在 Keil MDK 环境下使用 STM32 V3.4 库“小”教程

简介

写这篇“小”教程主要是和大家分享使用 STM32 的基本方法。在一年以前，我开始接触并开始使用 STM32。STM32 价格便宜，外设丰富，开发和仿真环境使用方便，一下子便爱上了它。我当时使用了 IAR 编译环境，固件库也是以前的 V2 版本。由于 ST 公司更新了 STM32 的固件库，所以想试着使用新固件库。刚开始使用新库时也遇到了一些问题，但是慢慢熟悉不但觉得不难不烦，反而觉得 V3 比 V2 更好用。在这里我和大家分享一下使用 V3.4 库的方法，希望大家喜欢，如有错误请指出，在下不慎感激。

这篇“小”教程分以下四步来说，第一步，获得库文件，并进行适当的整理；第二步，建立工程，并建立条理清晰的 GROUP；第三步，修改工程的 Option 属性；第四步：使用 JLINK 仿真调试。下面就分这四大步来逐个说明。

第一步 获得库文件，并进行适当的整理

第一步非常的简单，访问 ST 的官网上就可以获得最新的固件库，在我写“小”教程的时候最新的固件库是 V3.4。除了获得固件库之外还可以获得和固件库相关的说明文档。在以前的官网上可以下载到一篇名为《如何从 STM32F10xxx 固件库 V2.0.3 升级为 STM32F10xxx 标准外设库 V3.0.0》的应用手册，但是在现在的 ST 官网上却找不到这篇十分有用的应用文档，不过却可以在百度文库中找到，这篇文档详细说明了新固件库的文件结构，在 Keil 工程建立之前，值得一看。

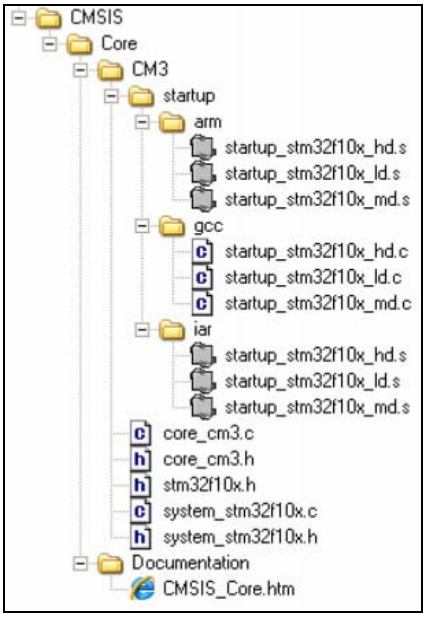

图 1 CMSIS 文件夹包含内容

图 1 是新固件库改动比较大的部分，ST 称为 CMSIS。在这个文件夹下面出现了一些新的源文件、头文件和启动代码，新的源文件如 core_cm.c system_stm32f10x.c，也有新的启动代码如 start_stm32f10x_h/m/ld.s。在第二部分会详细介绍这些文件到底有什么作用，以及和 V2 版本的区别。在这里我也补充一句，V3.4 还是和 V3.0 有点区别，V3.4 又比 V3.0 多出了几个启动代码。

我个人觉得这些文件“埋”的太深，使用起来有点不方便，所以我一般对这些文件进行一些整理，把相关文件放在一起，并取上一个标准化的名字，这些文件夹的名字一般和原始固件库文件夹的名字相同，只是把需要的文件放在一起。例如我把启动代码（startup）放在一个文件夹下面，而在这个文件下面只放 Keil MDK 有关的启动代码，把 IAR 和 GCC 的文件全部给去除了，这样做不但使得文件夹内容“清爽”也可以避免不必要的错误。一般在工程目录下面我会建立以下几个文件夹，如图 2 所示。当然还会建立两个很有用的文件夹，一个取名为 Listing，另一个取名为 Object。这两个文件夹会保存 Keil 编译连接过程中产生的一些文件，虽然是一个不起眼的细节但是也请大家关注，不然在工程目录下面“邋遢”的很！

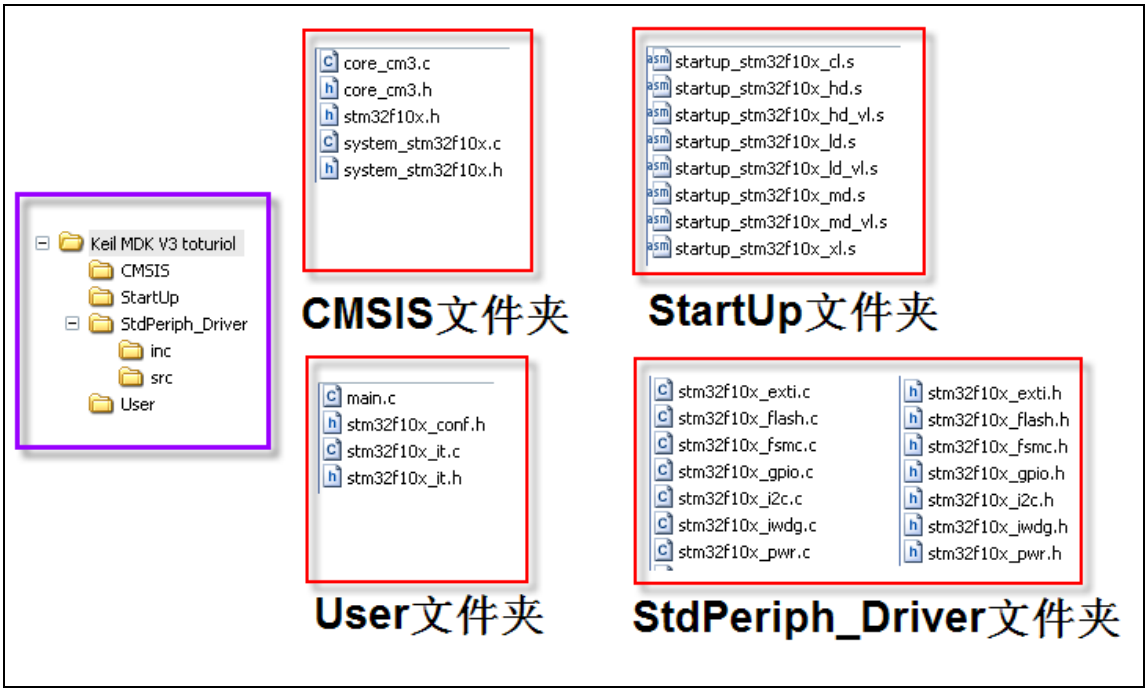

图 2 工程文件夹的结构和相关文件

下面来简单说说这些文件各有什么作用。在下重在应用，对里面的内容也知之甚少。

core_cm3.c/core_cm3.h

该文件是内核访问层的源文件和头文件，查看其中的代码多半是使用汇编语言编写的。在线不甚了解。

stm32f10x.h

该文件是外设访问层的头文件，该文件是最重要的头文件之一。例如定义了 CPU 是哪种容量的 CPU，中断向量等等。除了这些该头文件还定义了和外设寄存器相关的结构体，例如：

```
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

包含了那么多寄存器的定义，**那么在应用文件中（例如自己编写的 main 源文件）只需要包含 stm32f10x.h 即可**，而不是以前固件库的需要包含 stm32f10x_conf.h 这个头文件。

system_stm32f10x.c/h

该头文件也可以称为外设访问层的头文件和源文件。在该文件中可以定义系统的时钟频率，定义低速时钟总线和高速时钟总线的频率，其中最关键的函数就是 SystemInit()了，这个后面会详细介绍。总之这两个文件是新固件库的重点，有了它粮也大大简化了使用 stm32 的初始化工作。

stm32f10x_conf.h

这个文件和 V2 版本的库的内容是一样的，需要使用哪些外设就取消哪些外设的注释。例如需要使用 GPIO

功能，但不使用 SPI 功能，就可以这样操作。

```
#include "stm32f10x_gpio.h"
/*  #include "stm32f10x_spi.h"  */
```

main.c

这个文件就不用多说了，自己编写。

stm32f10x_it.c/h

这两个文件包含了 stm32 中断函数，在源文件和头文件中并没有把所有的中断入口函数都写出来，而只写了 ARM 内核的几个异常中断，其他的中断函数需要用户自己编写。stm32f10x_it.c 的最后给了这样一个模板。

```
/******************************************************************************/
/*               STM32F10x Peripherals Interrupt Handlers                     */
/*  Add here the Interrupt Handler for the used peripheral(s) (PPP), for the   */
/*  available peripheral interrupt handler's name please refer to the startup    */
/*  file (startup_stm32f10x_xx.s).                                            */
/******************************************************************************/
/**
  * @brief  This function handles PPP interrupt request.
  * @param  None
  * @retval None
  */
/*void PPP_IRQHandler(void)
{
}*/
```

从注释中的英文提示可以看出，中断向量的名称可以从相应的启动代码中找出，例如可以在 startup_stm32f10x_md.s 中找到 USART1 中断函数的名称——USART1_IRQHandler。其他的中断函数名可以以此类推，一一获得，在这里我就不一一复述了。

StdPeriph_Driver 文件夹

该文件夹有包含两个文件夹，一个是 src 文件夹，另一个是 inc 文件夹，顾名思义，一个里面放的是元件一个里面放的是头文件。这两个文件夹包含了所有的 STM32 的外设驱动函数，其实和 V2 版本也没有太大的变化。简单来说，外设的驱动相当于 windows 的驱动函数 API，这些驱动函数看到函数名基本就可以明白这个函数的作用，例如：GPIO_SetBits 可以置位某个 IO 口，相反 GPIO_ResetBits 则可以复位某个 IO 口。我个人认为熟练使用库可以大大提高编程的效率，同时规范使用库函数也可以提高程序的可读性，让团队中的其他程序员可以快速的明白代码的作用。

第二步，建立工程，并建立条理清晰的 GROUP

从这一步开始就开始和 Keil MDK 打交道了。首先建立一个 Keil 工程，这一小步再简单不过了，Project 菜单项中点击 New uVision Project，然后保存工程文件，路径自由设定并可以包含中文。

然后选择指定的 CPU 型号，如图 3 所示。例如选择 STM32F103RB。

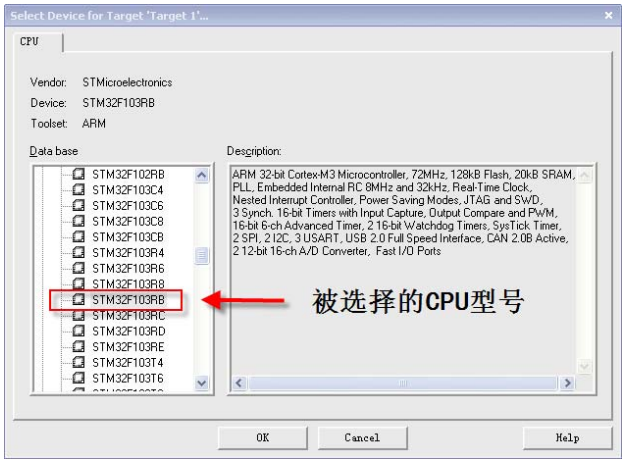

图 3 选择 CPU 型号

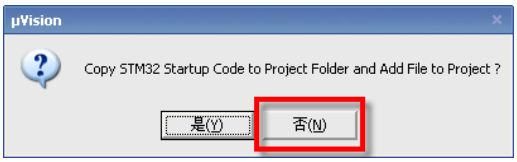

图 4 取消加载启动代码

接着弹出一个添加启动代码的窗口，**在这里请大家点击否**。因为这个启动代码是旧版本库的启动代码，新版的启动代码和这个不同，需要自己添加。所谓启动代码就是在 main 函数之前运行的代码。

以上的几个步骤和在 Keil 环境下使用 51 很相似，所以也不必多说。

选择 CPU 型号后就需要建立一个条理清晰的 Group，在这里我强调的是一个“条理性”。我尽可能的把同类的文件放在一起，并取名和工程文件目录中相同的名字，这样便于管理也避免不必要的错误。在 Target 1 选项上右击，在弹出菜单上选择 manage components，如图 5 所示。

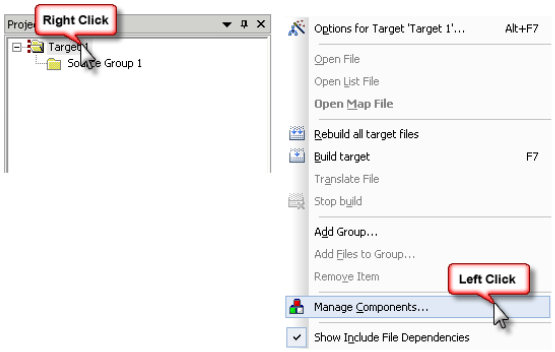

图 5 开始添加 Group

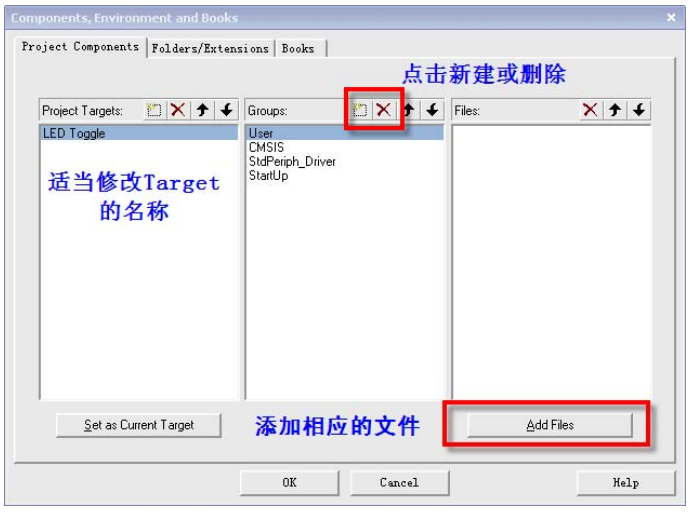

图 6 新建 Group

建立相应的 Group。例如 User，CMSIS，StdPeriph_Driver 和 StartUP，这些 Group 的名称和工程文件夹的名称保持一致，如图 6 所示。为每个 Group 添加同名文件夹下的源文件或者头文件，为了便于查看代码，我把源文件和头文件都添加进 Group 中（除 StdPeriph_Driver），**在这里注意过滤文件的类型**。StdPeriph_Driver 中只添加需要的源文件，例如建立一个 LED 闪烁的工程，那么这个工程除了进行必要的初始化之外，只需要包含 GPIO 的操作函数，当然需要使用 GPIO 就必须要使能 GPIO 的时钟，RCC 是绝对少不了的。所以只需要包含 misc.c，stm32f10x_gpio.c 和 stm32f10x_rcc.c。需要说明的是，虽然在有些 Group 中包含了一些头文件，但是 Keil 在编译连接的时却不知道头文件在什么地方，所以**一定要指定头文件的路径**。添加需要的文件之后，工程目录如图 7 所示。

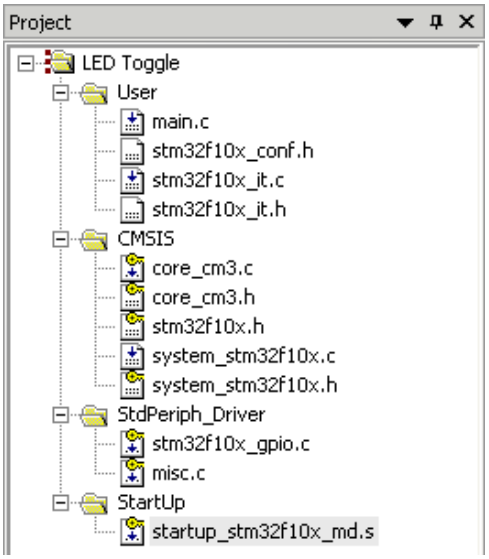
Project
LED Toggle
User
main.c
stm32f10x_conf.h
stm32f10x_it.c
stm32f10x_it.h
CMSIS
core_cm3.c
core_cm3.h
stm32f10x.h
system_stm32f10x.c
system_stm32f10x.h
StdPeriph_Driver
stm32f10x_gpio.c
misc.c
StartUp
startup_stm32f10x_md.s

图 7 工程文件夹目录

第三步 修改工程的 Option 属性

修改工程属主要目的是指定相关头文件的路径。接着上面说就是右击工程目录的 LED Toggle 则会出现 Option 选项卡，当然右击 User 或者其他的 Group 就不会出现 Option 选项卡，初学者极容易犯这个错误。

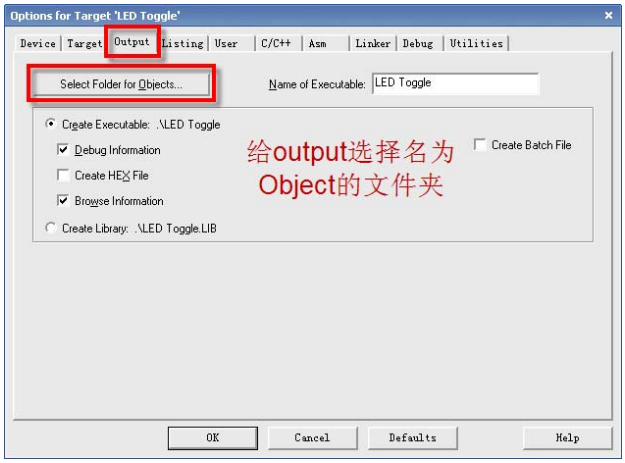

图 8 Output 选项卡

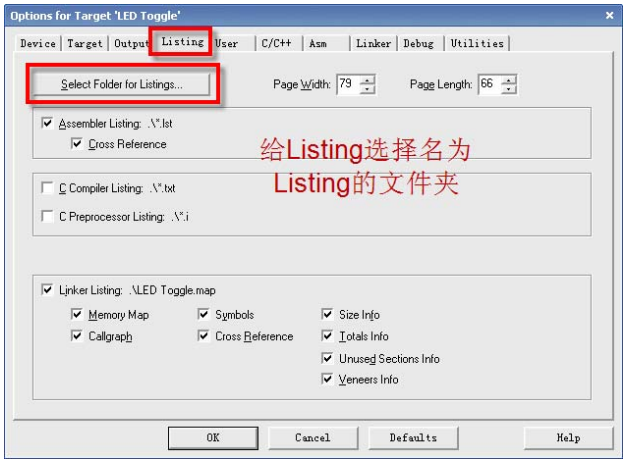

图 9 Listing 选项卡

给 Output 选择一个名为 Object 的文件夹，当然文件夹也可以是其他任何名称。给 Listing 选择一个名为 Listing 的文件夹，当然这个文件夹也可以是其他的名称。

在 C/C++选项卡下，需要输入两个非常重要的宏，一个宏是 USE_STDPERIPH_DRIVER，定义了这个宏和外设有关的函数才会包括进来，还有一个宏是 STM32F10X_MD，这个宏指定了 CPU 的容量，即中等容量的 STM32。除了设定两个宏之外，还要确定和工程有关的头文件的路径。在工程目录下面除了 StartUP 中没有相关头文件，而其他的文件中都有头文件，所以需要逐个指定。

写到这里除了仿真的选项没有设置之外，其他的参数都设定好了，此时如果编译连接工程的话，就应该显示没有错误和没有警告。当然也会遇到有错误和有警告的情况，根据错误提示耐心地寻找错误，总可以把问题迎刃而解。

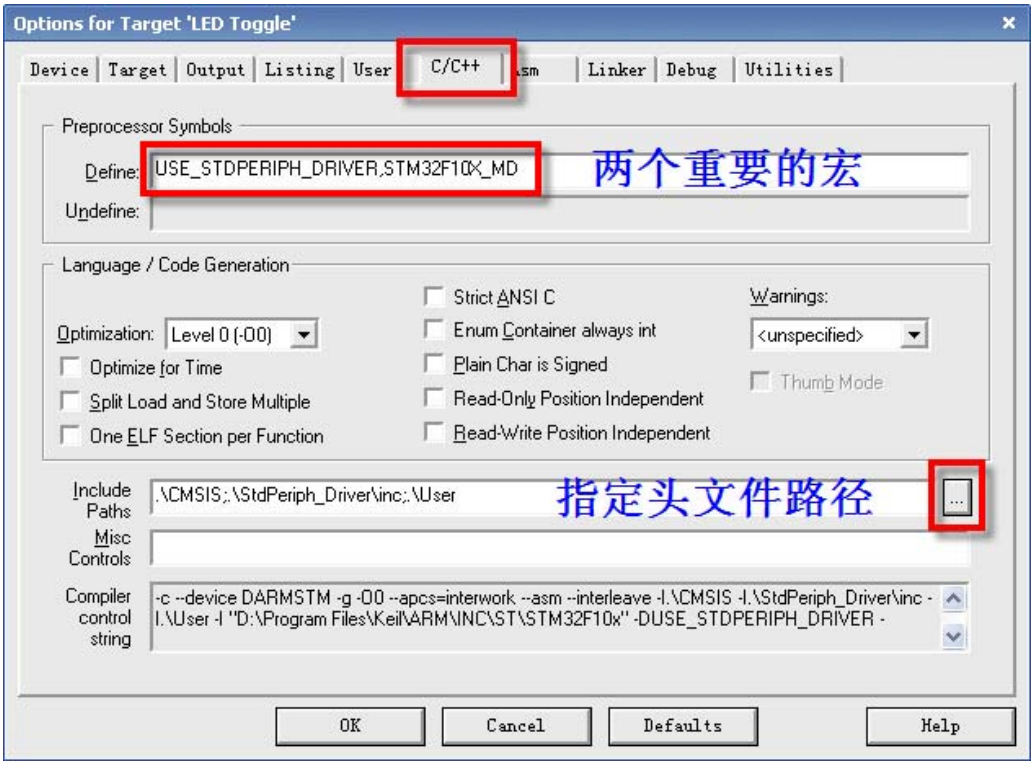

图 10 C/C++选项卡

第四步 使用 JLINK 仿真调试

在说 JLINK 仿真调试之前，我先说说我在 main.c 中写了点什么，为什么这样写，还有一个比较要命的问题的是如何初始化系统时钟。先说说我在 main 函数中写点什么，其实就是利用一个软件延时函数点亮然后熄灭一盏 LED。第一步需要初始化该 IO 口的时钟，第二步把该 IO 口配置成推挽输出形式，第三步就是进入无限循环，不断置位和复位该 IO 口。

具体的代码如下

```
int main(void)
{
  /*!< At this stage the microcontroller clock setting is already configured,
       this is done through SystemInit() function which is called from startup
       file (startup_stm32f10x_xx.s) before to branch to application main.
       To reconfigure the default setting of SystemInit() function, refer to
       system_stm32f10x.c file
     */
    /* 初始化 GPIOD 时钟 */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD , ENABLE);
    /* 初始化 GPIOD 的 Pin_2 为推挽输出*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    while (1)
    {
        /* 关闭 LED1 */
        GPIO_SetBits(GPIOD,GPIO_Pin_2);
        /* 延时 */
        Delay(0xAFFFF);
        /* 点亮 LED1 */
        GPIO_ResetBits(GPIOD,GPIO_Pin_2);
        /* 延时 */
        Delay(0xAFFFF);
    }
}
```

这段代码其实也非常简单，也是我从新固件库的例程中直接修改过来的。我想请大家注意的是其中的一段英文注释，这段英文注释什么意思呢。“在运行 main 函数之前，系统时钟已经完成初始化工作，在 main 函数之前，通过调用启动代码运行了 SystemInit 函数，而这个函数位于 system_stm32f10x.c”。根据文中的提示我们回到 system_stm32f10x.c 看看 SystemInit 如何初始化系统的。

在 system_stm32f10x.c 的开头便定义了系统的时钟频率，从下面的这段代码可以看出系统的频率被定义为 72MHZ，这也是绝大多数 STM32 运行时的频率。

```
#if defined (STM32F10X_LD_VL) || (defined STM32F10X_MD_VL) || (defined STM32F10X_HD_VL)
/* #define SYSCLK_FREQ_HSE    HSE_VALUE */
 #define SYSCLK_FREQ_24MHz  24000000
#else
/* #define SYSCLK_FREQ_HSE    HSE_VALUE */
/* #define SYSCLK_FREQ_24MHz  24000000 */
```

```
/* #define SYSCLK_FREQ_36MHz  36000000 */
/* #define SYSCLK_FREQ_48MHz  48000000 */
/* #define SYSCLK_FREQ_56MHz  56000000 */
#define SYSCLK_FREQ_72MHz  72000000
#endif
```

紧接着根据这个宏定义程序试图把系统时钟初始化为 72MHz，代码有点冗长，这里就不一一列出。在 SystemInit 函数中，调用了 SetSysClock 函数，如果设定时钟的频率为 72MHZ 则 SetSysCloc 调用 SetSysClockTo72 函数，该函数和 V2 版本固件库中的各范例中的 RCC_Configuration 很相似，主要完成把外部时钟 9 倍频后分配给系统时钟，APB1 时钟和 APB2 又由系统时钟分频获得。关键代码如下

```
    /* HCLK = SYSCLK */
    RCC->CFGR |= (uint32_t)RCC_CFGR_HPRE_DIV1;
    /* PCLK2 = HCLK */
    RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE2_DIV1;
    /* PCLK1 = HCLK */
    RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE1_DIV2;
```

在这里我想请大家注意一个细节，我个人觉得可能是 ST 开发人员的一个笔误，代码的原意是把 HCLK 时钟 2 分频过后提供给 APB1 时钟，但是注释的地方却写成了 APB1 的时钟和 AHB 的时钟频率相同。**这里请大家指正，这个总线的时钟频率到底是多少！**

从上面的分析可以看出，SystemInit 并不需要用户调用，启动代码会自动执行，这样相当于少了一个 RCC_Configuration 函数的绝大多数内容。请大家注意是绝大多数内容而不是全部，但是请大家格外注意使用到的外设还是要第一时间使得该外设的时钟，像这样的一句千万不要忘了。

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD , ENABLE);
```

让我们再回到 Option 选项卡。在选项卡中选择 Debug 选型卡，选择 Cortex M3 J-LINK。一般在 Run to main()上打钩，我个人除非特殊情况，一般不查看启动代码。如果第一次尝试 V3.4 库，倒可以验证一下 SystemInit 函数是不是自动运行了。

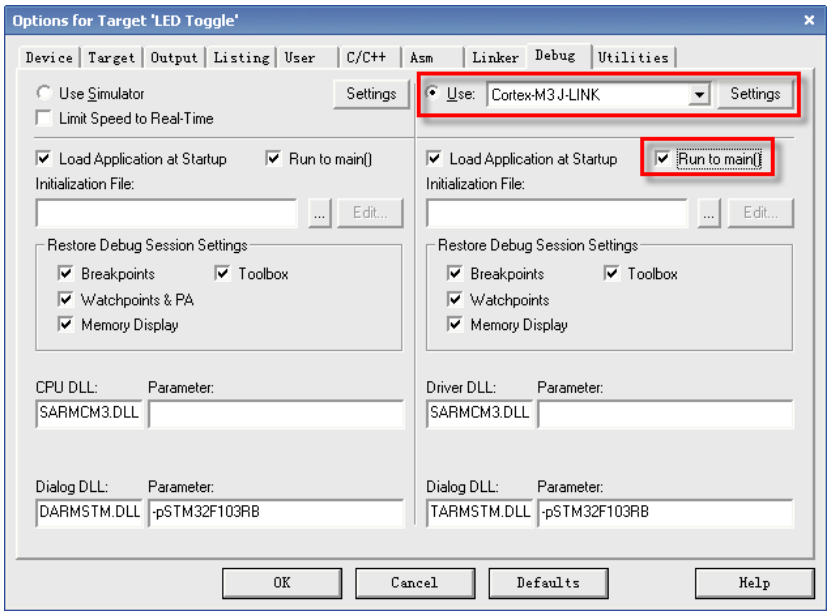

图 11 Debug 选项卡

接着点击 Utilities 选项卡，选择 Cortex-M3 J-LINK，然后在点击 Setting 按钮。

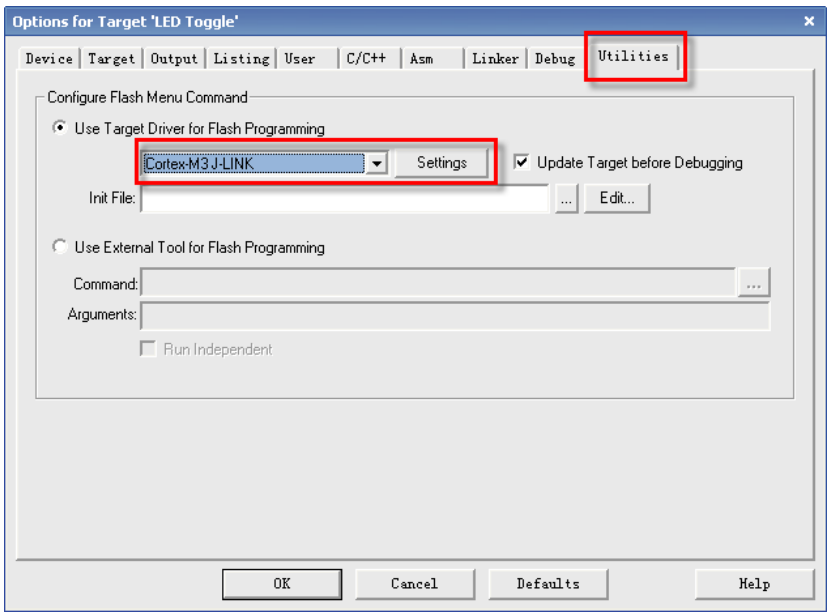

图 12 Utilities 选项卡

在点击 Setting 按钮出现的界面中，选择 Flash Download 选项卡，点击 Add 按钮，在众多的 CPU 型号中选择中等容量的 STM32，即 STM32F10X Med-density Flash。

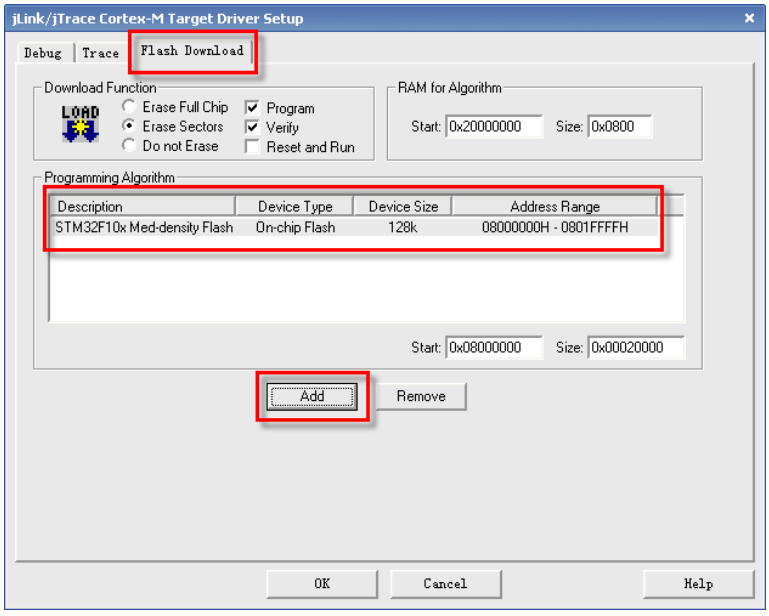

图 13 选择 CPU 的容量

紧接着点击 Debug 选项卡，当目标板和 JLINK 连接正确的话就图 12 的界面。在这里我强调，只有 JLINK 和目标板连接正确，并且目标板上电时，才会出现这样的界面。我个人一般选择 SW 下载模式，SW 下载模式只需要 2 个 IO 口，加上 VDD 和 GND 只需要 4 个 IO 口。这样可以节约 IO 口和 PCB 板的空间。关于 SW 仿真接口和 JTAG 仿真接口的详细资料，请大家查看相关的文档了。

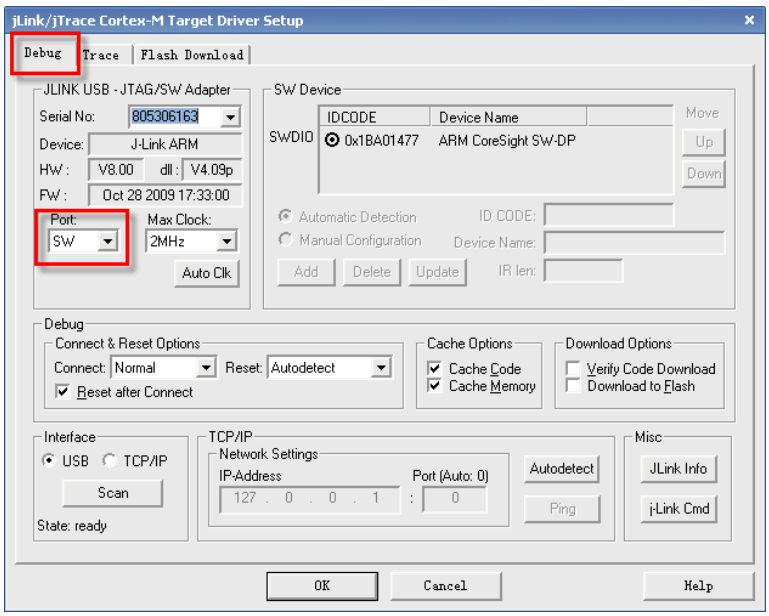

图 14 仿真接口选择

点击 OK 过后就可以放心大胆的开始 debug 了。可以通过设置断点或者单步运行，结合目标板的输出情况，验证程序是否按照要求运行。如果没有按照要求运行还是需要耐心的寻找问题。开始仿真后，程序停留在断点处，如图 13 所示。

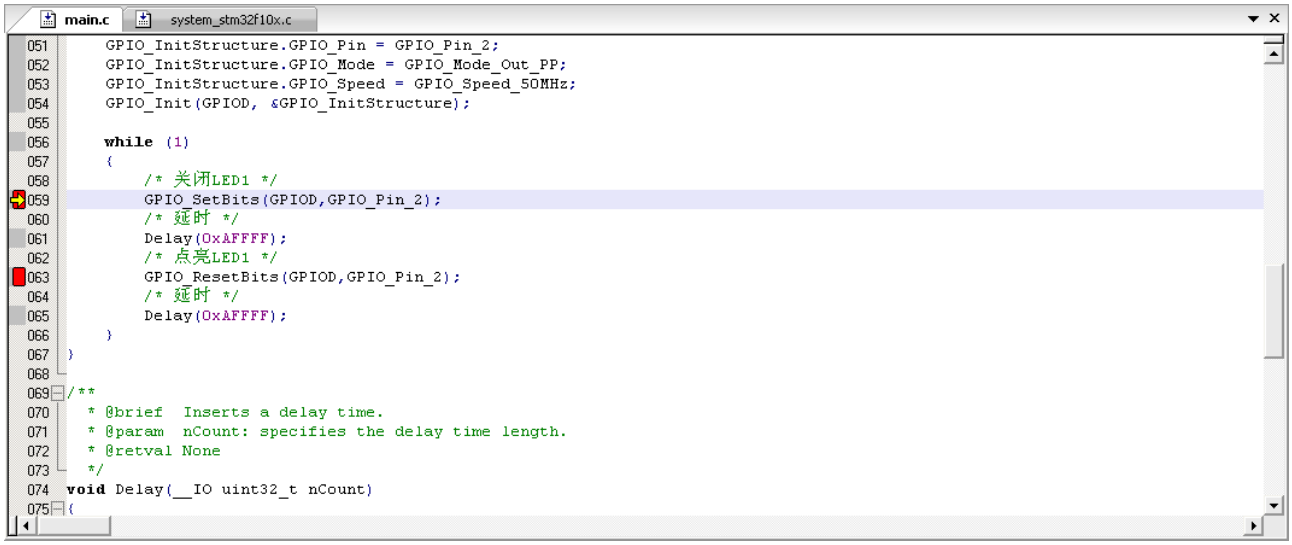

图 15 程序运行至断点

后记

在几个月以前，我自己做过一个 IAR 环境下使用 V2 库的视频教程。我把自己做好的视频教程放在了优酷网上面，并发到了 ourDev 的 STM32 板块。观看该视频教程的前辈给我提了三点意见，第一点，视频的内容不清楚，很多的操作看不清，所以我这次我就认认真真的写了一个“小”教程；第二，录制视频的时候声音不清楚，所以今天下午我跑到当地的数码市场重买了一个“麦”；第三，使用了旧版本的固件库，没有体现出新库的优势，所以这次我使用了新的固件库 V3.4。在写完这个小教程的前一晚我在 STM32 成功移植了 uc/OS-II，我想无论如何我都会和打击分享，同样是以文档加视频的方法。虽然这样做自己有点累，但是毕竟自己在网上下载了那么多的资料，自己也应该上传一点作为回报。今天是兔年的第一天，希望所有 STM32 的爱好者新年有大进步。