ECBM库使用手册

欢迎阅读ECBM库的使用手册，本手册是一系列说明文件的总称。通常会按外设或者某些特性来分类，文件内会介绍他们的API和用法。可先通过文件名快速找到自己想看的外设或库，再打开查看里面的目录找到自己需要功能。

版本说明

由于使用手册是按外设或某些特性分类的，那么每个文件的版本可能就会有所不同。在理想的情况下，若某外设的库没有更新，那么说明文件就不会更新。比如GPIO库，在没有bug的情况，基本不会再修改了。

如果你发现某些库的说明文档没有更新，但代码那边确实有改动了，说明这段文档是鸽了，可以在QQ群里催一催。

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：

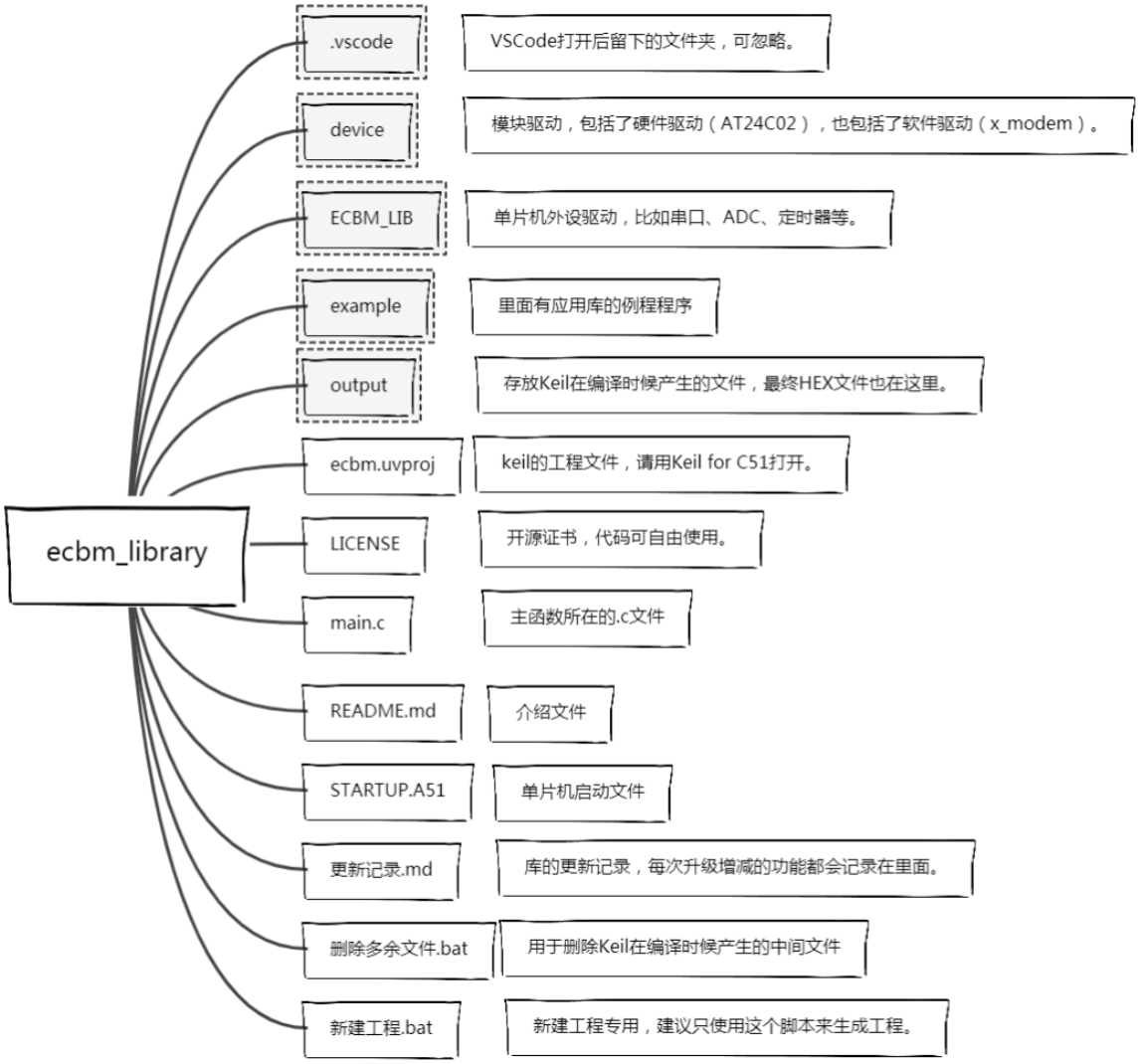
1. **保存历史代码。**程序员笑话里有一则就是“修修改改，结果甲方还是要第一版”，代码基本也不是一版就能定型。在不断的修改中，很容易就不记得第一版是怎么实现的了。但是通过Git可以把初版保存下来，需要的时候直接恢复成初版。
2. **对比代码修改。**Git可以把当前的代码和历史代码进行对比，你可以清楚的看到本次编程修改了哪些代码。
3. **代码更新。**在使用远程代码库（比如ECBM库）的时候，可以通过拉取功能将本地的代码更新和网络上一致的版本。

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。

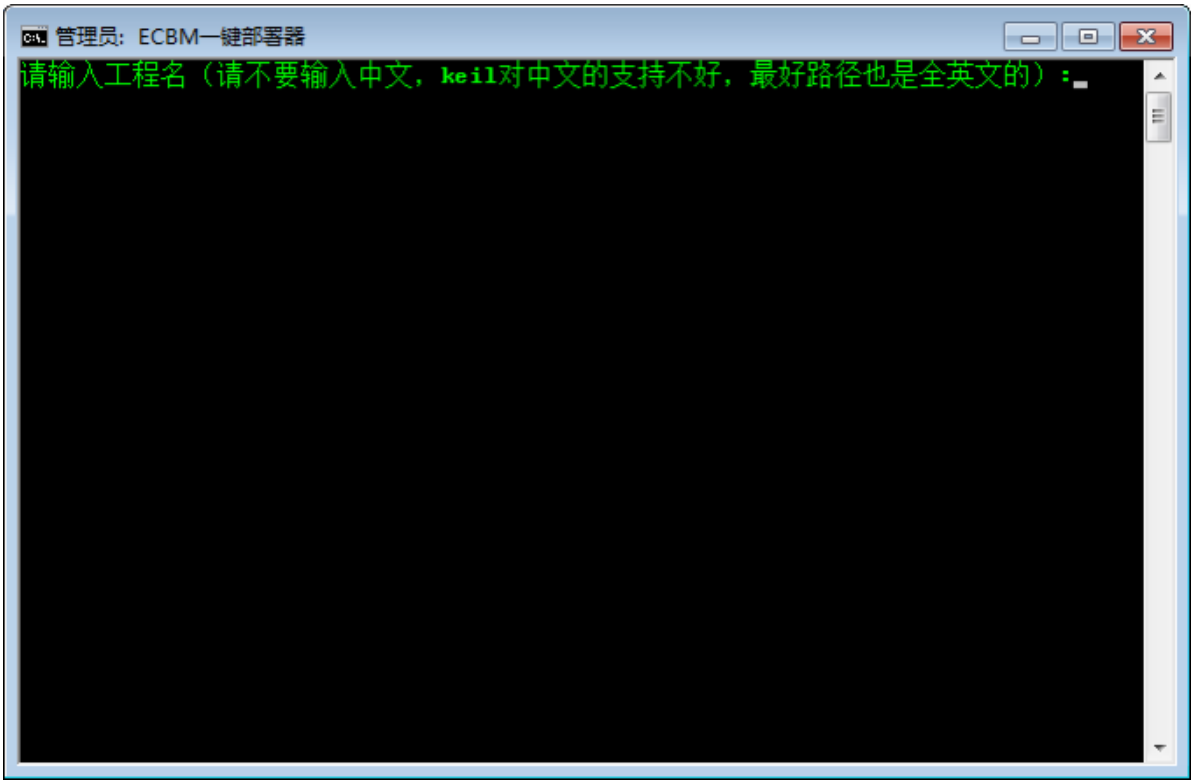
如何开始愉快的编程?

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。



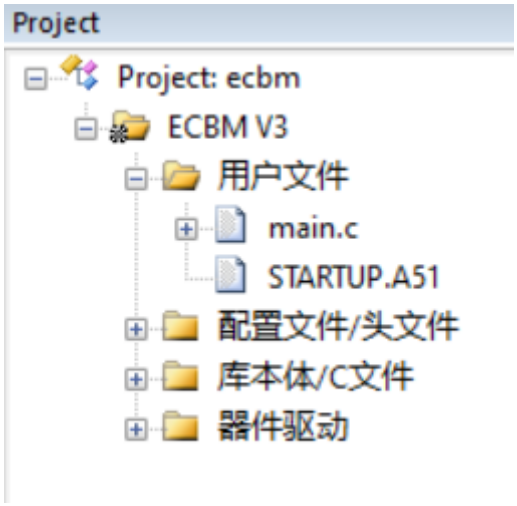
假设输入的是“test”，那么按下回车键之后，脚本会自动建立ecbm_test文件夹，并把库文件全复制到ecbm_test文件夹中。没被复制的都是些不重要的文件，比如开源证书。而output文件夹会在Keil编译的时候自动建立。

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：



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

【配置文件/头文件】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。

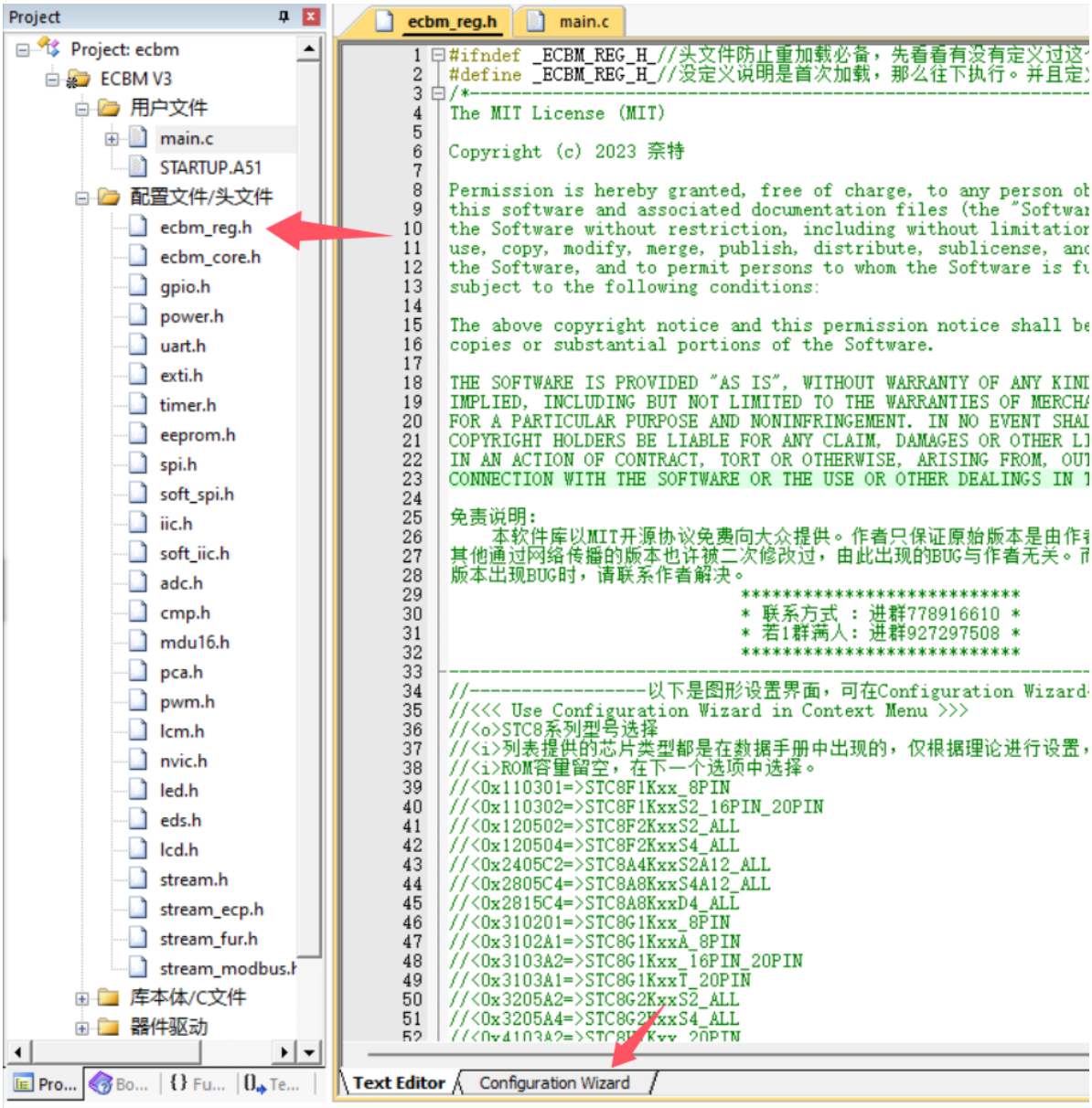
【库本体/C文件】放着ECBM库的.c文件，要参与编译就必须把.c放进来，这是正常操作。

【器件驱动】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。

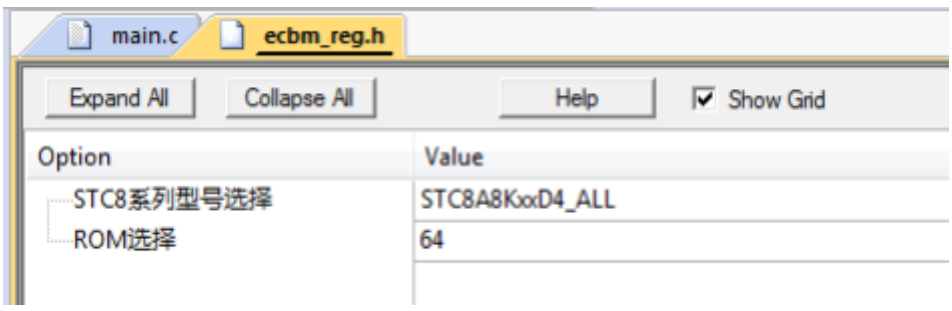
单片机型号设置

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【配置文件/头文件】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。

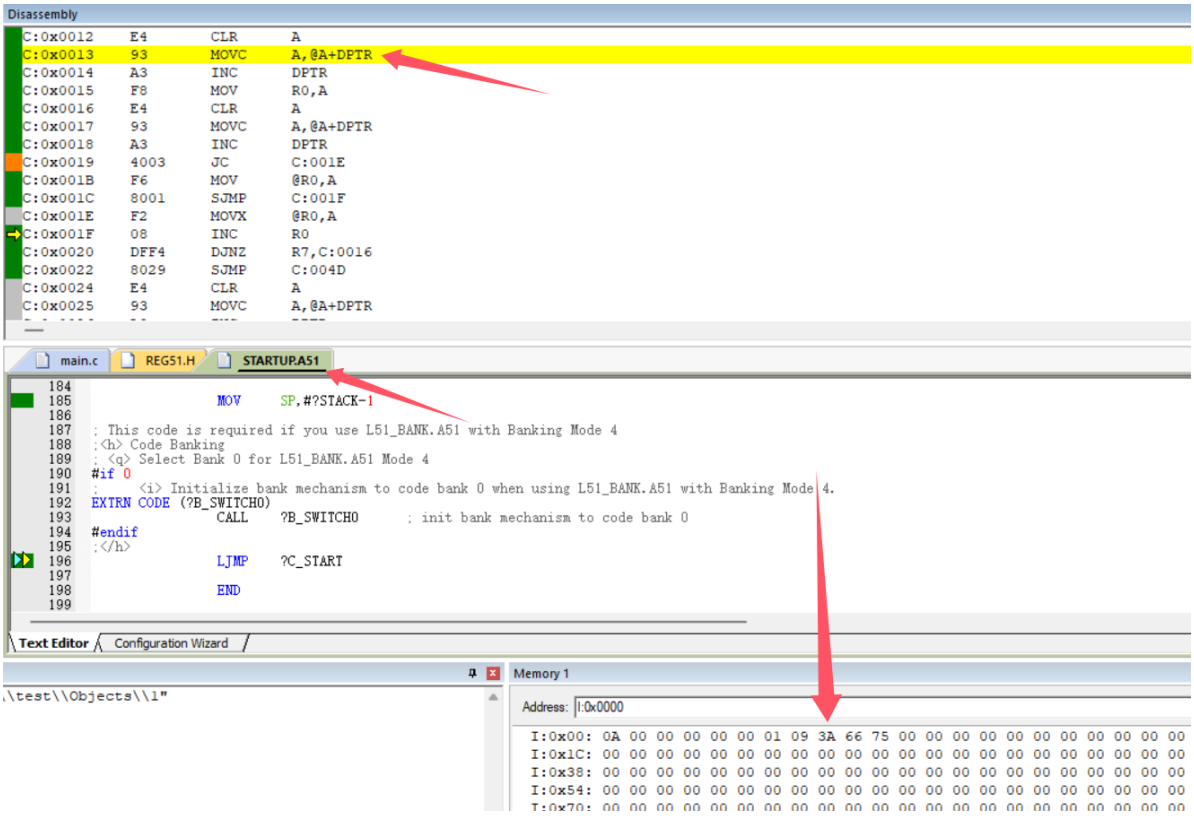
XDATA区初始化的大小设置

这个功能很重要，本来清除RAM的工作是在STARTUP.A51里完成的，清零之后会根据代码赋初值。但是有人反馈说单片机启动时间太慢，他们的应用中，需要单片机一上电就要尽可能快的进入工作状态，否则板子上的其他器件就会错过初始化的时机。而启动时间太久的罪魁祸首就在于清除RAM，因为这就是一个for语句加赋值0，需要清除的范围越大，消耗的时间自然就越久。

解决方案就是将清零的动作从STARTUP.A51移到main函数中。下面举例说明一下这样的移动会造成什么样的影响。

```
#include <reg51.h>//头文件
//清除和赋初值都在STARTUP.A51里执行。
unsigned char var=58;
void main(void){
    if(var==58){//肯定是会满足这个条件的，
        P00=1;//最后会执行这个。
    }else{
        P00=0;
    }
}
```

单片机原来的启动顺序是【STARTUP.A51】->【main.c】，在STARTUP.A51中清除IDATA和XDATA区之后，按代码赋值然后才到main.c里执行main函数。也就是说在上面的代码中，“u8 var=58;”这句话是在【STARTUP.A51】里完成的。我们可以编译上面的代码，然后点击调试来验证：

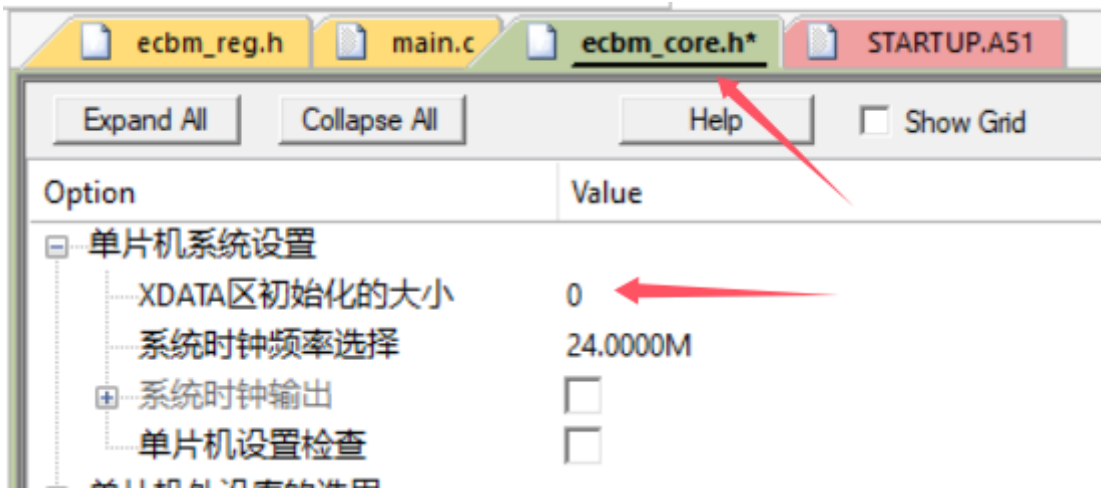


我们可以看到有一串代码的作用就是从CODE区将值拿出来，赋值到IDATA区。注意58的十六进制就是0x3A！也就是说这代码就是在给var赋初值的！而视窗也提示现在还在【STARTUP.A51】。（为了增强验证，一个0x3A是不够的，其实后面还有一堆变量，截图是过程当中，整个流程还没有结束）。

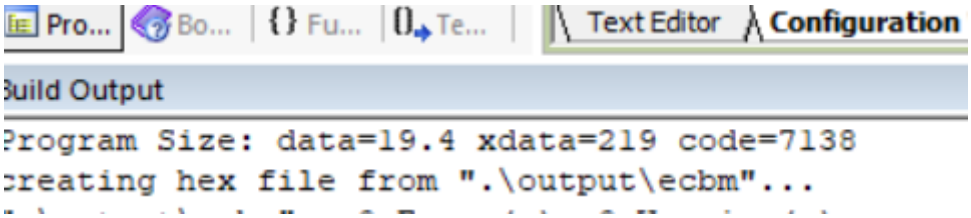
那么将清零流程移动到system_init之后是怎么样呢?

```
#include "ecbm_core.h"  //加载库函数的头文件。
u8 xdata var=58;//在【STARTUP.A51】里，var变量被赋值为58了，接下来运行main函数。
void main(void){        //main函数。
    system_init();      //系统初始化函数，清零语句目前在这里面。在这里XDATA区会被清零，var自然不能幸免。
    while(1){
        if(var==58){//由于var变成0，这个条件就不满足了，
            P00=1;
        }else{
            P00=0;//最后执行的是这个。
        }
    }
}
```

这样说明之后，大家应该会明白，为啥最近版本好像都有初值不对情况，因为这个功能在最近几个版本已经实装了，但是还没有相应的配置约束。为了不引起困惑，这个功能将会默认关闭，有相应的需要再打开。



如上图所示，设置为0就会关闭这个功能，设置为1024就会清除0~1023这个区域的XDATA空间。



可以先编译一轮，看看XDATA使用了多少空间，然后再来设置。比如上图就是用掉了219字节。

那么为什么非要这么复杂，非要清零呢？因为之前就遇到很多从VC入门（大学课程是这个顺序）转到嵌入式的，对内存的申请和调用都是电脑那一套，殊不知电脑运行的环境已经有很多库在打点了，但是单片机就是你一个人全权负责的。于是会发生以下这些情况：

```
u8 uart_cmd;//正常的定义。
...//其他代码，不展示了。
void main(void){
    uart_init();//初始化串口。
    EA=1;//开总中断。
    while(1){
        if(uart_cmd){
            uart_cmd=0;
            ...//其他代码，不展示了。
```

```
        }
    }
}
void uart_callback(void){//中断回调，也没问题。
    if(cmp("ON",SBUF)){//比较函数，用于比较收到的字符串是不是ON
        uart_cmd=1;
    }
    if(cmp("OFF",SBUF)){//比较函数，用于比较收到的字符串是不是OFF
        uart_cmd=2;
    }
}
```

这是截取的一段通过串口发送ON或者OFF来决定LED亮灭的程序，当时他的问题就是一上电就触发动作了，串口助手还什么都没发送呢。检查了电路杂波，检查了库用得对不对，为了排除库的原因又用裸机再写了一遍。结果还是不行，重新做了最小系统版还是不行。在群里求救，那时我刚好在研究启动文件，就让他把初值赋好，如果0代表没有动作，定义的时候就等于0。他回去一试，就是把u8 uart_cmd;改成u8 uart_cmd=0;就好了功能就正常了。事后他还很诧异，觉得变量定义之后没复制应该都是0才对。但很可惜不管是遗留问题还是设计问题，单片机可不是这样运作的。这就是真实发生的事件，之后V3.4.2版就加上了清零功能。

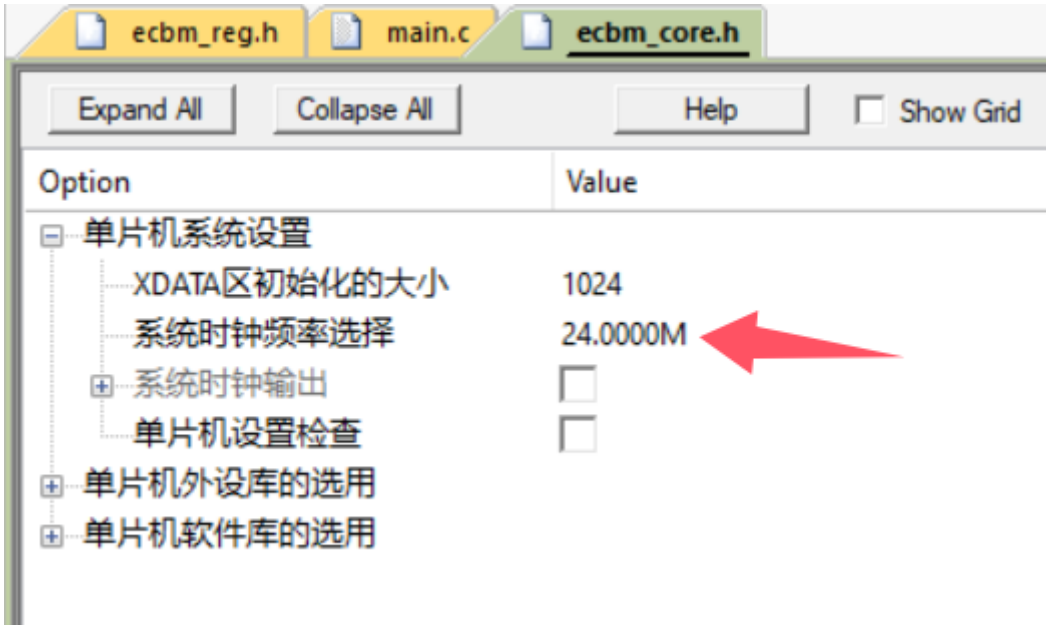
不过以我的项目经验而言，我推荐大家采用这种写法：

```
...//省略的代码。
u8 var;//变量定义的时候尽管定义。
u16 count;
sbit moto_en=P3^5;//IO定义也是一样。
...//省略的代码。
void io_init(void){//窍门就是把所有业务IO都放一起初始化，这样找起来也好找。
    gpio_mode(D35,GPIO_OUT);//D35不等于P35，gpio库.pdf对此有说明。
    moto_en=1;//STC8最新的型号都是默认高阻，所以先设置IO模式为推挽，再输出高电平。
}
void value_init(void){//把所有业务需要的变量都放在这里初始化。
    var=8;//根据需求，在这里定义初值。
    count=0;
}
void main(void){
    io_init();//对于某些外部器件，如果需要快速启动的，可以在system_init之前先执行。
    system_init();//系统初始化。
    value_init();//变量初始化，由于system_init函数里会清零XDATA，所以最好是排在
system_init函数后面。
    while(1){
        ...//省略的代码。
    }
}
```

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。


ecbm_reg.h
main.c
ecbm_core.h
Expand All
Collapse All
Help
Show Grid
Option
Value
单片机系统设置
XDATA区初始化的大小
1024
系统时钟频率选择
24.0000M
系统时钟输出
单片机设置检查
单片机外设库的选用
单片机软件库的选用