STC micro™
宏 晶 科 技

深圳国芯人工智能有限公司

STC32-M4

开发指南

STC MCU

技术支持网站：www.STCMCUDATA.com
资料更新日期：**2022/3/07**

第1章初学 STC32M4 的准备工作

俗话说万事开头难，学习一门新的知识，难的往往不是知识本身，而是如何快速上手，需要什么资料 和开发环境。一旦上手后，深入的学习就相对容易些了。

1.1 初学者重要提示

- 关于学习方法问题，可以看附件章节 A。
- 学会 STC32M4 相关资源的获取方法，做到心中有数，否则心里老是没底。
- STC32M4携带了 DTCM 和 ITCM，ITCM 用于运行指令，也就是程序代码，DTCM 用于数据存取，特点是 跟内核速度一样。
- STC32M4 系列配套标准库。

1.2 开发环境说明

- IDE：支持MDK 开发环境
  - MDK 推荐使用 MDK5.26 正式版及其以上版本。
- 调试器使用 ULINK2。
- 配套开发板是深圳国芯的：STC32-M4 开发板，MCU 是 STC32M4。

1.3 STC32M4开发资源查找

学习一款新的芯片，需要大家从官方获取两方面的资料，一个是相关的技术文档，比如参考手册、数 据手册、应用笔记等；另一个是软件包，官方在软件包中提供了外设驱动库和基于此库的大量例程。

学习STC32M4主要下载哪些相关手册呢? 主要有以下几个，这几个手册是我们经常要使用到的，不 光学习STC32需要这类手册，学习FPGA、DSP也是这些类型的手册，熟练查阅和使用这些手册也是电子 工程师必备的知识之一。

- 参考手册（Reference Manual）

对芯片每个外设的具体描述和功能介绍，比如我们要查 USART，SPI，DMA 相关寄存器和功能的介绍就可以使用这个手册。

- 数据手册（Data Sheet）

在我们要画 PCB 的时候用到这个手册的情况比较多，这个手册上面有关于这个系列芯片的引脚定义、电气特性、机械封装、料号定义等信息。

- Cortex-M3/M4 权威指南

这也是非常重要的参考资料，对于有兴趣了解 M3/M4 内核的同学，这个资料相当重要，了解了内核 才能更好的利用 M3/M4。

1.4 STC32M4调试方法

STC32M4 的调试方法主要为MDK调试:

MDK 调试方法在第 4 章进行了详细讲解。

1.5 总结

本章节就为大家讲解这么多，建议初学者花些时间对 STC32M4 的开发文档的章节结构了解一下，随着以后的学习最好可以达到熟练查看这些开发文档的程度。

第2章 STC32M4 的开发环境搭建

本章主要为大家讲解 STC32M4 开发板的开发板环境搭建，涉及 IDE，下载器和相关小软件的驱动安装。

2.1 初学者重要提示

2.2 常用的开发工具

2.3 MDK5 的安装和注册

2.4 STC-ISP下载与安装

2.5 ULINK2 驱动安装

2.6 总结

2.1 初学者重要提示

。

- 使用ULINK2 调试下载 STC32M4，推荐使用最新的电脑端驱动和对应的固件，详情见本章 2.6 小节。
- MDK5 的工程创建， 调试方法以及字体设置， 语法提示等设置在第 4 章和第 5 章有详细说 明。

2.2 常用的开发工具

开发STC32软件，需要配置一个仿真器（也可以叫下载器），同时必须在PC机上安装一系列软件。 可以使用ULINK2仿真器来下载和调试程序，使用STC-ISP下载软件来添加型号和头文件到KEIL中。

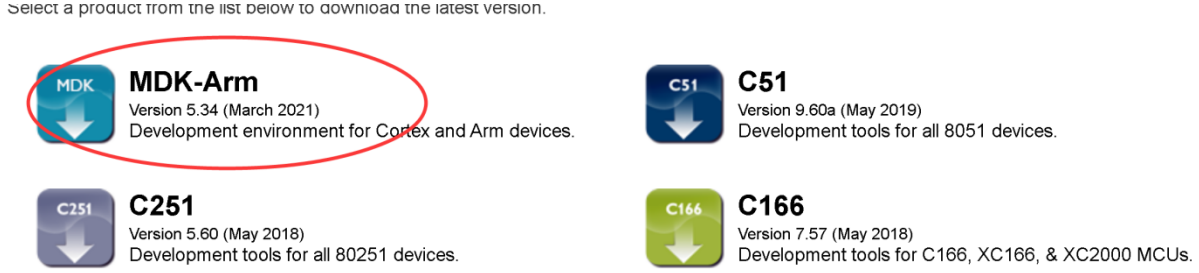
2.3 MDK5 的安装和注册

这里以 MDK5.26 为例进行说明。

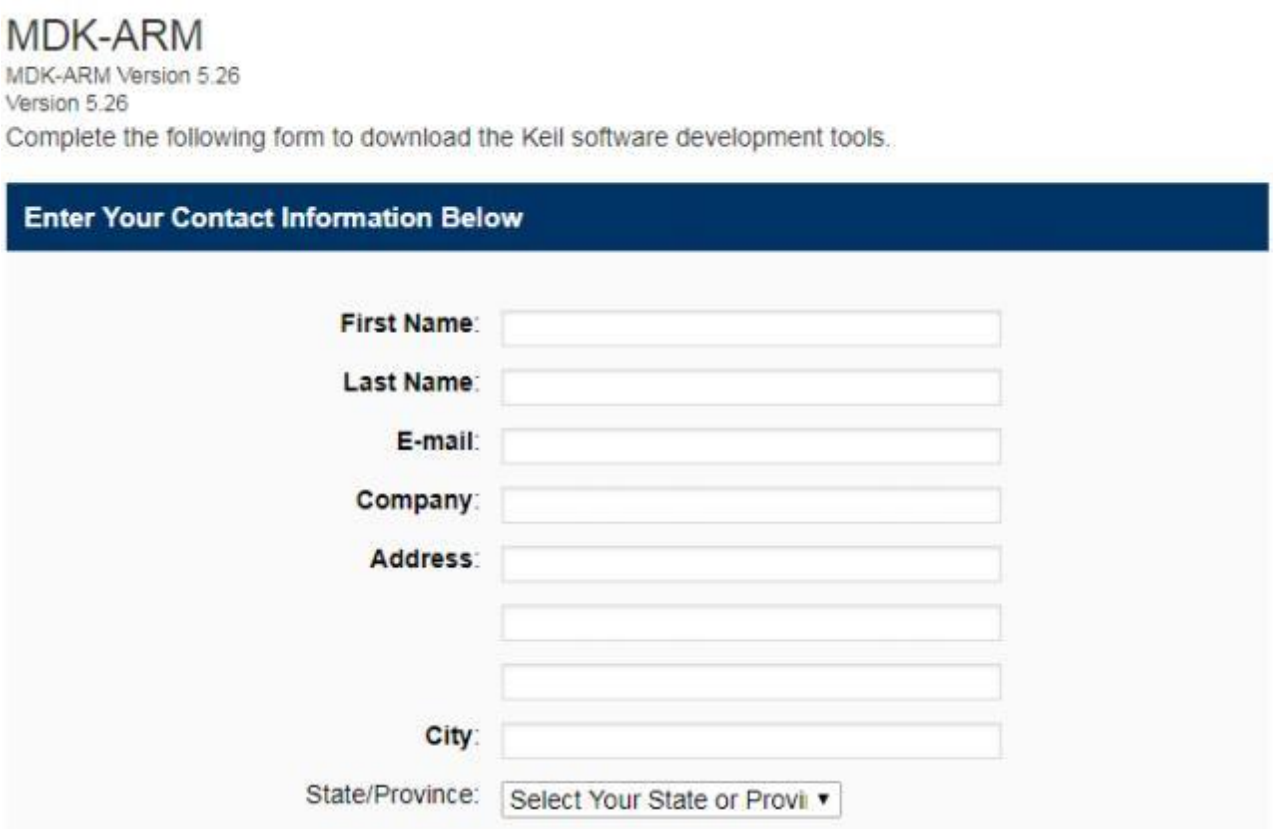
2.3.1下载 MDK5

(注，当前时间的官方版本还是 MDK5.26)

推荐直接在官方网站下载，地址： https://www.keil.com/download/product/



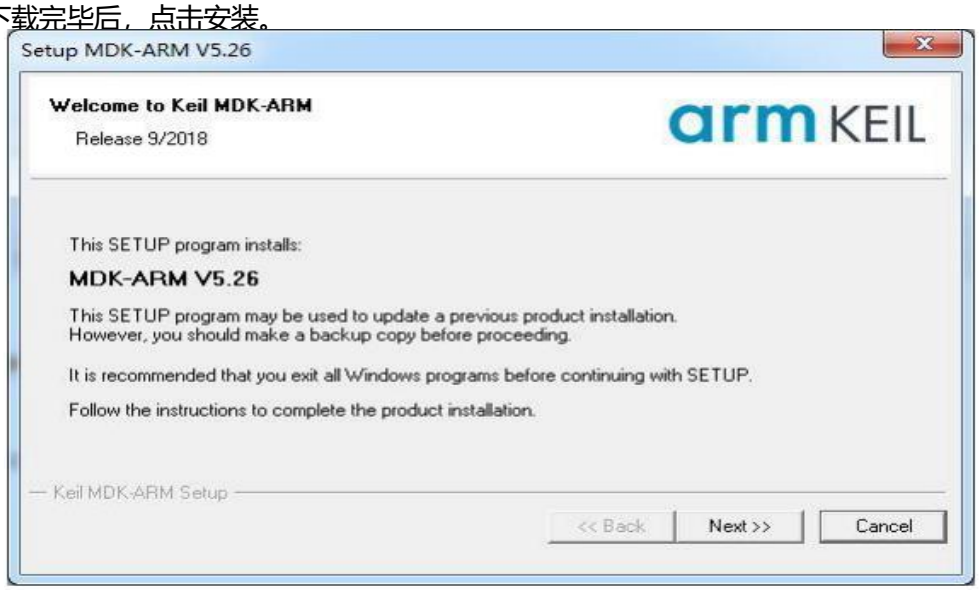
点击 MDK-Arm，来到下面地址后，需要大家简单填写注册信息后即可下载：

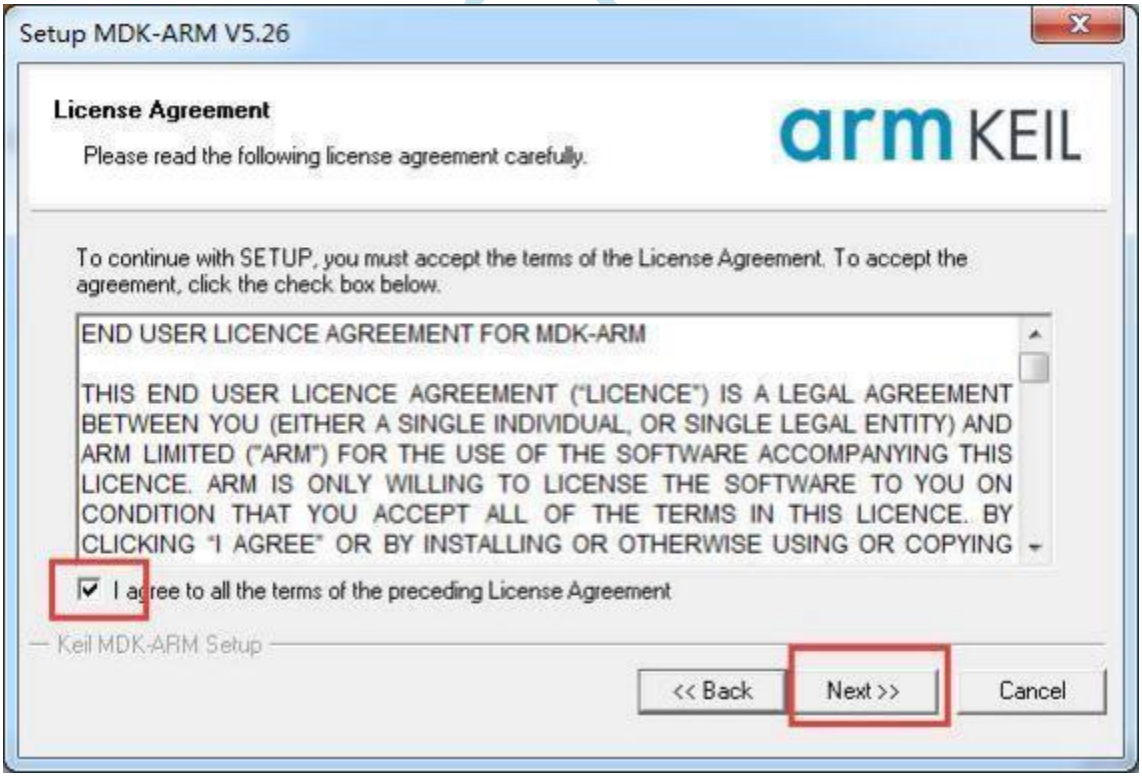


2.3.2安装 MDK5

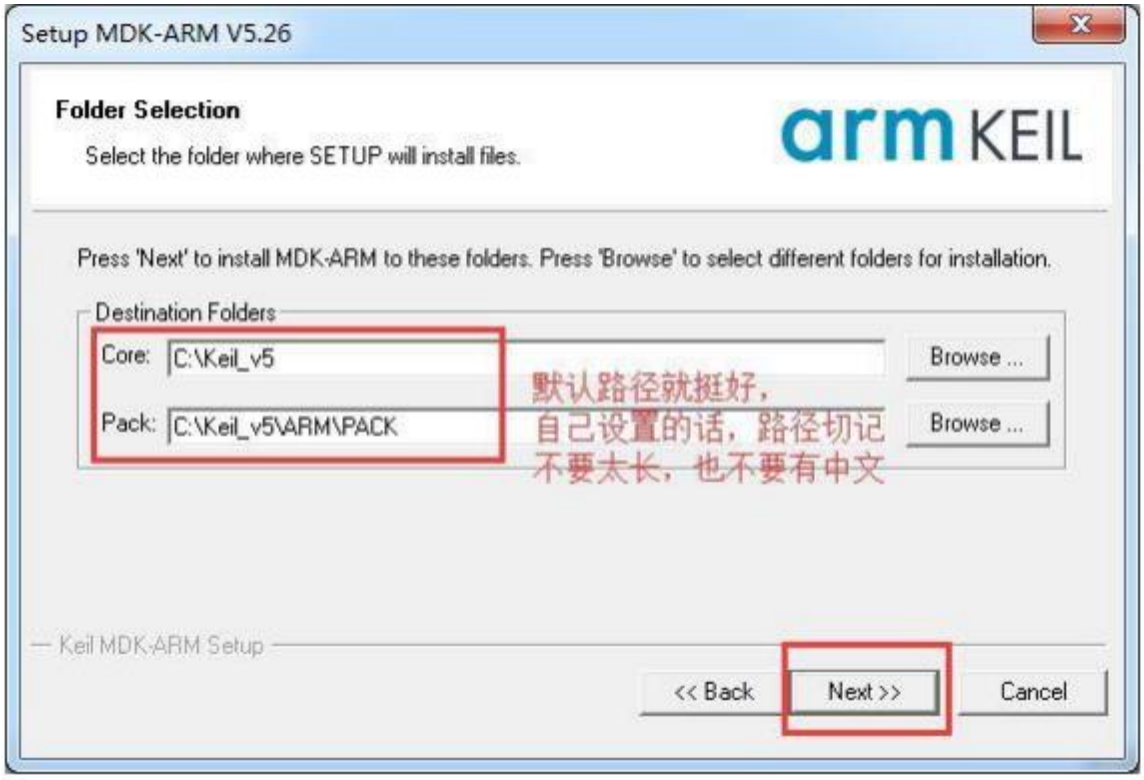
安装的时候注意不要有中文路径，路径越短越好。安装过程比较简单，一直下一步即可。

1.下载完毕后，点击安装。

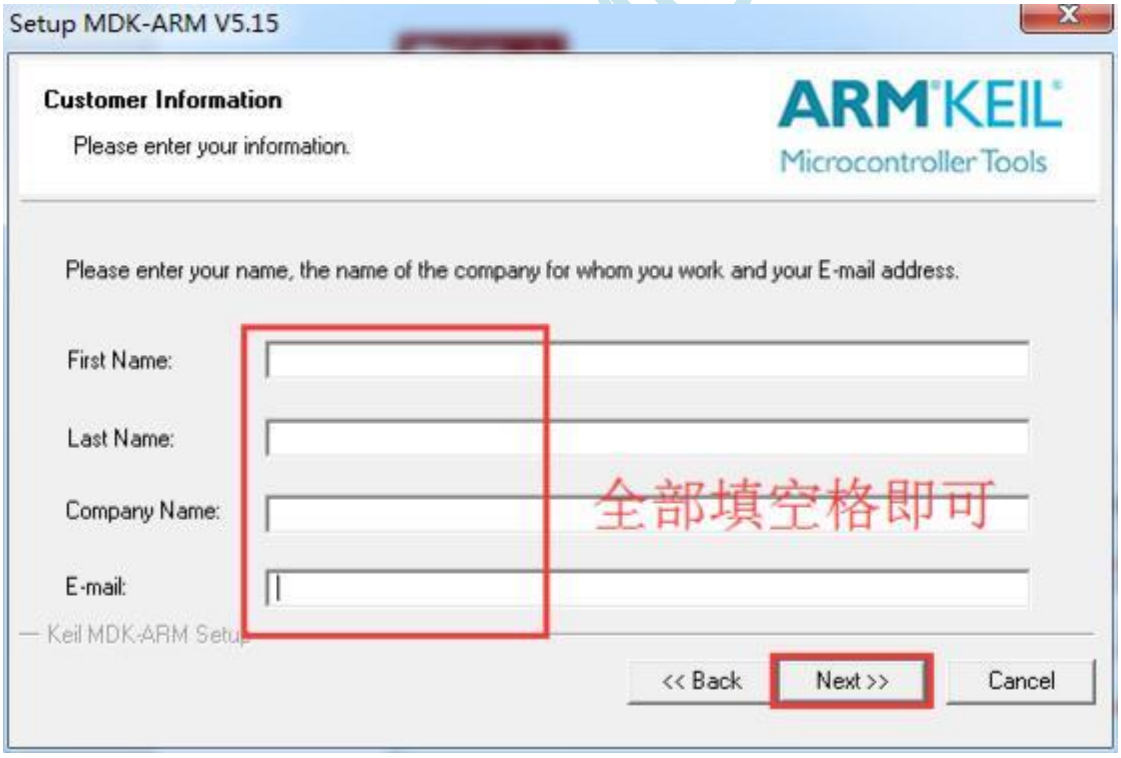




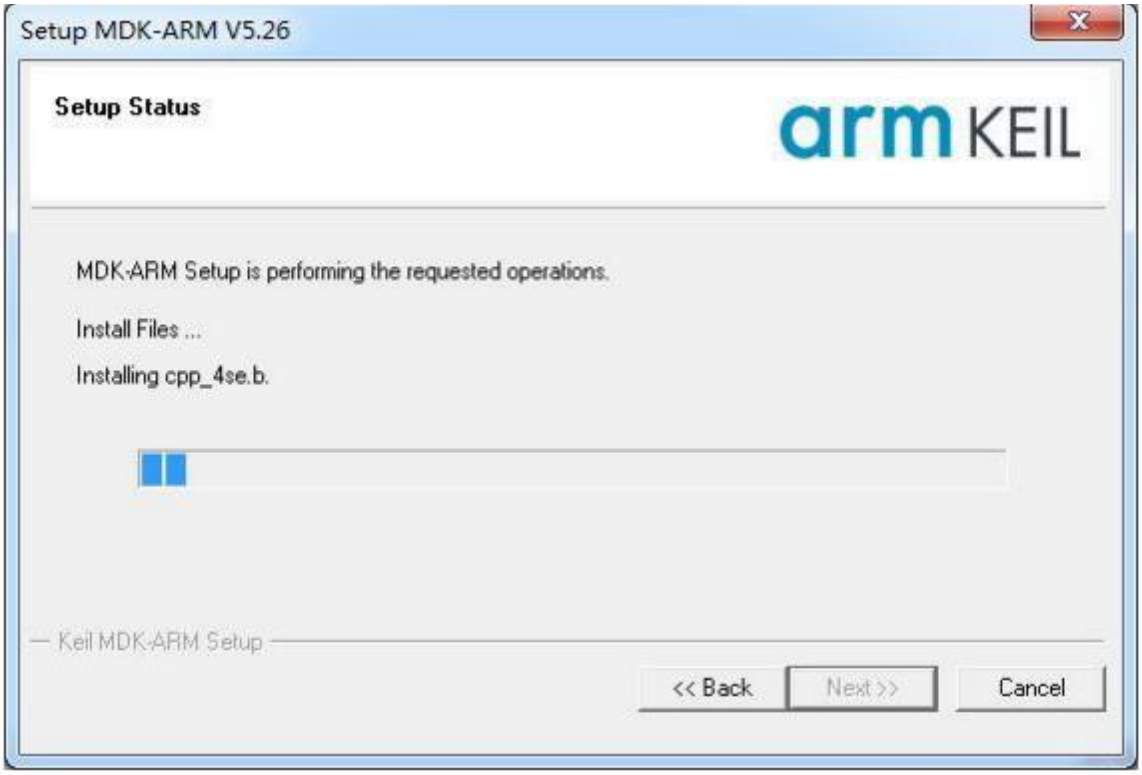
3. 注意安装路径。



4. 红色方框的个人信息随意填写，或者全部填写空格即可Next。



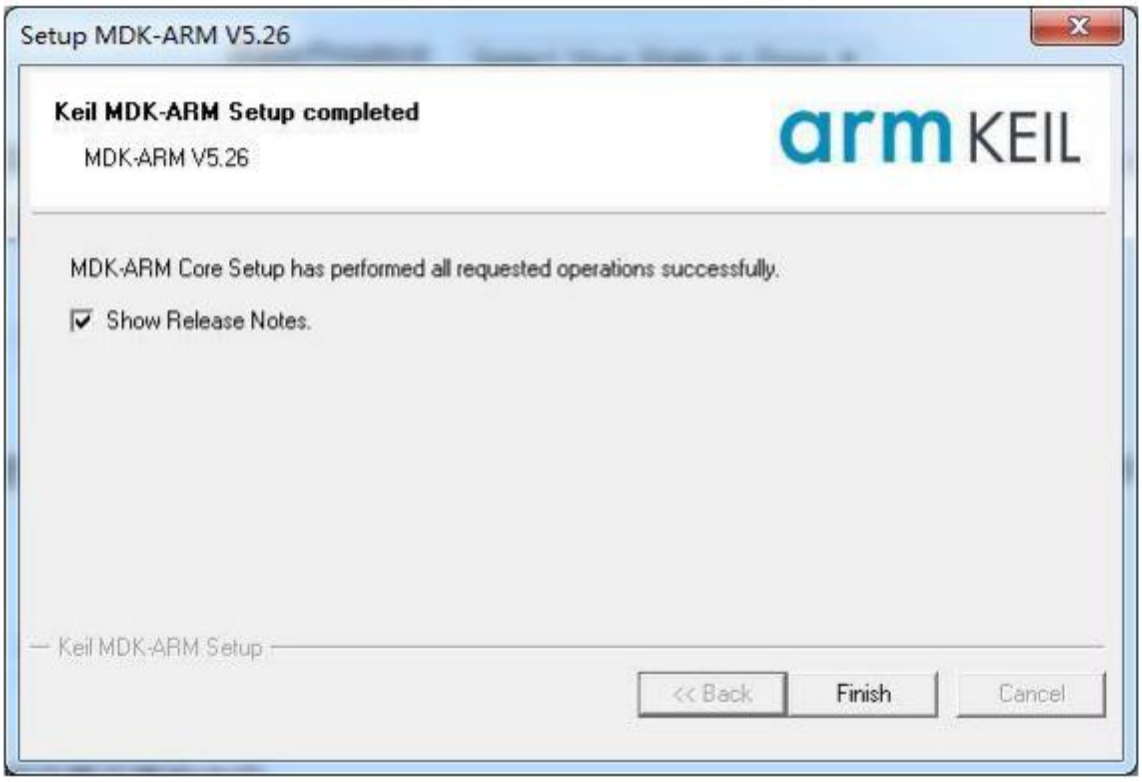
5. 剩下就是时间略长的安装过程。



6. 安装结束前会提示是否安装 ULINK 驱动，安装即可。



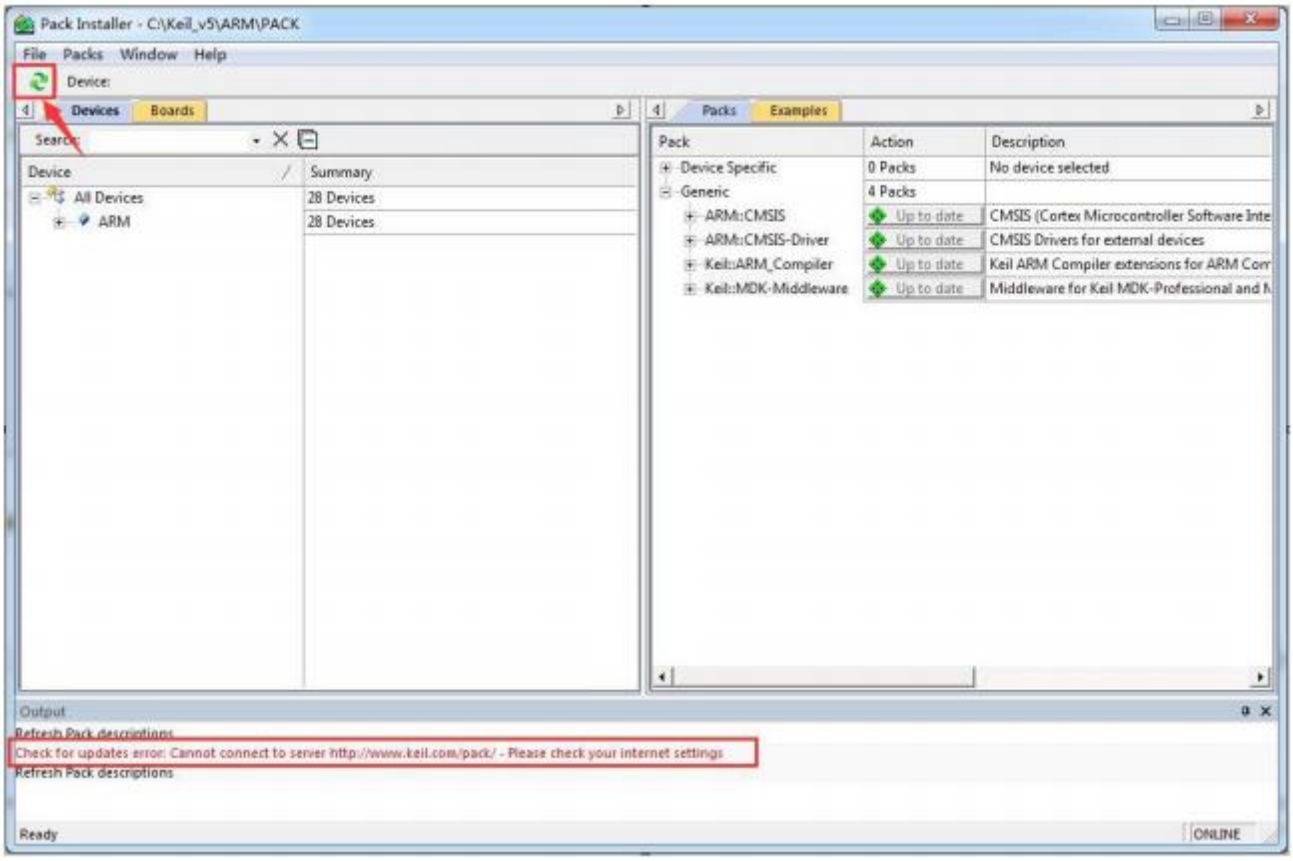
7.至此，MDK 就安装完毕了。



安装完毕后，如果弹出下面界面，是因为要更新安装包列表，这个要连接 MDK 服务器的。

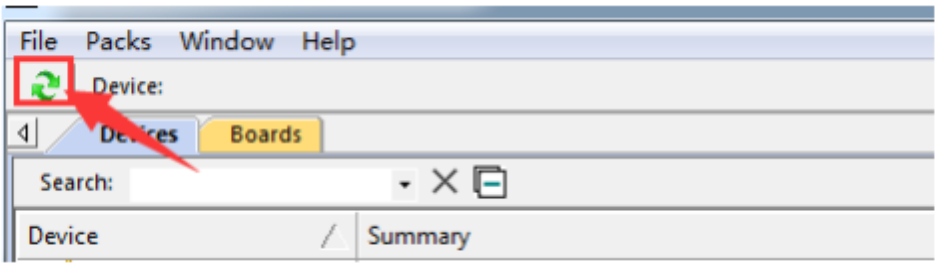


首次打开 MDK 会弹出下面的界面， 点击左上角的刷新图标， 如果是下面效果， 表示的确无法连接到 MDK 服务器，此时可以重启电脑试试。



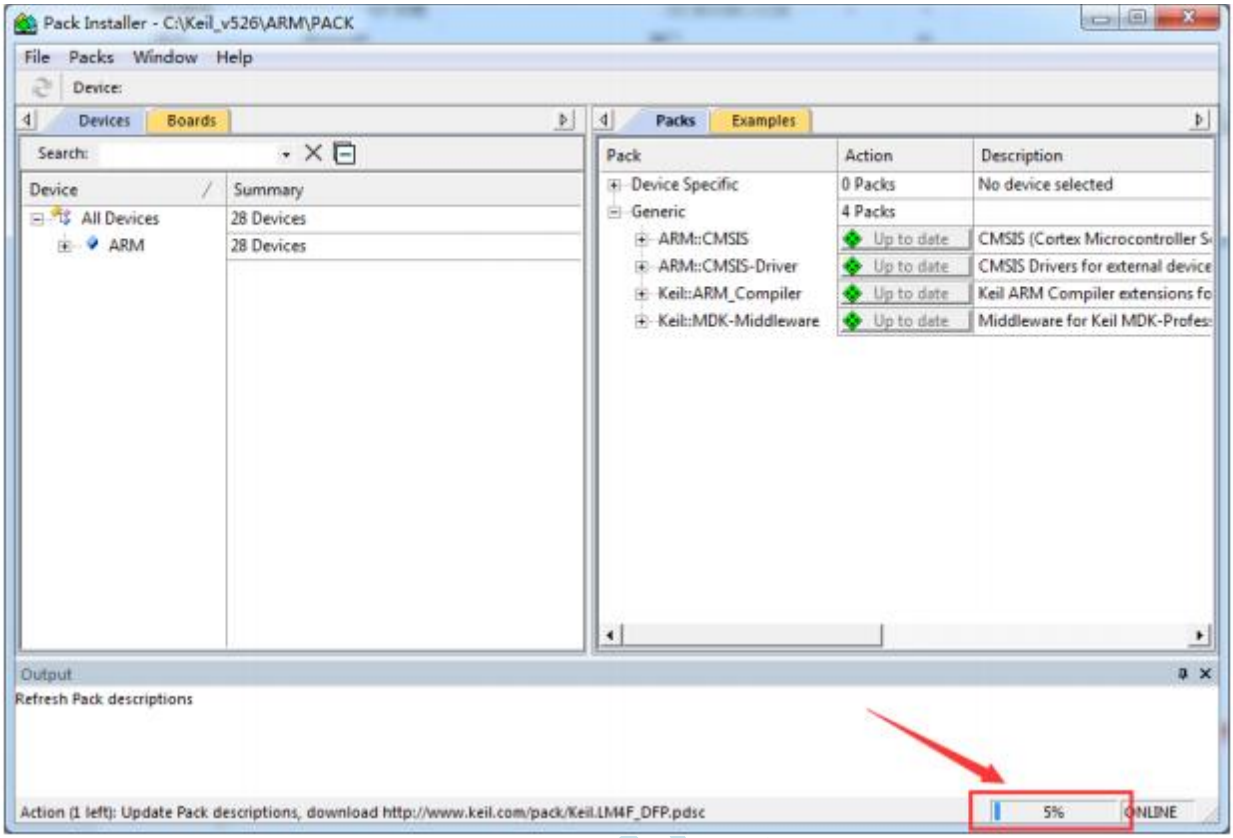
再次打开 MDK 后，点击这里弹出软件包安装界面，看看是否可以刷新成功：





如果还是有问题的话，基本不用管了，直接去官网下载相应的软件包，然后导入即可 （见下面 H7 软件包的安装）。

正常更新的时候，右下角有个更新进度：



更新完毕后的效果如下:

2.3.4注册 MDK5

1、 下载注册机 .

2、运行 MDK5，打开" File "下的" License Management... "，复制 CID 编号（11 字符）。3、运行破解软件 Keil_ARM_MDK_5.00_Keygen_serial_Crack.exe。有些杀毒软件（如卡巴斯基）会误 报为病毒程序，请临时屏蔽掉杀毒软件后再运行。4、 Keil_ARM_MDK_5.00_Keygen_serial_Crack 的 "target" 下拉列表框选择 "ARM" 。 5、第 2 行下拉列表框选择 "MDK Professional" 。6、将MDK 的 CID 编号粘贴到KEIL_lic中的 CID 编辑框内，然后点击 "Generate" 按钮。 7、把方框中生成的 30 位注册码，拷贝到 MDK 中的 "New License ID Code" 内。 8、点击 "AddLIC" 完成注册。如果截止日期太短的话，就继续用 Keil_ARM_MDK_5.00_Keygen_serial_Crack 生成注册码重新注册。



注册成功后的效果：

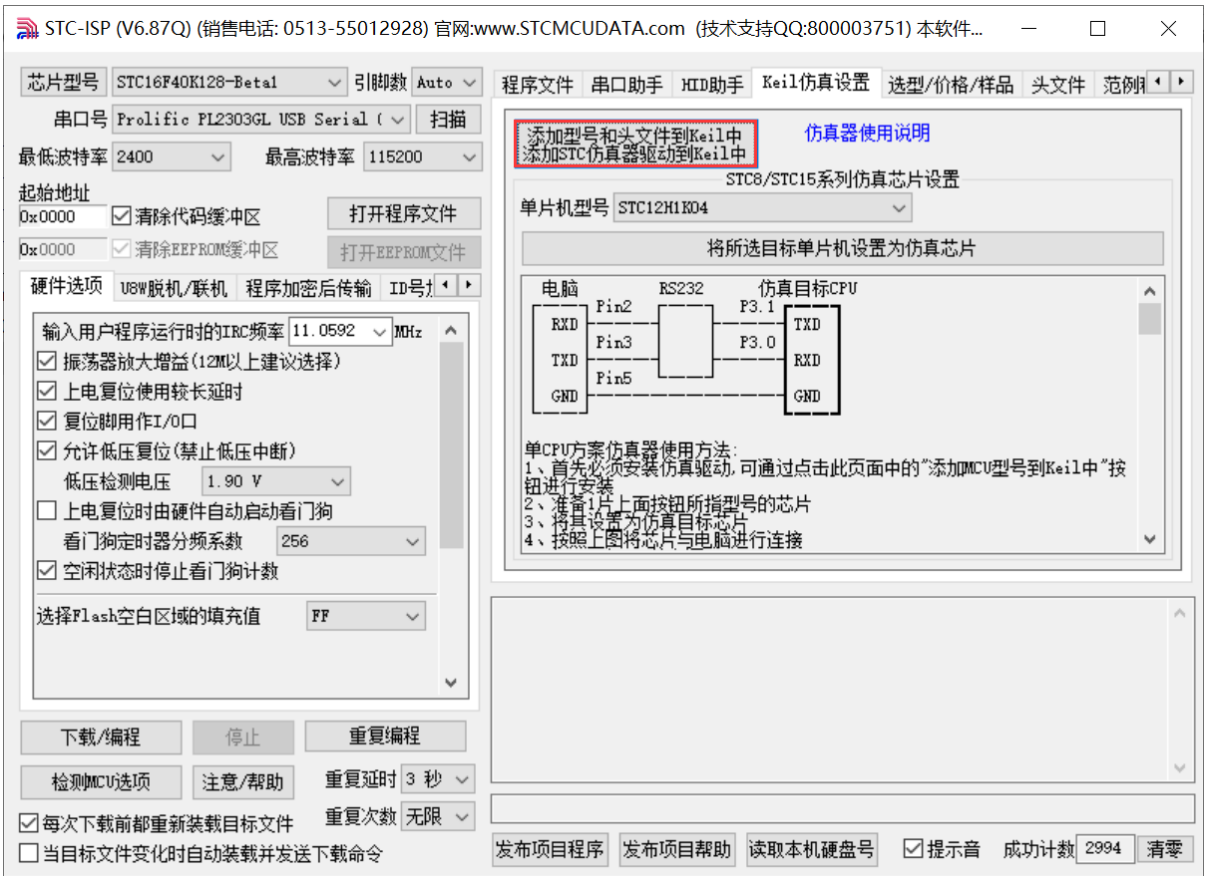
2.4 STC-ISP添加型号和头文件到KEIL

- 官方下载地址：

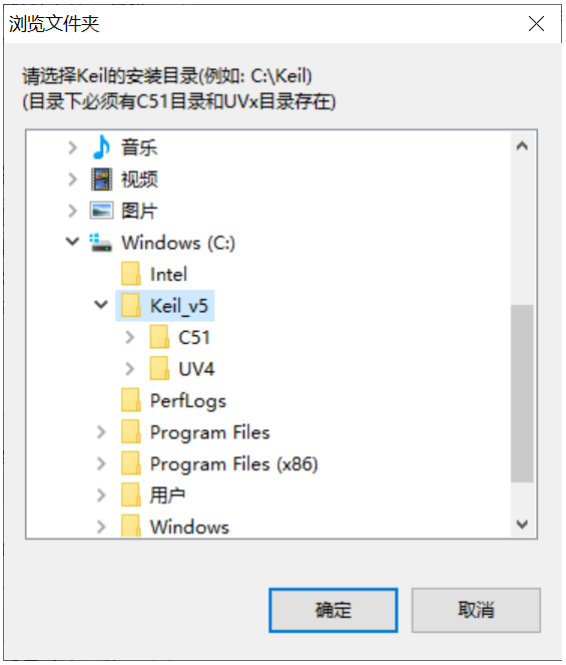
  https://www.stcmcudata.com/

- 使用Keil之前需要先安装 STC 的仿真驱动。STC 的仿真驱动的安装步骤如下：

首先开 STC 的 ISP 下载软件，然后在软件右边功能区的“Keil 仿真设置”页面中点击“添加型号和头文件到 Keil 中 添加 STC 仿真器驱动到 Keil 中”按钮：



按下后会出现如下画面：



将目录定位到Keil软件的安装目录，然后确定。安装成功后会弹出如下的提示框：



在Keil的相关目录中可以看到如下的文件，即表示驱动正确安装了

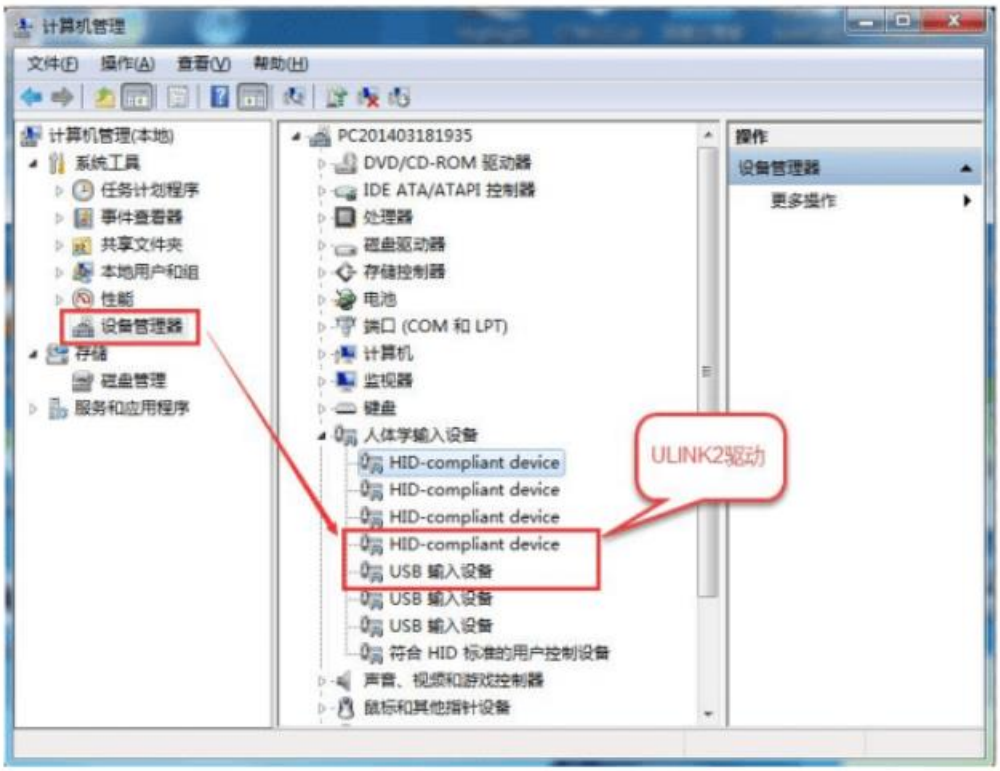
2.5 ULINK2驱动安装

调试下载 STC32M4，推荐使用最新版驱动和对应的固件。

- 为了方便大家下载， 连接如下：
  https://www.keil.com/support/man/docs/ulink2/ulink2_su_driver.htm
- 安装 ULINK2 电脑端驱动：

  下载完毕后，直接安装即可，安装比较简单。

- 安装完毕后，为了验证安装是否成功，看电脑端的设备管理器是否识别出来即可 （要将 ULink 2插到 电脑端）



另外 ，如果驱动没有安装成功，ULINK 2灯会一直闪烁；灯不闪烁了，说明安装成功了。

2.6总结

通过本章节将 STC32 开发环境搭建好之后就可以开启STC32M4 的学习之旅。

第3章 STC32M4工程模板建立（MDK5）

本章教程为大家介绍如何创建 MDK5 工程，配置方法以及经常用到的一些配置项。

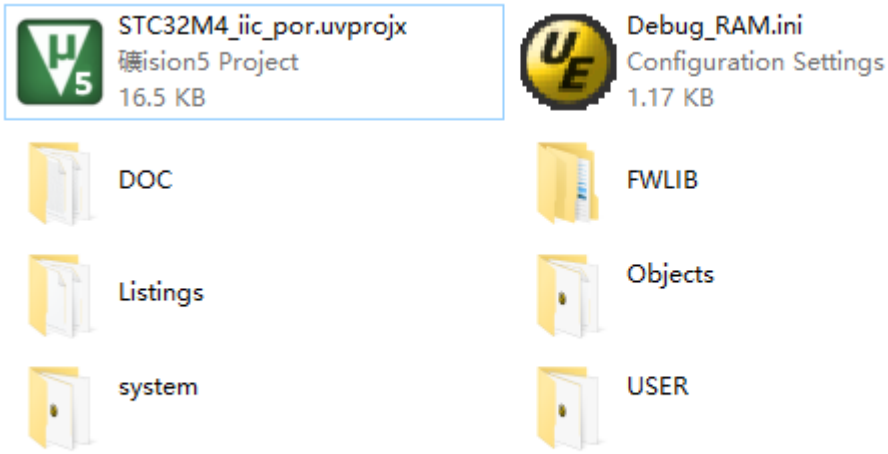
3.1 初学者重要提示
3.2 第 1 步，创建工程文件夹
3.3 第 2 步，创建新工程
3.4 第 3 步，设置工程名和工程分组
3.5 第 4 步，添加工程文件
3.6 第 5 步，Target Options 配置
3.7 第 6 步，最后一步验证是否创建成功
3.8 MDK 的字体，多核并行编译，语法提示等设置
3.9 总结

3.1 初学者重要提示

◆ MDK5 的工程创建过程还是比较重要的，务必实际上手操作一遍。

4.2第 1步， 创建工程文件夹

创建一个文件夹，里面包含如下文件（直接将跑iic例子的 Project 文件里面内容全部删除，然后整 个工程复制进来即可）。



◆ DOC 文件夹

用于记录例程的修改记录和例子简介。

◆ FWLIB 文件夹

这个是库文件夹。

◆ Listing 文件夹

存放编译器编译时候产生得C/汇编/链接的列表清单

◆ Objects 文件夹

存放编译产生的调试信息/hex文件/预览信息等

◆ Project 文件

这个是工程文件 .

◆ User 文件夹

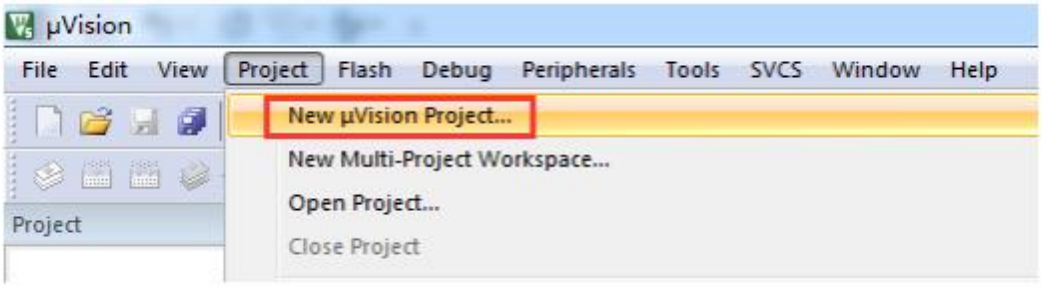
这个是要用户提供的，用户得各驱动程序放在这个文件夹

- Debug_RAM.ini

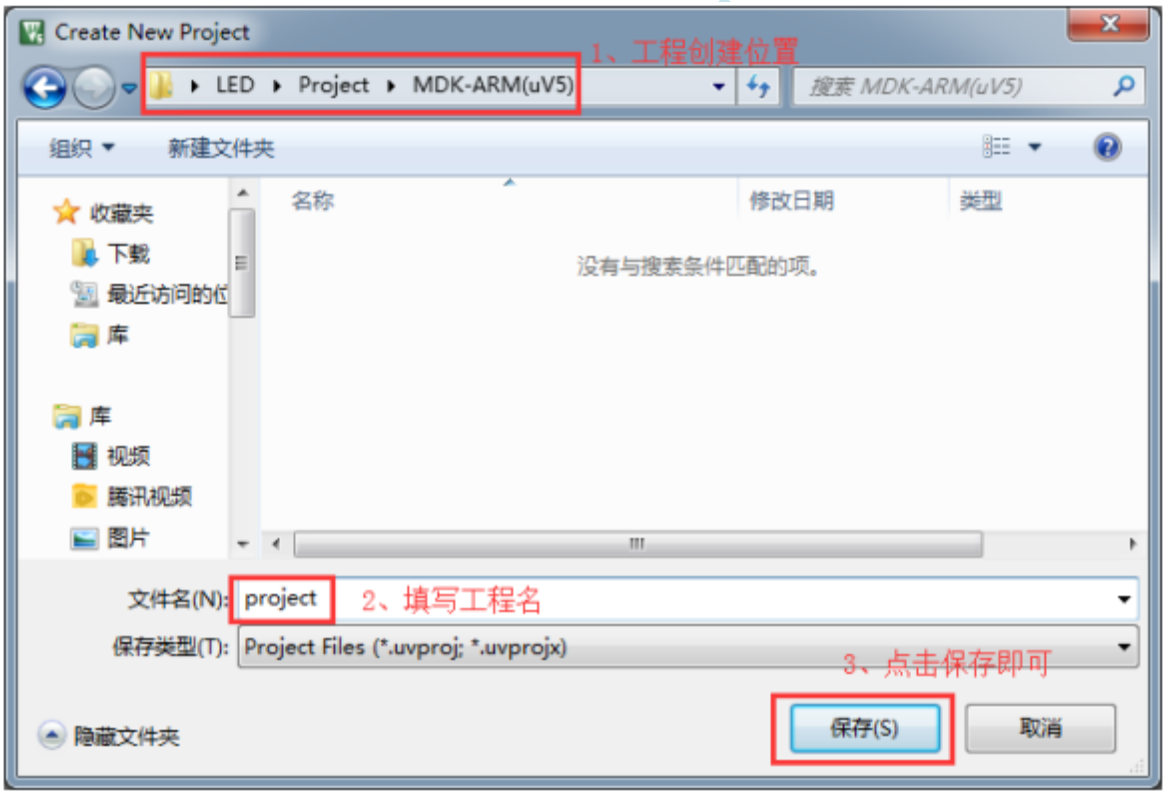
  这个是使用RAM得初始化文件。

3.3第 2步，创建新工程

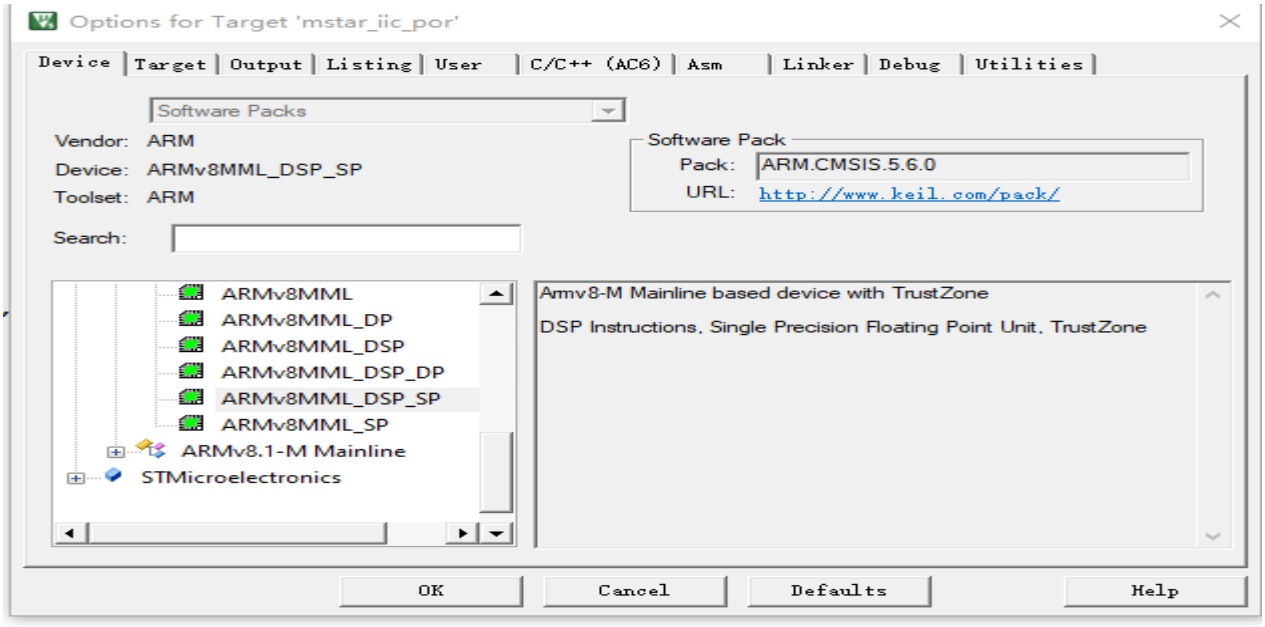
打开 MDK，选择 Project ->New uVision Project:



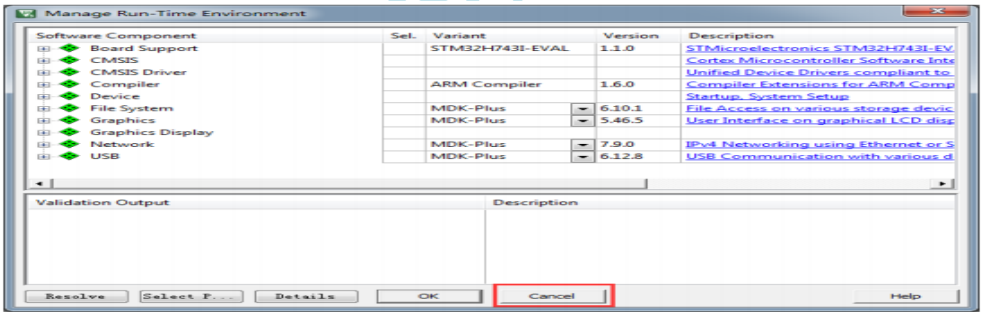
工程名设置为 project:



选择器件: ARMv8MML_DSP_SP

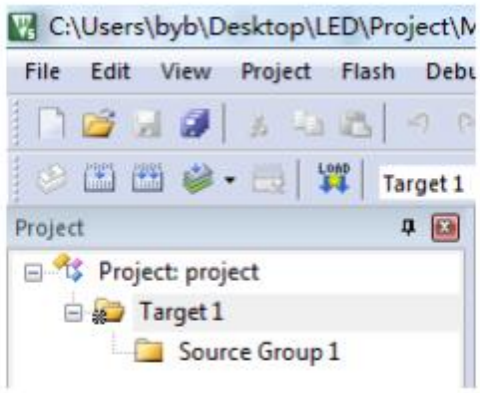


点击 OK 后弹出如下界面，点击 Cancel 取消，当前我们还不使用 RTE 环境创建，

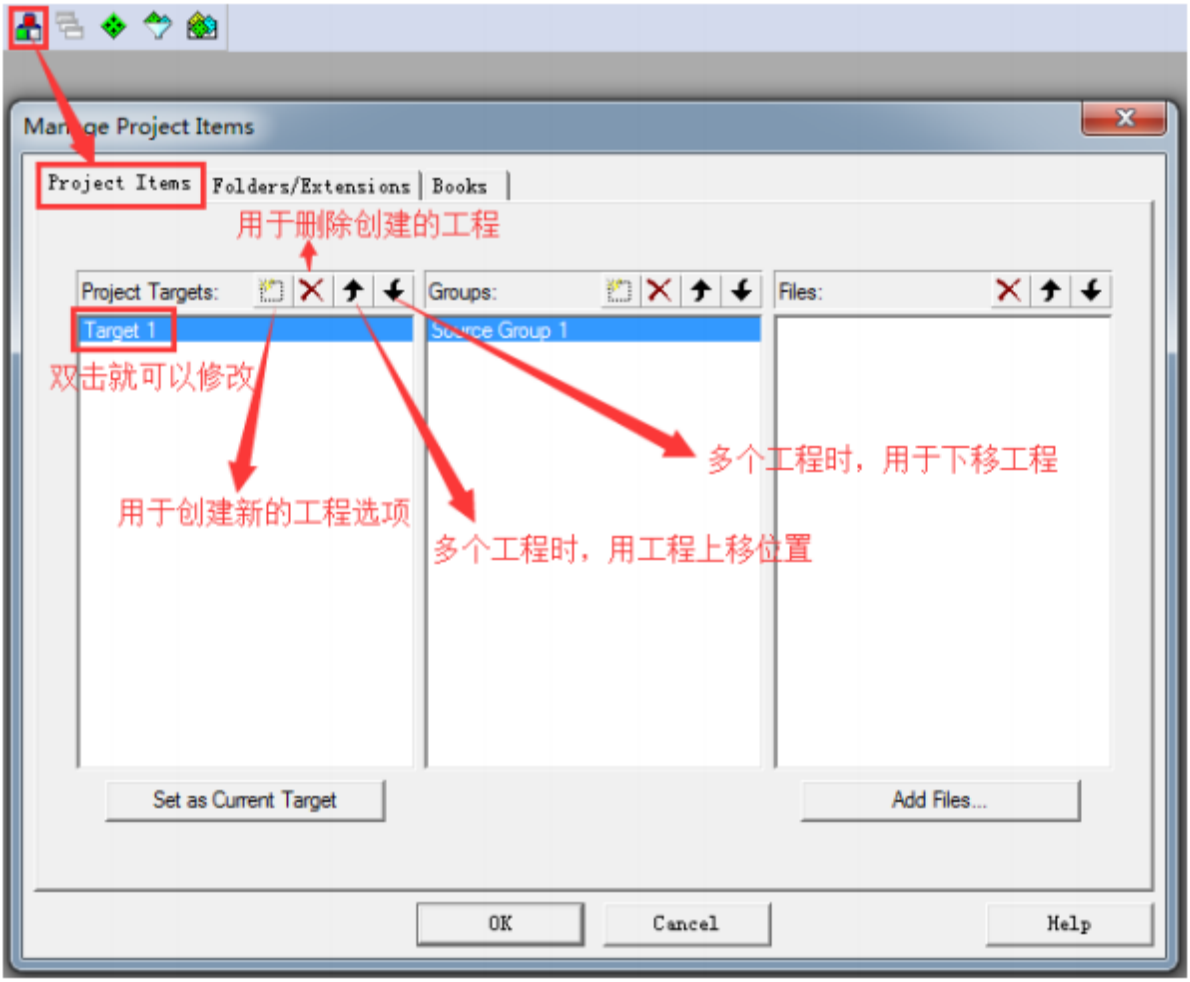


3.4 第 3步，设置工程名和工程分组名

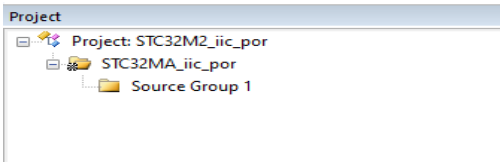
默认新建的工程是下面这样的:



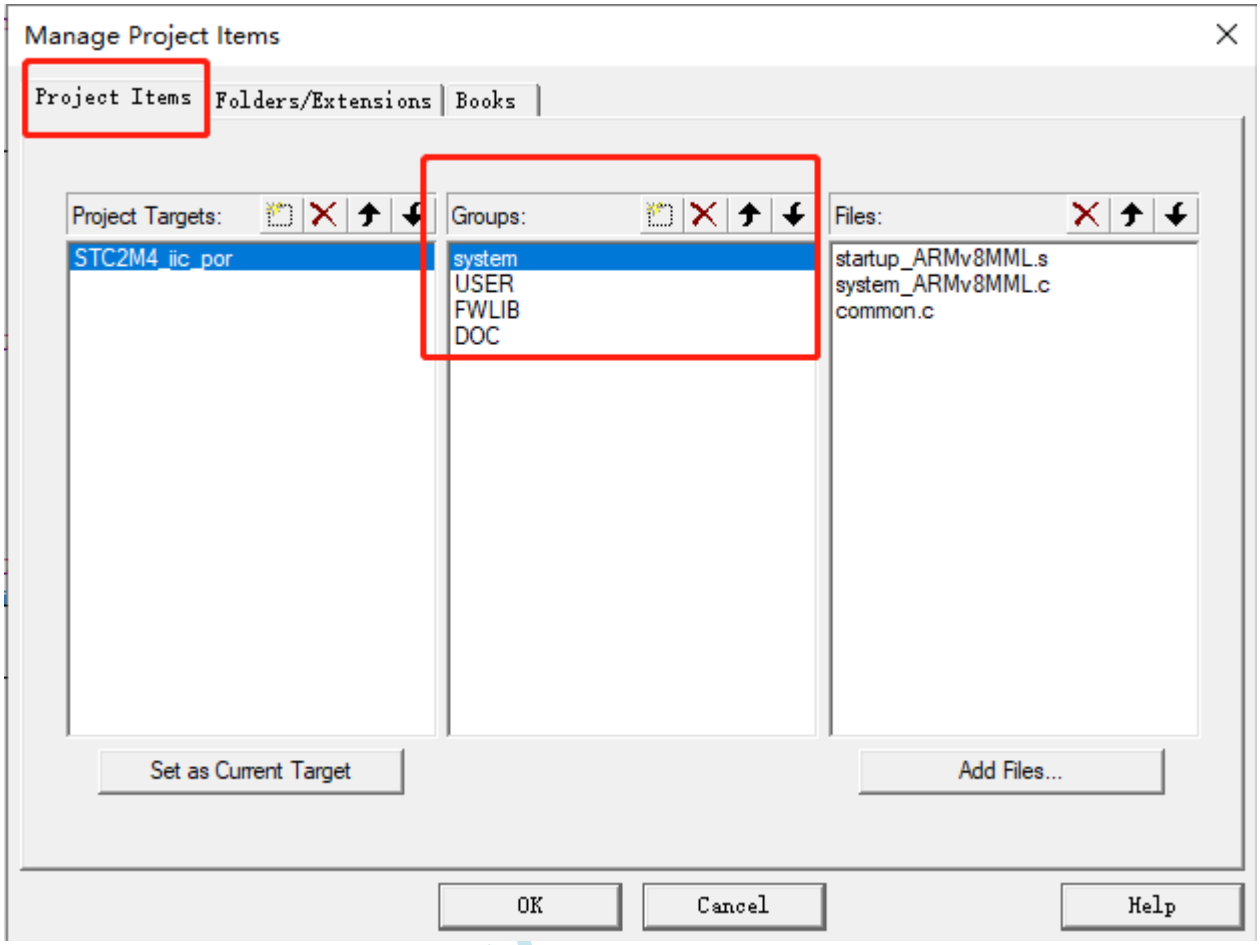
打开 Manage Project Items，双击 Targer1修改名字为 STC32M4_iic_por，或者任何其它方便自己管理的名称，设置完毕后点击 OK。



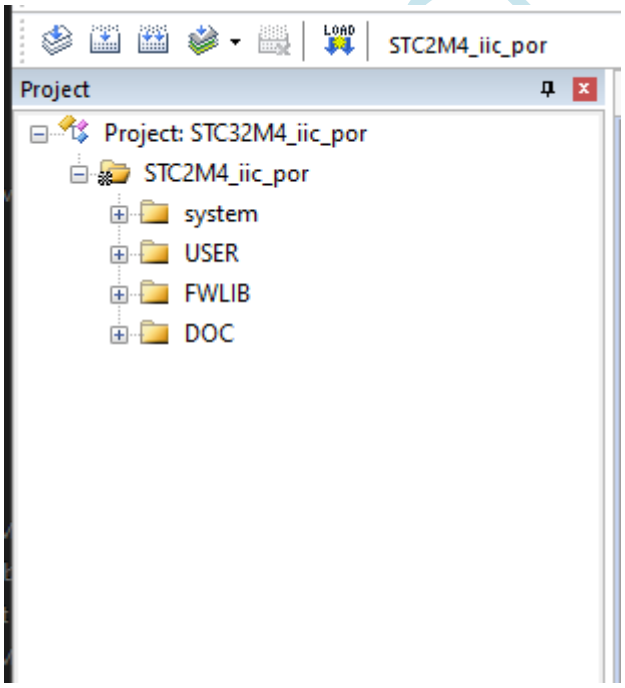
可以看到已经修改为STC32M4_iic_por:



再次打开 Manage Project Items，添加如下分组内容:

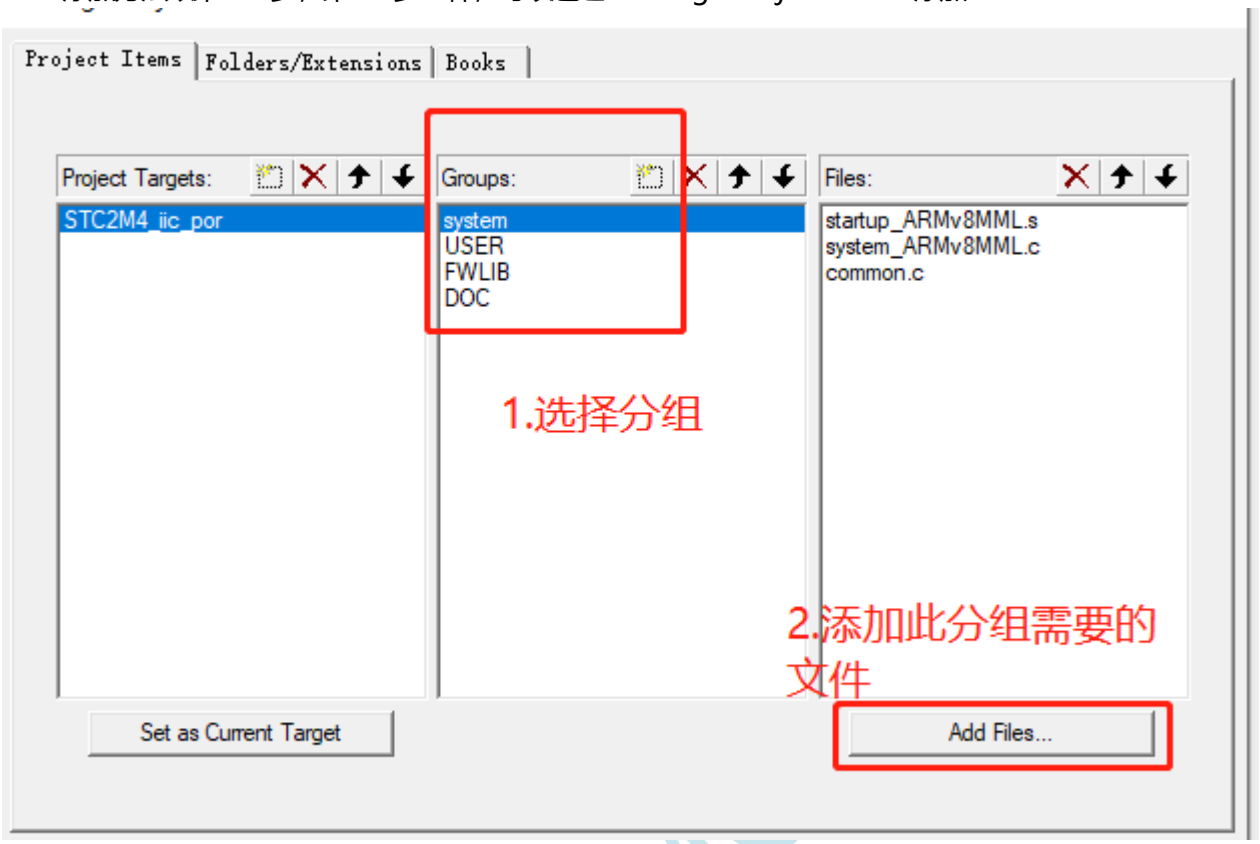


可以看到左侧工程列表已经更新:

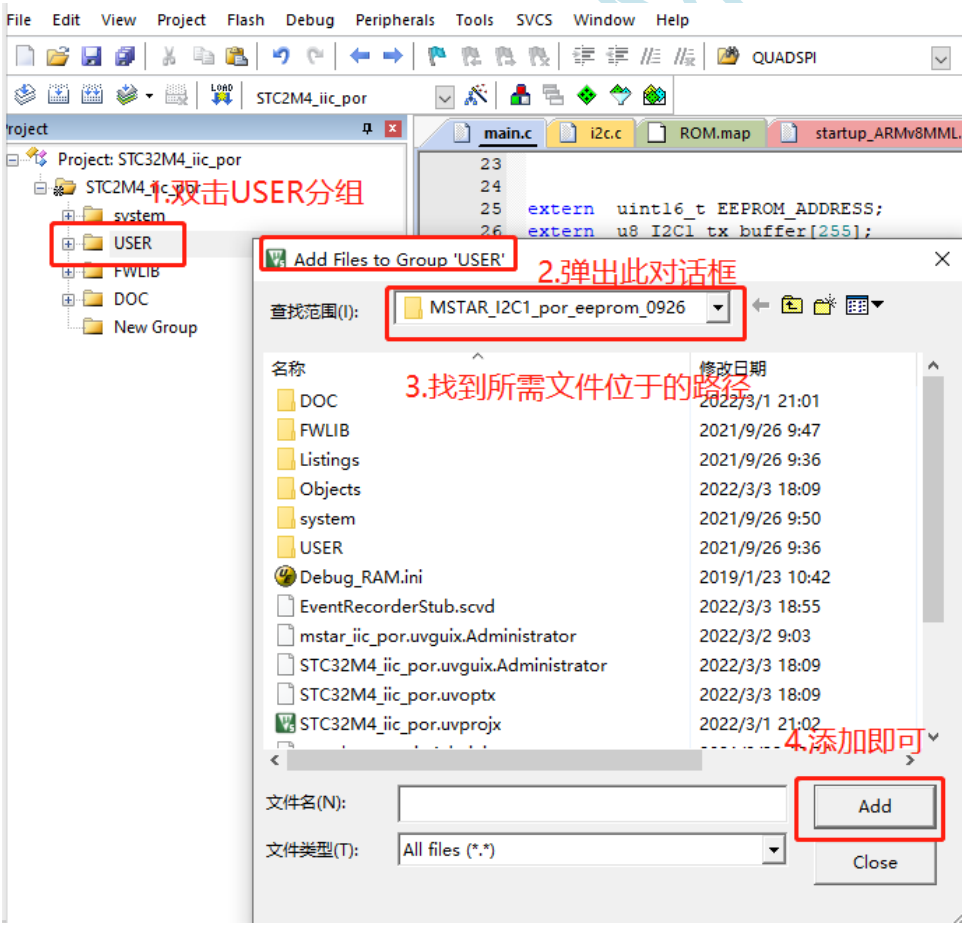


3.5 第4步，添加工程文件

添加方法跟第 2 步，第 3 步一样，可以通过 Manage Project Items 添加:



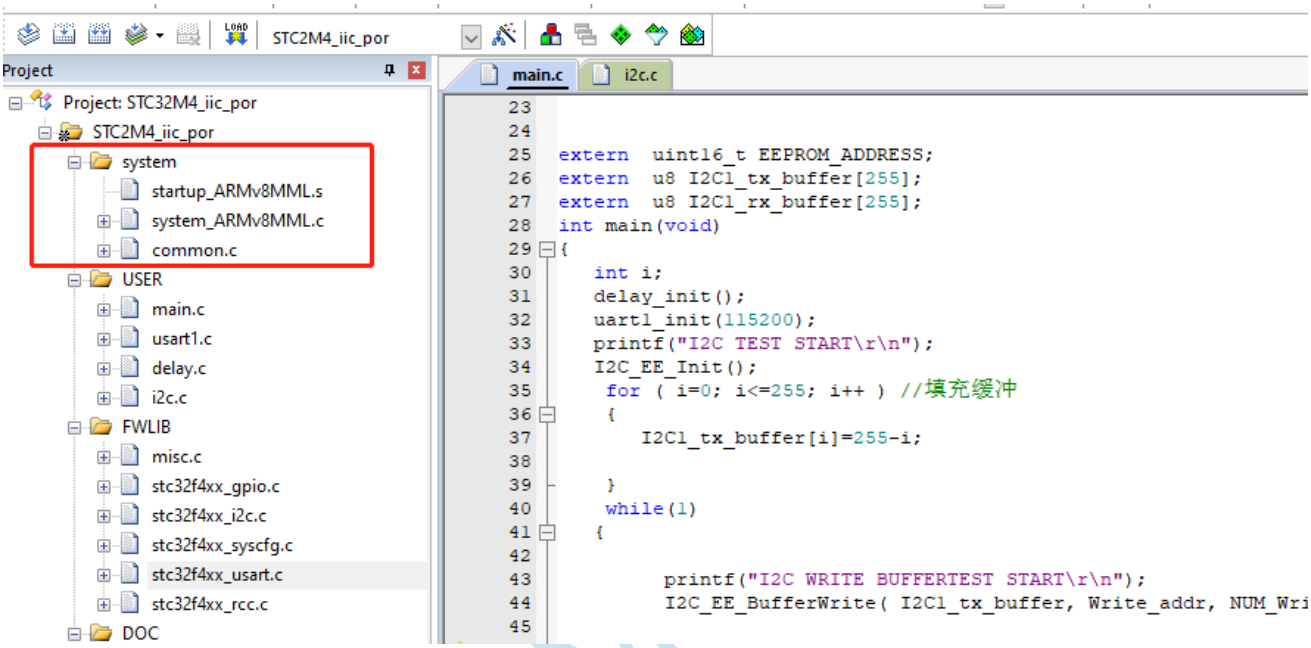
也可以直接双击工程列表里面的分组添加 （注意文件类型里面推荐设置为 ALL files 所有文件）：



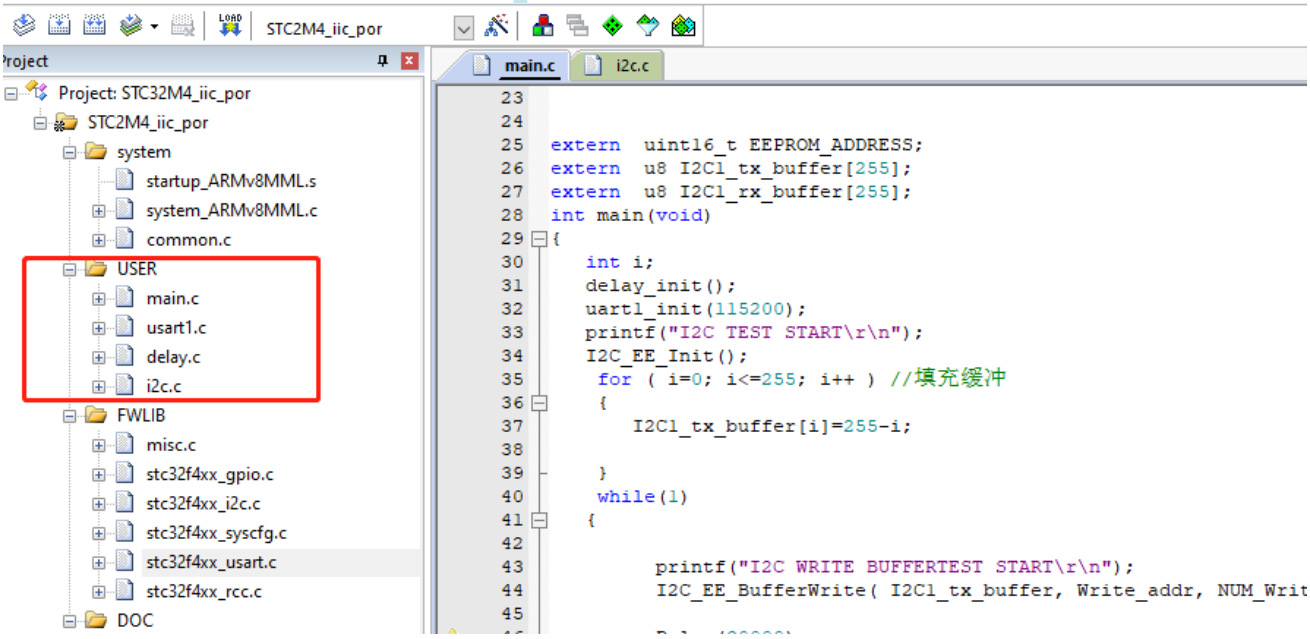
- 添加 system分组需要的文件

system_ARMv8MML.c 位于路径system，主要用 于复位 RCC 相关寄存器和中断向量表位置设置。

Startup_ARMv8MML.s位于路径system，主要用 启动文件的配置

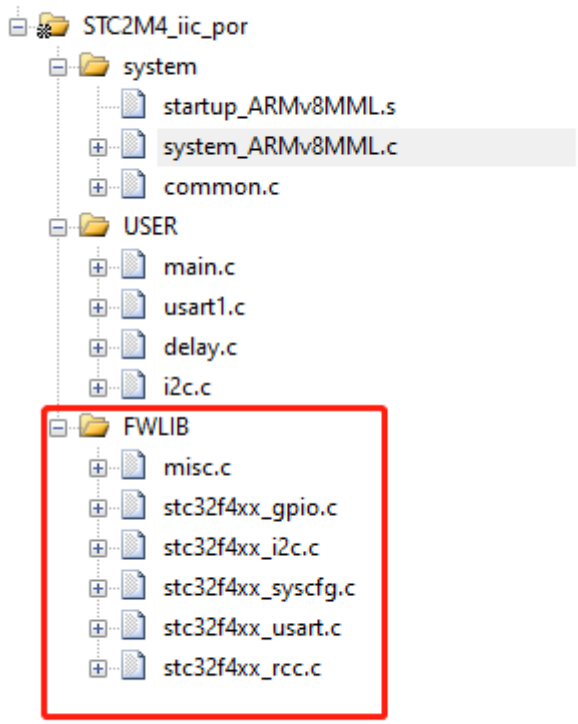


- 添加 USER 分组需要的内容



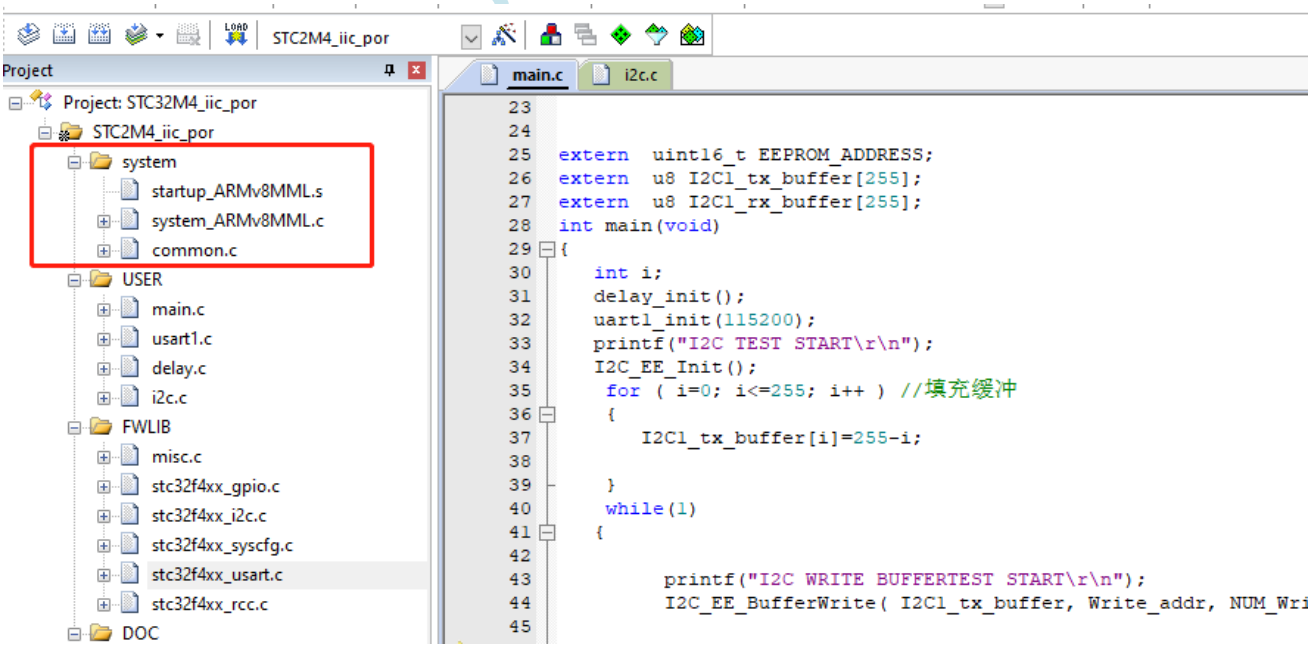
用户添加的各驱动函数

- 添加 FWLIB 分组需要的内容

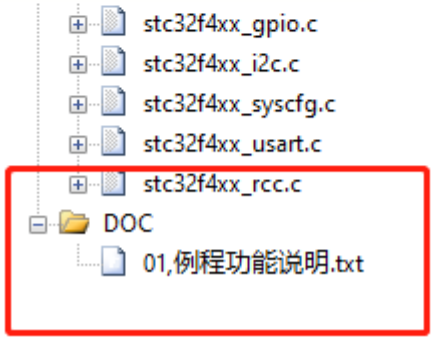


STC MCU

- 添加 SYSTEM 分组需要的内容



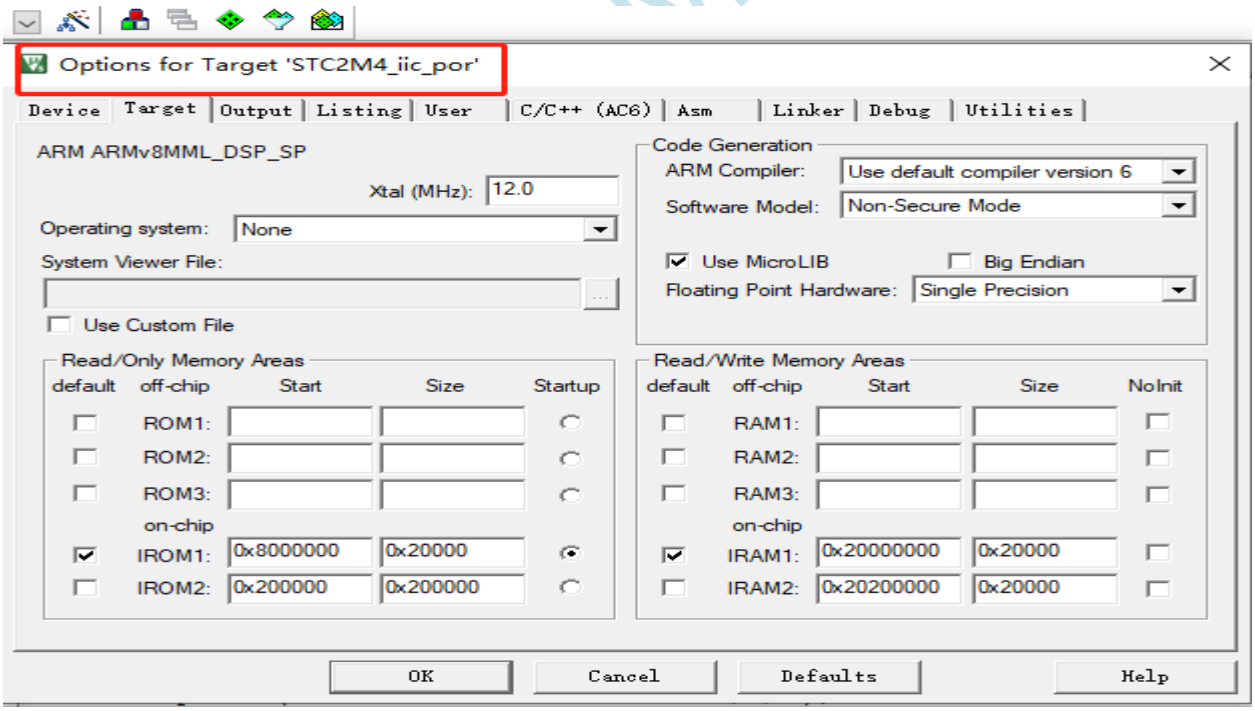
- 添加 Doc 分组需要的内容



这两个文件位于 Doc 文件夹，主要用于说明例程的功能和更新记录。

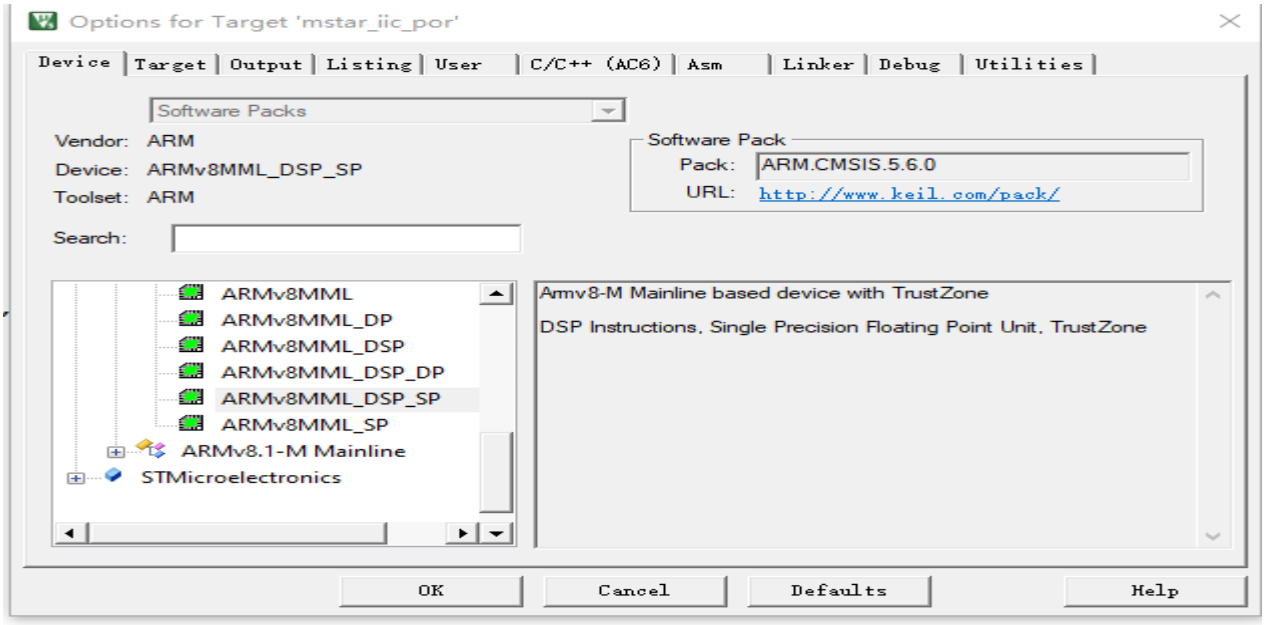
3.6 第 5步，Target Options配置

MDK 的配置基本都集中在 Target Options 对话框里面:

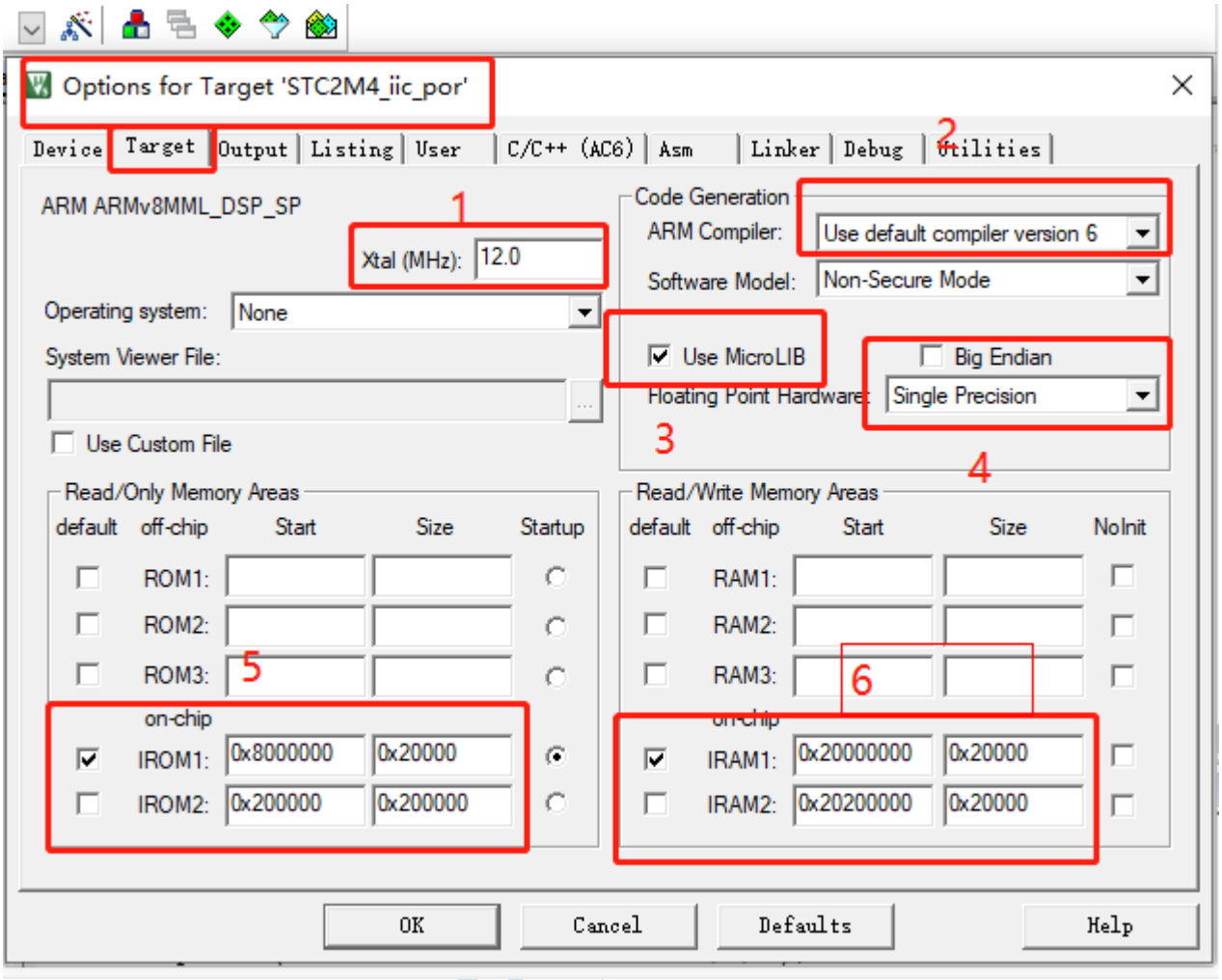


3.6.1Device 选项

这个选项不用管，因为我们在创建的时候已经选择了 ARMv8MML_DSP_SP



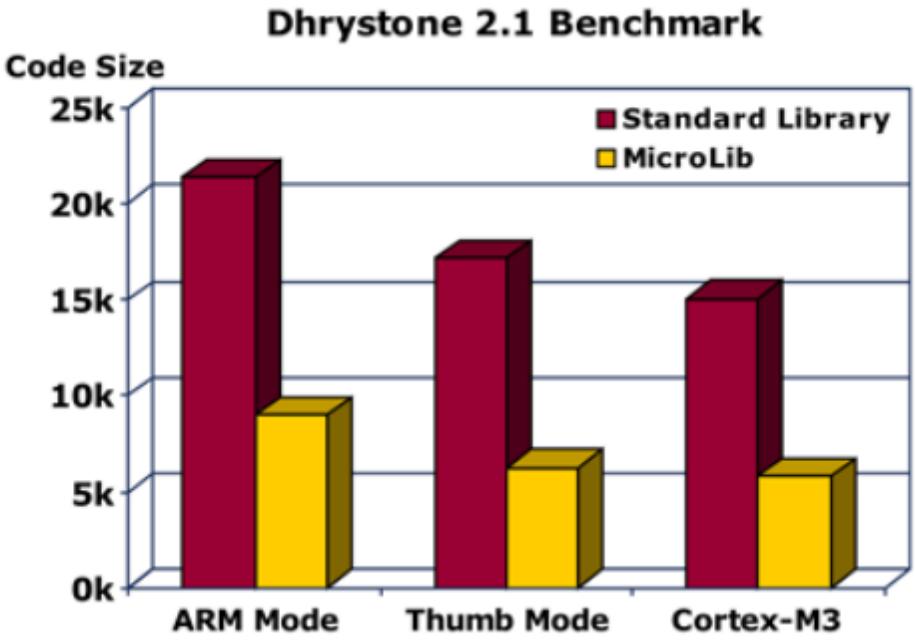
3.6.2Target 选项



1. XTAL(MHZ) :晶振12MHZ；因为我们一般不用软件仿真，这个选项不用管，也不推荐大家用MDK提供的软件仿真，问题便较多， 不用管。

2. 选择编译器，使用默认的 use default compiler version 6 即可。

3. 我们这里使用微库 MicroLib，这个库针对嵌入式应用，MicroLIB 做了深度优化，比使用 C 标准库所 需的 RAM 和 FLASH 空间都大大减小。 比如调用: <math.h>，<stdlib.h>，<stdio.h>，<string.h>。

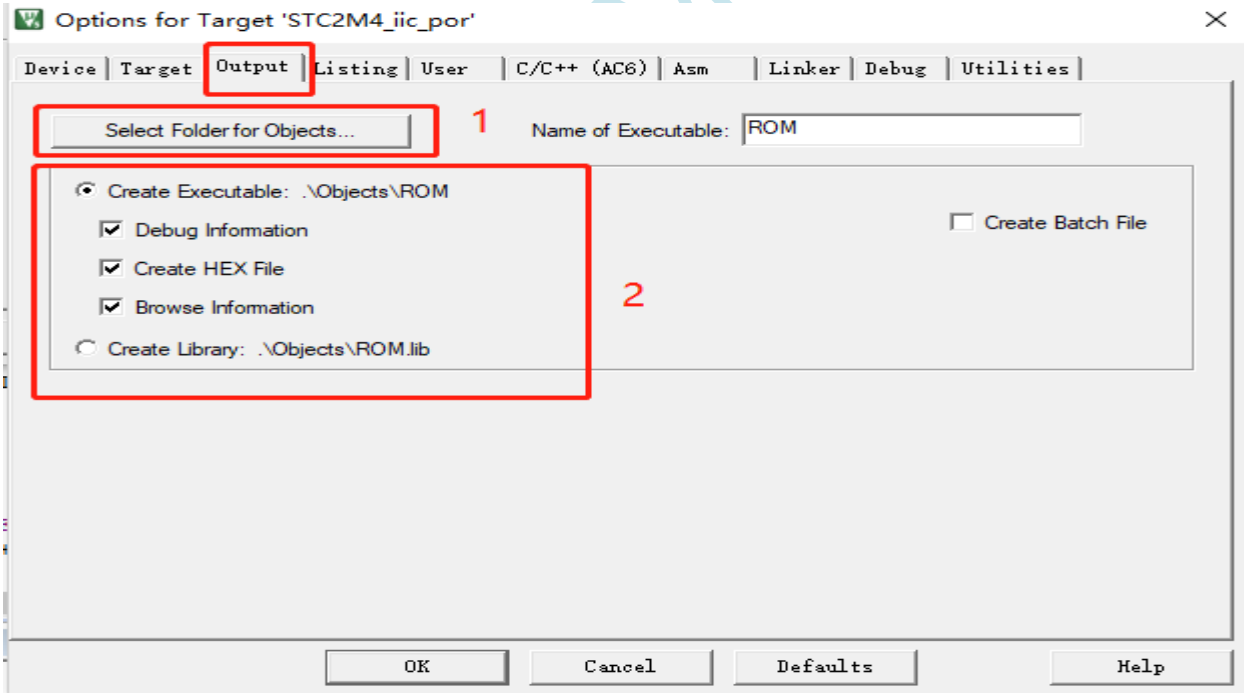


更多这方面的信息可以看 KEIL 官网： http://www.keil.com/arm/microlib.asp 。

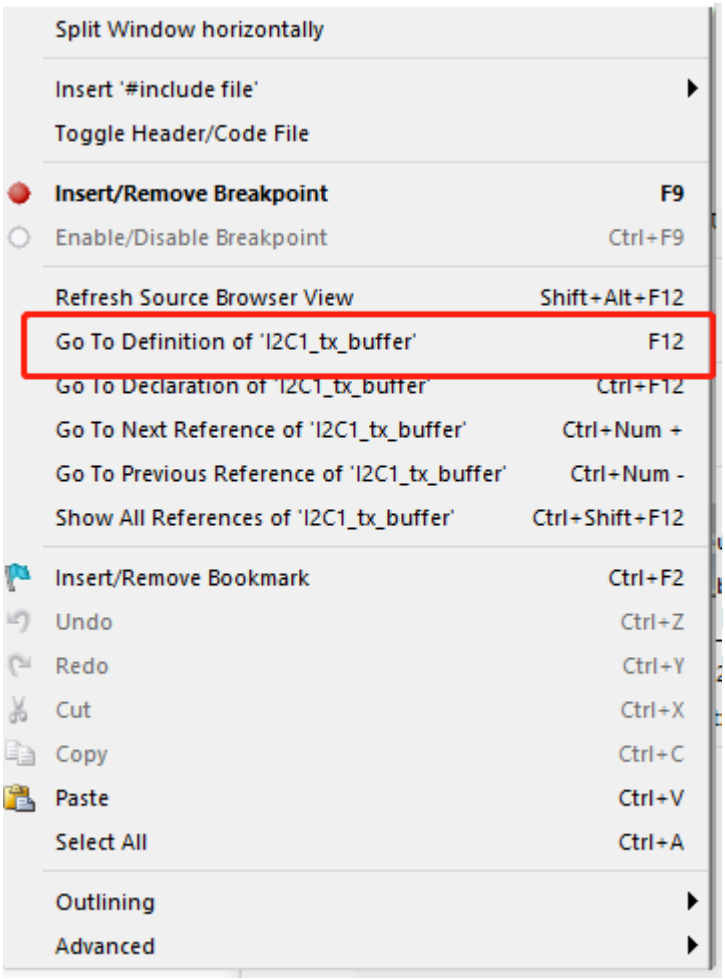
4.STC32M4 支持单精度浮点，所以这里选择 Single Precision 单精度浮点。

5.STC32M4 的 Flash 首地址是 0x0800 0000，大小 0x200000，即 2MB。

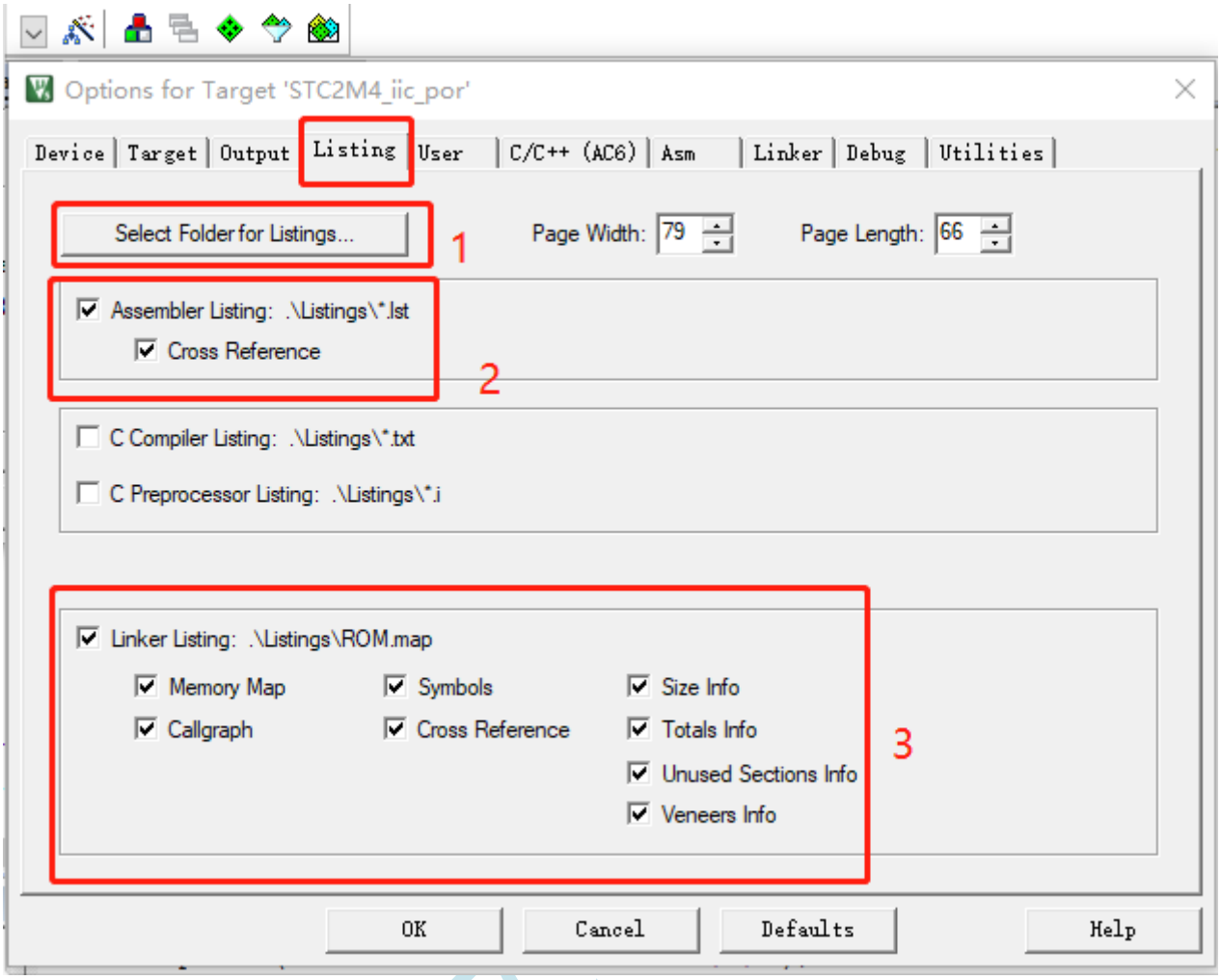
3.6.3Output 选项



1. 用于存放 MDK 编译过程中生成的 obj临时文件路径， MDK 在创建的时候已经帮我们创建好文件夹 Objects，这里无需修改，使用默认即可。 2.Debug Information 表示调试信息，选上。3.Create HEX File 表示是否生成 hex 文件，选上。4.Browse Information 表示是需要 go to def 的支持，使用 HAL 库话，推荐先不要选上，因为 MDK5 编译时间比较长.



3.6.4Listing 选项

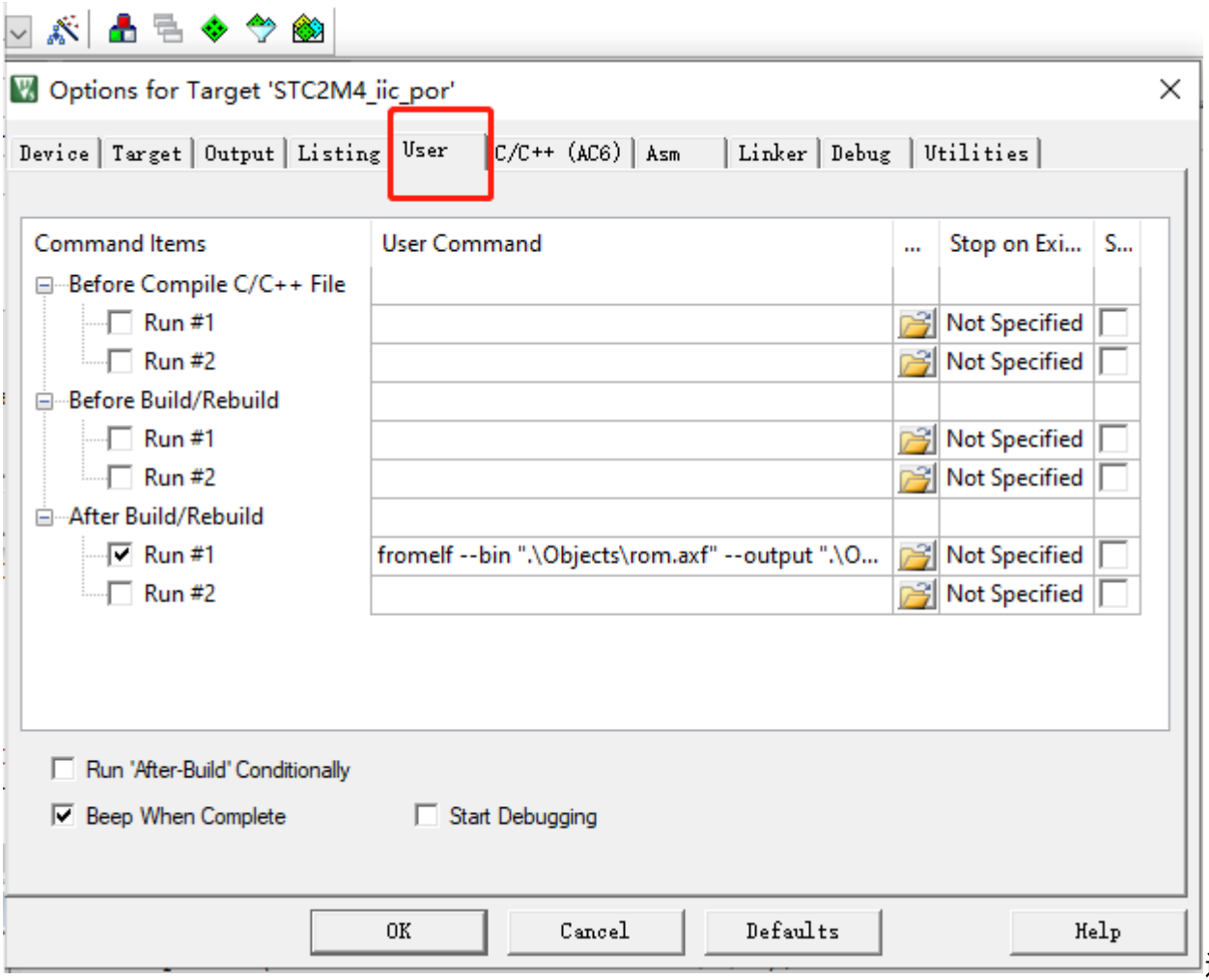


1.用于存放 MDK 编译过程中生成的 list 临时文件路径， MDK 在创建的时候已经帮我们创建好文件夹 Listings，这里无需修改，使用默认即可。

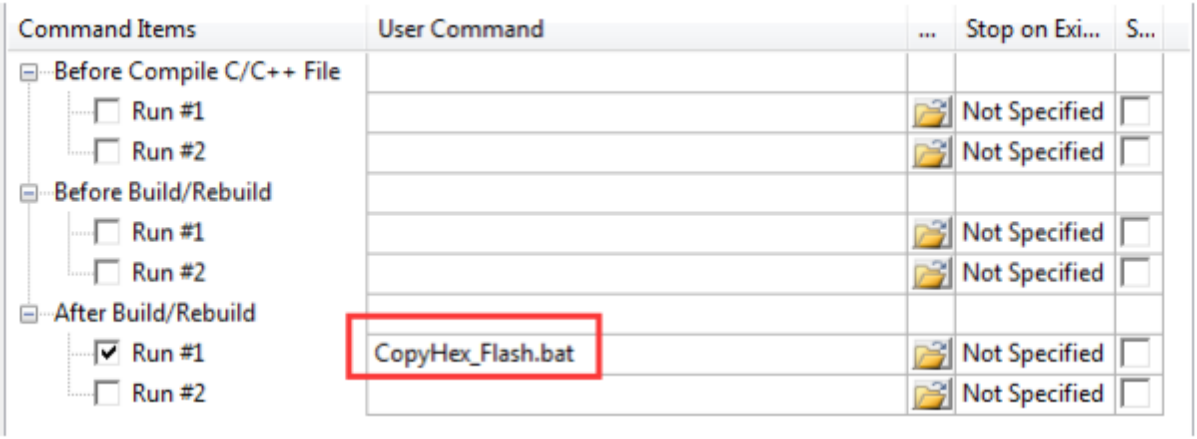
2. 是否生成*.lst 文件，选上。

3. 查看 MAP 文件要用上，全部选上。

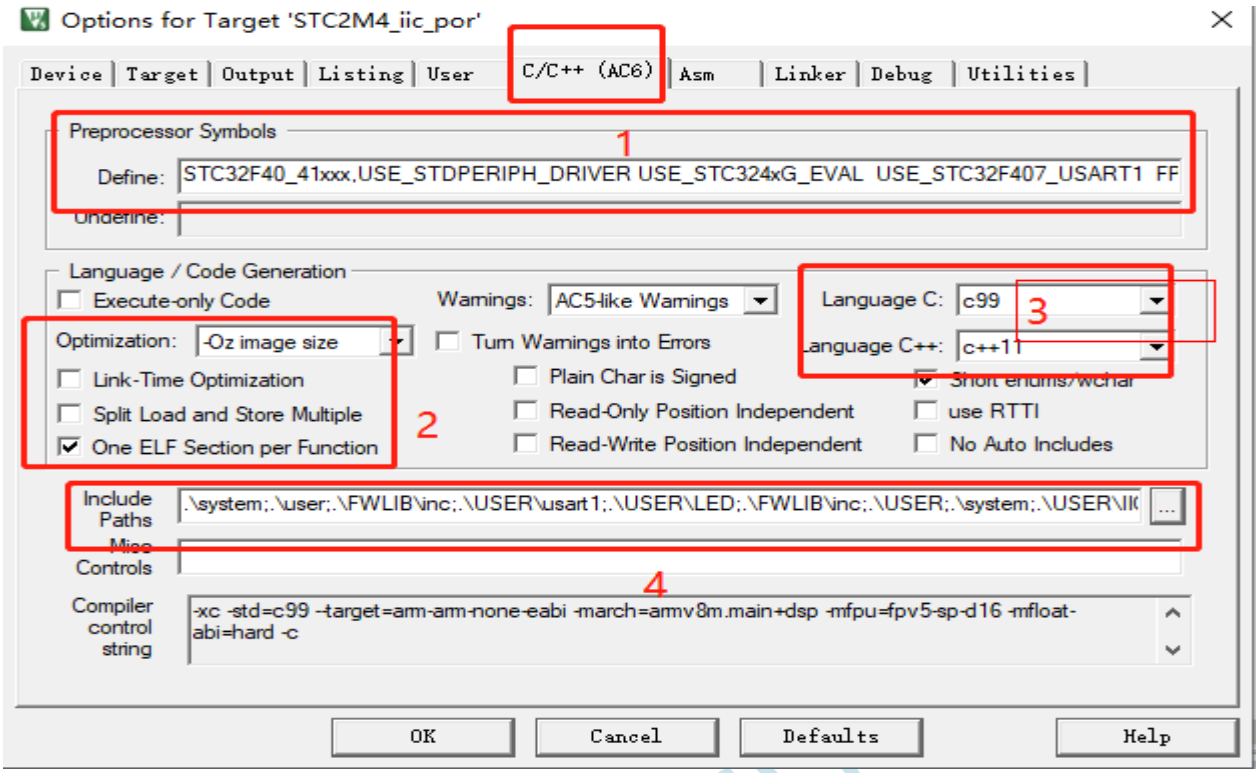
3.6.5User 选项



这个选项基本用不上，可以不用管，主要是用来指定编译之前和编译之后执行的用户程序。对于本教程配套的例子来说在这里添加了一个批处理功能，用来实现 Objects 文件夹里面的 hex 文件复制到路径 Project\MDK-ARM(uV5)里面，方便查找。



3.6.6C/C++选项



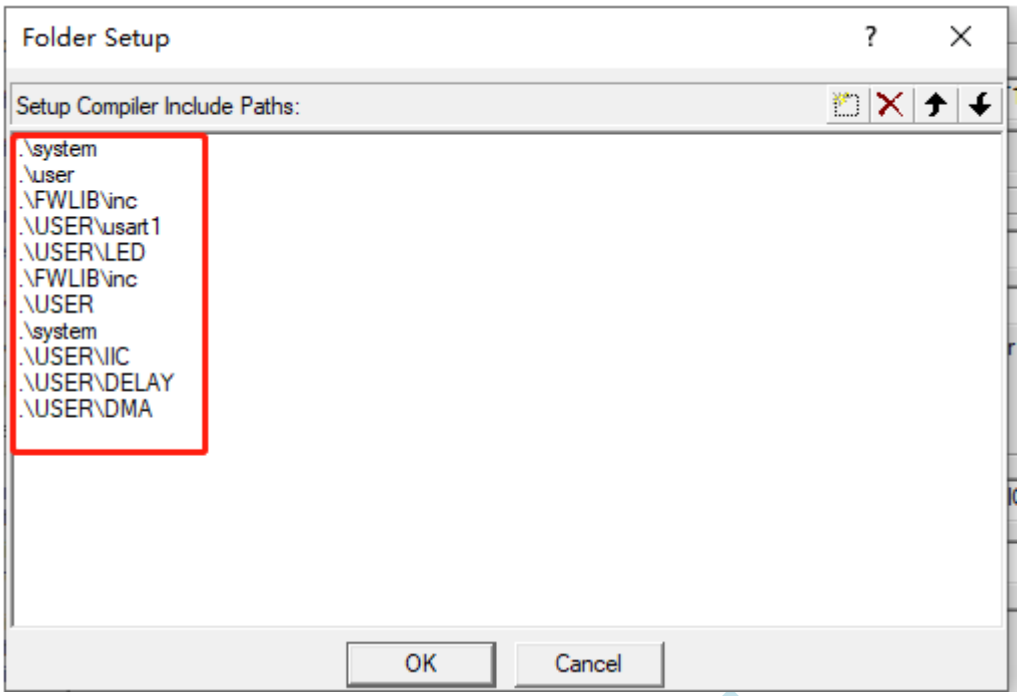
1. 预定义宏

   USE_STDPERIPH_DRIVER USE_STC324xG_EVAL表示使用标准库。

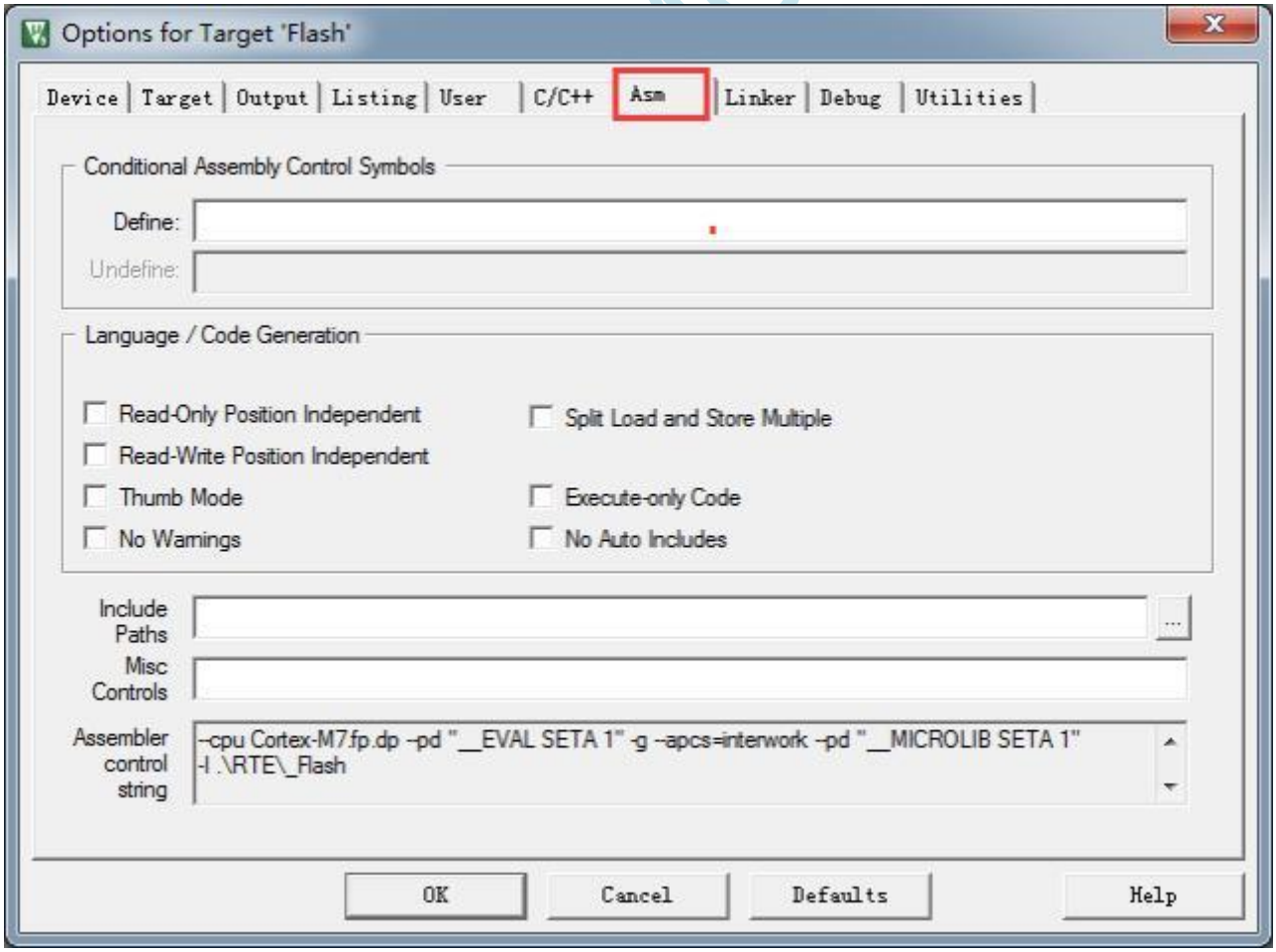
   STC32F4　表示使用 STC32F4 系列。

2. 选择 0 级优化，优化等级低，方便调试。

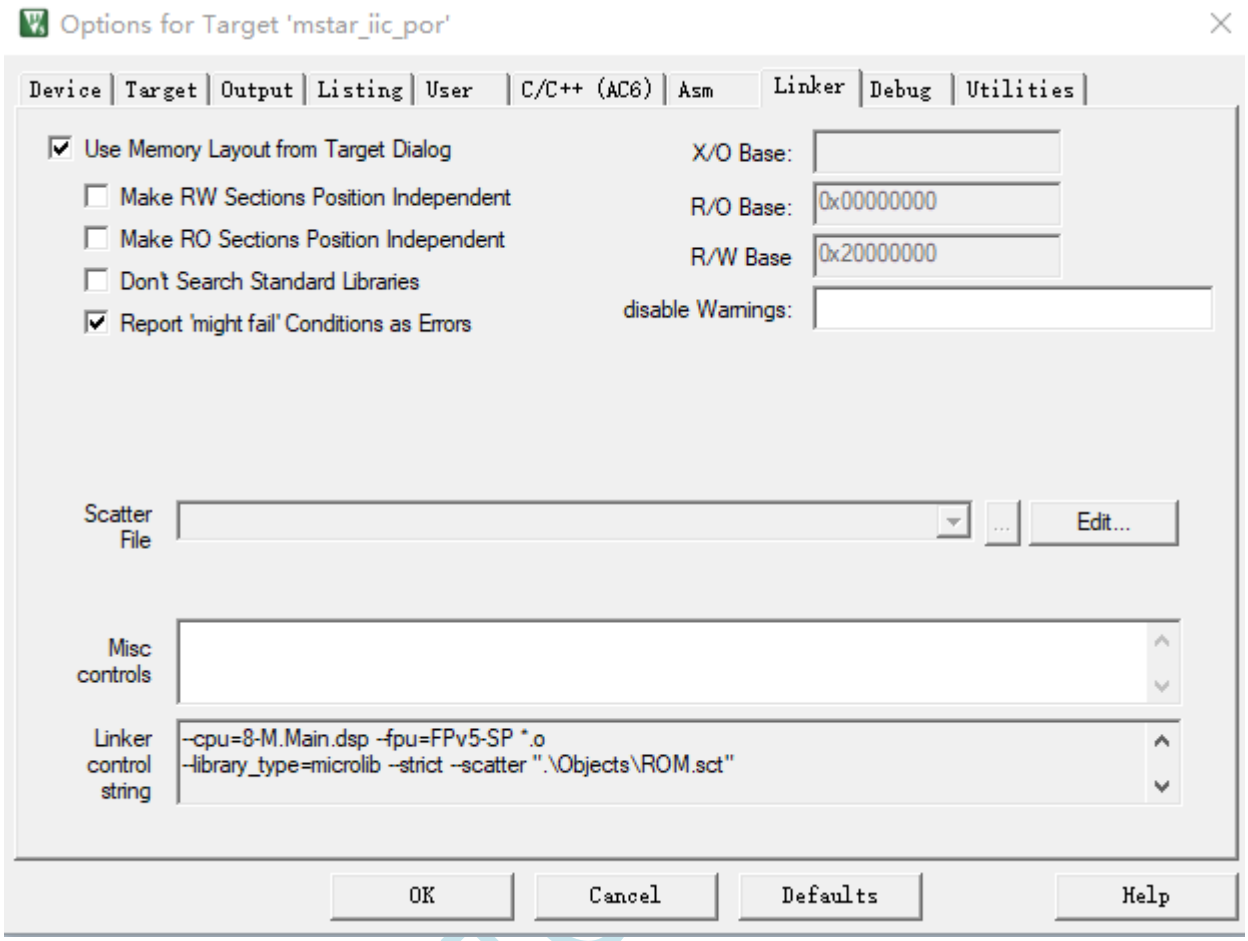
选项 One ELF Section per Function 用于删除工程中没有用到的冗余函数。3. 使用 C99 规范，在第 5 章有介绍。 4. 添加路径，把用到的头文件路径包含进来，点击最右侧的小按钮，弹出如下界面进行添加即可：


Folder Setup
Setup Compiler Include Paths:
.\system
.\user
.\FWLIB\inc
.\USER\usart1
.\USER\LED
.\FWLIB\inc
.\USER
.\system
.\USER\IIC
.\USER\DELAY
.\USER\DMA
OK
Cancel

3.6.7Asm 选项

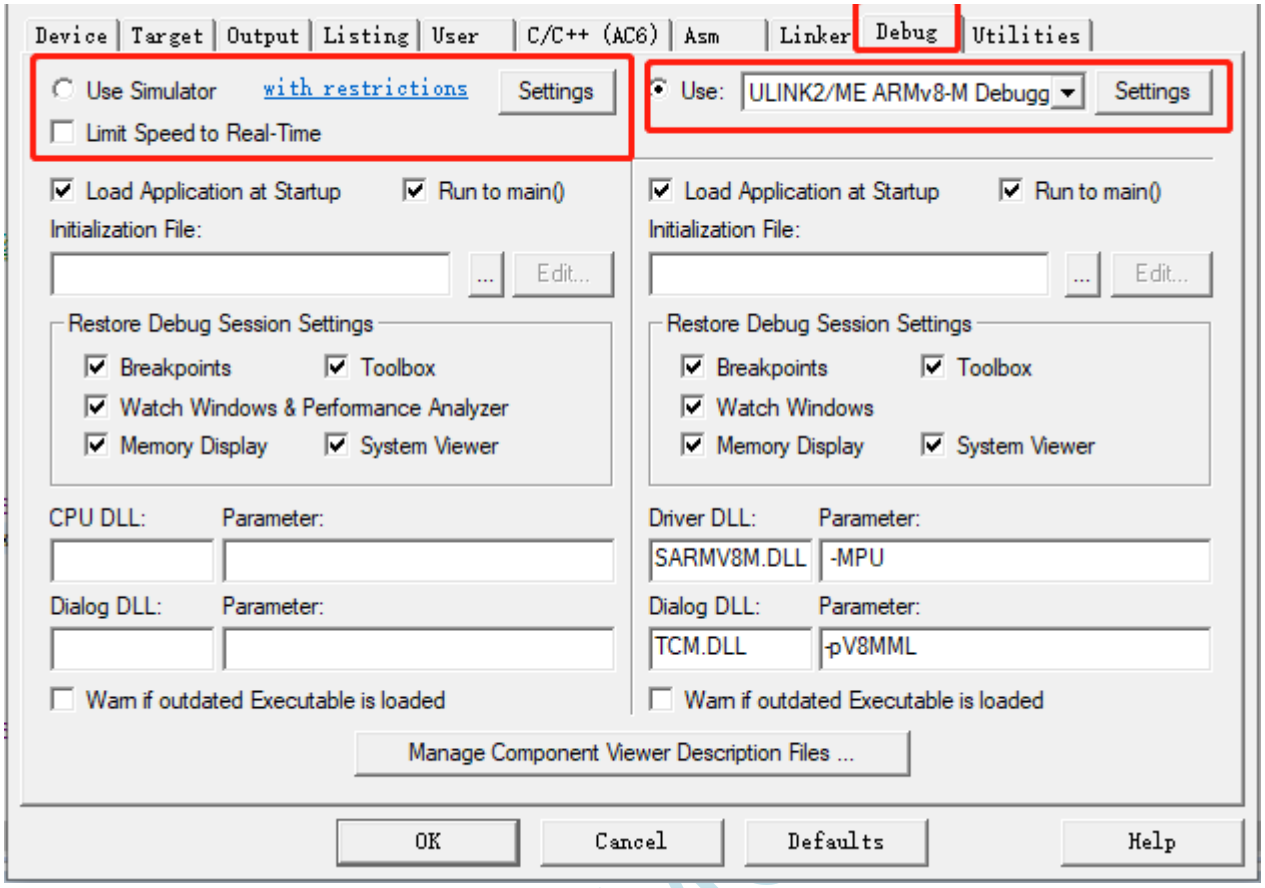

Options for Target 'Flash'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Conditional Assembly Control Symbols
Define:
Undefine:
Language / Code Generation
Read-Only Position Independent
Split Load and Store Multiple
Read-Write Position Independent
Thumb Mode
Execute-only Code
No Warnings
No Auto Includes
Include Paths
Misc Controls
Assembler control string
--cpu Cortex-M7.fp.dp --pd "__EVAL SETA 1" -g --apcs=interwork --pd "__MICROLIB SETA 1"
-I .\RTE_Flash
OK
Cancel
Defaults
Help

3.6.7Linker 选项



这一页不用选

3.6.8Debug 选项

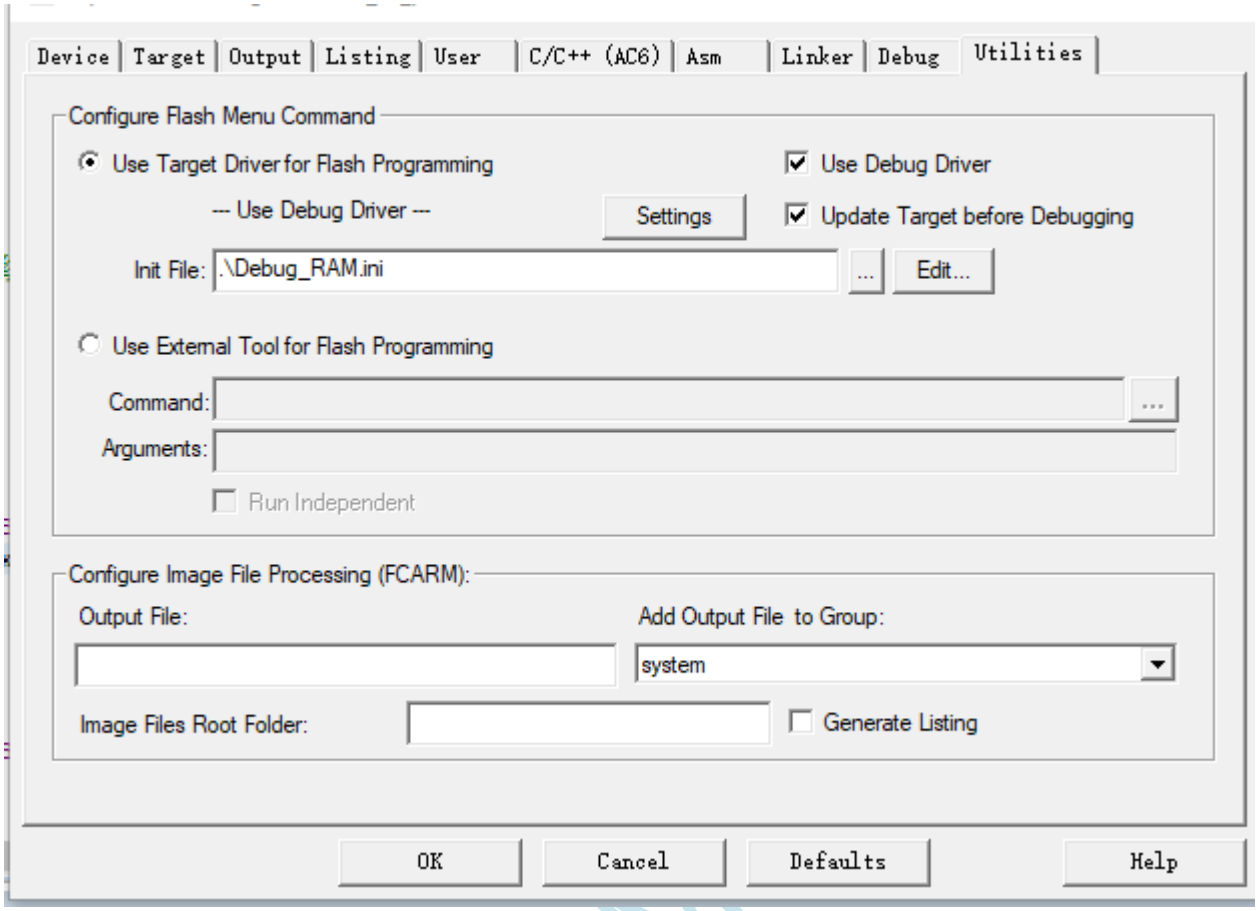


1. 软件仿真不推荐大家用，问题比较多。
2. 硬件仿真，这里选择的 ULINK 2，大家根据自己用的那种调试器选择相应型号即可。

具体的调试设置在下一章有详细说明，本章主要是保证创建的工程能够正常编译。

3.6.9Utilities 选项

这个选项主要用于编程工具选择，大家根据自己用的那种下载器选择相应型号即可。Update Target before Debugging 表示调试前更新芯片的程序。



3.7 第 6步，最后一步验证是否创建成功

点击全编译：编译结果如下0错误，0警告

```
compiling usart1.c...
compiling main.c...
compiling rtc.c...
compiling stc32f4xx_iwdg.c...
compiling stc32f4xx_gpio.c...
compiling stc32f4xx_exti.c...
compiling common.c...
compiling stc32f4xx_usart.c...
compiling stc32f4xx_rcc.c...
assembling startup_ARMv8MML.s...
compiling system_ARMv8MML.c...
linking...
Program Size: Code=2780 RO-data=488 RW-data=8 ZI-data=1040
FromELF: creating hex file...
After Build - User command #1: fromelf --bin ".\Objects\rom.axf" --output ".\Objects\rom.bin"
".\Objects\rom.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:02
```

3.8 MDK 的字体，多核并行编译，语法提示等设置

这里将 MDK 经常用到的设置做个说明。

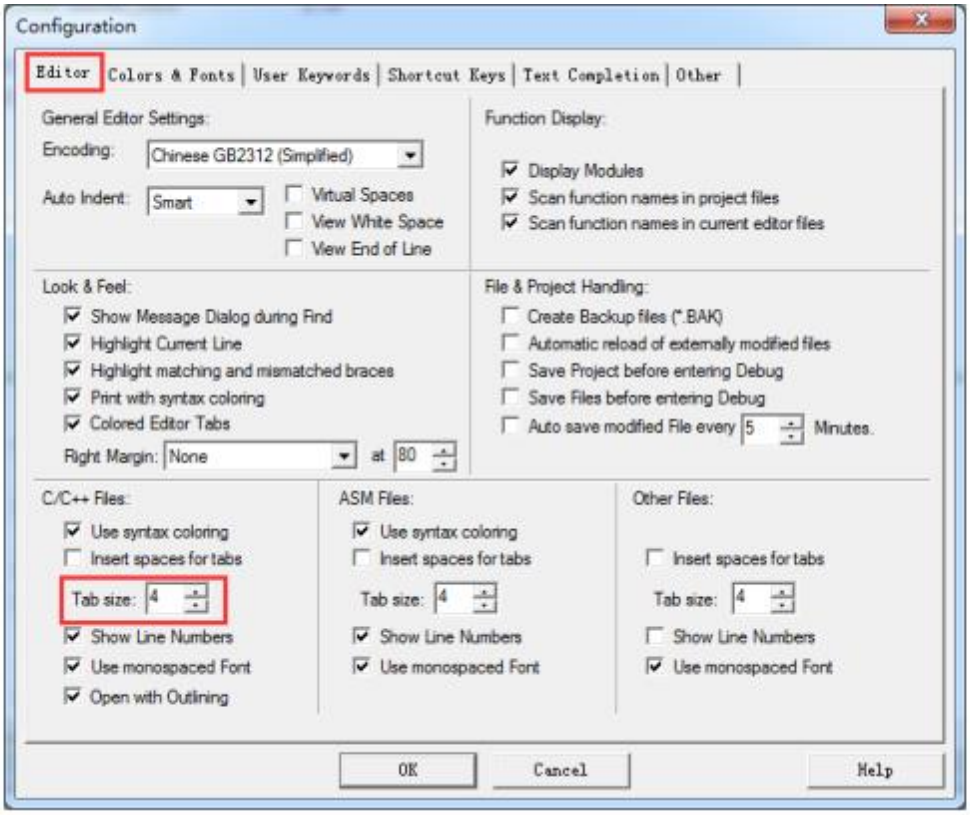
3.8.1Tab Size 设置

教程配套例子的 Tab size 要设置为 4，否则代码显示不整齐：

选择此图标：



弹出的窗口里面设置 Tab size 为 4：



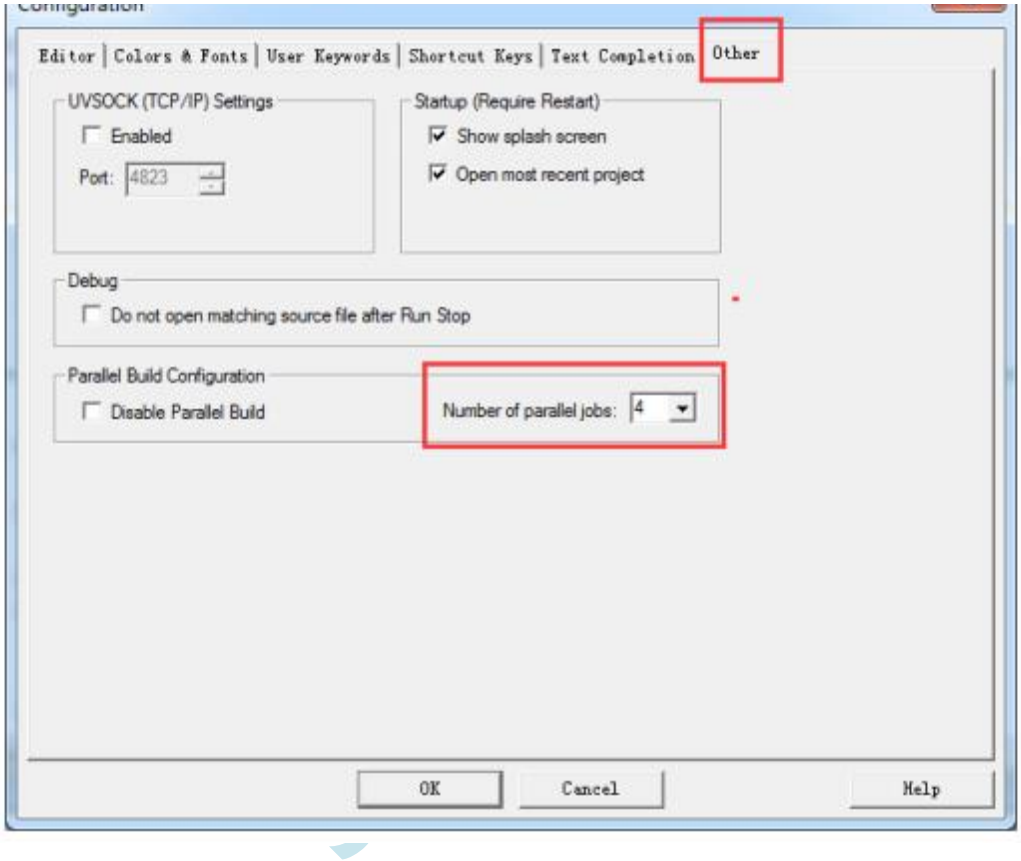
3.8.2使能多核并行编译

电脑有几个核, 都选上, 加快编译速度。 选择此

图标:



我这里是四核:



3.8.3字体设置

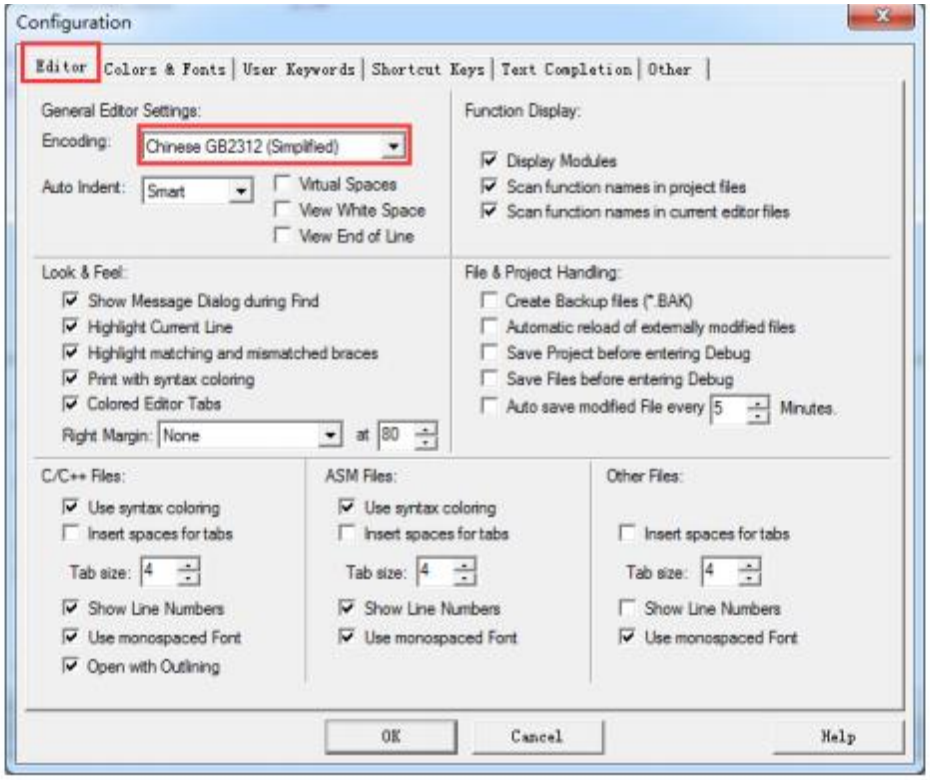
推荐大家用下微软雅黑，而且可以防止复制粘贴的时候出现乱码问题。

安装微软雅黑的时候要注意，电脑端默认自带了一个微软雅黑，安装的时候会出现字体是躺着的。

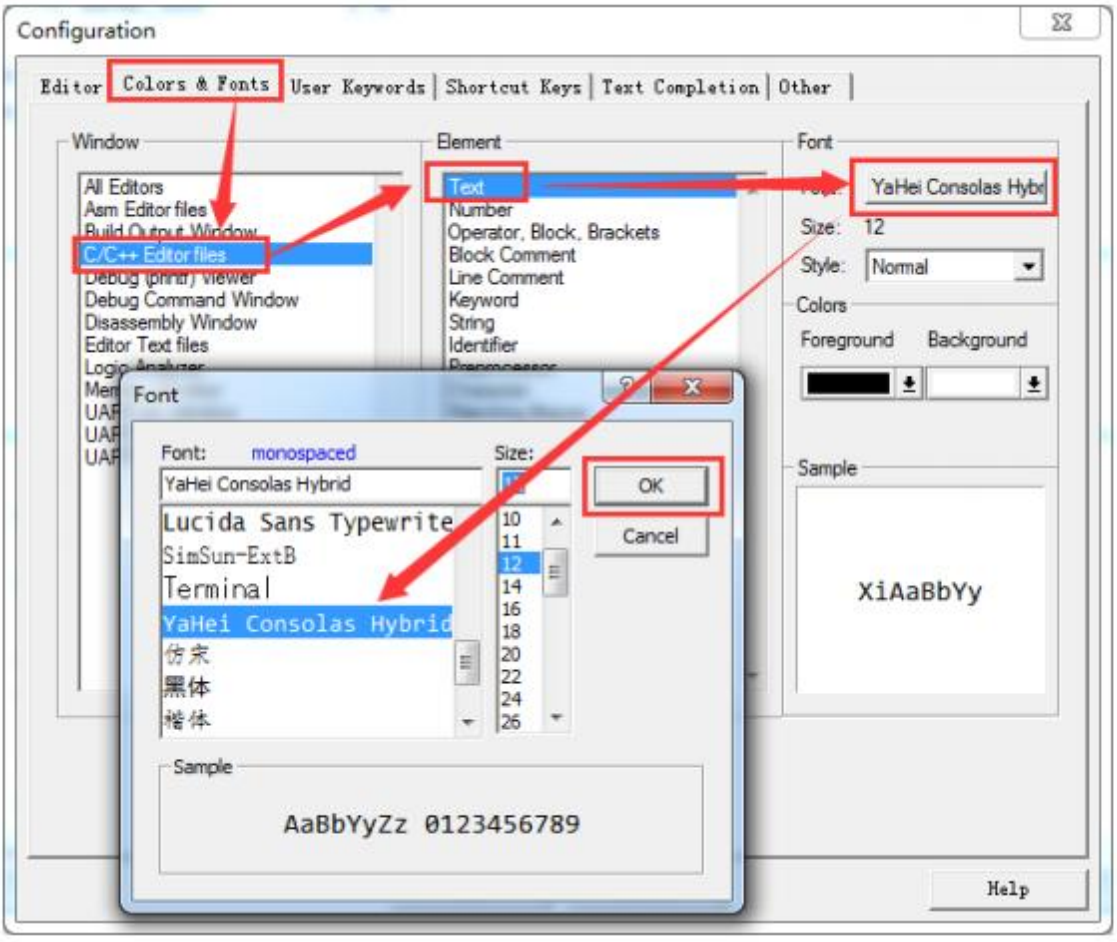
选择此图标:



选择中文简体:


Configuration
Editor
Colors & Fonts
User Keywords
Shortcut Keys
Text Completion
Other
General Editor Settings:
Encoding:
Chinese GB2312 (Simplified)
Auto Indent:
Smart
Virtual Spaces
View White Space
View End of Line
Function Display:
Display Modules
Scan function names in project files
Scan function names in current editor files
Look & Feel:
Show Message Dialog during Find
Highlight Current Line
Highlight matching and mismatched braces
Print with syntax coloring
Colored Editor Tabs
Right Margin:
None
at
80
File & Project Handling:
Create Backup files (*.BAK)
Automatic reload of externally modified files
Save Project before entering Debug
Save Files before entering Debug
Auto save modified File every
5
Minutes.
C/C++ Files:
Use syntax coloring
Insert spaces for tabs
Tab size:
4
Show Line Numbers
Use monospaced Font
Open with Outlining
ASM Files:
Use syntax coloring
Insert spaces for tabs
Tab size:
4
Show Line Numbers
Use monospaced Font
Other Files:
Insert spaces for tabs
Tab size:
4
Show Line Numbers
Use monospaced Font
OK
Cancel
Help

选择安装微软雅黑字体:


Configuration
Editor
Colors & Fonts
User Keywords
Shortcut Keys
Text Completion
Other
Window
All Editors
Asm Editor files
C/C++ Editor files
Debug Command Window
Disassembly Window
Editor Text files
Element
Text
Number
Operator, Block, Brackets
Block Comment
Line Comment
Keyword
String
Identifier
Font
YaHei Consolas Hybr
Size:
12
Style:
Normal
Colors
Foreground
Background
Sample
XiAaBbYy
Font
Font:
monospaced
YaHei Consolas Hybrid
Lucida Sans Typewrite
SimSun-ExtB
Terminal
YaHei Consolas Hybrid
仿宋
黑体
楷体
Size:
10
11
12
14
16
18
20
22
24
26
OK
Cancel
Sample
AaBbYyZz 0123456789
Help

效果如下:

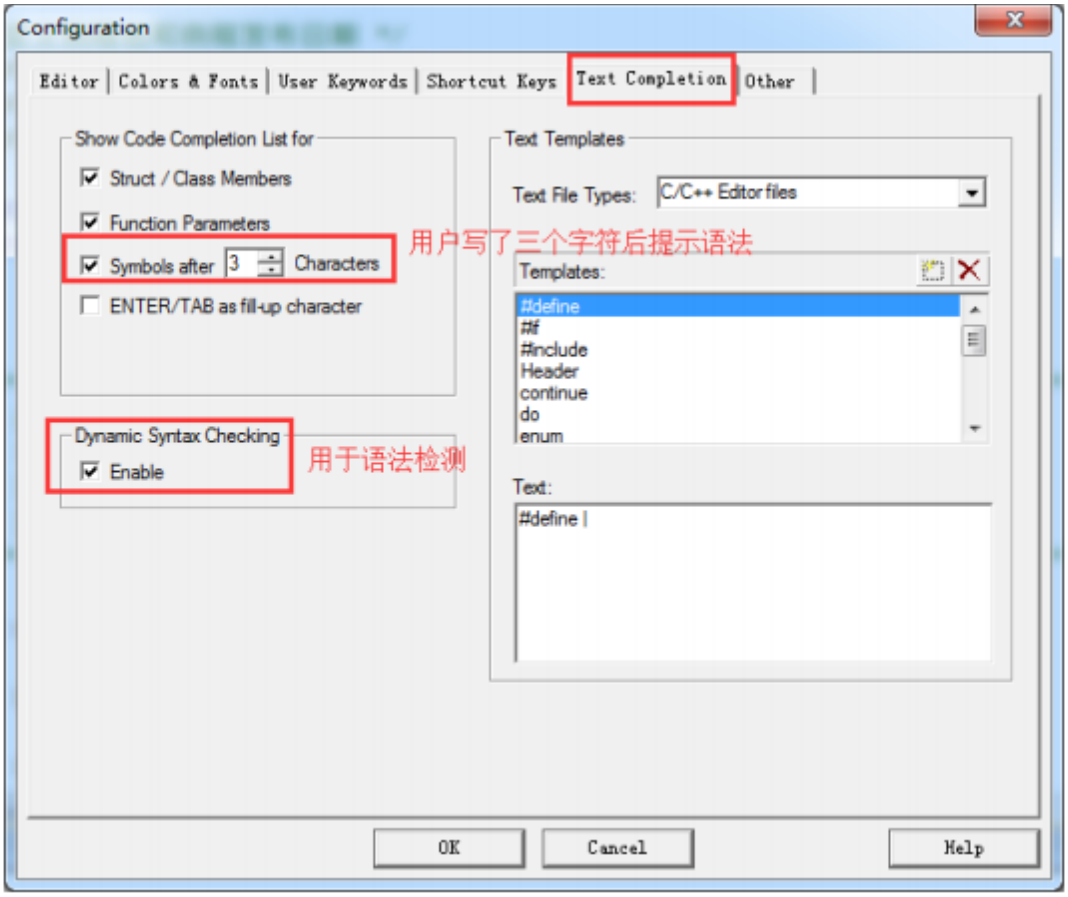
```
24
25  extern  uint16_t EEPROM_ADDRESS;
26  extern  u8 I2C1_tx_buffer[255];
27  extern  u8 I2C1_rx_buffer[255];
28  int main(void)
29  {
30      int i;
31      delay_init();
32      uart1_init(115200);
33      printf("I2C TEST START\r\n");
34      I2C_EE_Init();
35       for ( i=0; i<=255; i++ ) //填充缓冲
36       {
37          I2C1_tx_buffer[i]=255-i;
38
39       }
```

3.8.4语法提示和语法检测功能

选择此图标:



MDK 的语法提示和语法检测功能还行，大家可以根据需要开启:



3.9 总结

对于初学者来说，务必要实际动手操作一遍，以后移植其它程序代码大有裨益。

第4章 STC32M4 下载和调试方法（MDK5）

本章教程为大家介绍如何通过 MDK5 下载程序到芯片里面以及 MDK5 的调试和配置方法。

4.1 初学者重要提示

4.2 使用 MDK 调试和下载程序设置（ULINK2）

4.3 MDK 调试容易崩溃的解决办法

4.4 MDK 程序调试

4.5 总结

4.1 初学者重要提示

- 如果使用 ULINK2 调试下载 STC32M4
- MDK5 调试过程中容易崩溃的问题，请看本章 4.4 小节，比较重要。

。

4.3 使用 MDK调试和下载程序设置（ULINK2）

调试下载 STC32M4，请将 ULINK2 固件升级到最新版，关于固件升级和驱动安装等相关文件已经在 本教程第 2 章的 2.6 章节有说明。

在上个章节里面， 我们已经将 Options 对话框里面大部分设置选项都做了说明， 本章节主要是调试下载的设置。

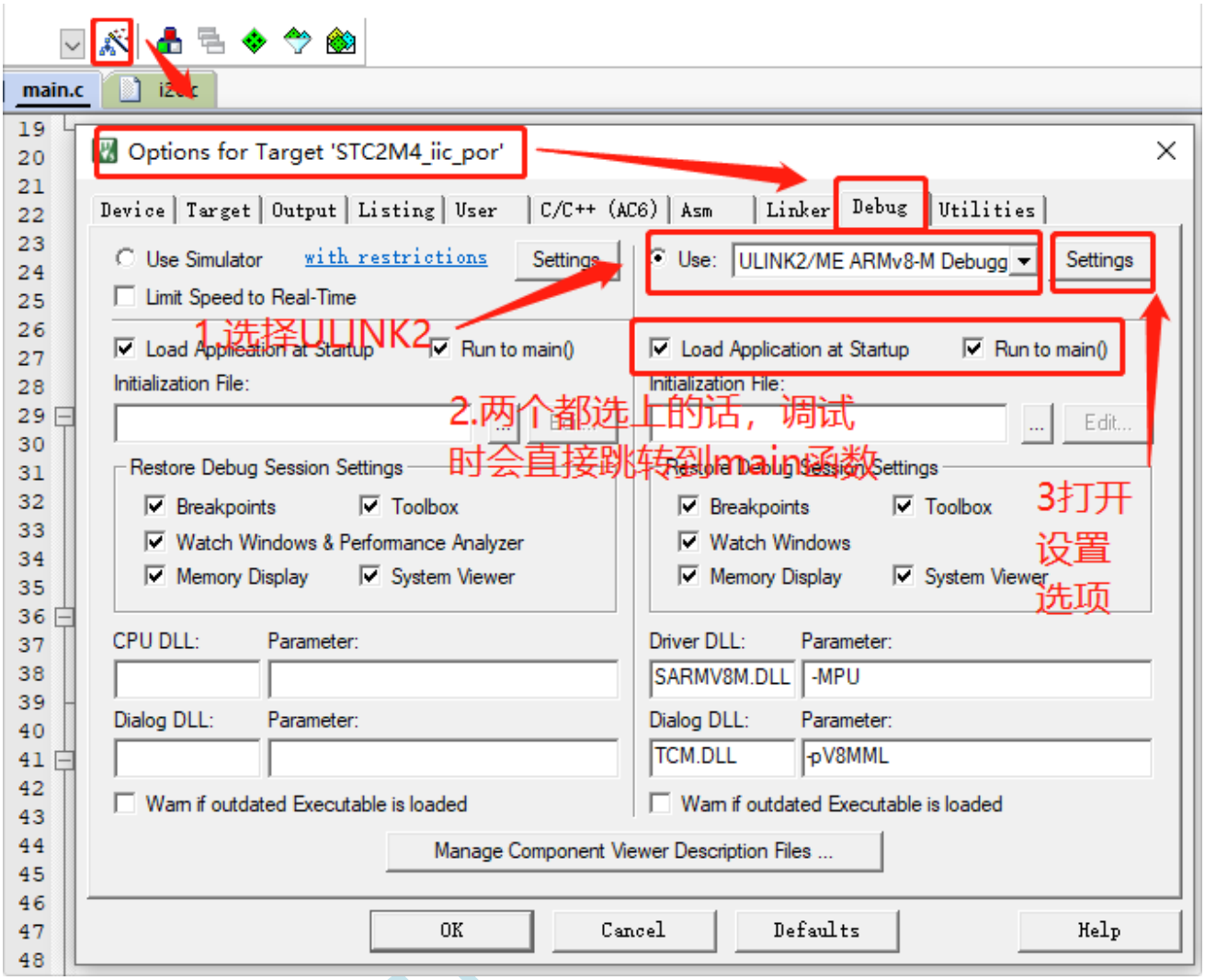
4.3.1开发板和 ULINK2 的接线图

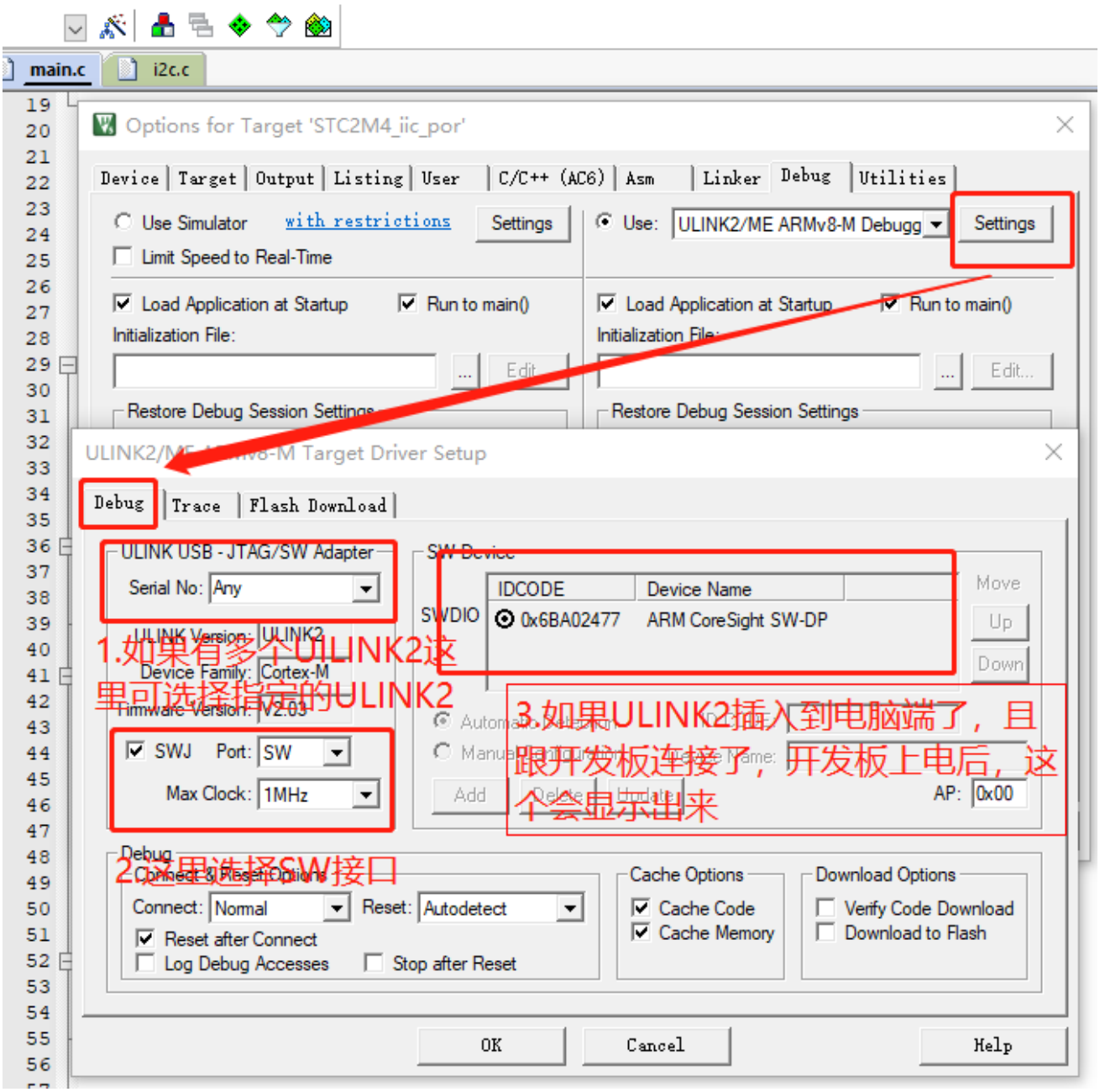
开发板接上 STLINK 的效果图如下：



4.3.2设置 Options 里面的 Debug 选项

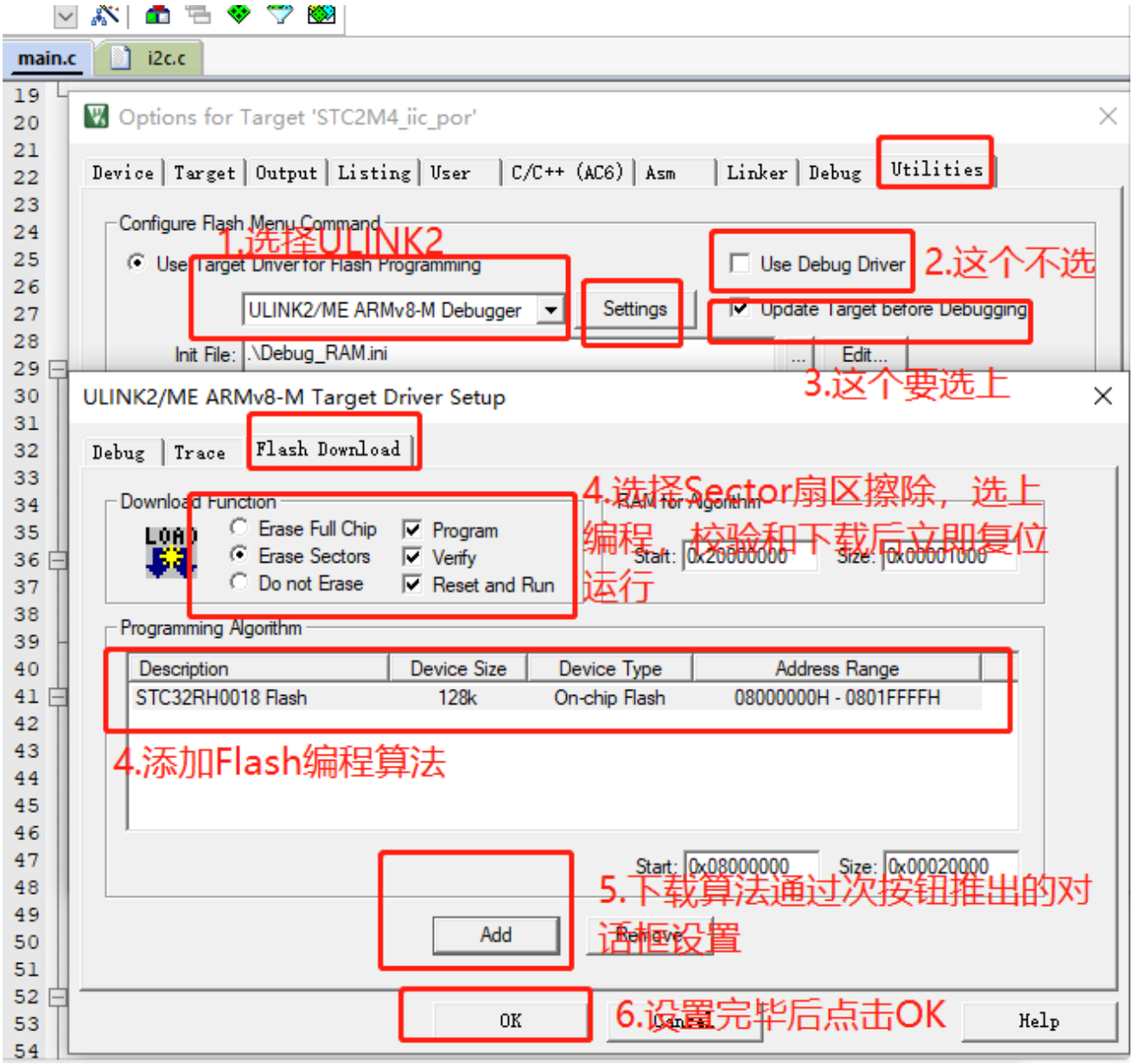
Debug选项用于调试设置


main.c
Options for Target 'STC2M4_iic_por'
Device
Target
Output
Listing
User
C/C++ (AC6)
Asm
Linker
Debug
Utilities
Use Simulator
with restrictions
Settings
Limit Speed to Real-Time
Load Application at Startup
Run to main()
Initialization File:
Restore Debug Session Settings
Breakpoints
Toolbox
Watch Windows & Performance Analyzer
Memory Display
System Viewer
CPU DLL:
Parameter:
Dialog DLL:
Parameter:
Warn if outdated Executable is loaded
Use:
ULINK2/ME ARMv8-M Debugg
Settings
Load Application at Startup
Run to main()
Initialization File:
Edit...
Restore Debug Session Settings
Breakpoints
Toolbox
Watch Windows
Memory Display
System Viewer
Driver DLL:
Parameter:
SARMV8M.DLL
-MPU
Dialog DLL:
Parameter:
TCM.DLL
-pV8MML
Warn if outdated Executable is loaded
Manage Component Viewer Description Files ...
OK
Cancel
Defaults
Help
1.选择ULINK2
2.两个都选上的话，调试
时会直接跳转到main函数
3打开
设置
选项



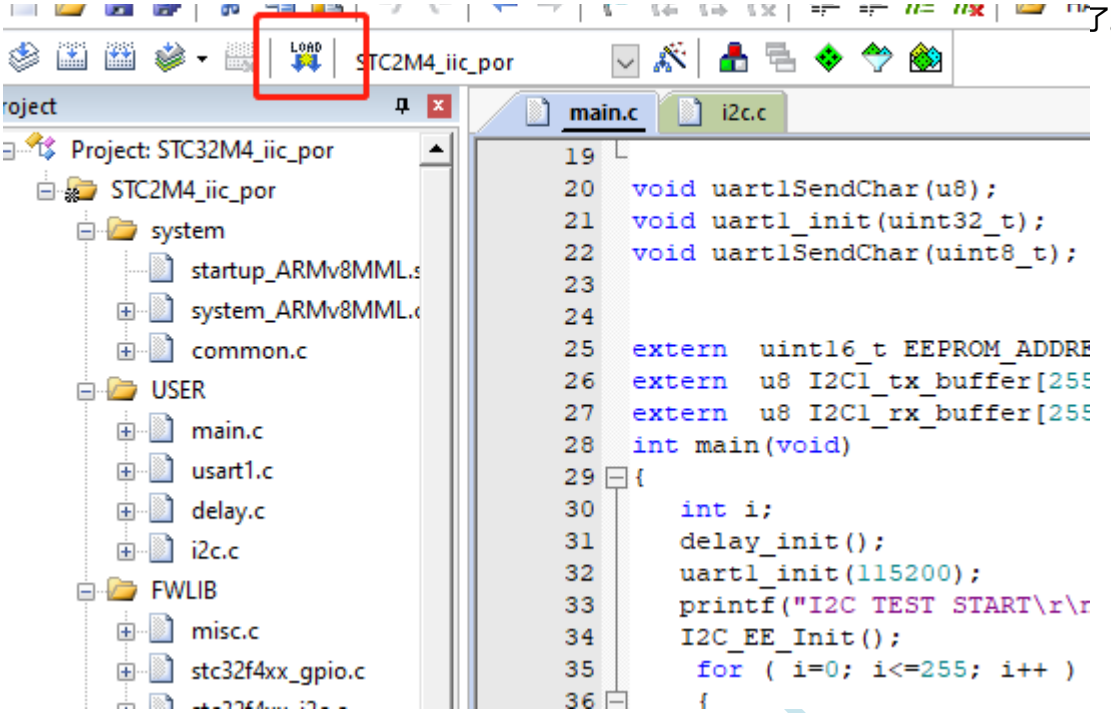
4.3.3设置 Options 里面的 Utilities 选项

Utilities 选项主要用于程序下载设置。

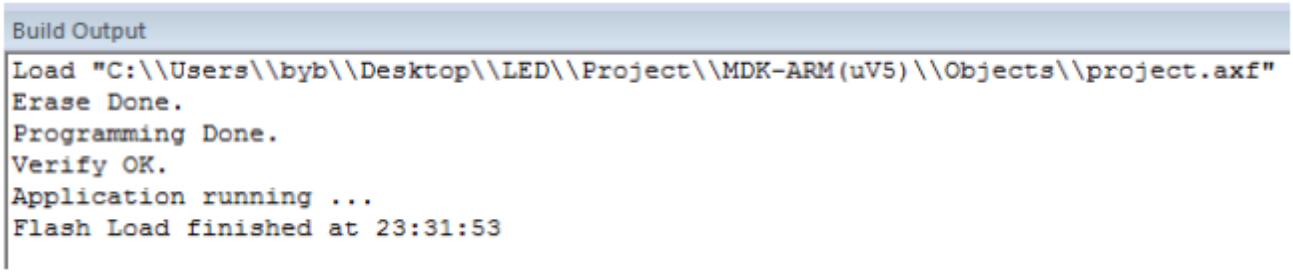

main.c
i2c.c
Options for Target 'STC2M4_iic_por'
Device
Target
Output
Listing
User
C/C++ (AC6)
Asm
Linker
Debug
Utilities
Configure Flash Menu Command
1.选择ULINK2
Use Target Driver for Flash Programming
ULINK2/ME ARMv8-M Debugger
Settings
Use Debug Driver
2.这个不选
Update Target before Debugging
Init File: .\Debug_RAM.ini
Edit...
3.这个要选上
ULINK2/ME ARMv8-M Target Driver Setup
Debug
Trace
Flash Download
Download Function
Erase Full Chip
Erase Sectors
Do not Erase
Program
Verify
Reset and Run
4.选择Sector扇区擦除，选上编程、校验和下载后立即复位运行
RAM for Algorithm
Start: 0x20000000
Size: 0x00001000
Programming Algorithm
Description
Device Size
Device Type
Address Range
STC32RH0018 Flash
128k
On-chip Flash
08000000H - 0801FFFFH
4.添加Flash编程算法
Start: 0x08000000
Size: 0x00020000
5.下载算法通过次按钮推出的对话框设置
Add
Remove
OK
Cancel
6.设置完毕后点击OK
Help

4.3.4程序下载

了。点击这里即可:

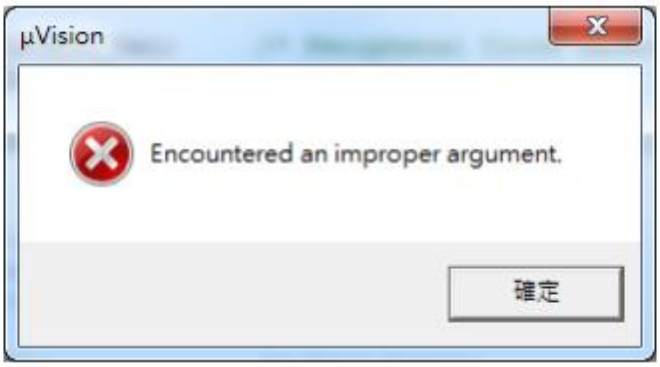


下载成功后可以看到开发板上的 LED 开始闪烁，MDK 的也有信息展示:



4.4 MDK调试容易崩溃的解决办法

进入到 MDK5.2X 后， MDK 调试的时候很容易崩溃，出现下面这个错误



出现这个错误的话，要注意下面两个问题: ◆ 如果出现这种情况的话，将工程放在桌面路径下，工程名不要有中文会好很多。路径深或者有中文， 都会造成这种情况。

◆ 退出调试的时候，如果使用断点了，务必清除所有断点，再退出，否则也会造成这种情况。

4.5 MDK程序调试

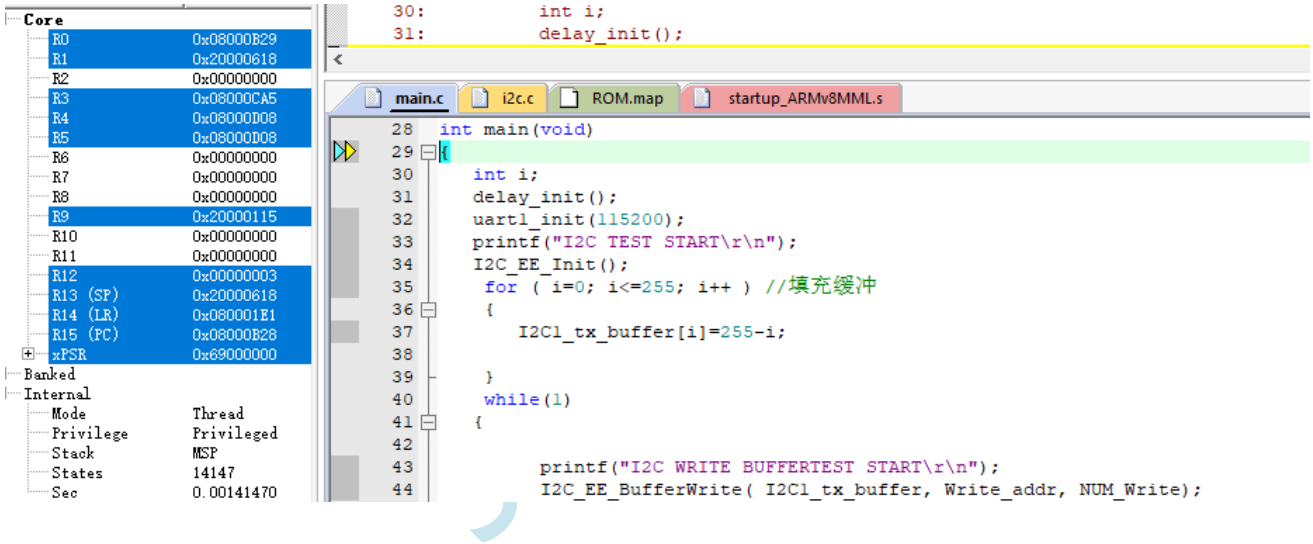
ULINK2 设置完毕后，调试步骤如下。

4.5.1全速运行和单步调试

点击如下按钮可以进入调试状态:



进入调试状态后的效果如下:



首先要认识下这 7 个图标的含义:



◆ 第 1 个图标是复位调试。◆ 第 2 个图标是全速运行。◆ 第 3 个图标是进入全速运行状态后，通过这个图标可以停止全速运行。 ◆ 第 4 个图标是 Step（Step one line)，单步调试，每次点击运行一行代码，所以只要有子函数就会 进入。 ◆ 第 5 个图标是 Step Over（Step over the current line),每次点击以函数为单位运行，不会进入函 数内部。◆ 第 6 个图标是 Step out（Step out of the current line） ，点击后退出当前运行的函数，执行下一个 函数。 ◆ 第 7 个图标是 Run to Cursor Line（Run to the current cursor line)，直接运行到用户设置的行， 设置方法比较简单，用户仅需在那行使用鼠标左键点击一下，可以看到前面有个箭头

```
24
25  extern  uint16_t EEPROM_ADDRESS;
26  extern  u8 I2C1_tx_buffer[255];
27  extern  u8 I2C1_rx_buffer[255];
28  int main(void)
29  {
30      int i;
31      delay_init();
32      uart1_init(115200);
33      printf("I2C TEST START\r\n");
34      I2C_EE_Init();
35       for ( i=0; i<=255; i++ ) //填充缓冲
36       {
37          I2C1_tx_buffer[i]=255-i;
38
39       }
```

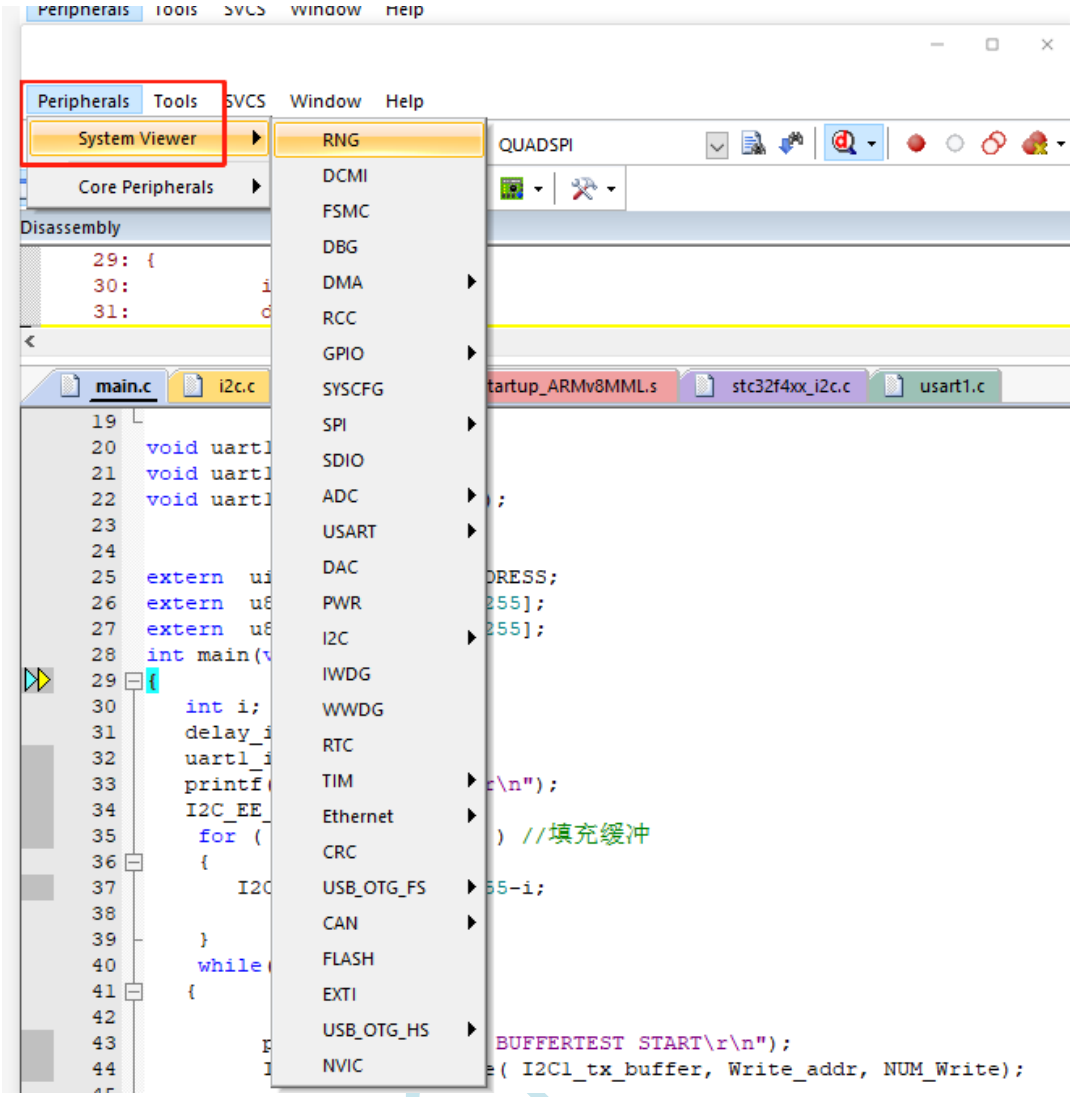
然后点击此图标即可，可以看到前面多了一个黄色的小箭头，说明已经运行到此行

```
28  int main(void)
29  {
30      int i;
31      delay_init();
32      uart1_init(115200);
33      printf("I2C TEST START\r\n");
34      I2C_EE_Init();
35       for ( i=0; i<=255; i++ ) //填充缓冲
36       {
37          I2C1_tx_buffer[i]=255-i;
38
39       }
```

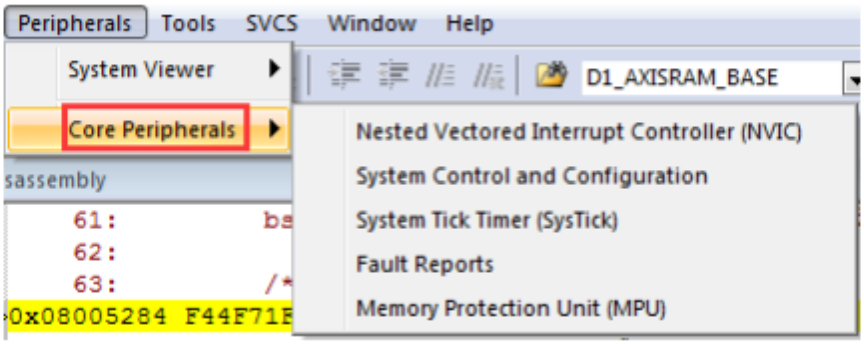
前面几个图标的功能，大家多测试几次就熟练了。

4.5.2外设寄存器的查看

要查看的外设寄存器内容在这个菜单里面:



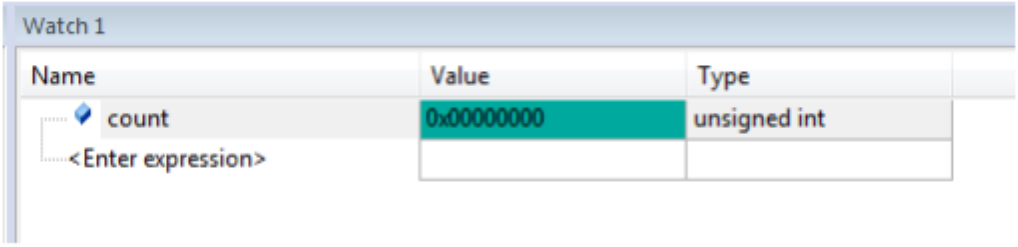
如果要查看内核外设寄存器，在这里:



4.5.3全局变量的查看

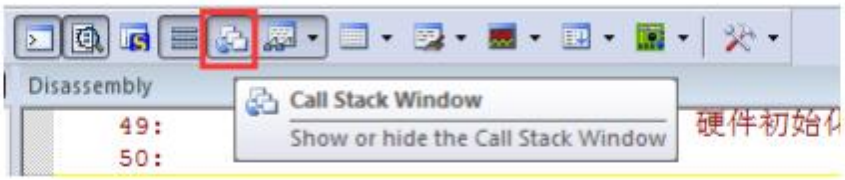
针对全局变量，MDK 专门提供了 Watch1 和Watch2 两个窗口，每个窗口可以添加很多变量。调试过程中，这些窗口里面添加的变量是会更新的。



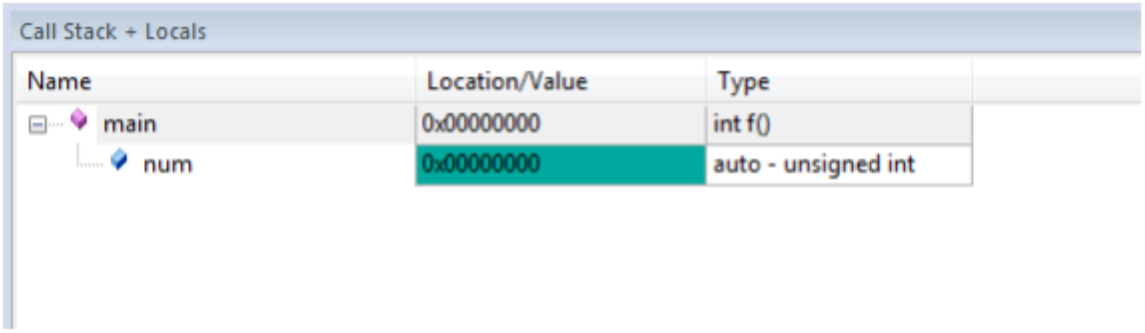


4.5.4局部变量的查看

针对局部变量， MDK 专门提供了一个栈窗口，用于查看函数中各种局部变量，这个不需要用户添加，MDK 会自动添加。

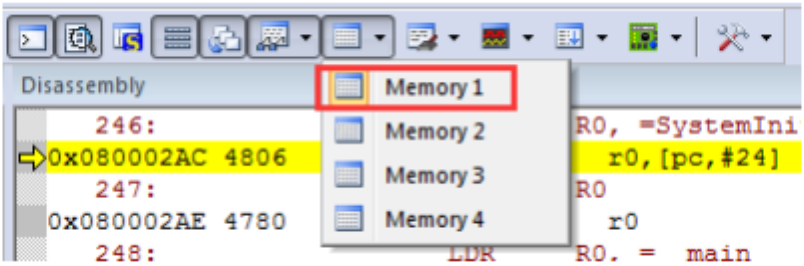


也可以从菜单 View ->Call Stack Window 进行添加。局部变量需要进入到那个函数里面才可以查看，这一点要特别注意，因为局部变量是来自栈空间的，进入函数申请占空间，退出函数后，栈空间就会被释放掉。比如 main 函数里面一个 num 局部变量:

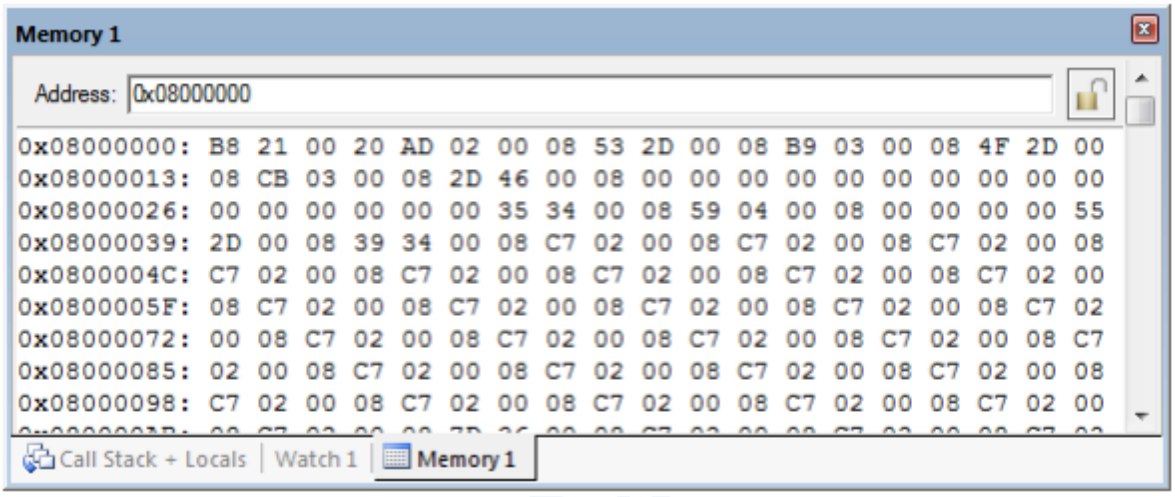


4.5.5Flash 区或者 RAM 区的查看

如果要查看 Flash 区或者 RAM 区的内容，MDK 提供了 4 个 Memory 窗口。

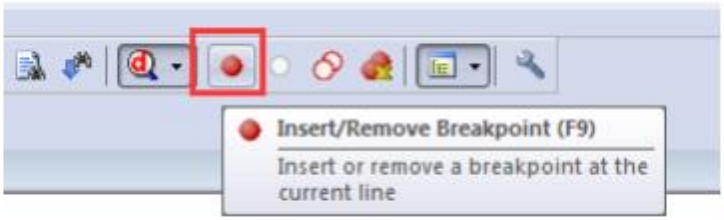


也可以从菜单 View - >Memory Windows 进行添加。比如要查看 0x0800 0000 地址开始的内容，直接输入即可，这个地址是 STC32M4 的 Flash 首地址:

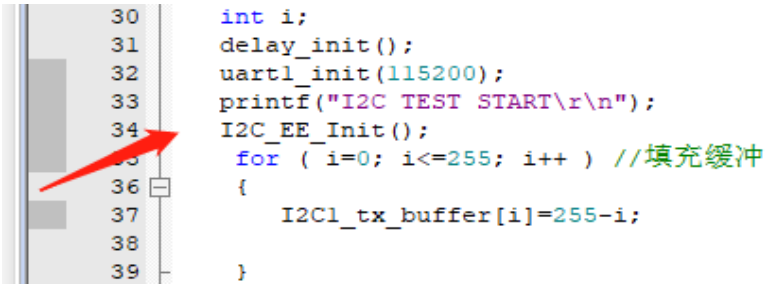


4.5.6断点调试

断点调试比较重要，进入调整状态后，如果大家想让程序在指定的位置停止运行，可以添加断点，通 过这个小红点来添加。



比如想在 34 行添加断点。



用户仅需在那行使用鼠标右键点击一下，可以看到前面有个箭头:

```
30    int i;
31    delay_init();
32    uart1_init(115200);
33    printf("I2C TEST START\r\n");
34    I2C_EE_Init();
35     for ( i=0; i<=255; i++ ) //填充缓冲
36     {
37        I2C1_tx_buffer[i]=255-i;
38
39     }
```

然后点击鼠标左键，可以看到已经添加上红色断点标志:

```
30    int i;
31    delay_init();
32    uart1_init(115200);
33    printf("I2C TEST START\r\n");
34    I2C_EE_Init();
35     for ( i=0; i<=255; i++ ) //填充缓冲
36     {
37        I2C1_tx_buffer[i]=255-i;
38
39     }
```

然后点击全速运行，如果程序有执行到这个地方，就会停在这里。可以看到黄色小箭头也显示在左侧了， 说明程序执行这个地方了:

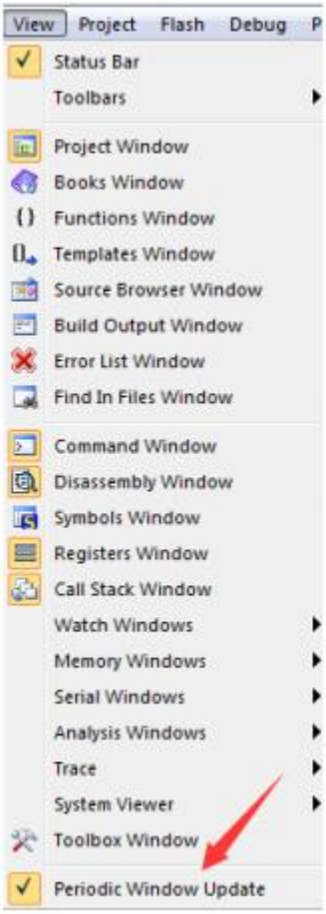
```
28  int main(void)
29  {
30    int i;
31    delay_init();
32    uart1_init(115200);
33    printf("I2C TEST START\r\n");
34    I2C_EE_Init();
35     for ( i=0; i<=255; i++ ) //填充缓冲
36     {
37        I2C1_tx_buffer[i]=255-i;
38
39     }
```

正常情况下，断点可以加到任何程序代码可以执行到的地方。另外注意，退出调试前，点击这个按钮取消 所有断点，防止 MDK 调试期间崩溃:



4.5.7寄存器， 全局变量和局部变量的动态更新

MDK 有个周期性更新选项， 如果大家要周期性的观察一些寄存器、 全局变量的更新、 需要把菜单 View - > Periodic Window Update 选上。



这个周期不能设置，大概 0.5s 左右更新一次。

4.6 总结

对于初学者来说，务必要实际动手操作一遍。特别是调试方法，要熟练掌握，因为以后工程代码调试 时，用到的最多。

第5章 STC32M4重要知识点数据类型，变量和堆栈

本章教程为大家介绍数据类型，变量和堆栈的相关知识。

5.1 初学者重要提示
5.2 数据类型
5.3 局部变量和全局变量
5.4 堆栈
5.5 局部变量，全局变量和堆栈实例
5.6 总结

5.1 初学者重要提示

- 如果对C语言不熟练的话，可以阅读C语言书：C Primer Plus（第五版）中文版.pdf
- 为了更好的学习本章知识点， 可以看之前做的视频教程第 10 章，

5.2 数据类型

了解数据类型之前要对 ANSI C 和 ISO C 的发展史有个了解，特别是 C89，C99 和 C11 的由来。

5.2.1ANSI C 和 ISO C 历史

- 1983 年， 美国国家标准协会（ANSI） 组成了一个委员会来创立 C 语言的标准。因为这个标准是 1989 年发布的，所以一般简称 C89 标准。有些人也把 C89 标准叫做 ANSI C。
- 在 1990 年, ANSI C89 标准被国际标准化组织(ISO)和国际电工委员会(IEC)采纳为国际标准, 名叫 ISO/IEC 9899:1990 - Programming languages C，有些人简称 C90 标准。因此，C89 和 C90 通 常指同一个标准，一般更常用C89 这种说法。
- 在 2000 年 3 月，国际标准化组织(ISO)和国际电工委员会(IEC)采纳了第二个 C 语言标准，名叫ISO/IEC 9899:1999 - Programming languages -- C，简称 C99 标准。

- 在 2011 年 12 月，国际标准化组织(ISO)和国际电工委员会(IEC)采纳了第三个 C 语言标准，名叫 ISO/IEC 9899:2011 - Information technology -- Programming languages -- C,简称 C11标准。它是 C 程序语言的最新标准。

  对于我们常用的编译器 MDK 和 IAR 而已，C89，C99 和 C11 均支持。

5.2.2ARM 架构（含 Cortex-M 系列）数据类型

ARM 架构（含 Cortex-M 系列）的数据类型定义如下:

Table 2.2 Size and Range of Data Types in ARM Architecture Including Cortex-M Processors

C and C99 (stdint.h) Data Type	Number of Bits	Range (Signed)	Range (Unsigned)
char, int8_t, uint8_t	8	−128 to 127	0 to 255
short int16_t, uint16_t	16	−32768 to 32767	0 to 65535
int, int32_t, uint32_t	32	−2147483648 to 2147483647	0 to 4294967295
Long	32	−2147483648 to 2147483647	0 to 4294967295
long long, int64_t, uint64_t	64	−(2^63) to (2^63 - 1)	0 to (2^64 - 1)
Float	32	$-3.4028234 \times 10^{38}$ to 3.4028234×10^{38}	
Double	64	$-1.7976931348623157 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
long double	64	$-1.7976931348623157 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
Pointers	32	0x0 to 0xFFFFFFFF	
Enum	8 / 16/ 32	Smallest possible data type, except when overridden by compiler option	
bool (C++ only), _Bool (C only)	8	True or false	
wchar_t	16	0 to 65535	

5.2.3头文件 stdint.h 对数据类型的定义

stdint.h 是 C99 中引进的一个标准 C 库的头文件。目前大部分单片机 C 编译器均支持, IAR 和 MDK 都支持。下面是部分内容 （来自 MDK）。

```
/* exact-width signed integer types */
typedef   signed          char
typedef   signed short     int8_t;int
typedef   signed           int16_t; int
                           int32_t;
typedef signed__INT64 int64_t;
```

```
typedef unsigned          char uint8_t;
typedef unsigned short int uint16_t;
typedef unsigned           int uint32_t;
typedef unsigned       __INT64 uint64_t;

/* 7.18.1.2 */

/* smallest type of at least n bits */
/* minimum-width signed integer types */
typedef   signed          char int_least8_t;
typedef   signed short int int_least16_t;
typedef   signed           int int_least32_t;
typedef   signed       __INT64 int_least64_t;

/* minimum-width unsigned integer types */
typedef unsigned          char uint_least8_t;
typedef unsigned short int uint_least16_t;
typedef unsigned           int uint_least32_t;
typedef unsigned       __INT64 uint_least64_t;

/* 7.18.1.3 */

/* fastest minimum-width signed integer types */
typedef   signed           int
typedef   signed           int_fast8_t;int
typedef   signed           int_fast16_t;
typedef   signed           int int_fast32_t;
                       __INT64 int_fast64_t;

    /* fastest minimum-width unsigned integer types */
typedef unsigned           int
typedef unsigned           uint_fast8_t;int
typedef unsigned           uint_fast16_t;
typedef unsigned           int uint_fast32_t;
                       __INT64 uint_fast64_t;
```

以 MDK5.26 为例，stdint.h 位于如下路径:

\Keil_v526\ARM\ARMCC\include

5.2.4程序中推荐的变量命名方式

看程序的时候，经常会看到各种各样的变量命名方式，例如声明一个 8 位无符号整数，常见的有如下 几种写法:

u8 err;

U8 err;

INT8U err;

UINT8 err;

uint8 err;

uint8_t err;

当大家阅读别人写的程序时，往往会看到风格各异的定义方式，移植部分程序时也不知道采用哪种方式更合适。

我们推荐大家采用最后一种定义方式，这种方法符合 C99 规范.

5.3 局部变量和全局变量

5.3.1局部变量

在一个函数内部定义的变量是内部变量，它只在本函数范围内有效，也就是说只有在本函数内才能使 用它们，在此函数以外是不能使用这些变量的，这称为局部变量。

使用局部变量注意以下问题：

- 不同函数中可以使用相同名字的变量，它们代表不同的对象，互不干扰。
- 形式参数也是局部变量。
- 局部变量的作用域在函数内部。

5.3.2全局变量

在函数内部定义的变量是局部变量，而在函数之外定义的变量称为外部变量，也就是全局变量。使 用全局变量的注意事项：

- 全局变量可以为本文件中其他函数所共用。它的有效范围为从定义变量的位置开始到本源文件结束。
- 设置全局变量的作用是增加了函数间数据联系的渠道。
- 如果在同一个源文件中， 外部变量和局部变量同名， 则在局部变量的作用范围内， 外部变量被“屏蔽”， 即外部变量将不起作用。

5.3.3使用全局变量的缺点

程序设计中，建议不要创建太多的全局变量，主要是出于以下三点考虑：

- 全局变量在程序的执行过程中都占用存储单元，而不是仅在需要时才占用存储单元。

- 函数的通用性降低了，因为函数在执行时要依赖于其所在的外部变量。如果将一个函数移植到另一个文件中，还要将有关的外部变量及其值一起移植过去。
- 使用全局变量过多，会降低程序的清晰性，特别是多个函数都调用此变量时。

5.3.4变量的存储类别

从变量的作用域来分，可以分为全局变量和局部变量，而从变量值存在的时间来看，可以分为静态存储方式和动态存储方式。

- 静态存储方式：指在程序运行期间由系统分配固定的存储空间方式。
- 动态存储方式：在程序运行期间根据需要进行动态的分配存储空间方式。 全局变量存储在静态存储区中 ，动态存储区可以存放以下数据：
- 函数形式参数，在调用函数时给形参分配存储空间。
- 局部变量（未加 static 声明的局部变量）。
- 函数调用时的现场保护和返回地址等。

5.3.5用 static 声明局部或者全局变量

有时候希望函数中的局部变量的值在函数调用结束后不消失而保留原值，即占用的存储单元不释放， 在下一次该函数调用时，该变量已有值，就是上一次函数调用结束时的值。这时可以使用关键字 static 进行声明。

用 static 声明一个变量的作用：

- 对局部变量用 static 声明，则使用该变量在整个程序执行期间不释放，为其分配的的空间始终存在。
- 全局变量用 static 声明，则该变量的作用域只限于本文件模块（即被声明的文件中）。

5.4 堆栈

5.4.1堆栈作用

栈(stack)空间，用于局部变量，函数调时现场保护和返回地址，函数的形参等。

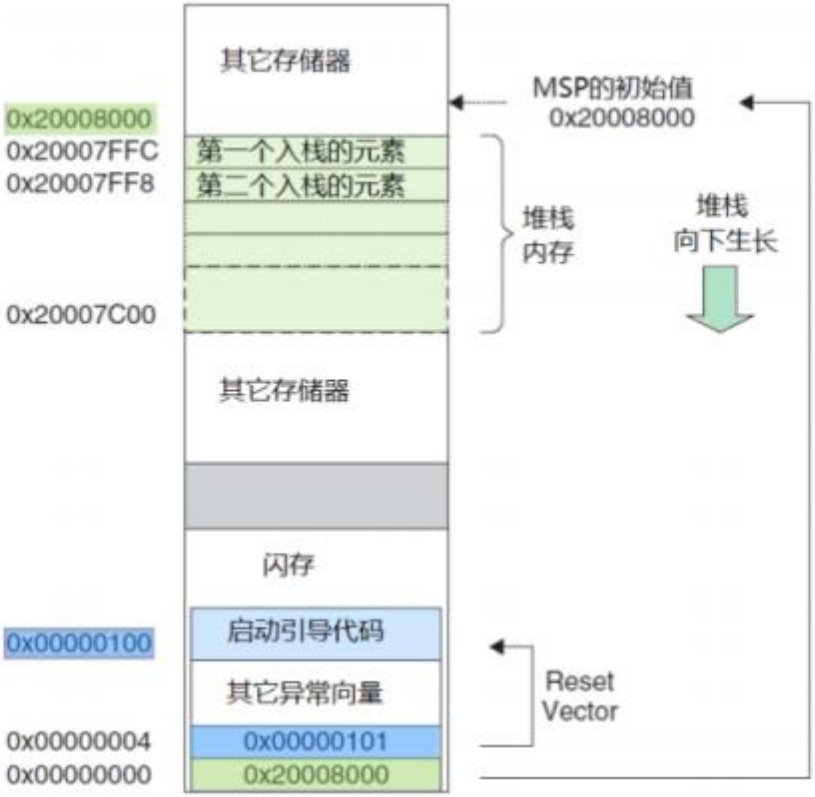
堆(heap)空间， 主要用于动态内存分配， 也就是说用 malloc，calloc, realloc 等函数分配的变量空间是 在堆上。以 STC32M4 为例，堆栈是在 startup_ARMv8MML.s 文件里面设置：

5.4.2寄存器组（堆栈指针寄存器）

Cortex – M4/M3 处理器拥有 R0-R15 的通用寄存器组。其中 R13 作为堆栈指针 SP。SP 有两 个，但在同一时刻只能有一个可以用。 ◆ 主堆栈指针（MSP):这是缺省的堆栈指针，它由 OS 内核、异常服务例程以及所有需要特权访问的 应用程序代码来使用。◆ 进程堆栈指针（PSP):用于常规的应用程序代码（不处于异常服务例程中时）。 另外以下两点要注意: ◆ 大多数情况下的应用，只需使用指针 MSP，而 PSP 多用于RTOS 中。 ◆R13 的最低两位被硬线连接到 0，并且总是读出 0，这意味着堆栈总是 4 字节对齐的。

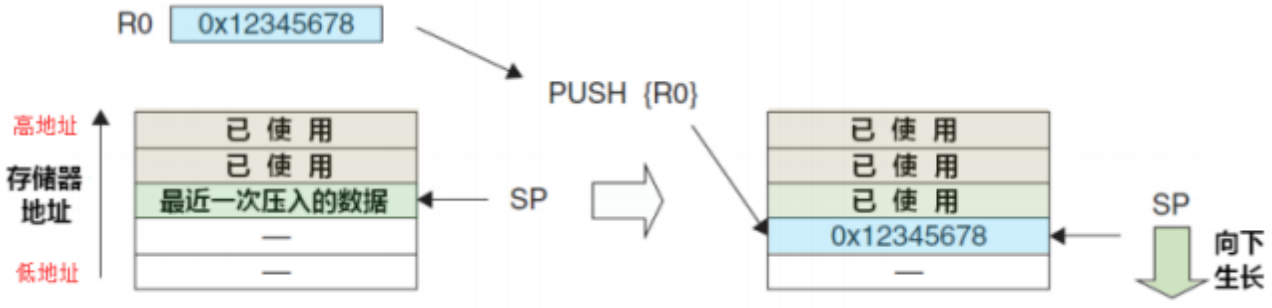
R0	通用寄存器
R1	通用寄存器
R2	通用寄存器
R3	通用寄存器
R4	通用寄存器
R5	通用寄存器
R6	通用寄存器
R7	通用寄存器
R8	通用寄存器
R9	通用寄存器
R10	通用寄存器
R11	通用寄存器
R12	通用寄存器
R13(MSP,PSP)	堆栈指针（SP）
R14	连接寄存器（LR）
R15	程序计数器（PC）

5.4.3Cortex-M4/M3 向下生长的满栈



5.4.4堆栈的基本操作

这里对入栈和出栈做个简单的介绍。PUSH 入栈操作：SP 先自减 4，再存入新的数值：



POP 出栈操作：先从 SP 指针处读出上一次被压入的值，再把 SP 指针自增 4：

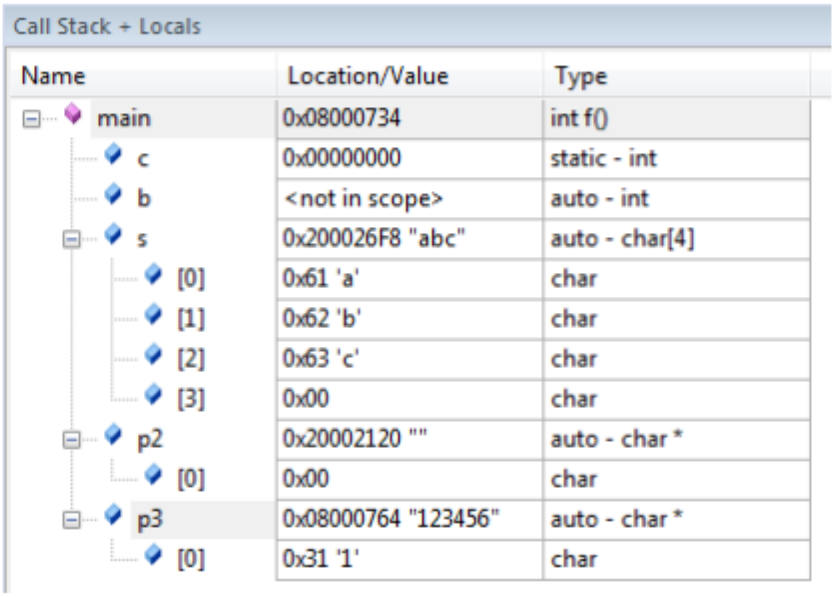
5.5 局部变量，全局变量和堆栈实例

通过下面的实例可以对局部变量，全局变量和堆栈有个感性的认识:

```
    uint32_t a = 0; //全局初始化区, 可以被其他c文件 extern 引用static uint32_t ss = 0;//静态变量，只允许在本文件使用 uint8_t *p1;//全局未初始化区 int main(void){ uint32_t b; //栈 uint8_t s[] = "abc";//栈
    uint8_t *p2;//栈 uint8_t *p3 = "123456"; //123456\0在常量区, p3在栈上。 static uint32_t c =0; //全局（静态）初始化区

    p1 = (uint8_t *)malloc(10); //在堆区申请了10个字节空间 p2 = (uint8_t *)malloc(20); //在堆区申请了20个字节空间 strcpy(p1, "123456");/* 123456字符串（结束符号是0(\0)，总长度7）放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方 */
}
```

通过查看 MAP 文件，可以看全局变量在 RAM 中的位置: `Symbol NameValue0v TypeSizeObject(Section)a 0x20000000 Data4 main.o(.data)p10x2000000c Data4 main.o(.data)ss0x20000004 Data4 main.o(.data)`而局部变量要调整状态进入 main函数里面查看:



5.6 总结

C 语言的基础知识点要掌握牢靠，对于后面学习STC32库源码大有裨益。

第6章STC32M4 的 FLASH，RAM和栈使用情况（map和htm文件）

本章为大家介绍编译器生成的 map 和 htm 文件进行解析， 通过这两个文件可以让大家对工程代码的认识程度提升一个档次。

6.1 初学者重要提示
6.2 MAP 文件分析
6.3 HTM 文件分析
6.4 总结

6.1初学者重要提示

- 本章节以 MDK 为例进行说明。

6.2MAP文件分析

通过 map 文件, 可以方便的查看工程 ROM/FLASH 和 RAM 的占用情况, 包括单个源文件, 甚至具 体到每个函数的 rom 大小都给出了，通过这些信息可以很好的进行代码优化。MAP 文件的内容可分为如下几部分:

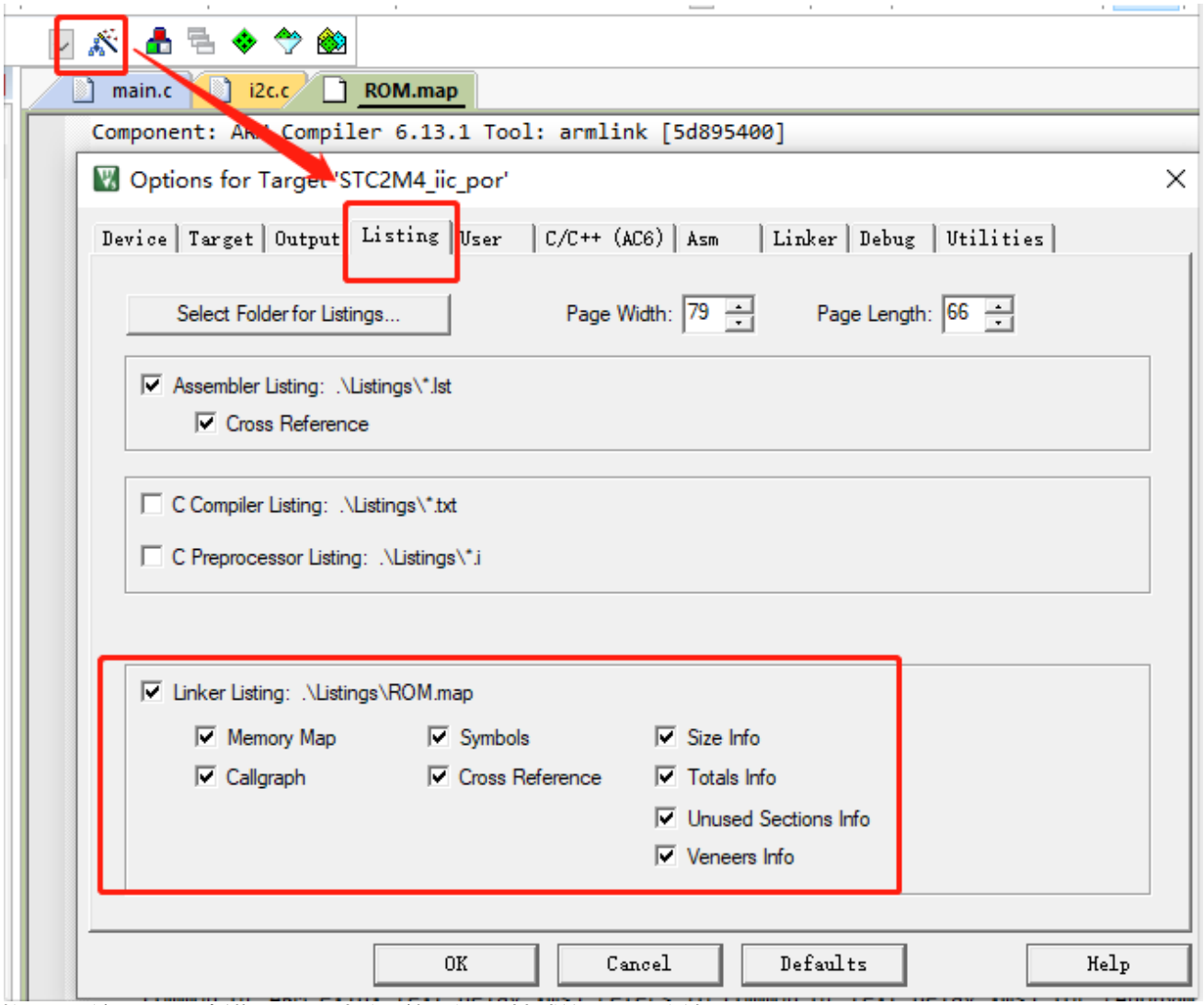
- Section Cross References
- Removing Unused input sections from the image
- Image Symbol Table (Local Symbols Global Symbols)
- Memory Map of the image
- Image component sizes

下面将这个几个部分的含义逐一为大家做个说明。

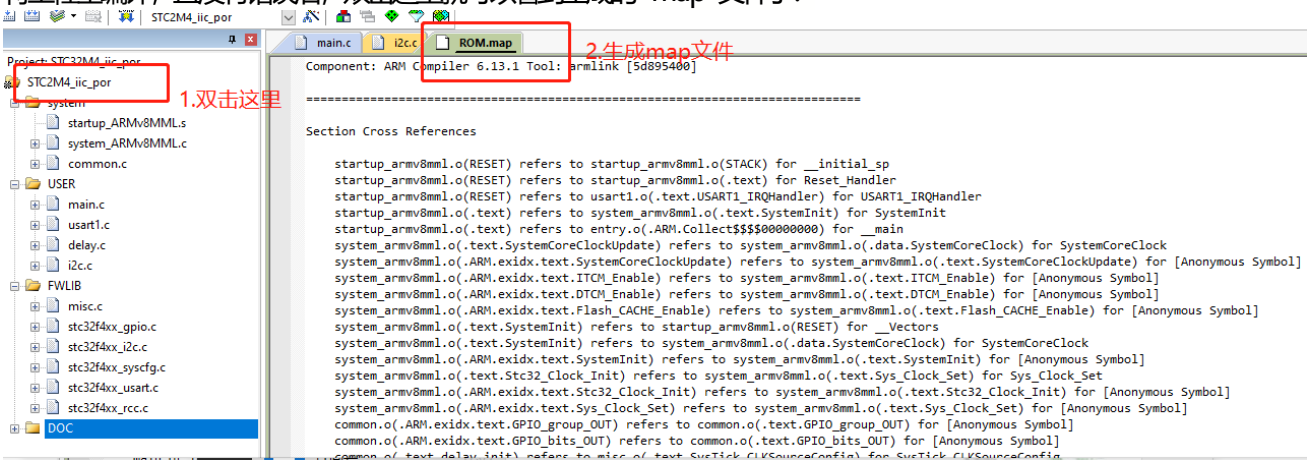
6.2.1MDK 配置

要生成 MAP 文件，MDK 中如下选项要选上:

STC MCU



将工程全编译，且没有错误后，双击这里就可以看到生成的 map 文件了:



6.2.2MAP 文件相关概念

认识 MAP 文件前先熟悉如下概念:

- 段(section) ：描述映像文件的代码和数据块。
- RO：Read-Only 的缩写，包括 RO-data(只读数据)和 RO-code(代码)。
- RW：Read-Write 的缩写，主要是 RW-data，RW-data 由程序初始化初始值。
- ZI： Zero-initialized 的缩写，主要是 ZI-data，由编译器初始化为 0。
- .text：与 RO-code 同义。
- .constdata：与 RO-data 同义。
- .bss：与 ZI-data 同义。
- .data：与 RW-data 同义

6.2.3Section Cross References

这部分主要是不同文件中函数的调用关系（部分截图）

main.c | i2c.c | ROM.map

```
Section Cross References

    startup_armv8mml.o(RESET) refers to startup_armv8mml.o(STACK) for __initial_sp
    startup_armv8mml.o(RESET) refers to startup_armv8mml.o(.text) for Reset_Handler
    startup_armv8mml.o(RESET) refers to usart1.o(.text.USART1_IRQHandler) for USART1_IRQHandler
    startup_armv8mml.o(.text) refers to system_armv8mml.o(.text.SystemInit) for SystemInit
    startup_armv8mml.o(.text) refers to entry.o(.ARM.Collect$$$$00000000) for __main
    system_armv8mml.o(.text.SystemCoreClockUpdate) refers to system_armv8mml.o(.data.SystemCoreClock) for SystemCoreClock
    system_armv8mml.o(.ARM.exidx.text.SystemCoreClockUpdate) refers to system_armv8mml.o(.text.SystemCoreClockUpdate) for [Anonymous Symbol]
    system_armv8mml.o(.ARM.exidx.text.ITCM_Enable) refers to system_armv8mml.o(.text.ITCM_Enable) for [Anonymous Symbol]
    system_armv8mml.o(.ARM.exidx.text.DTCM_Enable) refers to system_armv8mml.o(.text.DTCM_Enable) for [Anonymous Symbol]
    system_armv8mml.o(.ARM.exidx.text.Flash_CACHE_Enable) refers to system_armv8mml.o(.text.Flash_CACHE_Enable) for [Anonymous Symbol]
    system_armv8mml.o(.text.SystemInit) refers to startup_armv8mml.o(RESET) for __Vectors
    system_armv8mml.o(.text.SystemInit) refers to system_armv8mml.o(.data.SystemCoreClock) for SystemCoreClock
    system_armv8mml.o(.ARM.exidx.text.SystemInit) refers to system_armv8mml.o(.text.SystemInit) for [Anonymous Symbol]
    system_armv8mml.o(.text.Stc32_Clock_Init) refers to system_armv8mml.o(.text.Sys_Clock_Set) for Sys_Clock_Set
    system_armv8mml.o(.ARM.exidx.text.Stc32_Clock_Init) refers to system_armv8mml.o(.text.Stc32_Clock_Init) for [Anonymous Symbol]
    system_armv8mml.o(.ARM.exidx.text.Sys_Clock_Set) refers to system_armv8mml.o(.text.Sys_Clock_Set) for [Anonymous Symbol]
    common.o(.ARM.exidx.text.GPIO_group_OUT) refers to common.o(.text.GPIO_group_OUT) for [Anonymous Symbol]
    common.o(.ARM.exidx.text.GPIO_bits_OUT) refers to common.o(.text.GPIO_bits_OUT) for [Anonymous Symbol]
    common.o(.text.delay_init) refers to misc.o(.text.SysTick_CLKSourceConfig) for SysTick_CLKSourceConfig
    common.o(.text.delay_init) refers to common.o(.bss..L_MergedGlobals) for [Anonymous Symbol]
    common.o(.ARM.exidx.text.delay_init) refers to common.o(.text.delay_init) for [Anonymous Symbol]
    common.o(.text.delay_us) refers to common.o(.bss..L_MergedGlobals) for [Anonymous Symbol]
    common.o(.ARM.exidx.text.delay_us) refers to common.o(.text.delay_us) for [Anonymous Symbol]
    common.o(.text.delay_xms) refers to common.o(.bss..L_MergedGlobals) for [Anonymous Symbol]
    common.o(.ARM.exidx.text.delay_xms) refers to common.o(.text.delay_xms) for [Anonymous Symbol]
    common.o(.text.delay_ms) refers to common.o(.text.delay_xms) for delay_xms
    common.o(.ARM.exidx.text.delay_ms) refers to common.o(.text.delay_ms) for [Anonymous Symbol]
    main.o(.text.main) refers to common.o(.text.delay_init) for delay_init
    main.o(.text.main) refers to usart1.o(.text.uart1_init) for uart1_init
```

6.2.4Removing Unused input sections from the image

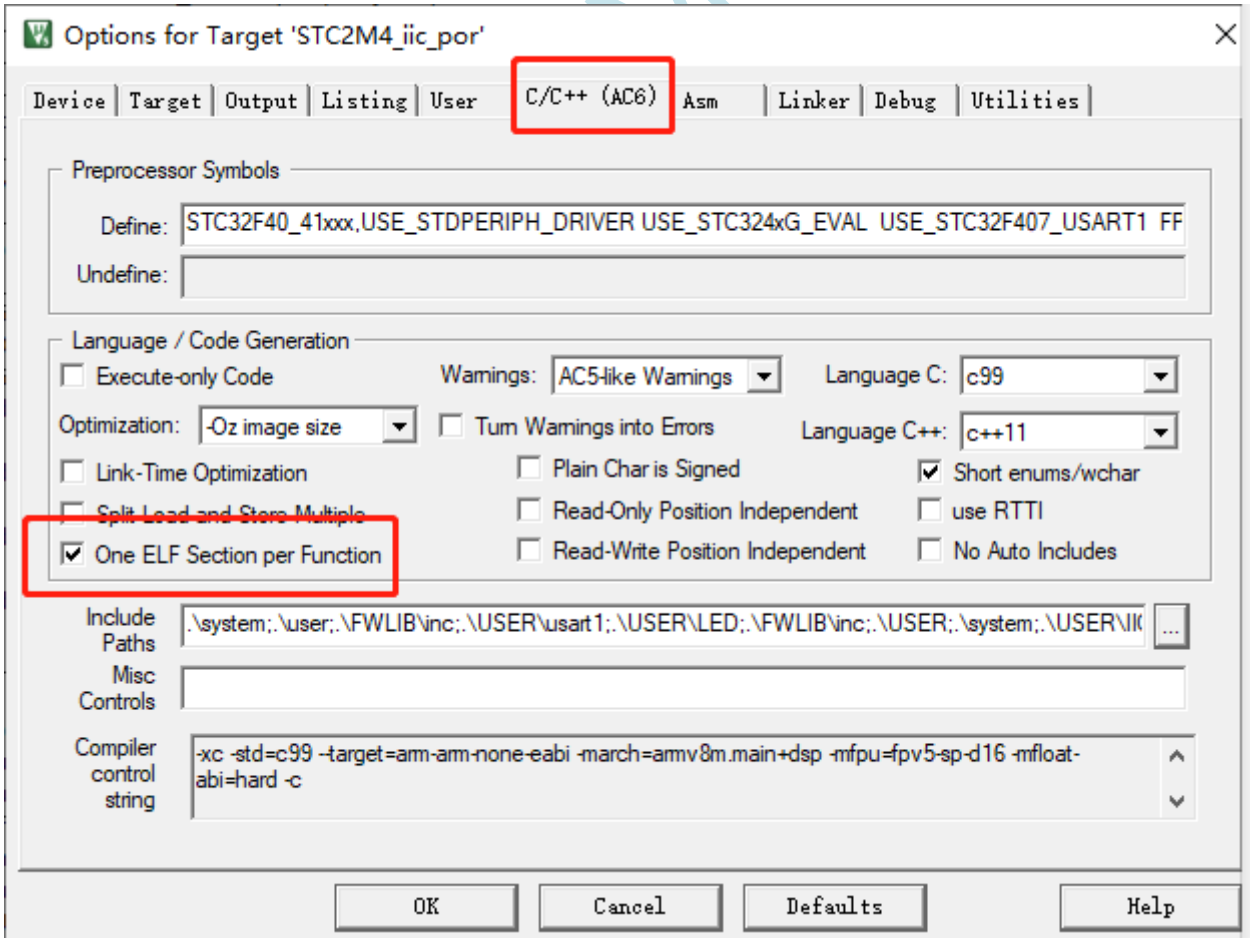
这部分主要是被删除的冗余函数，也就是添加到工程里面，但是没有调用到的，下面是部分被删除冗 余函数的截图效果:

main.c | i2c.c | ROM.map

```
Removing Unused input sections from the image.

    Removing startup_armv8mml.o(HEAP), (3072 bytes).
    Removing system_armv8mml.o(.text), (0 bytes).
    Removing system_armv8mml.o(.text.SystemCoreClockUpdate), (16 bytes).
    Removing system_armv8mml.o(.ARM.exidx.text.SystemCoreClockUpdate), (8 bytes).
    Removing system_armv8mml.o(.text.ITCM_Enable), (16 bytes).
    Removing system_armv8mml.o(.ARM.exidx.text.ITCM_Enable), (8 bytes).
    Removing system_armv8mml.o(.text.DTCM_Enable), (16 bytes).
    Removing system_armv8mml.o(.ARM.exidx.text.DTCM_Enable), (8 bytes).
    Removing system_armv8mml.o(.text.Flash_CACHE_Enable), (16 bytes).
    Removing system_armv8mml.o(.ARM.exidx.text.Flash_CACHE_Enable), (8 bytes).
    Removing system_armv8mml.o(.ARM.exidx.text.SystemInit), (8 bytes).
    Removing system_armv8mml.o(.text.Stc32_Clock_Init), (68 bytes).
    Removing system_armv8mml.o(.ARM.exidx.text.Stc32_Clock_Init), (8 bytes).
    Removing system_armv8mml.o(.text.Sys_Clock_Set), (212 bytes).
    Removing system_armv8mml.o(.ARM.exidx.text.Sys_Clock_Set), (8 bytes).
    Removing common.o(.text), (0 bytes).
    Removing common.o(.text.GPIO_group_OUT), (336 bytes).
    Removing common.o(.ARM.exidx.text.GPIO_group_OUT), (8 bytes).
    Removing common.o(.text.GPIO_bits_OUT), (80 bytes).
    Removing common.o(.ARM.exidx.text.GPIO_bits_OUT), (8 bytes).
    Removing common.o(.ARM.exidx.text.delay_init), (8 bytes).
    Removing common.o(.text.delay_us), (60 bytes).
    Removing common.o(.ARM.exidx.text.delay_us), (8 bytes).
    Removing common.o(.text.delay_xms), (64 bytes).
    Removing common.o(.ARM.exidx.text.delay_xms), (8 bytes).
    Removing common.o(.text.delay_ms), (54 bytes).
    Removing common.o(.ARM.exidx.text.delay_ms), (8 bytes).
    Removing main.o(.text), (0 bytes).
```

对于这个部分功能，用户最好将 MDK 中这个选项勾上，然后全编译工程，效果会比较好:



6.2.5Image Symbol Table

Image Symbol Table 主要分为两类，分别是 Local Symbols 和 Global Symbols。

- Local Symbols

LocalSymbols 记录了用 static 声明的全局变量地址和大小，C 文件中函数的地址和用 static 声明的函数代码大小，汇编文件中的标号地址（作用域限本文件）， 下面是部分截图:

```
Local Symbols

Symbol Name                              Value     Ov Type        Size  Object(Section)

../clib/microlib/init/entry.s            0x00000000   Number         0  entry.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry2.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry5.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry7a.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry7b.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry8a.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry8b.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry9a.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry9b.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry10a.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry10b.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry11a.o ABSOLUTE
../clib/microlib/init/entry.s            0x00000000   Number         0  entry11b.o ABSOLUTE
../clib/microlib/stdio/puts.c            0x00000000   Number         0  puts.o ABSOLUTE
../clib/microlib/stdio/puts.c            0x00000000   Number         0  puts_e.o ABSOLUTE
../clib/microlib/stdio/streams.c         0x00000000   Number         0  stdout.o ABSOLUTE
common.c                                 0x00000000   Number         0  common.o ABSOLUTE
dc.s                                     0x00000000   Number         0  dc.o ABSOLUTE
delay.c                                  0x00000000   Number         0  delay.o ABSOLUTE
handlers.s                               0x00000000   Number         0  handlers.o ABSOLUTE
i2c.c                                    0x00000000   Number         0  i2c.o ABSOLUTE
init.s                                   0x00000000   Number         0  init.o ABSOLUTE
main.c                                   0x00000000   Number         0  main.o ABSOLUTE
misc.c                                   0x00000000   Number         0  misc.o ABSOLUTE
stc32f4xx_gpio.c                         0x00000000   Number         0  stc32f4xx_gpio.o ABSOLUTE
stc32f4xx_i2c.c                          0x00000000   Number         0  stc32f4xx_i2c.o ABSOLUTE
stc32f4xx_rcc.c                          0x00000000   Number         0  stc32f4xx_rcc.o ABSOLUTE
stc32f4xx_syscfg.c                       0x00000000   Number         0  stc32f4xx_syscfg.o ABSOLUTE
stc32f4xx_usart.c                        0x00000000   Number         0  stc32f4xx_usart.o ABSOLUTE
system\startup_ARMv8MML.s                0x00000000   Number         0  startup_armv8mml.o ABSOLUTE
```

Global Symbols

Global Symbols 记录了全局变量的地址和大小，C 文件中函数的地址及其代码大小，汇编文件中的标号地址（作用域全工程），下面是部分截图:

```
Global Symbols

Symbol Name                             Value     Ov Type        Size  Object(Section)

BuildAttributes$$THM_ISAv4$E$P$D$K$B$S$7EM$8M$VFPi5$EXTD16$VFPS$VFMA$PE$A:L22UL41UL21$X:L11$S22US41US21$IEEE1$
__cpp_initialize__aeabi_                 - Undefined Weak Reference
__cxa_finalize                           - Undefined Weak Reference
__decompress                             - Undefined Weak Reference
_clock_init                              - Undefined Weak Reference
_microlib_exit                           - Undefined Weak Reference
__Vectors_Size                          0x00000188   Number         0  startup_armv8mml.o ABSOLUTE
__Vectors                               0x08000000   Data           4  startup_armv8mml.o(RESET)
__Vectors_End                           0x08000188   Data           0  startup_armv8mml.o(RESET)
__main                                  0x08000189   Thumb Code     0  entry.o(.ARM.Collect$$$$00000000)
_main_stk                               0x08000189   Thumb Code     0  entry2.o(.ARM.Collect$$$$00000001)
_main_scatterload                       0x0800018d   Thumb Code     0  entry5.o(.ARM.Collect$$$$00000004)
__main_after_scatterload                0x08000191   Thumb Code     0  entry5.o(.ARM.Collect$$$$00000004)
_main_clock                             0x08000191   Thumb Code     0  entry7b.o(.ARM.Collect$$$$00000008)
_main_cpp_init                          0x08000191   Thumb Code     0  entry8b.o(.ARM.Collect$$$$0000000A)
_main_init                              0x08000191   Thumb Code     0  entry9a.o(.ARM.Collect$$$$0000000B)
__rt_final_cpp                          0x08000199   Thumb Code     0  entry10a.o(.ARM.Collect$$$$0000000D)
__rt_final_exit                         0x08000199   Thumb Code     0  entry11a.o(.ARM.Collect$$$$0000000F)
Reset_Handler                           0x0800019d   Thumb Code     8  startup_armv8mml.o(.text)
NMI_Handler                             0x080001a5   Thumb Code     2  startup_armv8mml.o(.text)
HardFault_Handler                       0x080001a7   Thumb Code     2  startup_armv8mml.o(.text)
MemManage_Handler                       0x080001a9   Thumb Code     2  startup_armv8mml.o(.text)
BusFault_Handler                        0x080001ab   Thumb Code     2  startup_armv8mml.o(.text)
UsageFault_Handler                      0x080001ad   Thumb Code     2  startup_armv8mml.o(.text)
SecureFault_Handler                     0x080001af   Thumb Code     2  startup_armv8mml.o(.text)
SVC_Handler                             0x080001b1   Thumb Code     2  startup_armv8mml.o(.text)
DebugMon_Handler                        0x080001b3   Thumb Code     2  startup_armv8mml.o(.text)
PendSV_Handler                          0x080001b5   Thumb Code     2  startup_armv8mml.o(.text)
SysTick_Handler                         0x080001b7   Thumb Code     2  startup_armv8mml.o(.text)
ADC_IRQHandler                          0x080001b9   Thumb Code     0  startup_armv8mml.o(.text)
CAN1_RX0_IRQHandler                     0x080001b9   Thumb Code     0  startup_armv8mml.o(.text)
CAN1_RX1_IRQHandler                     0x080001b9   Thumb Code     0  startup_armv8mml.o(.text)
CAN1_SCE_IRQHandler                     0x080001b9   Thumb Code     0  startup_armv8mml.o(.text)
CAN1_TX_IRQHandler                      0x080001b9   Thumb Code     0  startup_armv8mml.o(.text)
```

6.2.6Memory Map of the image

映像文件可以分为加载域（Load Region）和运行域（Execution Region):加载域反映了 ARM 可 执行映像文件的各个段存放在存储器中的位置关系。下面是部分截图，另外映像中的入口点就是程序开始 执行的位置。

```
Memory Map of the image

  Image Entry point : 0x08000189

  Load Region LR_IROM1 (Base: 0x08000000, Size: 0x00000d18, Max: 0x00020000, ABSOLUTE)

    Execution Region ER_IROM1 (Exec base: 0x08000000, Load base: 0x08000000, Size: 0x00000d08, Max: 0x00020000, ABSOLUTE)

    Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x08000000   0x08000000   0x00000188   Data   RO          3    RESET               startup_armv8mml.o
    0x08000188   0x08000188   0x00000000   Code   RO        476  * .ARM.Collect$$$$00000000  mc_w.l(entry.o)
    0x08000188   0x08000188   0x00000004   Code   RO        483    .ARM.Collect$$$$00000001  mc_w.l(entry2.o)
    0x0800018c   0x0800018c   0x00000004   Code   RO        486    .ARM.Collect$$$$00000004  mc_w.l(entry5.o)
    0x08000190   0x08000190   0x00000000   Code   RO        488    .ARM.Collect$$$$00000008  mc_w.l(entry7b.o)
    0x08000190   0x08000190   0x00000000   Code   RO        490    .ARM.Collect$$$$0000000A  mc_w.l(entry8b.o)
    0x08000190   0x08000190   0x00000008   Code   RO        491    .ARM.Collect$$$$0000000B  mc_w.l(entry9a.o)
    0x08000198   0x08000198   0x00000000   Code   RO        493    .ARM.Collect$$$$0000000D  mc_w.l(entry10a.o)
    0x08000198   0x08000198   0x00000000   Code   RO        495    .ARM.Collect$$$$0000000F  mc_w.l(entry11a.o)
    0x08000198   0x08000198   0x00000004   Code   RO        484    .ARM.Collect$$$$00002712  mc_w.l(entry2.o)
    0x0800019c   0x0800019c   0x00000028   Code   RO          4    .text               startup_armv8mml.o
    0x080001c4   0x080001c4   0x00000024   Code   RO        498    .text               mc_w.l(init.o)
    0x080001e8   0x080001e8   0x00000010   Code   RO         90    .text.Delay         delay.o
    0x080001f8   0x080001f8   0x0000007e   Code   RO        156    .text.GPIO_Init     stc32f4xx_gpio.o
    0x08000276   0x08000276   0x00000028   Code   RO        180    .text.GPIO_PinAFConfig  stc32f4xx_gpio.o
    0x0800029e   0x0800029e   0x00000012   Code   RO        210    .text.I2C_AcknowledgeConfig  stc32f4xx_i2c.o
```

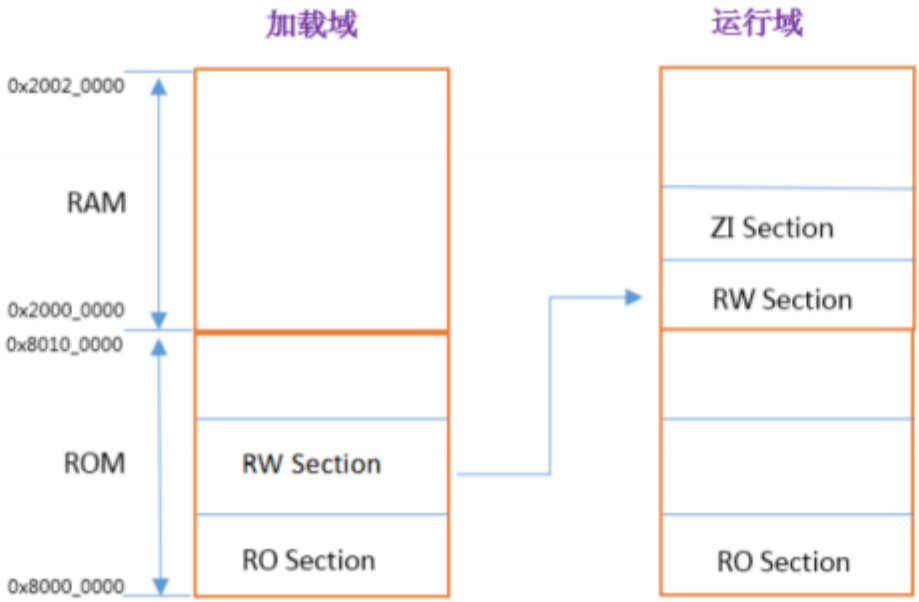
运行域反映了 ARM 可执行映像文件各个段真正执行时在存储器中的位置关系：

```
Execution Region RW_IRAM1 (Exec base: 0x20000000, Load base: 0x08000d08, Size: 0x00000618, Max: 0x00020000, ABSOLUTE)

Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

0x20000000   0x08000d08   0x00000004   Data   RW        497    .data               mc_w.l(stdout.o)
0x20000004   0x08000d0c   0x00000004   Data   RW        120    .data.I2CTimeout    i2c.o
0x20000008   0x08000d10   0x00000004   Data   RW         25    .data.SystemCoreClock  system_armv8mml.o
0x2000000c   0x08000d14   0x00000004   PAD
0x20000010         -      0x00000003   Zero   RW         47    .bss..L_MergedGlobals  common.o
0x20000013   0x08000d14   0x00000001   PAD
0x20000014         -      0x00000002   Zero   RW        119    .bss.EEPROM_ADDRESS  i2c.o
0x20000016         -      0x000000ff   Zero   RW        123    .bss.I2C1_rx_buffer  i2c.o
0x20000115         -      0x000000ff   Zero   RW        122    .bss.I2C1_tx_buffer  i2c.o
0x20000214         -      0x00000001   Zero   RW         79    .bss.rec_data1      usart1.o
0x20000215   0x08000d14   0x00000003   PAD
0x20000218         -      0x00000400   Zero   RW          1    STACK               startup_armv8mml.o
```

简单的说，加载域就是程序在 Flash 中的实际存储，而运行域是芯片上电后的运行状态，通过下面的框图可以有一个感性的认识：



通过上面的框图可以看出，RW 区也是要存储到 ROM/Flash 里面的，在执行映像之前，必须将已初始化的 RW 数据从 ROM 中复制到 RAM 中的执行地址并创建 ZI Section （初始化为 0 的变量区）。

注：0x8010_0000 和 0x8000_0000 要改成 0x0801_0000 和 0x0800_0000。

6.2.7Image component sizes

Image component sizes 映像组件大小比较重要，比如下面内容代表的含义：

```
Image component sizes

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

        24          4          0          0          3       3325   common.o
        16          0          0          0          0        693   delay.o
       900         40          0          4        512       5661   i2c.o
       212         84          0          0          0        553   main.o
       124         16          0          0          0       3766   misc.o
        40         10        392          0       1024        956   startup_armv8mml.o
       166          0          0          0          0       4237   stc32f4xx_gpio.o
       366         12          0          0          0       8258   stc32f4xx_i2c.o
       204         28         16          0          0      12757   stc32f4xx_rcc.o
       300         12          0          0          0       7335   stc32f4xx_usart.o
       204         36          0          4          0       4708   system_armv8mml.o
       212         16          0          0          1       5778   usart1.o

    ----------------------------------------------------------------------
      2772        258        440          8       1548      58027   Object Totals
         0          0         32          0          0          0   (incl. Generated)
         4          0          0          0          8          0   (incl. Padding)

    ----------------------------------------------------------------------

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Member Name

         0          0          0          0          0          0   entry.o
         0          0          0          0          0          0   entry10a.o
         0          0          0          0          0          0   entry11a.o
         8          4          0          0          0          0   entry2.o
         4          0          0          0          0          0   entry5.o
         0          0          0          0          0          0   entry7b.o
         0          0          0          0          0          0   entry8b.o
         8          4          0          0          0          0   entry9a.o
        30          0          0          0          0          0   handlers.o
        36          8          0          0          0         68   init.o
        36          6          0          0          0         80   puts.o
         0          0          0          4          0          0   stdout.o
```

◆ Code (inc. Data) ：显示代码占用了多少字节。 在此映像中， 有 19442 字节的代码， 其中包括 1832 字节的内联数据 (inc. data)，例如文字池和短字符串。◆ RO Data ：显示只读数据占用了多少字节（比如 const char buf[] = "123456"）。这是除 Code (inc. data) 列中包括的内联数据之外的数据。 ◆ RW Data：显示读写数据占用了多少字节。 ◆ ZI Data ：显示零初始化的数据占用了多少字节。◆ Debug ：显示调试数据占用了多少字节，例如，调试输入节以及符号和字符串。 ◆ Object Totals：显示链接到一起以生成映像的对象占用了多少字节。 ◆ (incl. Generated)：链接器会生成的映像内容，例如，交互操作中间代码。如果 Object Totals 行 包含此类型的数据，则会显示在该行中。本例中共有1016 字节的RO 数据，其中32 字节是链接 器生成的 RO 数据。◆ (incl. Padding)：链接器根据需要插入填充，以强制字节对齐。

下面的 Library Totals 显示已提取并作为单个对象添加到映像中的库成员占用了多少字节。

```
  Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Member Name

     0          0          0          0          0          0   entry.o
     0          0          0          0          0          0   entry10a.o
     0          0          0          0          0          0   entry11a.o
     8          4          0          0          0          0   entry2.o
     4          0          0          0          0          0   entry5.o
     0          0          0          0          0          0   entry7b.o
     0          0          0          0          0          0   entry8b.o
     8          4          0          0          0          0   entry9a.o
    30          0          0          0          0          0   handlers.o
    36          8          0          0          0         68   init.o
    30          0          0          0          0         68   llshl.o
    32          0          0          0          0         68   llushr.o
    36          0          0          0          0        108   memseta.o
  1154         54          0          0          0        352   printf8.o
     0          0          0          4          0          0   stdout.o
    98          0          0          0          0         92   uldiv.o

--------------------------------------------------------------------------------
  1438         70          0          4          0        756   Library Totals
     2          0          0          0          0          0   (incl. Padding)
```

下面是我们经常要查看的内容:

```
==============================================================================

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

     20880       1902       1016         64       8568    1452680   Grand Totals
     20880       1902       1016         64       8568    1452680   ELF Image Totals
     20880       1902       1016         64          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                21896 (  21.38kB)
    Total RW  Size (RW Data + ZI Data)              8632 (   8.43kB)
    Total ROM Size (Code + RO Data + RW Data)      21960 (  21.45kB)

==============================================================================
```

◆ Grand Totals：显示映像的真实大小。◆ ELF Image Totals：ELF(Executable and Linking Format)可执行链接格式映像文件大小。◆ ROM Totals：显示包含映像所需的ROM 的最小大小。这不包括 ZI 数据和存储在 ROM 中的调试 信息。

6.3HTM文件分析

MDK 将工程全编译， 且没有错误后， 就会生成此文件。以开发板配套例子为例， htm 文件位于路径: \Project\MDK-ARM(uV5)\Objects。 此文件的最大作用就是基本统计了所有被调用函数的栈 stack 使用情况（不考虑中断嵌套）。下面是 整个工程的最大栈需求:

```
<BR><P>
<H3>Maximum Stack Usage =         88 bytes + Unknown(Cycles, Untraceable Function Pointers)</H3><H3>
Call chain for Maximum Stack Depth:</H3>
main &rArr; uart1_init &rArr; USART_Init &rArr; RCC_GetClocksFreq
```

具体到每个函数也给出最大的栈深度 Max Depth，同时也给出函数本身的代码量大小和使用的栈大小，比如函数uart1_init，最大栈深度是 88 字节。函数本身占用代码大小 (Thumb 指令集) 140 字 节，使用栈 48 字节。

```
<P><STRONG><a name="[82]"></a>uart1_init</STRONG> (Thumb, 140 bytes, Stack size 48 bytes, usart1.o(.text.uart1_init))
<BR><BR>[Stack]<UL><LI>Max Depth = 88<LI>Call Chain = uart1_init &rArr; USART_Init &rArr; RCC_GetClocksFreq
</UL>
<BR>[Calls]<UL><LI><a href="#[7c]">&gt;&gt;</a>   USART_Init
```

通过这个文件，我们可以合理的配置启动文件里面的stack大小.

6.4总结

本章节的知识点比较重要，务必要熟练掌握。平时开发工程项目时，可以多关注 map 文件和 htm 文 件。

STC MCU

第7章 STC32M4启动过程详解

本章教程主要跟大家讲 STC32M4 的启动过程，这里的启动过程是指从 CPU 上电复位执行第 1 条指令开始 (汇编文件) 到进入 C 程序 main()函数入口之间的部分。启动过程相对来说还是比较重要的， 理 解了这个过程，对于以后分析程序还是有些帮助的，要不每次看到这个启动过程都会跳过， 直接去看主程 序了。对初学者来说， 看这个可能有些吃力， 不过不要紧，随着自己做过一些简单的应用之后再来看这章， 应该会有很多的帮助，由于我们的 M4 板子是基于 STC32M4XXX，所以我们这里主要针对 M4 系列的启动过程做一下分析。

7.1 初学者重要提示

7.2 启动文件分析

7.3 总结

7.1初学者重要提示

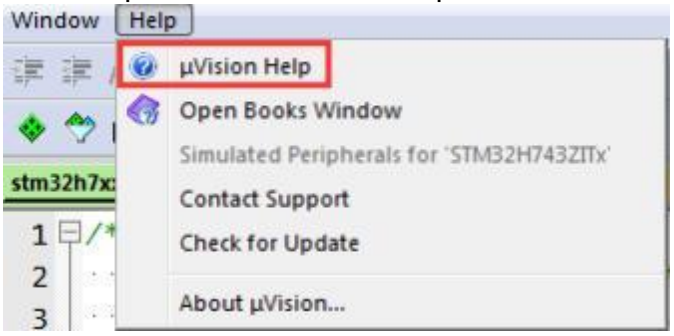
- 在学习启动代码得时候，会涉及到ARM得汇编指令和Cortex内核的指令，有关Cortex内核的指令我们可以参考《CM3权威指南CnR2》第四章：指令集。剩下的ARM汇编指令我们可以在MDK->Help->Uvision Help中搜索到 。

。

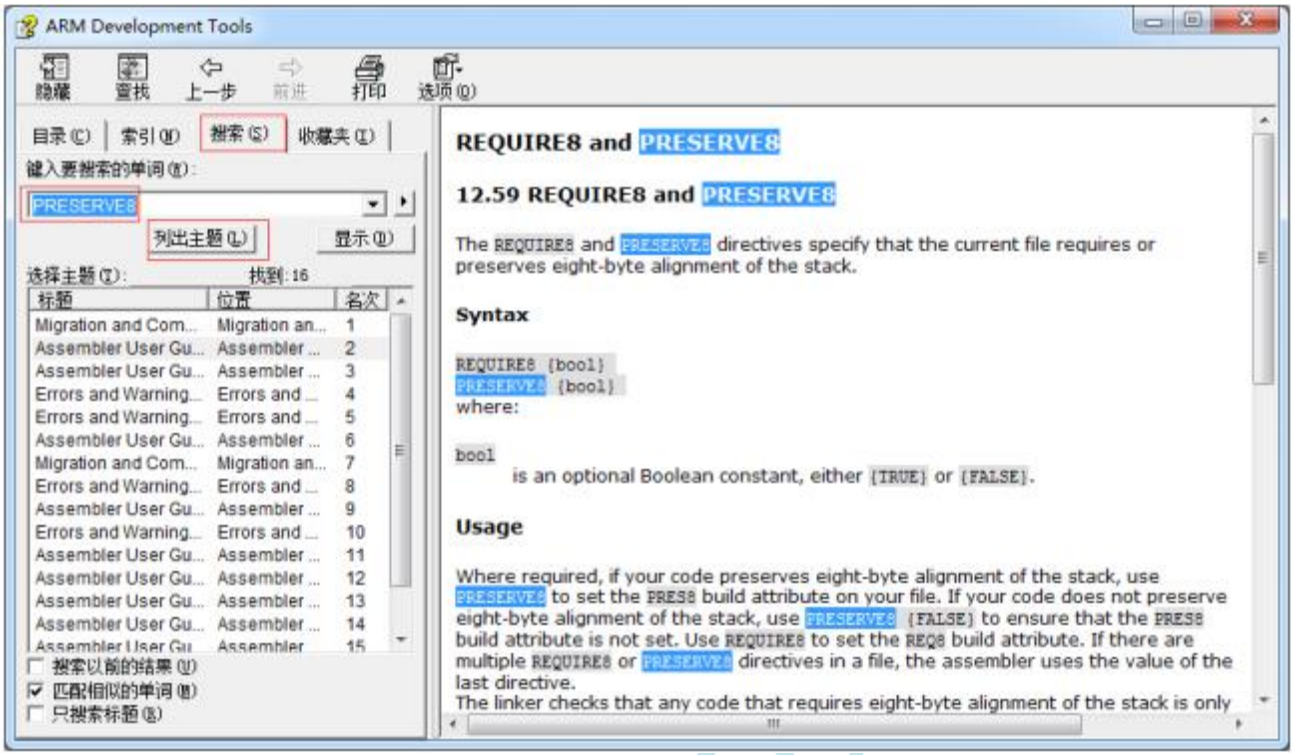
7.2启动文件分析

鉴于STC32M4开发板使用的是 STC32M4，下面我们详细的分析一下启动文件 startup_ARMv8MML.s。分析前，先掌握一个小技能，遇到不认识的指令或者关键词可以检索。

- 启动 MDK 软件， 在 Help 菜单点击 uVision Help



- 点击后弹出如下文件



在搜索栏输入你需要查询的单词进行查询， 然后点击“列出主题”按钮， 会将相关的知识点都罗列出来。此功能非常实用，建议熟练掌握。

下面先来看启动文件前面的介绍 (固件库版本： V1.2.0)

```
;******************** (C) COPYRIGHT 2017 STMicroelectronics ********************
;* File Name          : startup_STC32M443xx.s
;* @author  MCD Application Team
;* version            : V1.2.0
;* Date               : 29-December-2017
;* Description        : STC32M4xx devices vector table for MDK-ARM toolchain.
;*                      This module performs:
;*                      - Set the initial SP
;*                      - Set the initial PC == Reset_Handler
;*                      - Set the vector table entries with the exceptions ISR address
;*                      - Branches to __main in the C library (which eventually
;*                        calls main()).
;*                      After Reset the Cortex-M processor is in Thread mode,
;*                      priority is Privileged, and the Stack is set to Main.
;* <<< Use Configuration Wizard in Context Menu >>>
;*******************************************************************************
;
; Licensed under MCD-ST Liberty SW License Agreement V2, (the "License"); ;
You may not use this file except in compliance with the License.        ; You
may obtain a copy of the License at:
;
;        http://www.st.com/software_license_agreement_liberty_v2
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS,
; WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See
; the License for the specific language governing permissions and
```

```
; limitations under the License.
;
;*******************************************************************************
```

启动文件是后缀为.s 的汇编语言文本文件， 每行前面的分号表示此行是注释行。

启动文件主要完成如下工作，即程序执行过程：

- 设置堆栈指针 SP = __initial_sp。
- 设置 PC 指针 = Reset_Handler。
- 设置中断向量表。
- 配置系统时钟。
- 配置外部 SRAM/SDRAM 用于程序变量等数据存储（这是可选的） 。
- 跳转到 C 库中的 __main ，最终会调用用户程序的 main()函数。

Cortex-M 内核处理器复位后， 处于线程模式，指令权限是特权级别（最高级别） ，堆栈设置为使用主堆 栈 MSP。

7.3.1复位序列

硬件复位之后，CPU 内的时序逻辑电路首先完成如下两个工作（程序代码下载到内部 flash 为例，flash 首地址 0x0800 0000）

- 将 0x08000000 位置存放的堆栈栈顶地址存放到 SP 中(MSP)。
- 将 0x08000004 位置存放的向量地址装入 PC 程序计数器。

CPU 从 PC 寄存器指向的物理地址取出第 1 条指令开始执行程序，也就是开始执行复位中断服务程 序 Reset_Handler。



复位中断服务程序会调用 SystemInit()函数来配置系统时钟、配置 FMC 总线上的外部 SRAM/SDRAM ，然后跳转到 C 库中__main 函数。由 C 库中的__main 函数完成用户程序的初始化工作（比如： 变量赋初 值等） ，最后由__main 函数调用用户写的 main()函数开始执行 C 程序。

7.3.2代码分析

- 第 1 部分代码分析

下面的代码实现开辟栈(stack)空间, 用于局部变量、 函数调用、 函数的参数等。

```
1.  ; Amount of memory (in bytes) allocated for Stack ;
2.  Tailor this value to your application needs
```

STC MCU

```
3.    ; <h> Stack Configuration
4.    ;   <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
5.    ; </h>
6.
7.    Stack_Size      EQU     0x00000400
8.
9.                    AREA    STACK, NOINIT, READWRITE, ALIGN=3
10.   Stack_Mem       SPACE   Stack_Size
11.   __initial_sp
```

第 7 行: EQU 是表示宏定义的伪指令， 类似于 C 语言中的#define。伪指令的意思是指这个“指令”并不会生成二进制程序代码，也不会引起变量空间分配。

0x00000400 表示栈大小，注意这里是以字节为单位。

第 9 行: 开辟一段数据空间可读可写， 段名 STACK， 按照 8 字节对齐。 ARER 伪指令表示下面将开始定义一个代码段或者数据段。此处是定义数据段。 ARER 后面的关键字表示这个段的属性。

STACK ：表示这个段的名字，可以任意命名。

NOINIT: 表示此数据段不需要填入初始数据。

READWRITE：表示此段可读可写。

ALIGN=3 ：表示首地址按照 2 的 3 次方对齐， 也就是按照 8 字节对齐(地址对 8 求余数等于 0)。

第 10 行: SPACE 这行指令告诉汇编器给 STACK 段分配 0x00000400 字节的连续内存空间。

第 11 行: __initial_sp 紧接着 SPACE 语句放置， 表示了栈顶地址。 __initial_sp 只是一个标号， 标号主要用于表示一片内存空间的某个位置，等价于 C 语言中的“地址”概念。地址仅仅表示存储空间的一个位置，从 C 语言的角度来看，变量的地址，数组的地址或是函数的入口地址在本质上并无区别。

- 第 2 部分代码分析

下面的代码实现开辟堆(heap)空间， 主要用于动态内存分配， 也就是说用 malloc，calloc, realloc 等函数分配的变量空间是在堆上。

```
1.    ; <h> Heap Configuration
2.    ;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
3.    ; </h>
4.
5.    Heap_Size       EQU     0x00000200
6.
7.                    AREA    HEAP, NOINIT, READWRITE, ALIGN=3
8.    __heap_base
9.    Heap_Mem        SPACE   Heap_Size
10.   __heap_limit
```

这几行语句和上面第 1 部分代码类似。分配一片连续的内存空间给名字叫 HEAP 的段，也就是分配堆空间。堆的大小为 0x00000200。

__heap_base 表示堆的开始地址。

__heap_limit 表示堆的结束地址。

- 第 3 部分代码分析

```
1.                  PRESERVE8
2.                  THUMB
3.
4.
5.   ; Vector Table Mapped to Address 0 at Reset
6.                  AREA    RESET, DATA, READONLY
7.                  EXPORT  __Vectors
8.                  EXPORT  __Vectors_End
9.                  EXPORT  __Vectors_Size
```

第 1 行: PRESERVE8 指定当前文件保持堆栈八字节对齐。

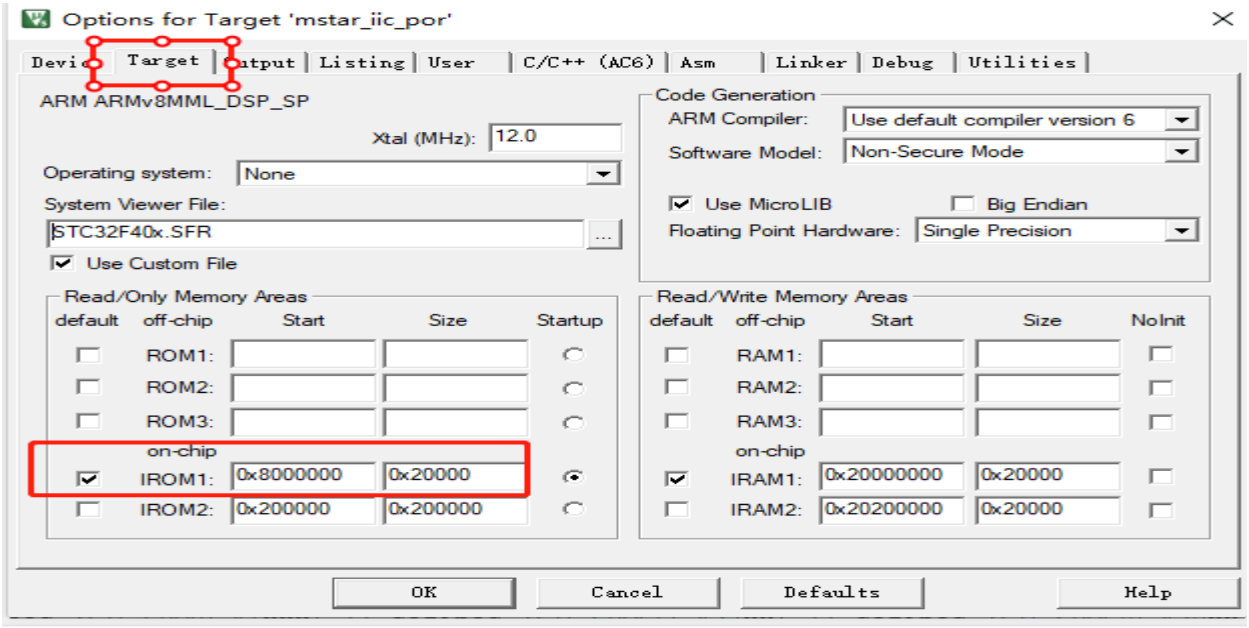
第 2 行: THUMB 表示后面的指令是THUMB 指令集 ，CM7 采用的是 THUMB - 2 指令集。

第 6 行: AREA 定义一块数据段, 只读, 段名字是 RESET。 READONLY 表示只读, 缺省就表示代码段了。 第 7-9 行: 3 行 EXPORT 语句将 3 个标号申明为可被外部引用, 主要提供给链接器用于连接库文件或 其他文件。

- 第 4 部分代码分析

```
1.  __Vectors       DCD     __initial_sp                ; Top of Stack
2.                  DCD     Reset_Handler               ; Reset Handler
3.                  DCD     NMI_Handler                 ; NMI Handler
4.                  DCD     HardFault_Handler           ; Hard Fault Handler
5.
6.                  中间部分省略未写
7.
8.                  DCD     0                           ; Reserved
9.                  DCD     WAKEUP_PIN_IRQHandler       ; Interrupt for all 6 wake-up pins
10.
11.
12. __Vectors_End
13.
14. __Vectors_Size  EQU  __Vectors_End - __Vectors
```

上面的这段代码是建立中断向量表，中断向量表定位在代码段的最前面。具体的物理地址由链接器的配置参数 (IROM1 的地址) 决定。如果程序在 Flash 运行, 则中断向量表的起始地址是 0x08000000 。



DCD 表示分配 1 个 4 字节的空间。每行 DCD 都会生成一个 4 字节的二进制代码。中断向量表 存放的实际上是中断服务程序的入口地址。当异常（也即是中断事件） 发生时， CPU 的中断系统会将相 应的入口地址赋值给 PC 程序计数器，之后就开始执行中断服务程序。

- 第 5 部分代码分析

```
1.                  AREA     |.text|, CODE, READONLY
2.
3.    ; Reset handler
4.    Reset_Handler    PROC
5.                     EXPORT  Reset_Handler                  [WEAK]
6.             IMPORT  SystemInit
7.             IMPORT  __main
8.
9.                     LDR     R0, =SystemInit
10.                    BLX     R0
11.                    LDR     R0, =__main
12.                    BX      R0
13.                    ENDP
```

第 1 行： AREA 定义一块代码段，只读， 段名字是 .text 。READONLY 表示只读。

第 4 行： 利用 PROC、 ENDP 这一对伪指令把程序段分为若干个过程， 使程序的结构更清晰。

第 5 行: WEAK 声明其他的同名标号优先于该标号被引用,就是说如果外面声明了的话会调用外面的。 这个声明很重要, 它让我们可以在 C 文件中任意地方放置中断服务程序, 只要保证 C 函数的名字和向量表中的名字一致即可。

第 6 行: IMPORT: 伪指令用于通知编译器要使用的标号在其他的源文件中定义。但要在当前源文件中引用，而且无论当前源文件是否引用该标号，该标号均会被加入到当前源文件的符号表中。

第 9 行: SystemInit 函数在文件 system_STC32M4xx.c 里面, 主要实现 RCC 相关寄存器复位和中断向 量表位置设置。

第 11 行: __main 标号表示 C/C++标准实时库函数里的一个初始化子程序__main 的入口地址。该程序 的一个主要作用是初始化堆栈(跳转__user_initial_stackheap 标号进行初始化堆栈的, 下面会讲到这个标 号)，并初始化映像文件, 最后跳转到 C 程序中的 main 函数。这就解释了为何所有的 C 程序必须有一 个 main 函数作为程序的起点。因为这是由 C/C++标准实时库所规, 并且不能更改。

◆ 第 6 部分代码分析

代码如下:

```
1.  ; Dummy Exception Handlers (infinite loops which can be modified)
2.
3.  NMI_Handler     PROC
4.                  EXPORT                                  [WEAK]
5.              NMI_Handler
6.                  B
7.
8.                  ENDP
9.  HardFault_Handler\
10.                 PROC                                    [WEAK]
11.                 EXPORT  HardFault_Handler
12.                 B
13.                 .
14.                 ENDP
15.
16.                 中间部分省略未写
17. Default_Handler PROC
18.                                                         [WEAK]
19.                 EXPORT  WWDG_IRQHandler                 [WEAK]
20.                 EXPORT  PVD_AVD_IRQHandler              [WEAK]
21.                 EXPORT  TAMP_STAMP_IRQHandler
22.                 中间部分省略未写
23. SAI4_IRQHandler
24. WAKEUP_PIN_IRQHandler
25.                 .
26.                 B
27.
28.                 ENDP
29.
30.                 ALIGN
```

第 5 行: 死循环, 用户可以在此实现自己的中断服务程序。 不过很少在这里实现中断服务程序, 一般多是 在其它的 C 文件里面重新写一个同样名字的中断服务程序,因为这里是 WEEK 弱定义的。 如果没有在其 它文件中写中断服务器程序,且使能了此中断,进入到这里后,会让程序卡在这个地方。

第 14 行: 缺省中断服务程序 (开始)

第 23 行: 死循环,如果用户使能中断服务程序,而没有在 C 文件里面写中断服务程序的话, 都会进入到这里。 比如在程序里面使能了串口 1 中断, 而没有写中断服务程序 USART1_IRQHandle ,那么串口中断 来了, 会进入到这个死循环。

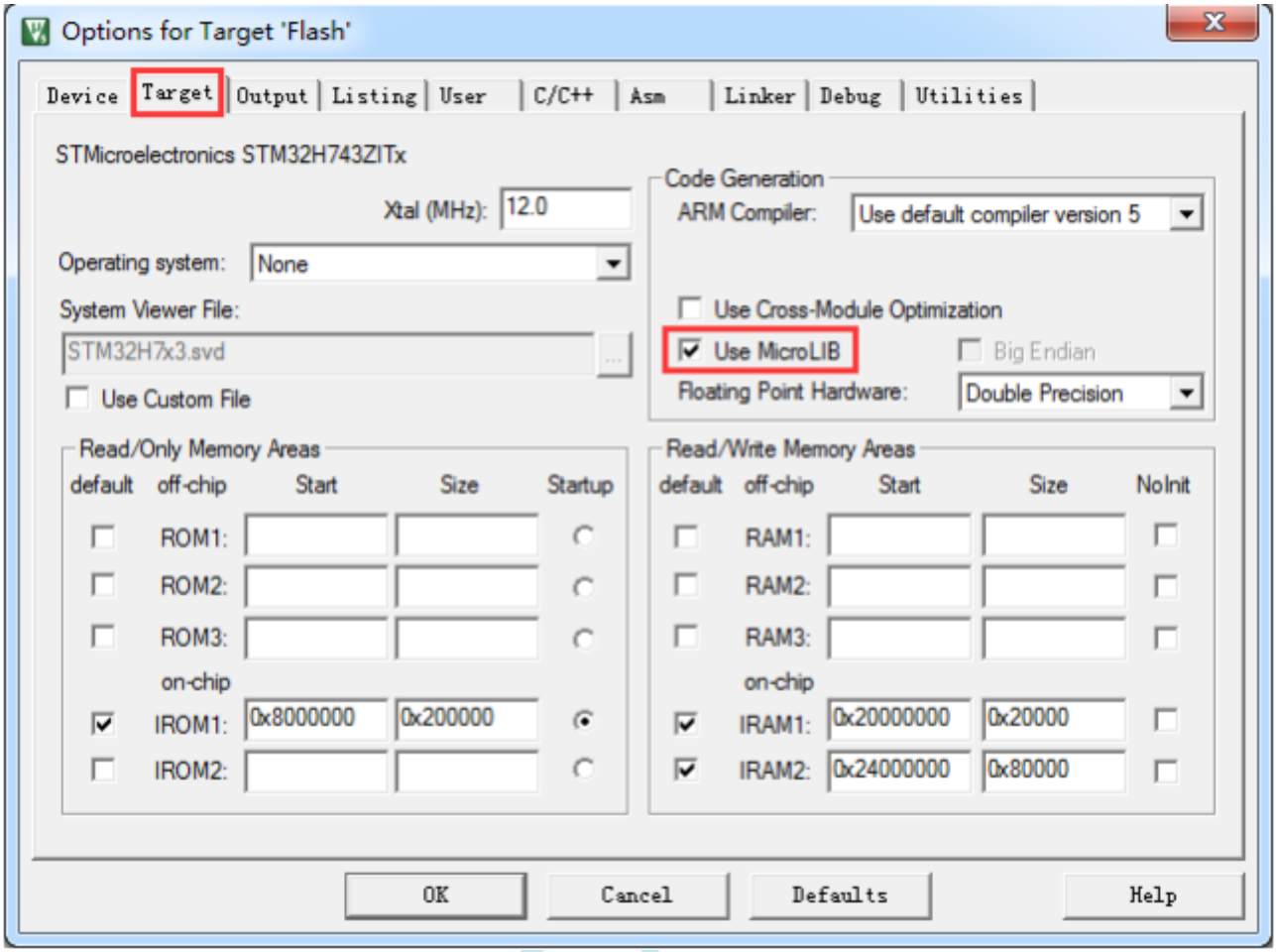
第 25 行: 缺省中断服务程序 (结束) 。

- 第 7 部分代码分析

启动代码的最后一部分:

```
1.  ;*******************************************************************************
2.  ; User Stack and Heap initialization
3.  ;*******************************************************************************
4.                  IF      :DEF:__MICROLIB
5.
6.                  EXPORT  __initial_sp
7.                  EXPORT  __heap_base
8.                  EXPORT  __heap_limit
9.
10.                 ELSE
11.
12.                 IMPORT  __use_two_region_memory
13.                 EXPORT  __user_initial_stackheap
14.
15. __user_initial_stackheap
16.
17.                 LDR     R0, =  Heap_Mem
18.                 LDR     R1, =(Stack_Mem + Stack_Size)
19.                 LDR     R2, = (Heap_Mem +  Heap_Size)
20.                 LDR     R3, = Stack_Mem
21.                 BX      LR
22.
23.                 ALIGN
24.
25.                 ENDIF
26.
27.                 END
```

第 4 行: 简单的汇编语言实现 IF.....ELSE.....语句。如果定义了 MICROLIB, 那么程序是不会执行 ELSE 分支的代码。 __MICROLIB 可能大家并不陌生,就在 MDK 的Target Option 里面设置。



第 5 行: __user_initial_stackheap 将由__main 函数进行调用。

- MicroLib

MicroLib 是MDK里面带的微库, 针对嵌入式应用, MicroLIB做了深度优化, 比使用C标准库所 需的RAM和FLASH 空间都大大减小比如调用: <math.h>, <stdlib.h>, <stdio.h>, <string.h>

更多相关知识可以地址: http://www.keil.com/arm/microlib.asp。

7.3总结

本章节讲解的启动过程分析还是比较重要的, 忘初学者务必掌握。

第8章 STC32M4 的电源

本章教程继续为大家讲解学习STC32M4 的必备知识点电源。

8.1 初学者重要提示

8.2 电源监控器

8.3 低功耗模式

8.4 总结

8.1初学者重要提示

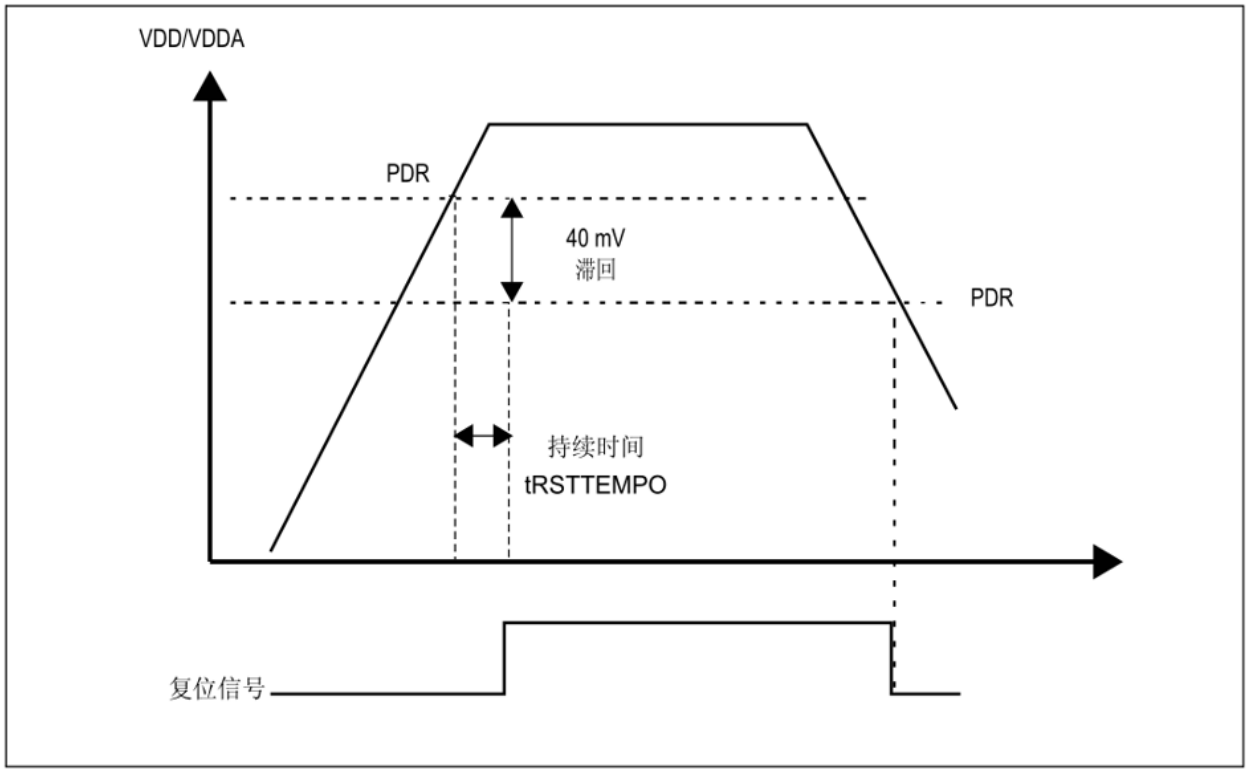
- 电源管理部分涉及到的各种低功耗方式会在后面章节中为大家讲解， 当前阶段仅需了解低功耗属于电 源管理部分即可。

8.2 电源监控器

电源是系统稳定运行的根本，主要分为以下几个知识点， 电源供电、供电监控、电源管理和低功耗。 当前阶段主要了解电源供电和硬件上电时序。

8.2.1 上电复位（POR）/掉电复位（PDR）

本器件内部集成有 POR/PDR 电路，可以从 1.8 V 开始正常工作。
当 V DD /V DDA 低于指定阈值 V POR/PDR 时，器件无需外部复位电路便会保持复位状态。
上电复位/ 掉电复位波形如下



8.2.2 欠压复位（BOR)

上电期间，欠压复位 (BOR) 将使器件保持复位状态，直到电源电压达到指定的 V BOR 阈值。

V BOR 通过器件选项字节进行配置。BOR 默认为关闭。可以选择 4 个 V BOR 阈值。

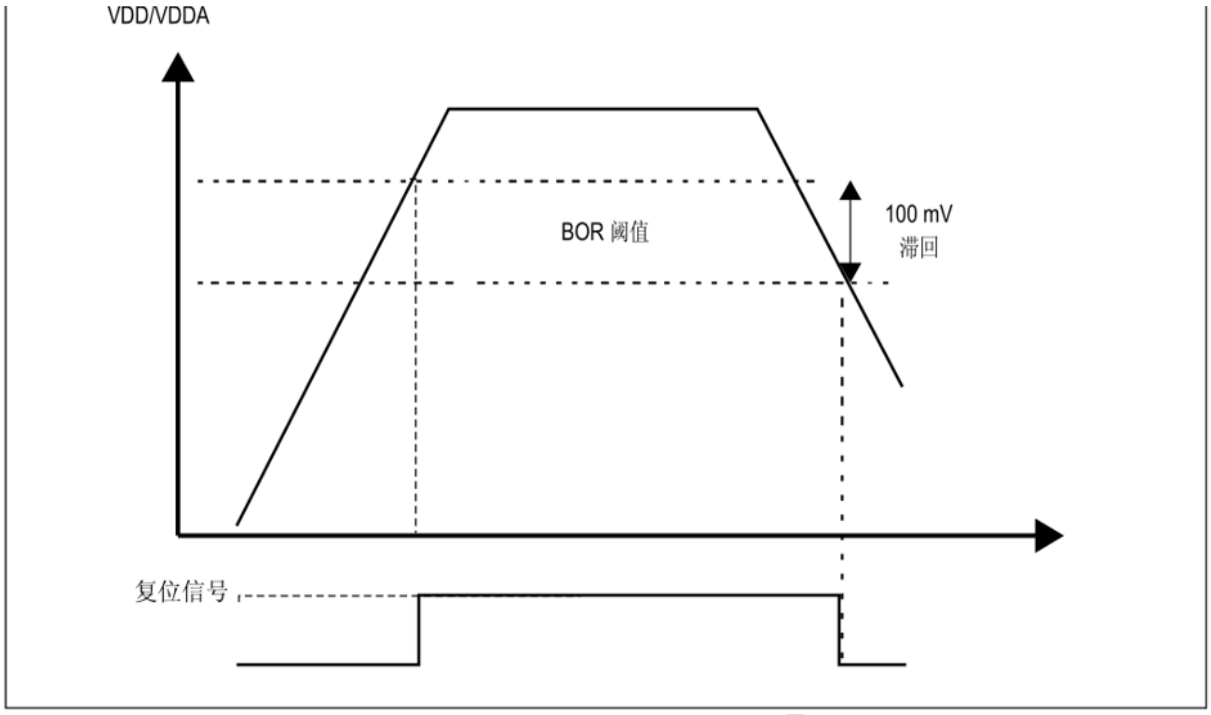
- BOR 关闭 (V BOR0)：1.80 V 到 2.10 V 电压范围的复位阈值级别
- BOR 级别 1 (V BOR1)：2.10 V 到 2.40 V 电压范围的复位阈值级别
- BOR 级别 2 (V BOR2)：2.40 V 到 2.70 V 电压范围的复位阈值级别
- BOR 级别 3 (V BOR3)：2.70 V 到 3.60 V 电压范围的复位阈值级别

当电源电压 (V DD) 降至所选 V BOR 阈值以下时，将使器件复位。

通过对器件选项字节进行编程可以禁止 BOR。要禁止 BOR 功能，V DD 必须高于 V BOR0 ，以启动器件选项字节编程序列。那么就只能由 PDR 监测掉电过程（参见 第 7.2.1 节：上电复位(POR)/ 掉电复位 (PDR)）

BOR 阈值滞回电压约为 100 mV（电源电压的上升沿与下降沿之间）。

BOR 阈值如下图



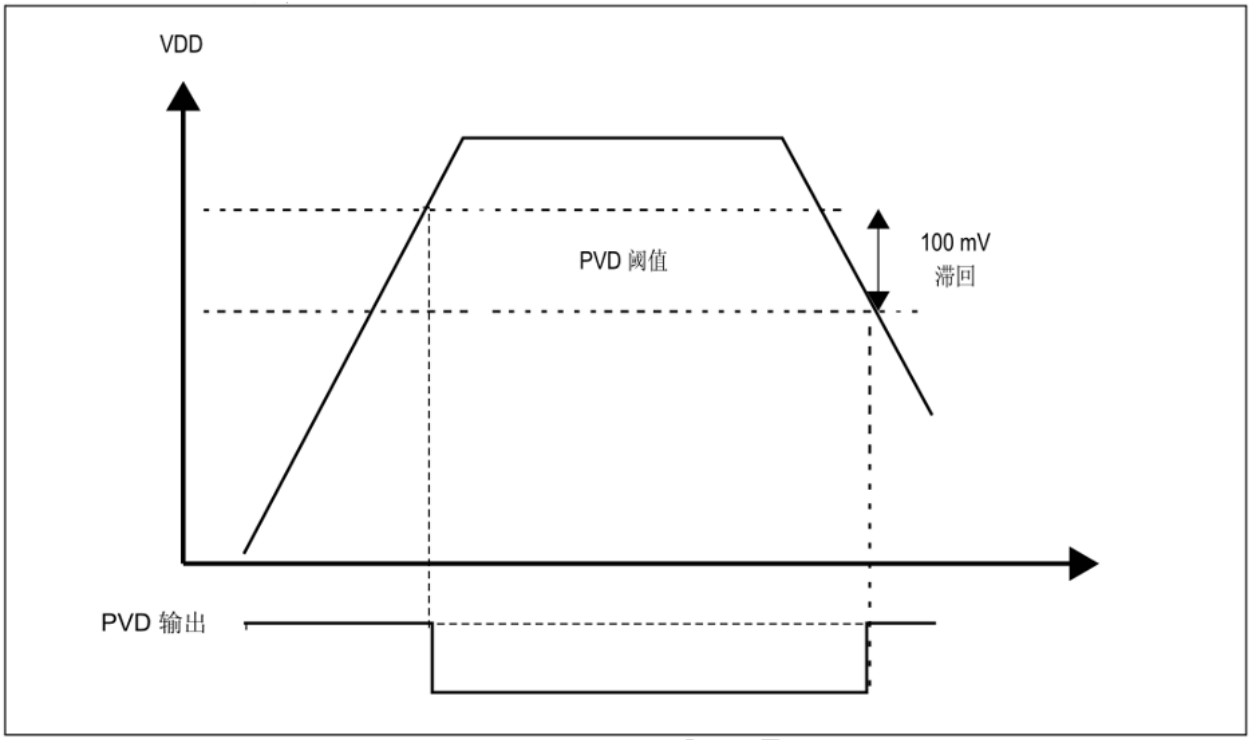
8.2.3 可编程电压检测器 (PVD)

可以使用 PVD 监视 V DD 电源，将其与PWR 电源控制寄存器 (PWR_CR) 中 PLS[2:0] 位所选的阈值进行比较。通过设置 PVDE 位来使能 PVD。

PWR 电源控制 / 状态寄存器 (PWR_CSR) 中提供了 PVDO 标志，用于指示 V DD 是大于还是小于 PVD 阈值。该事件内部连接到 EXTI 线 16，如果通过 EXTI 寄存器使能，则可以产生中断。当 V DD 降至 PVD 阈值以下以及/或者当 V DD 升至 PVD 阈值以上时，可以产生 PVD 输出中断，具体取决于 EXTI 线 16 上升沿/下降沿的配置。该功能的用处之一就是可以在中断

服务程序中执行紧急关闭系统的任务。

PVD 阈值如下图



8.2.4 电源去耦电容的选择

每个电源对 (VDD/VSS， VDDA/VSSA ...) 必须使用下述的滤波陶瓷电容去耦。这些电容必须尽量靠近芯片引脚，以确保器件正常工作。不建议去掉滤波电容来降低 PCB 尺寸或成本，这可能导致器件工 作不正常。

8.3 低功耗模式

默认情况下，系统复位或上电复位后，微控制器进入运行模式。在运行模式下，CPU 通过HCLK 提供时钟，并执行程序代码。系统提供了多个低功耗模式，可在 CPU 不需要运行时（例如等待外部事件时）节省功耗。由用户根据应用选择具体的低功耗模式，以在低功耗、短启动时间和可用唤醒源之间寻求最佳平衡。

器件有三个低功耗模式：

- 睡眠模式（Cortex™-MSTAR 内核停止，外设保持运行）
- 停止模式（所有时钟都停止）
- 待机模式（1.2 V 域断电）

此外，可通过下列方法之一降低运行模式的功耗：

- 降低系统时钟速度
- 不使用 APBx 和 AHBx 外设时，将对应的外设时钟关闭

8.3.1 降低系统时钟速度

在运行模式下，可通过对预分频寄存器编程来降低系统时钟（SYSCLK、HCLK、PCLK1 和PCLK2）速度。进入睡眠模式之前，也可以使用这些预分频器降低外设速度。

8.3.2 外设时钟门控

在运行模式下，可随时停止各外设和存储器的 HCLKx 和 PCLKx 以降低功耗。要进一步降低睡眠模式的功耗，可在执行 WFI 或 WFE 指令之前禁止外设时钟。
外设时钟门控由 AHB1 外设时钟使能寄存器 (RCC_AHB1ENR)、AHB2 外设时钟使能寄存器 (RCC_AHB2ENR)。

在睡眠模式下，复位 RCC_AHBxLPENR 和 RCC_APBxLPENR 寄存器中的对应位可以自动禁止外设时钟。

8.3.3 睡眠模式

进入睡眠模式

执行 WFI（等待中断）或 WFE（等待事件）指令即可进入睡眠模式。根据 Cortex™-MSTAR 系统控制寄存器中 SLEEPONEXIT 位的设置，可以通过两种方案选择睡眠模式进入机制：

- 立即休眠：如果 SLEEPONEXIT 位清零，MCU 将在执行 WFI 或 WFE 指令时立即进入睡眠模式。
- 退出时休眠：如果 SLEEPONEXIT 位置 1，MCU 将在退出优先级最低的 ISR 时立即进入睡眠模式。

退出睡眠模式

如果使用 WFI 指令进入睡眠模式，则嵌套向量中断控制器 (NVIC) 确认的任意外设中断都会将器件从睡眠模式唤醒。
如果使用 WFE 指令进入睡眠模式，MCU 将在有事件发生时立即退出睡眠模式。唤醒事件可通过以下方式产生：

- 在外设的控制寄存器使能一个中断，但不在 NVIC 中使能，同时使能 Cortex™-MSTAR 系统控制寄存器中的 SEVONPEND 位。当 MCU 从 WFE 恢复时，需要清除相应外设的中断挂起位和外设 NVIC 中断通道挂起位（在 NVIC 中断清除挂起寄存器中）。
- 配置一个外部或内部 EXTI 线为事件模式。当 CPU 从 WFE 恢复时，因为对应事件线的挂起位没有被置位，不必清除相应外设的中断挂起位或 NVIC 中断通道挂起位。

由于没有在进入/退出中断时浪费时间，此模式下的唤醒时间最短。
有关如何退出睡眠模式的详细信息，请参见 表 19 和 表 20。

表 19. 进入和退出立即休眠

立即休眠模式	说明
进入模式	WFI（等待中断）或 WFE（等待事件），且： — SLEEPDEEP = 0 及 — SLEEPONEXIT = 0 请参见 Cortex™-MSTAR 系统控制寄存器。
退出模式	如果使用 WFI 进入：中断

	如果使用 WFE 进入：唤醒事件
唤醒延迟	无

表 20. 进入和退出时休眠

退出时休眠	说明
进入模式	WFI（等待中断），且： — SLEEPDEEP = 0 及 — SLEEPONEXIT = 1 请参见 Cortex™-MSTAR 系统控制寄存器。
退出模式	中断
唤醒延迟	无

8.3.4 停止模式

停止模式基于 Cortex™-MSTAR 深度睡眠模式与外设时钟门控。调压器既可以配置为正常模式，也可以配置为低功耗模式。在停止模式下，1.2 V 域中的所有时钟都会停止，PLL、HSI和 HSE RC 振荡器也被禁止。内部 SRAM 和寄存器内容将保留。将 PWR_CR 寄存器中的 FPDS 位置 1 后，Flash 还会在器件进入停止模式时进入掉电状态。Flash 处于掉电模式时，将器件从停止模式唤醒将需要额外的启动延时（参见 表 21 ：停止工作模式 ）。

表 21. 停止工作模式

停止模式	**LPDS** 位	**FPDS** 位	唤醒延迟
STOP MR（主调压器）	0	0	HSI RC启动时间
STOP MR-FPD	0	1	HSI RC启动时间+Flash从掉电模式唤醒的时间
STOP LP	1	0	HSI RC启动时间+ 调压器从LP模式唤醒的时间
STOP LP-FPD	1	1	HSI RC启动时间+ Flash从掉电模式唤醒的时间+ 调压器从LP模式唤醒的时间

进入停止模式

如果正在执行 Flash 编程，停止模式的进入将延迟到存储器访问结束后执行。
如果正在访问 APB 域，停止模式的进入则延迟到 APB 访问结束后执行。

在停止模式下，可以通过对各控制位进行编程来选择以下功能：

- 独立的看门狗 (IWDG)：IWDG 通过写入其密钥寄存器或使用硬件选项来启动。而且一旦启动便无法停止，除非复位。独立看门狗 (IWDG) 中的 3 节 。
- 内部 RC 振荡器 (LSI RC)：通过 RCC 时钟控制和状态寄存器 (RCC_CSR) 中的 LSION位进行配置。
- 外部 32kHz 振荡器 (LSE OSC)：通过 RCC 备份域控制寄存器 (RCC_BDCR) 中的LSEON位进行配置。

在停止模式下，ADC 也会产生功耗，除非在进入停止模式前将其禁止。要禁止这些转换器，必须将 ADC_CR2 寄存器中的 ADON 位ENx 位都清零。

退出停止模式

有关如何退出停止模式的详细信息，请参见 表 22。

通过发出中断或唤醒事件退出停止模式时，将选择 HSI RC 振荡器作为系统时钟。
当调压器在低功耗模式下工作时，将器件从停止模式唤醒将需要额外的延时。在停止模式下一直开启内部调压器虽然可以缩短启动时间，但功耗却增大。

表 22. 进入和退出停止模式

停止模式	说明
进入模式	WFI（等待中断）或 WFE（等待事件），且： — 将 Cortex™-MSTAR 系统控制寄存器中的 SLEEPDEEP 位置 1 — 将电源控制寄存器 (PWR_CR) 中的 PDDS 位清零 — 通过配置 PWR_CR 中的 LPDS 位选择调压器模式。 注意：要进入停止模式，所有 EXTI 线挂起位（在 挂起寄存器 (EXTI_PR) 中）。否则将忽略进入停止模式这一过程，继续执行程序。
退出模式	如果使用 WFI 进入： 所有配置为中断模式的 EXTI 线（必须在 NVIC 中使能对应的 EXTI 中断向量）。如果使用 WFE 进入： 所有配置为事件模式的 EXTI 线。
唤醒延迟	表 21 ：停止工作模式

8.3.4 待机模式

待机模式下可达到最低功耗。待机模式基于 Cortex™-MSTAR 深度睡眠模式，其中调压器被禁止。因此 1.2 V 域断电。PLL、HSI 振荡器和 HSE 振荡器也将关闭。

进入待机模式

有关如何进入待机模式的详细信息，请参见 表 23。
在待机模式下，可以通过对各控制位进行编程来选择以下功能：

- 独立的看门狗 (IWDG)：IWDG 通过写入其密钥寄存器或使用硬件选项来启动。而且一旦启动便无法停止，除非复位。请参见 第 19 节：独立看门狗 (IWDG) 中的 第 19.3 节 。
- 内部 RC 振荡器 (LSI RC)：通过控制/状态寄存器 (RCC_CSR) 中的 LSION 位进行配置。
- 外部 32kHz 振荡器 (LSE OSC)：通过备份域控制寄存器 (RCC_BDCR) 中的 LSEON位进行配置。

退出待机模式

检测到外部复位（NRST 引脚）、IWDG 复位、WKUP 引脚上升沿、入侵事件或时间戳时间时，微控制器退出待机模式。从待机模式唤醒后，除 PWR 电源控制 / 状态寄存器 (PWR_CSR) 外，所有寄存器都将复位。
从待机模式唤醒后，程序将按照复位（启动引脚采样、复位向量已获取等）后的方式重新执行。PWR 电源控制 / 状态寄存器 (PWR_CSR) 中的 SBF 状态标志指示 MCU 已处于待机模式。
有关如何退出待机模式的详细信息，请参见 表 23。

表 23. 进入和退出待机模式

待机模式	说明

进入模式	WFI （等待中断）或WFE （等待事件），且： -将Cortex™-M4F系统控制寄存器中的SLEEPDEEP位置1 -将电源控制寄存器（PWR_CR）中的PDDS位置1 -将电源控制/状态寄存器（PWR_CSR）中的WUF位清零 -将与所选唤醒源（RTC闹钟A、RTC闹钟B、RTC唤醒、RTC入侵或 RTC时间戳标志）对应的RTC标志清零
退出模式	WKUP引脚上升沿、RTC闹钟（闹钟A和闹钟B）、RTC唤醒事件、RTC 入侵事件、RTC时间戳事件、NRST引脚外部复位和IWDG复位。
唤醒延退	复位阶段。

待机模式下的 I/O 状态

在待机模式下，除以下各部分以外，所有 I/O 引脚都处于高阻态.

调试模式

默认情况下，如果使用调试功能时应用程序将 MCU 置于停止模式或待机模式，调试连接将中断。这是因为 Cortex™-MSTAR 内核时钟停止了。
不过，通过设置 DBGMCU_CR 寄存器中的一些配置位，即使 MCU 进入低功耗模式，仍可使用软件对其进行调试。

8.4总结

本章节就为大家讲解这么多，本章节的知识点相对比较多，比较杂，不容易一下子都掌握了。随着后面章节的进行，还会深入的讲解这些知识点。

STC MCU

第9章 复位和RCC 时钟控制 (待确定)

9.1 初学者重要提示

9.2 复位

共有三种类型的复位，分别为系统复位、电源复位和备份域复位。

9.2.1 系统复位

除了时钟控制寄存器 CSR 中的复位标志和备份域中的寄存器外，系统复位会将其它全部寄存器都复位为复位值（请参见 图 4）。
只要发生以下事件之一，就会产生系统复位：
1. NRST 引脚低电平（外部复位）
2. 窗口看门狗计数结束（WWDG 复位）
3. 独立看门狗计数结束（IWDG 复位）

4. 软件复位（SW 复位）（请参见 软件复位 ）
5. 低功耗管理复位（请参见 低功耗管理复位 ）

软件复位

可通过查看 RCC 时钟控制和状态寄存器 (RCC_CSR) 中的复位标志确定。
要对器件进行软件复位，必须将 Cortex™-MSTAR 应用中断和复位控制寄存器中的 SYSRESETREQ 位置 1。有关详细信息，请参见 Cortex™-MSTAR 技术参考手册。

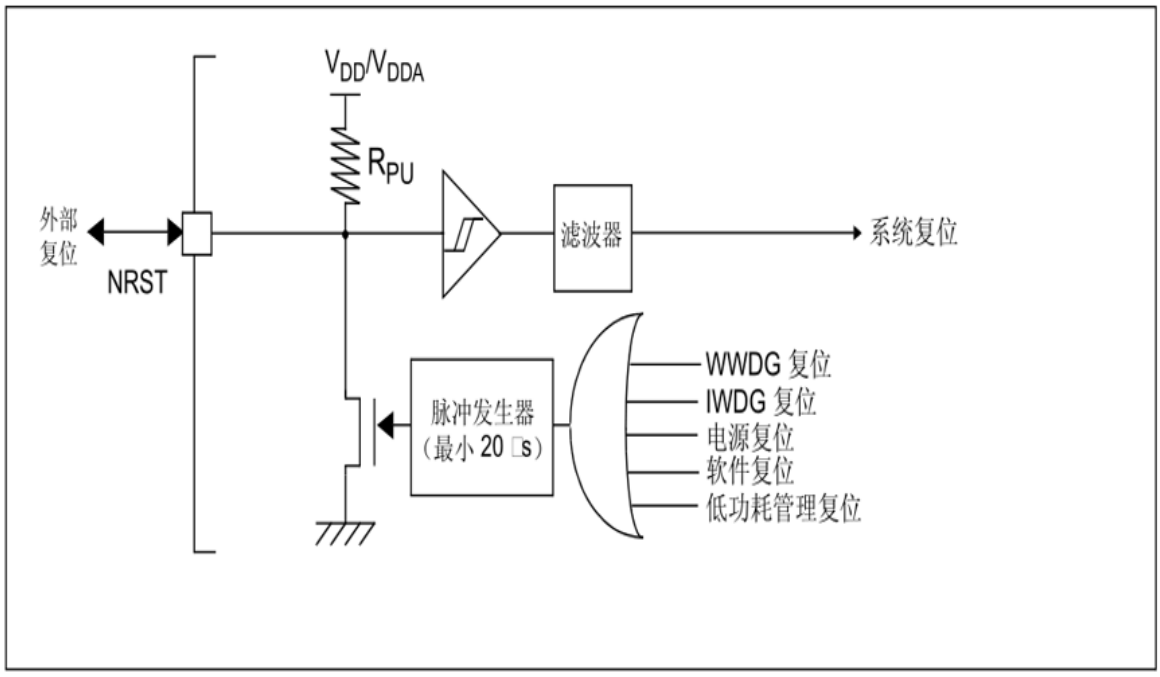
低功耗管理复位

引发低功耗管理复位的方式有两种：
1. 进入待机模式时产生复位：
此复位的使能方式是清零用户选项字节中的 nRST_STDBY 位。使能后，只要成功执行进入待机模式序列，器件就将复位，而非进入待机模式。
2. 进入停止模式时产生复位：
此复位的使能方式是清零用户选项字节中的 nRST_STOP 位。使能后，只要成功执行进入停止模式序列，器件就将复位，而非进入停止模式。

9.2.2 电源复位

只要发生以下事件之一，就会产生电源复位：
1. 上电/掉电复位（POR/PDR 复位）或欠压 (BOR) 复位
2. 在退出待机模式时
除备份域内的寄存器以外，电源复位会将其它全部寄存器设置为复位值（请参见 图 4）这些源均作用于 NRST 引脚，该引脚在复位过程中始终保持低电平。RESET 复位入口向量在存储器映射中固定在地址 0x0000_0004 。
芯片内部的复位信号会在 NRST 引脚上输出。脉冲发生器用于保证最短复位脉冲持续时间，可确保每个内部复位源的复位脉冲都至少持续 20 μs。对于外部复位，在 NRST 引脚处于低电平时产生复位脉冲。

图 12. 复位电路简图



备份域具有两个特定的复位，这两个复位仅作用于备份域本身（请参见 图 4）。

9.2.3 备份域复位

备份域复位会将RCC_BDCR 寄存器复位为各自的复位值。BKPSRAM 不受此复位影响。BKPSRAM 的唯一复位方式是通过 Flash 接口将 Flash 保护等级从 1 切换到 0。
只要发生以下事件之一，就会产生备份域复位：
1. 软件复位，通过将 RCC 备份域控制寄存器 (RCC_BDCR) 中的 BDRST 位置 1 触发。
2. 在电源 V DD 和 V BAT 都已掉电后，其中任何一个又再上电。

9.3 时钟

可以使用三种不同的时钟源来驱动系统时钟 (SYSCLK)：
- HSI 振荡器时钟
- HSE 振荡器时钟
- 主 PLL (PLL) 时钟

对于每个时钟源来说，在未使用时都可单独打开或者关闭，以降低功耗。

时钟控制器为应用带来了高度的灵活性，用户在运行内核和外设时可选择使用外部晶振或者使用振荡器，既可采用最高的频率，也可为USB以及 I2S等需要特定时钟的外设保证合适的频率。
可通过多个预分频器配置 AHB 频率、高速 APB (APB2) 和低速 APB (APB1)。AHB 域的最大频率为 72 MHz。高速 APB2 域的最大允许频率为 72 MHz。低速 APB1 域的最大允许频率为 72 MHz。
除以下时钟外，所有外设时钟均由系统时钟 (SYSCLK) 提供：
- 来自于特定 PLL 输出 (PLL48CLK) 的 USB时钟 (48 MHz).
- I2S 时钟

  要实现高品质的音频性能，可通过 I2S_CKIN 引脚的外部时钟提供 I2S 时钟。

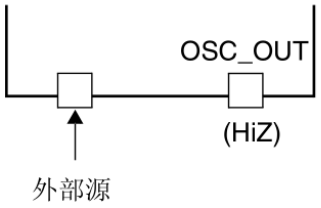
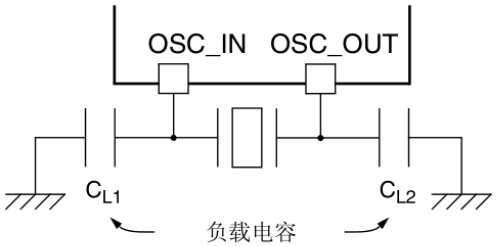
RCC 向 Cortex 系统定时器 (SysTick) 馈送 8 分频的 AHB 时钟 (HCLK)。SysTick 可使用此时钟作为时钟源，也可使用 HCLK 作为时钟源，具体可在 SysTick 控制和状态寄存器中配置。
定时器时钟频率由硬件自动设置。分为两种情况：
1. 如果 APB 预分频器为 1，定时器时钟频率等于 APB 域的频率。
2. 否则，等于 APB 域的频率的两倍 (×2)。

9.3.1 HSE 时钟

高速外部时钟信号 (HSE) 有 2 个时钟源：
- HSE 外部晶振/陶瓷谐振器
- HSE 外部用户时钟

谐振器和负载电容必须尽可能地靠近振荡器的引脚，以尽量减小输出失真和起振稳定时间。负载电容值必须根据所选振荡器的不同做适当调整。

图 14. HSE/LSE 时钟源

	硬件配置
外部时钟	
晶振/陶瓷谐振器	

在此模式下，必须提供外部时钟源。此模式通过将 RCC 时钟控制寄存器 (RCC_CR) 中的HSEBYP 和 HSEON 位置 1 进行选择。必须使用占空比约为 50% 的外部时钟信号（方波、正弦波或三角波）来驱动 OSC_IN 引脚，同时 OSC_OUT 引脚应保持为高阻态 (hi-Z)。请参见 图 14。
外部晶振/ 陶瓷谐振器（HSE 晶振）HSE 的特点是精度非常高。
相关的硬件配置如 图 14 所示。有关详细信息，请参见 数据手册 的电气特性部分。
RCC 时钟控制寄存器 (RCC_CR) 中的 HSERDY 标志指示高速外部振荡器是否稳定。在启动时，硬件将此位置 1 后，此时钟才可以使用。如在 RCC 时钟中断寄存器 (RCC_CIR) 中使能中断，则可产生中断。
HSE 晶振可通过 RCC 时钟控制寄存器 (RCC_CR) 中的 HSEON 位打开或关闭。

9.3.2 HSI 时钟

HSI 时钟信号由内部 16 MHz RC 振荡器生成，可直接用作系统时钟，或者用作 PLL 输入。HSI RC 振荡器的优点是成本较低（无需使用外部组件）。此外，其启动速度也要比 HSE 晶振块，但即使校准后，其精度也不及外部晶振或陶瓷谐振器。
校准因为生产工艺不同，不同芯片的 RC 振荡器频率也不同，因此会对每个器件进行出厂校准，达到 TA = 25 ° C 时 1% 的精度。
复位后，工厂校准值将加载到 RCC 时钟控制寄存器 (RCC_CR) 的 HSICAL[7:0] 位中。
如果应用受到电压或温度变化影响，则这可能也会影响到 RC 振荡器的速度。用户可通过RCC 时钟控制寄存器 (RCC_CR) 中的 HSITRIM[4:0] 位对 HSI 频率进行微调。
RCC 时钟控制寄存器 (RCC_CR) 中的 HSIRDY 标志指示 HSI RC 是否稳定。在启动时，硬件将此位置 1 后，HSI 才可以使用。
HSI RC 可通过 RCC 时钟控制寄存器 (RCC_CR) 中的 HSION 位打开或关闭。
HSI 信号还可作为备份时钟源（辅助时钟）使用，以防 HSE 晶振发生故障。

9.3.3 LSE 时钟

LSE 晶振是 32 kHz 低速外部 (LSE) 晶振或陶瓷谐振器。
LSE 晶振通过 RCC 备份域控制寄存器 (RCC_BDCR) 中的 LSEON 位打开和关闭。
RCC 备份域控制寄存器 (RCC_BDCR) 中的 LSERDY 标志指示 LSE 晶振是否稳定。在启动时，硬件将此位置 1 后，LSE 晶振输出时钟信号才可以使用。如在 RCC 时钟中断寄存器(RCC_CIR) 中使能中断，则可产生中断。

外部源（LSE 旁路）在此模式下，必须提供外部时钟源，最高频率不超过 1 MHz。此模式通过将 RCC 备份域控制寄存器 (RCC_BDCR) 中的 LSEBYP 和 LSEON 位置 1 进行选择。必须使用占空比约为 50%的外部时钟信号（方波、正弦波或三角波）来驱动 OSC32_IN 引脚，同时 OSC32_OUT 引脚应保持为高阻态 (Hi-Z)。请参见图 14。

9.3.4 LSI 时钟

LSI RC 可作为低功耗时钟源在停机和待机模式下保持运行，供独立看门狗 (IWDG) 和自动唤醒单元 (AWU) 使用。时钟频率在 32 kHz 左右。
LSI RC 可通过 RCC 时钟控制和状态寄存器 (RCC_CSR) 中的 LSION 位打开或关闭。
RCC 时钟控制和状态寄存器 (RCC_CSR) 中的 LSIRDY 标志指示低速内部振荡器是否稳定。
在启动时，硬件将此位置 1 后，此时钟才可以使用。如在 RCC 时钟中断寄存器 (RCC_CIR) 中使能中断，则可产生中断。

9.3.5 系统时钟 (SYSCLK) 选择

在系统复位后，默认系统时钟为 HSI。在直接使用 HSI时钟源来作为系统时钟时，该时钟源无法停止。只有在目标时钟源已就绪时，才可从一个时钟源切换到另一个。如果选择尚未就绪的时钟源，则切换在该时钟源就绪时才会进行。RCC 时钟控制寄存器 (RCC_CR) 中的状态位指示哪个（些）时钟已就绪，以及当前哪个时钟正充当系统时钟。

9.3.6 看门狗时钟

如果独立看门狗 (IWDG) 已通过硬件选项字节或软件设置的方式启动，则 LSI 振荡器将强制打开且不可禁止。在 LSI 振荡器稳定后，时钟将提供给 IWDG。

9.3.7 时钟输出功能

共有一个微控制器时钟输出 (MCLK) 引脚：
● MCLK
用户可通过可配置的预分配器（从 1 到 5）向 MCLK引脚 输出四个不同的时钟源：
— HSI 时钟
— LSE 时钟
— HSE 时钟
所需的时钟源通过 RCC 时钟配置寄存器 (RCC_CFGR) 中的 MCO1PRE[2:0] 和 MCO1[1:0]位选择。

9.3.8 基于 PWM5/PWM11 的内部/ 外部时钟测量

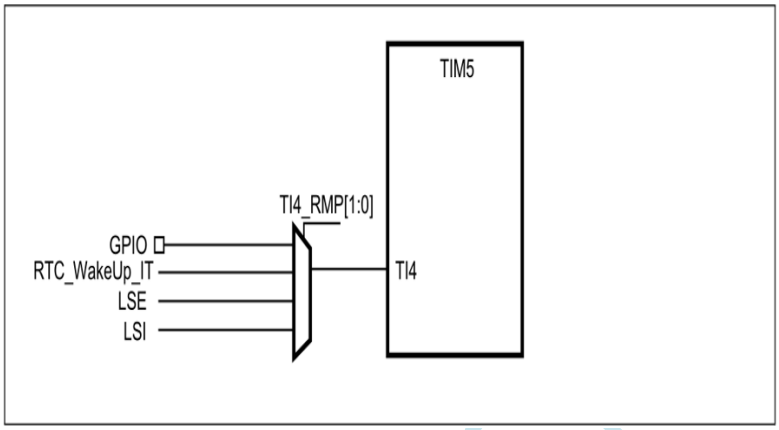
所有时钟源的频率都可通过对 PWM5 channel4 和 PWM11 channel1 的输入捕获进行间接测量，如图 15 所示。

基于 PWM5 channel4 的内部/ 外部时钟测量

PWM5 具有一个输入复用器，可选择输入捕获是由 I/O 触发还是由内部时钟触发。此选择通过 PWM5_OR 寄存器的 TI4_RMP [1:0] 位执行。
将 LSE 连接到 channel4 输入捕获的主要目的是为了精确测量 HSI（这需要将 HSI 用作系统时钟源）。借助 LSE 信号连续边沿之间的 HSI 时钟计数数量，即可对内部时钟周期进行测量。利用 LSE 的

高精度（通常为几十 ppm），用户能以同一分辨率测定时钟频率，并可通过对时钟源进行微调来补偿由生产工艺造成的和/或与温度和电压相关的频率偏差。
HSI 振荡器设有针对此目的的专用校准位，且支持用户访问。
其基本原理是基于相对的测量（例如，HSI/LSE 比）：因此，精度与两个时钟源之比紧密相关。比率越大，测量效果越好。
同时也可测量 LSI 频率：这对于没有晶振的应用场合非常实用。超低功耗 LSI 振荡器具有较大的生产工艺偏差：通过测量该振荡器与 HSI 时钟源的比率，可以借助 HSI 精度为其测定频率。
LSI 频率通过以下步骤测量：
1. 使能 PWM5 定时器并将 channel4 配置为输入捕获模式。
2. 将 PWM5_OR 寄存器中的 TI4_RMP 位设置为 0x01，以在内部将 LSI 时钟连接到 PWM5 channel4 输入捕获来实现校准。
3. 通过 PWM5 捕获/比较 4 事件或中断测量 LSI 时钟频率。
4. 使用测得的 LSI 频率来按照所需的时基，并且/或者用其来计算IWDG 超时。

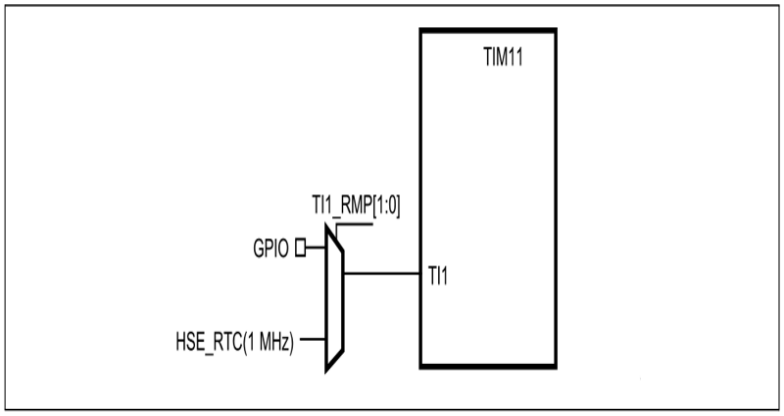
图 15. PWM5 在输入捕获模式下的频率测量



基于 PWM11 channel1 的内部/外部时钟测量

PWM11 具有一个输入复用器，可选择输入捕获是由 I/O 触发还是由内部时钟触发。此选择通过 PWM11_OR 寄存器的 TI1_RMP [1:0] 位执行。HSE_RTC 时钟（由一个可编程预分频器分频的 HSE）连接到通道 1 输入捕获，以粗略指示外部晶振频率。这要求 HSI 为系统时钟源。此功能非常实用，例如可借此测定谐波频率或分谐波频率（—50/+100% 偏差），进而确保符合 IEC 60730/IEC 61335 标准。

图 16. PWM11 在输入捕获模式下的频率测量



9.4总结

本章节就为大家讲解这么多，本章节的知识点相对比较多，比较杂，不容易一下子都掌握了。随着后面章节的进行，还会深入的讲解这些知识点。

STC MCU