STREAM框架

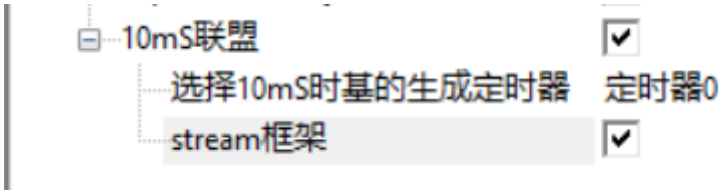
（抢先阅读版，请踊跃反馈文档难懂的部分）

之所以要把一个框架和外设库并列，是因为stream虽然是串口的应用，但足够复杂和强大。

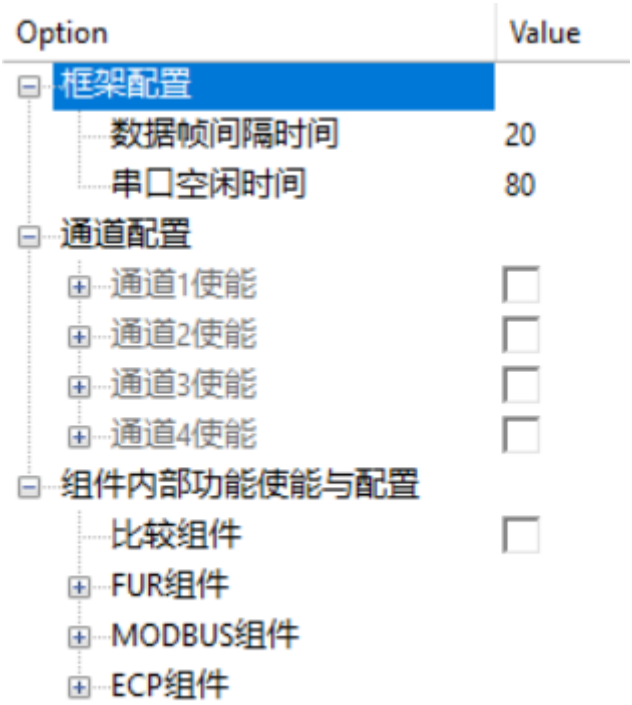
这里的stream可不是那个有名的游戏平台。stream有“流”的含义，对应这个框架是用于处理串口数据流的。

stream框架的开发初衷是为了解决多个串口协议同时生效时的串口中断处理时间过长的问题。**通过建立缓存将串口数据保存下来，然后在主函数中再处理。**这样可以保证中断能快速的退出，不影响其他函数的执行。

目前stream框架已经深度集成到ECBM库中，并归入10mS联盟之中。可以打开《10mS联盟.pdf》简单了解一下。



如图所示，使能10mS联盟之后在将stream框架的使能勾上就行了。接下来，去stream.h可以设置stream的各种参数和组件的使能及参数。

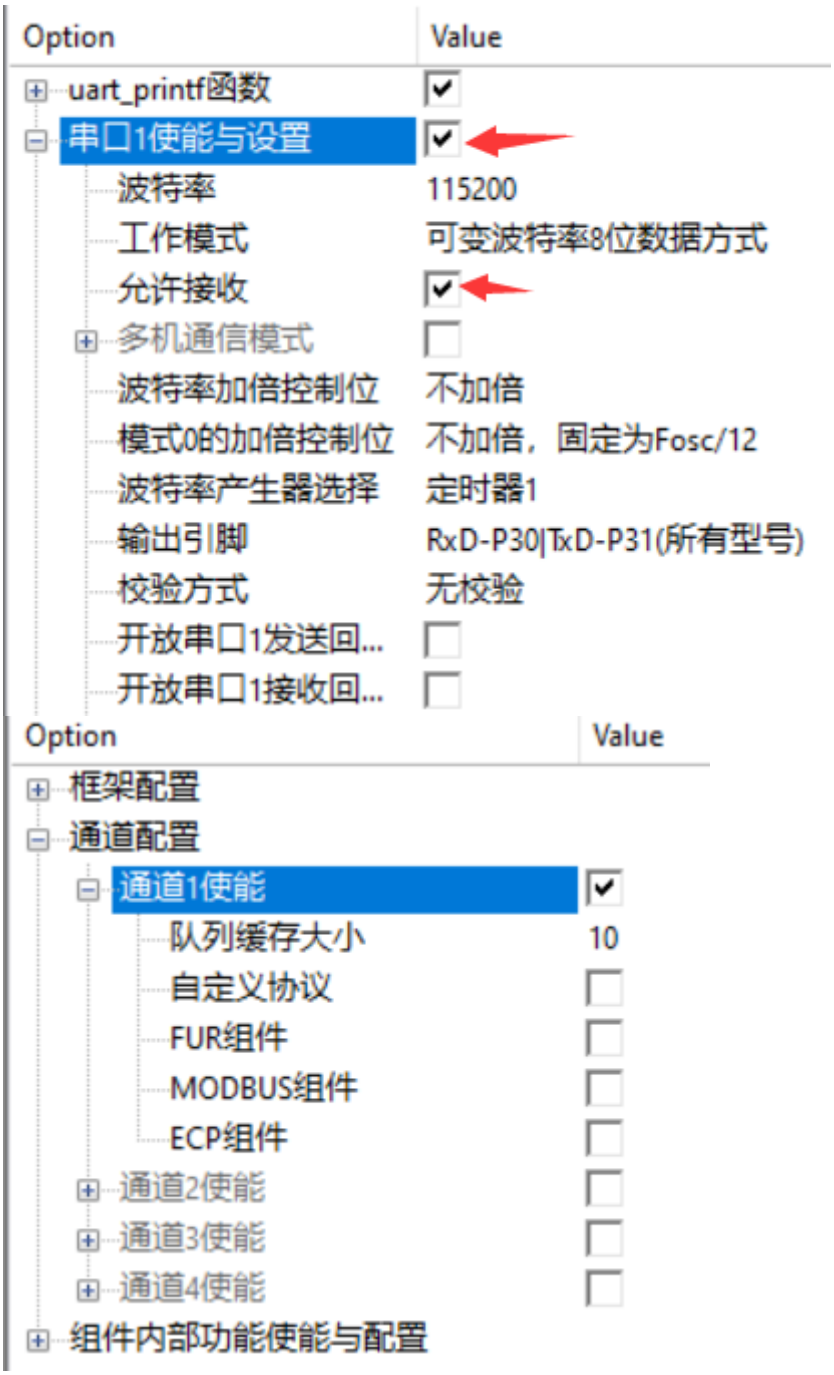


框架配置

- 数据帧间隔时间：当超过这个时间没有收到新的数据时，stream会认为已经接收完一帧数据，于是就会置标志位让解析函数去解析。
- 串口空闲时间：当超过这个时间没有收到新的数据时，stream会认为通信已经结束，将会把所有组件的状态机还原成初态。

通道配置

这里的通道，其实就是串口。4个通道对应4个串口。stream可以看做是一种解析服务，虽然依附于串口但也和串口相互独立。在这边使能通道的时候，记得也去UART库去打开对应的串口使能。



如上图所示，分别打开串口1和通道1的使能就可以了。当然串口的【允许接收】要记得开。stream会自动挂载到串口的接收中断里，所以**不需要**在UART库中特地开放接收回调。当然如果你需要去判断接收数据可以开，但是不用特地的为了stream去开接收回调。

- 队列缓存大小：当串口接收到数据的时候，会存到这个缓存中，所以缓存的大小由接收的数据大小来决定。如果协议是不定长度的，可以选取最大长度来作为缓存大小。**注意每个通道的缓存大小是独立的。**
- FUR组件：使能之后，stream会按FUR协议解析数据流，并按协议调用对应的回调函数。
- MODBUS组件：使能之后，stream会按modbus协议解析数据流，并按协议调用对应的回调函数。
- ECP组件：使能之后，stream会按ECP协议解析数据流，并按协议调用对应的回调函数。

API

ecbm_stream_init

函数原型：void ecbm_stream_init(void);

描述

流处理初始化函数。

输入

无

输出

无

返回值

无

调用例程

```
#include "ecbm_core.h" //加载库函数的头文件。
void main(void){ //main函数，必须的。
    system_init(); //系统初始化函数，也是必须的。
    ecbm_stream_init(); //stream框架初始化。
    while(1){
        ecbm_stream_main();//放到主循环不断执行。
    }
}
```

注意事项

无

ecbm_stream_main

函数原型：void ecbm_stream_main(void);

描述

流处理主函数函数。

输入

无

输出

无

返回值

无

调用例程

```
#include "ecbm_core.h" //加载库函数的头文件。
void main(void){ //main函数，必须的。
    system_init(); //系统初始化函数，也是必须的。
    ecbm_stream_init(); //stream框架初始化。
    while(1){
        ecbm_stream_main();//放到主循环不断执行。
    }
}
```

注意事项

1. 本函数执行的间隔决定了通信回复的快慢。如果像例程那样主循环只调用本函数，那么通信回复是即时的。如果主循环执行的事情比较多，那么通信回复就比较慢，毕竟只有执行到本函数的时候，才能对串口数据进行解析和回复。
2. 如果确实主循环执行的事比较多，那么可能在这段时间内接收的数据会超过缓存的的大小，因此要是遇到这样的情况，请加大缓存的大小。在stream.h那里可以设置。

ecbm_stream_exe

函数原型：void ecbm_stream_exe(u8 dat);

描述

流处理程序函数。

输入

- dat：串口收到的数据。

输出

无

返回值

无

调用例程

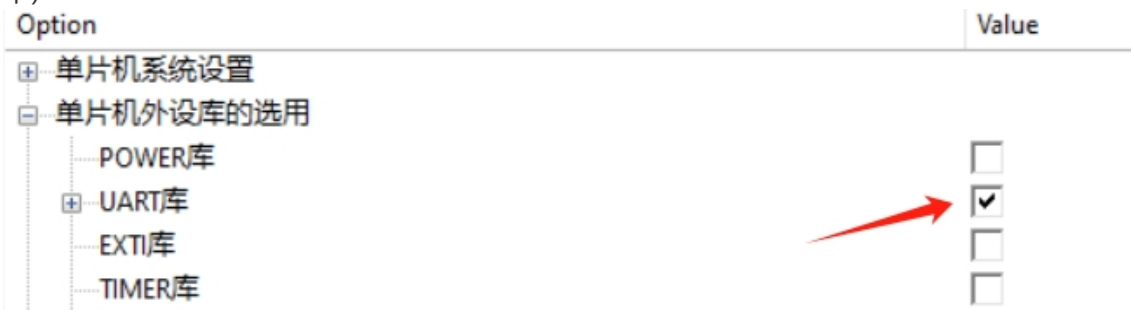
```
#include "ecbm_core.h" //加载库函数的头文件。
void main(void){ //main函数，必须的。
    system_init(); //系统初始化函数，也是必须的。
    ecbm_stream_init(); //stream框架初始化。
    while(1){
        ecbm_stream_main();//放到主循环不断执行。
    }
}
void ecbm_stream_exe(u8 dat){
//这里是和组件对接的函数，所以在这里，只做定义，还没加上内容。待会说到组件的时候再补充。
}
```

注意事项

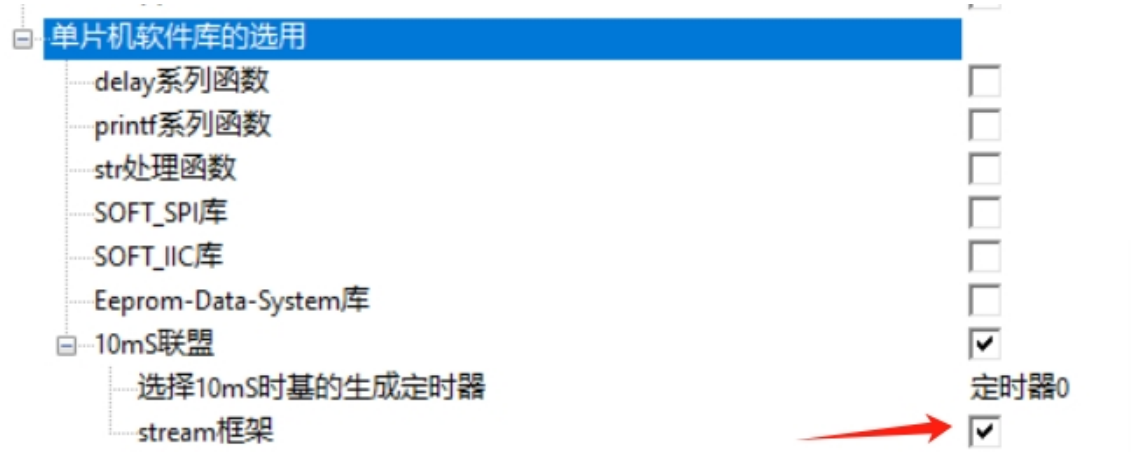
1. 本函数用于比较组件，将会在比较组件里细说。

应用步骤

1. stream框架是依附串口的解析框架，首先确保串口库的使能是打开的。（本设置在ecbm_core.h中）



2. stream隶属于10mS联盟，所以也得确保“10mS联盟”的使能是打开的。接着使能“stream框架”。（本设置在ecbm_core.h中）



3. 请至少使能一个stream通道，否则编译会出警告。通道1对应串口1，通道2对应串口2，以此类推。上面的【通道配置】可能说明会更详细些。（本设置在stream.h中）

4. 将stream的初始化函数和stream的主函数放入你自己的主函数中（代码示例是写在main.c中）：

```
#include "ecbm_core.h"  //加载库函数的头文件。
void main(void){        //main函数，必须的。
    system_init();      //系统初始化函数，也是必须的。
    ecbm_stream_init(); //stream框架初始化。
    while(1){
        ecbm_stream_main();//stream框架主函数。
    }
}
```

至此，stream框架已经搭建成功！

比较组件

比较组件的使用方法就是通过比较串口接收的数据和给定字符串的比较值，当比较值和字符串的字数相等的时候，说明比较成功。

API

ecbm_stream_strcmp

函数原型：void ecbm_stream_strcmp(u8 dat,u8 code * str,u8 * count);

描述

流处理比对函数。

输入

- dat：和ecbm_stream_user_exe对接的接口。
- str：需要比对的字符串。

输出

- count：比对的计数值，当该值和字符串长度相等时，表示比对成功。

返回值

无

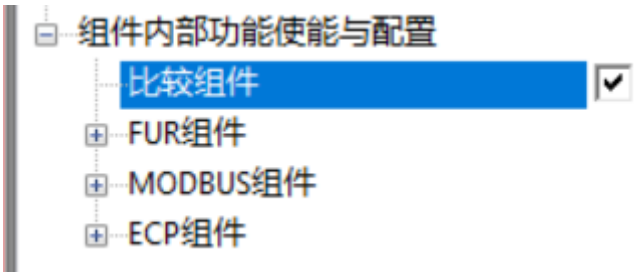
调用例程

```
#include "ecbm_core.h" //加载库函数的头文件。
void main(void){ //main函数，必须的。
    system_init(); //系统初始化函数，也是必须的。
    ecbm_stream_init(); //stream框架初始化。
    gpio_mode(D55,GPIO_OUT);//初始化该引脚为推挽模式，电路上连接着一个LED，低电平点亮。
    while(1){
        ecbm_stream_main();//跑stream框架。
    }
}
u8 cmp_on=0,cmp_off=0;//存比较值的变量。
void ecbm_stream_user_exe(u8 ch,u8 dat){
    if(ch==1){//如果是串口1接收到的数据，
        ecbm_stream_strcmp(dat,"LED_ON" ,&cmp_on );//和字符串LED_ON比较。
        ecbm_stream_strcmp(dat,"LED_OFF",&cmp_off);//和字符串LED_OFF比较。
```

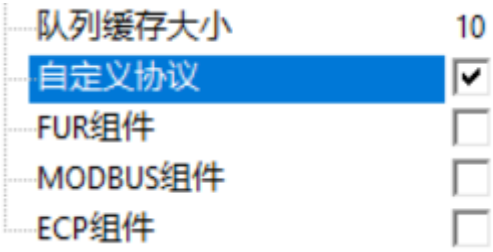
```
        if(cmp_on ==sizeof("LED_ON")-1) {P55=0;}//和字符串LED_ON比较，有6个字符正常的时，说明比较成功。点亮LED。
        if(cmp_off==sizeof("LED_OFF")-1){P55=1;}//和字符串LED_OFF比较，有7个字符正常的时，说明比较成功。熄灭LED。
        //因为sizeof()函数会把字符串结尾的/0也计算在内，所以要减1。
    }
}
```

注意事项

1. 使用之前必须打开本组件使能，在stream.h里可以看到使能设置：



然后比较的内容是非常自由的，所以只能用户自己去定义，因此还要在【通道配置】那里打开【自定义协议】的使能。



2. 字符串只能是静态的，若想实现动态字符串，可仿制本函数的实现方法自己写一个。
3. 串口助手发送字符串时，一定得注意大小写。因为本函数比较的是字符的ASCII码，“LED_ON”和“led_on”对于本函数而言是不一样的。

FUR组件

FUR组件是Fast-Uart-Reg的缩写，直译为“快速-串口-寄存器”。这3个词描述了这个组件的特点：

1. 快速上手使用；
2. 基于串口；
3. 直接读写单片机的寄存器。

同时fur有毛发的意思，因为这个组件也像毛发一样轻盈。为何轻盈？因为FUR的构造足够简单，不管指令怎么变化，最终的执行效果就只有读和写两个动作而已。

使用攻略

要想使用FUR就得明白一个寄存器的概念：一个寄存器应该具备一个访问地址，同时一个寄存器能存放一定量的数据。在参考工业最常用的modbus协议之后，fur的寄存器定义为16位寄存器，并具备16位地址。也就是说最大可访问65536个u16型寄存器。同时有8位的ID位，也就是能支持ID为0~255共256个器件共同使用。那么下面就介绍详细的指令：

读指令

[地址]?;

[地址@id]?;

该指令用于读取某一个地址，或者某一个器件的某一个地址的寄存器的值。比如"[0]?;"就是查询地址为0的寄存器的值；又比如"[1@3]?."就是查询ID为3的器件里的1号寄存器的值。

加指令

[地址]+=数值;

[地址@id]+=数值;

该指令用于向某一个地址，或者向某一个器件的某一个地址的寄存器加上一个数值。比如"[0]+=5;"就是把0号寄存器的值加上5，若原来的值是2，那么该指令之后寄存器的值为7。多个器件在总线上时加"@id"来限定接收指令的器件。

减指令

[地址]-=数值;

[地址@id]-=数值;

该指令用于向某一个地址，或者向某一个器件的某一个地址的寄存器减去一个数值。比如"[0]-=5;"就是把0号寄存器的值减去5，若原来的值是20，那么该指令之后寄存器的值为15。多个器件在总线上时加"@id"来限定接收指令的器件。

乘指令

[地址]*=数值;

[地址@id]*=数值;

该指令用于向某一个地址，或者向某一个器件的某一个地址的寄存器乘以一个数值。比如"[0]*=5;"就是把0号寄存器的值乘以5，若原来的值是2，那么该指令之后寄存器的值为10。多个器件在总线上时加"@id"来限定接收指令的器件。

除指令

[地址]/=数值;

[地址@id]/=数值;

该指令用于向某一个地址，或者向某一个器件的某一个地址的寄存器除以一个数值。比如"[0]/=5;"就是把0号寄存器的值除以5，若原来的值是24，那么该指令之后寄存器的值为4(整型寄存器、小数部分直接去掉)。多个器件在总线上时加"@id"来限定接收指令的器件。

与指令

[地址]&=数值;

[地址@id]&=数值;

该指令用于向某一个地址，或者向某一个器件的某一个地址的寄存器与上一个数值。比如"[0]&=0xFFF0;"就是把0号寄存器的值与上0xFFF0，若原来的值是24(0x0018)，那么该指令之后寄存器的值为16(0x0018&0xFFF0等于0x0010)。多个器件在总线上时加"@id"来限定接收指令的器件。

或指令

[地址]|=数值;

[地址@id]|=数值;

该指令用于向某一个地址，或者向某一个器件的某一个地址的寄存器或上一个数值。比如"[0]|=3;"就是把0号寄存器的值与上0x0003，若原来的值是24(0x0018)，那么该指令之后寄存器的值为27(0x0018|0x0003等于0x001B)。多个器件在总线上时加"@id"来限定接收指令的器件。

异或指令

[地址]^=数值;

[地址@id]^=数值;

该指令用于向某一个地址，或者向某一个器件的某一个地址的寄存器异或上一个数值。比如"[0]^=15;"就是把0号寄存器的值与上0x000F，若原来的值是24(0x0018)，那么该指令之后寄存器的值为23(0x0018^0x000F等于0x0017)。多个器件在总线上时加"@id"来限定接收指令的器件。

位操作指令

[地址].位数=数值;

[地址@id].位数=数值;

该指令用于向某一个地址，或者向某一个器件的某一个地址的寄存器的其中一位置一或置零。比如"[0].1=1;"就是把0号寄存器的D1值置一，若原来的值是24(0x0018)，那么该指令之后寄存器的值为26(D1位置一后0x0018变成0x001A)。位数可填0~15，数值最好只填写0或者1。多个器件在总线上时加"@id"来限定接收指令的器件。

赋值指令

[地址]=数值;

[地址@id]=数值;

该指令用于向某一个地址，或者向某一个器件的某一个地址的寄存器赋予一个数值。比如"[0]=15;"就是把0号寄存器赋值15，若原来的值是24，那么该指令之后寄存器的值为15。多个器件在总线上时加"@id"来限定接收指令的器件。

总结

看起来好像指令很多，但主框架只有两个，一个写一个读。

对寄存器有写入操作的：[寄存器地址@器件ID]操作数=数值;

对寄存器有读取操作的：[寄存器地址@器件ID]?;

- 寄存器地址：0~65535。
- 器件ID：0~255。
- 操作数：**+ - * / & | ^ .****位数**(位数范围：0~15)。无操作数就是直接赋值。
- 数值：0到65535。当然也支持十六进制写法0x0000到0xFFFF。

API

es_fur_get_id

函数原型：u8 es_fur_get_id(u8 ch);

描述

FUR获取ID函数，这个函数需要返回本机的ID号。

输入

- ch：本次通信的来源，通道号。

输出

无

返回值

- 本机的ID号。

调用例程

返回固定ID：

```
u8 es_fur_get_id(u8 ch){
    ch=ch;//演示中，通道忽略。
    return 1;
}
```

返回可变ID：

```
u8 mcu_id=1;
u8 es_fur_get_id(u8 ch){
    ch=ch;//演示中，通道忽略。
    return mcu_id;
}
```

注意事项

1. 因为本协议是含有地址的，在多个器件共用一条串口总线（比如485总线）时，可以通过地址来区分和哪个器件通信。因此本函数不能忽略，一定得定义出来。
2. 可变ID应用于那些可更改ID号的器件，通常ID更改后还要能保存，但保存ID的操作不是本协议的讨论范围就不放出来了。
3. **这个函数是必须要定义的！**

es_fur_read_reg

函数原型：u16 es_fur_read_reg(u8 ch,u16 addr);

描述

FUR读取寄存器函数。

输入

- ch：本次通信的来源，通道号。
- addr：要读取数据的寄存器的地址。

输出

无

返回值

- 该地址对应的寄存器的数据。

调用例程

写法1：

```
u16 reg[20];//某处定义的数组，名字的长度都随意，不一定非得叫reg，这只是举例。
u16 es_fur_read_reg(u8 ch,u16 addr){
    ch=ch;//演示中，通道忽略。
    return reg[addr];//返回数组的值或者是其他变量。
}
```

写法2：

```
u16 reg[20];//某处定义的数组，名字的长度都随意，不一定非得叫reg，这只是举例。
u16 es_fur_read_reg(u8 ch,u16 addr){
    ch=ch;//演示中，通道忽略。
    if(addr==0){//当读取地址是0的时候，
        return (u16)(P0);//返回P0的电平值。
    }else if(addr==1){//当读取地址是1的时候，
        return (u16)(P1);//返回P1的电平值。
    }else if(addr==2){//当读取地址是2的时候，
        return (u16)(P2);//返回P2的电平值。
    }else if(addr==3){//当读取地址是3的时候，
        return (u16)(P3);//返回P3的电平值。
    }else{//如果是其他地址，
        return reg[addr-4];//就返回数组reg的值，因为上面占用了0~3地址，所以地址为4的时候才对应数组的第一个数，因此要减4。
    }
}
```

注意事项

1. 本函数会在上位机发送一个读指令或者其他涉及到读取的指令的时候执行。比如上位机发送**[1]?;**解析之后会调用本函数，同时参数addr的值为1。若按上面的例子来看，假如P1的值为0x05，那么本函数返回0x05。上位机就会收到**(1)=5;**
2. **这个函数是必须要定义的！**

es_fur_write_reg

函数原型：void es_fur_write_reg(u8 ch,u16 addr,u16 dat);

描述

FUR写入寄存器函数。

输入

- ch：本次通信的来源，通道号。
- addr：要写入数据的寄存器的地址。
- dat：要写入的数据。

输出

无

返回值

无

调用例程

存入缓存：

```
void es_fur_write_reg(u8 ch,u16 addr,u16 dat){
    ch=ch;//演示中，通道忽略。
    reg[addr]=dat;//存入缓存
}
```

作为触发：

```
void es_fur_write_reg(u8 ch,u16 addr,u16 dat){
    ch=ch;//演示中，通道忽略。
    if(addr==0){//当地址为0时，
        P0=(u8)(dat);//将dat值发到P0。
    }else if(addr==1){//当地址为1时，
        if(dat){//若dat不是0，
            LED_ON;//点亮LED。
        }else{//否则，
            LED_OFF;//熄灭LED。
        }
    }
}
```

注意事项

1. 本函数会在上位机发送一个写指令或者其他涉及到写入的指令的时候执行。比如上位机发送**[1]=123;**解析之后会调用本函数，同时参数addr的值为1，参数dat的值为123。
2. **这个函数是必须要定义的！**

es_fur_master_receive_callback

函数原型：void es_fur_master_receive_callback(u8 ch,u16 addr,u16 dat);

描述

FUR主机接收回调函数，当本机作为主机发送指令给从机之后，从机返回数据时会调用本函数。

输入

- ch：本次通信的来源，通道号。
- addr：从机返回的寄存器地址。
- dat：从机的该地址的数据。

输出

无

返回值

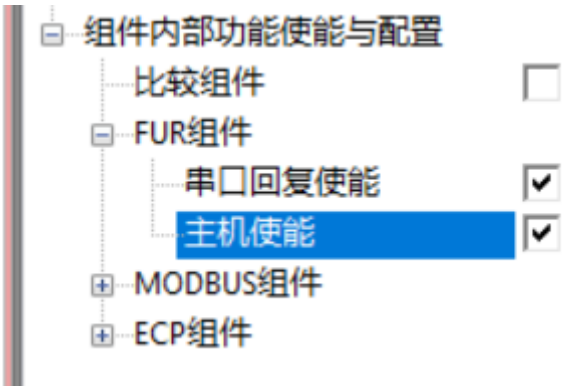
无

调用例程

```
//假如从机模块的15号寄存器里存着模块测量到的温度。
//定义回调函数
void es_fur_master_receive_callback(u8 ch,u16 addr,u16 dat){
    ch=ch;//演示中，通道忽略。
    if(addr==15){//当从机返回的数据是15号寄存器时，
    oled_printf(&oled,0,0,"temp=%d",dat);//在OLED上打印出温度信息。
    }
}
//在main里执行读取函数就行了。
void main(){
    ...//其他代码
    while(1){
        if(temp_flag){//到了更新温度的时候了，
            temp_flag=0;//先清零标志位。
            es_fur_master_read(15,0);//向从机发送读取15号的指令。收到回复的时候会执行回调
函数。
        }
    }
}
```

注意事项

1. 本函数在使用前，需要使能FUR主机。



2. 如果不用主机功能，那么这个函数就没必要定义。

es_fur_master_send

函数原型：void es_fur_master_send(u16 addr,u8 id,u16 dat);

描述

FUR主机发送函数，用于向一个支持FUR的设备发送一个数据。

输入

- addr：要发送的目标寄存器地址。
- id：目标设备的ID。
- dat：要发送的数据。

输出

无

返回值

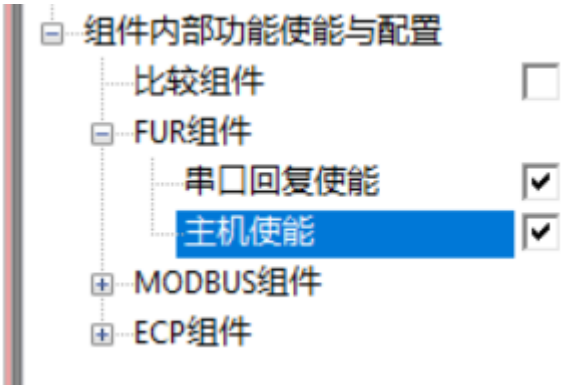
无

调用例程

```
void main(){
    ...//其他代码
    while(1){
        ...//其他代码
        if(key==0x25){//按下某个按键的时候。
            es_fur_master_send(15,1,0);//向1号从机发送的15号寄存器赋值为0。
        }
    }
}
```

注意事项

1. 本函数在使用前，需要使能FUR主机，否则不会参与编译。



es_fur_master_read

函数原型：void es_fur_master_read(u16 addr,u8 id);

描述

FUR主机读取函数，用于读取一个支持FUR的设备的数据。

输入

- addr：要读取的目标寄存器地址。
- id：目标设备的ID。

输出

无

返回值

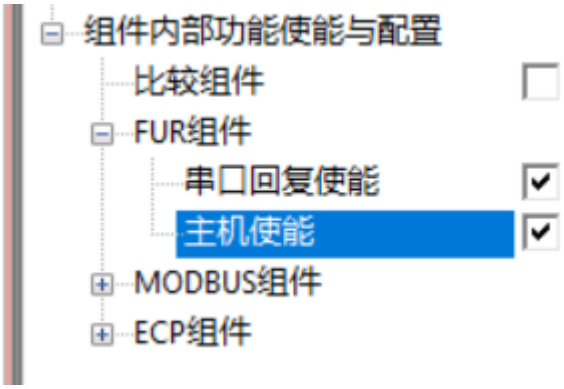
无

调用例程

```
//假如从机模块的15号寄存器里存着模块测量到的温度。
//定义回调函数
void es_fur_master_receive_callback(u8 ch,u16 addr,u16 dat){
    ch=ch;//演示中，通道忽略。
    if(addr==15){//当从机返回的数据是15号寄存器时，
        oled_printf(&oled,0,0,"temp=%d",dat);//在OLED上打印出温度信息。
    }
}
//在main里执行读取函数就行了。
void main(){
    ...//其他代码
    while(1){
        if(temp_flag){//到了更新温度的时候了，
            temp_flag=0;//先清零标志位。
            es_fur_master_read(15,0);//向从机发送读取15号的指令。收到回复的时候会执行回调函数。
        }
    }
}
```

注意事项

1. 本函数在使用前，需要使能FUR主机，否则不会参与编译。

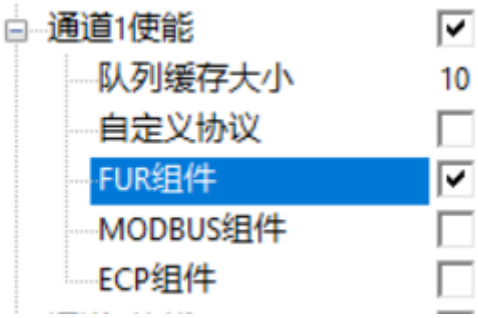


应用步骤

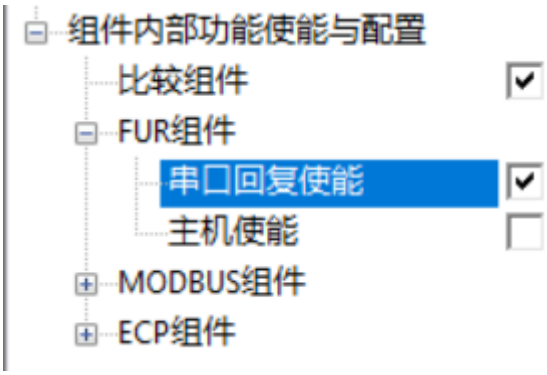
从机的用法

1. 在stream.h中，选择你要解析的通道下面的FUR组件使能，比如想在串口1使用FUR就选择通道1。

   每个通道的使能都是独立的。



2. 在【组件内部功能使能与配置】中，有两项可以选择：【串口回复使能】和【主机使能】。



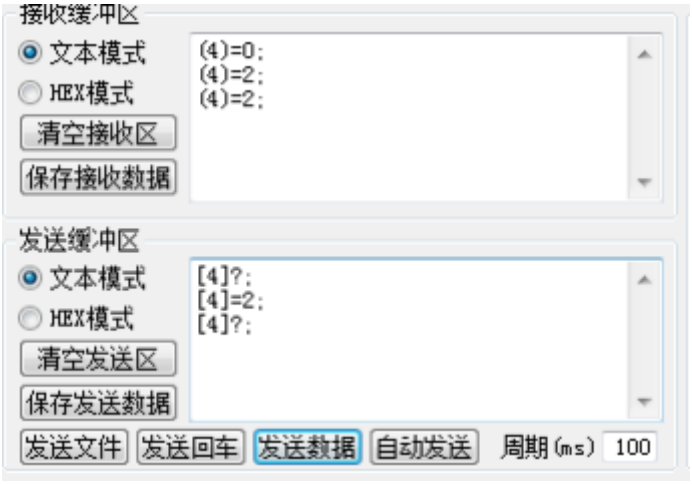
   它们都是优化性质的选项，如果你用不到就可以取消掉以节省空间。

3. 定义好**es_fur_get_id**、**es_fur_read_reg**和**es_fur_write_reg**这3个函数就OK了。

标准最小化FUR从机代码一览：

```
#include "ecbm_core.h" //加载库函数的头文件。
void main(void){ //main函数，必须的。
    system_init(); //系统初始化函数，也是必须的。
    ecbm_stream_init(); //stream框架初始化。
    while(1){
        ecbm_stream_main();//跑stream框架。
    }
}
u8 es_fur_get_id(u8 ch){
    ch=ch;//演示中，通道忽略。
    return 1;//0是广播地址，所以不能是0。
}
u16 yanshi=0;
u16 es_fur_read_reg(u8 ch,u16 addr){
    ch=ch;//演示中，通道忽略。
    addr=addr;//演示代码而已。
    return yanshi;//演示代码而已。
}
void es_fur_write_reg(u8 ch,u16 addr,u16 dat){
    ch=ch;//演示中，通道忽略。
    addr=addr;//演示代码而已。
    yanshi=dat;//演示代码而已。
}
```

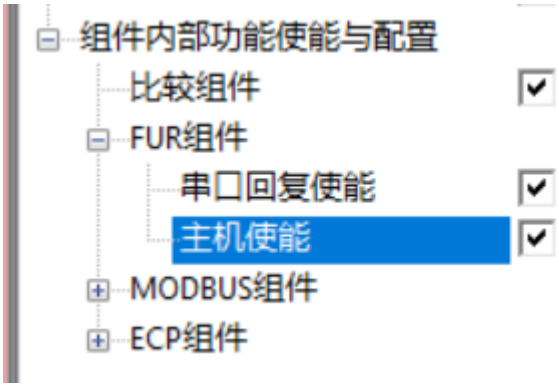
如上的代码会对变量yanshi进行读写，此时用串口助手发送3条指令“[4]?;”、“[4]=2;”和“[4]?;”。用处就是先查询地址为4的寄存器值，然后写2到寄存器4中，最后再查询寄存器4的值。串口反馈如下：



可以看到代码里yanshi初始化为0，所以第一次查询是0。接着写入2，返回写入之后值变成了2。最后再查询一遍验证看看，还是2说明写入成功。不过一般串口通信没有被干扰的话，也不用特地去验证的，直接看写入指令返回的值就知道写入成功与否。

主机的用法

FUR最初是作为电脑调试来定制的，所以单片机端恒为从机。不过由于指令比较简洁明了，作为单片机之间通信的协议也不无不可。用法和从机的差不多，只是在从机的基础上，打开【主机使能】。



然后定义好回调函数**es_fur_master_receive_callback**就可以了。主机根据需求调用**es_fur_master_send**或者**es_fur_master_read**函数和从机通信，当收到从机返回的数据之后就会调用**es_fur_master_receive_callback**函数。这几个函数的详情可以参考API里的描述。

modbus-rtu组件

简介

modbus-rtu组件就是常说的MODBUS了，是一个在工业上非常常见的通信协议。它有很多种变种，其中用的最多的就是modbus-rtu和modbus-ascii。他们的区别就是rtu以原始数据表示数据，ascii以字符形式表示数据。ascii更加方便人类读取，rtu在解码方面更方便机器读取，所以这里采用的是modbus-rtu。

使用攻略

modbus的用法、原理和通信格式在网上随处可见，这里就不赘述了。本组件收纳了常用的7个指令：

1. 【01】读线圈。
2. 【05】写单个线圈。
3. 【03】读寄存器。
4. 【06】写单个寄存器。
5. 【10】写多个寄存器。

6. 【02】读离散量输入。
7. 【04】读输入寄存器。

因为modbus是比较常见的工业协议，且内容比较多，所以下面将分为从机API和主机API来讲解，可以从左侧的目录快速达到想要看到的位置。

从机API

es_modbus_rtu_get_id

函数原型：u8 es_modbus_rtu_get_id(u8 ch);

描述

获取本机ID函数，modbus通讯中会调用，要返回本机的ID号。

输入

- ch：数据来源的通道。

输出

无

返回值

- 本机的ID号。

调用例程

返回固定ID：

```
u8 es_modbus_rtu_get_id(u8 ch){
    ch=ch;//演示中，通道忽略。
    return 1;
}
```

返回可变ID：

```
u8 mcu_id=1;
u8 es_modbus_rtu_get_id(u8 ch){
    ch=ch;//演示中，通道忽略。
    return mcu_id;
}
```

注意事项

1. 因为本协议是含有地址的，在多个器件共用一条串口总线（比如485总线）时，可以通过地址来区分和哪个器件通信。因此本函数不能忽略，一定得定义出来。
2. 可变ID应用于那些可更改ID号的器件，通常ID更改后还要能保存，但保存ID的操作不是本协议的讨论范围就不放出来了。
3. **这个函数是必须要定义的！**

es_modbus_cmd_read_io_bit

函数原型：void es_modbus_cmd_read_io_bit(u8 ch,u16 addr,u8 * dat);

描述

读取离散输入寄存器函数，功能码02H会用到。

输入

- ch：数据来源的通道。
- addr：主机传来的地址信息。

输出

- dat：该地址对应的离散输入寄存器的数据，只有0和1两种可能。

返回值

无

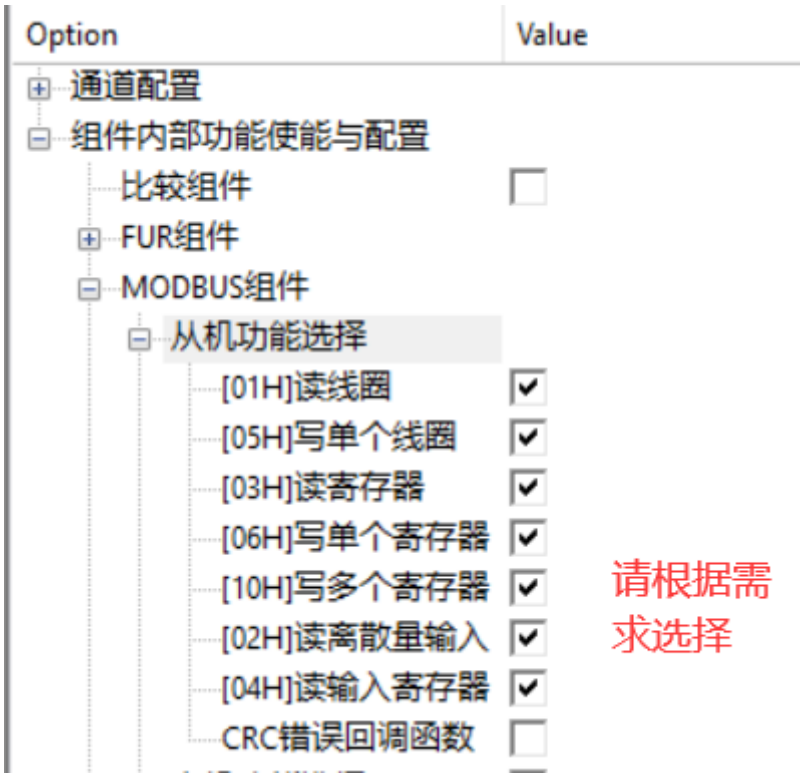
调用例程

```
void es_modbus_cmd_read_io_bit(u8 ch, addr,u8 * dat){
    ch=ch;//演示中，通道忽略。
    if(addr==0x0000){//当主机要读地址0的时候，
        if(P00){//例程中，这个地址对应着P0.0的状态。
            *dat=1;//是高电平就返回1。
        }else{
            *dat=0;//是低电平就返回0。
        }
    }
}
```

注意事项

1. 本函数是指令【02H】的支持函数，当上位机发送【02H】指令时，就会执行本函数；上位机要读取的地址会填充到参数addr上，需要你用dat来返回这个地址所对应的值。离散输入寄存器和线圈寄存器类似，都是一个位的寄存器，也就是只有0和1两个值。
2. **本函数是从机函数，且只有【02H】指令才会调用。因此如果没有用到【02H】指令就不需要定义本函数。**

3. 本库默认打开所有指令的使能，如果项目没有用到【02H】指令，可在stream.h中把【02H】的使能关掉。



es_modbus_cmd_read_io_reg

函数原型：void es_modbus_cmd_read_io_reg(u8 ch,u16 addr,u16 * dat);

描述

读取输入寄存器函数，功能码04H会用到。

输入

- ch：数据来源的通道。
- addr：主机传来的地址信息。

输出

- dat：该地址对应的输入寄存器的数据。

返回值

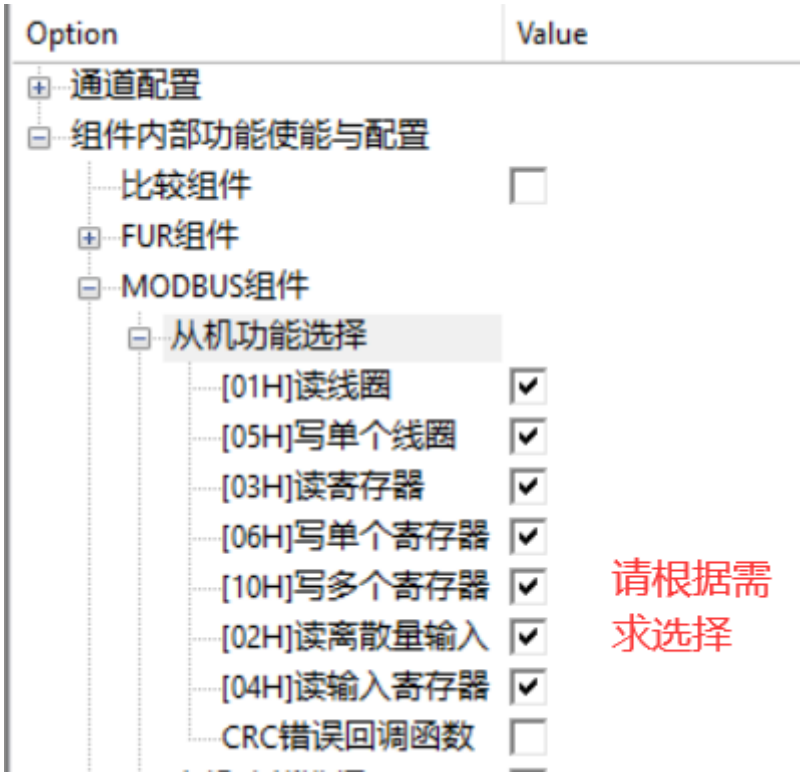
无

调用例程

```
void es_modbus_cmd_read_io_reg(u8 ch, addr,u16 * dat){
    ch=ch;//演示中，通道忽略。
    if(addr==0x0000){//当主机要读地址0的时候，
        *dat=(u16)(P0);//例程中，这个地址对应着P0的状态。但P0是8位的，最好强转类型成16位。
    }
}
```

注意事项

1. 本函数是指令【04H】的支持函数，当上位机发送【04H】指令时，就会执行本函数；上位机要读取的地址会填充到参数addr上，需要你用dat来返回这个地址所对应的值。输入寄存器是16位的寄存器。
2. **本函数是从机函数，且只有【04H】指令才会调用。因此如果没有用到【04H】指令就不需要定义本函数。**
3. 本库默认打开所有指令的使能，如果项目没有用到【04H】指令，可在stream.h中把【04H】的使能关掉。



es_modbus_cmd_write_bit

函数原型：void es_modbus_cmd_write_bit(u8 ch,u16 addr,u8 dat);

描述

写单个线圈寄存器函数，功能码05H会用到。

输入

- ch：数据来源的通道。
- addr：主机传来的地址信息。
- dat：要写入该地址的数据。

输出

无

返回值

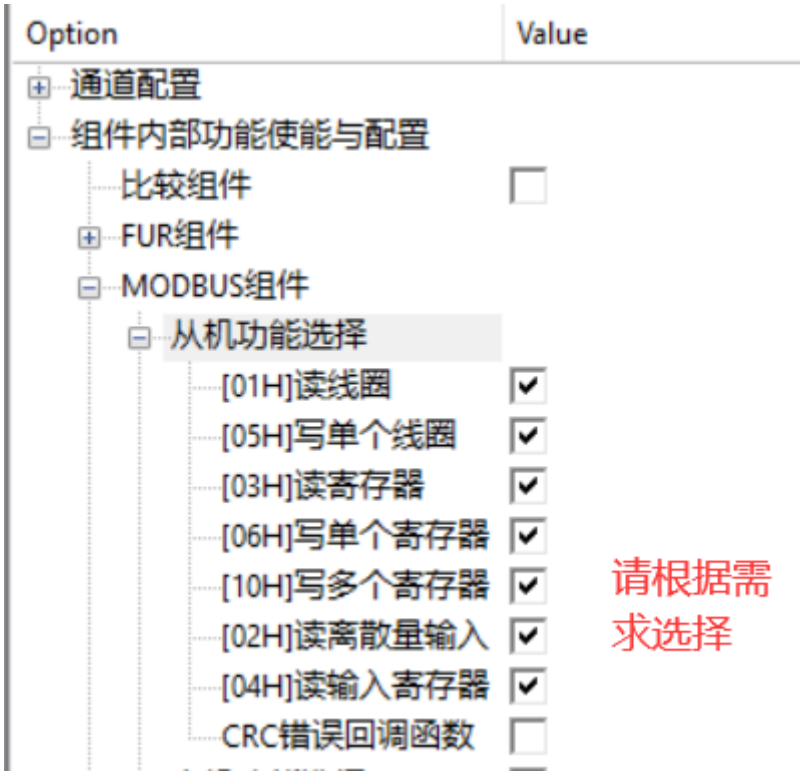
无

调用例程

```
u16 test=0x1234;//假设有个变量test，然后test的D0位映射到线圈0x0000号。
void es_modbus_cmd_write_bit(u8 ch,u16 addr,u8 dat){
    ch=ch;//演示中，通道忽略。
    if(addr==0x0000){//当主机要写地址0的线圈时候，
        if(dat){//例程中，这个地址对应着变量test的D0位。
            test|=0x0001;//如果写入的是1，就令D0位变成1。
        }else{
            test&=0xFFFE;//如果写入的是0，就令D0位变成0。
        }
    }
}
```

注意事项

1. 本函数是指令【05H】的支持函数，当上位机发送【05H】指令时，就会执行本函数；上位机要写入的地址会填充到参数addr上，上位机要写入的数据填充到参数dat上。线圈寄存器是一位寄存器，就是说只有0和1。
2. **本函数是从机函数，且只有【05H】指令才会调用。因此如果没有用到【05H】指令就不需要定义本函数。**
3. 本库默认打开所有指令的使能，如果项目没有用到【05H】指令，可在stream.h中把【05H】的使能关掉。



es_modbus_cmd_read_bit

函数原型：void es_modbus_cmd_read_bit(u8 ch,u16 addr,u8 * dat);

描述

读单个线圈寄存器函数，功能码01H会用到。

输入

- ch：数据来源的通道。
- addr：主机传来的地址信息。

输出

- dat：该地址对应的线圈寄存器的数据。

返回值

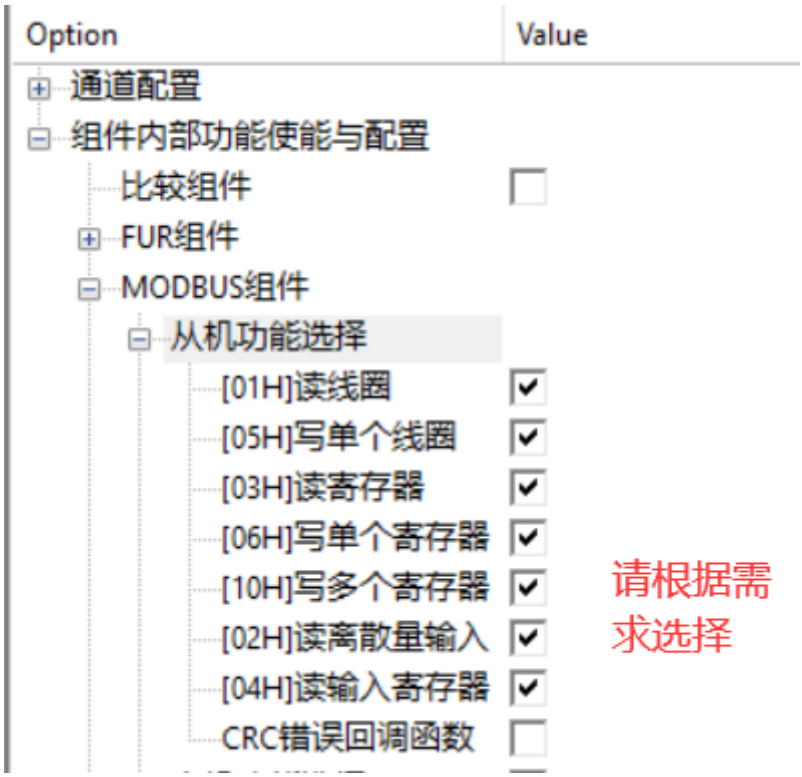
无

调用例程

```
u16 test=0x1234;//假设有个变量test，然后test的D0位映射到线圈0x0000号。
void es_modbus_cmd_read_bit(u8 ch,u16 addr,u8 * dat){
    ch=ch;//演示中，通道忽略。
    if(addr==0x0000){//当主机要读地址0的时候，
        if(test&0x0001){//例程中，这个地址对应着test变量的D0位。
            *dat=1;//D0位为1就返回1。
        }else{
            *dat=0;//D0位为0就返回0。
        }
    }
}
```

注意事项

1. 本函数是指令【01H】的支持函数，当上位机发送【01H】指令时，就会执行本函数；上位机要写入的地址会填充到参数addr上，需要你用dat来返回这个地址所对应的值。线圈寄存器是一位寄存器，就是说只有0和1。
2. **本函数是从机函数，且只有【01H】指令才会调用。因此如果没有用到【01H】指令就不需要定义本函数。**
3. 本库默认打开所有指令的使能，如果项目没有用到【01H】指令，可在stream.h中把【01H】的使能关掉。



es_modbus_cmd_write_reg

函数原型：void es_modbus_cmd_write_reg(u8 ch,u16 addr,u16 dat);

描述

写单个保持寄存器函数，功能码06H和10H会用到。

输入

- ch：数据来源的通道。
- addr：主机传来的地址信息。
- dat：要写入该地址的数据。

输出

无

返回值

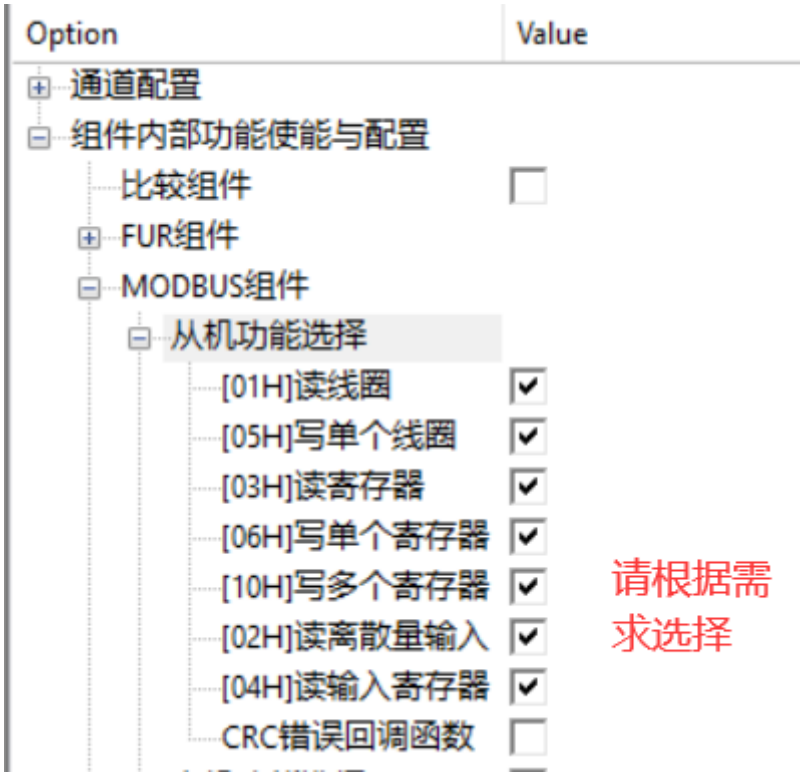
无

调用例程

```
u16 reg[20];
void es_modbus_cmd_write_reg(u8 ch,u16 addr,u16 dat){
    ch=ch;//演示中，通道忽略。
    reg[addr]=dat;//例程中的数组reg代表了保持寄存器。
}
```

注意事项

1. 本函数是指令【06H】【10H】的支持函数，当上位机发送【06H】或【10H】指令时，就会执行本函数；上位机要写入的地址会填充到参数addr上，上位机要写入的数据填充到参数dat上。保持寄存器是16位寄存器。
2. **本函数是从机函数，且只有【06H】或【10H】指令才会调用。因此如果【06H】和【10H】指令都没有用到就不需要定义本函数。**
3. 本库默认打开所有指令的使能，如果项目没有用到【06H】或【10H】指令，可在stream.h中把【06H】或【10H】的使能关掉。



es_modbus_cmd_read_reg

函数原型：void es_modbus_cmd_read_reg(u8 ch,u16 addr,u16 * dat);

描述

读单个保持寄存器函数，功能码03H会用到。

输入

- ch：数据来源的通道。
- addr：主机传来的地址信息。

输出

- dat：该地址对应的保持寄存器的数据。

返回值

无

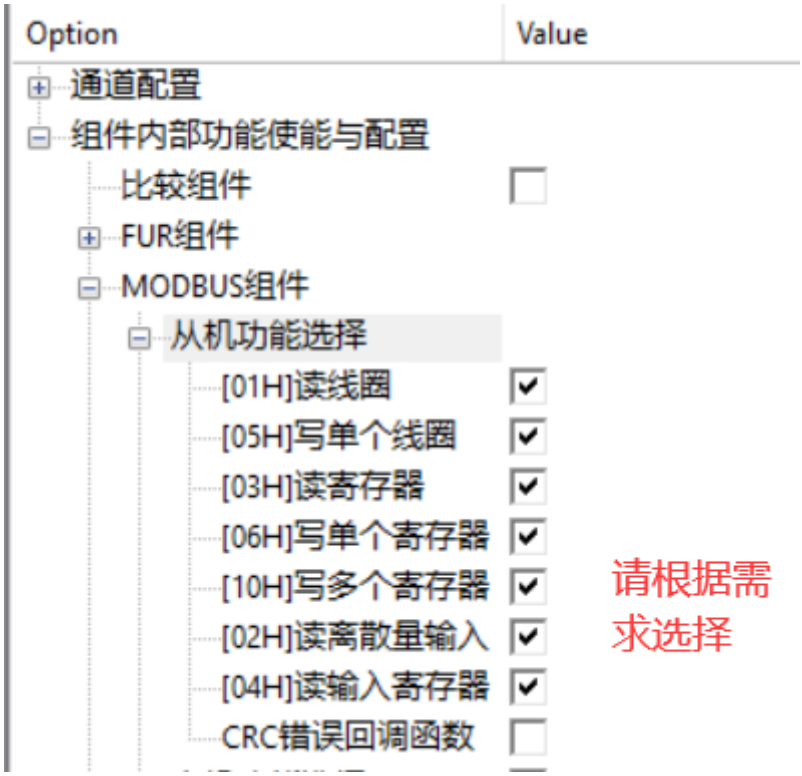
调用例程

```
u16 reg[20];
void es_modbus_cmd_read_reg(u8 ch,u16 addr,u16 * dat){
    ch=ch;//演示中，通道忽略。
    *dat=reg[addr];//例程中的数组reg代表了保持寄存器。
}
```

注意事项

1. 本函数是指令【03H】的支持函数，当上位机发送【03H】指令时，就会执行本函数；上位机要写入的地址会填充到参数addr上，需要你用dat来返回这个地址所对应的值。保持寄存器是16位寄存器。

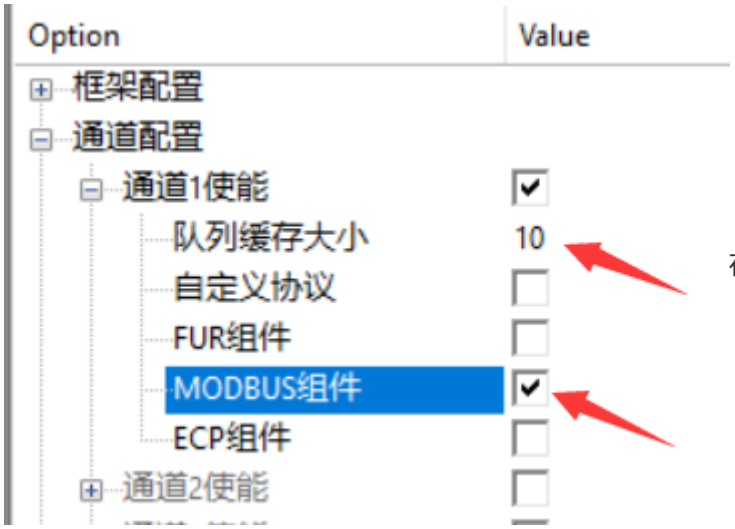
2. **本函数是从机函数，且只有【03H】指令才会调用。因此如果【03H】指令没有用到就不需要定义本函数。**
3. 本库默认打开所有指令的使能，如果项目没有用到【03H】指令，可在stream.h中把【03H】的使能关掉。

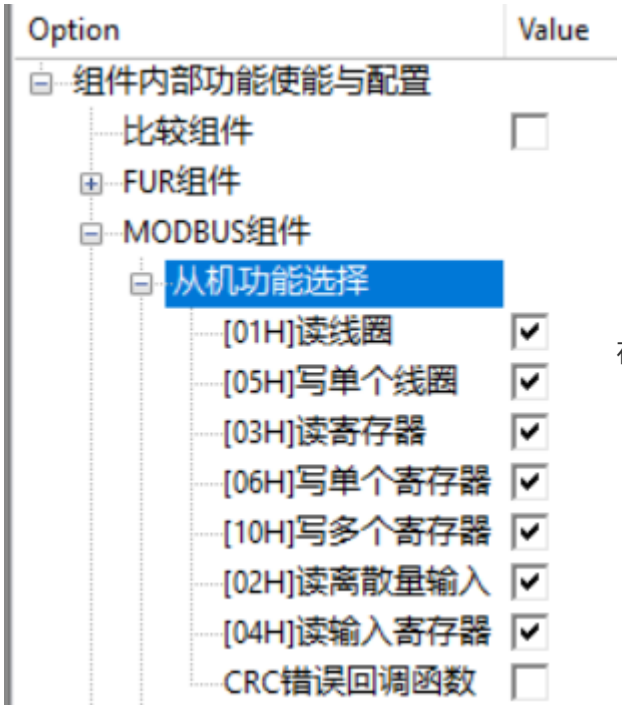


从机应用攻略

modbus主要就是从机应用，仔细看这部分，就绝对会用了！

1. 使能10mS联盟，可以在《10mS联盟.pdf》上看到步骤。
2. 使能stream框架，可以在本文的开头看到步骤。
3. 在通道配置里，选择【MODBUS组件】。如果你用的是串口1就是在通道1里选，用的是串口2就在通道2里选。在stream框架内可以支持4个串口都使用modbus，十分强大！当然在使用之前先估算一下大概一帧用到多少字节，然后修改【队列缓存大小】。如果无法估计，那么在ram允许的情况下尽量设大一些就好了，不过最好不要超过256。



4. 

在【组件内部功能使能与配置】里找到【MODBUS组件】，在下面的【从机功能选择】里选择需要编译的指令码。本库收录了这些常见的指令码，但也不是每个都会用在项目上。将不要的指令码划掉可以优化flash空间，因为库里做了宏定义处理，没有选择的指令码是不参与编译的。
5. 如果使能了【01H】读线圈，就必须定义**es_modbus_cmd_read_bit**函数。
6. 如果使能了【05H】写单个线圈，就必须定义**es_modbus_cmd_write_bit**函数。
7. 如果使能了【03H】读寄存器，就必须定义**es_modbus_cmd_read_reg**函数。
8. 如果使能了【06H】【10H】的任意一个写寄存器功能，就必须定义**es_modbus_cmd_write_reg**函数。
9. 如果使能了【02H】读离散量输入，就必须定义**es_modbus_cmd_read_io_bit**函数。
10. 如果使能了【04H】读输入寄存器，就必须定义**es_modbus_cmd_read_io_reg**函数。
11.

标准最小化modbus从机代码一览：

```
#include "ecbm_core.h" //加载库函数的头文件。
void main(void){ //main函数，必须的。
    system_init(); //系统初始化函数，也是必须的。
    while(1){
        ecbm_stream_main();
    }
}
u8 es_modbus_rtu_get_id(void){
    return 1;//可以为一个常数，也可以是一个变量。但是范围必须在1~247之间。
}
u16 test=0;
void es_modbus_cmd_read_io_bit(u16 addr,u8 * dat){
    if(addr==0x0000){//当主机要读地址0的时候，
        if(P00){//例程中，这个地址对应着P0.0的状态。
            *dat=1;//是高电平就返回1。
        }else{
            *dat=0;//是低电平就返回0。
        }
    }
}
void es_modbus_cmd_read_io_reg(u16 addr,u16 * dat){
    if(addr==0x0000){//当主机要读地址0的时候，
        *dat=(u16)P0;//例程中，这个地址对应着P0的状态。
```

```
    }
}
void es_modbus_cmd_write_bit(u16 addr,u8 dat){
    if(addr==0x0000){//当主机要写地址0的线圈时候，
        if(dat){//例程中，这个地址对应着变量test的D0位。
            test|=0x0001;//如果写入的是1，就令D0位变成1。
        }else{
            test&=0xFFFE;//如果写入的是0，就令D0位变成0。
        }
    }
}
void es_modbus_cmd_read_bit(u16 addr,u8 * dat){
    if(addr==0x0000){//当主机要读地址0的时候，
        if(test&0x0001){//例程中，这个地址对应着test变量的D0位。
            *dat=1;//D0位为1就返回1。
        }else{
            *dat=0;//D0位为0就返回0。
        }
    }
}
u16 reg[10]={0};
void es_modbus_cmd_write_reg(u16 addr,u16 dat){
    reg[addr]=dat;//例程中的数组reg代表了所有保持寄存器。
}
void es_modbus_cmd_read_reg(u16 addr,u16 * dat){
    *dat=reg[addr];//例程中的数组reg代表了所有保持寄存器。
}
```

注意事项

1. 里面的函数大都和指令有关，如果没有使能某些指令，就不需要定义其对应的函数。具体对应关系可以参考前面的内容。
2. 以上展示的是从机的示例结构，函数的内容是需要你自己的去实现的，示例里面的内容仅仅是为了演示。

es_modbus_rtu_set_slave_mode

函数原型：void es_modbus_rtu_set_slave_mode(void);

描述

设置从机模式函数，调用任意主机发送函数都会把modbus切换到主机模式，而后就得靠这个函数切换回来。

输入

无

输出

无

返回值

无

调用例程

```
void main(void){
    ...//其他代码。
    while(1){
        ...//其他代码。
        if(key==0x01){//假如按下了某个按键，
            es_modbus_rtu_master_0x01(1,1,0,1);//向1通道的1号从机发送【01H】指令，此时modbus组件会变成主机模式。
        }
    }
}
void es_modbus_rtu_master_0x01_callback(u8 ch,u8 id,u16 addr,u8 dat){//当接收到从机回复的时候。
    ch=ch;//演示中，通道忽略。
    if((id==0x01)&&(addr==0){//判断是不是1号从机返回的线圈0数据。
        if(dat){LED_1_ON;}else{LED_1_OFF;}//是1就点亮LED1，是0就熄灭LED1。下同。
        es_modbus_rtu_set_slave_mode();//由于数据接收处理完毕，调用本函数切换回从机模式。
    }
}
```

注意事项

1. 本函数一旦调用，modbus组件会立刻变成从机模式，请确保主机的信息接收完毕再转成从机，否则就会丢数据。
2. modbus组件没有切换主机模式函数，只要发送任意主机函数就能自动变成主机模式。

es_modbus_rtu_master_0x01

函数原型：void es_modbus_rtu_master_0x01(u8 ch,u8 id,u16 addr,u16 len);

描述

主机01号功能码发送函数。

输入

- ch：要发送该指令的通道。
- id：对方的ID。
- addr：要读的线圈地址。
- len：要读的数量。

输出

无

返回值

无

调用例程

```
es_modbus_rtu_master_0x01(1,1,0,3);//从串口1链接的ID为1的器件里读取3个线圈寄存器的值。由于线圈起始地址为0，那么就是读取0、1、2这3个地址。
```

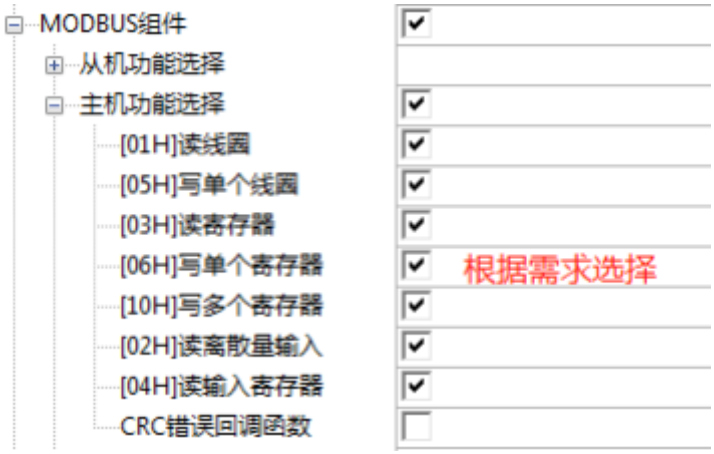
回调函数

在调用上面的主机发送函数之后，会等待从机的返回。如果从机返回了数据，将会执行回调函数。因此还需要定义一个回调函数，原型为：void es_modbus_rtu_master_0x01_callback(u16 addr,u8 dat);其中addr为从机返回的地址，dat为对应的数据。

```
void es_modbus_rtu_master_0x01_callback(u16 addr,u8 dat){//上面的例子里读取了3个线圈值。那么这里的例子就对这3个进行处理。
    if(addr==0){//判断线圈0，
        if(dat){LED_1_ON;}else{LED_1_OFF;}//是1就点亮LED1，是0就熄灭LED1。下同。
    }
    if(addr==1){
        if(dat){LED_2_ON;}else{LED_2_OFF;}
    }
    if(addr==2){
        if(dat){LED_3_ON;}else{LED_3_OFF;}
    }
}
```

注意事项

1. 本函数是指令【01H】的主机发送函数，使用时请确保从机支持【01H】指令。
2. 发送函数和发送回调函数是一一对应的，使用【01H】指令发送函数的时候，一定要定义对应的回调函数，否则无法知道从机返回的数据。就算不需要这个数据，回调函数也必须要定义。
3. 本函数是主机函数，所以要使用时不仅要打开主机使能，还要打开【01H】指令使能。



es_modbus_rtu_master_0x02

函数原型：void es_modbus_rtu_master_0x02(u8 id,u16 addr,u16 len);

描述

主机02号功能码发送函数。

输入

- id：对方的ID。
- addr： 要读的离散输入寄存器地址。
- len：要读的数量。

输出

无

返回值

无

调用例程

```
es_modbus_rtu_master_0x02(1,0,3);//从ID为1的器件里读取3个离散输入寄存器的值。由于起始地址为0，那么就是读取0、1、2这3个地址。
```

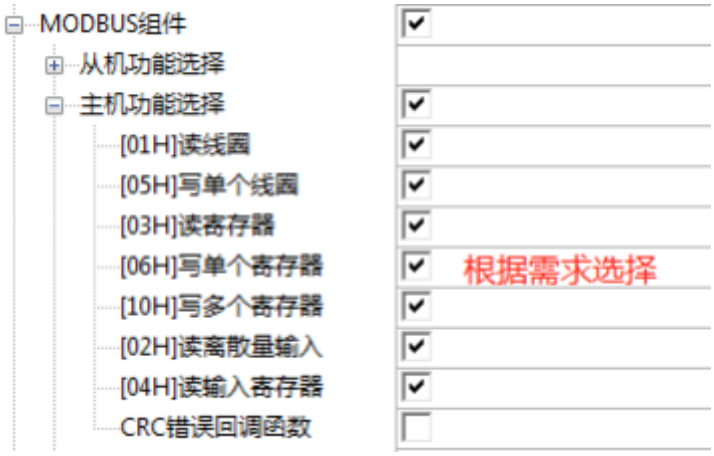
回调函数

在调用上面的主机发送函数之后，会等待从机的返回。如果从机返回了数据，将会执行回调函数。因此还需要定义一个回调函数，原型为：void es_modbus_rtu_master_0x02_callback(u16 addr,u8 dat);其中addr为从机返回的地址，dat为对应的数据。

```
void es_modbus_rtu_master_0x02_callback(u16 addr,u8 dat){//上面的例子里读取了3个线圈值。那么这里的例子就对这3个进行处理。
    if(addr==0){//判断线圈0，
        if(dat){LED_1_ON;}else{LED_1_OFF;}//是1就点亮LED1，是0就熄灭LED1。下同。
    }
    if(addr==1){
        if(dat){LED_2_ON;}else{LED_2_OFF;}
    }
    if(addr==2){
        if(dat){LED_3_ON;}else{LED_3_OFF;}
    }
}
```

注意事项

1. 本函数是指令【02H】的主机发送函数，使用时请确保从机支持【02H】指令。
2. 发送函数和发送回调函数是一一对应的，使用【02H】指令发送函数的时候，一定要定义对应的回调函数，否则无法知道从机返回的数据。就算不需要这个数据，回调函数也必须要定义。
3. 本函数是主机函数，所以要使用时不仅要打开主机使能，还要打开【02H】指令使能。



es_modbus_rtu_master_0x03

函数原型：void es_modbus_rtu_master_0x03(u8 id,u16 addr,u16 len);

描述

主机03号功能码发送函数。

输入

- id：对方的ID。
- addr： 要读的保持寄存器地址。
- len：要读的数量。

输出

无

返回值

无

调用例程

```
es_modbus_rtu_master_0x03(1,0,3);//从ID为1的器件里读取3个保持寄存器的值。由于起始地址为0，那么就是读取0、1、2这3个地址。
```

回调函数

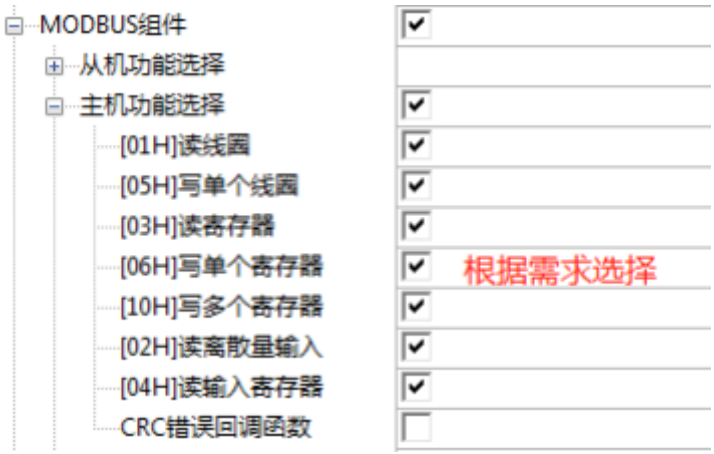
在调用上面的主机发送函数之后，会等待从机的返回。如果从机返回了数据，将会执行回调函数。因此还需要定义一个回调函数，原型为：void es_modbus_rtu_master_0x03_callback(u16 addr,u16 dat);其中addr为从机返回的地址，dat为对应的数据。

```
void es_modbus_rtu_master_0x03_callback(u16 addr,u16 dat){//上面的例子里读取了3个寄存器值。那么这里的例子就对这3个进行处理。
    if(addr==0){//判断地址0，
        oled_printf("[0]=%u",dat);//比如地址0需要显示到OLED。
    }
    if(addr==1){
        if(dat>100){BEEP_ON;}//比如地址1的值大于100就报警。
        if(dat<20){BEEP_OFF;}//小于20就恢复。
    }
    if(addr==2){
        test=dat;//比如地址2的内容就存起来。
    }
}
```

注意事项

1. 本函数是指令【03H】的主机发送函数，使用时请确保从机支持【03H】指令。
2. 发送函数和发送回调函数是一一对应的，使用【03H】指令发送函数的时候，一定要定义对应的回调函数，否则无法知道从机返回的数据。就算不需要这个数据，回调函数也必须要定义。

3. 本函数是主机函数，所以要使用时不仅要打开主机使能，还要打开【03H】指令使能。



es_modbus_rtu_master_0x04

函数原型：void es_modbus_rtu_master_0x04(u8 id,u16 addr,u16 len);

描述

主机04号功能码发送函数

输入

- id：对方的ID。
- addr：要读的输入寄存器地址。
- len：要读的数量。

输出

无

返回值

无

调用例程

```
es_modbus_rtu_master_0x04(1,0,3);//从ID为1的器件里读取3个输入寄存器的值。由于起始地址为0，那么就是读取0、1、2这3个地址。
```

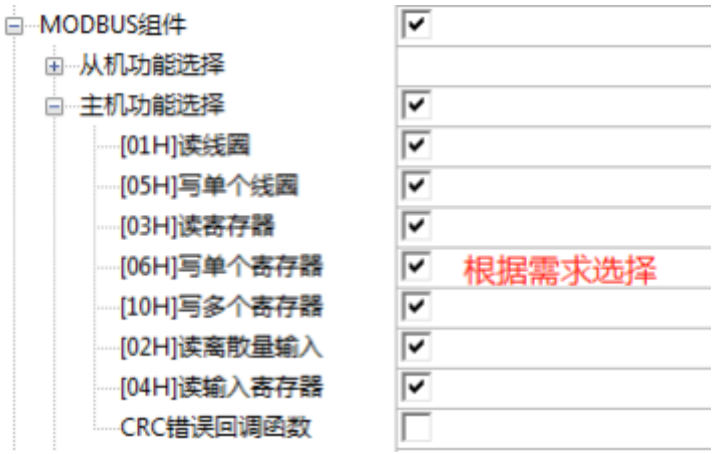
回调函数

在调用上面的主机发送函数之后，会等待从机的返回。如果从机返回了数据，将会执行回调函数。因此还需要定义一个回调函数，原型为：void es_modbus_rtu_master_0x04_callback(u16 addr,u16 dat);
其中addr为从机返回的地址，dat为对应的数据。

```
void es_modbus_rtu_master_0x04_callback(u16 addr,u16 dat){//上面的例子里读取了3个寄存器值。那么这里的例子就对这3个进行处理。
    if(addr==0){//判断地址0，
        oled_printf("[0]=%u",dat);//比如地址0需要显示到OLED。
    }
    if(addr==1){
        if(dat>100){BEEP_ON;}//比如地址1的值大于100就报警。
        if(dat<20){BEEP_OFF;}//小于20就恢复。
    }
    if(addr==2){
        test=dat;//比如地址2的内容就存起来。
    }
}
```

注意事项

1. 本函数是指令【04H】的主机发送函数，使用时请确保从机支持【04H】指令。
2. 发送函数和发送回调函数是一一对应的，使用【04H】指令发送函数的时候，一定要定义对应的回调函数，否则无法知道从机返回的数据。就算不需要这个数据，回调函数也必须要定义。
3. 本函数是主机函数，所以要使用时不仅要打开主机使能，还要打开【04H】指令使能。



es_modbus_rtu_master_0x05

函数原型：void es_modbus_rtu_master_0x05(u8 id,u16 addr,u8 dat);

描述

主机05号功能码发送函数。

输入

- id：对方的ID。
- addr：要写的线圈地址。
- len： 要写的数据。

输出

无

返回值

无

调用例程

```
es_modbus_rtu_master_0x05(1,0,1);//往ID为1的器件的0号线圈里写1。
```

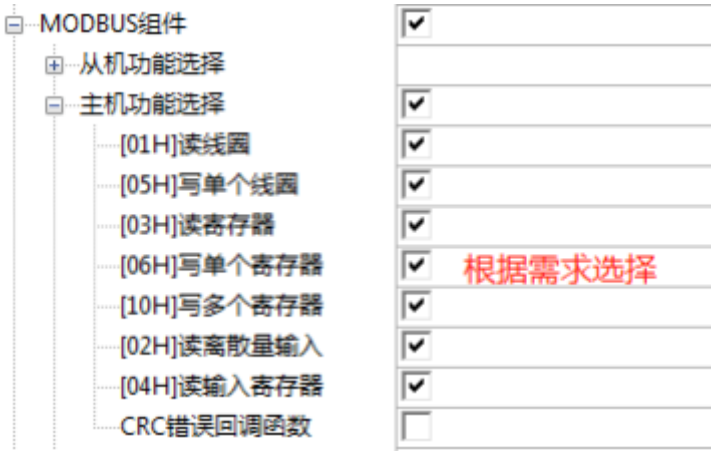
回调函数

在调用上面的主机发送函数之后，会等待从机的返回。如果从机返回了数据，将会执行回调函数。因此还需要定义一个回调函数，原型为：void es_modbus_rtu_master_0x05_callback(u16 addr,u8 dat);其中addr为从机返回的地址，dat为对应的数据。

```
void es_modbus_rtu_master_0x05_callback(u16 addr,u8 dat){//上面的例子里写了一个线圈值。
    if(addr==0){//返回是地址0，说明写入成功了。
        oled_printf("写入成功");//做个提示。
    }
}
```

注意事项

1. 本函数是指令【05H】的主机发送函数，使用时请确保从机支持【05H】指令。
2. 发送函数和发送回调函数是一一对应的，使用【05H】指令发送函数的时候，一定要定义对应的回调函数，否则无法知道从机返回的数据。就算不需要这个数据，回调函数也必须要定义。
3. 本函数是主机函数，所以要使用时不仅要打开主机使能，还要打开【05H】指令使能。



es_modbus_rtu_master_0x06

函数原型：void es_modbus_rtu_master_0x06(u8 id,u16 addr,u16 dat);

描述

主机06号功能码发送函数。

输入

- id：对方的ID。
- addr：要写的保持寄存器地址。
- len： 要写的数据。

输出

无

返回值

无

调用例程

```
es_modbus_rtu_master_0x06(1,0,0x123);//往ID为1的器件的0号保持寄存器里写0x123。
```

回调函数

在调用上面的主机发送函数之后，会等待从机的返回。如果从机返回了数据，将会执行回调函数。因此还需要定义一个回调函数，原型为：void es_modbus_rtu_master_0x06_callback(u16 addr,u16 dat);其中addr为从机返回的地址，dat为对应的数据。

```
void es_modbus_rtu_master_0x06_callback(u16 addr,u16 dat){//上面的例子里写了一个寄存器值。
    if(addr==0){//返回是地址0，说明写入成功了。
        oled_printf("写入成功");//做个提示。
    }
}
```

注意事项

1. 本函数是指令【06H】的主机发送函数，使用时请确保从机支持【06H】指令。
2. 发送函数和发送回调函数是一一对应的，使用【06H】指令发送函数的时候，一定要定义对应的回调函数，否则无法知道从机返回的数据。就算不需要这个数据，回调函数也必须要定义。
3. 本函数是主机函数，所以要使用时不仅要打开主机使能，还要打开【06H】指令使能。

es_modbus_rtu_master_0x10

函数原型：void es_modbus_rtu_master_0x10(u8 id,u16 addr,u16 * dat,u16 len);

描述

主机16号功能码发送函数。

输入

- id：对方的ID。
- addr：要写的保持寄存器地址。
- dat：要写的数据数组。
- len： 要写的数据。

输出

无

返回值

无

调用例程

```
es_modbus_rtu_master_0x10(1,0,buf,5);//往ID为1的器件的0~4号保持寄存器里写入缓存buf的值。
```

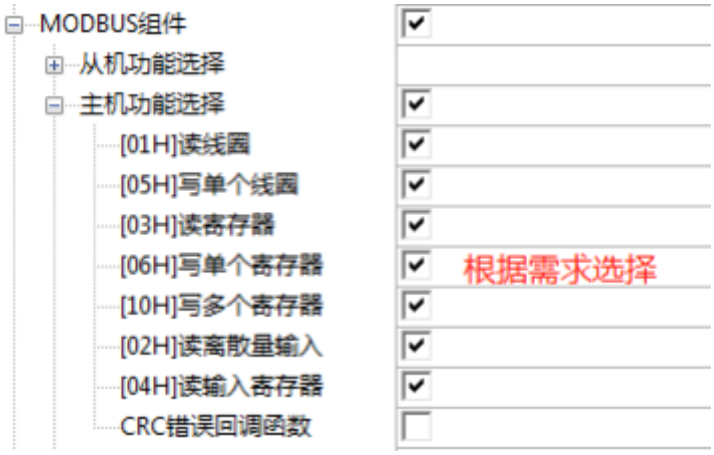
回调函数

在调用上面的主机发送函数之后，会等待从机的返回。如果从机返回了数据，将会执行回调函数。因此还需要定义一个回调函数，原型为：void es_modbus_rtu_master_0x10_callback(u16 addr,u16 len);其中addr为从机返回的地址，len为已经写入的数据的数量。

```
void es_modbus_rtu_master_0x10_callback(u16 addr,u16 len){//上面的例子里写了一个寄存器值。
    if(addr==0){//返回是地址0，说明写入成功了。
        oled_printf("写入成功了%u字节",len);//做个提示。
    }
}
```

注意事项

1. 本函数是指令【10H】的主机发送函数，使用时请确保从机支持【10H】指令。
2. 发送函数和发送回调函数是一一对应的，使用【10H】指令发送函数的时候，一定要定义对应的回调函数，否则无法知道从机返回的数据。就算不需要这个数据，回调函数也必须要定义。
3. 本函数是主机函数，所以要使用时不仅要打开主机使能，还要打开【10H】指令使能。



es_modbus_rtu_master_err_code_callback

函数原型：void es_modbus_rtu_master_err_code_callback(u8 cmd,u8 err_code);

描述

错误码回调函数，当从机返回的是错误码时调用。

输入

- cmd：出错的功能码。
- err_code：错误码。

输出

无

返回值

无

调用例程

```
void es_modbus_rtu_master_err_code_callback(u8 cmd,u8 err_code){
//可以重新和从机通信，也可以什么都不做。
}
```

注意事项

1. 参数cmd会告诉你是哪个指令发生了错误，参数err_code则会告诉你错误的类型。网上可以找到关于错误码的详细介绍。
2. **这个函数是必须要定义的！**

其他设置或功能

CRC错误回调函数

目前有从机的CRC回调函数es_modbus_rtu_crc_err_callback和主机的CRC回调函数es_modbus_rtu_master_crc_err_callback。设计的用途是在当接收数据帧的CRC对应不上时调用。不过目前市面上的上位机似乎都没有传输错误重发的设计，导致本功能从编写到现在都没有发挥过作用。将来也有可能会删除该功能，因此这里不详细展开。

通用缓存总数

这个功能涉及两个地方，从机部分用于保存【10H】指令接收的数据。主机部分用于保存发送信息。因此该值可以尽量设置在10及10以上。另外这个缓存已经做过宏操作处理，只要用到它的功能都关闭，这个缓存就不会被编译出来，这已经能有效的避免了浪费空间的问题，因此也不要把这个值设置成0。

ECP组件

简介

这是一个为了解决modbus-rtu痛点的一个自研协议，名字就是ECBM-Communication-Protocol的缩写，即ECBM通信协议。

- 痛点1： modbus-rtu是一主多从的结构。一旦总线中有两个主机，那么从机的回传将无法正确的传送到指定主机上。为此ECP在协议中包含发送者ID，保证了主机A的发出的指令，其回传也只会传到主机A。
- 痛点2： modbus-rtu以ID号作为帧头，因为比较简单而存在误触发的可能。要知道目前modbus从机都是以状态机实现的，如果误触导致状态机离开判断帧头状态的话，后面的数据也一并会判断失误。为此ECP的帧头扩展到3字节，把误触率从256分之一降低到16777216分之一。
- 痛点3： modbus-rtu的寄存器是16位，而51大多都是8位的寄存器。为此ECP一个地址对应的是一个8位寄存器，更贴合51单片机。
- 痛点4： modbus-rtu是工控常用协议，但发展至今又是线圈又是保持寄存器又是输入寄存器的概念太多。ECP做了优化，只有地址-数据这个概念，且一个地址对应一个数据，方便记忆。而且用串口助手还能看到寄存器分布表（这个功能需要自己写代码实现）不需要再去背地址了。
- 痛点5： modbus-rtu没有修改位指令，虽然可以把寄存器的位数据映射到线圈，但如果对方没做这个映射，就还是得“读-改-写”。而ECP可以直接发送置位和复位指令过去，跳过“读”和“写”直接“改”。

协议格式

ECP是基于串口的协议，所以硬件波形（指的是波特率、奇偶校验、停止位那种）和串口的设置一致，只要串口能正常通信就可以使用ECP。因此下面只解说软件层面的格式：

含义	取值范围
帧头	0xEC
长度	0x00~0xFF
长度反码	0xFF~0x00
接收方ID	0x00~0xFF
发送方ID	0x00~0xFF
操作码	0x00~0x09
地址31~24	0x00
地址23~16	0x00
地址15~8	0x00~0xFF
地址7~0	0x00~0xFF
数据/参数（不定长）	0x00~0xFF
…	…
CRC	0x00~0xFF
CRC	0x00~0xFF
帧尾	0xBE

格式解说

- 介于串口是没有一个明显的开始帧的，因此如何从一堆数据里分离出帧头就是最大问题，本协议采用的方法是除了一个固定值0xEC之外，增加长度和长度的反码来联合作为帧头。需要同时判断3个字节都符合条件才认为是一个合法帧头，这样一下就把误触发的概率降低了。
- 接收方和发送方ID的取值范围为1~254。其中0被用于是广播作用，255被用于标识为电脑。
- 操作（回复）码一共有6个，分3种操作：触发，读取，写入。每种操作都有一个回复码。其中，触发代表着该帧不会读取或写入数据，仅仅用于触发某些动作和操作；读取代表着该帧需要从协议地址开始读取指定长度的数据；写入代表着该帧需要从协议地址开始写入指定长度的数据。
- 地址在协议中是保留了32位的格式，但是这是为了兼容32位机而保留的，51这样的8位机只能使用低16位，高16位恒定为0x0000。
- 数据/参数是一个复用的意思，当操作码是读取的时候，这里就得填读取数据的个数，这就是当参数用。当操作码是写入的时候，这里填的就是要写入的数据，这里就是当数据用。当参数用的时候，只会用到1个字节。当数据用的时候，长度不固定，可以由数据帧的总长度减去固定的帧头帧尾长度得到数据的长度。
- CRC是modbus同款，直接复制库里面的函数用就行了。
- 帧尾，固定为0xBE。

API

为了压缩体积，部分能重复调用的代码会被封装成函数，但这种函数不需要也不应该让用户去调用，所以它们不会在这里被列出来。下面列举的是用户能调用的，会用得到的函数。

es_ecp_get_id

函数原型：u8 es_ecp_get_id(u8 ch);

描述

获取本机的ID函数。

输入

- ch：调用本函数的通道号。

输出

无

返回值

- 本机的ID值。

调用例程

所有通道都是同一个地址：

```
u8 es_ecp_get_id(u8 ch){
    ch=ch;//防止报错的一种写法。
    rerun 1;//返回ID为1。
}
```

不同通道不同地址：

```
u8 es_ecp_get_id(u8 ch){
    switch(ch){
        case 1:return 1;break;//通道1使用ID1。
        case 2:return 8;break;//通道2使用ID8。
    }
    return 0;//防止报错的一种写法，带返回值的函数，至少需要有一个返回值。上面的return都是有条件的，keil会认为这个参数除了1和2以外的其他值没有return而报错。
}
```

注意事项

1. **本函数需要用户去定义。**在协议解析或者发送过程中，框架会自动调用该函数。
2. **该ID请避开广播地址！**

es_ecp_trig

函数原型：u8 es_ecp_trig(u8 ch,u16 addr);

描述

ECP的触发函数。

输入

- ch：调用本函数的通道号。
- addr：被触发的地址。

输出

无

返回值

- 0：正常执行。
- 1：地址错误。

调用例程

```
u8 es_ecp_trig(u16 addr){
    if(addr==0){//触发地址为0的时候，
        LED_OFF;//关闭LED。
    }else if(addr==1){//触发地址为1的时候，
        LED_ON;//打开LED。
    }else{//其他地址未使用，
        return 1;//都返回地址错误。
    }//如果没有进入上面的return 1，那么说明执行的都是正确地址。
    return 0;//所以返回正常执行。
}
```

注意事项

1. **本函数只需要用户去定义，不需要用户调用。**当对方向本机发送触发操作的时候，框架会自动调用该函数。
2. 请按照返回值示例所示，将执行结果返回，以便ECP协议回复对方。

es_ecp_write

函数原型：u8 es_ecp_write(u8 ch,u16 addr,u8 * buf,u8 len);

描述

ECP的写入函数。

输入

- ch：调用本函数的通道号。
- addr：被写入的首地址。
- buf：准备写入的数据。
- len：准备写入数据的长度。

输出

无

返回值

- 0：正常执行。
- 1：地址错误。（比如该地址没有映射到任何寄存器上。）
- 2：数据错误。（比如该地址对应的寄存器是需要某个区间的值，而参数给的数据不在这个区间。）
- 3：长度错误。（比如该地址对应了一个16位寄存器，而且还需要同时写入两个字节的数据，但参数只给了一个字节。）

调用例程

```
u8 es_ecp_write(u16 addr,u8 * buf,u8 len){
    u8 i;
    for(i=0;i<len;i++){//以长度来做一个循环。
        es_ecp_test[addr+i]=buf[i];//将缓存存入指定数组中，或者其他介质中。
    }
    return 0;//正常执行就返回0。
}
```

注意事项

1. **本函数只需要用户去定义，不需要用户调用。**当对方向本机发送写入操作的时候，框架会自动调用该函数。
2. 请按照返回值示例所示，将执行结果返回，以便ECP协议回复对方。

es_ecp_read

函数原型：u8 es_ecp_read(u8 ch,u16 addr,u8 * buf,u8 len);

描述

ECP的读取函数。

输入

- ch：调用本函数的通道号。
- addr：被读取的首地址。
- buf：保存读取数据的缓存。
- len：读取数据的长度。

输出

无

返回值

- 0：正常执行。
- 1：地址错误。（比如该地址没有映射到任何寄存器上。）
- 2：数据错误。（比如该地址对应的寄存器是需要某个区间的值，而参数给的数据不在这个区间。）
- 3：长度错误。（比如该地址对应了一个16位寄存器，而且还需要同时写入两个字节的数据，但参数只给了一个字节。）

调用例程

```
u8 es_ecp_read(u16 addr,u8 * buf,u8 len){
    u8 i;
    for(i=0;i<len;i++){//以长度来做一个循环。
        buf[i]=es_ecp_test[i];//将数据从原来的介质里存入缓存中。
    }
    return 0;//正常执行就返回0。
}
```

注意事项

1. **本函数只需要用户去定义，不需要用户调用。**当对方向本机发送读取操作的时候，框架会自动调用该函数。
2. 请按照返回值示例所示，将执行结果返回，以便ECP协议回复对方。

es_ecp_read_callback

函数原型：void es_ecp_read_callback(u8 ch,u16 addr,u8 * buf,u8 len);

描述

ECP的读取回调函数。

输入

- ch：调用本函数的通道号。
- addr：被读取的首地址。
- buf：保存读取数据的缓存。
- len：读取数据的长度。

输出

无

返回值

无

调用例程

```
void es_ecp_read_callback(u16 addr,u8 * buf,u8 len){
    if(addr==0x0000){//当返回的数据是地址0时，
        oled_show(0,0,"%s",buf);//将读到的数据以字符串形式显示到屏幕上。
    }
}
```

注意事项

1. **本函数只需要用户去定义，不需要用户调用。**当本机向对方发送读取操作，且对方回复的时候，框架会自动调用该函数。

es_ecp_master_write

函数原型：void es_ecp_master_write(u8 ch,u8 id,u16 addr,u8 * buf,u8 len);

描述

ECP主机写函数。

输入

- ch：要发送ECP协议的通道号。
- id：接收方的ID。
- addr：要写入的首地址。
- buf：准备写入的数据。
- len：准备写入数据的长度。

输出

无

返回值

无

调用例程

```
u8 buf[]="Hello PC!";//定义了字符串。
...//其他代码。
es_ecp_master_write(1,9,0,buf,10);//向串口1发送一条写入操作的数据，其中目标器件的ID为9，写入首地址为0，要发送的数据是buf的内容，字符串以/0结尾，所以长度是10。
```

注意事项

1. 本函数用于向一个器件写入一串数据。虽然举例用了字符串，但是不局限与字符串，任何数据都可以。甚至单个字节都行。

es_ecp_master_read

函数原型：void es_ecp_master_read(u8 ch,u8 id,u16 addr,u8 len);

描述

ECP主机写函数。

输入

- ch：要发送ECP协议的通道号。
- id：接收方的ID。
- addr：要读取的首地址。
- len：要读取的数据长度。

输出

无

返回值

无

调用例程

```
es_ecp_master_read(1,9,0,10);//向串口1发送一条读取操作的数据，其中目标器件的ID为9，读取的首地址为0，要读取10个字节的数据。
```

回调函数

在调用上面的主机发送函数之后，会等待从机的返回。如果从机返回了数据，将会执行回调函数。因此还需要定义一个回调函数，原型为：void es_ecp_master_read_callback(u8 ch,u16 addr,u8 * buf,u8 len);其中ch为返回的通道号，addr为从机返回的地址，buf为对应的数据缓存，len为缓存大小。

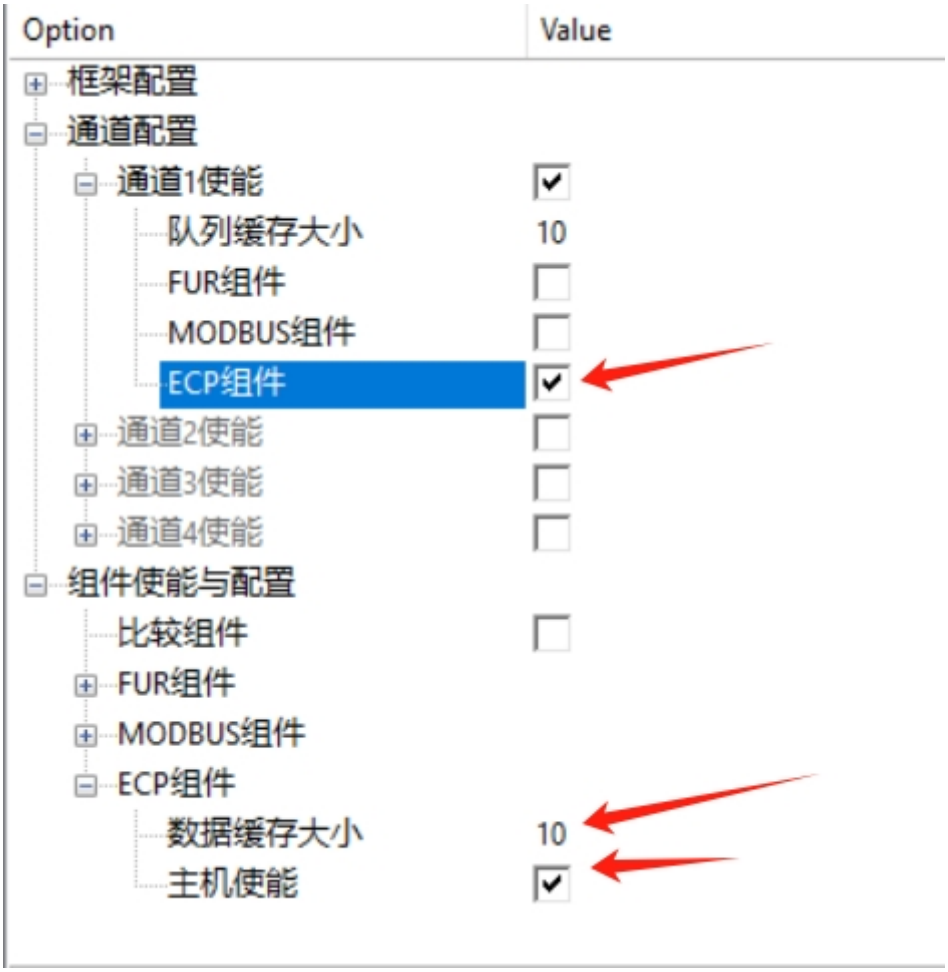
```
void es_ecp_master_read_callback(u8 ch,u16 addr,u8 * buf,u8 len){
    if(addr==0x0000){//当返回的数据是地址0时，
        oled_show(0,0,"%s",buf);//将读到的数据以字符串形式显示到屏幕上。
    }
}
```

注意事项

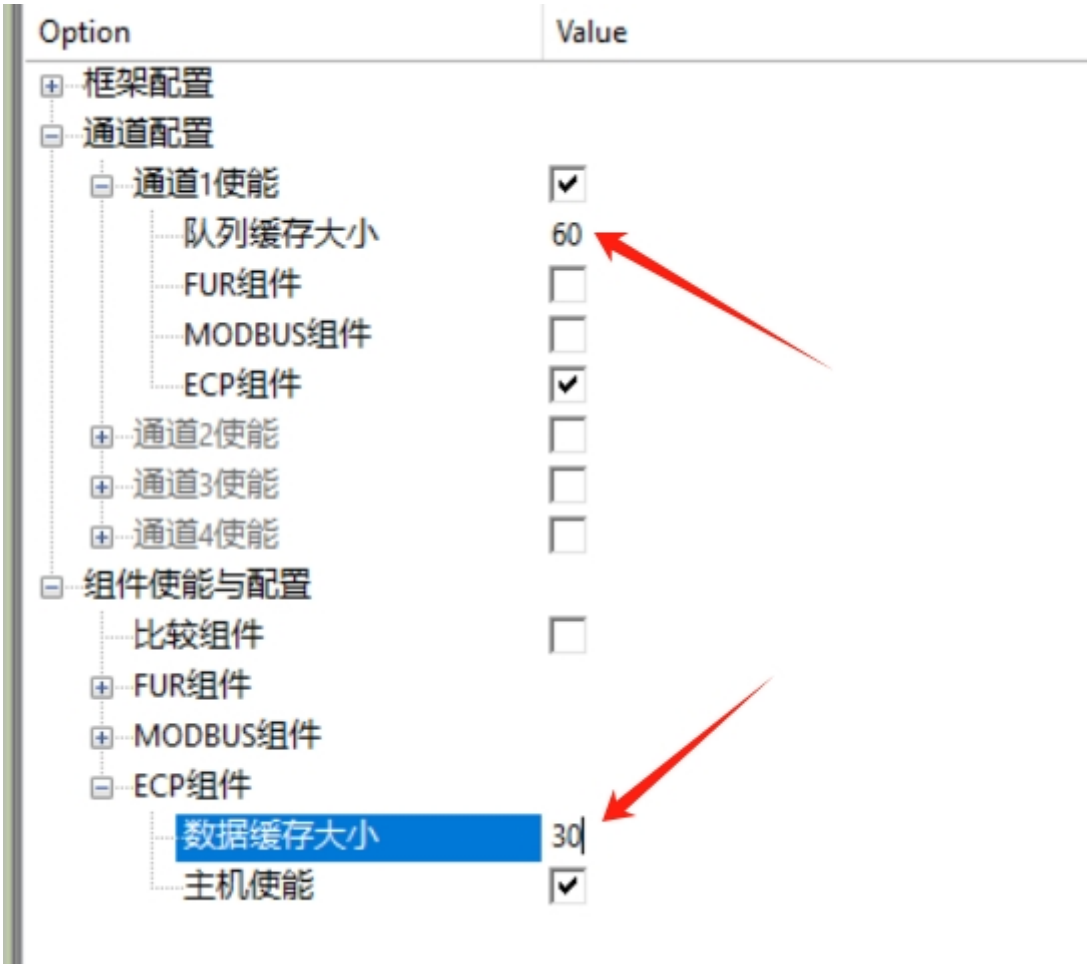
1. 本函数用于向一个器件读取一串数据。基于ECP的单工特性，读取的数据不会在本次通信中读出。
2. 在调用本函数之后，如果对方收到并回复了，那么读取到的数据是通过es_ecp_read_callback函数存入本机的，因此需要在es_ecp_read_callback处定义好收到数据后的处理代码。

应用步骤

1. 按照stream的应用步骤搭好stream框架。
2. 假如需要通道1即串口1使用ECP协议，需要在“通道1使能”里使能“ECP组件”。其他通道也是同样的操作方法。（本设置在stream.h中）



3. 如果要更改ECP组件的设置，可以在“组件使能与配置”里找到ECP组件的设置，如上图所示。其中“数据缓存大小”应当小于上面的“队列缓存大小”，不然就会浪费宝贵的xdata空间。如果确实需要更大的数据缓存，那么也必须先增大“队列缓存大小”再加大“数据缓存大小”。（本设置在stream.h中）



4. 如果不需要ECP主机，可以取消“主机使能”以释放flash空间。（本设置在stream.h中）
5. 按照API中的示例，定义es_ecp_get_id函数。
6. 按照API中的示例，定义es_ecp_trig函数。
7. 按照API中的示例，定义es_ecp_write函数。
8. 按照API中的示例，定义es_ecp_read函数。
9. 如果使能了主机，那么还需要按照API中的示例，定义es_ecp_master_read_callback函数。

至此，一个完整的能运行得ECP协议就搭建完成。main.c的参考代码如下：

```
#include "ecbm_core.h"  //加载库函数的头文件。
void main(void){        //main函数，必须的。
    system_init();      //系统初始化函数，也是必须的。
    ecbm_stream_init(); //stream框架初始化。
    while(1){
        ecbm_stream_main();//stream框架主函数。
    }
}
u8 es_ecp_get_id(u8 ch){
    ch=ch;//示例中防报错的写法而已，实际项目没必要100%照搬。
    return 1;
}
u8 es_ecp_trig(u8 ch,u16 addr){
    ch=ch;//示例中防报错的写法而已，实际项目没必要100%照搬。
    addr=addr;
    return 0;
}
u8 es_ecp_write(u8 ch,u16 addr,u8 * buf,u8 len){
    ch=ch;//示例中防报错的写法而已，实际项目没必要100%照搬。
    addr=addr;
    buf=buf;
    len=len;
    return 0;
}
u8 es_ecp_read(u8 ch,u16 addr,u8 * buf,u8 len){
    ch=ch;//示例中防报错的写法而已，实际项目没必要100%照搬。
    addr=addr;
    buf=buf;
    len=len;
    return 0;
}
void es_ecp_master_read_callback(u8 ch,u16 addr,u8 * buf,u8 len){
    ch=ch;//示例中防报错的写法而已，实际项目没必要100%照搬。
    addr=addr;
    buf=buf;
    len=len;
}
```