

ECBM库使用手册

欢迎阅读ECBM库的使用手册，本手册将讲述每个库的用途，API和用法，并建立了目录可供快速查阅。

版本说明

本文档基于ECBM库V3.2.2版本，太低的版本可能一些设置和操作的方法与本文描述的不一致，可以先使用V3.2.2版本来照着本文学习。

编程工具

在使用ECBM库之前，请确保以下软件、程序都下载安装完毕。如果已经安装过旧的版本，依然建议你通过链接获取最新版本。

- [编译器、Keil for C51](#)
- [下载、烧录、多功能工具STC-ISP](#)
- [ECBM源码](#)

或者加1群778916610，在群文件中就可以下载最新版本。如果1群加满了可以进入2群:927297508。

代码管理工具

作为一名工程师，我强烈推荐大家安装Git和Tortoise Git。Git是一款非常好用的代码管理工具，不管你是做完毕设就转行，还是立志做一名工程师，Git都可以为你当前的编程保驾护航。Git的主要用处包括但不限于：

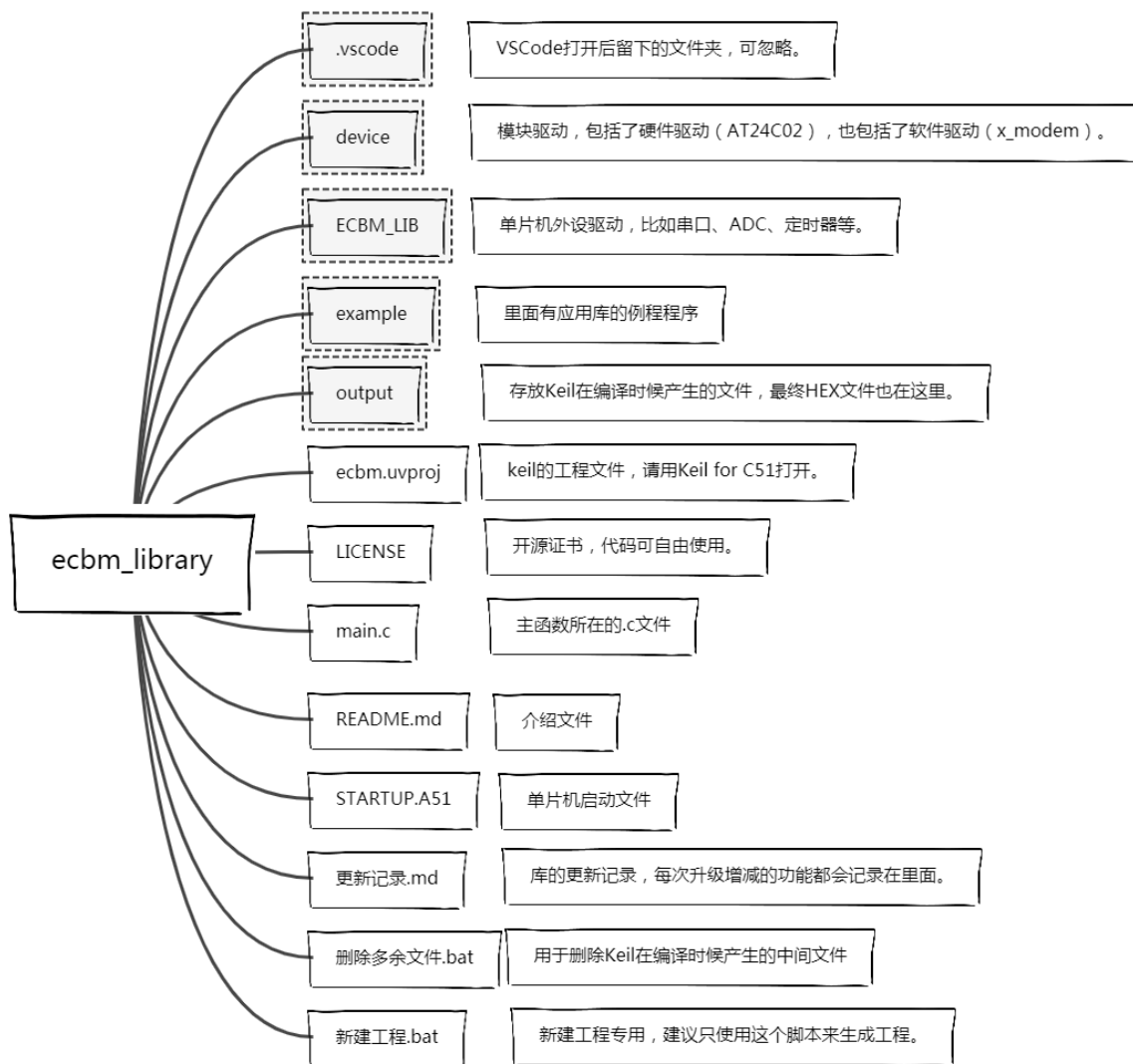
1. **保存历史代码。**程序员笑话里有一则就是“修修改改，结果甲方还是要第一版”，代码基本也不是一版就能定型。在不断的修改中，很容易就不记得第一版是怎么实现的了。但是通过Git可以把初版保存下来，需要的时候直接恢复成初版。
2. **对比代码修改。**Git可以把当前的代码和历史代码进行对比，你可以清楚的看到本次编程修改了哪些代码。
3. **代码更新。**在使用远程代码库（比如ECBM库）的时候，可以通过拉取功能将本地的代码更新和网络上一致的版本。

而Tortoise Git是Git的一个可视化外壳。Git本身只提供了一个指令行界面和一个很难用的英文UI界面，Tortoise Git可以为Git的每个功能提供快速到达的右键选项，并且在安装中文语音包之后，这些选项和设置都是中文界面，非常好用。

Git的安装和使用教程可以参考廖雪峰出的[Git教程](#)，主要学习学习如果建立仓库、提交代码、回溯代码就行了。

文件结构

下载好源码之后，解压、打开之后会看到如下的文件结构：



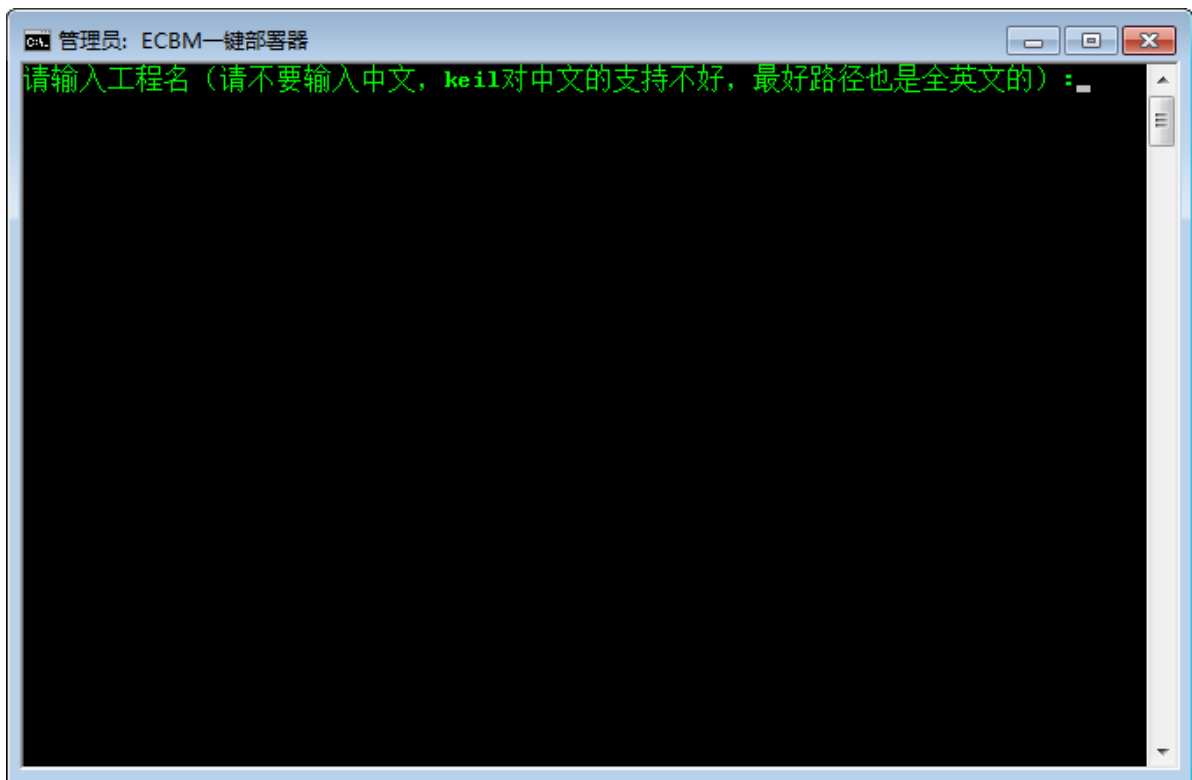
这里面存放的就是库函数的源文件，建议除了修复BUG以外不要对其进行改动和编辑。

如何开始愉快的编程？

之所以提到了**不要随意改动源文件**，主要是为了保留原始的库版本。因为库本身没有编译成lib格式，而是以文本文件存在，那么库文件就会有被修改的风险。假如某次修改库的时候改了不该改的地方导致整个库无法使用了，此时还能有一份备份可供还原。

所以我做了一个脚本，这个脚本会复制必要的文件到新工程的文件夹中，这样在新的工程文件夹中修改代码就不会影响到源文件了。**因此强烈推荐通过新建工程.bat来建立工程。**

双击新建工程.bat就会看到如下界面，在此界面中输入英文的名字或者中文的拼音，因为Keil是国外的软件对中文的支持不是太好，所以路径中最好不要有中文字符。



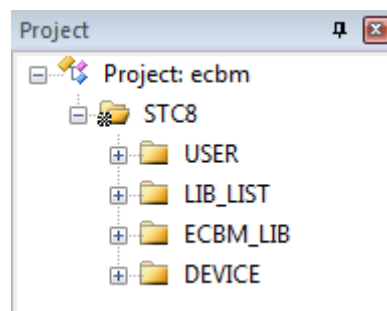
假设输入的是“test”，那么按下回车键之后，脚本会自动建立ecbm_test文件夹，并把库文件全复制到ecbm_test文件夹中。没被复制的都是些不重要的文件，比如开源证书。而output文件夹会在Keil编译的时候自动建立。

进入ecbm_test文件夹，打开ecbm.uvproj，开始愉快的编程吧！

工程简介和基础设置

工程结构

用Keil打开工程文件之后，在左边的工程区可以看到这样的结构：



USER用于存放main.c、STARTUP.A51和其他用户建立的.c文件。

LIB_LIST放着ECBM库的.h文件，因为.h文件里放着配置选项，所以单独拿出来方便快速进入到各个.h文件中。

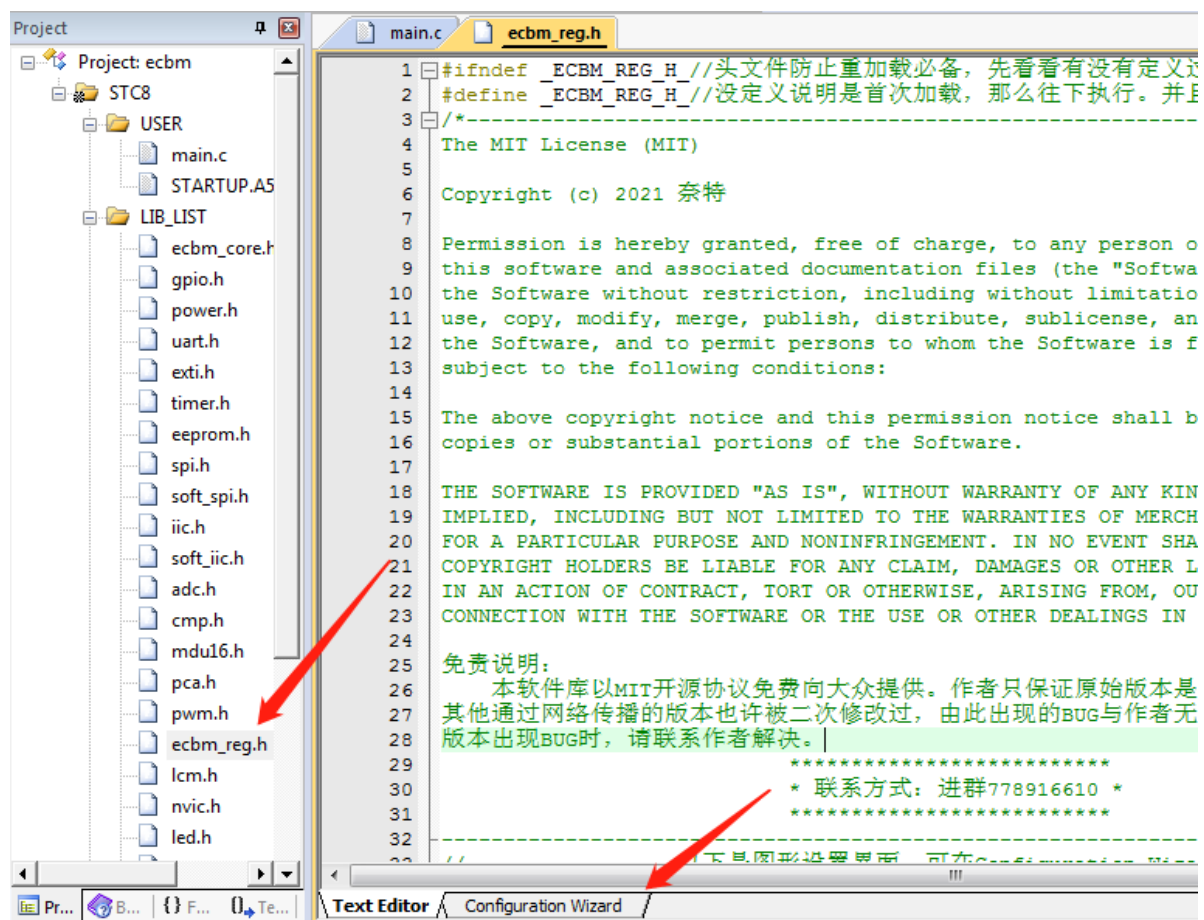
ECBM_LIB放着ECBM库的.c文件，要参与编译就必须把.c放进来，这是正常操作。

DEVICE放着模块的驱动，默认是空的。但就像上面所说的，要想让模块驱动参与编译就必须把模块驱动的.c文件放进来。

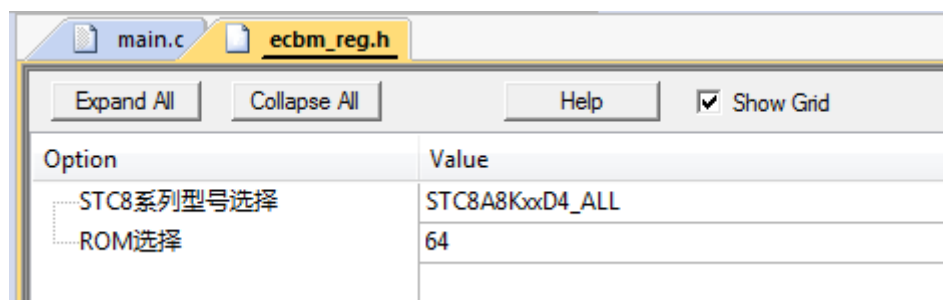
单片机型号设置

STC8系列目前有F、A、G、H、C这几个系列，虽然它们的8051寄存器都是一致的，但是扩展8051的功能寄存器却是有所小小的差别。举个例子：串口1属于标准的8051外设，所以他们的寄存器都是一样的；PWM属于标准8051没有的外设，所以STC8G和STC8H的PWM寄存器有所差别。**因此单片机型号一定要设置正确，才能访问正确的寄存器。**

操作方法为：在Keil左侧的LIB_LIST下找到ecbm_reg.h，双击打开ecbm_reg.h。然后在窗口的左下角点Configuration Wizard标签进入图形化配置界面。



所谓图形化，就和电脑系统的发展一样。一开始的dos系统只是纯指令操作的，后来出现了字符画，再后来出了Windows。UI的产生降低了使用难度，把很多复杂的功能简单得呈现出来。一个函数库要想兼容多个型号，就必然要有很多宏定义来设置参数，在没有图形化的时候，一排一排宏定义看得人头晕。现在好了，ECBM按照Keil的规则写了图形化配置界面，方便了大家配置。



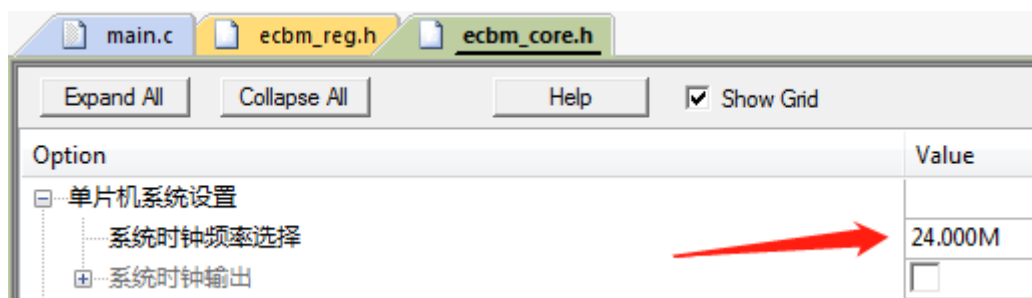
选项中的xx代表ROM容量，ROM容量在下面选择。比如上图中就表示设置型号为STC8A8K64D4。ALL代表该型号所有引脚数都能使用这个参数。与之相对应的就是STC8G1Kxx_16PIN_20PIN和STC8G1Kxx_8PIN，他们虽然型号一样，但是一个是16脚、20脚封装，一个是8脚封装。引脚数不一样导致了引脚的复用功能不一样，也就导致了寄存器的内容不一样。因此一定要选择正确的型号。

时钟参数设置

库里面有很多地方涉及到时钟，比如延时函数、波特率计算等。时钟参数一定要设置和实际使用的时钟一致。

打开ecbm_core.h，进入图形化配置界面。在【单片机系统设置】下的【系统时钟频率选择】选择单片机运行的时钟频率。一般这个频率就是你在STC-ISP烧录程序时设置的频率或者是外置晶振的频率。

实例：我使用STC8A8K64D4，在STC-ISP中设置了24MHz的频率。如图所示。



GPIO库

GPIO库是基础中的基础，所以默认就是一定要开启并参与编译的。

API

gpio_uppull

函数原型：void gpio_uppull(u8 pin,u8 en);

描述

IO口上拉电阻配置函数，可设定打开或关闭某个IO口的内置上拉电阻。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。
- en：使能开关，1代表打开上拉电阻；0代表关闭上拉电阻。

输出

无

返回值

无

调用例程

直接参数调用：

```
1 gpio_uppull(D11,1); //打开P1.1脚的上拉电阻。
```

变量做参数调用：

```
1 u8 pin_set[5]={D10,D21,D22,D41,D40}; //定义一个数组，存放5个IO口的编号。
2 u8 i; //临时变量。
3 for(i=0;i<5;i++){ //打开这5个IO口的上拉电阻。
4     gpio_uppull(pin_set[i],1);
5 }
```

注意事项

1. 本函数调用的寄存器是在扩展寄存器区，所以要保证扩展寄存器的访问使能是打开的。目前system_init()函数内部会自动打开这个使能，但是如果你为了优化而没有调用system_init()函数的话，需要加上“EX_SFR_ENABLE；”。
2. 本函数不支持多个IO同时开启上拉电阻，需要用for循环一个一个设置。

gpio_uppull_ex

函数原型：void gpio_uppull_ex(u8 port,u8 pin,u8 en);

描述

IO口上拉电阻配置函数扩展版，可批量设定打开或关闭几个IO口的内置上拉电阻。

输入

- port：P口的编号，比如P0口就是GPIO_P0。
- pin：IO的编号，GPIO_PIN_0~GPIO_PIN_7对应了Px.0~Px.7。
- en：使能开关，1代表打开上拉电阻；0代表关闭上拉电阻。

输出

无

返回值

无

调用例程

```
1  gpio_uppull_ex(GPIO_P0,GPIO_PIN_0|GPIO_PIN_1,1); //P0.0和P0.1口打开内部的上拉电阻。
```

注意事项

1. 本函数调用的寄存器是在扩展寄存器区，所以要保证扩展寄存器的访问使能是打开的。目前system_init()函数内部会自动打开这个使能，但是如果你为了优化而没有调用system_init()函数的话，需要加上“EX_SFR_ENABLE；”。
2. 本函数和gpio_uppull函数相比，支持同一个P口的多个IO同时开启上拉电阻。

gpio_mode

函数原型：void gpio_mode(u8 pin,u8 mode);

描述

IO口工作模式设置函数，可设定某个IO口的工作模式。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。
- mode：工作的模式，有以下4个可供选择。
 1. GPIO_PU / GPIO_IN：弱上拉模式，此模式下IO的上拉能力弱，下拉能力强。常用在输入输出双向口。

2. GPIO_HZ：高阻模式，此模式下IO不能对外输出高低电平，但能读取IO电平。一般用在输入口。
3. GPIO_OD：开漏模式，此模式下IO不能输出高电平，需要有额外的上拉电阻才能输出高电平。一般用在总线上。
4. GPIO_PP / GPIO_OUT：推挽模式，此模式下的上拉下拉能力都强，常用在输出口。

输出

无

返回值

无

调用例程

直接参数调用：

```
1 | gpio_mode(D11,GPIO_OUT); //把P1.1脚设置为推挽（输出）模式。
```

变量做参数调用：

```
1 | u8 pin_set[5]={D10,D21,D22,D41,D40}; //定义一个数组，存放5个IO口的编号。
2 | u8 i; //临时变量。
3 | for(i=0;i<5;i++){ //把这5个IO口设置为输入（弱上拉）模式。
4 |     gpio_mode(pin_set[i],GPIO_IN);
5 | }
```

联动调用：

```
1 | u8 pin_set[5]={D10,D21}; //定义一个数组，存放2个IO口的编号。假设把它们当做IIC口。
2 | u8 i; //临时变量。
3 | for(i=0;i<2;i++){
4 |     gpio_mode(pin_set[i],GPIO_IN); //把这2个IO口设置为开漏模式。
5 |     gpio_uppull(pin_set[i],1); //IIC总线的设定就是开漏+上拉。
6 | }
```

注意事项

1. 本函数不支持多个IO同时设置工作模式，需要用for循环一个一个设置。

gpio_mode_ex

函数原型：void gpio_mode_ex(u8 port,u8 pin,u8 mode);

描述

设置IO口工作模式函数扩展版，可同时设置同一P口的多个IO口。

输入

- port：P口的编号，比如P0口就是GPIO_P0。
- pin：IO的编号，GPIO_PIN_0~GPIO_PIN_7对应了Px.0~Px.7。
- mode：工作的模式，有以下4个可供选择。
 1. GPIO_PU / GPIO_IN：弱上拉模式，此模式下IO的上拉能力弱，下拉能力强。常用在输入输出双向口。
 2. GPIO_HZ：高阻模式，此模式下IO不能对外输出高低电平，但能读取IO电平。一般用在输入口。
 3. GPIO_OD：开漏模式，此模式下IO不能输出高电平，需要有额外的上拉电阻才能输出高电平。一般用在总线上。
 4. GPIO_PP / GPIO_OUT：推挽模式，此模式下的上拉下拉能力都强，常用在输出口。

输出

无

返回值

无

调用例程

```
1 | gpio_mode_ex(GPIO_P0,GPIO_PIN_0|GPIO_PIN_1,GPIO_HZ); //P0.0和P0.1口设置为高阻态模式。
```

注意事项

1. 本函数和gpio_mode函数相比，支持多个IO同时设置工作模式。

gpio_speed

函数原型：void gpio_speed(u8 pin,u8 speed);

描述

IO口速度设置函数，可设定某个IO口的工作速度。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。
- speed：IO速度，有以下2个可供选择。
 1. GPIO_FAST：快速模式。
 2. GPIO_SLOW：慢速模式。

输出

无

返回值

无

调用例程

直接参数调用：

```
1  gpio_speed(D11,GPIO_FAST); //把P1.1脚设置为快速模式。
```

变量做参数调用：

```
1  u8 pin_set[5]={D10,D21,D22,D41,D40}; //定义一个数组，存放5个IO口的编号。
2  u8 i; //临时变量。
3  for(i=0;i<5;i++){ //把这5个IO口设置为快速模式。
4      gpio_speed(pin_set[i],GPIO_FAST);
5  }
```

注意事项

1. 本函数不支持多个IO同时设置工作模式，需要用for循环一个一个设置。
2. 其实我实测两个模式似乎没啥区别。
3. 本函数调用的寄存器是在扩展寄存器区，所以要保证扩展寄存器的访问使能是打开的。目前system_init()函数内部会自动打开这个使能，但是如果你为了优化而没有调用system_init()函数的话，需要加上“EX_SFR_ENABLE；”。

gpio_current

函数原型：void gpio_current(u8 pin,u8 current);

描述

IO口驱动电流设置函数，可设定某个IO口的驱动电流。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。
- current：IO驱动电流，有以下2个可供选择。
 1. GPIO_STR：增强驱动模式。
 2. GPIO_GEN：正常驱动模式。

输出

无

返回值

无

调用例程

直接参数调用：

```
1 gpio_current(D11,GPIO_STR); //把P1.1脚设置为大电流模式。
```

变量做参数调用：

```
1 u8 pin_set[5]={D10,D21,D22,D41,D40}; //定义一个数组，存放5个IO口的编号。
2 u8 i; //临时变量。
3 for(i=0;i<5;i++){ //把这5个IO口设置为大电流模式。
4     gpio_current(pin_set[i],GPIO_STR);
5 }
```

注意事项

1. 本函数不支持多个IO同时设置工作模式，需要用for循环一个一个设置。
2. 即使是大大电流模式也不应该超过20mA，以防IO烧毁。
3. 本函数调用的寄存器是在扩展寄存器区，所以要保证扩展寄存器的访问使能是打开的。目前system_init()函数内部会自动打开这个使能，但是如果你为了优化而没有调用system_init()函数的话，需要加上“EX_SFR_ENABLE；”。

gpio_write

函数原型：void gpio_write(u8 port,u8 dat);

描述

P口写入函数，直接写入8位数据到某个P口上。

输入

- port：P口编号，宏定义GPIO_P0~GPIO_P7，比如P1口就是GPIO_P1。
- dat：要输出的数据。

输出

无

返回值

无

调用例程

直接参数调用：

```
1 gpio_write(GPIO_P1,0x05); //P1口输出0x05。
```

变量做参数调用：

```

1  u8 led_type[8]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};//定义一个数组，存放8个
   数据。
2  u8 i;//临时变量。
3  for(i=0;i<8;i++){
4      gpio_write(GPIO_P1,led_type[i]);//研究数组数据即可知这段的效果就是从P1.0到P1.7
   依次点亮的流水灯效果。
5      delay_ms(500);//流水灯不用太快，延时500ms。
6  }

```

注意事项

1. 本函数在“gpio_write(GPIO_P1,0);”的时候和“P1=0;”是等价的，甚至“P1=0;”的运行速度会快非常多。但是本函数的优点不是运行速度，而是可以用变量作为参数，比如“gpio_write(SBUF,0);”这样写就能用串口实时控制某个P口全置0。

gpio_read

函数原型：u8 gpio_read(u8 port);

描述

P口读出函数，读取某个P口的数据（8位同时读，也就是并口通信）。

输入

- port：P口编号，宏定义GPIO_P0~GPIO_P7，比如P1口就是GPIO_P1。

输出

无

返回值

- 读到的P口数据。

调用例程

直接参数调用：

```

1  u8 val;//定义一个变量。
2  val=gpio_read(GPIO_P1);//P1口的值存入变量val中。

```

变量做参数调用：

```

1  u8 val[3];//定义一个数组，存放3个数据。
2  u8 i;//临时变量。
3  for(i=0;i<3;i++){
4      val[i]=gpio_read(GPIO_P1+i);//GPIO_P1的宏定义值就是1，所以GPIO_P1+1等价于
   GPIO_P2。这个循环的效果就是读取P1~P3的值存到数组中。
5  }

```

注意事项

1. 本函数在“val=gpio_read(GPIO_P1);”的时候和“val=P1;”是等价的，同样“val=P1;”的运行速度会快非常多。但是本函数的优点不是运行速度，而是可以用变量作为参数，比如“SBUF=gpio_read(SBUF);”这样写就能用串口实时读取某个P口的值。

gpio_out

函数原型：void gpio_out(u8 pin,u8 value);

描述

IO口输出函数，用于输出高低电平。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。
- value：0代表低电平；非0代表高电平。

输出

无

返回值

无

调用例程

直接参数调用：

```
1 | gpio_out(D15,1); //P1.5脚输出高电平。
```

变量做参数调用：

```
1 | u8 pin_list[3]={D15,D24,D32}; //定义一个数组，存放3个引脚编号。
2 | u8 i; //临时变量。
3 | for(i=0;i<3;i++){
4 |     gpio_out(pin_list[i],1); //依次把这3个IO置高。
5 | }
```

注意事项

1. 和上面说的一样，本函数的优点在于控制的IO口是可变的，这一点为后面的库的引脚变动提供了很大的方便。如果不是为了这个便利性的话，还不如直接用“P15=1;”这样的语句来的快。

gpio_in

函数原型：u8 gpio_in(u8 pin);

描述

IO口输入函数，用于读取某个IO的电平状态。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。

输出

无

返回值

- 该IO的电平状态。0代表低电平；1代表高电平。

调用例程

直接参数调用：

```
1 u8 val;//定义一个变量。
2 val=gpio_in(D15);//把P1.5脚的电平状态赋给val。
```

变量做参数调用：

```
1 u8 pin_list[3]={D15,D24,D32};//定义一个数组，存放3个引脚编号。
2 u8 value[3];//定义一个数组存放电平状态。
3 u8 i;//临时变量。
4 for(i=0;i<3;i++){
5     value[i]=gpio_in(pin_list[i]);//依次把这3个IO的电平状态装进数组。
6 }
```

注意事项

1. 和上面说的一样，本函数的优点在于控制的IO口是可变的，这一点为后面的库的引脚变动提供了很大的方便。如果不是为了这个便利性的话，还不如直接用“val=P15;”这样的语句来的快。

gpio_toggle

函数原型：void gpio_toggle(u8 pin);

描述

IO口翻转函数，用于翻转某个IO的电平状态。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。

输出

无

返回值

无

调用例程

直接参数调用：

```
1 gpio_toggle(D15); //把P1.5脚的电平状态翻转。
```

变量做参数调用：

```
1 u8 pin_list[3]={D15,D24,D32}; //定义一个数组，存放3个引脚编号。
2 u8 i; //临时变量。
3 for(i=0;i<3;i++){
4     gpio_toggle(pin_list[i]); //依次把这3个IO的电平状态翻转。
5 }
```

注意事项

1. 和上面说的一样，本函数的优点在于控制的IO口是可变的，这一点为后面的库的引脚变动提供了很大的方便。如果不是为了这个便利性的话，还不如直接用“P15=!P15;”这样的语句来的快。

gpio_toggle_fast

函数原型：void gpio_toggle_fast(u8 port,u8 pin);

描述

IO口快速翻转函数，gpio_toggle函数的加速版。

输入

- port：P口的编号，比如要控制P1.0，这里填写GPIO_P1
- pin：P口IO的编号，比如要控制P1.0，这里填写GPIO_PIN_0。

输出

无

返回值

无

调用例程

单个IO翻转：

```
1 gpio_toggle_fast(GPIO_P1,GPIO_PIN_0); //把P1.0脚的电平状态翻转。
```

同P口多个IO翻转：

```
1 gpio_toggle_fast(GPIO_P1,GPIO_PIN_0|GPIO_PIN_5); //把P1.0脚和P1.5脚的电平状态翻转。
```

注意事项

1. 快速版函数删除了引脚解析，所以不仅加快了运行速度还支持多个IO（当然必须在同一个P口）同时翻转。缺点就是需要两个参数。

gpio_out_fast

函数原型：void gpio_out_fast(u8 port,u8 pin,u8 val);

描述

IO口快速输出函数，gpio_out函数的加速版。

输入

- port：P口的编号，比如要控制P1.0，这里填写GPIO_P1
- pin：P口IO的编号，比如要控制P1.0，这里填写GPIO_PIN_0。
- val：要输出的电平值，0代表低电平；1代表高电平。

输出

无

返回值

无

调用例程

单个IO输出高电平：

```
1 | gpio_out_fast(GPIO_P1,GPIO_PIN_0,1); //P1.0脚输出高电平。
```

同P口多个IO输出低电平：

```
1 | gpio_out_fast(GPIO_P1,GPIO_PIN_0|GPIO_PIN_5,0); //P1.0脚和P1.5脚输出低电平。
```

注意事项

1. 快速版函数删除了引脚解析，所以不仅加快了运行速度还支持多个IO（当然必须在同一个P口）同时输出一个电平值。缺点就是需要两个参数。

gpio_in_fast

函数原型：u8 gpio_in_fast(u8 port,u8 pin);

描述

IO口快速输入函数，gpio_in函数的加速版。

输入

- port：P口的编号，比如要控制P1.0，这里填写GPIO_P1
- pin：P口IO的编号，比如要控制P1.0，这里填写GPIO_PIN_0。

输出

无

返回值

该IO的电平状态。0代表低电平；1代表高电平。

调用例程

读IO的电平状态：

```
1 | val=gpio_in_fast(GPIO_P1,GPIO_PIN_0);//把P1.0脚的电平状态赋给val。
```

注意事项

1. 和上面的快速函数差不多，这个也是删除了引脚解析达到加速的目的。但是本函数只支持一个IO的电平状态读取。

优化建议

打开gpio.h的图形化配置界面，在【普通优化设置】中把没有出现的IO口的使能取消掉。

举例：目前使用的是STC8G1K08A-8PIN单片机，只有P3.0~P3.3和P5.4、P5.5。那么可以这样选择：

Option	Value
<input checked="" type="checkbox"/> 普通优化设置	
P0口使能	<input type="checkbox"/>
P1口使能	<input type="checkbox"/>
P2口使能	<input type="checkbox"/>
P3口使能	<input checked="" type="checkbox"/>
P4口使能	<input type="checkbox"/>
P5口使能	<input checked="" type="checkbox"/>
P6口使能	<input type="checkbox"/>
P7口使能	<input type="checkbox"/>

这样一来，不仅能优化空间，还能加快执行速度。而下面的【深度优化】是优化各个函数用的，没有使能的函数将不会参与编译，所以当且仅当整个工程都没有用到这些函数的时候才能优化掉，否则会出大问题。

POWER库

POWER库是有关于单片机的电源操作库。在使用本库之前先到ecbm_core.h里使能。双击打开ecbm_core.h文件，然后进入图形化配置界面，使能POWER库。

Option	Value
<input checked="" type="checkbox"/> 单片机系统设置	
<input checked="" type="checkbox"/> 深度优化和测试功能	
<input checked="" type="checkbox"/> 单片机外设库的选用	
POWER库	<input checked="" type="checkbox"/>

API

power_reset_code

函数原型：void power_reset_code(void);

描述

单片机复位函数，复位后从main函数开始运行。

输入

无

输出

无

返回值

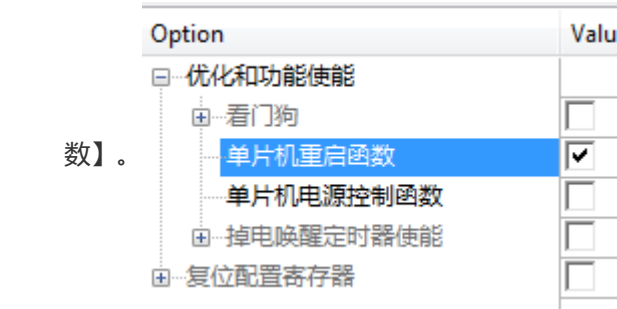
无

调用例程

```
1  if(SBUF=='*'){//串口接收到*号的时候，
2      power_reset_code();//重启单片机。
3  }
```

注意事项

- 1. 如果是想做软重启自动下载功能，那么这个函数不能满足要求，需要power_reset_isp函数。
- 2. 为了优化空间，本函数默认不使能。请在power.h文件的图形化配置界面里使能【单片机重启函数】。



power_reset_isp

函数原型：void power_reset_isp(void);

描述

单片机复位函数，复位后从单片机BootLoader开始运行。

输入

无

输出

无

返回值

无

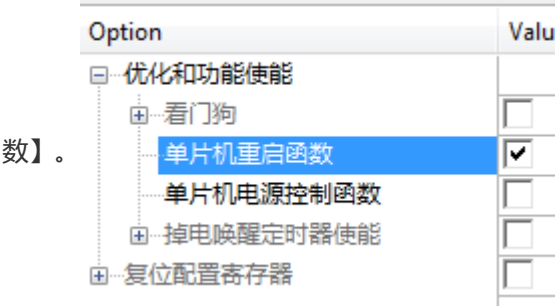
调用例程

自动下载功能的核心：

```
1  if(SBUF==0x7F){ //STC-ISP在下载前会发送一连串的0x7F。
2      count++; //每收到一次就让计数值+1。
3      if(count==200){ //收到200个0x7F的时候，认为是合法的下载流。
4          power_reset_isp(); //重启单片机，下载代码。
5      }
6  }else{ //如果收到的不是0x7F，说明这可能是正常的数据。
7      count=0; //清零计数值。
8  }
```

注意事项

- 1. 为了优化空间，本函数默认不使能。请在power.h文件的图形化配置界面里使能【单片机重启函数】。



power_powerdown

函数原型：void power_powerdown(void);

描述

掉电函数，单片机进入掉电模式，CPU和外设的电源都会被关掉。

输入

无

输出

无

返回值

无

参数配置

在掉电模式下，单片机可以由内部的一个掉电定时器唤醒。如果要使用定时唤醒功能，首先在图形化配置界面下使能【掉电唤醒定时器使能】，并且在【定时时间】选项处填写掉电定时器的定时时间。

Option	Value
[-] 优化和功能使能	
[+] 看门狗	<input type="checkbox"/>
单片机重启函数	<input type="checkbox"/>
单片机电源控制函数	<input type="checkbox"/>
[-] 掉电唤醒定时器使能	<input checked="" type="checkbox"/>
定时时间	4095
[+] 复位配置寄存器	<input type="checkbox"/>

掉电定时器使用内部的32KHz的频率工作，其定时时间计算公式和预估时间如下所示：

$$\text{掉电唤醒定时器定时时间} = \frac{10^6 \times 16 \times \text{计数次数}}{F_{\text{wt}}} \quad (\text{微秒})$$

调用例程

关机键的核心：

```
1  if(KEY_OFF==0){ //当按键OFF按下时。
2      power_powerdown(); //进入掉电模式，此时功耗很低。
3  }
```

注意事项

- 1. 掉电模式可以通过外部触发的中断唤醒。不仅仅是外部中断脚，串口接收中断或者IIC起始帧中断这种也行，只要这个信号是从外部进来的就可以。
- 2. 内部能唤醒单片机的只有掉电计时器，如有需要请在图形化配置界面里提前设置好。
- 3. 如果希望单片机只能用某一个中断唤醒，那么在进入掉电模式前，请把其他的中断使能关掉，只留下那个中断。
- 4. 掉电状态下被唤醒了之后，CPU将会从掉电指令后继续执行，不会重启到main函数。
- 5. 为了优化空间，本函数默认不使能。请在power.h文件的图形化配置界面里使能【单片机电源控制函数】。

	Option	Value
	[-] 优化和功能使能	
	[+] 看门狗	<input type="checkbox"/>
	单片机重启函数	<input type="checkbox"/>
	单片机电源控制函数	<input checked="" type="checkbox"/>
	[-] 掉电唤醒定时器使能	<input type="checkbox"/>
	[+] 复位配置寄存器	<input type="checkbox"/>

power_cpu_idle

函数原型：void power_cpu_idle(void);

描述

CPU空闲函数，单片机进入空闲模式，CPU的电源都会被关掉，但是外设的电源依然存在。

输入

无

输出

无

返回值

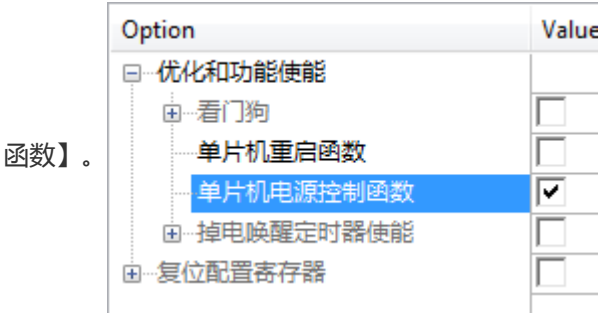
无

调用例程

```
1 while(1){//主循环中，
2     power_cpu_idle();//进入空闲模式，等待某种条件唤醒。
3     do_something();//唤醒后执行某些动作。
4 }
```

注意事项

- 1. 空闲模式由于外设的电源没有关，所以外设依然会工作，单片机整体的功耗基本不会下降太多。
- 2. 似乎这个功能太鸡肋，因为掉电模式也可以从掉电语句后面开始执行代码。所以在一些最新的STC型号里，取消了空闲模式。因此我推荐用掉电模式就可以了，空闲模式最好不用。
- 3. 为了优化空间，本函数默认不使能。请在power.h文件的图形化配置界面里使能【单片机电源控制函数】。



wdt_start

函数原型：void wdt_start(void);

描述

看门狗开启函数，看门狗打开后直到重启前都不能关掉。

输入

无

输出

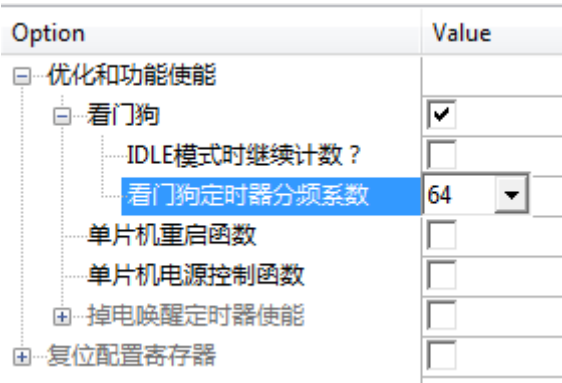
无

返回值

无

参数配置

看门狗就是一个不断在跑的定时器，如果看门狗定时器溢出，看门狗就会复位单片机。而这个时间在power.h文件的图形化配置界面里可以看到：



看门狗的定时时间和分频系数有关，公式和大致的预估时间如下图所示：

WDT_PS[2:0]: 看门狗定时器时钟分频系数

WDT_PS[2:0]	分频系数	12M 主频时的溢出时间	20M 主频时的溢出时间
000	2	≈ 65.5 毫秒	≈ 39.3 毫秒
001	4	≈ 131 毫秒	≈ 78.6 毫秒
010	8	≈ 262 毫秒	≈ 157 毫秒
011	16	≈ 524 毫秒	≈ 315 毫秒
100	32	≈ 1.05 秒	≈ 629 毫秒
101	64	≈ 2.10 秒	≈ 1.26 秒
110	128	≈ 4.20 秒	≈ 2.52 秒
111	256	≈ 8.39 秒	≈ 5.03 秒

看门狗溢出时间计算公式如下：

看门狗溢出时间 =

$$\frac{12 \times 32768 \times 2^{(WDT_PS+1)}}{SYSclk}$$

调用例程

```
1 | wdt_start(); //打开看门狗。
```

注意事项

- 1. 看门狗一但开启就不能关闭，除非单片机重启或断电。

2. 使用看门狗之前在图形化配置界面勾选【看门狗】的使能。

Option	Value
<input checked="" type="checkbox"/> 优化和功能使能	
<input checked="" type="checkbox"/> 看门狗	<input checked="" type="checkbox"/>
IDLE模式时继续计数？	<input type="checkbox"/>
看门狗定时器分频系数	64
单片机重启函数	<input type="checkbox"/>
单片机电源控制函数	<input type="checkbox"/>
<input checked="" type="checkbox"/> 掉电唤醒定时器使能	<input type="checkbox"/>
<input checked="" type="checkbox"/> 复位配置寄存器	<input type="checkbox"/>

3. 如果在空闲模式下使用看门狗，要勾选【IDLE模式时继续计数】。但勾选之后必须在看门狗溢出前喂狗，否则单片机就会重启。

wdt_feed

函数原型：void wdt_feed(void);

描述

看门狗喂狗函数，清零看门狗定时器。

输入

无

输出

无

返回值

无

调用例程

```
1 wdt_start(); //先打开看门狗。
2 while(1){ //主循环里
3     do_something(); //做某事。
4     wdt_feed(); //做完之后喂狗。
5 }
```

注意事项

1. 先执行wdt_start函数打开看门狗，然后喂狗函数才有意义。
2. 喂狗函数的执行位置不限，执行时间也不限但要在看门狗溢出之前至少执行一次。
3. 如果某个函数的执行时间大于看门狗的溢出时间，那么最好在该函数内部添加多句喂狗函数。**严禁为了偷懒而在定时器中断里喂狗**，因为有时候在执行一些函数的时候会陷入死循环中，但此时不影响中断的跳转。如果在中断喂狗，意味这个异常不会退出。

power_rstcfg_init

函数原型：void power_rstcfg_init(void);

描述

复位寄存器初始化函数。

输入

无

输出

无

返回值

无

参数配置

在使用这个函数之前先使能【复位配置寄存器】：

Option	Value
优化和功能使能	
复位配置寄存器	<input checked="" type="checkbox"/>
低压复位	<input type="checkbox"/>
RST脚功能	普通IO口(P54)
低压检测阈值电压设置	2.2V

如果需要低压复位，那么就勾选【低压复位】使能。低压复位的功能就是当单片机的VCC低于某个电压的时候，复位单片机。在下面的【电压检测阈值电压设置】那里设置触发低压复位的电压阈值。**需要注意的是，检测到低压之后，不一定非得复位。**设置好阈值电压之后，不勾选【低压复位】使能，单片机就会触发低电压中断。可以利用该中断，做一些断电保存参数的事情。

RST脚在STC8里是可以复用成IO口的，对应着P5.4。

在这里可以设置成IO口或者RST脚，由于本函数是main函数里执行，那么STC-ISP上的配置就会在执行本函数的时候失效。比如在STC-ISP上设置RST/P5.4脚为复位脚，而在ECBM库中设置为P5.4脚。那么在单片机BootLoader阶段，该脚是复位脚，在执行main之后，该脚就变成了P5.4脚。

调用例程

无，该函数会在system_init函数中自动调用。

注意事项

1. 本函数所执行的操作和设置，都可以在STC-ISP上设置。所以从优化角度来看可以不用这部分。
2. 之所以写成库，是为了一种情况：开源固件的时候，有些设置比较重要（比如P5.4脚一定得是RST脚），而使用开源固件的人不一定会去STC-ISP上设置这个选项，那么这个时候该函数就能发挥作用了。

优化建议

基本每个系列的函数（比如重启系列有两个，电源控制有两个）都有使能选项，不用到的函数不使能即可。

UART库

UART库是有关于单片机的串口操作库。在使用本库之前先到ecbm_core.h里使能。双击打开ecbm_core.h文件，然后进入图形化配置界面，使能UART库。虽然为了自动下载功能，UART库是默认使能的，但也最好确认一遍。

Option	Value
单片机系统设置	
深度优化和测试功能	
单片机外设库的选用	
POWER库	<input type="checkbox"/>
UART库	<input checked="" type="checkbox"/>
自动下载功能	<input checked="" type="checkbox"/>

所谓的【自动下载功能】是指ECBM库会在串口接收中断处，做一个重复下载流的判断。当接收到233个重复的下载流数据之后，就会重启单片机已达到免冷启动的效果。

API

uart_init

函数原型：void uart_init(void);

描述

串口初始化函数。

输入

无

输出

无

返回值

无

参数配置

本函数初始化的参数都是由图形化配置界面来设置，双击打开uart.h文件，进入图形化配置界面。

Option	Value
<input checked="" type="checkbox"/> uart_printf函数	<input checked="" type="checkbox"/>
printf函数缓存	64
<input type="checkbox"/> stream框架	<input type="checkbox"/>
挂载的串口	串口1
挂载的定时器	定时器3
<input checked="" type="checkbox"/> 串口1使能与设置	<input checked="" type="checkbox"/>
波特率	115200
工作模式	可变波特率8位数据方式
允许接收	<input checked="" type="checkbox"/>
+ 多机通信模式	<input type="checkbox"/>
波特率加倍控制位	不加倍
模式0的加倍控制位	不加倍，固定为Fosc/12
波特率产生器选择	定时器1
输出引脚	RxD-P30 TxD-P31(所有型号)
校验方式	无校验
开放串口1发送回调函数	<input type="checkbox"/>
开放串口1接收回调函数	<input type="checkbox"/>
+ 485控制功能	<input type="checkbox"/>

设置说明如下：

串口1

- **串口1使能与设置**：勾选了这个之后，串口1相关的代码才会编译。
- **波特率**：就是常说的波特率，这里列出来的值都是标准的波特率。
- **工作模式**：串口虽然有同步和异步之分，但90%的应用都是异步的。选择【可变波特率8位数据方式】或者【可变波特率9位数据方式】就行，其中不用校验就用8位，用校验就用9位。
- **允许接收**：需要接收数据就勾选吧，一般这都是要的。
- **多机通信模式**：通过软件协议也能实现多机通信，而且多机通信基本也都要有协议。所以这个功能算鸡肋了。
- **波特率加倍控制位**：建议保持默认。
- **模式0的加倍控制位**：建议保持默认。
- **波特率产生器选择**：可以在【定时器1】和【定时器2】中选一个。
- **输出引脚**：根据需要在选项里选择吧。注意括号里的提示，有些型号不能映射到某些IO口的。
- **校验方式**：无校验、奇校验、偶校验、0校验、1校验可选。
- **开放串口1发送回调函数**：使能之后必须定义发送回调函数，否则单片机会跑飞。
- **开放串口1接收回调函数**：使能之后必须定义接收回调函数，否则单片机会跑飞。
- **485控制功能**：使能之后，该串口的所有发送函数都会追加一个或两个控制脚去控制485芯片的收发方向。

串口2

- **串口2使能与设置**：勾选了这个之后，串口2相关的代码才会编译。
- **波特率**：就是常说的波特率，这里列出来的值都是标准的波特率。
- **通信位数**：异步模式。不用校验就选择【8位数据】，用校验就用【9位数据】。
- **允许接收**：需要接收数据就勾选吧，一般这都是要的。
- **多机通信模式**：通过软件协议也能实现多机通信，而且多机通信基本也都要有协议。所以这个功能算鸡肋了。
- **输出引脚**：根据需要在选项里选择吧。注意括号里的提示，有些型号不能映射到某些IO口的。
- **校验方式**：无校验、奇校验、偶校验、0校验、1校验可选。
- **开放串口2发送回调函数**：使能之后必须定义发送回调函数，否则单片机会跑飞。

- **开放串口2接收回调函数**：使能之后必须定义接收回调函数，否则单片机会跑飞。
- **485控制功能**：使能之后，该串口的所有发送函数都会追加一个或两个控制脚去控制485芯片的收发方向。

串口3

- **串口3使能与设置**：勾选了这个之后，串口3相关的代码才会编译。
- **波特率**：就是常说的波特率，这里列出来的值都是标准的波特率。
- **通信位数**：异步模式。不用校验就选择【8位数据】，用校验就用【9位数据】。
- **波特率产生器选择**：可以在【定时器2】和【定时器3】中选一个。
- **允许接收**：需要接收数据就勾选吧，一般这都是要的。
- **多机通信模式**：通过软件协议也能实现多机通信，而且多机通信基本也都要有协议。所以这个功能算鸡肋了。
- **输出引脚**：根据需要在选项里选择吧。注意括号里的提示，有些型号不能映射到某些IO口的。
- **校验方式**：无校验、奇校验、偶校验、0校验、1校验可选。
- **开放串口3发送回调函数**：使能之后必须定义发送回调函数，否则单片机会跑飞。
- **开放串口3接收回调函数**：使能之后必须定义接收回调函数，否则单片机会跑飞。
- **485控制功能**：使能之后，该串口的所有发送函数都会追加一个或两个控制脚去控制485芯片的收发方向。

串口4

- **串口4使能与设置**：勾选了这个之后，串口4相关的代码才会编译。
- **波特率**：就是常说的波特率，这里列出来的值都是标准的波特率。
- **通信位数**：异步模式。不用校验就选择【8位数据】，用校验就用【9位数据】。
- **波特率产生器选择**：可以在【定时器2】和【定时器4】中选一个。
- **允许接收**：需要接收数据就勾选吧，一般这都是要的。
- **多机通信模式**：通过软件协议也能实现多机通信，而且多机通信基本也都要有协议。所以这个功能算鸡肋了。
- **输出引脚**：根据需要在选项里选择吧。注意括号里的提示，有些型号不能映射到某些IO口的。
- **校验方式**：无校验、奇校验、偶校验、0校验、1校验可选。
- **开放串口4发送回调函数**：使能之后必须定义发送回调函数，否则单片机会跑飞。
- **开放串口4接收回调函数**：使能之后必须定义接收回调函数，否则单片机会跑飞。
- **485控制功能**：使能之后，该串口的所有发送函数都会追加一个或两个控制脚去控制485芯片的收发方向。

小科普

- **奇校验**：当要发送的8位数据里有奇数个1时，校验位为0；有偶数个1时，校验位为1。总体的1的数量为奇数。
- **偶校验**：当要发送的8位数据里有奇数个1时，校验位为1；有偶数个1时，校验位为0。总体的1的数量为偶数。
- **0校验**：校验位恒定为0。
- **1校验**：校验位恒定为1。

调用例程

打开自动下载功能的情况下：

```
1 #include "ecbm_core.h"//加载库函数的头文件。
2 void main(){//main函数，必须的。
3     system_init();//uart_init函数已经在这里面执行了。
4     while(1){
5
6     }
7 }
```

不打开自动下载功能的情况下：

```
1 #include "ecbm_core.h"//加载库函数的头文件。
2 void main(){//main函数，必须的。
3     system_init();//系统初始化函数，也是必须的。
4     uart_init();//初始化串口。
5     while(1){
6
7     }
8 }
```

注意事项

1. 如果在ecbm_core.h设置了自动下载功能的话，uart_init函数是会自动在system_init函数执行的时候被调用的。不需要自己再执行一遍。
2. uart_init执行的时候，将会设置波特率、映射IO口等信息。
3. 串口2只能是定时器2产生波特率，不能更改。
4. 定时器2可以给两个甚至4个串口提供波特率。条件是它们的**波特率必须一样**。

uart_set_io

函数原型：void uart_set_io(u8 id,u8 io);

描述

串口输出IO设置函数，可以将串口映射到别的IO口上。

输入

- id：串口的编号，按通用说法从1开始。
- io：要切换的目标IO。

输出

无

返回值

无

参数配置

在uart.h文件中的图形化配置界面里，打开所需的【串口n的使能与设置】，接下来在【输出引脚】处选择串口要映射的IO口。需要注意单片机的型号是否有这个IO口。

输出引脚	RxD-P30 TxD-P31(所有型号)
校验方式	RxD-P30 TxD-P31(所有型号)
开放串口1发送回调函数	RxD-P36 TxD-P37(除STC8G1K08和STC8G1K08A以外)
开放串口1接收回调函数	RxD-P16 TxD-P17(除STC8G1K08和STC8G1K08A以外)
485控制功能	RxD-P43 TxD-P44(除STC8G1K08和STC8G1K08A以外)
	RxD-P32 TxD-P33(仅限STC8G1K08和STC8G1K08A)
	RxD-P54 TxD-P55(仅限STC8G1K08和STC8G1K08A)

在图形化配置界面设置了之后，就算没有调动本函数，也会在uart_init函数里执行生效。

调用例程

由于STC8系列型号众多，封装脚数又会影响到IO的复用情况，所以**如果不是有串口反复切换的需求，还是推荐用图形化配置来设置IO映射**。图形化配置界面不仅有详细的型号说明，还不需要去背那一大串宏定义。

如果确实有切换IO的需求，那么请参考以下的宏定义：

- UART1_PIN_P30_P31：串口1映射到P3.0和P3.1，适合所有型号。
- UART1_PIN_P36_P36：串口1映射到P3.6和P3.7，适合除STC8G1K08和STC8G1K08A以外的型号。
- UART1_PIN_P16_P17：串口1映射到P1.6和P1.7，适合除STC8G1K08和STC8G1K08A以外的型号。
- UART1_PIN_P43_P44：串口1映射到P4.3和P4.4，适合除STC8G1K08和STC8G1K08A以外的型号。
- UART1_PIN_P32_P33：串口1映射到P3.2和P3.3，适合STC8G1K08和STC8G1K08A这两个型号。
- UART1_PIN_P54_P55：串口1映射到P5.4和P5.5，适合STC8G1K08和STC8G1K08A这两个型号。
- UART2_PIN_P10_P11：串口2映射到P1.0和P1.1，适合所有型号。
- UART2_PIN_P40_P42：串口2映射到P4.0和P4.2，适合STC8F和STC8A这两个系列。
- UART2_PIN_P46_P47：串口2映射到P4.6和P4.7，适合STC8G和STC8H这两个系列。
- UART3_PIN_P00_P01：串口3映射到P0.0和P0.1，适合所有型号。
- UART3_PIN_P50_P51：串口3映射到P5.0和P5.1，适合所有型号。
- UART4_PIN_P02_P03：串口1映射到P0.2和P0.3，适合所有型号。
- UART4_PIN_P52_P53：串口1映射到P5.2和P5.3，适合所有型号。

```
1  uart_set_io(1,UART1_PIN_P30_P31); //串口1映射到P3.0脚和P3.1脚。
2  uart_string(1,"串口1发送测试-3031"); //发送一个字符串。
3  uart_set_io(1,UART1_PIN_P16_P17); //串口1映射到P1.6脚和P1.7脚。
4  uart_string(1,"串口1发送测试-1617"); //发送一个字符串。
```

注意事项

1. 宏定义没有标识型号，一定要先看单片机的引脚图确认该型号能映射该IO口。简单的例子：对于STC8G1K08A-8PIN这个型号来说，是没有P4口的，那么UART1_PIN_P43_P44这个宏定义在这个型号是无效的。
2. 不是每一个型号都有4个串口，在使用的时候也要留意。
3. STC的命名规则里，RXD2和TXD2代表的是串口2，RXD_2和TXD_2代表的是串口1的第2组映射口。一定得留意！

uart_set_baud

函数原型：void uart_set_baud(u8 id,u32 baud);

描述

串口波特率设置函数。

输入

- id：串口的编号，按通用说法从1开始。
- baud：要设置的波特率。

输出

无

返回值

无

调用例程

```
1  if(strcmp(buf, "AT9600")){//比较接收到的字符串里有没有“AT9600”的字样。  
2      uart_set_baud(1,9600);//有的话就设置波特率为9600。  
3  }else if(strcmp(buf, "AT115200")){//比较接收到的字符串里有没有“AT115200”的字样。  
4      uart_set_baud(1,115200);//有的话就设置波特率为115200。  
5  }
```

注意事项

1. 本函数执行之后，设置的波特率会立即生效。因此尽量在通信结束后再执行，否则会导致后续的数据错误。
2. 虽然参数是可以随便输入任意波特率，但还是推荐那些常用的9600或者115200。否则计算出来的波特率和设置的波特率可能会相差很大。

uart_char

函数原型：void uart_char(u8 id,u8 ch);

描述

串口单个字节发送函数。

输入

- id：串口的编号，按通用说法从1开始。
- ch：要发送的字符或者数据。

输出

无

返回值

无

调用例程

矩阵按键按下后发送按键键值：

```
1 key_deal(); //矩阵键盘处理函数。
2 if(key_1==0){ //判断按键1的标志位，
3     uart_char(1, '1'); //按下时向串口1发送字符‘1’。
4 }
5 if(key_2==0){ //判断按键2的标志位，
6     uart_char(1, '2'); //按下时向串口1发送字符‘2’。
7 }
8 //剩余14个按键处理代码省略。
```

注意事项

1. 本函数兼顾了奇校验、偶校验、0校验和1校验这4种通用校验。在设置了校验模式之后，不需要修改本函数，函数内部会自动加上校验。
2. 由于永久内置校验功能会导致效率降低，所以各个检验功能都是由宏定义开关来决定是否编译。因此**没有办法在单片机运行的时候更换校验模式**，校验模式在编译的时候就已经固定下来了。
3. 每个串口的校验模式是独立的。

uart_char_9

函数原型：void uart_char_9(u8 id,u8 ch,u8 bit9);

描述

串口单个字节发送函数，发送9位数据。

输入

- id：串口的编号，按通用说法从1开始。
- ch：要发送的字符或者数据。
- bit9：要发送的第9位数据。

输出

无

返回值

无

调用例程

自定义协议之“0代表地址1代表数据”：

```
1 | uart_char_9(1,14,0); //发送给ID为14的从机。(第9位是0, 按自定义协议是地址。)  
2 | uart_char_9(1,0xA5,1); //帧头。(第9位是1, 按自定义协议是数据。)  
3 | uart_char_9(1,0x05,1); //数据1。  
4 | uart_char_9(1,0x12,1); //数据2。  
5 | uart_char_9(1,0x17,1); //校验和, 一般就是数据1+数据2。  
6 | uart_char_9(1,0x5A,1); //帧尾。
```

注意事项

1. 本函数和uart_char函数的区别就在于检验位是开放的, 可以由用户自定义一种检验位。也可以像例程那样当做地址/数据区分位, 反正就是十分自由。
2. 如果说uart_char函数的检验模式不能在运行时候修改很遗憾的话, 可以用本函数来实现可变校验通信。

uart_string

函数原型 : void uart_string(u8 id,u8 * str);

描述

串口字符串发送函数。

输入

- id : 串口的编号, 按通用说法从1开始。
- str : 要发送的字符或者数据。

输出

无

返回值

无

调用例程

经典的hello world :

```
1 | uart_string(1,"Hello world\r\n"); //发送字符串Hello world并回车。
```

注意事项

1. 本函数基于uart_char函数, 所以校验模式也是支持奇、偶、0、1这4种校验。

uart_printf

函数原型 : void uart_printf(u8 id,u8 * str,...);

描述

串口打印函数。

输入

- id：串口的编号，按通用说法从1开始。
- str：要发送的格式化内容。
- ...：格式化的参数。

输出

无

返回值

无

调用例程

调试运行次数：

```
1  while(1){//主循环里
2      if(in_io){//判断输入标志位，如果有信号来的话，
3          in_io=0;//先清除标志位。
4          count++;//统计信号数量。
5      }
6      if(time>=1000){//判断时间变量，如果等于1000，说明到1秒了。
7          time=0;//清零变量，从0开始计时。
8          uart_printf(1,"一秒内收到%u个信号\r\n",count);//打印count变量的值。
9          count=0;//清零，为下一次的统计做准备。
10     }
11 }
```

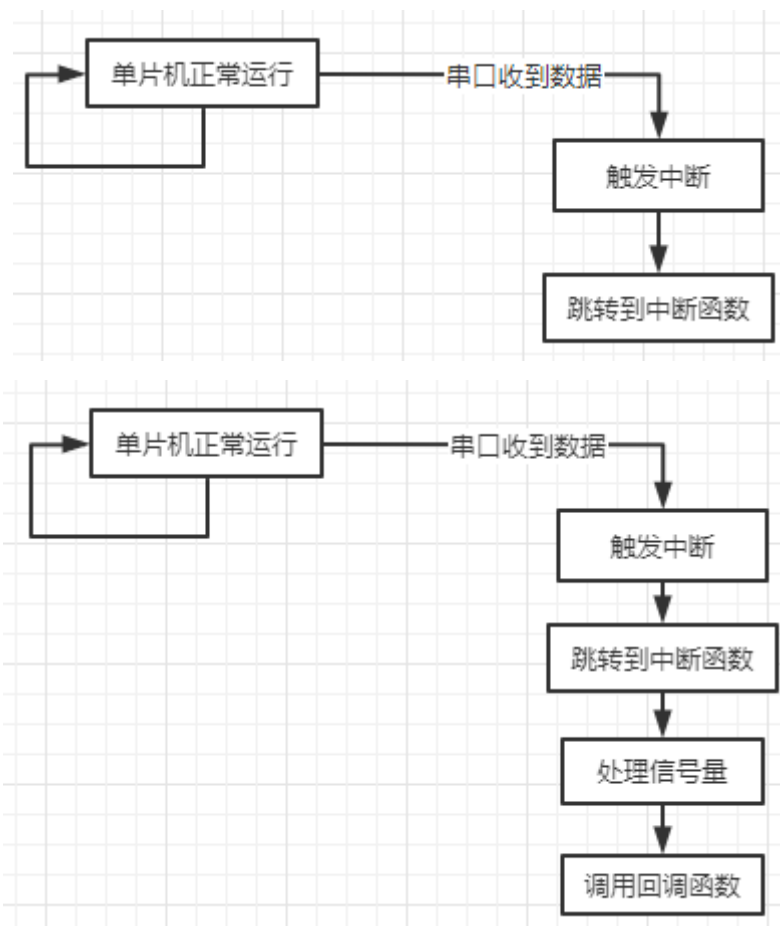
注意事项

1. 本函数基于uart_string函数，所以校验模式也是支持奇、偶、0、1这4种校验。
2. 本函数是使用stdio库实现的格式化转换，一些高手会自己写printf来节约flash空间。所以本函数是可以通过关闭使能来优化掉的。
3. 占位符%d和%u在C51的printf函数里只能用于u16型变量的显示，uart_printf函数也是基于printf系列的，所以也有这个问题。解决办法有二，一是将占位符换成%bd和%bu，二是把u8的变量强转成u16型。
4. ECBM库所有的串口发送函数都是中断发送，因此**不要在其他中断函数中调用uart_printf函数**，否则会引起中断嵌套直至跑飞。

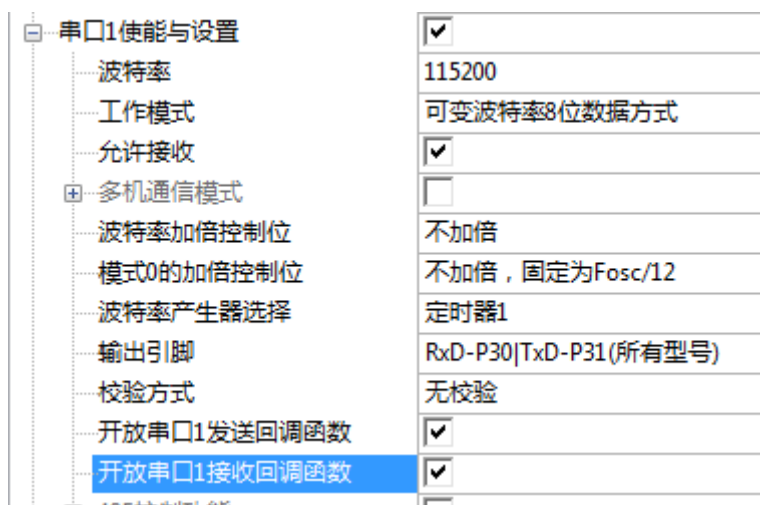
回调函数

串口的发送可以通过调用发送函数，那么接收主要是靠回调函数。“回调函数”没有那么神秘，一般而言正常的函数都是设计者定义好函数内容，由使用者决定在何处调用。回调函数是设计者先决定好在何处调用，然后由使用者来决定函数的内容。

串口的使用有查询法和中断法之分，回调函数是中断法的应用。与常规中断法的流程区别是：



在使用回调函数前，根据需要使能发送回调或者接收回调。



然后在某个.c文件里定义回调函数，为了快速和安全的跳转，本库没有把回调函数用函数指针传入中断处理函数，而是直接调用了指定名字的函数。这意味着回调函数的名字不能随便起，只能是以下这些名字：

- uart1_receive_callback：串口1接收回调函数。
- uart1_send_callback：串口1发送回调函数。
- uart2_receive_callback：串口2接收回调函数。
- uart2_send_callback：串口2发送回调函数。
- uart3_receive_callback：串口3接收回调函数。
- uart3_send_callback：串口3发送回调函数。
- uart4_receive_callback：串口4接收回调函数。
- uart4_send_callback：串口4发送回调函数。

调用例程

```
1 #include "ecbm_core.h"//加载库函数的头文件。
2 void main(){//main函数，必须的。
3     system_init();//系统初始化函数。
4     while(1){
5
6     }
7 }
8 void uart1_receive_callback(void){//接收回调函数
9     if(SBUF=='1'){//收到字符‘1’的时候，
10         LED_ON;//点亮LED。
11     }else if(SBUF=='0'){//收到字符‘0’的时候，
12         LED_OFF;//熄灭LED。
13     }
14 }
```

注意事项

1. 每发送一个字节的数据或者接收一个字节的数据，就会执行一次对应的回调函数。
2. 回调函数在串口中断中调用，因此不要在回调函数内执行任何串口发送函数，否则中断嵌套会导致程序崩溃。
3. 最好在main.c里定义回调函数。

优化建议

1. 没有用到的串口不使能。
2. 不使能uart_printf函数，可以自己实现一个更小巧的打印函数。
3. 不是必要的话，就不开校验位。

EXTI库

EXTI就是常说的外部中断，EXTI库就是关于单片机的外部中断的操作库。在使用本库之前先到ecbm_core.h里使能。双击打开ecbm_core.h文件，然后进入图形化配置界面，使能EXTI库。

Option	Value
单片机系统设置	
深度优化和测试功能	
单片机外设库的选用	
POWER库	<input type="checkbox"/>
UART库	<input checked="" type="checkbox"/>
EXTI库	<input checked="" type="checkbox"/>
TIMER库	<input type="checkbox"/>

API

exti_init

函数原型：void exti_init(void);

描述

外部中断初始化函数。

输入

无

输出

无

返回值

无

参数配置

本函数初始化的参数都是由图形化配置界面来设置，双击打开exti.h文件，进入图形化配置界面。

Option	Value
<input checked="" type="checkbox"/> 外部中断0	<input checked="" type="checkbox"/>
中断模式选择	只下降沿
初始化时就打开中断？	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> 外部中断1	<input checked="" type="checkbox"/>
中断模式选择	上升沿/下降沿
初始化时就打开中断？	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> 外部中断2	<input checked="" type="checkbox"/>
初始化时就打开中断？	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> 外部中断3	<input checked="" type="checkbox"/>
初始化时就打开中断？	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> 外部中断4	<input checked="" type="checkbox"/>
初始化时就打开中断？	<input checked="" type="checkbox"/>

设置说明如下：

外部中断0

- **外部中断0**：勾选这个才会编译外部中断0的初始化代码。
- **中断模式选择**：有【上升沿/下降沿】和【只下降沿】两个可以选择。
- **初始化时就打开中断**：勾选后外部中断0的中断使能会在初始化函数里打开。

外部中断1

- **外部中断1**：勾选这个才会编译外部中断1的初始化代码。
- **中断模式选择**：有【上升沿/下降沿】和【只下降沿】两个可以选择。
- **初始化时就打开中断**：勾选后外部中断1的中断使能会在初始化函数里打开。

外部中断2

- **外部中断2**：勾选这个才会编译外部中断2的初始化代码。
- **初始化时就打开中断**：勾选后外部中断2的中断使能会在初始化函数里打开。

外部中断3

- **外部中断3**：勾选这个才会编译外部中断3的初始化代码。
- **初始化时就打开中断**：勾选后外部中断3的中断使能会在初始化函数里打开。

外部中断4

- **外部中断4**：勾选这个才会编译外部中断4的初始化代码。
- **初始化时就打开中断**：勾选后外部中断4的中断使能会在初始化函数里打开。

小科普

- 这里的外部中断是和标准51类似工作原理的外部中断。STC8的最新型号里还有其他原理实现的外部中断，为防止混淆ECBM库会把那个新的中断称之为**IO中断**。
- STC8的中断模式是【上升沿/下降沿】和【只下降沿】。标准51是【低电平】和【下降沿】。

调用例程

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  void main(){//main函数，必须的。
3      system_init();//系统初始化函数。
4      exti_init();//初始化外部中断脚。
5      while(1){
6
7      }
8  }
```

注意事项

1. 只有外部中断0和外部中断1可以选择中断模式。
2. 【上升沿/下降沿】模式下，无论是上升沿还是下降沿都会触发中断，此时需要读一遍IO口的电平值来确认是什么边沿触发的中断。

exti_start

函数原型：void exti_start(u8 id);

描述

打开中断函数。

输入

- id：打开的外部中断编号，可输入范围是0~4。

输出

无

返回值

无

调用例程

```
1  if(control_mode==KEY_MODE){//如果操作模式切换成按键控制，
2      exti_start(0); //打开按键所在的外部中断0。
3  }
```

注意事项

1. 如果在图形化配置界面选择了【初始化时就打开中断】，那么本函数将会在exti_init函数执行的时候自动调用，不需要额外执行了。反过来，假设没有选择【初始化时就打开中断】，那么一定得执行本函数，外部中断才能正常工作。

exti_stop

函数原型：void exti_stop(u8 id);

描述

关闭中断函数。

输入

- id：关闭的外部中断编号，可输入范围是0~4。

输出

无

返回值

无

调用例程

```
1  if(control_mode==UART_MODE){//如果操作模式切换成串口控制，
2      exti_stop(0); //关闭按键所在的外部中断0。
3  }
```

注意事项

无

exti_set_mode

函数原型：void exti_set_mode(u8 id,u8 mode);

描述

中断模式设置函数，受单片机框架原因只有外部中断0和外部中断1可用。

输入

- id：设置的外部中断编号，0或者1。
- mode：设置的中断模式。

输出

无

返回值

无

调用例程

工作的模式可以在图形化配置界面设置，也可以通过调用函数来修改，实现动态触发的效果。

参数的宏定义是：

- EXTI_MODE_UP_DOWN：上升沿和下降沿都会触发中断。
- EXTI_MODE_DOWN：只有下降沿才会触发中断。

```
1 | exti_set_mode(0, EXTI_MODE_DOWN); //将外部中断0的中断模式设置成下降沿中断。
```

注意事项

1. 外部中断没有相应的标志显示是什么边沿触发的中断，在设置上升沿和下降沿模式之后，需要手动判断IO的电平来区分边沿的类型。比如触发中断后，IO是高电平，说明是上升沿触发。同理，触发中断后，IO是低电平，说明是下降沿触发。

回调函数

在串口库那一章里有提到过回调函数的执行原理。目前为了统一管理中断，ECBM库已经定义好了中断处理函数，并开放了回调函数供大家使用。只要按着回调函数的名字定义一个一模一样的函数就行。

- exti0_it_callback
- exti1_it_callback
- exti2_it_callback
- exti3_it_callback
- exti4_it_callback

调用例程

```
1 | #include "ecbm_core.h" //加载库函数的头文件。
2 | void main() { //main函数，必须的。
3 |     system_init(); //系统初始化函数。
4 |     exti_init(); //初始化外部中断脚。
5 |     while(1) {
6 |
7 |     }
8 | }
9 | void exti0_it_callback(void) { //这是外部中断0的回调函数。
10 |     LED=!LED; //当触发外部中断0中断时，取反LED的亮灭状态。
11 | }
```

注意事项

1. 回调函数的名字一定得正确，因为中断里面会调用这些回调函数，如果名字不正确就调用不了。
2. 本函数不需要也不能在其他地方调用！只需要定义了即可。因为在外中断触发的时候，单片机的硬件会自动去调用的。

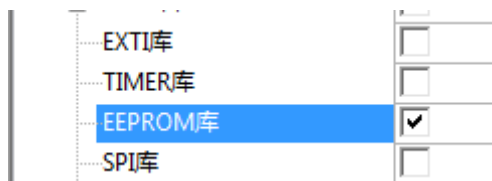
优化建议

本库内容比较简单，所以可优化的地方不多。基本保证没用到的功能不使能就行。

EEPROM库

STC的eeprom用法和平时常见的以AT24C02为代表的eeprom不一样。从名字上来说它们都可以叫“电可擦编程只读存储器(Electrically Erasable Programmable Read-Only Memory)”。但AT24C02能以字节为单位进行读写，而STC的eeprom是用flash模拟的，所以写入可以写单个字节、而擦除只能擦除整个扇区。

在使用本库之前先到ecbm_core.h里使能。双击打开ecbm_core.h文件，然后进入图形化配置界面，使能EEPROM库。



API

eeprom_init

函数原型：void eeprom_init(void);

描述

eeprom初始化函数。

输入

无

输出

无

返回值

无

调用例程

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  void main(){//main函数，必须的。
3      system_init();//系统初始化函数。
4      eeprom_init();//初始化eeprom。
5      while(1){
6
7      }
8  }
```

注意事项

无

eeprom_erase

函数原型：void eeprom_erase(u16 addr);

描述

eeprom擦除函数。

输入

- addr：要擦除的地址。

输出

无

返回值

无

小科普

STC的eeprom是用flash的模拟的，flash的特点有：

1. 读可以一个字节一个字节读。
2. 写可以一个字节一个字节写，但是写操作只能把数据里的1写成0（“1写成1”和“0写成0”这两种因为数据不变所以算不上写入）。

比如：地址0原来的数据是0x09（0000 1001），写入0x01（0000 0001）是可以的，因为写入0x01会把D7位从0写成0、D6位从0写成0、D5位从0写成0、D4位从0写成0、D3位从1写成0、D2位从0写成0、D1位从0写成0、D0位从1写成1。本次操作中，除了D3位是1写0外，其他位数据不变。

如果地址0原来的数据是0x09（0000 1001），写入0x03（0000 0011）是不可以的，因为写入0x03会把D7位从0写成0、D6位从0写成0、D5位从0写成0、D4位从0写成0、D3位从1写成0、D2位从0写成0、D1位从0写成1、D0位从1写成1。本次操作中，只有两位改变了，分别是D3位的1写0和D1的0写1。其中D3位能写成功，D1位的0写1不成功，所以D1还是0。于是源数据0x09（0000 1001）只有D3位1写0，最终结果就是0x01（0000 0001）。

这就是为什么STC的eeprom每次写入之前都要擦除的原因。

3. 擦除是按扇区擦除，擦除是flash能把数据里0写成1的唯一操作。擦除操作把一个扇区共512字节的数据全变成0xFF，之后就可以任意写入任意值了。

调用例程

```
1 eeprom_erase(20); //擦除地址20所在的扇区。
```

注意事项

1. STC8的扇区是512字节。所以本函数只要一执行，参数地址所在的扇区都会被擦除。比如例程中擦除地址为20，地址20所在的扇区0的范围是0~511。所以0~511地址的数据都会被擦除掉。
2. 本库中的eeprom操作是通过IAP寄存器的，STC的硬件在运行时会在IAP寄存器里自动加入flash空间偏移，因此无论是哪个型号的STC8单片机，eeprom的地址都是从0开始的！STC手册上给的地址偏移仅针对MOVC法，IAP法永远都是从地址0开始！

eeprom_write

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

描述

eeprom写入函数。

输入

- addr：要写入的地址。
- dat：要写入该地址的数据。

输出

无

返回值

无

调用例程

上电后的第一次写入：

```
1 //单片机flash在下载的时候都擦除一遍了，因此默认就是0xFF，所以第一次不用擦除。  
2 eeprom_write(20,125); //向地址20写入数据125。
```

上电后的第N次写入：

```
1 eeprom_erase(20); //第N次写入的时候，由于其地址内容不一定是0xFF，所以要先擦除地址20所在  
    的扇区。  
2 eeprom_write(20,15); //向地址20写入数据15。
```

注意事项

1. 单片机电压低的时候，会导致写入异常。尽量保证单片机在3V以上电压工作吧。
2. 本库中的eeprom操作是通过IAP寄存器的，STC的硬件在运行时会在IAP寄存器里自动加入flash空间偏移，因此无论是哪个型号的STC8单片机，eeprom的地址都是从0开始的！STC手册上给的地址偏移仅针对MOVC法，IAP法永远都是从地址0开始！

EEPROM_READ

函数原型：u8 EEPROM_READ(u16 addr);

描述

EEPROM读取函数。

输入

- addr：要读取的地址。

输出

无

返回值

- 该地址的数据。

调用例程

```
1 val=EEPROM_READ(20); //读取地址20的数据赋予变量val。
```

注意事项

1. 有人反馈说EEPROM写入数据立刻读取会读取到异常数据，可以适当在写入函数和读取函数之间加入几毫秒延时。
2. 本库中的EEPROM操作是通过IAP寄存器的，STC的硬件在运行时会在IAP寄存器里自动加入flash空间偏移，因此无论是哪个型号的STC8单片机，EEPROM的地址都是从0开始的！STC手册上给的地址偏移仅针对MOVC法，IAP法永远都是从地址0开始！

EEPROM_READ_EX

函数原型：void EEPROM_READ_EX(u16 addr,u8 * dat,u16 num);

描述

EEPROM批量读取函数。

输入

- addr：要读取的地址。
- num：要读取的数量。

输出

- dat：读取到的数据。

返回值

无

调用例程


单个字节读取：

```
1 u8 val;  
2 eeprom_read_ex(0,&val,1); //从地址0读取一个数据到变量val。
```

多个字节读取：

```
1 u8 val[10];  
2 eeprom_read_ex(0,val,10); //从地址0开始连续读取10个数据到数组val。
```

注意事项

1. 本函数需要在图形化配置界面使能【开放EEPROM延伸函数】才能使用。
2. 有人反馈说eeprom写入数据立刻读取会读取到异常数据，可以适当在写入函数和读取函数之间加入几毫秒延时。
3. 本库中的eeprom操作是通过IAP寄存器的，STC的硬件在运行时会在IAP寄存器里自动加入flash空间偏移，因此无论是哪个型号的STC8单片机，eeprom的地址都是从0开始的！STC手册上给的地址偏移仅针对MOVC法，IAP法永远都是从地址0开始！

eeprom_write_ex

函数原型：void eeprom_write_ex(u16 addr,u8 * dat,u16 num);

描述

eeprom批量写入函数。

输入

- addr：要写入的地址。
- dat：要写入的数据。
- num：要写入的数量。

输出

无

返回值

无

调用例程

单个字节写入：

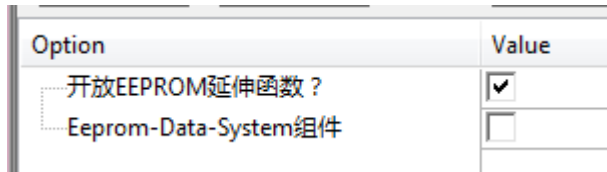
```
1 u8 val;  
2 val=100;  
3 eeprom_write_ex(0,&val,1); //把变量val的值写到地址0。
```

多个字节写入：

```
1 u8 val[10]={10,32,43,4,65,96,17,38,49,20}; //10个数据
2 eeprom_write_ex(0,val,10); //将以上10个数据写入地址0~9。
```

注意事项

1. 本函数需要在图形化配置界面使能【开放EEPROM延伸函数】才能使用。



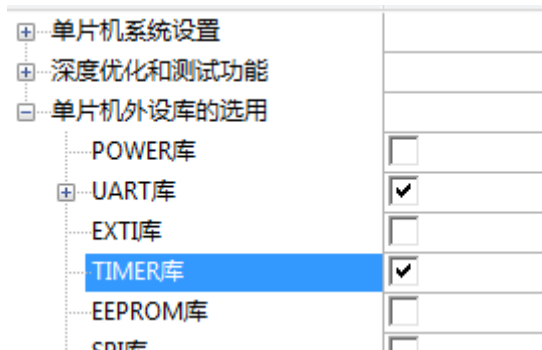
2. 由于擦除操作一次会擦除512字节的数据，所以本函数会在xdata区开设512字节的缓存用于缓存没有被修改的数据。如果单片机xdata空间紧张，可以不用这个函数。
3. 本库中的eeprom操作是通过IAP寄存器的，STC的硬件在运行时会在IAP寄存器里自动加入flash空间偏移，因此无论是哪个型号的STC8单片机，eeprom的地址都是从0开始的！STC手册上给的地址偏移仅针对MOVC法，IAP法永远都是从地址0开始！

优化建议

如果存入的数据很少，或者是不需要频繁改变。可以把扩展函数的使能取消掉，不仅可以节省FLASH空间还能节省512字节的XDATA空间。

TIMER库

TIMER就是常说的定时器，TIMER库就是关于单片机定时器的操作库。在使用本库之前先到ecbm_core.h里使能。双击打开ecbm_core.h文件，然后进入图形化配置界面，使能TIMER库。



定时器的本质就是计数器，因此在接下来的文章中，你会看到定时/计数两个方向的应用。他们的区别在于：计数器是对外部脉冲进行计数，由于外部脉冲的周期和个数都不确定，所以只能统计脉冲的个数；定时器是对内部系统时钟进行计数，由于内部系统时钟的脉冲周期是确定的，根据公式“脉冲周期值×脉冲个数=经过的时间”就能算出时间。也就是说只要外部脉冲是连续不断的、周期恒定的，也可以用外部脉冲来做定时应用。

API

timer_start

函数原型：void timer_start(u8 id);

描述

定时器开启函数。

输入

- id：要开启的定时器编号，从0开始。

输出

无

返回值

无

调用例程

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  void main(){//main函数，必须的。
3      system_init();//系统初始化函数，也是必须的。
4      timer_init();//先初始化定时器。
5      timer_start(0);//再打开定时器0。
6      while(1){
7
8      }
9  }
```

注意事项

1. 使用本函数之前，要先初始化定时器。
2. 在【定时器开关】选项选择了“软硬件开关”之后，除了执行本函数之外还需要对应的引脚输入高电平才可开启定时器。

Option	Value
☐ 定时器0使能与设置	<input checked="" type="checkbox"/>
▢ 定时器开关	软硬件开关。 ▾
▢ 计数来源	对系统时钟计数（定时器应用）

timer_stop

函数原型：void timer_stop(u8 id);

描述

定时器关闭函数。

输入

- id：要关闭的定时器编号，从0开始。

输出

无

返回值

无

调用例程

```
1  if(key_stop==0){//当停止按键被按下的时候，  
2      timer_stop(0); //关闭定时器0。  
3  }
```

注意事项

无

timer_out

函数原型：void timer_out(u8 id,u8 en);

描述

定时器输出控制函数。

输入

- id：要设置输出的定时器编号，从0开始。
- en：1代表开启时钟输出，0代表关闭时钟输出。

输出

无

返回值

无

输出引脚

- **定时器0**：P35脚。
- **定时器1**：P34脚。
- **定时器2**：P13脚。
- **定时器3**：P05脚。
- **定时器4**：P07脚。

调用例程

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  void main(){//main函数，必须的。
3      system_init();//系统初始化函数，也是必须的。
4      timer_init();//先初始化定时器。
5      timer_out(0,1);//然后打开定时器0的时钟输出。
6      timer_start(0);//最后打开定时器0。
7      while(1){
8
9      }
10 }
```

注意事项

1. 时钟输出的频率和定时器的溢出率有关，原理上来说定时器每溢出一次，对应的输出脚取反一次。这个过程是硬件全自动的，比软件取反要快很多，特别是几MHz的时钟输出。

timer_init

函数原型：void timer_init(void);

描述

定时器初始化函数。

输入

无

输出

无

返回值

无

参数配置

在图形化配置界面设置的参数，将会在执行本函数的时候写到定时器寄存器中。因此要保证选项选择正确无误。

Option	Value
<input checked="" type="checkbox"/> 定时器0使能与设置	<input checked="" type="checkbox"/>
定时器开关	软硬件开关。
计数来源	对系统时钟计数（定时器应用）
工作模式	16位自动重载模式
时钟分频	系统时钟不分频
对外输出时钟	<input checked="" type="checkbox"/>
定时时间/计数数量	65500
定时器0的中断使能	<input type="checkbox"/>
<input type="checkbox"/> 定时器1使能与设置	<input type="checkbox"/>
定时器开关	软件开关
计数来源	对系统时钟计数（定时器应用）
工作模式	16位自动重载模式
时钟分频	系统时钟不分频
对外输出时钟	<input type="checkbox"/>
定时时间/计数数量	65535
定时器1的中断使能	<input type="checkbox"/>
<input type="checkbox"/> 定时器2使能与设置	<input type="checkbox"/>
<input type="checkbox"/> 定时器3使能与设置	<input type="checkbox"/>
<input type="checkbox"/> 定时器4使能与设置	<input type="checkbox"/>
定时器1使能与设置 勾选该选项会使能定时器1，开放和定时器1相关的操作函数。若未使用定时器1，可以关掉优化空间。	

设置说明如下：

定时器0

- **定时器0使能与设置**：勾选之后，定时器0的代码才会参与编译。
- **定时器开关**：【软件开关】是指用代码开启关闭定时器，【软硬件开关】是指在代码开启之后，还需要P32为高电平才能开启定时器。
- **计数来源**：用作定时器的时候选择【对系统时钟计数（定时器应用）】，用作计数器的時候选择【对外部T0（P34）脚的脉冲信号计数（计数器应用）】。
- **工作模式**：以前性能低没得选，现在只推荐选【16位自动重载模式】。
- **时钟分频**：有【系统时钟不分频】和【系统时钟12分频（Fosc/12）】可以选择，根据实际情况来选。但是注意这个分频只能影响到定时器，不会影响定时器以外的外设的工作频率（串口用到了定时器，就还会受影响）。
- **对外输出时钟**：使能之后，P35脚就会输出时钟。
- **定时时间/计数数量**：在定时模式下，这个参数决定了定时时间；在计数模式下，这个参数决定了计数的个数。
- **定时器0的中断使能**：使能之后，定时器0溢出就触发中断。

定时器1

- **定时器1使能与设置**：勾选之后，定时器1的代码才会参与编译。
- **定时器开关**：【软件开关】是指用代码开启关闭定时器，【软硬件开关】是指在代码开启之后，还需要P33为高电平才能开启定时器。
- **计数来源**：用作定时器的时候选择【对系统时钟计数（定时器应用）】，用作计数器的時候选择【对外部T1（P35）脚的脉冲信号计数（计数器应用）】。
- **工作模式**：以前性能低没得选，现在只推荐选【16位自动重载模式】。
- **时钟分频**：有【系统时钟不分频】和【系统时钟12分频（Fosc/12）】可以选择，根据实际情况来选。但是注意这个分频只能影响到定时器，不会影响定时器以外的外设的工作频率（串口用到了定时器，就还会受影响）。
- **对外输出时钟**：使能之后，P34脚就会输出时钟。

- **定时时间/计数数量**：在定时模式下，这个参数决定了定时时间；在计数模式下，这个参数决定了计数的个数。
- **定时器1的中断使能**：使能之后，定时器1溢出就触发中断。

定时器2

- **定时器2使能与设置**：勾选之后，定时器2的代码才会参与编译。
- **计数来源**：用作定时器的时候选择【对系统时钟计数（定时器应用）】，用作计数器的时候选择【对外部T2（P12）脚的脉冲信号计数（计数器应用）】。
- **工作模式**：以前性能低没得选，现在只推荐选【16位自动重载模式】。
- **时钟分频**：有【系统时钟不分频】和【系统时钟12分频（Fosc/12）】可以选择，根据实际情况来选。但是注意这个分频只能影响到定时器，不会影响定时器以外的外设的工作频率（串口用到了定时器，就还会受影响）。
- **对外输出时钟**：使能之后，P13脚就会输出时钟。
- **定时时间/计数数量**：在定时模式下，这个参数决定了定时时间；在计数模式下，这个参数决定了计数的个数。
- **定时器2的中断使能**：使能之后，定时器2溢出就触发中断。

定时器3

- **定时器3使能与设置**：勾选之后，定时器3的代码才会参与编译。
- **计数来源**：用作定时器的时候选择【对系统时钟计数（定时器应用）】，用作计数器的时候选择【对外部T3（P04）脚的脉冲信号计数（计数器应用）】。
- **工作模式**：以前性能低没得选，现在只推荐选【16位自动重载模式】。
- **时钟分频**：有【系统时钟不分频】和【系统时钟12分频（Fosc/12）】可以选择，根据实际情况来选。但是注意这个分频只能影响到定时器，不会影响定时器以外的外设的工作频率（串口用到了定时器，就还会受影响）。
- **对外输出时钟**：使能之后，P05脚就会输出时钟。
- **定时时间/计数数量**：在定时模式下，这个参数决定了定时时间；在计数模式下，这个参数决定了计数的个数。
- **定时器3的中断使能**：使能之后，定时器3溢出就触发中断。

定时器4

- **定时器4使能与设置**：勾选之后，定时器4的代码才会参与编译。
- **计数来源**：用作定时器的时候选择【对系统时钟计数（定时器应用）】，用作计数器的时候选择【对外部T4（P06）脚的脉冲信号计数（计数器应用）】。
- **工作模式**：以前性能低没得选，现在只推荐选【16位自动重载模式】。
- **时钟分频**：有【系统时钟不分频】和【系统时钟12分频（Fosc/12）】可以选择，根据实际情况来选。但是注意这个分频只能影响到定时器，不会影响定时器以外的外设的工作频率（串口用到了定时器，就还会受影响）。
- **对外输出时钟**：使能之后，P07脚就会输出时钟。
- **定时时间/计数数量**：在定时模式下，这个参数决定了定时时间；在计数模式下，这个参数决定了计数的个数。
- **定时器4的中断使能**：使能之后，定时器4溢出就触发中断。

调用例程

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  void main(){//main函数，必须的。
3      system_init();//系统初始化函数，也是必须的。
4      timer_init();//初始化定时器。
5      while(1){
6
7      }
8  }
```

注意事项

1. 本函数将会初始化所有使能的定时器。

timer_set_timer_mode

函数原型：void timer_set_timer_mode(u8 id,u16 us);

描述

定时器设置定时模式函数。

输入

- id：要设置为定时模式的定时器编号，从0开始。
- us：定时的时间，范围参考下面的说明。

输出

无

返回值

无

定时时间预估

单片机主频 (MHz)	最小时间 (uS)	最大时间 (uS)
5.5296	2	65535
6.000	1	65535
11.0592	1	65535
12.000	1	65535
18.432	2	42666
20.000	1	39321
22.1184	1	35555
24.000	1	32768
27.000	1	29127
30.000	1	26214
33.000	1	23831
33.1776	1	23703
35.000	1	22469
36.864	1	21333
40.000	1	19660
44.2368	1	17777
45.000	1	17476

调用例程

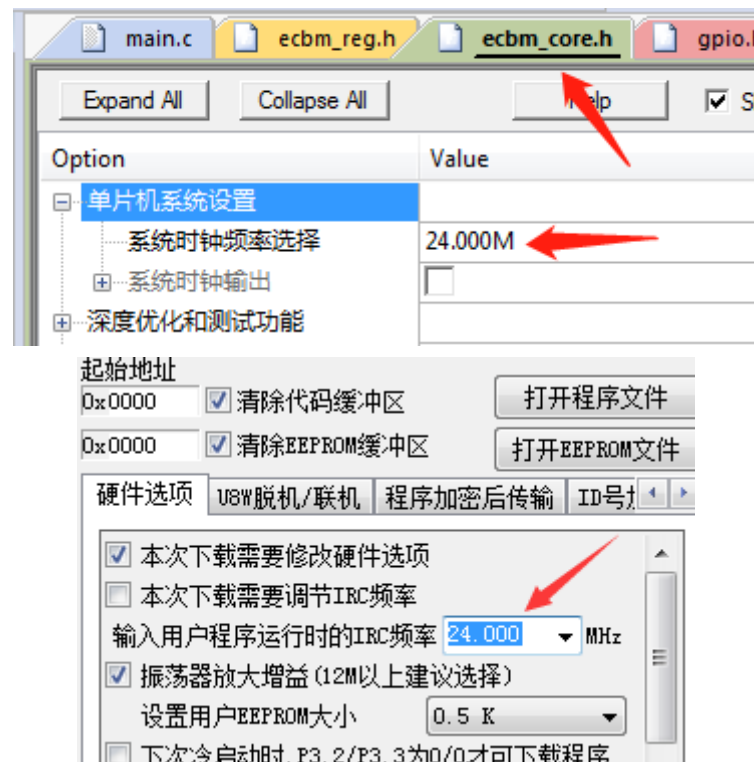
```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  void main(){//main函数，必须的。
3      system_init();//系统初始化函数，也是必须的。
4      timer_init();//初始化定时器。
5      timer_set_timer_mode(1,10000);//设置定时器1的定时时间为10000uS也就是10ms。
6      timer_start(1);//打开定时器1。
7      while(1){
8
9      }
10 }
11
```

注意事项

1. 运行本函数之后，会将指定的定时器的模式换成定时模式，同时根据参数计算出分频和初值。届时在图形化配置界面设置的这3个参数将会被本函数的计算值覆盖掉。

定时器1使能与设置	<input checked="" type="checkbox"/>
定时器开关	软件开关
计数来源	对系统时钟计数（定时器应用）
工作模式	16位自动重载模式
时钟分频	系统时钟不分频
对外输出时钟	<input type="checkbox"/>
定时时间/计数数量	65535
定时器1的中断使能	<input type="checkbox"/>

2. 基于上一点，当你发现定时器的运行效果和你在图形化配置界面设置的效果相差甚远的时候，可以找找是否在某处调用了本函数。
3. 本函数的计算依赖于系统主频的设置，部分时间会有少许计算误差，但如果定时严重不准的话，请注意这两处的设置是否一致：



timer_set_value

函数原型：void timer_set_value(u8 id,u16 value);

描述

定时器设定计数值函数。

输入

- id：要设定计数值的定时器编号，从0开始。
- value：设置的计数值，范围是0~65535。

输出

无

返回值

无

调用例程

```
1 timer_init();//初始化定时器。
2 timer_set_value(0,65530);//设置定时器0的计数值。
3 timer_start(0);//打开定时器0。
```

注意事项

1. 51单片机的定时器是从初值向上计数，计满65536溢出重载初值，所以你想计数20个的时候，需要给 $65536-20=65516$ 的初值。

timer_get_value

函数原型：u16 timer_get_value(u8 id);

描述

定时器计数值获取函数。

输入

- id：要获取计数值的定时器编号，从0开始。

输出

无

返回值

- 指定编号的定时器计数值。

调用例程

```
1 u16 val;//定义的变量。
2 timer_stop(0);//先停止定时器0。
3 val=timer_get_value(0);//读取定时器0的计数值。
4 timer_start(0);//重新打开定时器0。
```

注意事项

1. 最好在读取之前把定时器关闭，以防在读取的过程中，计数值又发生了改变。

回调函数

在串口库那一章里有提到过回调函数的执行原理。目前为了统一管理中断，ECBM库已经定义好了中断处理函数，并开放了回调函数供大家使用。只要按着回调函数的名字定义一个一模一样的函数就行。

- timer0_it_callback
- timer1_it_callback
- timer2_it_callback
- timer3_it_callback

- timer4_it_callback

调用例程

```
1 void timer0_it_callback(void){//这是定时器0的中断处理函数。
2     LED=!LED;    //当定时器0中断时，取反LED的亮灭状态。
3 }
```

注意事项

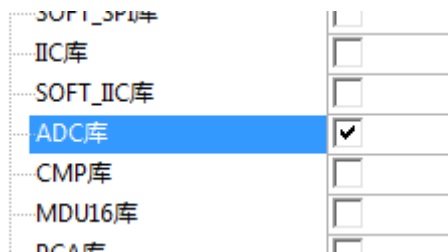
1. 回调函数的名字一定得正确，因为中断里面会调用这些回调函数，如果名字不正确就调用不了。
2. 本函数不需要也不能在其他地方调用！只需要定义了即可。因为在定时器中断触发的时候，单片机的硬件会自动去调用的。

优化建议

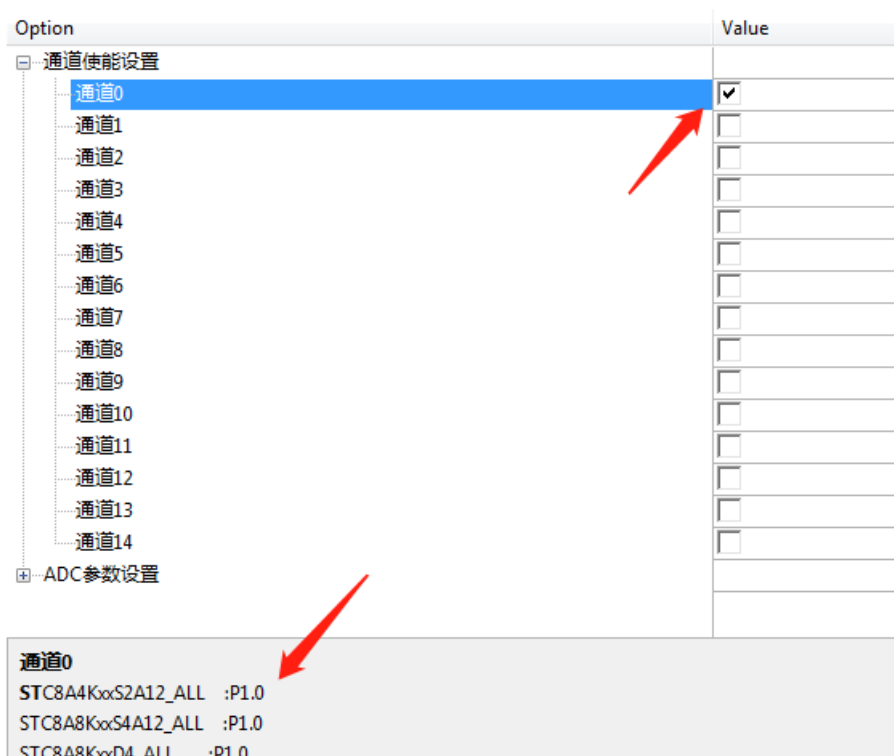
用不到的定时器就不去开使能，如果定时时间固定，可以删掉timer_set_timer_mode来节约空间。

ADC库

ADC就是Analog to Digital Converter，是一种能把模拟的电压信号转换成数字信号的器件。ADC库就是关于单片机模数转换的操作库。在使用本库之前先到ecbm_core.h里使能。双击打开ecbm_core.h文件，然后进入图形化配置界面，使能ADC库。



ADC是单片机获取环境模拟量的重要输入外设，通常都会有多个通道。所以在使用之前还需要在adc.h的图形配置界面使能所需要的通道才能正常读取。



如图，在下面的说明里还能看到该通道在不同型号单片机下对应的引脚。

API

adc_init

函数原型：void adc_init(void);

描述

ADC初始化函数。

输入

无

输出

无

返回值

无

参数配置

在图形化配置界面设置的参数，将会在执行本函数的时候写到ADC寄存器中。因此要保证选项选择正确无误。

Option	Value
<input checked="" type="checkbox"/> 通道使能设置	
<input checked="" type="checkbox"/> ADC参数设置	
ADC的分频系数	15
ADC的对齐方式	右对齐
舍弃低位数据	<input type="checkbox"/>
ADC中断	<input type="checkbox"/>
<input checked="" type="checkbox"/> ADC扩展功能	

设置说明如下：

- ADC的分频系数：这个参数决定了ADC的转换速度，但是在应用中发现分频数在6以下的时候，数据会跳动得比较厉害。因此推荐输入7~15。
- ADC的对齐方式：在大于8位小于16位的ADC中，会需要两个寄存器来存放转换好的AD值。不足16位的部分将会补0。以12位为例，若是左对齐，则低4位全为0。若是右对齐，则高4位全为0。他们的效果如下图所示。

寄存器		ADC_RES								ADC_RESL								对应数值
左对齐	分布	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0	0	
	最小值	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	最大值	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	65520
	步进值	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	16
右对齐	分布	0	0	0	0	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
	最小值	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	最大值	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	4095
	步进值	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

- 舍弃低位数据：从上面的设置中，可以看出右对齐才符合我们正常的使用习惯。那么左对齐的用法我猜测是用来滤波的。因为在正常情况下，ADC的数值跳动都集中在低几位中。如果设置了左对齐再舍弃掉ADC_RESL寄存器的值，那么跳动的那几位数据就会被舍弃掉，于是剩下比较平稳的高8位数据。

- ADC中断：使能之后将会打开ADC的中断使能，但是不推荐使用ADC中断。
- ADC扩展功能：目前还没有正式测试，不推荐使用这部分功能。因为不是每一个型号都有这个，有这功能的型号还缺货。

调用例程

```
1 #include "ecbm_core.h"//加载库函数的头文件。
2 void main(){//main函数，必须的。
3     system_init();//系统初始化函数，也是必须的。
4     adc_init();//初始化ADC
5     while(1){
6
7     }
8 }
```

注意事项

1. 本函数将将ADC已使能的通道的对应引脚设置为高阻态。

adc_read

函数原型：u16 adc_read(u8 ch);

描述

读取AD值函数。

输入

- ch：要读取AD值的通道编号，从0开始。

输出

无

返回值

- 该通道的AD值。

调用例程

串口获取单通道：

```
1 #include "ecbm_core.h"//加载库函数的头文件。
2 void main(){//main函数，必须的。
3     system_init();//系统初始化函数，也是必须的。
4     adc_init();//先初始ADC。
5     while(1){
6         delay_ms(1000);//每秒发送一次AD值到串口。
7         debug("%u\r\n",adc_read(0));//发送通道0的AD值。
8     }
9 }
```

串口获取多通道：


```

1  #include "ecbm_core.h"//加载库函数的头文件。
2  void main(){//main函数，必须的。
3      system_init();//系统初始化函数，也是必须的。
4      adc_init();//先初始ADC。
5      while(1){
6          delay_ms(1000);//每秒发送一次AD值到串口。
7          debug("[0]=%u\r\n",adc_read(0));//发送通道0的AD值。
8          debug("[2]=%u\r\n",adc_read(2));//发送通道2的AD值。
9      }
10 }

```

注意事项

1. 使用本函数之前，要先初始化ADC。
2. 本函数用到哪个通道，就得先在adc.h里使能哪个通道。否则会读取失败。

adc_voltage

函数原型：float adc_voltage(u8 ch,float vref);

描述

读取电压函数。

输入

- ch：要读取的通道编号，从0开始。
- vref：ADC的Vref引脚电压，单位为伏。

输出

无

返回值

- 该通道的电压值，单位为伏。

调用例程

```

1  #include "ecbm_core.h"//加载库函数的头文件。
2  void main(){//main函数，必须的。
3      system_init();//系统初始化函数，也是必须的。
4      adc_init();//先初始ADC。
5      while(1){
6          delay_ms(1000);//每秒发送一次AD值到串口。
7          debug("%f\r\n",adc_voltage(0,3.10f));//发送通道0的电压值，3.10为vref引脚
            的实测电压值，即3.10v。
8      }
9  }

```

注意事项

无

adc_read_vref

函数原型：float adc_read_vref(void);

描述

ADC读取Vref函数。

输入

无

输出

无

返回值

- 单片机Vref的电压值。

小科普

很多人被“基准电压”这个词糊弄到了，就觉得STC内部的1.19V基准电压是Vref，其实这是错误的想法。下面针对几种主流认知——说明：

- “这个1.19V电压不随着VCC的变化而改变，且可以用来计算AD值，那么1.19V的作用和Vref一样。”这个观点主要是不理解Vref的作用，认为Vref只是一个用来校准电压的东西。实际上Vref的用处不止是校准电压，它还决定了ADC能测量的最大电压。我们可以把ADC想象成一个大型的比较器，被测电压进来先和Vref的256分之1（假设是8位ADC）比较，如果被测电压小于Vref的256分之1就返回0，于是AD值就是0；如果被测电压大于等于Vref的256分之1且小于Vref的256分之2就返回1，于是AD值就是1。如果被测电压大于Vref的256分之2，就按这个规律一直比较下去直到比较出结果。所以说如果1.19V是Vref的话，那基本1.19V以上的电压都测不到了。
- “官方手册都没提到Vref，和电压基准有关的就只有1.19V了，不是它还会是谁？”有这种观点的人，估计深受官方广告手册的毒害。直到STC8的手册出来之前，STC的手册基本和广告传单差不多。所以有些人就没理解STC单片机的结构。首先重要的一点就是ADC一定需要Vref的，但是STC为了简化引脚，会在单片机内部把Vref和AVCC都连接到VCC上。于是很多人就没见过有Vref的存在。不过现在STC8也有了把AVCC和Vref都引出引脚的型号了，比如STC8A8K64S4A12。
- “1.19V不是Vref，那搞这个1.19V多此一举干嘛？”有些网友充分理解Vref和1.19V的区别后就会有这个问题。从型号规划上来看，不是所有型号都有ADC，但是所有型号都有1.19V。再联想手册了有提到单片机内置有LDO，所以我断定这个1.19V主要是为了LDO服务的。连到ADC的15通道只是为了提供便利，可以为客户省下一个TL431芯片。

调用例程

```

1  #include "ecbm_core.h"//加载库函数的头文件。
2  float vref;
3  void main(){//main函数，必须的。
4      system_init();//系统初始化函数，也是必须的。
5      adc_init();//先初始ADC。
6      vref=adc_read_vref();//读取vref的电压值。
7      while(1){
8          delay_ms(1000);//每秒发送一次AD值到串口。
9          debug("%F\r\n",adc_voltage(0,vref));//发送通道0的电压值。
10     }
11 }

```

注意事项

1. 在使用本函数之前先初始化ADC，否则一定得不到正确的值。

adc_it_start

函数原型：void adc_it_start(void);

描述

开启ADC中断函数。

输入

无

输出

无

返回值

无

调用例程

```

1  if(key_flag==0){//如果按键按下，
2      adc_it_stop();//先关闭ADC中断。
3      ...//其他代码。
4      adc_it_start();//再打开ADC中断。
5  }

```

注意事项

1. 要使用ADC的中断，必须先先在adc.h的图形化配置界面使能ADC中断。
2. adc_init函数里会打开ADC中断，因此本函数实际上要和adc_it_stop搭配一起使用的。也就是说假如ADC中断没有被关闭过，就没必要再用这个函数，因为中断一直会开启着。

adc_it_stop

函数原型：void adc_it_stop(void);

描述

关闭ADC中断函数。

输入

无

输出

无

返回值

无

调用例程

```
1  if(key_flag==0){//如果按键按下，
2      adc_it_stop();//先关闭ADC中断。
3      ...//其他代码。
4      adc_it_start();//再打开ADC中断。
5  }
```

注意事项

1. 要使用ADC的中断，必须先要在adc.h的图形化配置界面使能ADC中断。
2. 本函数执行后会关闭ADC中断，使用adc_start函数可以再度打开ADC中断。

adc_read_start

函数原型：void adc_read_start(u8 ch);

描述

ADC转换开始函数。

输入

- ch：要读取AD值的通道号，从0开始。

输出

无

返回值

无

调用例程

```
1 | adc_read_start(0); //准备读通道0的值。
```

注意事项

1. 本函数仅仅是开始一次转换，还不能马上得到AD值，当AD转换结束时会触发中断，在中断里才能读到本次测量的AD值。

adc_read_it和中断处理函数

函数原型：u16 adc_read_it(void);

描述

ADC读取AD值函数。

输入

无

输出

无

返回值

- 触发本次中断的通道AD值。

调用例程

基本ADC

```
1 | u16 adc_value;  
2 | ...//其他代码。  
3 | adc_read_start(0); //准备读通道0的值。  
4 | ...//其他代码。  
5 | void fun1(void) ADC_IT_NUM{ //这是ADC的中断处理函数。  
6 |     adc_value=adc_read_it(); //上次执行adc_read_start是读取通道0，所以这里读取到的是  
   |     通道0的AD值。  
7 | }
```

注意事项

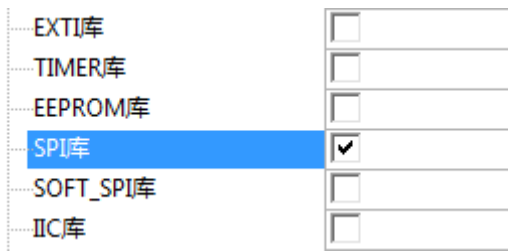
1. ADC中断只由AD转换完成标志位触发，也就是说adc_read_it函数经常会伴随着ADC中断处理函数一起出现。且adc_read_it只能放在中断处理函数中使用。两者是挂钩的。
2. 假如有多个通道的adc_read_start函数执行，那么可能会引发多次中断，顺序是adc_read_start函数执行的通道顺序。
3. 如果你在读这段话的时候感觉头晕，那请不要用中断法来读取AD值。直接用adc_read函数就行，简单快捷。我也觉得中断法没多大用处。

优化建议

去掉所有中断法，只使用查询法获取AD值。

SPI库

SPI是串行外设接口（Serial Peripheral Interface）的缩写，SPI库就是关于单片机的SPI的操作库。在使用本库之前先到ecbm_core.h里使能。双击打开ecbm_core.h文件，然后进入图形化配置界面，使能SPI库。



API

spi_init

函数原型：void spi_init(void);

描述

SPI初始化函数。

输入

无

输出

无

返回值

无

参数配置

本函数初始化的参数都是由图形化配置界面来设置，双击打开spi.h文件，进入图形化配置界面。

Option	Value
主/从机	主机
SS引脚使能	使能SS脚
数据收发顺序	先收/发数据的高位（MSB）
SPI时钟	SYSCLK/4
SPI时钟极性控制	SCLK空闲时为低电平
SPI时钟相位控制	在时钟变化的第一个边沿
SPI输出管脚	SS-P12 MOSI-P13 MISO-P14 SCLK-P15(全系列，除STC8G的8脚和STC8H带U或T后缀以外)

设置说明如下：

- 主/从机：用于设置SPI的是主机模式还是从机模式。目前的库只有主机发送接收函数，从机的数据处理需要自己实现了。
- SS引脚使能：这个就是SPI从机模式片选脚，如果使能的话，当单片机的该脚被拉低时，无论之前设置的是主机还是从机，都会被强制设置成从机。因此如果只当主机的话，就不要使能该功能。
- 数据收发顺序：可以先发高位或者先发低位。比如要发送0x29（二进制为0010 1001），先发高位就是按照00101001的顺序发，先发低位就是按照10010100的顺序发。选择哪个需要参考目标器件

的数据手册。

- SPI时钟：通过系统时钟分频得到，分频数越小，SPI的时钟越快。但是STC8的引脚输出速度也不算太快，所以SPI的时钟也不是越快就越好，太快的话IO速度反应不过来，输出的数据就会出错。
- SPI时钟极性控制：原文标准化的描述不容易理解，我翻译一下，所谓极性就是在不通信的时候，SCLK脚为高电平还是低电平。
- SPI时钟相位控制：相位这个词一出来，说不定就会想到数学里的波形相位。但是SPI的相位没那么复杂。下面的小科普会着重说明相位和极性的关系。
- SPI输出管脚：按照选项内容和实际需求选择即可。注意选项括号里的提示，有些型号的引脚可能会不同。

小科普

在一般的SPI手册里，都会把两个极性和两个相位搭配的四种情况的时序图列出来。别说是新手了，我在会用SPI之后回来总结成库都会晕头转向的。接下来教大家一个方法：

- 先看器件是什么边沿驱动的。对于同步传输的协议来说，一定是靠时钟脚的边沿触发数据的发送或者接收，所以第一步就是看驱动的边沿是上升沿还是下降沿。好好对比下面两张图：一般而言，数据变化是需要一定时间稳定，所以就看数据的中间对应着是上升沿还是下降沿。所以不难看出图一就是上升沿，图二就是下降沿。

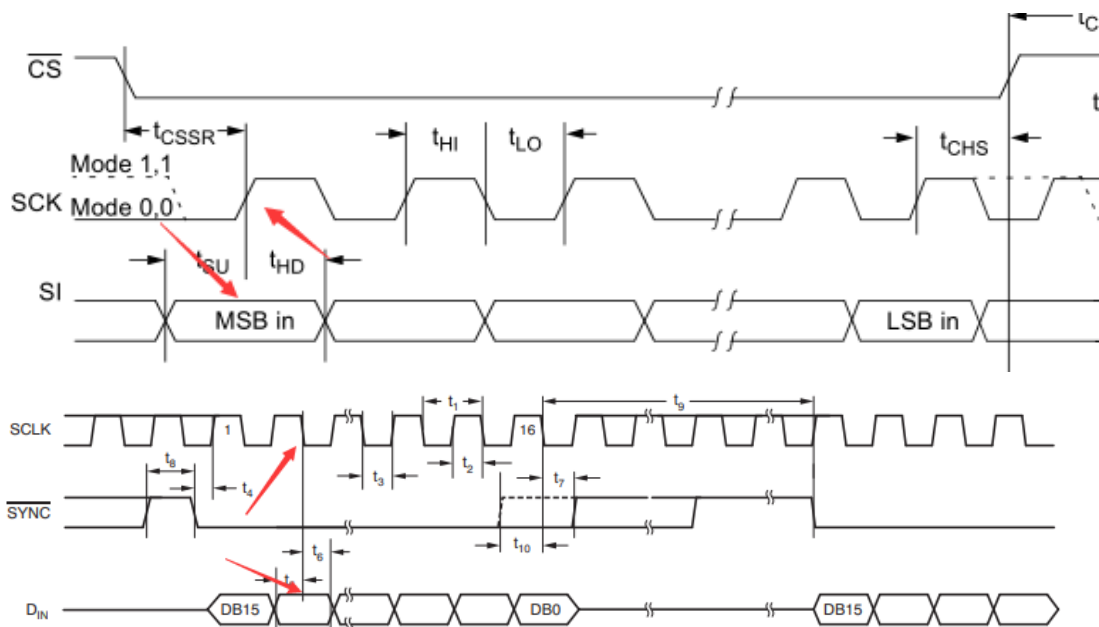


Figure 1. Serial Write Operation

- 然后就是选定时钟线的闲时电平，就是在不通信的时候，IO口保持的电平值。这个主要看器件手册的要求，但如果没有特殊要求，那么随便选一个就行。于是现在有两个属性是确定下来了：一个是触发边沿，一个是闲时电平。
- 基于你对闲时电平的选择，就能立马对选项【SPI时钟极性控制】做出选择！
- 到这里，应该就快理解相位了吧。假如你选择了闲时电平为低电平。而器件要求的触发边沿为下降沿，那么好好想一想：一个低电平的后面是不可能下降沿的对吧，因为下降沿的定义就是高电平跳到低电平的那个瞬间。于是乎为了触发数据传送，就必须先拉高时钟线（这是空闲状态到工作状态的第一个边沿）然后再拉低产生下降沿（这是空闲状态到工作状态的第二个边沿）。看看括号里的提示，现在应该明白选项【SPI时钟相位控制】的意思了吧，这里就应该选择“在时钟变化的第二个边沿”。同理，如果闲时电平为高电平，器件要求下降沿触发。那么从空闲状态转到工作状态时，时钟线可以立马拉低产生下降沿，所以就选择“在时钟变化的第一个边沿”。

调用例程

```
1 #include "ecbm_core.h"//加载库函数的头文件。
2 void main(){//main函数，必须的。
3     system_init();//系统初始化函数。
4     spi_init();//初始化spi。
5     while(1){
6
7     }
8 }
```

注意事项

1. SPI的原理简单，但是细节很多，最好要对照手册确认模式都选择正确。

spi_set_pin

函数原型：void spi_set_pin(u8 group);

描述

SPI的引脚设置函数。

输入

- group：引脚所在的分组。

输出

无

返回值

无

分组定义

基本宏定义的名字就说明了SPI将会用到哪些IO了：

- SPI_PIN_P12_P13_P14_P15。
- SPI_PIN_P22_P23_P24_P25。
- SPI_PIN_P74_P75_P76_P77。
- SPI_PIN_P35_P34_P33_P32。
- SPI_PIN_P54_P40_P41_P43。
- SPI_PIN_P55_P54_P33_P32。
- SPI_PIN_P54_P13_P14_P15。

在调用之前，请确认当前的型号确实有这些脚。前期确认一遍，不会耽误太多时间。

调用例程

```
1 if(run_mode==1){//当运行模式为1的时候，
2     spi_set_pin(SPI_PIN_P12_P13_P14_P15);//控制P1连接的SPI器件。
3 }else{//在其他模式下，
4     spi_set_pin(SPI_PIN_P22_P23_P24_P25);//控制P2连接的SPI器件。
5 }
```


注意事项

无

spi_send

函数原型：u8 spi_send(u8 dat);

描述

SPI发送接收函数。

输入

- dat：要发送的数据。

输出

无

返回值

- 接收到的数据

调用例程

既发送也接收：

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  u8 dat_in,dat_out;//两个缓存。
3  void main(){//main函数，必须的。
4      system_init();//系统初始化函数。
5      spi_init();//初始化spi。
6      while(1){
7          if(RI){//当接收到串口信息时。
8              RI=0;//清除接收标志位。
9              dat_in=SBUF;//把串口收到的数据保存下来。
10             dat_out=spi_send(dat_in);//发送该数据，同时接收SPI返回的数据。
11             SBUF=dat_out;//将SPI返回的数据发送到串口。
12         }
13     }
14 }
```

只发送：

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  void main(){//main函数，必须的。
3      system_init();//系统初始化函数。
4      spi_init();//初始化spi。
5      while(1){
6          spi_send(0x55);//发送该数据0x55。
7          delay_ms(500);//每隔500ms发送一次。
8      }
9  }
```

只接收：

```

1  #include "ecbm_core.h"//加载库函数的头文件。
2  u8 dat_out;//缓存
3  void main(){//main函数，必须的。
4      system_init();//系统初始化函数。
5      spi_init();//初始化spi。
6      while(1){
7          dat_out=spi_send(0xFF);//接收数据。
8          delay_ms(500);//每隔500ms接收一次。
9      }
10 }

```

注意事项

1. SPI协议就是有发送有接收，且发送和接收都是同时发生的。因此在只接收的情况下，也必须发个0xFF才能接收到数据。

优化建议

本库比较简单，只有3个函数，所以没有可优化的地方。

SOFT_SPI库

SOFT_SPI库就是用软件实现SPI通讯的操作库。速度上比硬件SPI慢，但引脚分布比硬件SPI自由，同时还能支持多个SPI通道。在使用本库之前先到ecbm_core.h里使能。双击打开ecbm_core.h文件，然后进入图形化配置界面，使能SOFT_SPI库。



API

soft_spi_init

函数原型：void soft_spi_init(soft_spi_def * dev,u8 clk,u8 mosi,u8 miso,u8 mode);

描述

软件SPI初始化函数。

输入

- clk：软件SPI的时钟脚。
- mosi：软件SPI的数据输出脚。
- miso：软件SPI的数据输入脚。
- mode：软件SPI的工作模式。

输出

- dev：保存了以上输入参数的信息包。

返回值

无

参数配置

本函数初始化的参数都是由图形化配置界面来设置，双击打开spi.h文件，进入图形化配置界面。

Option	Value
MOSI脚使能	<input checked="" type="checkbox"/>
MISO脚使能	<input checked="" type="checkbox"/>
数据收发顺序	先收/发数据的高位（MSB）
SPI时钟极性控制	SCLK空闲时为低电平
SPI时钟相位控制	在时钟变化的第二个边沿

设置说明如下：

- MOSI脚使能：勾选之后，软件SPI可以对外发送数据。SPI主机必须要主动发送信号去读从机。而目前软件SPI只有主机模式，所以这个选择在现在必须勾选上。等以后出了从机模式，才能根据需要关闭该使能。
- MISO脚使能：勾选之后，软件SPI可以接收外界的数据。对于一些只收不发的SPI从机（比如OLED模块）来说，可以省掉。
- 数据收发顺序：和硬件SPI一致。
- SPI时钟极性控制：和硬件SPI一致。
- SPI时钟相位控制：和硬件SPI一致。

调用例程

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  soft_spi_def dev1;
3  void main(){//main函数，必须的。
4      system_init();//系统初始化函数。
5      soft_spi_init(&dev1,D10,D20,D21,Dxx,ECBM_SOFT_SPI_MODE);//初始化软件spi。
6      //时钟脚定义为P1.0。
7      //数据输出脚为P2.0。
8      //数据输入脚为P2.1。
9      //片选控制脚不需要，用Dxx代替。
10     //用图形化配置界面的信息来设置软件SPI的工作模式。
11     while(1){
12
13     }
14 }
```

注意事项

1. 在初始化前，先定义器件的信息包。
2. 没用到的引脚一律用Dxx表示。

soft_spi_send

函数原型：u8 soft_spi_send(u8 dat);

描述

软件SPI发送接收函数。

输入

- dat：要发送的数据。

输出

无

返回值

- 接收到的数据

调用例程

既发送也接收：

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  soft_spi_def dev1;
3  u8 dat_in,dat_out;//两个缓存。
4  void main(){//main函数，必须的。
5      system_init();//系统初始化函数。
6      soft_spi_init(&dev1,D10,D20,D21,ECBM_SOFT_SPI_MODE);//初始化软件spi。
7      //时钟脚定义为P1.0。
8      //数据输出脚为P2.0。
9      //数据输入脚为P2.1。
10     //用图形化配置界面的信息来设置软件SPI的工作模式。
11     while(1){
12         if(RI){//当接收到串口信息时。
13             RI=0;//清除接收标志位。
14             dat_in=SBUF;//把串口收到的数据保存下来。
15             dat_out=soft_spi_send(dat_in);//发送该数据，同时接收SPI返回的数据。
16             SBUF=dat_out;//将SPI返回的数据发送到串口。
17         }
18     }
19 }
```

只发送：

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  soft_spi_def dev1;
3  void main(){//main函数，必须的。
4      system_init();//系统初始化函数。
5      soft_spi_init(&dev1,D10,D20,D21,ECBM_SOFT_SPI_MODE);//初始化软件spi。
6      //时钟脚定义为P1.0。
7      //数据输出脚为P2.0。
8      //数据输入脚为P2.1。
9      //用图形化配置界面的信息来设置软件SPI的工作模式。
10     while(1){
```

```

11     soft_spi_send(0x55); //发送该数据0x55。
12     delay_ms(500); //每隔500ms发送一次。
13 }
14 }

```

只接收：

```

1  #include "ecbm_core.h" //加载库函数的头文件。
2  soft_spi_def dev1;
3  u8 dat_out; //缓存
4  void main() { //main函数，必须的。
5      system_init(); //系统初始化函数。
6      soft_spi_init(&dev1, D10, D20, D21, ECBM_SOFT_SPI_MODE); //初始化软件spi。
7      //时钟脚定义为P1.0。
8      //数据输出脚为P2.0。
9      //数据输入脚为P2.1。
10     //用图形化配置界面的信息来设置软件SPI的工作模式。
11     while(1) {
12         dat_out = soft_spi_send(0xFF); //接收数据。
13         delay_ms(500); //每隔500ms接收一次。
14     }
15 }

```

注意事项

1. SPI协议就是有发送有接收，且发送和接收都是同时发生的。因此在只接收的情况下，也必须要发个0xFF才能接收到数据。

soft_spi_set_pin

函数原型：void soft_spi_set_pin(soft_spi_def * dev);

描述

SPI的引脚设置函数。

输入

- dev：软件SPI的信息包。

输出

无

返回值

无

调用例程

```

1  #include "ecbm_core.h" //加载库函数的头文件。
2  soft_spi_def dev1, dev2;
3  void main() { //main函数，必须的。
4      system_init(); //系统初始化函数。
5      soft_spi_init(&dev1, D10, D20, D21, ECBM_SOFT_SPI_MODE); //初始化软件spi。
6      //时钟脚定义为P1.0。

```

```

7      //数据输出脚为P2.0。
8      //数据输入脚为P2.1。
9      //用图形化配置界面的信息来设置软件SPI的工作模式。
10     soft_spi_init(&dev2,D55,D35,D54,0x89); //初始化软件spi。
11     //时钟脚定义为P5.5。
12     //数据输出脚为P3.5。
13     //数据输入脚为P5.4。
14     //自定义的工作模式0x89，含义可参考soft_spi.h里的注释。
15     while(1){
16         soft_spi_set_pin(&dev1); //切换到dev1。
17         soft_spi_send(0x01); //往dev1发送0x01。
18         soft_spi_send(0x11); //再往dev1发送0x11。
19         soft_spi_set_pin(&dev2); //切换到dev2。
20         soft_spi_send(0x02); //往dev2发送0x02。
21     }
22 }

```

注意事项

1. 为了让软件SPI的发送接收函数在使用上和硬件SPI一模一样，soft_spi_send函数里只会对默认引脚进行操作。因此在定义了多个软件SPI器件的情况下，一定得靠soft_spi_set_pin函数来切换默认引脚值。

优化建议

本库比较简单，只有3个函数对用户开放，所以没有可优化的地方。那些标记“内联版”的函数都是给其他库调用的，虽然用户用不到，但是不能优化掉。

NVIC库

NVIC库就是内嵌向量中断控制器:Nested Vectored Interrupt Controller (NVIC)的操作库。ECBM所有的中断函数都在nvic.c文件中统一管理。鉴于51的中断系统也不是很复杂，所以目前NVIC库主要是提供中断优先级的设置函数。本库默认使能打开。可直接去nvic.h去设置优先级。

API

nvic_set_priority

函数原型：void nvic_set_priority(void);

描述

设置优先级函数，只有在各个.h文件中使能了中断才会设置其优先级。

输入

无

输出

无

返回值

无

参数配置

本函数初始化的参数都是由图形化配置界面来设置，双击打开nvic.h文件，进入图形化配置界面。

Option	Value
EXTI0	56
TIMER0	55
EXTI1	54
TIMER1	53
UART1	52
ADC	51
LVD	50
PCA	49
UART2	48
SPI	47
EXTI2	46
EXTI3	45
TIMER2	44
EXTI4	43
UART3	42
UART4	41
TIMER3	40
TIMER4	39
CMD	38

USB
权重值越大，优先级越高。

在该界面下，通过下拉框选择各个中断的优先级关系。注意这里显示的数值是权重值，值越大，优先级越高。

调用例程

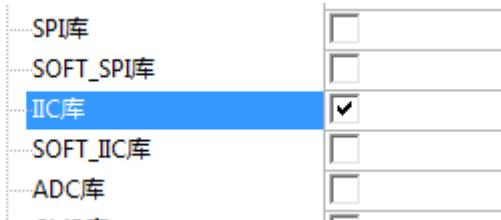
```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  void main(){//main函数，必须的。
3      system_init();//系统初始化函数。
4      nvic_set_priority();//设置中断优先级。
5      while(1){
6
7      }
8  }
```

注意事项

1. 有些中断的优先级是固定的，这是STC的硬件决定了的，是没办法调整的。
2. 有些中断号，KEIL是不支持的，需要提前打上中断号补丁。

IIC库

IIC是集成电路总线接口（Inter-Integrated Circuit）的缩写，IIC库就是关于单片机的IIC的操作库。在使用本库之前先到ecbm_core.h里使能。双击打开ecbm_core.h文件，然后进入图形化配置界面，使能IIC库。



API

iic_master_init

函数原型：void iic_master_init(void);

描述

IIC主机初始化函数。

输入

无

输出

无

返回值

无

参数配置

本函数初始化的参数都是由图形化配置界面来设置，双击打开iic.h文件，进入图形化配置界面。

Option	Value
IIC速度	100KHz
IIC默认管脚	SCL-P15 SDA-P14(全系列,除STC8G1K08和STC8G1K08A以外)
无响应超时时间	10

设置说明如下：

- IIC速度：有100KHz和400KHz可选。虽然可以设置成其他速度，但是这两个是比较常用的。
- IIC默认管脚：按照选项内容和实际需求选择即可。注意选项括号里的提示，有些型号的引脚可能会不同。
- 无响应超时时间：没有单位，数值越大，时间就越久。不过可以确定就是uS级别。

调用例程

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  void main(){//main函数，必须的。
3      system_init();//系统初始化函数。
4      iic_master_init();//初始化iic。
5      while(1){
6
7      }
8  }
```

注意事项

1. 目前还没有IIC从机的功能，只能做主机使用。

iic_set_pin

函数原型：void iic_set_pin(u8 group);

描述

IIC的引脚设置函数。

输入

- group：引脚所在的分组。

输出

无

返回值

无

分组定义

基本宏定义的名字就说明了IIC将会用到哪些IO了：

- IIC_PIN_P32_P33
- IIC_PIN_P54_P55
- IIC_PIN_P15_P14
- IIC_PIN_P25_P24
- IIC_PIN_P77_P76

在调用之前，请确认当前的型号确实有这些脚。前期确认一遍，不会耽误太多时间。

调用例程

```
1  if(run_mode==1){//当运行模式为1的时候，
2      iic_set_pin(IIC_PIN_P32_P33);//控制P3连接的IIC器件。
3  }else{//在其他模式下，
4      iic_set_pin(IIC_PIN_P54_P55);//控制P5连接的IIC器件。
5  }
```

注意事项

1. 在执行本函数的时候会自动设置相应的管脚为开漏输出，并使能内部上拉电阻。
2. iic_master_init函数已经在内部调用了本函数，如果不切换管脚的话，没必要使用本函数。

iic_reset_pin

函数原型：void iic_reset_pin(u8 group);

描述

IIC的还原引脚设置函数。

输入

- group：引脚所在的分组。

输出

无

返回值

无

分组定义

基本宏定义的名字就说明了IIC将会用到哪些IO了：

- IIC_PIN_P32_P33
- IIC_PIN_P54_P55
- IIC_PIN_P15_P14
- IIC_PIN_P25_P24
- IIC_PIN_P77_P76

在调用之前，请确认当前的型号确实有这些脚。前期确认一遍，不会耽误太多时间。

调用例程

```
1  if(run_mode==1){//当运行模式为1的时候，
2      iic_set_pin(IIC_PIN_P32_P33); //控制P3连接的IIC器件。
3      iic_reset_pin(IIC_PIN_P54_P55); //还原P54和P55。
4  }else{//在其他模式下，
5      iic_set_pin(IIC_PIN_P54_P55); //控制P5连接的IIC器件。
6      iic_reset_pin(IIC_PIN_P32_P33); //还原P32和P33。
7  }
```

注意事项

1. 在执行本函数的时候会自动设置相应的管脚为弱上拉，并关闭内部上拉电阻。
2. 一般在引脚复用的情况下，才需要调用这个函数来还原。否则可以一直保持开漏模式。

操作函数

函数原型：

- void iic_start(void);
- void iic_stop(void);
- void iic_write(u8 dat);
- void iic_write_ack(void);
- void iic_write_nack(void);
- u8 iic_read(void);
- bit iic_read_ack(void);

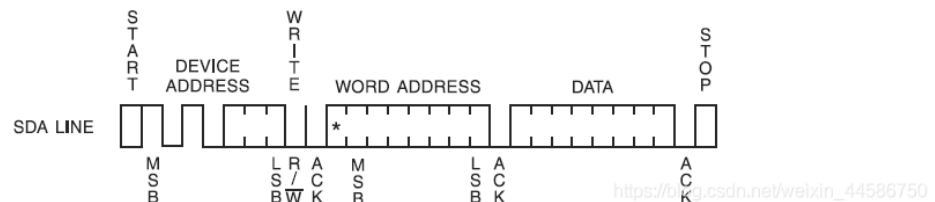
描述

IIC的操作相关函数，需要根据实际时序去调用。

调用例程

比如下图是AT24C02芯片的写一个字节的时序：

Figure 8. Byte Write



可以看出来时序动作是：【开始】【写器件地址和写入位】【等ACK】【写片内地址】【等ACK】【写数据】【等ACK】【结束】。所以可以得到如下代码。

```
1 void at24c02_write_byte(u8 addr,u8 dat){
2     iic_start();    //开始。
3     iic_write(0xA0); //写器件地址和写入位。
4     iic_read_ack(); //等从机ACK。
5     iic_write(addr); //写片内地址。
6     iic_read_ack(); //等从机ACK。
7     iic_write(dat); //写数据。
8     iic_read_ack(); //等从机ACK。
9     iic_stop();    //结束。
10 }
```

注意事项

1. 根据时序写就行，但目前STC的硬件IIC似乎有点问题，如果驱动不成功，可以换软件IIC试试。

优化建议

本库比较简单，所以没有可优化的地方。

SOFT_IIC库

SOFT_IIC库就是用代码模拟的IIC，优点是IO口可以随意安排，缺点是速度会慢一些。在使用本库之前先到ecbm_core.h里使能。双击打开ecbm_core.h文件，然后进入图形化配置界面，使能SOFT_IIC库。

SOFT_SPI库	<input type="checkbox"/>
IIC库	<input type="checkbox"/>
SOFT_IIC库	<input checked="" type="checkbox"/>
ADC库	<input type="checkbox"/>
CMP库	<input type="checkbox"/>

API

soft_iic_init

函数原型：void soft_iic_init(soft_iic_def * dev,u8 scl,u8 sda);

描述

软件IIC主机初始化函数。

输入

- scl：信号线对应的IO口。
- sda：数据线对应的IO口。

输出

- dev：软件IIC的器件信息包。

返回值

无

调用例程

```
1  #include "ecbm_core.h"//加载库函数的头文件。
2  soft_iic_def at24;    //定义一个软件IIC操作对象，名字随意，但是推荐和目标器件有联系，
    方便记忆。
3  void main(){//main函数，必须的。
4      system_init();//系统初始化函数。
5      soft_iic_init(&at24,D10,D11);//该对象所连接的引脚，先输入时钟脚SCL，再输入数据脚
    SDA。
6      while(1){
7
8      }
9  }
```

注意事项

1. 一定要先定义一个结构体实例，比如上面的at24。这个是接下来软件IIC操作的重要标识，先定义再执行本初始化函数。
2. 在没有利用中断的情况下，软件实现的IIC几乎没有办法做到实时从机。为了不占用宝贵的中断资源，以后也不会有软件IIC从机的开发计划。

soft_iic_set_pin

函数原型：void soft_iic_set_pin(soft_iic_def * dev);

描述

软件IIC引脚切换函数。在多器件的应用下，切换IIC函数的操作对象。

输入

- dev：切换的目标器件信息包。

输出

无

返回值

无

调用例程

```
1 soft_iic_set_pin(&sht30); //将IIC对象切换到名为SHT30的信息包。
2 val=get_temp();          //读取SHT30的温度数据。
3 soft_iic_set_pin(&at24c02); //将IIC对象切换到名为at24c02的信息包。
4 at24_write(0,val);        //将温度数据写入到eeprom（24c02）里。
```

注意事项

1. 软件IIC的操作函数都是对默认器件的操作，所以当需要操作多个器件时，一定要先切换。
2. 如果只操作一个器件，或者多个器件都是在一条IIC总线上的话，也不需要执行本函数，因为在初始化的时候就已经弄好了。

操作函数

函数原型：

- void soft_iic_start(void);
- void soft_iic_stop(void);
- void soft_iic_write(u8 dat);
- void soft_iic_write_ack(void);
- void soft_iic_write_nack(void);
- u8 soft_iic_read(void);
- bit soft_iic_read_ack(void);

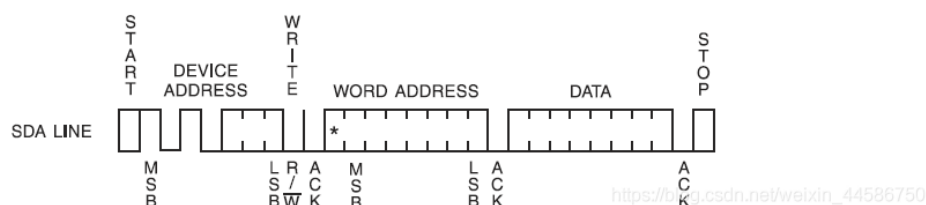
描述

软件IIC的操作相关函数，需要根据实际时序去调用。

调用例程

比如下图是AT24C02芯片的写一个字节的时序：

Figure 8. Byte Write



可以看出来时序动作是：【开始】【写器件地址和写入位】【等ACK】【写片内地址】【等ACK】【写数据】【等ACK】【结束】。所以可以得到如下代码。

```
1 void at24c02_write_byte(u8 addr,u8 dat){
2     soft_iic_start();    //开始。
3     soft_iic_write(0xA0); //写器件地址和写入位。
4     soft_iic_read_ack(); //等从机ACK。
5     soft_iic_write(addr); //写片内地址。
6     soft_iic_read_ack(); //等从机ACK。
7     soft_iic_write(dat);  //写数据。
8     soft_iic_read_ack(); //等从机ACK。
9     soft_iic_stop();      //结束。
10 }
```

注意事项

无

优化建议

本库比较简单，所以没有可优化的地方。

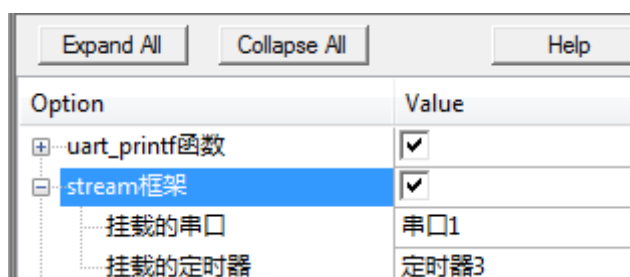
STREAM框架

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。

对于使用ECBM库的人来说，串口部分和定时器部分的处理代码已经做好。只需要在uart.h上使能stream框架再设置一下参数，调用几个程序就能很方便的处理串口数据。



如图所示，将stream挂载到串口1，那么接下来使用串口1进行通信，就可以使用stream里面的组件了。而这个挂载的定时器，是因为stream中有超时判断，需要一个定时器去跑时间。因此不要复用这里用到的定时器，否则stream就不能正常运行。

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。

Option	Value
队列缓存大小	135
数据帧间隔时间	20
串口空闲时间	80
组件使能与配置	
比较组件	<input type="checkbox"/>
+ FUR组件	<input type="checkbox"/>
+ MODBUS组件	<input type="checkbox"/>
X-MODEM组件	<input type="checkbox"/>
+ ECP组件	<input type="checkbox"/>

设置说明如下：

- 队列缓存大小：stream的原理，就是在串口接收那，只存接收到的数据，不做任何处理。因此这个缓存设置得越大，能保存的数据就越多，当然也要注意别超过了单片机的RAM大小。
- 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。
- 组件使能与配置：勾选对应组件就能使能该组件。如果组件还有参数需要设置，那么它的左侧会有+号，如图中的FUR组件、MODBUS组件和ECP组件。

API

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

```
1 #include "ecbm_core.h" //加载库函数的头文件。
2 void main(void){       //main函数，必须的。
3     system_init();     //系统初始化函数，也是必须的。
4     while(1){
5         ecbm_stream_main();//放到主循环不断执行。
6     }
7 }
```

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

```
1  #include "ecbm_core.h"    //加载库函数的头文件。
2  void main(void){          //main函数，必须的。
3      system_init();        //系统初始化函数，也是必须的。
4      while(1){
5          ecbm_stream_main();//放到主循环不断执行。
6      }
7  }
8  void ecbm_stream_exe(u8 dat){
9      //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。
10 }
```

注意事项

1. 本函数用于对接各个组件，具体的对接方法将会在组件的说明里放出。

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_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

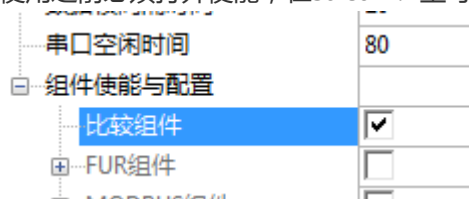
无

调用例程

```
1  #include "ecbm_core.h" //加载库函数的头文件。
2  void main(void){      //main函数，必须的。
3      system_init();    //系统初始化函数，也是必须的。
4      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。
5      while(1){
6          ecbm_stream_main();//跑stream框架。
7      }
8  }
9  u8 cmp_on=0,cmp_off=0;//存比较值的变量。
10 void ecbm_stream_exe(u8 dat){
11     ecbm_stream_strcmp(dat,"LED_ON" ,&cmp_on );//和字符串LED_ON比较。
12     ecbm_stream_strcmp(dat,"LED_OFF",&cmp_off);//和字符串LED_OFF比较。
13     if(cmp_on ==6){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。
14     if(cmp_off==7){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。
15 }
```

注意事项

1. 使用之前必须打开使能，在stream.h里可以看到使能设置：



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

简介

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位置一，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(void);

描述

FUR获取ID函数，这个函数需要返回本机的ID号。

输入

无

输出

无

返回值

- 本机的ID号。

调用例程

返回固定ID：

```
1  u8 es_fur_get_id(void){
2      return 1;
3  }
```

返回可变ID：

```
1  u8 mcu_id=1;
2  u8 es_fur_get_id(void){
3      return mcu_id;
4  }
```

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(u16 addr);

描述

FUR读取寄存器函数。

输入

- 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：

```
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。
2  u16 es_fur_read_reg(u16 addr){
3      return reg[addr]; //返回数组的值或者是其他变量。
4  }
```

写法2：

```
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。
2  u16 es_fur_read_reg(u16 addr){
3      if(addr==0){ //当读取地址是0的时候，
4          return (u16)(P0); //返回P0的电平值。
5      }else if(addr==1){ //当读取地址是1的时候，
6          return (u16)(P1); //返回P1的电平值。
7      }else if(addr==2){ //当读取地址是2的时候，
8          return (u16)(P2); //返回P2的电平值。
9      }else if(addr==3){ //当读取地址是3的时候，
10         return (u16)(P3); //返回P3的电平值。
11     }else{ //如果是其他地址，
12         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。
13     }
14 }
```

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(u16 addr,u16 dat);

描述

FUR写入寄存器函数。

输入

- 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：

```
1 void es_fur_write_reg(u16 addr,u16 dat){
2     reg[addr]=dat;//存入缓存
3 }
```

作为触发：

```
1 void es_fur_write_reg(u16 addr,u16 dat){
2     if(addr==0){//当地址为0时，
3         P0=(u8)(dat);//将dat值发到P0。
4     }else if(addr==1){//当地址为1时，
5         if(dat){//若dat不是0，
6             LED_ON;//点亮LED。
7         }else{//否则，
8             LED_OFF;//熄灭LED。
9         }
10    }
11 }
```

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(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

- 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

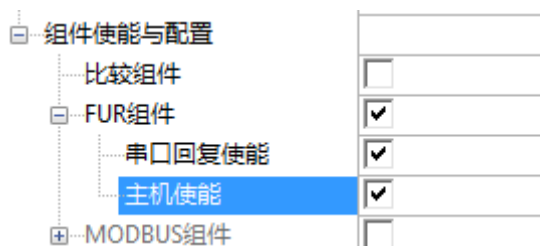
无

调用例程

```
1 //假如从机模块的15号寄存器里存着模块测量到的温度。
2 //定义回调函数
3 void es_fur_master_receive_callback(u16 addr,u16 dat){
4     if(addr==15){//当从机返回的数据是15号寄存器时，
5         oled_printf(&oled,0,0,"temp=%d",dat);//在OLED上打印出温度信息。
6     }
7 }
8 //在main里执行读取函数就行了。
9 void main(){
10     ...//其他代码
11     while(1){
12         if(temp_flag){//到了更新温度的时候了，
13             temp_flag=0;//先清零标志位。
14             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。
15         }
16     }
17 }
```

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

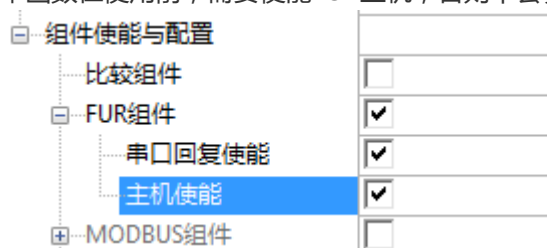
无

调用例程

```
1 void main(){
2     ...//其他代码
3     while(1){
4         ...//其他代码
5         if(key==0x25){//按下某个按键的时候。
6             es_fur_master_send(15,1,0);//向1号从机发送的15号寄存器赋值为0。
7         }
8     }
9 }
```

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

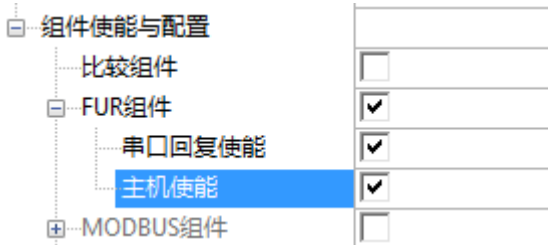
无

调用例程

```
1 //假如从机模块的15号寄存器里存着模块测量到的温度。
2 //定义回调函数
3 void es_fur_master_receive_callback(u16 addr,u16 dat){
4     if(addr==15){//当从机返回的数据是15号寄存器时，
5         oled_printf(&oled,0,0,"temp=%d",dat);//在OLED上打印出温度信息。
6     }
7 }
8 //在main里执行读取函数就行了。
9 void main(){
10     ...//其他代码
11     while(1){
12         if(temp_flag){//到了更新温度的时候了，
13             temp_flag=0;//先清零标志位。
14             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。
15         }
16     }
17 }
```

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_exe

函数原型：void es_fur_exe(u8 dat);

描述

FUR解析执行函数，放进stream框架里运行。

输入

- dat：和stream框架对接的参数。

输出

无

返回值

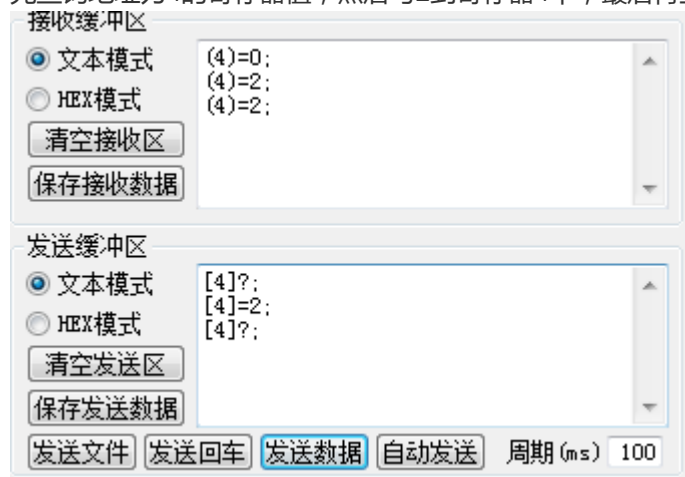
无

调用例程

标准最小化FUR从机代码一览：

```
1  #include "ecbm_core.h" //加载库函数的头文件。
2  void main(void){       //main函数，必须的。
3      system_init();     //系统初始化函数，也是必须的。
4      while(1){
5          ecbm_stream_main();
6      }
7  }
8  void ecbm_stream_exe(u8 dat){
9      dat=dat;
10     es_fur_exe(dat);
11 }
12 u8 es_fur_get_id(void){
13     return 1; //0是广播地址，所以不能是0。
14 }
15 u16 yanshi=0;
16 u16 es_fur_read_reg(u16 addr){
17     addr=addr; //演示代码而已。
18     return yanshi; //演示代码而已。
19 }
20 void es_fur_write_reg(u16 addr,u16 dat){
21     addr=addr; //演示代码而已。
22     yanshi=dat; //演示代码而已。
23 }
```

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。

注意事项

- 1. 如果只是从机的话，只要在ecbm_stream_exe里填入es_fur_exe（第10行），定义es_fur_get_id、es_fur_read_reg和es_fur_write_reg这3个函数就行了。
- 2. FUR支持只收不回，如果发现串口助手发送正确的指令后，单片机没有回复，除了检查代码是否正确之外，还要检查回复使能是否打开。



stream.h文件里。

- 3. 主机代码的演示在上面的es_fur_master_read和es_fur_master_send函数的说明里，往上翻一翻就看到了。

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。

API

es_modbus_rtu_get_id

函数原型：u8 es_modbus_rtu_get_id(void);

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

无

输出

无

返回值

- 本机的ID号。

调用例程

返回固定ID：

```
1  u8 es_modbus_rtu_get_id(void){
2      return 1;
3  }
```

返回可变ID：

```
1  u8 mcu_id=1;
2  u8 es_modbus_rtu_get_id(void){
3      return mcu_id;
4  }
```

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(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

- 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

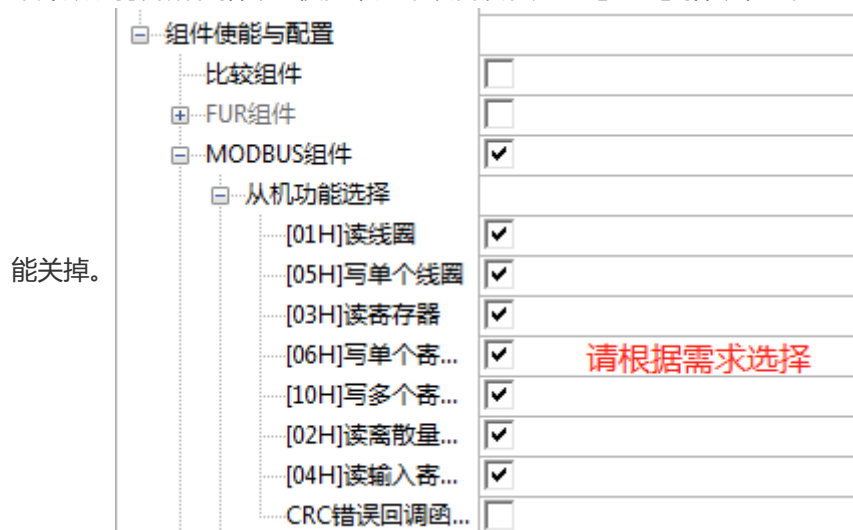
```

1 void es_modbus_cmd_read_io_bit(u16 addr,u8 * dat){
2     if(addr==0x0000){//当主机要读地址0的时候,
3         if(P00){//例程中, 这个地址对应着P0.0的状态。
4             *dat=1;//是高电平就返回1。
5         }else{
6             *dat=0;//是低电平就返回0。
7         }
8     }
9 }

```

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，需要你通过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



es_modbus_cmd_read_io_reg

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

描述

读取输入寄存器函数，功能码04H会用到。

输入

- 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

```
1 void es_modbus_cmd_read_io_reg(u16 addr,u16 * dat){
2     if(addr==0x0000){//当主机要读地址0的时候,
3         *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。
4     }
5 }
```

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，需要你用地at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



es_modbus_cmd_write_bit

函数原型：void es_modbus_cmd_write_bit(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

- 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

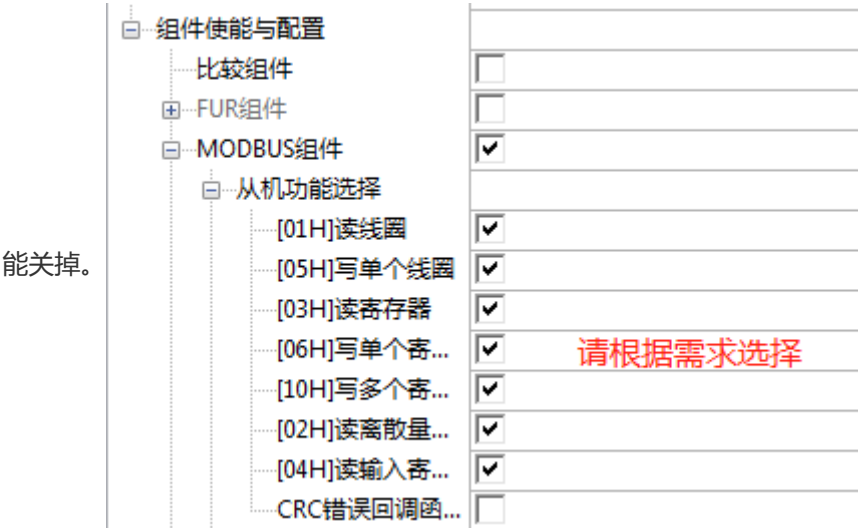
无

调用例程

```
1  u16 test=0x1234; //假设有个变量test，然后test的D0位映射到线圈0x0000号。
2  void es_modbus_cmd_write_bit(u16 addr,u8 dat){
3      if(addr==0x0000){ //当主机要写地址0的线圈时候，
4          if(dat){ //例程中，这个地址对应着变量test的D0位。
5              test|=0x0001; //如果写入的是1，就令D0位变成1。
6          }else{
7              test&=0xFFFE; //如果写入的是0，就令D0位变成0。
8          }
9      }
10 }
```

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



es_modbus_cmd_read_bit

函数原型：void es_modbus_cmd_read_bit(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

- 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

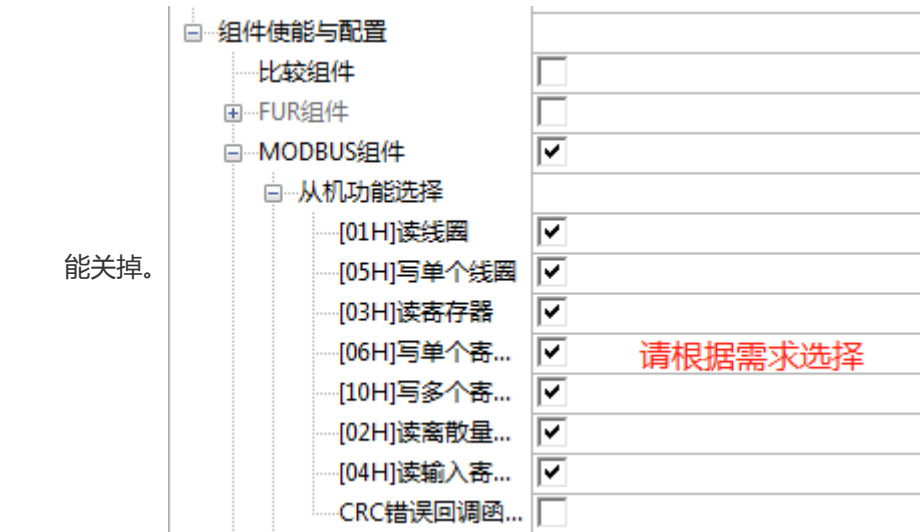
无

调用例程

```
1  u16 test=0x1234; //假设有个变量test，然后test的D0位映射到线圈0x0000号。
2  void es_modbus_cmd_read_bit(u16 addr,u8 * dat){
3      if(addr==0x0000){ //当主机要读地址0的时候，
4          if(test&0x0001){ //例程中，这个地址对应着test变量的D0位。
5              *dat=1; //D0位为1就返回1。
6          }else{
7              *dat=0; //D0位为0就返回0。
8          }
9      }
10 }
```

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，需要你使用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



es_modbus_cmd_write_reg

函数原型：void es_modbus_cmd_write_reg(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

- 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

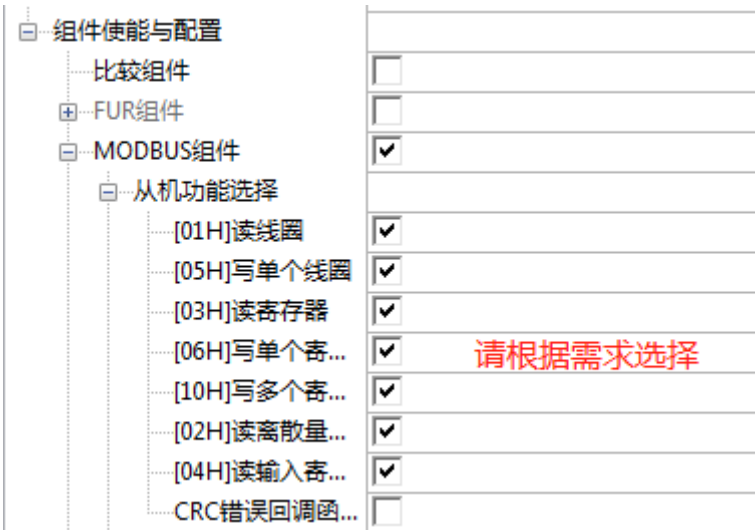
无

调用例程

```
1  u16 reg[20];
2  void es_modbus_cmd_write_reg(u16 addr,u16 dat){
3      reg[addr]=dat;//例程中的数组reg代表了保持寄存器。
4  }
```

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(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

- 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

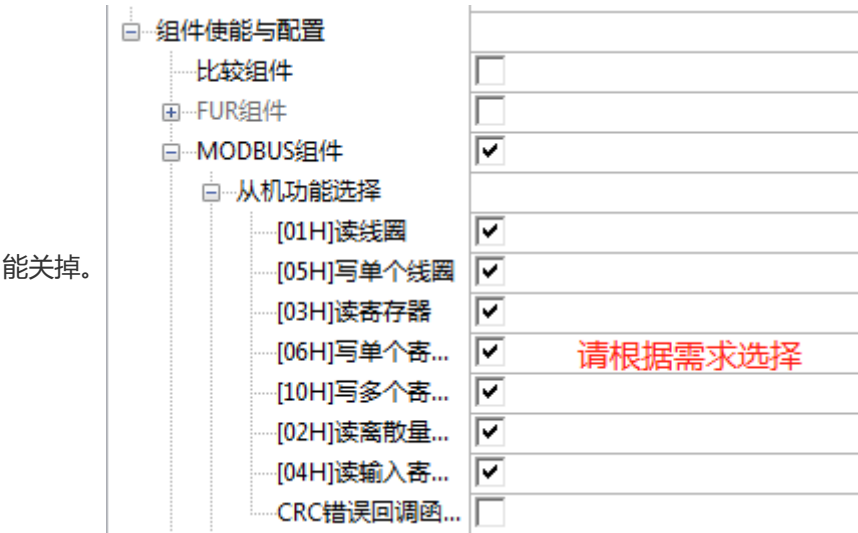
无

调用例程

```
1  u16 reg[20];
2  void es_modbus_cmd_read_reg(u16 addr,u16 * dat){
3      *dat=reg[addr];//例程中的数组reg代表了保持寄存器。
4  }
```

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，需要你用地at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



es_modbus_rtu_exe

函数原型：void es_modbus_rtu_exe(u8 dat);

描述

modbus_rtu解析执行函数，放进stream框架里运行。

输入

- dat：和stream框架对接的参数。

输出

无

返回值

无

调用例程

标准最小化modbus从机代码一览：

```
1  #include "ecbm_core.h" //加载库函数的头文件。
2  void main(void){ //main函数，必须的。
3      system_init(); //系统初始化函数，也是必须的。
4      while(1){
5          ecbm_stream_main();
6      }
7  }
8  void ecbm_stream_exe(u8 dat){
9      dat=dat;
10     es_modbus_rtu_exe(dat);
11 }
12 u8 es_modbus_rtu_get_id(void){
13     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。
14 }
15 u16 test=0;
16 void es_modbus_cmd_read_io_bit(u16 addr,u8 * dat){
17     if(addr==0x0000){ //当主机要读地址0的时候，
18         if(P00){ //例程中，这个地址对应着P0.0的状态。
19             *dat=1; //是高电平就返回1。
20         }else{
21             *dat=0; //是低电平就返回0。
22         }
23     }
24 }
25 void es_modbus_cmd_read_io_reg(u16 addr,u16 * dat){
26     if(addr==0x0000){ //当主机要读地址0的时候，
27         *dat=(u16)P0; //例程中，这个地址对应着P0的状态。
28     }
29 }
30 void es_modbus_cmd_write_bit(u16 addr,u8 dat){
31     if(addr==0x0000){ //当主机要写地址0的线圈时候，
32         if(dat){ //例程中，这个地址对应着变量test的D0位。
33             test|=0x0001; //如果写入的是1，就令D0位变成1。
34         }else{
35             test&=0xFFFE; //如果写入的是0，就令D0位变成0。
36         }
37     }
38 }
39 void es_modbus_cmd_read_bit(u16 addr,u8 * dat){
40     if(addr==0x0000){ //当主机要读地址0的时候，
41         if(test&0x0001){ //例程中，这个地址对应着test变量的D0位。
42             *dat=1; //D0位为1就返回1。
43         }else{
44             *dat=0; //D0位为0就返回0。
45         }
46     }
47 }
48 u16 reg[10]={0};
49 void es_modbus_cmd_write_reg(u16 addr,u16 dat){
50     reg[addr]=dat; //例程中的数组reg代表了所有保持寄存器。
51 }
```

```

52 void es_modbus_cmd_read_reg(u16 addr,u16 * dat){
53     *dat=reg[addr];//例程中的数组reg代表了所有保持寄存器。
54 }

```

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

```

1 void main(void){
2     ...//其他代码。
3     while(1){
4         ...//其他代码。
5         if(key==0x01){//假如按下了某个按键，
6             es_modbus_rtu_master_0x01(1,0,1);//向从机发送【01H】指令，此时modbus
组件会变成主机模式。
7         }
8     }
9 }
10 void es_modbus_rtu_master_0x01_callback(u16 addr,u8 dat){//当接收到从机回复的时
候。
11     if(addr==0){//判断线圈0，
12         if(dat){LED_1_ON;}else{LED_1_OFF;}//是1就点亮LED1，是0就熄灭LED1。下同。
13         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。
14     }
15 }

```

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 id,u16 addr,u16 len);

描述

主机01号功能码发送函数。

输入

- 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

```
1 es_modbus_rtu_master_0x01(1,0,3); //从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。

```
1 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。
2     if(addr==0){ //判断线圈0，
3         if(dat){LED_1_ON;}else{LED_1_OFF;} //是1就点亮LED1，是0就熄灭LED1。下同。
4     }
5     if(addr==1){
6         if(dat){LED_2_ON;}else{LED_2_OFF;}
7     }
8     if(addr==2){
9         if(dat){LED_3_ON;}else{LED_3_OFF;}
10    }
11 }
```

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。

MODBUS组件	<input checked="" type="checkbox"/>
+ 从机功能选择	
- 主机功能选择	<input checked="" type="checkbox"/>
[01H]读线圈	<input checked="" type="checkbox"/>
[05H]写单个线圈	<input checked="" type="checkbox"/>
[03H]读寄存器	<input checked="" type="checkbox"/>
[06H]写单个寄存器	<input checked="" type="checkbox"/>
[10H]写多个寄存器	<input checked="" type="checkbox"/>
[02H]读离散量输入	<input checked="" type="checkbox"/>
[04H]读输入寄存器	<input checked="" type="checkbox"/>
CRC错误回调函数	<input type="checkbox"/>

根据需求选择

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

```
1 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。

```

1 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。
2     if(addr==0){//判断线圈0,
3         if(dat){LED_1_ON;}else{LED_1_OFF;}//是1就点亮LED1, 是0就熄灭LED1。下同。
4     }
5     if(addr==1){
6         if(dat){LED_2_ON;}else{LED_2_OFF;}
7     }
8     if(addr==2){
9         if(dat){LED_3_ON;}else{LED_3_OFF;}
10    }
11 }

```

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。

MODBUS组件	<input checked="" type="checkbox"/>
+ 从机功能选择	
- 主机功能选择	<input checked="" type="checkbox"/>
[01H]读线圈	<input checked="" type="checkbox"/>
[05H]写单个线圈	<input checked="" type="checkbox"/>
[03H]读寄存器	<input checked="" type="checkbox"/>
[06H]写单个寄存器	<input checked="" type="checkbox"/> 根据需求选择
[10H]写多个寄存器	<input checked="" type="checkbox"/>
[02H]读离散量输入	<input checked="" type="checkbox"/>
[04H]读输入寄存器	<input checked="" type="checkbox"/>
CRC错误回调函数	<input type="checkbox"/>

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

```
1 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。

```
1 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。
2     if(addr==0){ //判断地址0，
3         oled_printf("[0]=%u",dat); //比如地址0需要显示到OLED。
4     }
5     if(addr==1){
6         if(dat>100){BEEP_ON;} //比如地址1的值大于100就报警。
7         if(dat<20){BEEP_OFF;} //小于20就恢复。
8     }
9     if(addr==2){
10        test=dat; //比如地址2的内容就存起来。
11    }
12 }
```

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

```
1 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

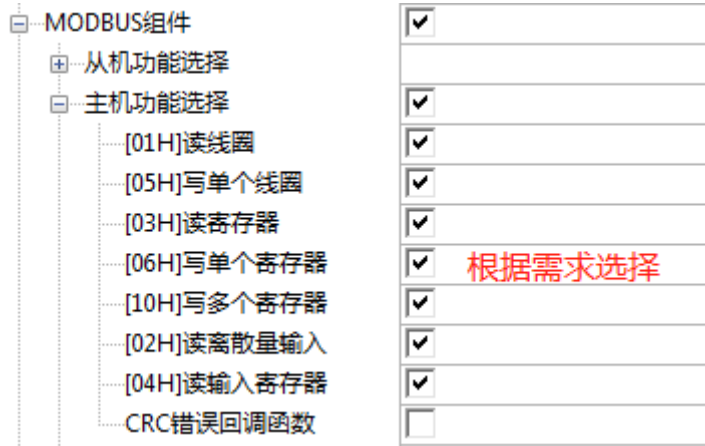
在调用上面的主机发送函数之后，会等待从机的返回。如果从机返回了数据，将会执行回调函数。因此还需要定义一个回调函数，原型为：void es_modbus_rtu_master_0x04_callback(u16 addr,u16 dat); 其中addr为从机返回的地址，dat为对应的数据。

```
1 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。
2     if(addr==0){ //判断地址0，
3         oled_printf("[0]=%u",dat); //比如地址0需要显示到OLED。
4     }
5     if(addr==1){
6         if(dat>100){BEEP_ON;} //比如地址1的值大于100就报警。
7         if(dat<20){BEEP_OFF;} //小于20就恢复。
8     }
9     if(addr==2){
10        test=dat; //比如地址2的内容就存起来。
11    }
12 }
```

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

```
1 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。

```
1 void es_modbus_rtu_master_0x05_callback(u16 addr,u8 dat){ //上面的例子里写了一个
  线圈值。
2     if(addr==0){ //返回是地址0，说明写入成功了。
3         oled_printf("写入成功"); //做个提示。
4     }
5 }
```

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

```
1 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。

```
1 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。
2     if(addr==0){ //返回是地址0，说明写入成功了。
3         oled_printf("写入成功"); //做个提示。
4     }
5 }
```

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

```
1 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。

```

1 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。
2     if(addr==0){//返回是地址0，说明写入成功了。
3         oled_printf("写入成功了%u字节",len);//做个提示。
4     }
5 }

```

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

```

1 void es_modbus_rtu_master_err_code_callback(u8 cmd,u8 err_code){
2     //可以重新和从机通信，也可以什么都不做。
3 }

```

注意事项

- 1. 参数cmd会告诉你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。



X-Modem组件

简介

XMODEM协议是一种使用拨号调制解调器的个人计算机通信中广泛使用的异步文件运输协议。这种协议以128字节块的形式传输数据，并且每个块都使用一个校验和过程来进行错误检测。在本库中实现的是上位机往单片机发送数据的方向，也可以说是从机部分吧。并且本库使用的是校验和而不是CRC。

Option	Value
队列缓存大小	135
数据帧间隔时间	20
串口空闲时间	80
组件使能与配置	
比较组件	<input type="checkbox"/>
FUR组件	<input type="checkbox"/>
MODBUS组件	<input type="checkbox"/>
X-MODEM组件	<input checked="" type="checkbox"/>
ECP组件	<input type="checkbox"/>

API

es_xmodem_write_reg

函数原型：void es_xmodem_write_reg(u8 pack,u8 offset,u8 buf);

描述

xmodem数据输入函数，这个函数会将接受到数据写入到单片机的某处。

输入

- pack：包编号，鉴于xmodem具备重发功能，需要通过包编号来判断是新数据还是重发的数据。
- offset：包内数据偏移，xmodem一个包有128字节。所以该值将会从0加到127。
- buf：当前偏移对应的数值。

输出

无

返回值

无

调用例程

```
1 void es_xmodem_write_reg(u8 pack,u8 offset,u8 buf){
2     //将包编号和包内偏移合并成一个地址量，即可把数据对接写入到eeprom。
3     eeprom_write((u16)(pack-1)*128+(u16)offset,buf); //包编号从1开始，地址计算却从
    0开始，所以要减一。
4 }
```

注意事项

1. 当上位机向单片机写数据的时候，就会调用本函数。
2. 本协议一帧有128字节的数据，再加上校验和序号之类的，起码要保证队列缓存大小在135以上。

在stream.h那里可以设置。

Option	Value
队列缓存大小	135
数据帧间隔时间	20

es_xmodem_start

函数原型：void es_xmodem_start(void);

描述

xmodem开始接收函数，调用后会通知上位机开始发送数据。

输入

无

输出

无

返回值

无

调用例程

```
1  if(key==0){//当按键按下时，
2      es_xmodem_start();//开始传输。
3  }
```

注意事项

1. 调用本函数时，会向上位机发起传输请求，然后上位机开始发送数据下来。

es_xmodem_start

函数原型：void es_xmodem_start(void);

描述

xmodem开始接收函数，调用后会通知上位机开始发送数据。

输入

无

输出

无

返回值

无

调用例程

```
1  if(key==0){//当按键按下时，
2      es_xmodem_start();//开始传输。
3  }
```

注意事项

1. 调用本函数时，会向上位机发起传输请求，然后上位机开始发送数据下来。

es_xmodem_exe

函数原型：void es_xmodem_exe(u8 dat);

描述

xmodem解析执行函数，放进stream框架里运行。

输入

- dat：和stream框架对接的参数。

输出

无

返回值

无

调用例程

标准最小化xmodem从机代码一览：

```
1  #include "ecbm_core.h" //加载库函数的头文件。
2  sbit key=P1^1; //使用实体按键来开始传输。
3  void main(void){ //main函数，必须的。
4      system_init(); //系统初始化函数，也是必须的。
5      eeprom_init(); //本例是上位机传输数据到单片机eeprom上。
6      while(1){
7          if(key==0){ //示例中不做按键消抖了。
8              es_xmodem_start(); //按下按键时，开始传输数据。
9          }
10         ecbm_stream_main();
11     }
12 }
13 void ecbm_stream_exe(u8 dat){
14     dat=dat;
15     es_xmodem_exe(dat);
16 }
17 void es_xmodem_write_reg(u8 pack,u8 offset,u8 buf){
18     //将包编号和包内偏移合并成一个地址量，即可把数据对接写入到eeprom。
19     eeprom_write((u16)(pack-1)*128+(u16)offset,buf); //包编号从1开始，地址计算却
    从0开始，所以要减一。
20 }
```

注意事项

1. 无

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”。

不过目前ECP还在设计和完善中。敬请期待。