

Seasky串口通信协议

可用于视觉，以及其他自研模块（仅限串口通信）

一、串口配置

通信方式是串口，配置为波特率 115200，8 位数据位，1 位停止位，无硬件流控，无校验位。

二、接口协议说明

- 以下所有低位在前发送

1. 通信协议格式

帧头	数据ID	数据	帧尾
protocol_header(4-byte)	cmd_id(2-byte)	data (n-byte)	frame_tail(2-byte, CRC16, 整包校验)

2. 帧头详细定义

帧头	偏移位置	字节大小	内容
sof(CMD)	0	1	数据帧起始字节，固定值为 0xA5
data_length	1	2	数据帧中 data 的长度
crc_check	3	1	帧头CRC校验

3. cmd_id 命令码 ID 说明(字节偏移 4，字节大小 2)

命令码	数据段长度	功能说明
0x0001(可修改)		BMI088姿态数据
0x0002		INA226数据

4. 数据段data (n-byte)

数据	偏移位置	字节大小	内容
flags_register	6	2	16位标志置位寄存器
float_date (len)	8	4 * len	float数据内容 (4 * len-byte)

5. frame_tail(CRC16，整包校验)

6. 使用待发送数据更新获取发送数据帧

```
void get_protocol_send_data
(uint16_t send_id,          //信号id
 uint16_t flags_register, //16位寄存器
```

```
float    *tx_data,          //待发送的float数据
uint8_t  float_length,     //float的数据长度
uint8_t  *tx_buf,          //待发送的数据帧
uint16_t *tx_buf_len)      //待发送的数据帧长度
```

使用说明，如定义需要发送的数据为 `flags_register_t` (寄存器，用于传递部分标志), `tx_data[2]` (假设只发视觉的两个数据x,y), `tx_data_len = 2` (发送的float数据长度为2), 信号id定义为 `vis_id = 0x000A`;

定义发送数据发送缓冲区 `uint8_t tx_buf[100]`, `uint16_t tx_buf_len = 0` 则上述函数调用后会更新 `tx_buf` 中的数据，以及更新发送数据长度 `tx_buf_len`

```
/*更新数据发送缓冲区*/
get_protocol_send_data(vis_id,
                       &flags_register_t,
                       tx_data,
                       &tx_data_len,
                       tx_buf,
                       tx_buf_len);
/*发送缓冲区数据，多字节发送函数，按字节形式发送，linux或win只需调用底层的字节发送接口函数*/
usart6_send(tx_buf,tx_buf_len);
```

7. 接收数据获取id，获取flags_register，rx_data

```
uint16_t get_protocol_info
(uint8_t  *rx_buf,          //接收到的原始数据
 uint16_t *rx_pos,          //原始数据指针
 uint16_t *flags_register, //接收数据的16位寄存器地址
 float    *rx_data)         //接收的float数据存储地址
```

使用说明，首先将接收数据存储于 `rx_buf[]`, 每接收一个数据 `*rx_pos` 的值会加一，以便于存储下一个数据，原始数据缓冲的存储由中断方式完成，在linux或win上可能有所区别，调用此函数，函数中包含 `crc` 校验，数据解读等相关处理，最后会更新 `flags_register` 的值即为收到的16位寄存器的值，`rx_data[]` 即为收到的float 数据。

值得注意的是如果改函数调用的返回值为非零时，才是获取到的正确ID，因此如果要获取正确ID需要加入判断，如下所示：

```
void uart_task(void *pvParameters)
{
    while(1)
    {
        uint16_t id_t = 0;
        id_t = get_protocol_info
```

```

        (USART_RX_BUF,          //接收到的原始数据
        &USART_RX_STA,          //原始数据指针
        &ina226_t.ctr_t.flags, //接收数据的16位寄存器地址
        ina226_t.ctr_t.rx_data); //接收的float数据存储地址
        if(id_t!=0)ina226_t.ctr_t.ctr_id=id_t;
        vTaskDelay(10);
    }
}

```

8. Seasky串口通信协议

bsp_protocol.h

```

#ifndef _BSP_PROTOCOL_H
#define _BSP_PROTOCOL_H

#include "main.h"

#define PROTOCOL_CMD_ID 0XA5

#define OFFSET_BYTE 8 //出数据段外，其他部分所占字节数

typedef struct
{
    __packed struct
    {
        uint8_t  sof;
        uint16_t data_length;
        uint8_t  crc_check; //帧头CRC校验
    } header;                //数据帧头
    uint16_t cmd_id;          //数据ID
    uint16_t frame_tail;      //帧尾CRC校验
} protocol;

/*更新发送数据帧，并计算发送数据帧长度*/
void get_protocol_send_data
(uint16_t send_id,          //信号id
uint16_t flags_register, //16位寄存器
float    *tx_data,         //待发送的float数据
uint8_t  float_length, //float的数据长度
uint8_t  *tx_buf,         //待发送的数据帧
uint16_t *tx_buf_len);    //待发送的数据帧长度

/*接收数据处理*/
uint16_t get_protocol_info
(uint8_t  *rx_buf,          //接收到的原始数据
uint16_t *rx_pos,          //原始数据指针
uint16_t *flags_register, //接收数据的16位寄存器地址
float    *rx_data);        //接收的float数据存储地址

```

```
/*中断函数传值处理*/  
void PROTOCOL_RX_IRQ(uint8_t res,uint8_t *rx_buf,uint16_t *rx_buf_pos);  
#endif
```

bsp_protocol.c

```
/*  
    @SEASKY---2020/09/05  
    Seasky串口通信协议  
*/  
#include "bsp_protocol.h"  
  
#include "bsp_crc8.h"  
#include "bsp_crc16.h"  
  
/*获取CRC8校验码*/  
uint8_t Get_CRC8_Check(uint8_t *pchMessage,uint16_t dwLength)  
{  
    return crc_8(pchMessage,dwLength);  
}  
/*检验CRC8数据段*/  
uint8_t CRC8_Check_Sum(uint8_t *pchMessage,uint16_t dwLength)  
{  
    uint8_t ucExpected = 0;  
    if ((pchMessage == 0) || (dwLength <= 2)) return 0;  
    ucExpected = Get_CRC8_Check(pchMessage, dwLength-1);  
    return ( ucExpected == pchMessage[dwLength-1] );  
}  
/*获取CRC16校验码*/  
uint16_t Get_CRC16_Check(uint8_t *pchMessage,uint32_t dwLength)  
{  
    return crc_16(pchMessage,dwLength);  
}  
/*检验CRC16数据段*/  
uint16_t CRC16_Check_Sum(uint8_t *pchMessage, uint32_t dwLength)  
{  
    uint16_t wExpected = 0;  
    if ((pchMessage == 0) || (dwLength <= 2))  
    {  
        return 0;  
    }  
    wExpected = Get_CRC16_Check ( pchMessage, dwLength - 2);  
    return (((wExpected & 0xff) == pchMessage[dwLength - 2] )&&  
(((wExpected >> 8) & 0xff) == pchMessage[dwLength - 1]));  
}  
/*检验数据帧头*/  
uint8_t protocol_heade_Check(
```

```

    protocol *pro,
    uint8_t *rx_buf,
    uint16_t *rx_pos)
{
    if(rx_buf[0] == PROTOCOL_CMD_ID)
    {
        pro->header.sof = rx_buf[0];
        if(CRC8_Check_Sum(&rx_buf[0],4))
        {
            pro->header.data_length = (rx_buf[2]<<8) | rx_buf[1];
            pro->header.crc_check = rx_buf[3];
            pro->cmd_id = (rx_buf[5]<<8) | rx_buf[4];
            return 1;
        }
    }
    else
    {
        *rx_pos = 0;
    }
    return 0;
}

```

```
float float_protocol(uint8_t *dat_t)
```

```

{
    uint8_t f_data[4];
    f_data[0] = *(dat_t+0);
    f_data[1] = *(dat_t+1);
    f_data[2] = *(dat_t+2);
    f_data[3] = *(dat_t+3);
    return *(float*)f_data;
}

```

```
/*
```

此函数根据待发送的数据更新数据帧格式以及内容，实现数据的打包操作
后续调用通信接口的发送函数发送tx_buf中的对应数据

```
*/
```

```

void get_protocol_send_data
(uint16_t send_id,          //信号id
uint16_t flags_register,   //16位寄存器
float *tx_data,            //待发送的float数据
uint8_t float_length,     //float的数据长度
uint8_t *tx_buf,          //待发送的数据帧
uint16_t *tx_buf_len)     //待发送的数据帧长度
{

```

```

    uint16_t crc16;
    uint16_t data_len;

```

```
    data_len = float_length*4+2;
```

```
    /*帧头部分*/
```

```
    tx_buf[0] = PROTOCOL_CMD_ID;
```

```
    tx_buf[1] = data_len & 0xff;
```

```
    //低位在前
```

```
    tx_buf[2] = (data_len >> 8) & 0xff;
```

```
    //低位在前
```

```

tx_buf[3] = Get_CRC8_Check(&tx_buf[0],3); //获取CRC8校验位

/*数据的信号id*/
tx_buf[4] = send_id & 0xff;
tx_buf[5] = (send_id>>8) & 0xff;

/*建立16位寄存器*/
tx_buf[6] = flags_register & 0xff;
tx_buf[7] = (flags_register>>8) & 0xff;

/*float数据段*/
for(int i=0; i<4*float_length; i++)
{
    tx_buf[i+8] = ((uint8_t*)(&tx_data[i/4]))[i%4];
}

/*整包校验*/
crc16 = Get_CRC16_Check(&tx_buf[0],data_len+6);
tx_buf[data_len+6] = crc16 & 0xff;
tx_buf[data_len+7] = (crc16 >>8) & 0xff;

*tx_buf_len = data_len+8;
}
/*
    此函数用于处理接收数据,
    返回数据内容的id
*/

uint16_t get_protocol_info
(uint8_t *rx_buf,          //接收到的原始数据
uint16_t *rx_pos,          //原始数据指针
uint16_t *flags_register,  //接收数据的16位寄存器地址
float *rx_data)            //接收的float数据存储地址
{
    static protocol pro;
    static uint16_t date_length;
    if(protocol_heade_Check(&pro,rx_buf,rx_pos))
    {
        date_length = OFFSET_BYTE + pro.header.data_length;
        while(CRC16_Check_Sum(&rx_buf[0],date_length))
        {
            *flags_register = (rx_buf[7]<<8) | rx_buf[6];
            for(int i=0; i<(pro.header.data_length-2)/4; i++)
            {
                rx_data[i] = float_protocol(&rx_buf[8+4*i]);
            }
            for(int i=0; i<date_length; i++)
            {
                rx_buf[i] = 0;
            }
            *rx_pos = 0;
        }
    }
}

```

```
        return pro.cmd_id;
    }

    }
    return 0;
}
/*放入中断接收函数*/
void PROTOCOL_RX_IRQ(uint8_t res,uint8_t *rx_buf,uint16_t *rx_buf_pos)
{
    rx_buf[*rx_buf_pos]=res;
    (*rx_buf_pos) +=1;
    if(rx_buf[0]!=PROTOCOL_CMD_ID)
    {
        *rx_buf_pos = 0;
    };
}
```

9. 通信协议依赖文件(CRC8,CRC16算法)

bsp_crc16.h

bsp_crc16.c

bsp_crc8.h

bsp_crc8.c