个CSInfoStructure,结果存到参数2对应的地址
 * @param
 *         infoArray  存有一个数据段的uint8类型的数组
 *         infoStrc  从串口读取到的字节数 (packages的大小)
 * @retval  无
 */
void CSInfo_InfoArray2CSInfoS(uint8_t* const infoArray,ptrCSInfo infoStrc)
{
    uint8_t* u8PtrOStrc = (uint8_t*)infoStrc;
    for(int i=0;i<INFOSIZE;i++){
        *(u8PtrOStrc+i) = infoArray[i];
    }
}

```

到这里，我们就完成了使用串口发送结构体的任务，而且了解了封包和解包的基本思路。

我把上位机的源代码链接放到这里，需要的可以单击自取。读取和解析的代码分别在Sources/CSInfoReader.c和Sources/CSInfoParser.c文件中。

下位机的代码暂未上传，需要的朋友可以留言索取。