

---

ST (意法半导体) 8 位微控制器系列

# STM8 系列单片机入门

王志杰



2011-06-15

---

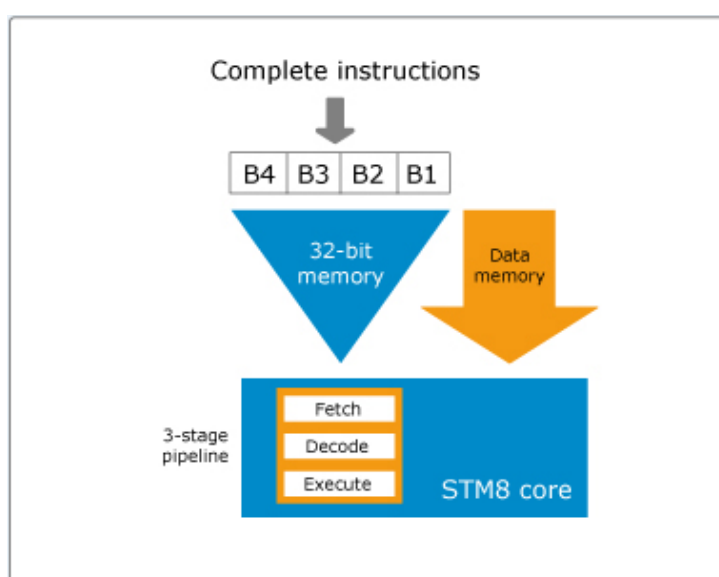
# 目 录

1 STM8 系列微控制器简介.....	3
1.1 STM8S 系列.....	3
1.2 STM8L 系列.....	5
1.3 STM8A 系列.....	7
1.4 STM8 系列微控制器网站.....	9
2 STM8 集成开发环境简介.....	10
2.1 ST TOOLSET.....	10
2.2 COSMIC.....	11
2.3 IAR.....	17
3 STM8 创建项目.....	18
3.1 使用 STVD 创建汇编项目.....	18
3.2 使用 COSMIC 创建 C 语言项目.....	39
3.3 使用 IAR 创建 C 语言项目.....	56
4 STM8 程序设计.....	71
4.1 STM8S 应用例程.....	71
5 STM8 开发工具.....	83
5.1 STX-RLINK.....	83
5.2 ST-LINK.....	84
5.3 ST-LINK/V2.....	85
6 STM8 EMC 设计注意事项.....	87

# 1 STM8 系列微控制器简介

ST 的 8 位微控制器平台基于高性能 8 位内核，配有先进的成套外设。该平台采用 ST 的 130 nm 嵌入式非易性存储器专有技术。STM8 通过增强型堆栈指针运算、先进的寻址模式和新指令实现快速、安全的开发。STM8 平台支持三种主要产品线：

- STM8S，通用微控制器
- STM8L，超低功耗 EnergyLite™ 微控制器
- STM8A，汽车级微控制器



## 1.1 STM8S 系列

2009 年 3 月 4 日，意法半导体发布了针对工业应用和消费电子开发的微控制器 STM8S 系列产品。

STM8S 平台打造 8 位微控制器的全新世代，高达 20 MIPS 的 CPU 性能和 2.95-5.5V 的电压范围，有助于现有的 8 位系统向电压更低的电源过渡。新产品嵌入的 130nm 非易失性存储器是当前 8 位微控制器中最先进的存储技术之一，并提供真正的 EEPROM 数据写入操作，可达 30 万次擦写极限。在家用电器、加热通风空调系统、工业自动化、电动工具、个人护理设备和电源控制管理系统等各种产品设备中，新产品配备的丰富外设可支持精确控制和监视功能。功能包括 10 位模数转换器，最多有 16 条通道，转换用时小于 3 微秒；先进的 16 位控制定时器可用于马达控制、捕获/比较和 PWM 功能。其它外设包括一个 CAN2.0B 接口、两个 U(S)ART 接口、一个 I<sup>2</sup>C 端口、一个 SPI 端口。

STM8S 平台的外设定义与 STM32 系列 32 位微控制器相同。外设共用性有助于提高不同产品间的兼容性，让设计灵活有弹性。应用代码可移植到 STM32 平台上，获得更高的性能。除设计灵活外，STM8S 的组件和封装在引脚上完全兼容，让开发人员得到更大的自由空间，以便优化引脚数量和外设性能。引脚兼容还有益于平台化设计决策，产品平台化可节省上市时间，简化产品升级过程。

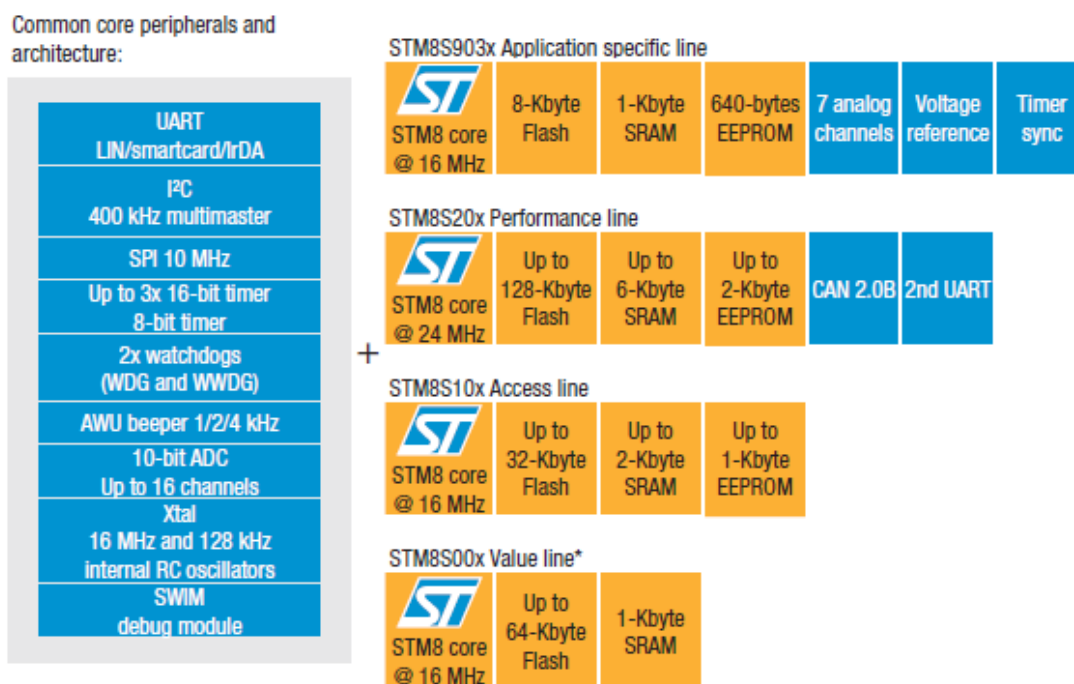
## STM8S 主要特点:

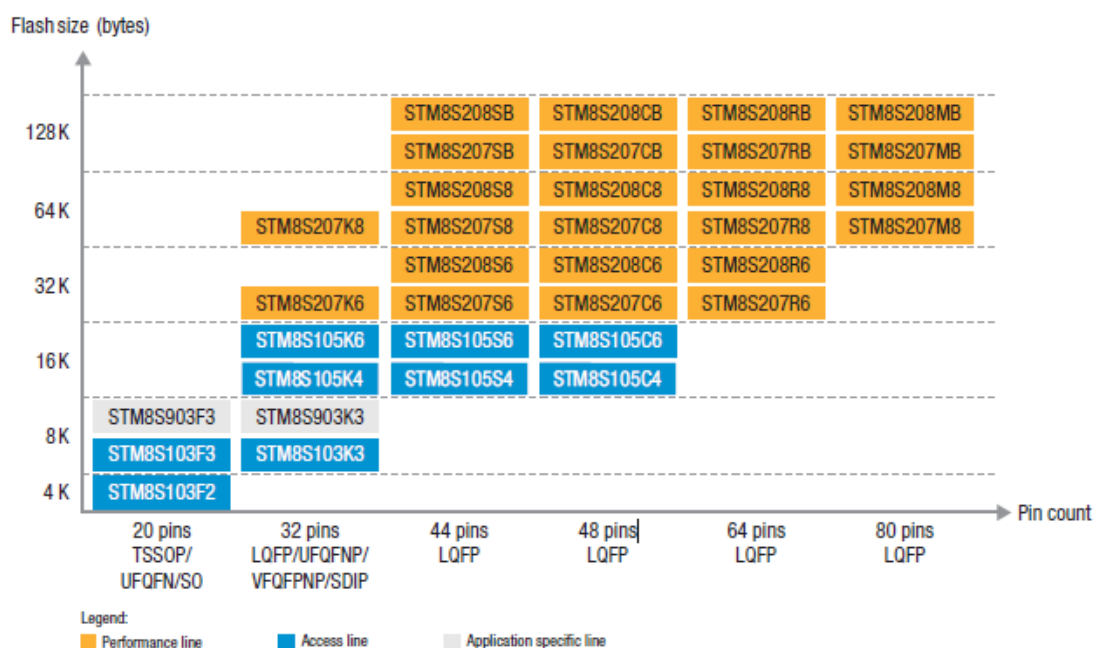
- 速度达 20 MIPS 的高性能内核
- 抗干扰能力强，品质安全可靠
- 领先的 130 纳米制造工艺，优异的性价比
- 程序空间从 4K 到 128K，芯片选择从 20 脚到 80 脚，宽范围产品系列
- 系统成本低，内嵌 EEPROM 和高精度 RC 振荡器
- 开发容易，拥有本地化工具支持

## STM8S 主要应用:

- 汽车电子：传感器、致动器、安全系统微控制器、DC 马达、车身控制、汽车收音机、LIN 节点、加热/通风空调
- 工业应用：家电、家庭自动化、马达控制、空调、感应、计量仪表、不间断电源、安全
- 消费电子：电源、小家电、音响、玩具、销售点终端机、前面板、电视、监视设备
- 医疗设备：个人护理产品、健身器材、便携护理设备、医院护理设备、血压测量、血糖测量、监控、紧急求助

## STM8S 产品线：





## 1.2 STM8L 系列

2009年9月15日，意法半导体宣布，首批整合其高性能8位架构和最近发布的超低功耗创新技术的8位微控制器开始量产。以节省运行和待机功耗为特色，STM8L系列下设三个产品线，共计26款产品，涵盖多种高性能和多功能应用。

设计工程师利用全新的STM8L系列可提高终端产品的性能和功能，同时还能满足以市场为导向的需求，例如，终端用户对节能环保产品的需求，便携设备、各种医疗设备、工业设备、电子计量设备、感应或安保设备对电池使用周期的要求。设计人员将选择STM8L这类超低功耗的微控制器，以符合低功耗产品设计标准，如“能源之星”、IEA的“1W节能计划”或欧盟的EuP法令。

这三条STM8L产品线都基于意法半导体的超低功耗技术平台，这个平台采用意法半导体独有的超低泄漏电流优化的130nm制程。独一无二的技术优势包括在1.65V到3.6V的整个电源电压范围内达到CPU最大工作频率，发挥CPU的全部性能。此外，由于采用一个片上稳压器，功耗与V<sub>dd</sub>电压无关，所以具有更高的设计灵活性，并有助于简化产品设计。

其它创新特性包括低功耗嵌入式非易失性存储器和多个电源管理模式，包括5.4μA低功耗运行模式、3.3μA低功耗待机模式、1μA主动停止模式（实时时钟运行）和350nA停止模式。STM8L可以在4μs内从停止模式唤醒，支持频繁使用最低功耗模式。低功耗外设，包括小于1μA的实时时钟和自动唤醒（AWU）模块，有助于进一步节省电能。总之，这个平台可将动态电流消耗降到150μA/MHz。

**STM8L主要特点:**

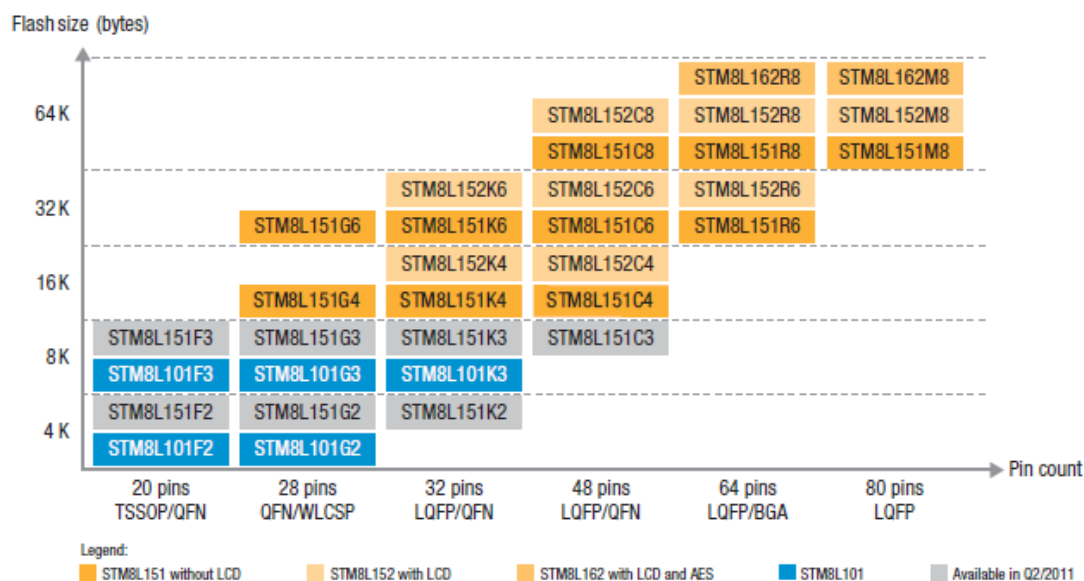
- STM8 16 MHz CPU
- 内置 4-32 KB 闪存，多达 2KB SRAM
- 三个系列:跨系列的引脚对引脚兼容、软件相互兼容、外设相互兼容
- 电源电压：1.8 V-3.6 V (断电时，最低 1.65 V)
- 超低功耗模式：保持 SRAM 内容时，最低功耗 350nA
- 运行模式动态功耗低至 150µA/MHz
- 最先进的数字和模拟外设接口
- 工作温度范围:-40°C 到+85 °C，可高达 125 °C
- 免费的触感固件库

**STM8L主要应用:**

- 便携医疗设备
- 玩具和游戏
- 公路收费
- 电池供电设备
- 个人保健产品
- 保安传感器

**STM8L 产品线：**

Common core peripherals and architecture: Communication peripherals USART, SPI, PC Multiple 16-bit timer Internal 16 MHz and 38 kHz RC oscillators Watchdog (dual watchdogs on STM8L15x/16x) Reset circuitry POR/PDR 2x comparators Touch-sensing (Up to 16 channels)	STM8L162  up to 64-Kbyte Flash Up to 4-Kbyte SRAM Reset + BOR PVD Main osc. input 1-16 MHz Up to 2-Kbyte data EEPROM RTC with 32 kHz osc. Up to 4 channels DMA 12-bit ADC (1 µs) Temp. sensor 12-bit DAC LCD 8 x 40 AES 128-bit
	STM8L152  up to 64-Kbyte Flash Up to 4-Kbyte SRAM Reset + BOR PVD Main osc. input 1-16 MHz Up to 2-Kbyte data EEPROM RTC with 32 kHz osc. Up to 4 channels DMA 12-bit ADC (1 µs) Temp. sensor 12-bit DAC LCD 8 x 40
	+ STM8L151  up to 64-Kbyte Flash Up to 4-Kbyte SRAM Reset + BOR PVD Main osc. input 1-16 MHz Up to 2-Kbyte data EEPROM RTC with 32 kHz osc. Up to 4 channels DMA 12-bit ADC (1 µs) Temp. sensor 12-bit DAC
	STM8L101  up to 8-Kbyte Flash* Up to 1.5-Kbyte SRAM



## 1.3 STM8A 系列

意法半导体公司推出的 STM8A 是一款专门用于满足汽车应用的特殊需求的 8-位 Flash 微控制器。这些模块化产品提供了真数据 EEPROM 以及软件和引脚兼容性，适用的程序存储器尺寸范围为 8KB 至 256KB 和 20 至 128-引脚封装。所有器件的工作电压均为 3V 至 5V，并且其工作温度扩展到了 145°C。

### STM8A 主要特点：

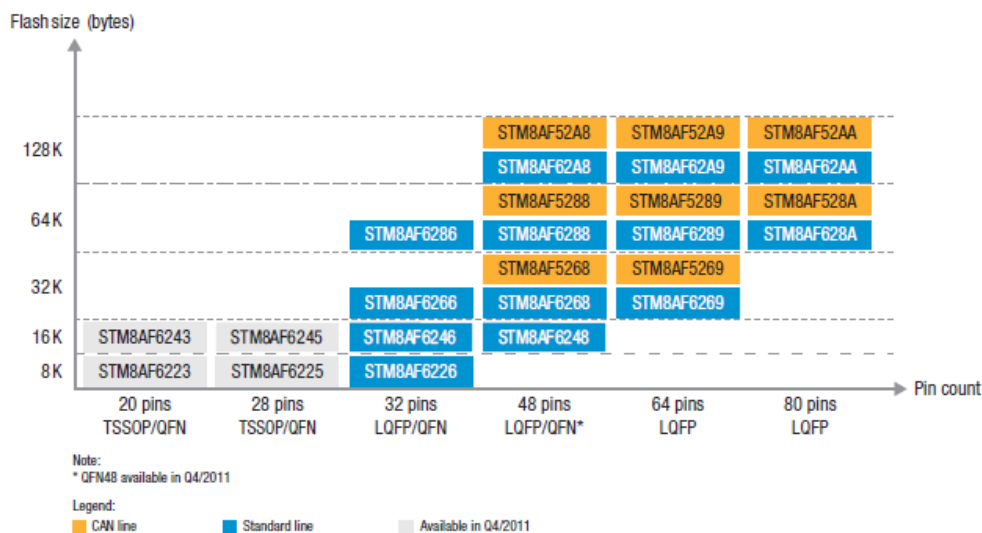
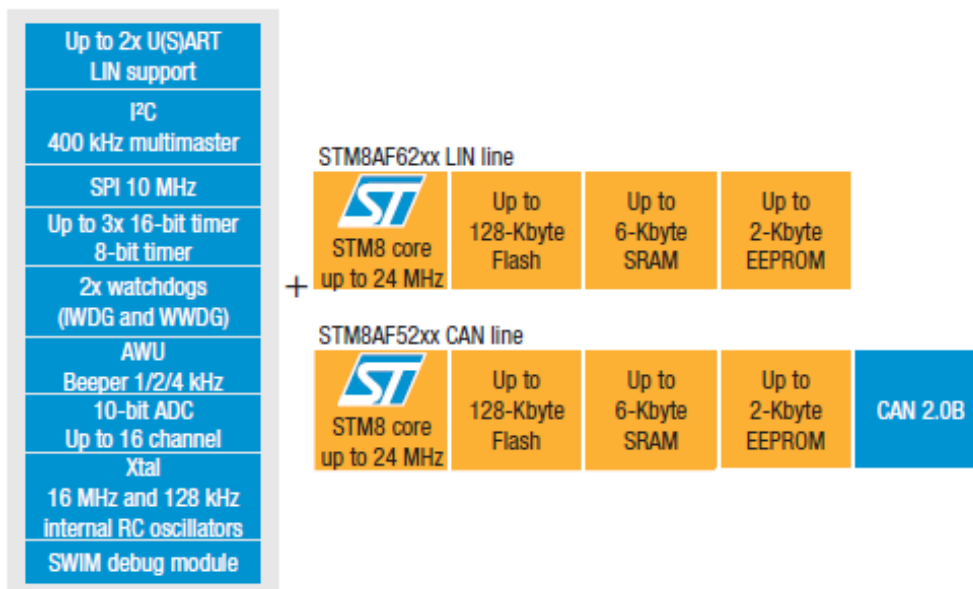
- 集成式真数据 EEPROM
- 16MHz 和 128kHz RC 振荡器
- 高效的 STM8 内核：在 16MHz 的频率下可以实现 10MIPS 的性能
- 应用安全性高：独立的看门狗定时器、时钟安全系统
- 所有产品均具有 LIN 2.0 和自同步功能
- 电源电压：3.3V 和 5V
- 最高工作温度:145 °C

### STM8A 主要应用：

- 激励器，车体控制器，传感器，DC 电机控制，安全微控制器，LIN 节点，汽车无线电，HVAC

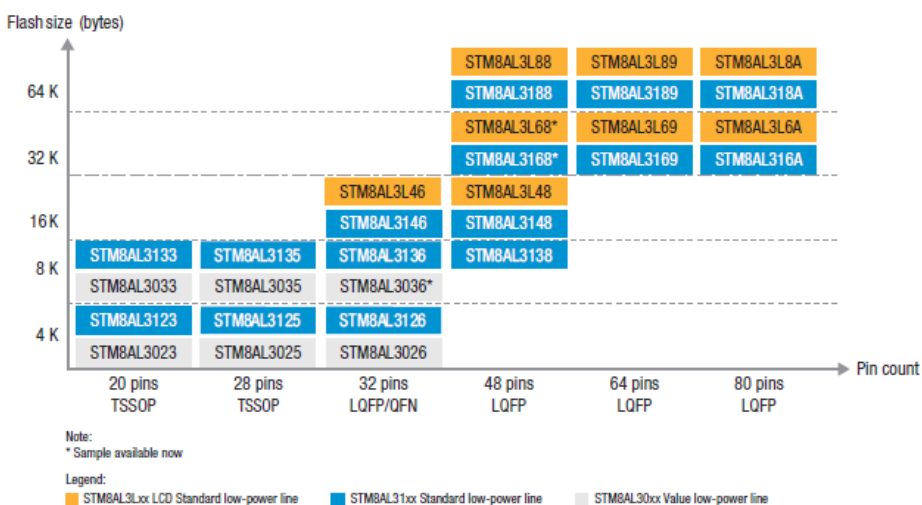
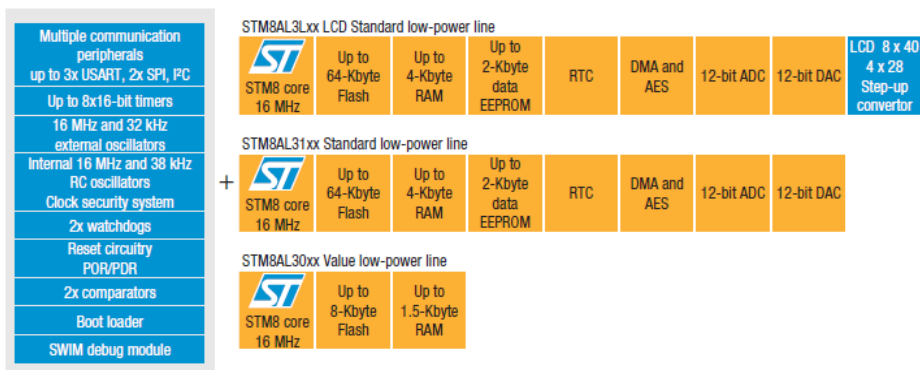
## STM8A 产品线：

Common core peripherals and architecture:





Common core peripherals and architecture:



## 1.4 STM8 系列微控制器网站

可登录ST网站<http://www.st.com> 寻找相关产品资料。或者直接登录MCU的页面<http://www.st.com/mcu> 寻找相关的产品资料。也可以通过以下地址直接进入相关的STM8 系列主要网址。

- STM8A汽车电子产品系列

<http://www.st.com/stm8a>

- STM8L超低功耗产品系列:

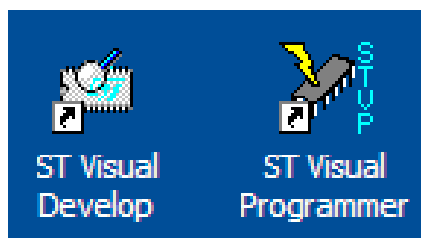
<http://www.st.com/stm8l>

- STM8S标准产品系列 :

<http://www.st.com/stm8s>



在下载安装完成后，一般会在桌面产生两个快捷方式：ST Visual Develop 和 ST Visual Programmer。



ST Visual Develop (STVD)：编辑调试烧录软件。

ST Visual Programmer (STVP)：烧录软件。

## 2.2 COSMIC

Cosmic Software 公司有着 20 多年的历史了。其产品也赢得了一些著名厂商的认可。Cosmic Software 公司(Cosmic Software Inc.)的 Cosmic C 编译器(Cosmic C compiler)及全套嵌入开发工具支持 STM8 系列产品的开发。Cosmic 产品包括 C 交叉编译器、汇编、连接器、ANSI 库、仿真器、硬件调试器和易于使用的集成开发环境 (IDEA)。



目前，Cosmic Software 公司提供了 16K 和 32K 代码大小限制的全功能的免费软件。此软件可从其公司网站上免费下载，网址是：<http://www.cosmicsoftware.com>。

在进入下载页面后，一般会出现一个注册页面，如下图所示。按照其要求填写，提交后，便可获得一个 C 编译器的下载连接。

[http://www.cosmicsoftware.com/download\\_stm8\\_32k.php](http://www.cosmicsoftware.com/download_stm8_32k.php)

Home / Download / stm8 FREE 32k

### Register for the FREE stm8 32k version

Fill and submit the form below to download the free stm8 compiler 32K version.  
To use this product you must register with Cosmic Software (this page) and then get a special license after installation.  
After your registration, you will be able to download the software; download it and then start the installation.  
The installation procedure will instruct you to send a message to Cosmic Software at [stm8\\_32k@cosmic.fr](mailto:stm8_32k@cosmic.fr) with your PC data; as a result you will receive the appropriate free license (limited to 1 year) for this product.

* Name	<input type="text"/>
* Company	<input type="text"/>
Address	<input type="text"/>
ZIP Code	<input type="text"/>
City	<input type="text"/>
* Country	-- Select --
Phone	<input type="text"/>
Fax	<input type="text"/>
* E-mail	<input type="text"/>
<input type="button" value="Submit"/> <input type="button" value="Clear"/>	

或者尝试直接从下面地址下载 32K 代码大小限制的版本：（不需要下载注册了）

[http://www.cosmicsoftware.com/cxstm8\\_32k.exe](http://www.cosmicsoftware.com/cxstm8_32k.exe)（32K 代码限制版本，免费）

[http://www.cosmicsoftware.com/cxstm8\\_16k.exe](http://www.cosmicsoftware.com/cxstm8_16k.exe)（16K 代码限制版本，免费）

### 注意：

#### ◇ License 文件：

- ✓ 若要使用免费的 C 编译器，还需要向 Cosmic Software 公司申请一个 license 文件。并且限制一台计算机对应一个 license 文件。

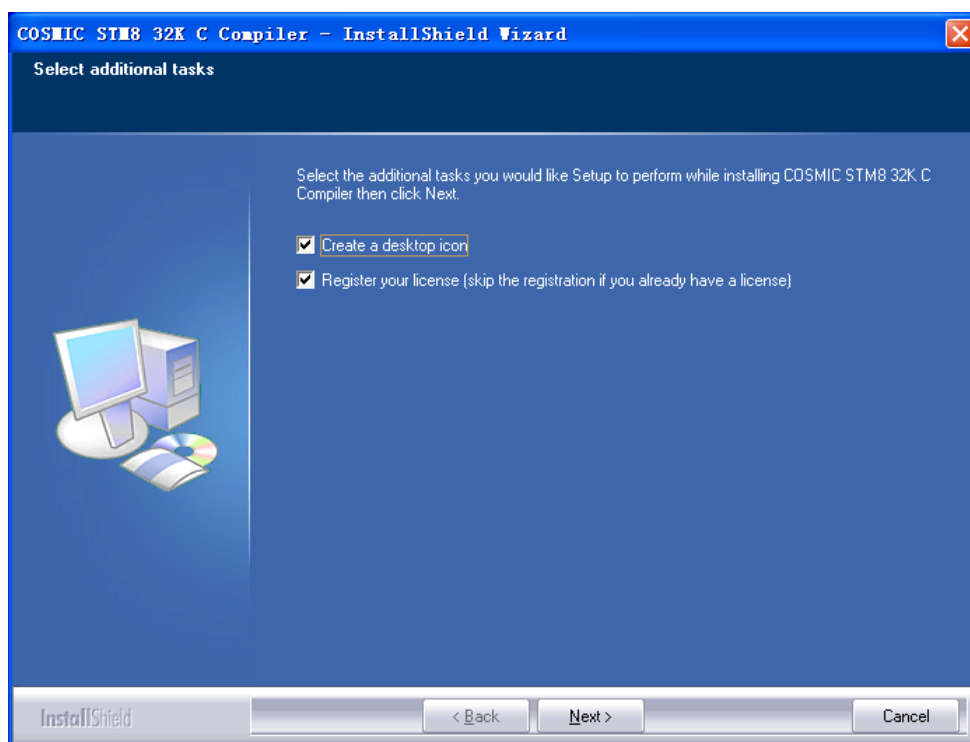
#### ◇ 获得 license 的 2 种方法：

- ✓ 第一种方法：在安装过程中，安装向导会引导注册，获得 license 文件
- ✓ 第二种方法：通过直接发送 EMAIL 的方式，获得 license 文件

下面详细介绍两种获得获得 license 文件的方法。

#### 第一种方法：通过安装向导获得 COSMIC 的 license 文件：

COSMIC C 编译器在安装过程中，出现下图的提示。提示创建桌面图标和申请软件使用协议。可以选择“Register your license”来申请 license 文件，选择“Next”。

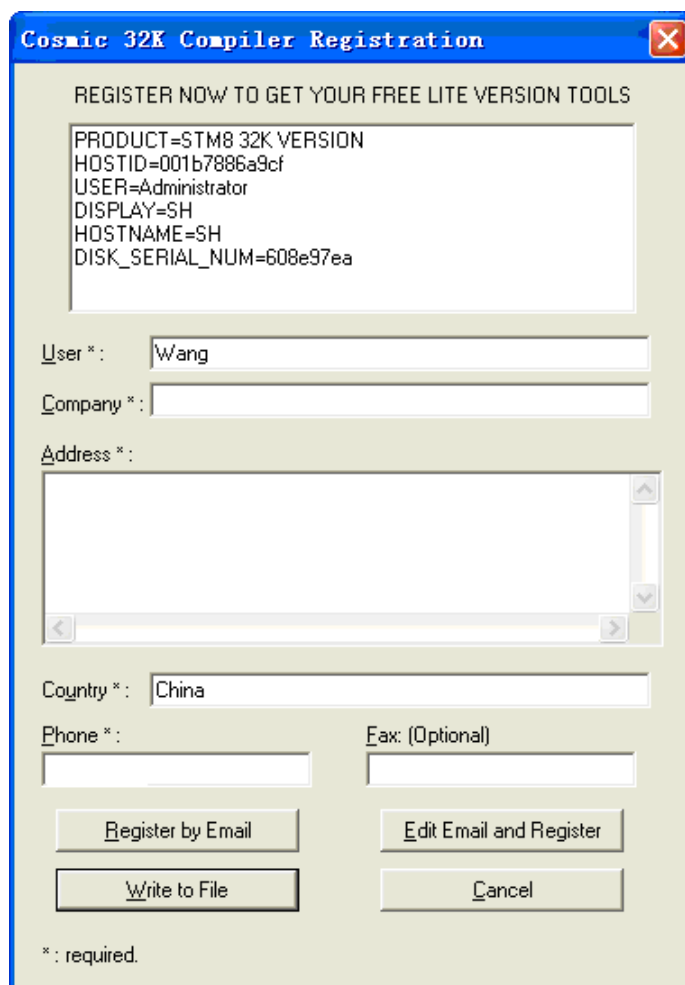


接着会弹出下面的对话框，意思是注册获得一个免费的版本，收到一个 license 文件后，就可使用该软件。继续！



根据要求填写下面的注册信息，要用英文填写完整。选择“ Register by Email”，此时会启动计算机系统默认的邮件软件，发送申请协议的邮件。注意，带“\*”号的一定要填写。在 Country 中一定要注明 China。建议用英文书写。

**特别注意是，申请时发送 email 的程序，是 WINDOWS 默认的程序。可以通过 Internet Explorer 的菜单“工具>Internet 选项”来查看或指定系统默认的邮件程序。**



如果填写准确完整无误后，则可发出。Cosmic Software 公司会回复一个 email，并带有一个 license.lic 文件。（Cosmic Software 公司回复时间不确定，有时差！！！若几天没有收到回复，则可能是注册信息不完整，Cosmic Software 无法回复）

将获得的 license.lic 文件拷贝到安装目录下的 license 目录下即可。一般地，默认目录为 C:\Program Files\COSMIC\CXSTM8\_32K\license。这样就可以使用 C 编译器了。

### 第二种方法：通过直接发送 EMAIL 获得 license 文件：

如果在安装过程中没有申请协议，也可以通过直接发送 EMAIL 的方式来获得 lincense.lic 文件。

在 COSMIC 的安装目录下（一般地，默认安装目录是 C:\Program Files\COSMIC\CXSTM8\_32K），找到 lmreg32k.exe 文件，运行获得本计算机的相关信息（Product information）。

打开邮件程序，编写一个 EMAIL,以获得一个 lincense.lic 文件。下面是一个例子：

收件人: **stm8\_32k@cosmic.fr**

标题: **STM8 32k License Request**

The information below should be sent to  
[stm8\\_32k@cosmic.fr](mailto:stm8_32k@cosmic.fr)  
to obtain a license for this product  
Registration Information for:

User: Wang  
Company: Company Name  
Address: PuDong South Road PuDong ,Shanghai 200120, P.R.China  
Country: China  
Phone: +86-021-00000000  
Fax: +86-021-00000000

Product Information:

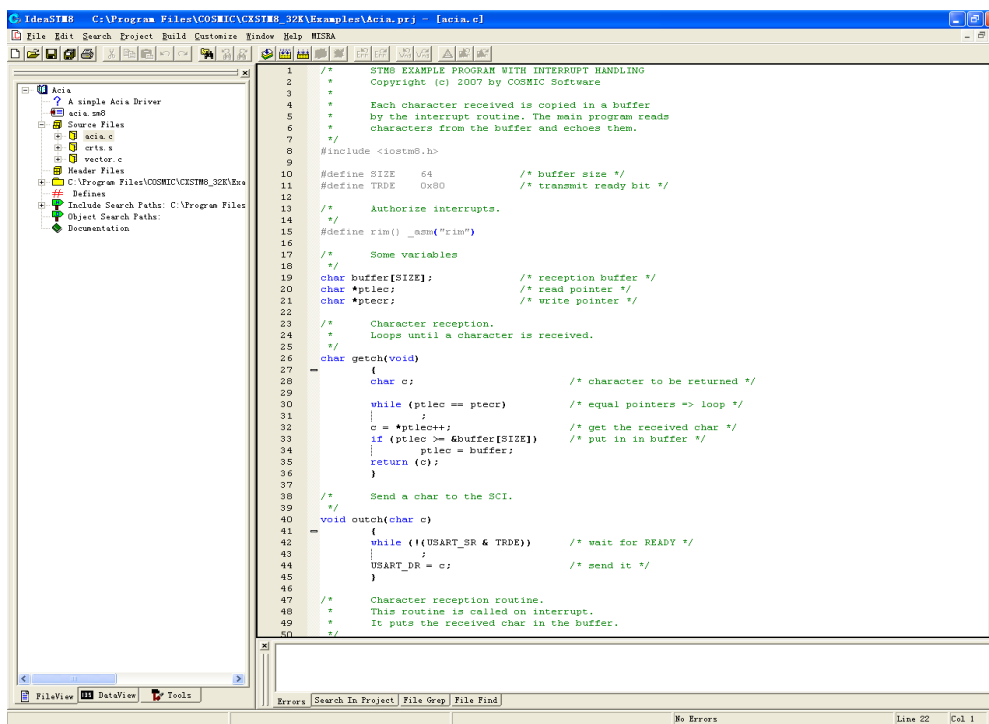
PRODUCT=STM8 32K VERSION  
HOSTID=001b7886a9cf  
USER=Administrator  
DISPLAY=SH  
HOSTNAME=SHPC  
DISK\_SERIAL\_NUM=608e97ea

### 说明：

- Registration Information for：需要填写完整准确无误
- Product Information:信息需要通过运行 lmreg32k.ext 来获得

将获得的 license.lic 文件拷贝到安装目录下的 license 目录下即可。一般地，默认目录为 C:\Program Files\COSMIC\CXSTM8\_32K\license。这样就可以使用 C 编译器了。

COSMIC 公司为用户提供了一个集成的开发环境 ( IDE ) , 其运行后界面如下图所示 :



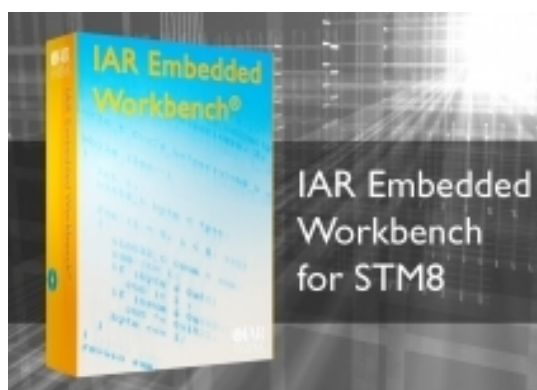
### 建议：

建议将 COSMIC C 编译器外挂于 ST TOOLSET ( STVD ) 中, 使用 ST 提供的集成开发环境 STVD 来进行软件的调试开发。

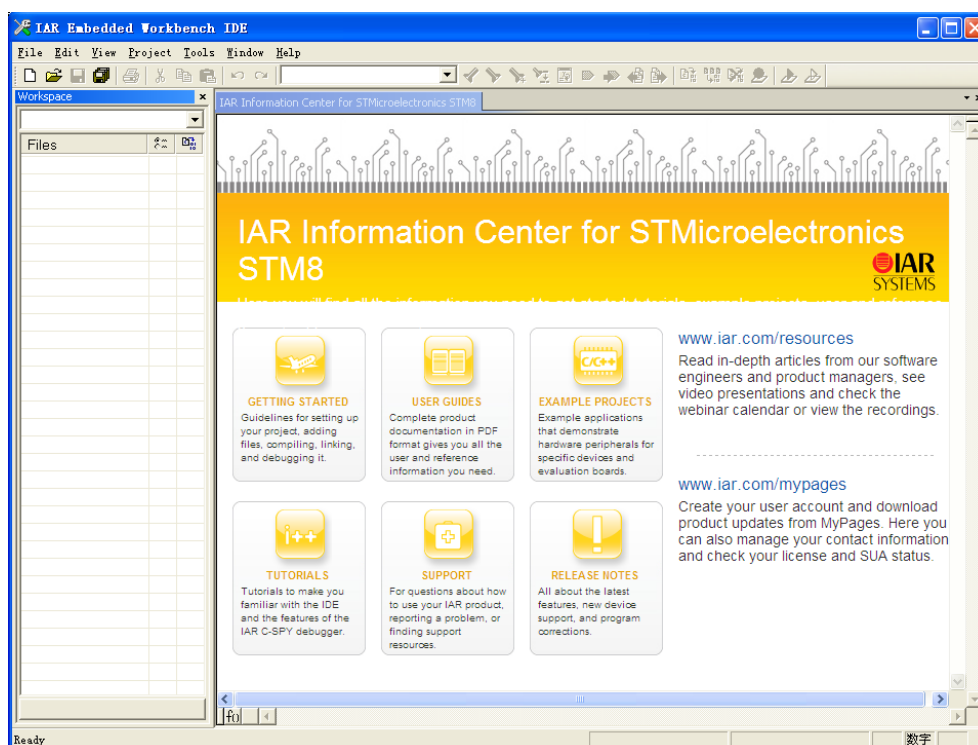


## 2.3 IAR

IAR Systems 推出开发工具“STM8 系列嵌入式设计工作台”（EWSTM8），支持 8 位微控制器市场主流的 STM8（STM8A,STM8L,STM8S）系列产品。IAR EWSTM8 嵌入式设计工作台提供一整套开发工具，包括一个项目管理器、编辑器和项目创建工具（C 语言编译器和链接器）。该工作台还为开发人员提供调试功能，可以连接意法半导体价格低廉的在线调试器 ST-LINK 以及先进的高端仿真器 STice。



IAR 提供 8K 代码限制和 30 天评估版的 C 编译器。可从 [www.iar.com](http://www.iar.com) 网站上直接下载。安装完成之后，运行 IAR 集成开发环境，其界面如图所示：



## 3 STM8 创建项目

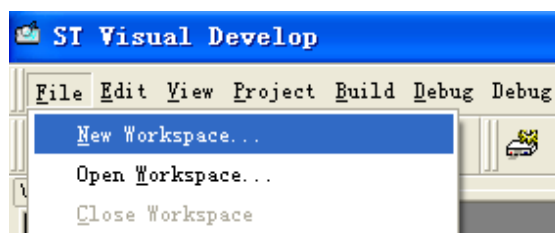
### 本章提示：

- ◇ 本章主要介绍如何使用不同的编译器建立 STM8 的应用项目
  - ✓ STVD：创建一个汇编语言项目
  - ✓ COSMIC：创建一个 C 语言项目
  - ✓ IAR：创建一个 C 语言项目

### 3.1 使用 STVD 创建汇编项目

#### 3.1.1 新建

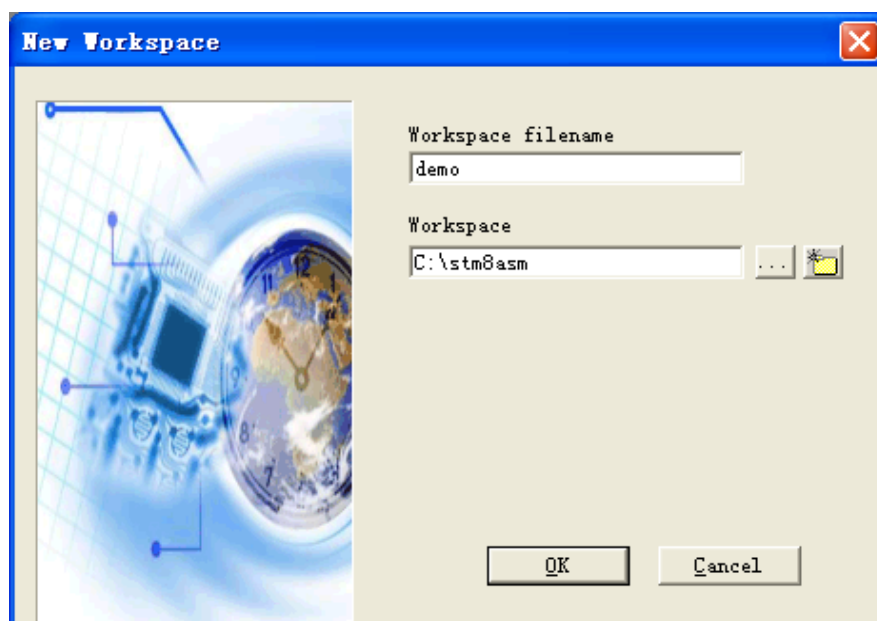
从开始菜单启动 ST Visual Develop ( STVD )。在 STVD 中选择菜单 “File>New Workspace...”，新建一个工作区。如下图所示：



在出现如下图的对话框中，选择 “Create workspace and project”，如下图所示：

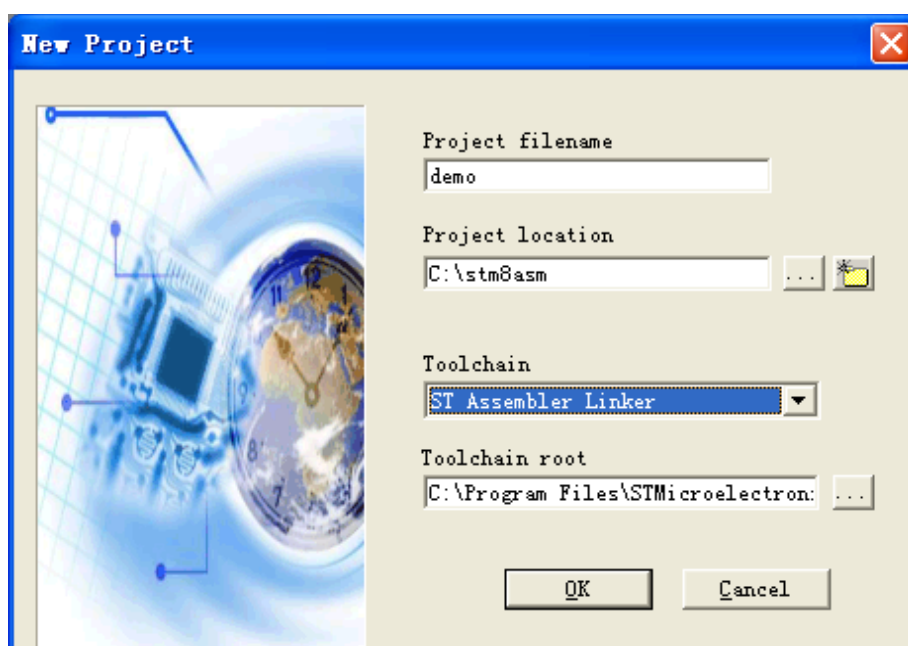


点击“确定”，出现如下对话框，在 Workspace filename 中输入 workspace 名字：demo，选择 workspace 所要保存的路径：c:\stm8asm。

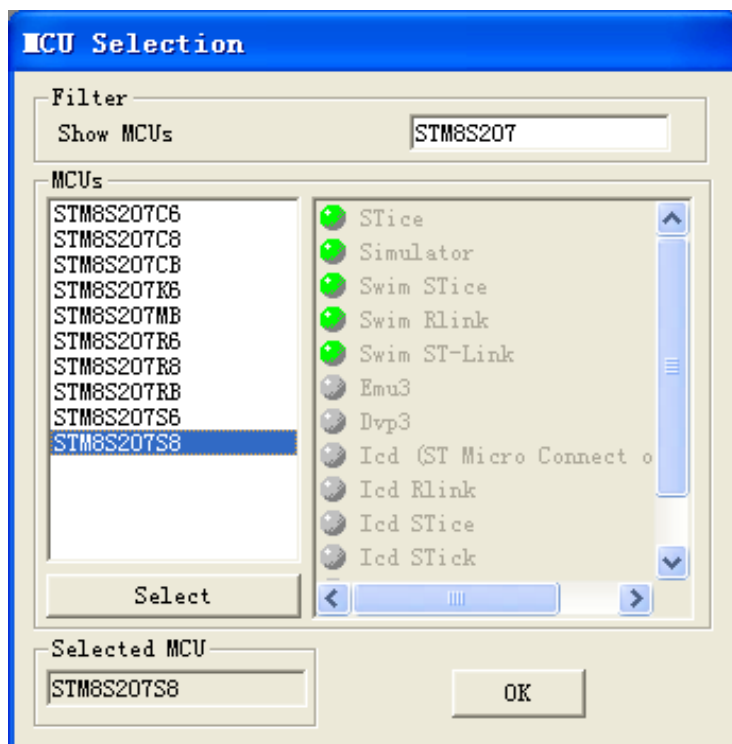


在 workspace filename 完成后，点击“OK”，  
在 project filename 中输入 project 名字: demo。  
在 Project location 中选择 project 保存的路径，一般与 workspace 保存在同一个目录下。

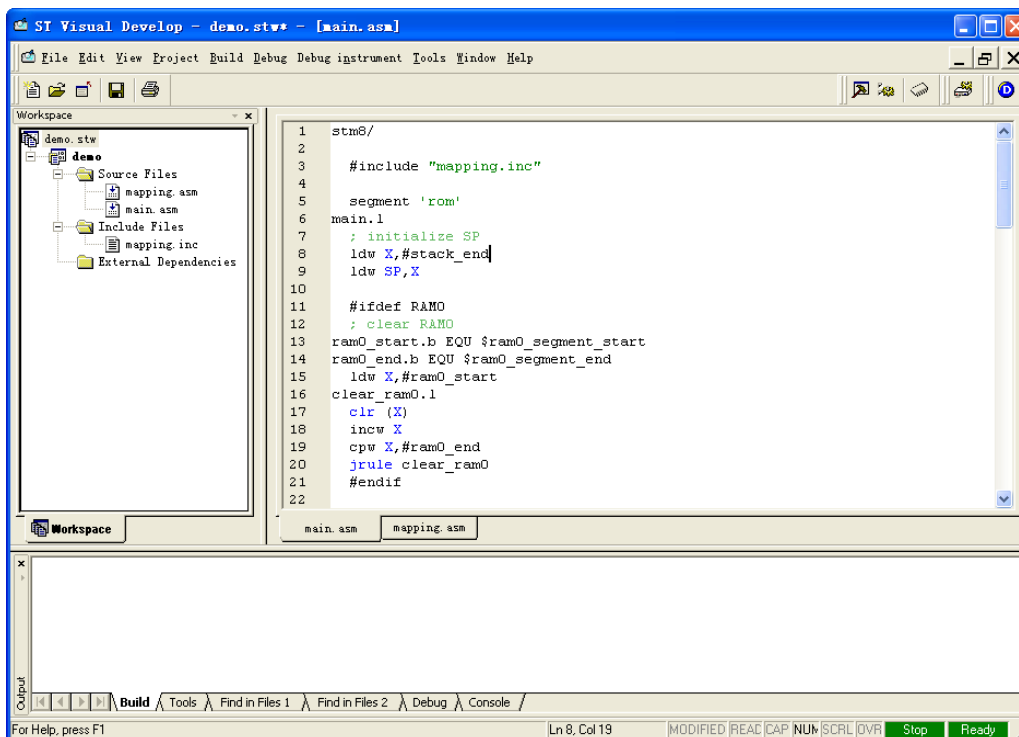
选择工具链 Toolchain: ST Assembler Linker，ST Assembler 是 ST 提供的免费的汇编编译器。Toolchain Root 一般是默认安装的。如果在 ST TOOLSET 安装过程中改变了安装目录，需要确认安装路径。相关设置如图所示：



在 project filename 完成后，点击“OK”，在 MCU Selection 对话框中，选择 MCU 型号。可以在空白框中输入型号中的部分字符可快速筛选目标型号。如图所示：



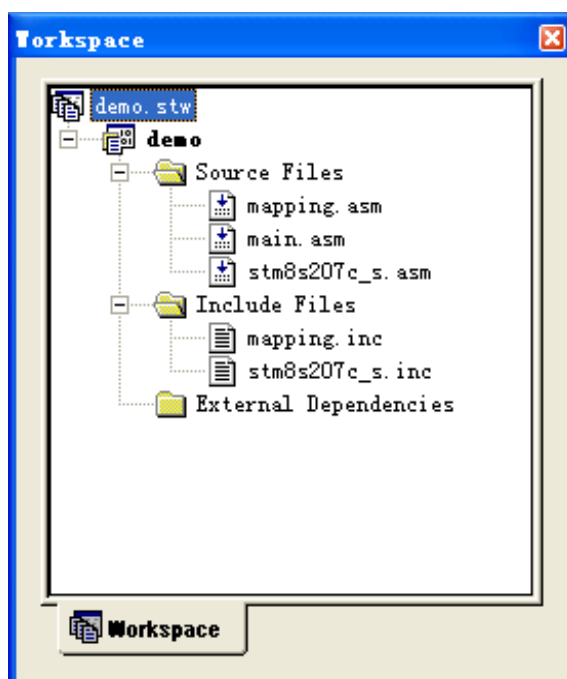
至此，workspace 和 project 创建完成。完成后界面如下图所示：



STVD 在项目中自动添加了 main.asm, mapping.asm 和 mapping.inc 文件。其说明如下：

文件	说明
main.asm	<ul style="list-style-type: none"> <li>- 基本的程序架构</li> <li>- 文件中包含了中断向量和中断函数 ( NonHandledInterrupt ) , 用户可根据此进行参考修改</li> <li>- 清 RAM0, RAM1 和清堆栈程序。用户可删除此部分程序, 自己进行变量的初始化。建议在程序开始时对自己定义的变量初始化。</li> </ul>
mapping.asm	- 定义了段的名字和地址(ram0, ram1, stack, eeprom, rom, vectit)
mapping.inc	- 定义了段 ( ram0, ram1, stack ) 的起始和结束地址

从 C:\Program Files\STMicroelectronics\st\_toolset\asm\include (默认安装目录) 找到相关 MCU 型号的寄存器定义文件到当前工程目录下 ( 本例中用到 STM8S207C\_S.ASM 和 STM8S207C\_S.INC ) , 并添加到工程项目中。添加后的 workspace 如下图所示：



用户需要根据自己的要求, 修改 main.asm 和 interrupt.asm 文件。

在修改中断时, 先在中断向量表中找到对应的中断地址, 把 NonHandledInterrupt 中断名字修改成自定义的中断名字, 其他部分不需要修改 ( \$82000000 是编译器定义的地址 )。然后使用一个 interrupt 关键字定义一个中断函数。

下面是一个自定义中断函数的例子：

```
;自定义中断函数
    interrupt My_Interrupt_Name
My_Interrupt_Name.1
;
;...中断处理代码
;
    ired

;中断向量表
segment 'vectit'
    dc.l {$82000000+main}          ; reset
    dc.l {$82000000+ My_Interrupt_Name} ; trap
    /...
    ;其他中断
    /...
end
```

**说明：**

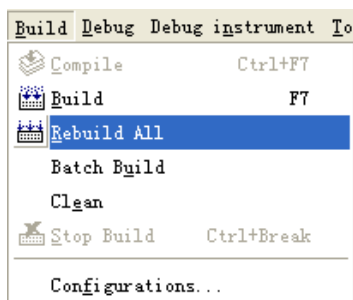
- dc.l {\$82000000+main}，复位向量在复位后直接跳转到 main 处执行。

修改 main.asm，并说明如下：

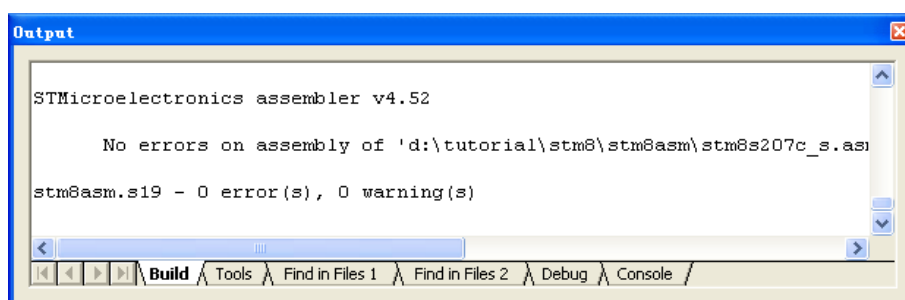
<pre>stm8/ ; 第一行必须保留而且要顶格写.指定目标单片机的指令集 ;* 包含文件 #include "mapping.inc" #include "stm8s207c_s.inc"</pre>	<b>Include 区</b>
<pre>;* 常量定义 #define MYDF1 3 MYCNT1 EQU 2</pre>	<b>常量区</b>
<pre>;/* ram0 区变量定义 segment 'ram0' ;从此行以后是 ram0 变量区 MY_RAM0_VAR1 DS.B 1 ;保留 1 个字节的变量空间</pre>	<b>RAM0 变量区</b>
<pre>;/* ram1 区变量定义 segment 'ram1' ;从此行以后是 ram1 变量区 MY_RAM1_VAR1 DS.B 128 ;定义 128 个字节的 一组变量空间</pre>	<b>RAM1 变量区</b>
<pre>;* 主程序 (ROM) segment 'rom' ;从此行以后是 rom 代码区 main.l ; 初始化 main_loop.l ;..... JRA main_loop</pre>	<b>主程序区</b>
<pre>Subroutine.l Ret</pre>	<b>子程序区</b>
<pre>;* 中断程序 interrupt NonHandledInterrupt NonHandledInterrupt.l Iret</pre>	<b>中断程序区</b>
<pre>;*中断向量映射 segment 'vectit' dc.l {\$82000000+main} ; reset dc.l {\$82000000+NonHandledInterrupt} ; trap dc.l {\$82000000+NonHandledInterrupt} ; irq0 ;..... (此处省略) dc.l {\$82000000+NonHandledInterrupt} ; irq28 dc.l {\$82000000+NonHandledInterrupt} ; irq29 end</pre>	<b>中断向量映射</b>

### 3.1.2 编译

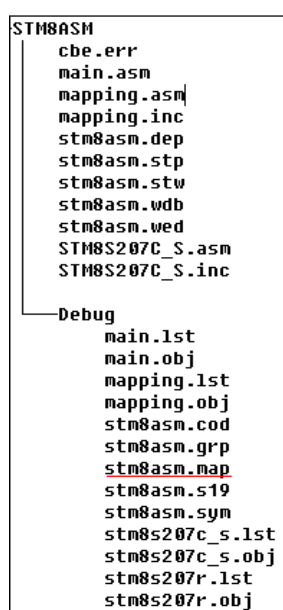
源程序编辑完成后，可选择菜单“Build>Rebuild All”，编译工程。如下图所示：



若源程序正确无误，会显示如下：



若要查看编译后的程序代码的大小，可找到工程目录下的 Debug 目录，在 Debug 目录下有个.map 文件。本例是 stm8asm.map，如下所示：





用文本编辑器打开.map 文件，可查看使用的资源的大小。

```

1
2 Segment List
3 -----
4
5 d:\tutorial\stm8\stm8asm\main.asm[1] 90- 111 0 - 9 'ram0' [void]
6 d:\tutorial\stm8\stm8asm\main.asm[2] 112- 126 100 - 1E3 'ram1' [void]
7 d:\tutorial\stm8\stm8asm\main.asm[3] 127- 504 8080 - 81F5 'rom' [text]
8 d:\tutorial\stm8\stm8asm\main.asm[4] 505- 538 8000 - 807F 'vectit' [data]
9 d:\tutorial\stm8\stm8asm\stm8s207c_s.asm[2] 16- 315 5000 - 7F76 'periph2' [void]
10
11
12 Class List
13 -----
14
15 0 ' ram0' byte from 0 to 9 (lim FF) = 3% D
16 0 ' ram1' byte from 100 to 1E3 (lim BFF) = 8% D
17 0 ' stack' byte from C00 to BFF (lim FFF) = 0% V
18 0 ' eeprom' byte from 4000 to 3FFF (lim 45FF) = 0% V
19 0 ' rom' byte from 8080 to 81F5 (lim 17FFF) = 0% C
20 0 ' vectit' byte from 8000 to 807F (lim 807F) =100% S
21 0 ' periph' byte from 0 to FFFFFFFF (lim 7F) = 0% V
22 0 ' periph2' byte from 5000 to 7F76 (lim FFFFFFFF) = 0% D

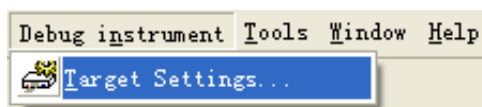
```

在 stm8asm.map 文件里，记录了 ram0,ram1,rom,stack,eeprom,rom 等的使用情况。本例的代码地址范围是：'rom' byte from 8080 to 81F5。程序代码大小是：  
 $81F5 - 8080 = 175(\text{HEX}) = 373$  个字节

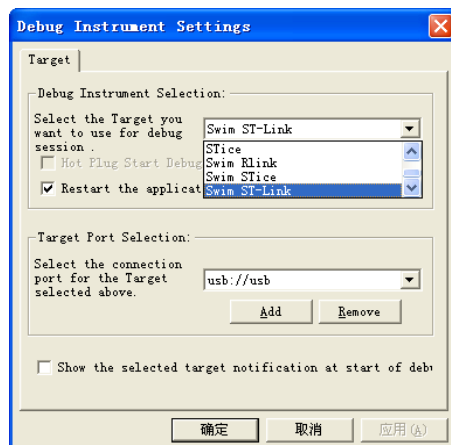
### 3.1.3 调试

本节介绍在 ST Visual Develop 环境中的软件调试说明。

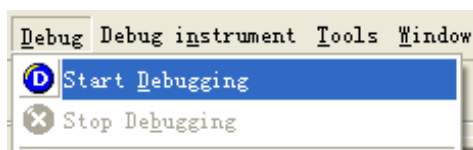
首先，先选择调试与仿真的工具。可通过菜单“Debug instrument>Target Settings”来设置，如下图所示：



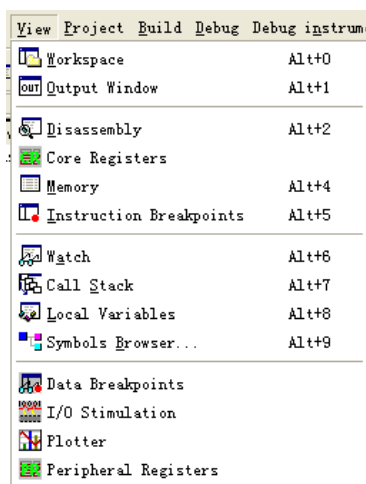
在弹出的对话框中，列表中列出了此开发环境支持的所有工具。常用的在线调试工具是 Swim Rlink 和 Swim ST-LINK，Simulator 是软件仿真。



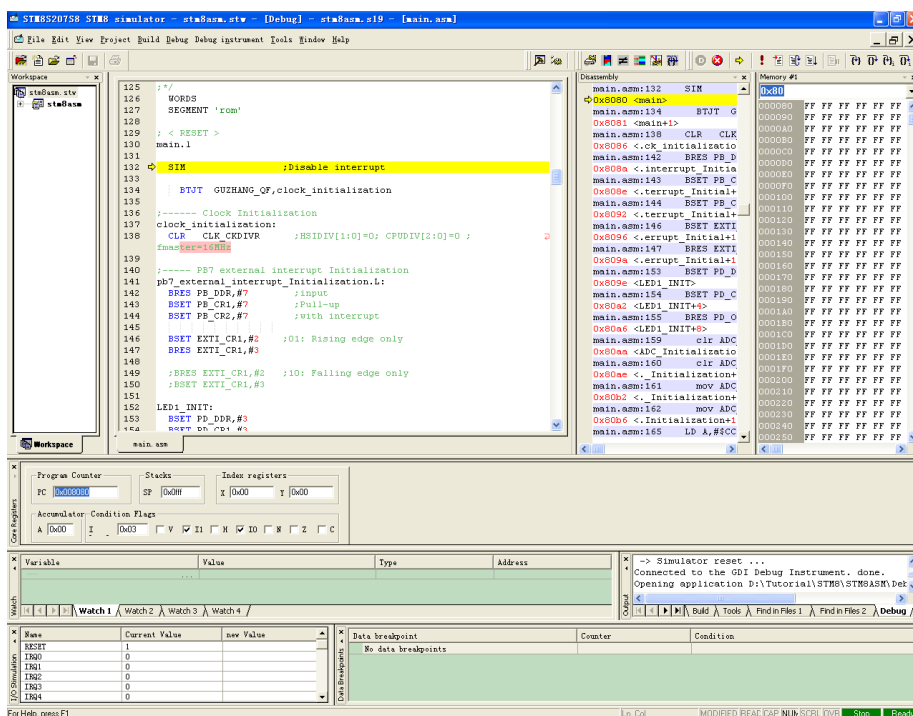
可从菜单选择“ Debug> Start Debugging”，可进入调试状态，如下图所示：



可通过 View 的下拉菜单的菜单项进行不同的显示，如图所示。



进入调试状态后，界面如下图所示：



## 3.1.4 烧录

### 3.1.4.1 使用 STVD 中烧录

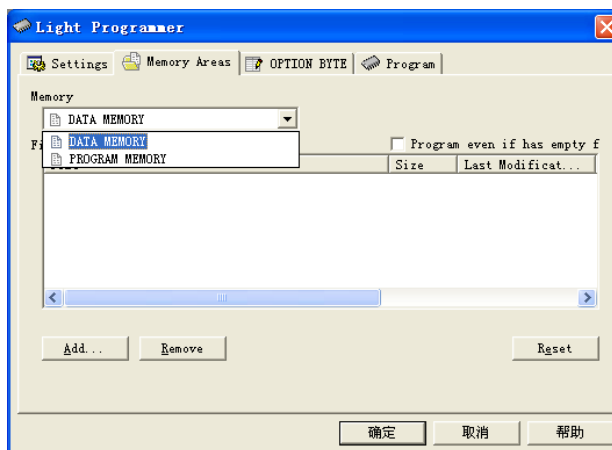
选择 Tools > Programmer ，出现如下烧录界面。请确认烧录的 Target Device 型号，选择正确的硬件设备。



#### 3.1.4.1.1 Settings 选项

对于 STM8 系列单片机的 Hardware> hardware 可选择 RLINK，ST-LINK，STICE 等工具进行烧录。“Run STVP”可以直接运行 STVP，若不想使用 STVD 的编程器。

#### 3.1.4.1.2 Memory Areas 选项

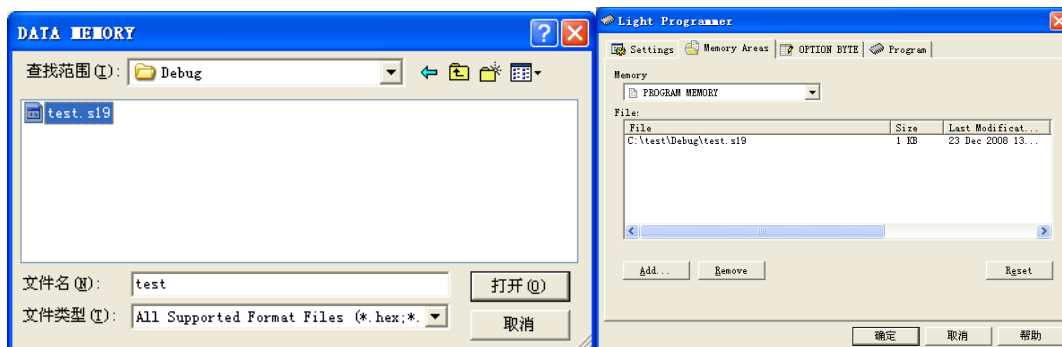


Memory 可以选择 DATA MEMORY 和 PROGRAM MEMORY。

DATA MEMROY : EEPROM 数据

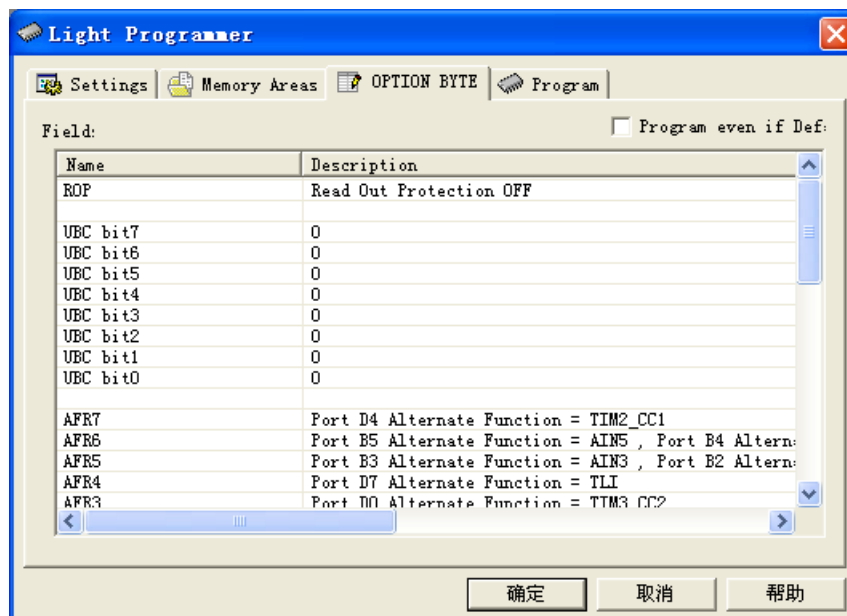
PROGRAM MEMORY : 程序

点击 Add... 可以添加要烧录的目标文件，如图所示



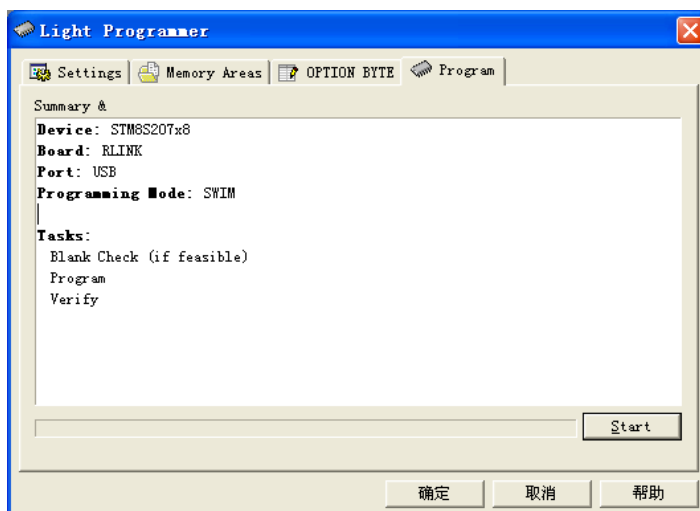
### 3.1.4.1.3 OPTION BYTE 选项

根据应用配置合适的 OPTION BYTE 选项



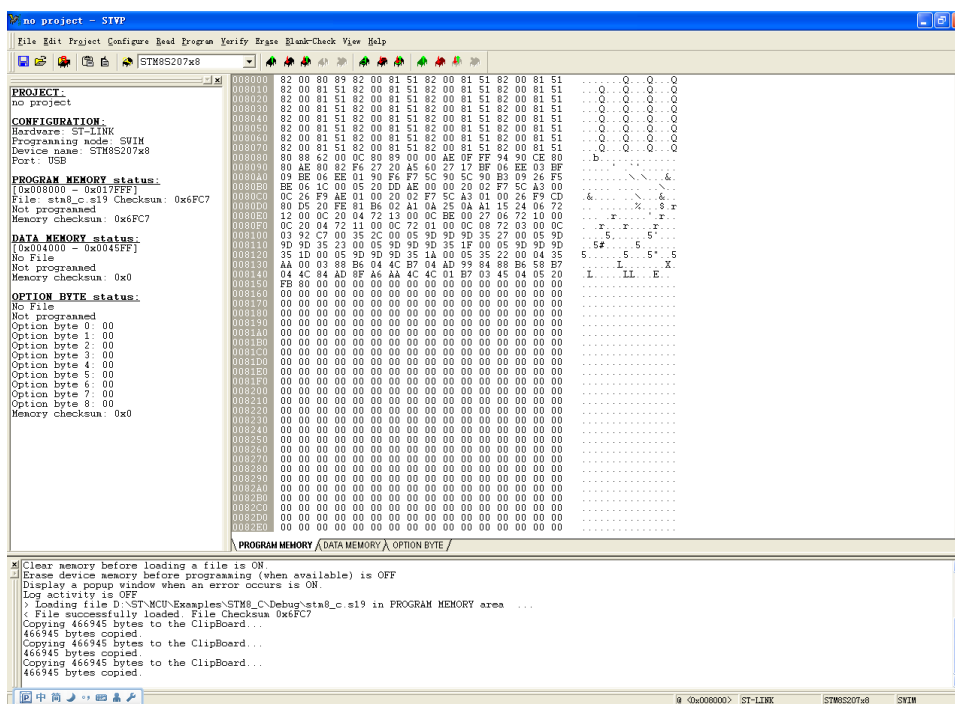
### 3.1.4.1.4 Program 选项

配置完成之后，点击 Start 即可可以进行烧录



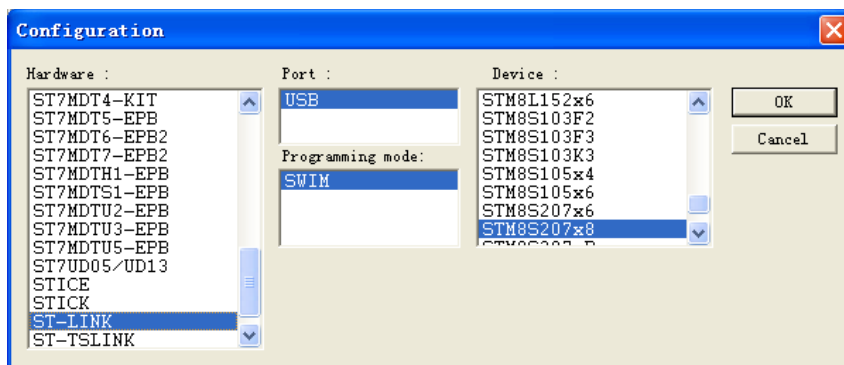
### 3.1.4.2 使用 STVP 烧录

可以运行独立的烧录软件 ST Visual Programmer (STVP)，运行“开始”>ST Toolset>Development Tools > ST Visual Programmer。如图所示：



### 3.1.4.2.1 烧录软件配置

运行 Configure > Configure ST Visual Programmer , 如图所示 :



Hardware: 烧录工具

Port: USB

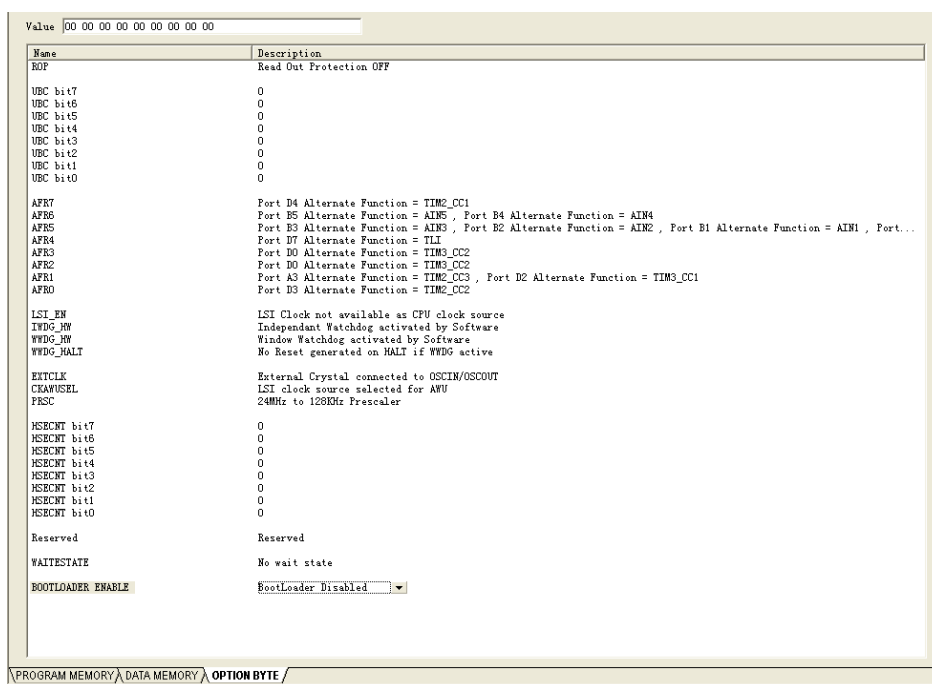
Programming mode: SWIM

Device: 选择要烧录的型号

打开要烧录的目标文件 File > Open...

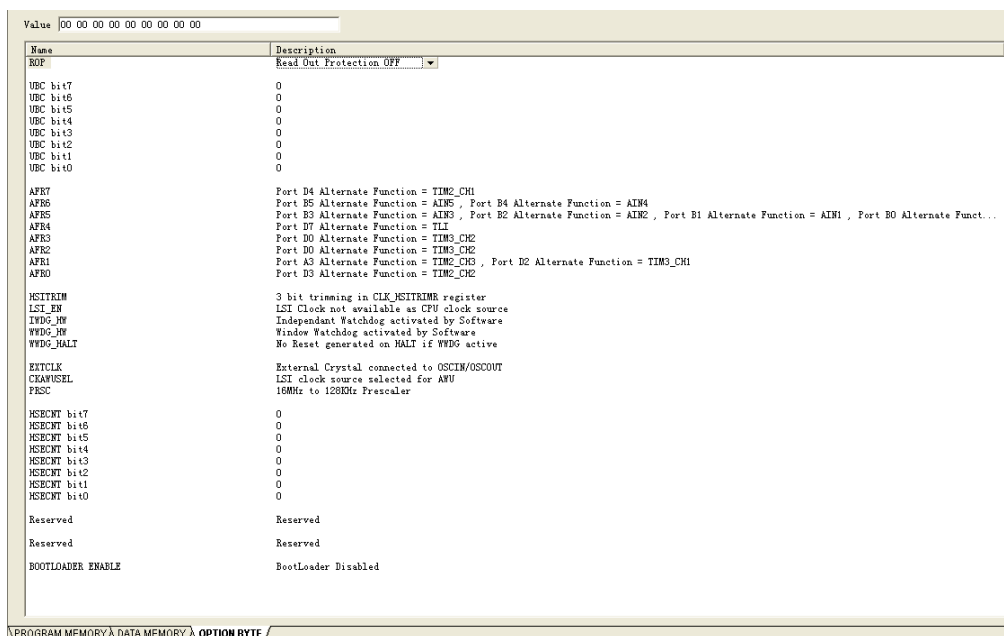
DATA MEMORY: EEPROM 数据区

### 3.1.4.2.2 配置 OPTION BYTE 选项



Value: 可直接在此框内直接输入配置好的 OPTION BYTE 值，OPTION BYTE 内容根据 Value 的值自动配置好。

以 STM8S105S4 为例说明 OPTION BYTE 配置：



**ROP**：是读出保护设置。若设置了 ON，那么程序是就无法读出。（ST 的保密性比较高）

**UBC [7:0]**：用户启动代码区。一般用户在做 IAP 时，需要保护的代码部分设置。

**AFR[7:0]**：备选功能重映射选项。通过此来设置需要的功能。比如同一个引脚会有不同的功能。可通过此选项来设置需要的功能。

**HSITRIM**：高速内部时钟调节寄存器大小

**LSI\_EN**：低速内部时钟使能

**IWDG\_HW**：独立看门狗

**WWDG\_HW**：窗口看门狗激活

**WWDG\_HALT**：当芯片进入停机模式时窗口看门狗的复位动作

**EXT\_CLK**：外部时钟选择

**CKAWUSEL**：自动唤醒单元/时钟

**PRSC[1:0]**：AWU 时钟预分频

**HSECT[7:0]**：HSE 晶体振荡器稳定时间

**BOOTLOADER ENABLE**：如果用户使用 UART 来下载程序，可通过此选项位来设置。

更多的信息，可参考 STM8S105S4 的数据手册

## 3.1.5 汇编语言相关说明

### 3.1.5.1 汇编源代码行格式

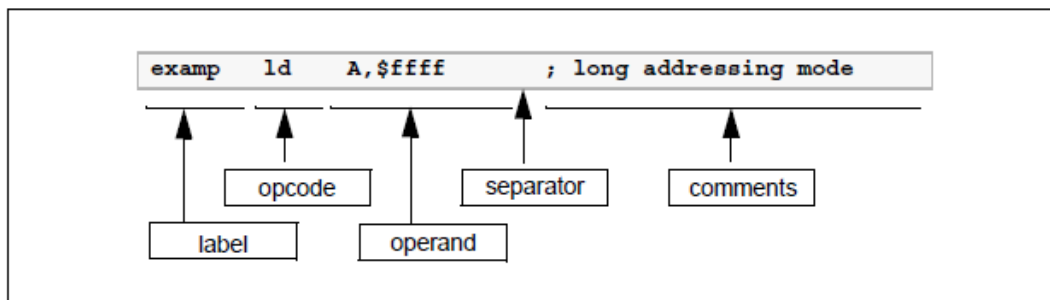
程序的第一行，必须顶头写一条 `stm8\`，STVD 会调用 `stm8.tab` 描述文件。写错或不写都不能编译。

```
stm8\
```

汇编源代码行的一般格式如下：

```
[label[:]] <space> [opcode] <space> [operand] <space> [;comment]
```

一个源代码行的例子，如下：



下面对源代码行进行一些说明：

在汇编语言中，任何超过 30 个字符的 label 都会被编译器截短。并且会给出一个警告。若截短后两个符合同名，则会报一个错误。

Label 的长度大小，默认地，是 WORD 类型（2 个字节长）。也可以通过加后缀来指定 label 的长度。

```
.b for byte
.w for word
.l for long
```

例如：

```
lab      equ 0      ; word-size label (default)
label1.b equ 5      ; byte-size label
label2.l equ 123    ; long label

        segment byte at: 80 'ram'
        bytes      ; force the size of the label to bytes
count ds.b      ; byte-size label
pointer ds.w    ; byte-size label with a word-size
                ; space reserved at this address
                ; 字节 label, 在此地址保留一个 word-size 空间
```



在多文件项目中，如果一个程序文件中要引用另一个程序文件定义的 label,定义此 label 的另一个程序文件必须声明为 PUBLIC,引用此 label 的程序文件必须声明此 label 为 EXTERN。EXTERN 和 PUBLIC 必须是成对出现。

例如：

```

module 1
EXTERN    _sig1.w          ; import _sig1
EXTERN    _sig2.w          ; import _sig2
PUBLIC    _handlers       ; export _handlers
        segment byte 'P'
_handlers:                ; define _handlers
jp _sig1                    ; refer to _sig1
jp _sig2                    ; refer to _sig2
        end

module 2
EXTERN    _handlers.w      ; import _handlers (addr. is a word)
PUBLIC    _sig2            ; export _sig2
        segment byte 'P'
_sig2:                    ; define _sig2
...
call _handlers             ; refer to _handlers
...
ret
end

```

### 3.1.5.2 数字表达方式

汇编编译器默认地支持 MOTOROLA 数字格式。也可以通过使用.INTEL, .TEXAS, .ZILOG 指令支持 INTEL, TEXAS, ZILOG 格式。

格式	Hex	Binary	Octal	当前 PC 指针
Motorola	\$ABCD or &ABCD	%100	~665	*(use MULT for MULTIPLY)
Intel	oABCDh	100b	665o or 665q	\$
Texas	>ABCD	?100	~665	\$
Zilog	%ABCD	%(2)100	%(8)665	\$

例如：

```

lab03 equ 10          ; 十进制数 10
lab04 equ $10         ; 十六进制数 10
ld A,$ffff           ; 长寻址模式
ld A,#$cb             ; 立即数寻址模式
ld A,#100             ; 十进制记数法

```

### 3.1.5.3 寻址模式

不同的寻址模式，操作数的目标地址范围也不同。不同寻址模式范围，如下：

- 0-\$FF，短寻址模式，1 个字节
- \$100-\$FFFF，长寻址模式 2 个字节
- \$10000-\$FFFFFF，扩展寻址模式，3 个字节

在应用中，若对数据存取有速度要求，建议将一些存取速度要求快的变量存放到 0-\$FF 区间，也就是“ram0”区。这样，数据的存取速度最快。

### 3.1.5.4 汇编伪指令

#### 3.1.5.4.1 段伪指令 SEGMENT

STM8 汇编编译器是通过 SEGMENT 来组织和管理代码和数据的。可以通过 SEGMENT 伪指令，来定义可寻址空间的指令代码和数据。

SEGMENT 的格式如下：

[<名称>] SEGMENT <排列类型> <组合> <类别> [COD]

- **[<名称>]**：此项可以省略，但当一段在多个地方使用时，必须用名字定义，且名称应相同，同一名称的多个段，将被分配到一个相邻的地址空间。名称最多可以用 11 个字符表示。
- **<排列类型>**：在定义段的时候，该段的起始地址被自动地分配到表 2 所要求的地址，默认的排列类型是 Para。

排列类型	性质	举例
Byte	没有专门要求，任何地址都可以	
Word	起始地址是偶数	2
Long	起始地址是 4 的倍数	4
Para	起始地址是 16 的倍数	10h
64	起始地址是 64 的倍数	40h
128	起始地址是 128 的倍数	80h
Page	起始地址是 256 的倍数	100h
1K	起始地址是 1024 的倍数	400h
4K	起始地址是 4096 的倍数	1000h

## &lt;组合选项&gt;

关键字	性质
没有	如果没有该组合选项，当前定义的段就被分配到同类中最后一个已被分配段的后面
AT, 开始地址[-结束地址]	定义一个新的类所在的存储空间的起始地址和结束地址。一个类的起始地址和结束地址只能定义一次，结束地址部分可以省略。开始地址和结束地址均为十六进制数
Common	所有同类且同名的用 common 定义的段，将被分配到具有相同起始地址的存储区。地址由连接器定义

## &lt;分类名称&gt;

类是段的组合。类的概念本身没有任何其他属性，可以创建多个类。类的概念可以根据不同内存区的特性，帮助组织可寻址空间。下表列出了一组典型的分类的例子。

类名称 ( 举例 )	类的类型与用途
'ROM'	只读存储器，用于存储程序
'RAM0'	读写存储器，在 0 页 ( 地址在 100h 以下 )
'RAM'	读写存储器，用于 16 位地址区
'STACK'	读写存储器，用于堆栈
'IO'	用于 I/O 寄存器 ( 地址始终在 0 页 )
'VECTOR'	中断矢量地址区

## [COD]

选择项[COD]是一个 0-9 的数字，如果没有此选项，默认值是 0。

**3.1.5.4.2 汇编结束指令 END**

这条伪指令用在程序文件的末尾，表示某个程序文件的结束。包含文件不需 END 作为结束。

**3.1.5.4.3 定义变量伪指令 DS**

从编程角度上讲，变量是一段用于保存特定数据对象的内存。伪指令 DS 的功能是：从指定地址开始保留 DS 之后表达式的值所规定的数量的存储单元，以备使用。变量数据的存储类型由伪指令 DS.B、DS.W 和 DS.L 来指定，它们分别表示存储 1 字节、1 个字或者双字的数据，它们的格式如下：

操作码	操作数
DS.B	[需分配字节的数量]

DS.W                    [需分配字的数量]  
DS.L                    [需分配长字的数量]

若在上述指令中未指定需分配的数量，则默认值为 1。DS 指令前可加一个标号，由 DS 定义的操作数的首字节地址与标号名称相关联，也与前一条指令所分配的存储地址相关联。它表示指定类型的变量从与标号相关的地址开始分配。

例：     segment byte at 50 'RAM' ;假设 aByte 起始地址为 50h  
      aByte: DS.B 1            ;定义一个字节变量  
      aWord : DS.W 1         ;定义一个字变量

汇编后，aByte 的地址将是 50h，从 50h 开始保留一个字节；aWord 地址为 51h，从 51h 开始保留 1 个字或 2 个字节。通过伪指令 SEGMENT 指定类与段的地址，汇编程序即可将 DS 定义的变量分配到确定的地址。

### 3.1.5.4.4 定义常量伪指令 DC

在汇编程序中，常量可以是 2 种类型之一：常数或者符号。定义常量的伪指令有如下几条：DC.B、DC.W、DC.L、BYTE、WORD、LONG 和 STRING。DC 伪指令与上述的 DS 伪指令工作方式相似，它的功能是从内存指定单元开始定义（存储）若干个字节的数值或 ASCII 码字符，常用于定义数据常数表。它们的格式如下：

操作码	操作数
DC.B	<表达式或字符串>[, <表达式或字符串>.....]
DS.W	<表达式>[, <表达式>.....]
DS.L	<表达式>[, <表达式>.....]

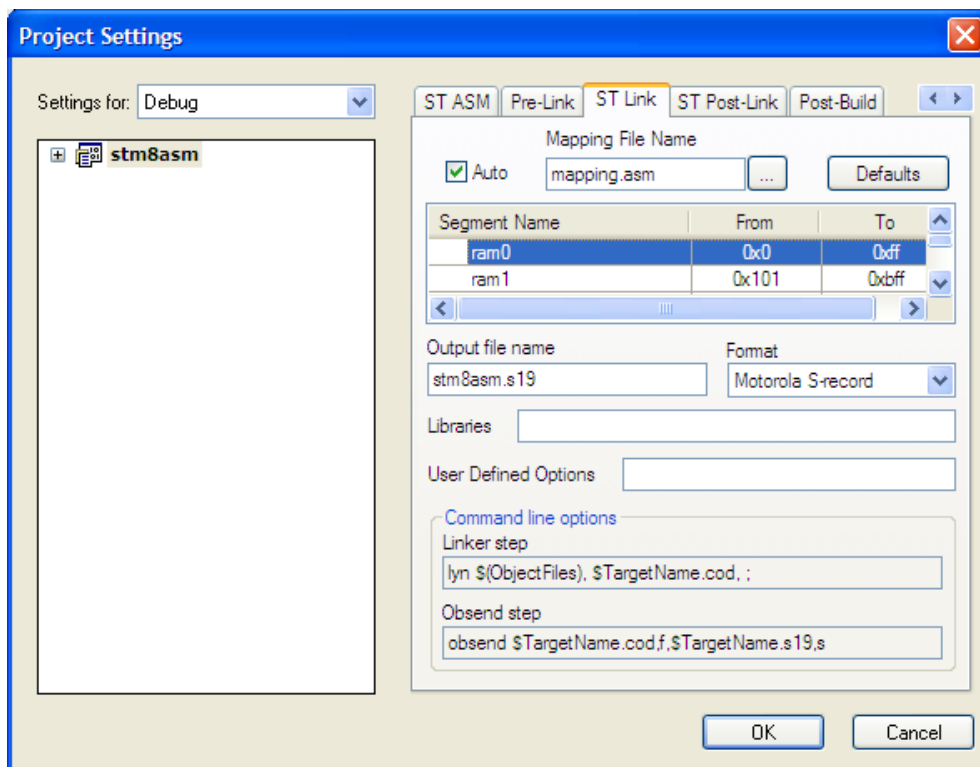
它们将内存设置为操作数或指定的值。

例： DC.B 1,2,3                ;定义了 01,02,03 共 3 个字节  
      DC.B "HELLO",03        ;定义了 48,45,4C,4C,4F,03 共 6 个字节  
      DC.W 1,2,3,4,\$1234     ;定义了 0001,0002,0003,0004,1234h 共 5 个字  
      DC.L 1,\$12345678       ;定义了 00000001,12345678 共 2 个长字

### 3.1.5.5 项目设置

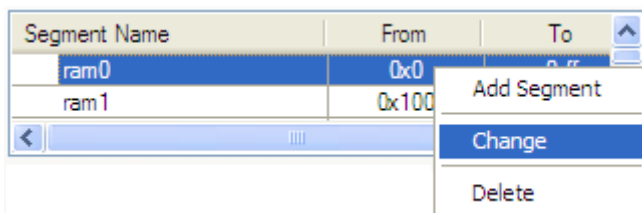
对项目的管理与设置，一般是通过菜单“Project>Settings...”来设置的。可依据项目的需要进行设置。

下面对“ST LINK”选项，做个说明。选项内容如下图所示：



默认地，Auto 是自动选择的。也就是说，STVD 在创建工程的时候，会自动创建一个 mapping.asm 和 mapping.inc 文件，并自动关联 mapping.asm 文件。在两个 mapping 文件里面定义了段的名称和开始结束地址。

在 segment 列表中，右击会出现如下菜单，可以对段进行“添加新段”、“更改”、删除等操作。若有更改，STVD 则会自动更新 mapping.asm 和 mapping.inc 文件。



“Add Segment”，可以创建自己的段。在新定义的段里，可以存放一些自定义的数据和程序代码，便于管理和分配资源。一般地，使用默认的就是可以满足要求了。

“Change”，可以更改段的起始结束地址。默认地，在定义的地址范围内由编译器来进行管理和资源分配的，用户无需关心。如变量，编译器会自动分配其地址。

### 自定义段和变量：

在一些特殊的应用中，用户想自己来管理一些特殊的变量。如在 ram1 段中开辟 2 个字节的空间，由自己来管理分配。可添加一个新段“myram1”，修改起始结束地址，“from” 0x100 “to” 0x101。修改“ram1”的地址为 0x102。如下图所示：

Segment Name	From	To
ram1	0x102	0xbff
myram1	0x100	0x101

当添加新段完成后，在 mapping.inc 文件里面，会自动修改 ram1 的起始结束地址范围。

```
#define ram1_segment_start 102
```

在 mapping.asm 文件中，会自动添加新段'myram1'：

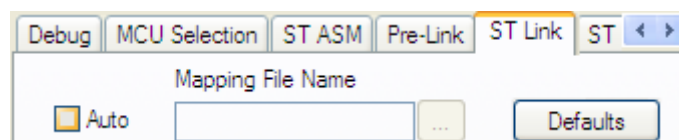
```
WORDS          ; The following addresses are 16 bits long
segment byte at 100-101 'myram1'
```

在程序中，声明'myram1'就可以使用管理自定义的段和变量了。

```
WORDS
SEGMENT 'myram1'
MYRAM1_VAR1 DS.B 1
MYRAM1_VAR2 DS.B 1
```

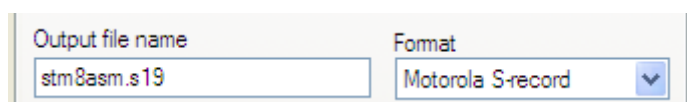
### 自定义 mapping 文件

若想定义自己的 mapping 文件，则可取消“Auto”选项。如下图所示：



在这种情况下，用户必须来写自己的 mapping 文件（可参考 STVD 自动生成的文件）；或者在程序中用 segment 伪指令来定义。

**输出目标文件格式：**



STVD 默认输出是.s19 文件（Motorla 格式）。可选择输出 Intel 的 HEX 文件。在 ST 的工具中一般都使用默认的 s19 格式文件。

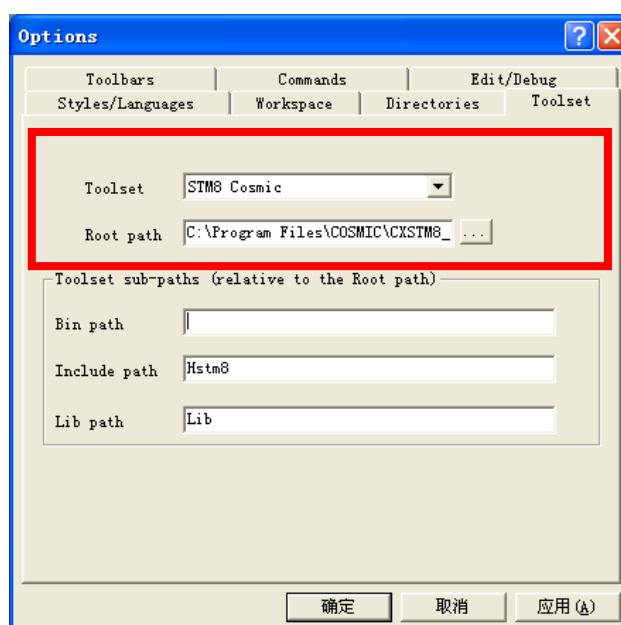
## 3.2 使用 COSMIC 创建 C 语言项目

COSMIC 提供了一个的 IDE 环境，用户可使用 COSMIC IDE 进行软件的开发。

建议在 STVD 中外挂 COSMIC C 编译器，进行软件开发。STVD 提供了比较友好的开发调试界面。

### 3.2.1 STVD 设置

要用 STVD 开发 COSMIC C 语言，首先要在 STVD 中对 COSMIC C 编译器进行设置。运行 ST Visual Develop 集成开发环境,选择菜单“ Tools -> Options”



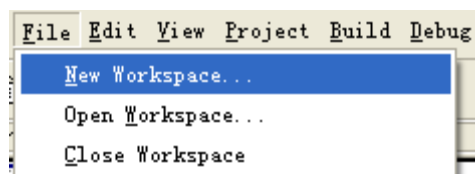
在出现的对话框中选择“ Toolset” 选项卡. 再选择下拉菜单,选中“ STM8S Cosmic”。设置编译器的安装路径。然后选择“ 确定” 。

本例中选择的“ Root path” 是：C:\Program Files\COSMIC\CXSTM8\_32K  
至此,就完成了 COSMIC C 编译器的设置完成。

**注意：**很多人在此很是疑惑，点击“确定”后，会弹出一个提醒窗口。它只是提醒你是否指定了正确的路径。如果你确认路径无误后，不要管那个提醒就可以了。在创建一个“new project”时，在 toolchain 中选择 COSMIC 时，Toolchain Root 会自动关联显示出此处设置的路径。就是能让 STVD 能找到 COSMIC C 编译器。若关联不对，则需要重新设置或重新指定。

### 3.2.2 新建

在主菜单条中，选择 File > New Workspace...



在 New Workspace 窗口中，点击 Create workspace and project 图标，然后点击 OK



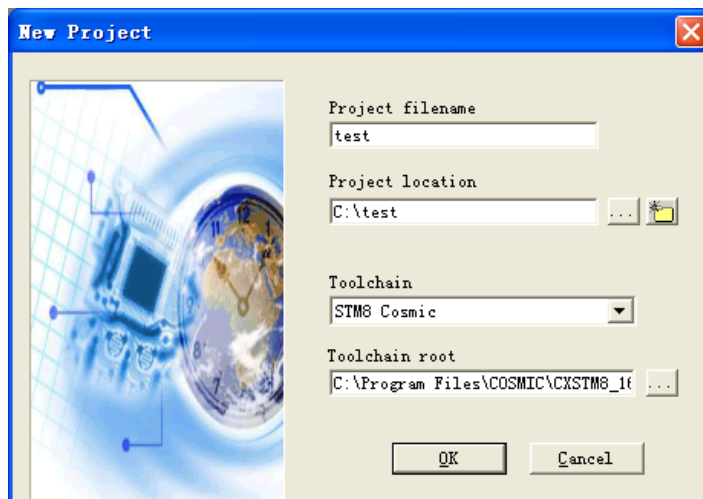
1. 在 Workspace Filename 字段中，输入一个 workspace 名字
2. 选择 workspace 和项目保存的路径

结果：

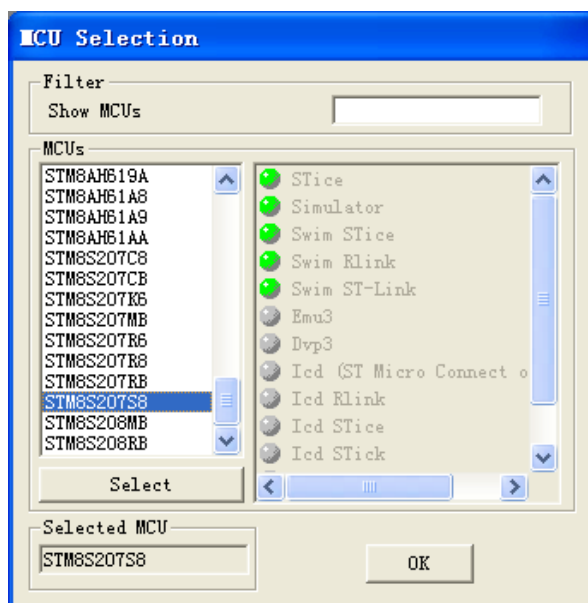
STVD 的 Workspace 窗口包含一个 workspace 图标

工作目录中包含文件：<workspace name>.stw, <workspace name>.wsp and <workspace name>.wed。

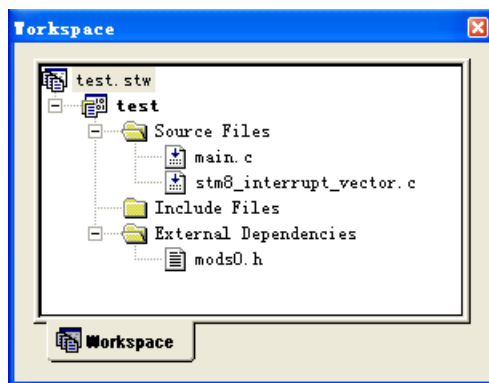




3. 在 Project filename 字段中输入一个项目名字
4. 在 Project Location 字段中选择一个工程保存的路径。默认地，使用 workspace 使用的路径。
5. 在 Toolchain 列表框中，选择 STM8 Cosmic。
6. 在 Toolchain Root 字段中，输入路径。然后点击 OK

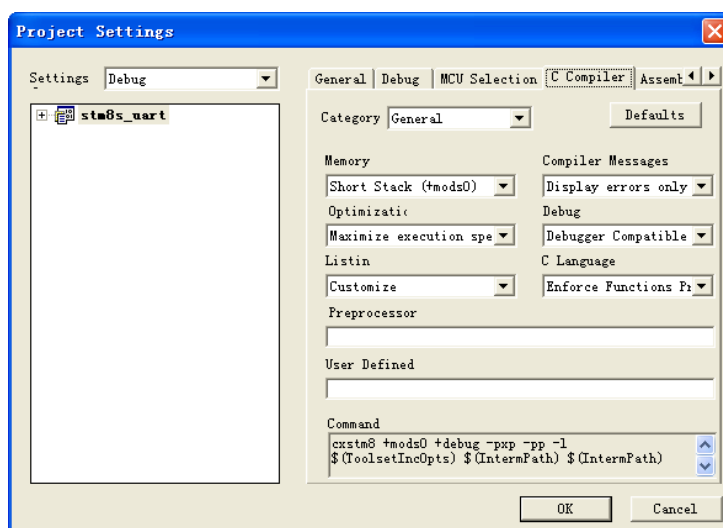


7. 在 MCU Selection 窗口中，选择需要的 MCU。也可以通过 Project Setting 窗口来选择 MCU。
8. 点击 OK
9. 保存 workspace 和 project



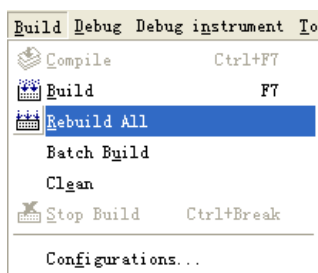
main.c 和 stm8\_interrupt\_vector.c 是自动添加的.用户只需要根据不同的 MCU 的中断向量不同填写相应的中断处理函数即可.

打开 “Project->Settings...”, 可以对当前的工程进行设置, 包括 C 编译器、连接器等选择。

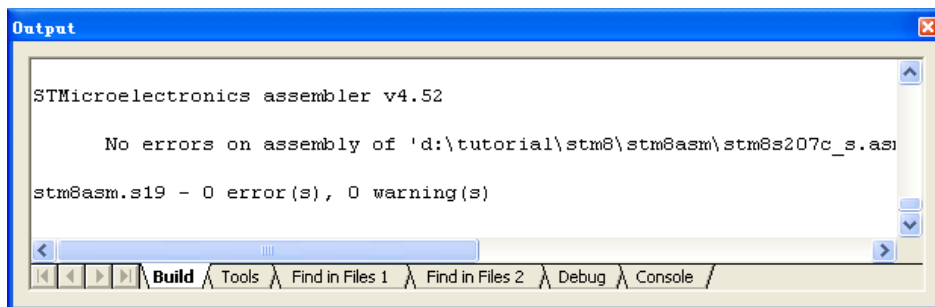


### 3.2.3 编译

选择菜单 “Build>Rebuild All”, 编译当前的工程。如下图所示：



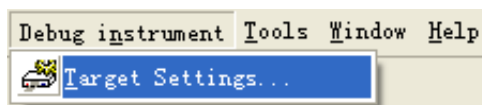
若程序正确无误，会显示如下：



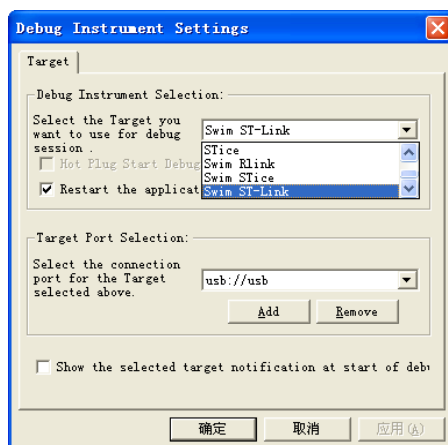
若要查看编译后的程序代码的大小，可找到工程目录下的 Debug 目录，在 Debug 目录下有个.map 文件。

### 3.2.4 调试

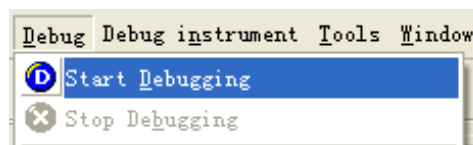
首先要设置好调试工具,可通过菜单 “Debug instrument>Target Settings”来设置，如下图所示：



在弹出的对话框中，列表中列出了此开发环境支持的所有工具。一般常用工具是 Swim ST-LINK。



可从菜单选择“ Debug> Start Debugging”，可进入调试状态，如下图所示：



### 3.2.5 烧录

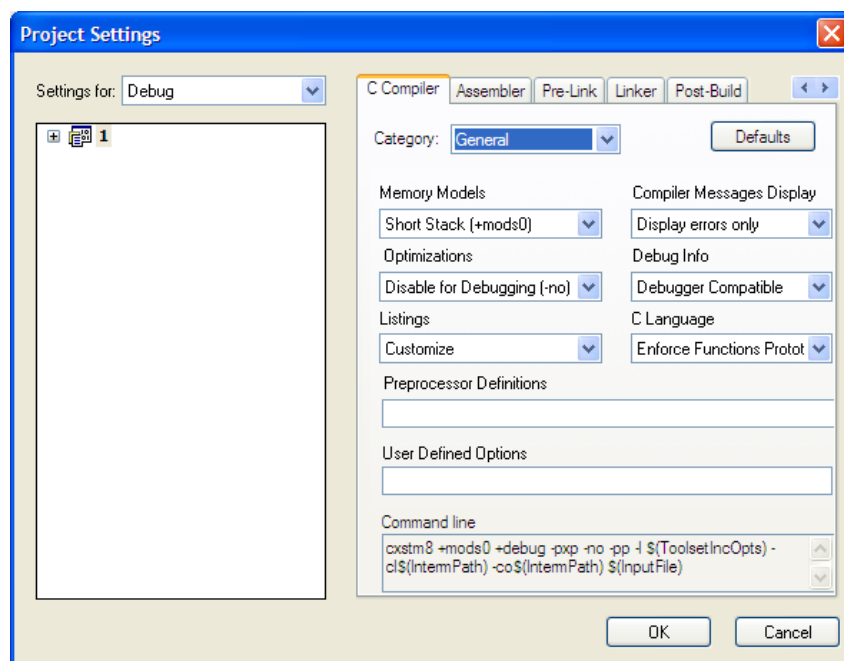
请参考 STVD 汇编语言程序设计章节中的“烧录”。因界面一样，操作类似。

### 3.2.6 COSMIC C 语言相关说明

#### 3.2.6.1 COSMIC C 语言项目设置

通过菜单“Project>Setings...”，可以对当前工程进行项目设置。包括编译器选项和连接选项等。

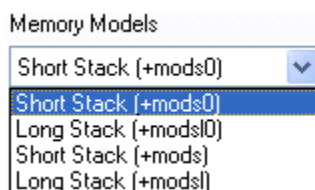
“C Compiler”选项：



说明：

- Memory Models：

通过下来菜单，可选择不同的存储器模式。如下图：



存储器模式说明如下：

➤ 代码小于 64K

STM8 编译器支持两种存储器模式。函数指针和数据指针默认是@near 指针（2 个字节）

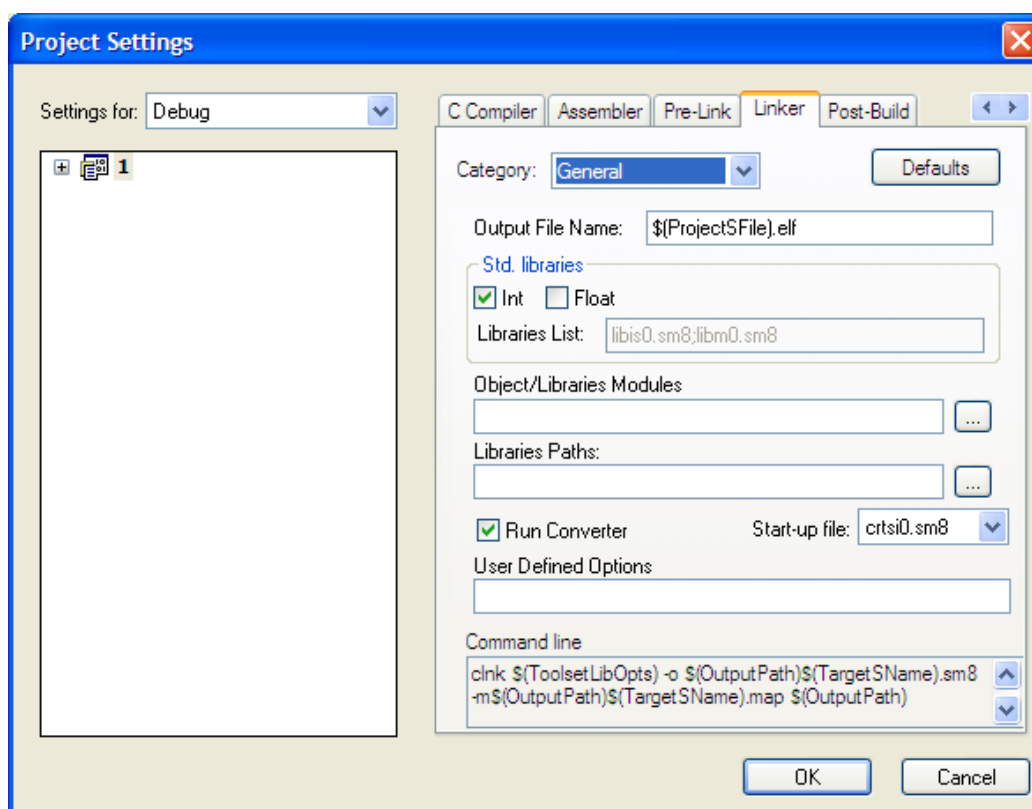
- stack short (modso) 全局变量默认 short range 类型。任何在 long range 范围的全局变量必须明确地用@near 来访问，除非通过指针访问。
- Stack Long (modsl) 全局变量默认为 long range 类型。任何在 short range 类型中的变量必须明确地用@tiny 来访问。

➤ 代码大于 64K

STM8 编译器支持两种存储器模式。函数指针默认是@far 指针（3 个字节），数据指针默认为@near 类型（2 个字节），除非用@far 明确地声明。

- Stack Short (modso) 全局变量默认 short range 类型。任何在 long range 范围的全局变量必须明确地用@near 来访问，除非通过指针来访问。
- Stack Long (modsl) 全局变量默认为 long range 类型。任何在 short range 类型中的变量必须明确地用@tiny 来访问。

“Linker”选择：

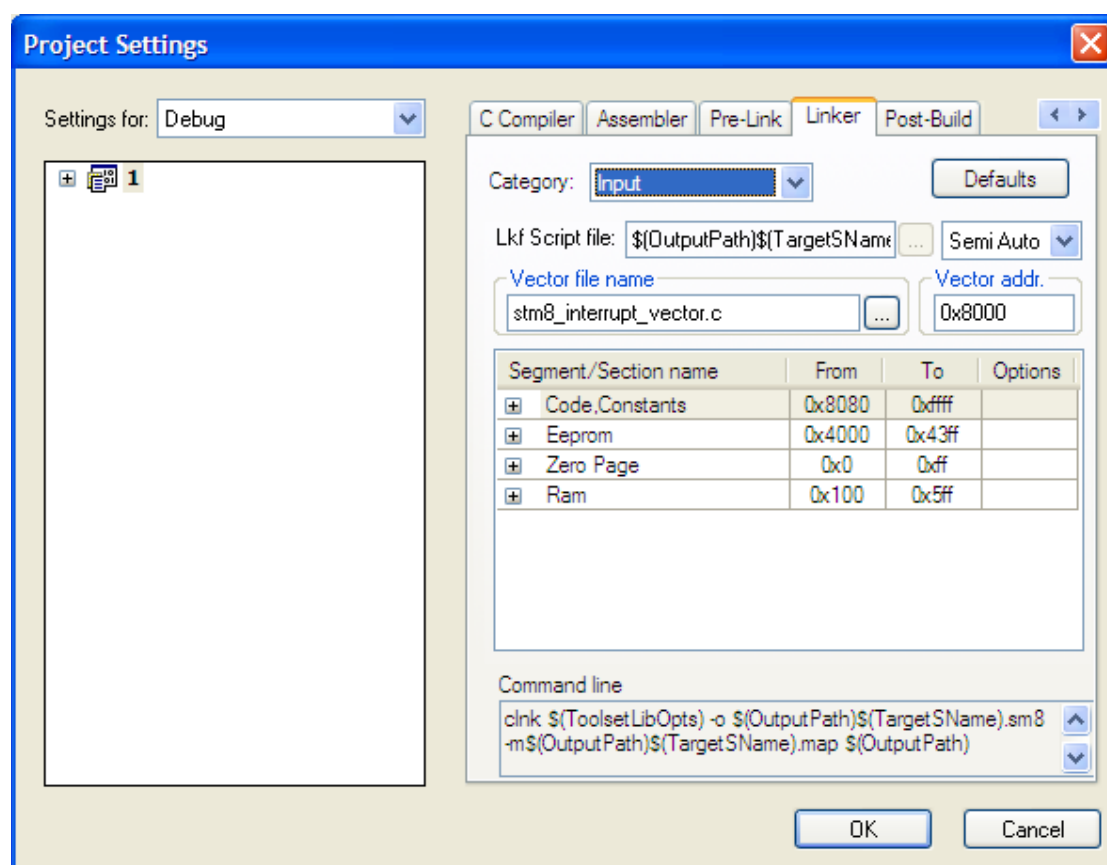
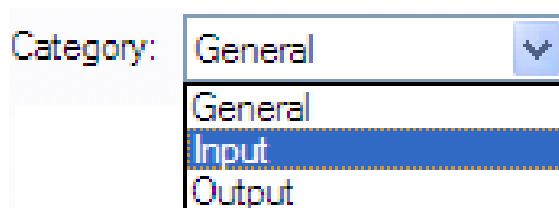


说明：

- Std. libraries：COSMIC C 编译器提供了整数和浮点运算库。可通过选择 Int 和 Float 来实现。COSMIC 会自动关联相应的函数库。
- Start-up file: 程序跳转到 main 函数前执行的启动代码。

从“Category”下拉菜单中，可以选择“Input”来查看段的配置情况、lcf 文件以及中断文件等。

可根据需要，进行自定义设置。一般地，默认的设置就可以满足要求了。



### 3.2.6.2 COSMIC C 语言段定义(Section)

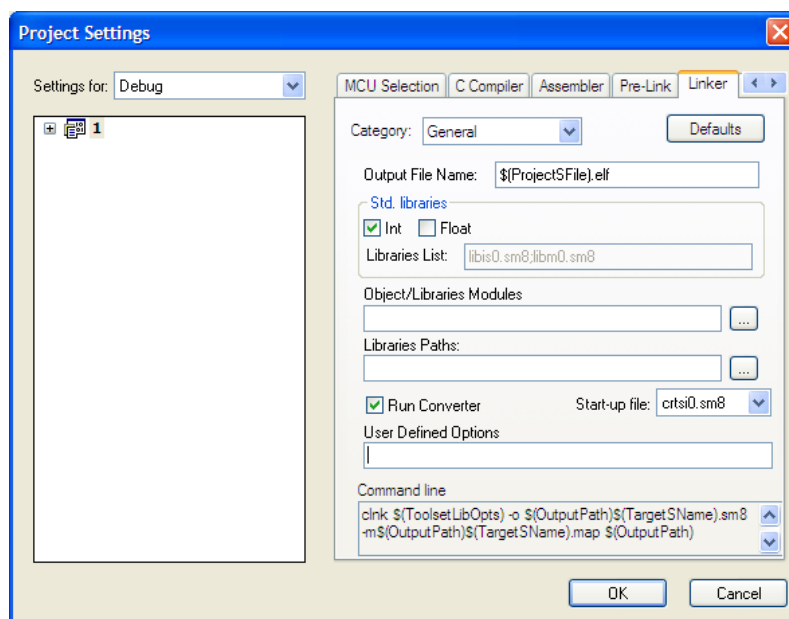
COSMIC C 编译器使用默认预定义的段输出不同部分的机器码。默认段是：

段	描述
.text	可执行代码
.const	文本字符和常数
.fconst	大常量(@far)
.data	初始化变量(@near)
.bss	未初始化变量(@near)
.bsct	零页内的初始化变量(@tiny, 默认)
.ubsct	零页内的未初始化变量(@tiny, 默认)
.fdata	大变量(@far)
.eeprom	EEPROM 内任何变量 ( @eeprom )
.bit	位变量

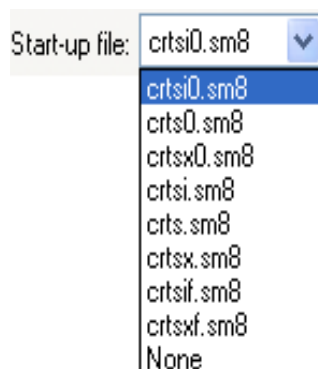
### 3.2.6.3 COSMIC C 语言启动程序

COSMIC 有一个启动程序，就是在单片机复位之后，在程序跳转至 main 函数之前，插入一段汇编代码做一些初始化的动作。其包括：

- 段的初始化 ( 如 bss )
- 拷贝 ROM 到 RAM(如果程序中需要的话，一般是指 COSMIC 内嵌的的拷贝函数)
- 堆栈指针初始化



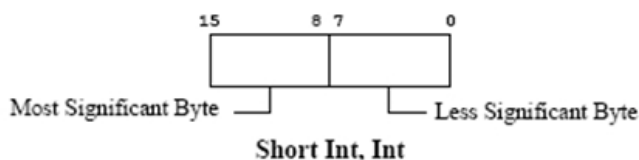
编译器提供了几个启动文件，对数据进行初始化设置。可选择的文件及初始化说明，如下所示：



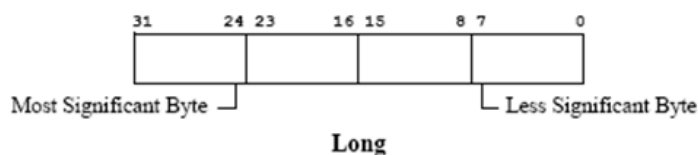
Startup	Initialize	From Table in
<a href="#">crtsi(0).s</a>	@near	@near
<a href="#">crtsx(0).s</a>	@near and @far	@near
<a href="#">crtsif(0).s</a>	@near	@far
<a href="#">crtsxf(0).s</a>	@near and @far	@far

### 数据类型

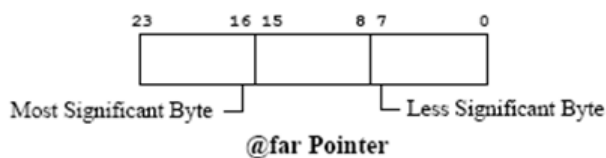
Short int 和 int 默认为 2 个字节。



Long int 默认为 4 个字节



@tiny 指针 ( short range ) 默认为 1 个字节；@near 指针 ( long range ) 默认为 2 个字节；@far 指针默认为 3 个字节



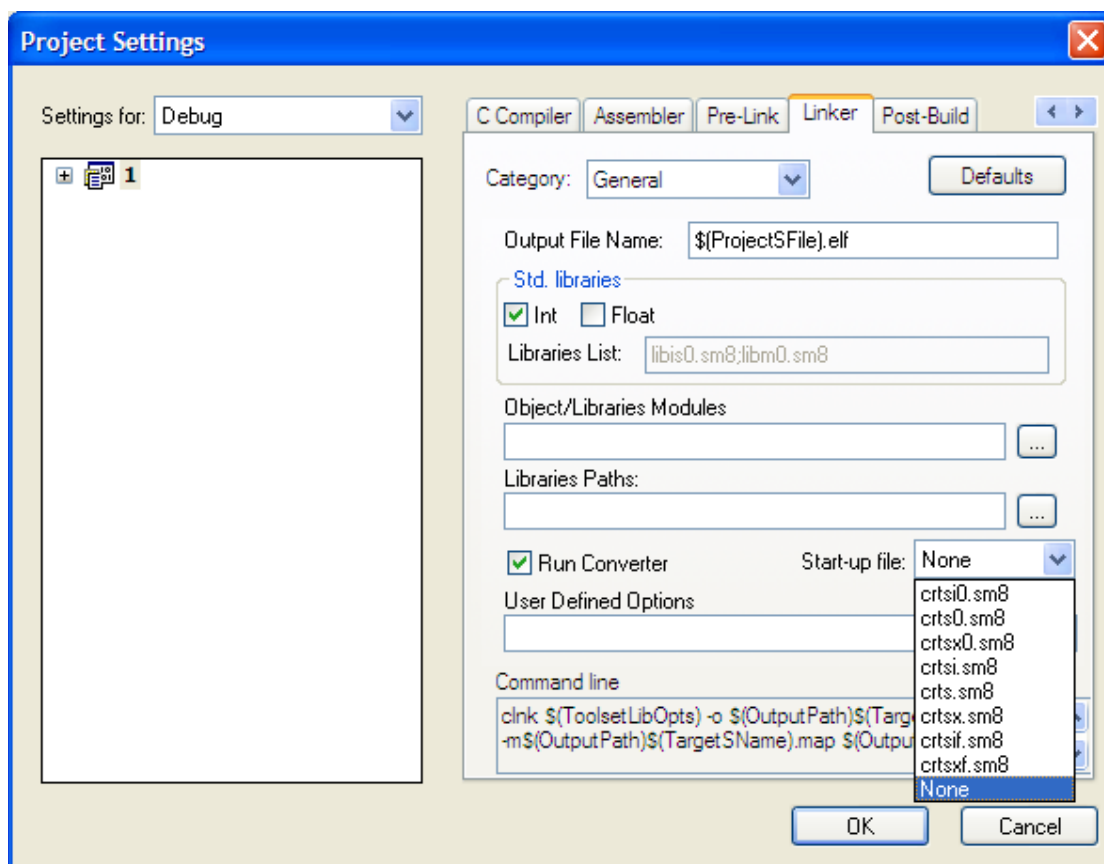


下面是 crtsio.sm8 反汇编之后的代码：

0x8084 <__stext+1>	0x0FFF	CLR	(0xff,SP)	CLR	(0xff,SP)
0x8086 <__stext+3>	0x94	LDW	SP,X	LDW	SP,X
0x8087 <__stext+4>	0x90CE8080	LDW	Y,0x8080	LDW	Y,__idesc__
0x808b <__stext+8>	0xAE8082	LDW	X,#0x8082	LDW	X,#0x8082
0x808e <__stext+11>	0xF6	LD	A,(X)	LD	A,(X)
0x808f <__stext+12>	0x2720	JREQ	0x80b1	JREQ	0x80b1
0x8091 <__stext+14>	0xA560	BCP	A,#0x60	BCP	A,#0x60
0x8093 <__stext+16>	0x2717	JREQ	0x80ac	JREQ	0x80ac
0x8095 <__stext+18>	0xBF03	LDW	0x03,X	LDW	c_x,X
0x8097 <__stext+20>	0xEE03	LDW	X,(0x03,X)	LDW	X,(0x03,X)
0x8099 <__stext+22>	0xBF06	LDW	0x06,X	LDW	c_y,X
0x809b <__stext+24>	0xBE03	LDW	X,0x03	LDW	X,c_x
0x809d <__stext+26>	0xEE01	LDW	X,(0x01,X)	LDW	X,(0x01,X)
0x809f <__stext+28>	0x90F6	LD	A,(Y)	LD	A,(Y)
0x80a1 <__stext+30>	0xF7	LD	(X),A	LD	(X),A
0x80a2 <__stext+31>	0x5C	INCW	X	INCW	X
0x80a3 <__stext+32>	0x905C	INCW	Y	INCW	Y
0x80a5 <__stext+34>	0x90B306	CPW	Y,0x06	CPW	Y,c_y
0x80a8 <__stext+37>	0x26F5	JRNE	0x809f	JRNE	0x809f
0x80aa <__stext+39>	0xBE03	LDW	X,0x03	LDW	X,c_x
0x80ac <__stext+41>	0x1C0005	ADDW	X,#0x0005	ADDW	X,#0x0005
0x80af <__stext+44>	0x20DD	JRT	0x808e	JRT	0x808e
0x80b1 <__stext+46>	0xAE0000	LDW	X,#0x0000	LDW	X,#0x0000
0x80b4 <__stext+49>	0x2002	JRT	0x80b8	JRT	0x80b8
0x80b6 <__stext+51>	0xF7	LD	(X),A	LD	(X),A
0x80b7 <__stext+52>	0x5C	INCW	X	INCW	X
0x80b8 <__stext+53>	0xA30009	CPW	X,#0x0009	CPW	X,#0x0009
0x80bb <__stext+56>	0x26F9	JRNE	0x80b6	JRNE	0x80b6
0x80bd <__stext+58>	0xAE0100	LDW	X,#0x0100	LDW	X,#0x0100
0x80c0 <__stext+61>	0x2002	JRT	0x80c4	JRT	0x80c4
0x80c2 <__stext+63>	0xF7	LD	(X),A	LD	(X),A
0x80c3 <__stext+64>	0x5C	INCW	X	INCW	X
0x80c4 <__stext+65>	0xA30100	CPW	X,#0x0100	CPW	X,#0x0100
0x80c7 <__stext+68>	0x26F9	JRNE	0x80c2	JRNE	0x80c2
0x80c9 <__stext+70>	0xCD80CF	CALL	0x80cf	CALL	main
0x80cc <__exit>	0x20FE	JRT	0x80cc	JRT	_exit

若不想让 COSMIC C 编译器来进行初始化，如何来设置呢？也就是说，在单片机复位后，直接跳转至 main 处执行。在 main 开始处，按照自己的设计，做一些变量和外设等的初始化动作。

在 STVD 开发环境中，启动文件的设置如图所示：（project->Settings...）



STM8 复位后直接跳转至 main 处执行,可按如下步骤修改：

- 从 Start-up file 处的下来菜单中，选择 “None”
- 修改复位函数：
  - 将原来的 `{0x82, (interrupt_handler_t)_stext}, /* reset */`
  - 修改为 `{0x82, (interrupt_handler_t)main}, /* reset */`
- 修改外部函数声明：
  - 将原来的 `extern void _stext(); /* startup routine */`
  - 修改为 `extern void main(); /* startup routine */`

**注意：确保在变量使用前，都做了正确的初始化。**

### 3.2.6.4 COSMIC C 语言中嵌入汇编指令

COSMIC C 编译器提供两种方法嵌入汇编指令。

第一种方法是#asm 和#endasm 嵌入汇编指令块

第二种方法是嵌入行汇编。单独一行汇编指令。

第一种方法语法：

```
#asm //开始汇编指令块
;...
#endasm //结束汇编指令块
```

第二种方法语法：

```
_asm("嵌入的汇编代码",符合 C 语言规则的参数...);
```

例如，执行单条指令

```
_asm("ld _mya,a");
```

若在一行内执行多条指令：

```
_asm("push a\n ld a,88\n ld _mya,a\n inc a\n pop a\n call _subroutine\n");
```

下面是一个嵌入汇编的例子：

```
#include "stm8s207c_s.h"
unsigned char i,mya;
void subroutine(void){}
main()
{
    mya=0x22;
    #asm // #asm 要顶格书写
        push a
        ld a,_mya
        inc a
        ld _mya,a
        call _subroutine
        pop a
    #endasm

    _asm("push a\n ld a,88\n ld _mya,a\n pop a\n call _subroutine\n");
    while (1) {i = mya;}
}
```

### 3.2.6.5 COSMIC C 语言位变量

COSMIC C 编译器使用 `_Bool` 类型定义位变量。布尔变量可用做结构或联合类型，连续的布尔变量放在一个字节中。

布尔变量定义，例子如下：

```
_Bool   in_range;
_Bool   p_valid;
char    *ptr;

in_range = (value >= 10) && (value <= 20);
p_valid = ptr;                               /* p_valid is true if
ptr not 0 */
if (p_valid && in_range)
*ptr = value;
```

### 3.2.6.6 COSMIC C 语言常量定义

常量定义例子：

```
char * const x;                               /* const pointer to char */
int * volatile y;                             /* volatile pointer to int */
const float pi = 355.0 / 113.0; /* pi is never changed */
```

定义一个常量表

```
const unsigned char constring[]
={0x2C, 0x27, 0x23, 0x1F, 0x1D, 0x1A, 0x17, 0x16, 0x13, 0x11, 0x0F, 0x0E, 0x00
, 0x0D, 0x09, 0x15 };
unsigned char i;

i = constring[3]; // i = 0x1F
```

### 3.2.6.7 COSMIC C 语言在 RAM 中运行程序

可使用 COSMIC 中的函数 `_fctcpy` 将 FLASH 中的代码拷贝到 RAM 中，并运行。`_fctcpy` 从 FLASH 中拷贝一段可移动代码段到 RAM 中。`_fctcpy` 寻找 linker 定义的描述符（此描述符是在 RAM 中定义段的第一个字符）。本例子在 RAM 中定义了一个段 `.CODE_IN_RAM`。所以地一个字符是 'C'。

需要在程序中应用 `int _fctcpy ( char name ) ;`

在 Ram 中创建一个 '**.CODE\_IN\_RAM**' 段。并在 Option 中输入 '**-ic**'

'**.CODE\_IN\_RAM**' 表示在 RAM 中定义一个 `CODE_IN_RAM` 的段。程序可从此起始地址运行。`_fctcpy` 函数使用此段的名字的第一个描述符 'C'。

'**-ic**' 表示标记这个段为可移动的段。可将 FLASH 中的代码移动到此。

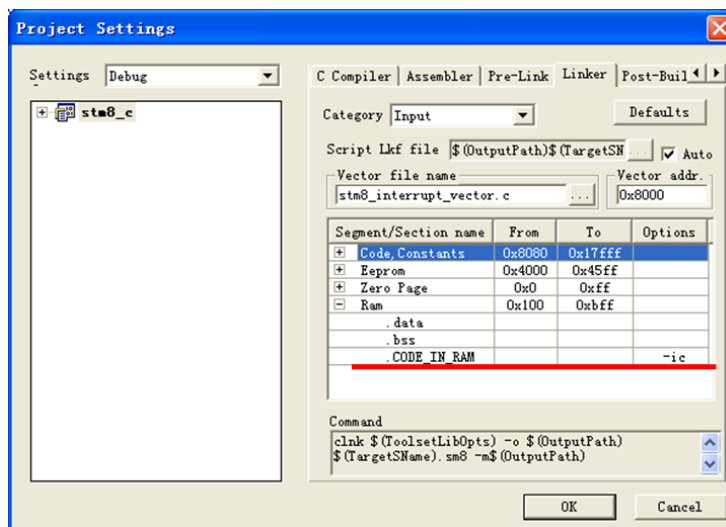
在程序代码中完成函数：

```
#pragma section (CODE_IN_RAM)
functions() {
    var1 = 0x99;
    var2 = 0x88;
    var3 = 0x77;
}

routine() {
    var1++;
    var2++;
    var3++;
}

#pragma section ()
```

打开 project->settings...->Linker 如下图所示配置：



例程如下：

```
unsigned char var1,var2,var3;
int _fctcpy(char name);

#pragma section (CODE_IN_RAM)
functions()
{
    var1 = 0x99;
    var2 = 0x88;
    var3 = 0x77;
}
routine()
{
    var1++;
    var2++;
    var3++;
}
#pragma section ()

main()
{
    _fctcpy('C');          //拷贝 pragma section(CODE_IN_RAM) 中的函数到
RAM
    functions();          // 调用RAM中的 functions 函数
    routine();            // 调用RAM 中的 routine 函数

    while (1)
    {
        _asm("nop");
        _asm("nop");
        _asm("nop");
        _asm("nop");
    }
}
```

#### 3.2.6.8 COSMIC C 语言产生 HEX 文件

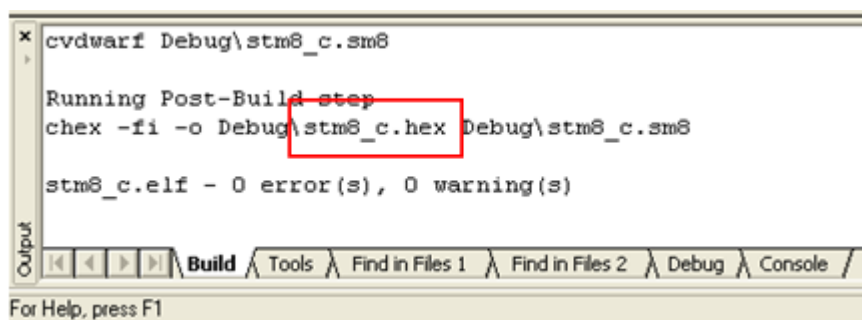
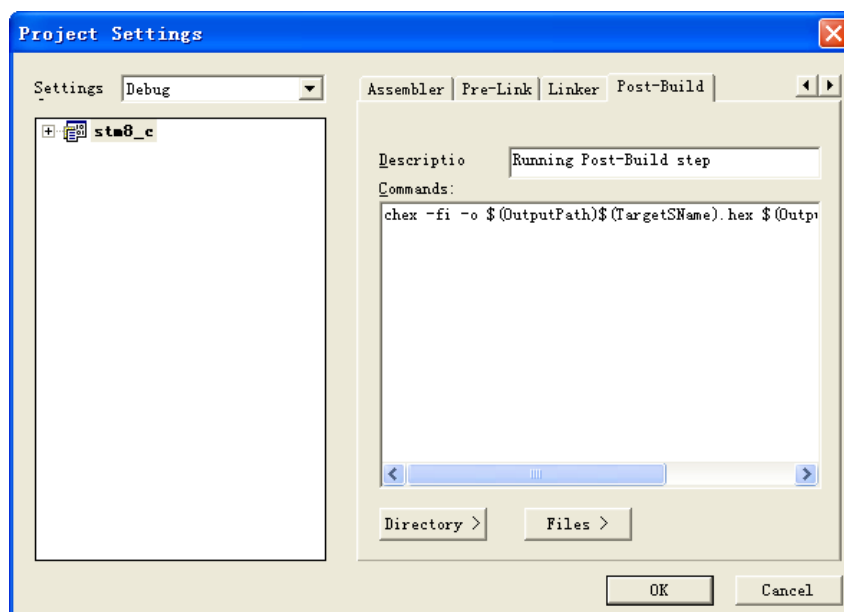
设置 Project-> Settings...->Post-Build

原来设置：

```
chex -o $(OutputPath)$(TargetSName).s19 $(OutputPath)$(TargetSName).sm8
```

设置为：

```
chex -fi -o $(OutputPath)$(TargetSName).hex $(OutputPath)$(TargetSName).sm8
```



## 3.3 使用 IAR 创建 C 语言项目

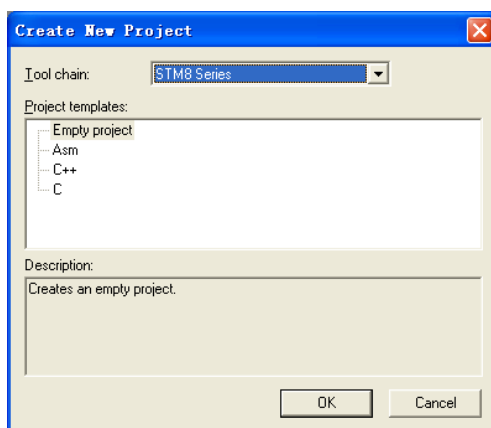
### 3.3.1 新建

#### a) 创建一个 Workspace

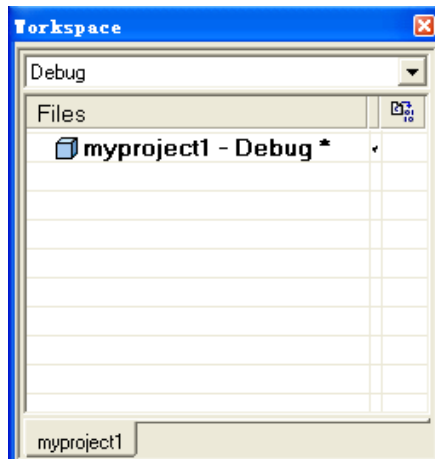
首先，创建一个 workspace。选择菜单 File>New>Workspace

#### b) 创建一个 Project

- 1) 创建一个新的工程，选择 Project>Create New Project，创建新工程的对话框，如下图所示：



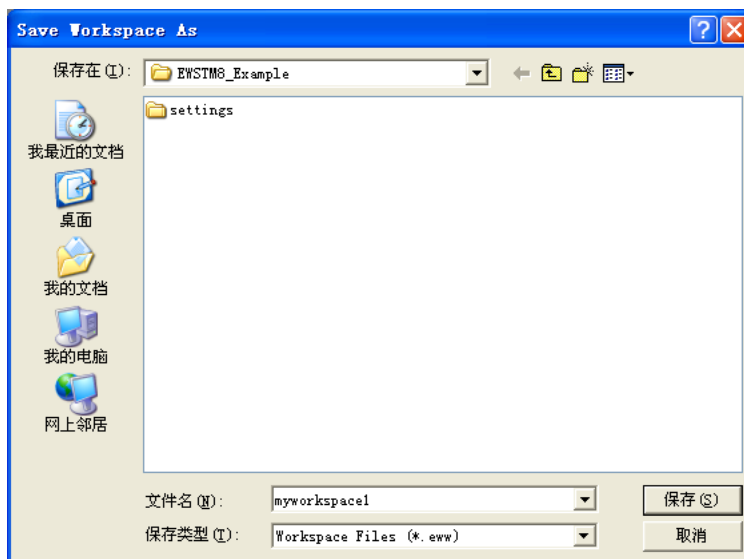
- 2) Tool chain 默认是 STM8 Series。无需再选择。
- 3) 在 Project templates 中选择 “Empty project”
- 4) 弹出 Save As 对话框，选择 project 保存的路径，并输入 project 的名字  
在 Workspace 窗口中，显示如下





有 Debug 和 Release 两个配置。默认配置是 Debug。

- 5) 在添加文件到工程之前，先保存 workspace。选择 File>Save Workspace, 指定要保存的路径，并输入 workspace 的名字。



一个 workspace 文件的扩展名是 eww. 此文件列出了添加到 workspace 中的所有的 project。相关当前会话信息，比如 windows 的保存路径和断点保存于 projects\settings 目录下。

#### c) 添加文件到工程

可选择 Project>Add Files 选择要添加的 C 文件。找到相应 MCU 型号的头文件，拷贝到当前的工程目录中。默认地，头文件在 IAR 安装目录 inc 下：  
C:\Program Files\IAR Systems\Embedded Workbench 6.0 Kickstart\stm8\inc。

本例中使用了 iostm8s105s6.h。编写一个简单的 C 程序：

## 源文件 main.c

```
/* Includes -----*/
#include "iostm8s105s6.h"
/* Global variables -----*/
unsigned int ms_count;

void CLK_Configuration(void)
{
    CLK_CKDIVR = 0x00;    /* fCPU=fMASTER = 16MHz */
}

void GPIO_Init(void)
{
    PD_DDR |= 0x0D;      /* Output. */
    PD_CR1 |= 0x0D;      /* PushPull. */
    PD_CR2 = 0x00;      /* Output speed up to 2MHz. */
}

void TIM2_Init(void)
{
    TIM2_PSCR = 0x00;    /* Configure TIM2 prescaler =16. */
    TIM2_ARRH = 0xC1;    /* Configure TIM2 period. */
    TIM2_ARRL = 0x7F;    /* Configure TIM2 period. */
    TIM2_CNTRH = 0xC1;
    TIM2_CNTRL = 0x7F;
    TIM2_CR1 |= 0x81;    /* Enable TIM2. */
    TIM2_IER |= 0x01;    /* Enable TIM2 OVR interrupt. */
}

main()
{
    asm("sim"); /* disable interrupts */
    ms_count = 0;
    CLK_Configuration();
    GPIO_Init();
    TIM2_Init();
    asm("rim"); /* enable interrupts */
    while (1)
    {
        asm("nop");
        asm("nop");
        asm("nop");
    }
}
```

**中断**

声明一个中断向量，说明如下：

```
#pragma vector=0x02
__interrupt void interrupt_handler(void)
```

说明:

**#pragma vector** : 是 IAR 中断向量指令

**=0x02** : 其数字代表中断向量编号。STM8 的地址是从 0x00800 开始，IAR 的中断编号从 0 开始。中断向量号依次按照中断地址递增。如：

复位向量是 0x008000，中断向量号是 0x00

TRAP 的中断地址是 0x008004 中断向量号是 0x01

SPI 中断号是 0x0C;详细见下表

**\_\_interrupt void interrupt\_handler(void)**: 是声明一个中断函数，注意 interrupt 是两个下划线，interrupt\_handler 是中断函数名字，可自己定义。

Interrupt vector mapping STM8S105xx

---

**7 Interrupt vector mapping**

Table 11. Interrupt mapping

IRQ no.	Source block	Description	Wakeup from halt mode	Wakeup from active-halt mode	Vector address
	RESET	Reset	Yes	Yes	0x00 8000
	TRAP	Software interrupt	-	-	0x00 8004
0	TLI	External top level interrupt	-	-	0x00 8008
1	AWU	Auto wake up from halt	-	Yes	0x00 800C
2	CLK	Clock controller	-	-	0x00 8010
3	EXTI0	Port A external interrupts	Yes <sup>(1)</sup>	Yes <sup>(1)</sup>	0x00 8014
4	EXTI1	Port B external interrupts	Yes	Yes	0x00 8018
5	EXTI2	Port C external interrupts	Yes	Yes	0x00 801C
6	EXTI3	Port D external interrupts	Yes	Yes	0x00 8020
7	EXTI4	Port E external interrupts	Yes	Yes	0x00 8024
8	EXTI4	Port E external interrupts	Yes	Yes	0x00 8028
9	Reserved	Reserved	-	-	0x00 802C
10	SPI	End of transfer	Yes	Yes	0x00 8030
11	TIM1	TIM1 update/overflow/underflow/trigger/break	-	-	0x00 8034
12	TIM1	TIM1 capture/compare	-	-	0x00 8038
13	TIM2	TIM2 update /overflow	-	-	0x00 803C
14	TIM2	TIM2 capture/compare	-	-	0x00 8040
15	TIM3	Update/overflow	-	-	0x00 8044
16	TIM3	Capture/compare	-	-	0x00 8048
17		Reserved	-	-	0x00 804C
18		Reserved	-	-	0x00 8050
19	I <sup>2</sup> C	I <sup>2</sup> C interrupt	Yes	Yes	0x00 8054
20	UART2	Tx complete	-	-	0x00 8058
21	UART2	Receive register DATA FULL	-	-	0x00 805C
22	ADC1	ADC1 end of conversion/analog watchdog interrupt	-	-	0x00 8060
23	TIM4	TIM4 update/overflow	-	-	0x00 8064
24	Flash	EOP/WR_PG_DIS	-	-	0x00 8068
Reserved					0x00 806C to 0x00 807C

```
#pragma vector=0x09
__interrupt void EXTI4_IRQHandler(void)
```

0x01

0x09

0x1A

## 一个简单的中断函数 stm8s105\_interrupt.c

```
/* Includes ----- */
#include "iostm8s105s6.h"

/* External variables ----- */
extern unsigned int ms_count;

/* Defines an interrupt handler for TIM2_UPDATE vector. */
#pragma vector=15
__interrupt void TIM2_UPDATE_IRQHandler(void)
{
    TIM2_SR1 &=~(0x01);

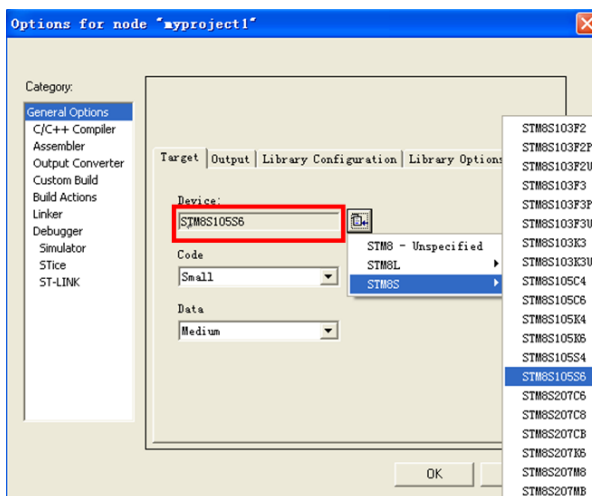
    ms_count++;

    if(ms_count == 500) /* 0.5 秒中断 */
    {
        PD_ODR ^= 0x01;
        ms_count = 0;
    }
}
```

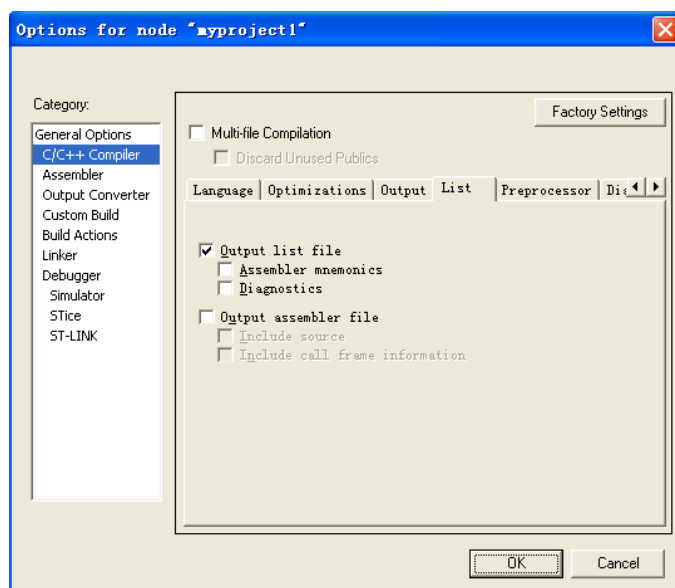
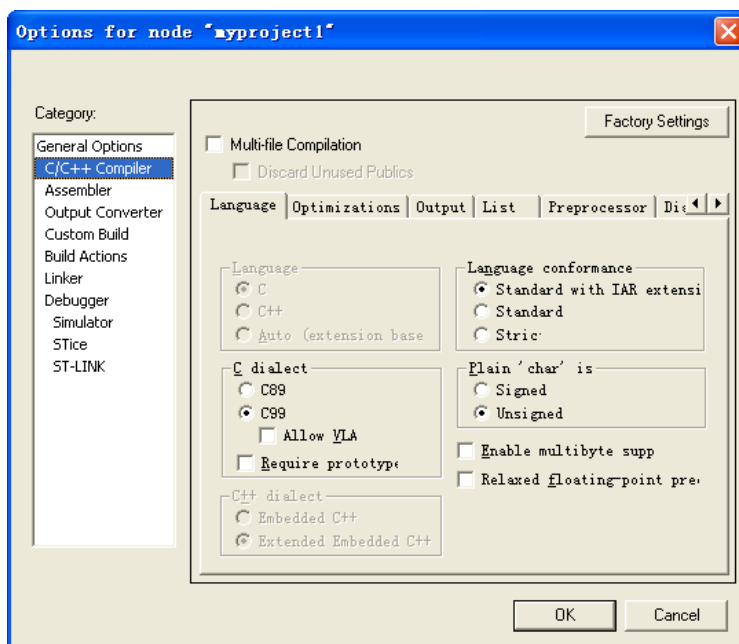
## d) 工程选项配置

选择 Project>Options，或者在 Workspace 窗口，选中 project 名字，右击选择选择“Options...”

- 1) 在 Category 中，选择“General Options”，如图：在 Target 的 Device 中，选择相应的 MCU 型号。其他选择默认。

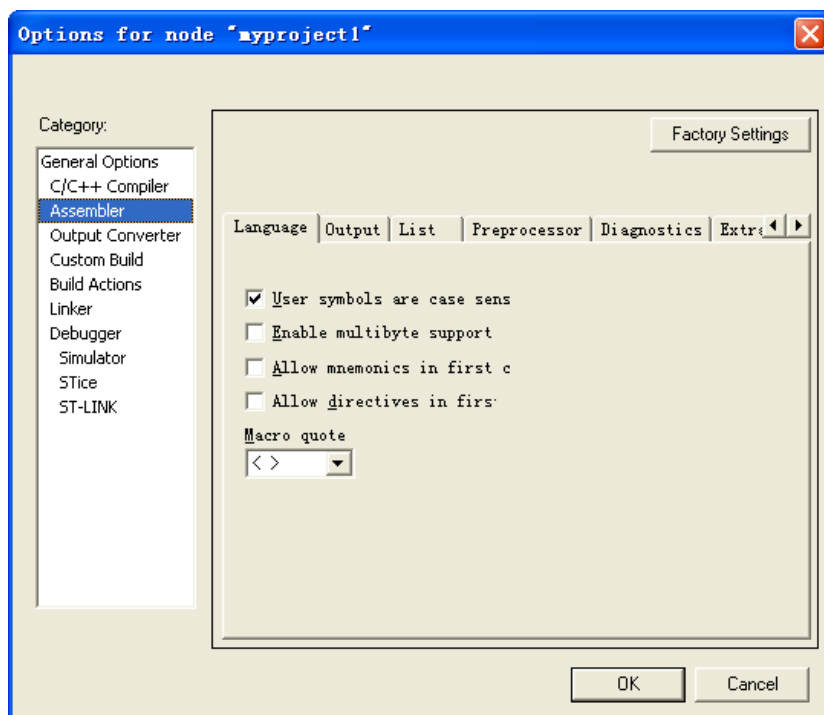


2) 在 Category 中, 选择 C/C++ Compiler, 显示 compiler 选项页

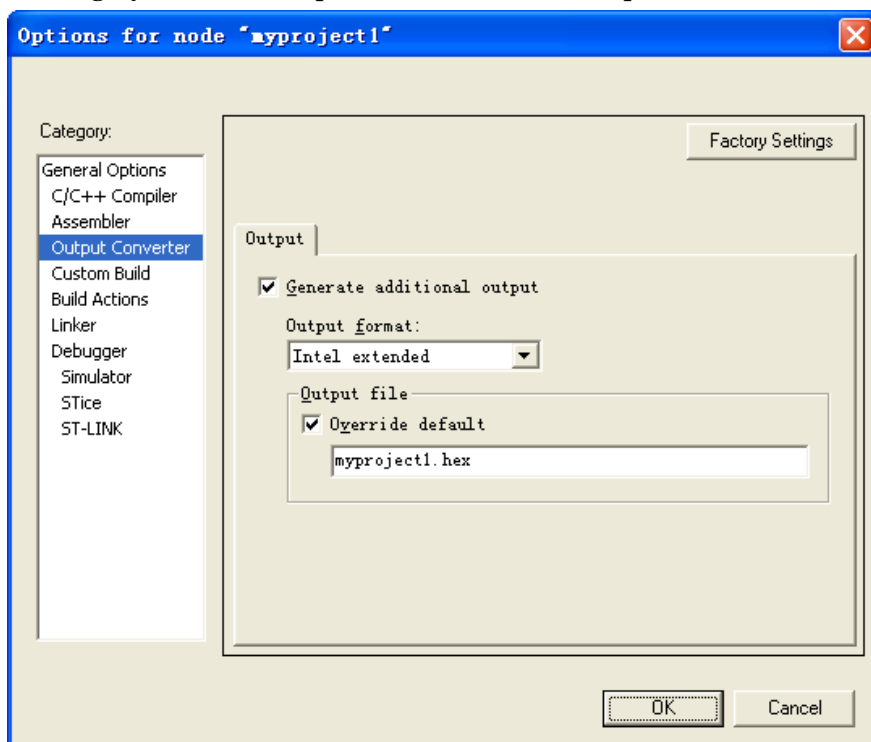


选择 Output list file, 输出列表文件

3) 在 Category 中, 选择 Assembler, 显示 Assembler 选项页

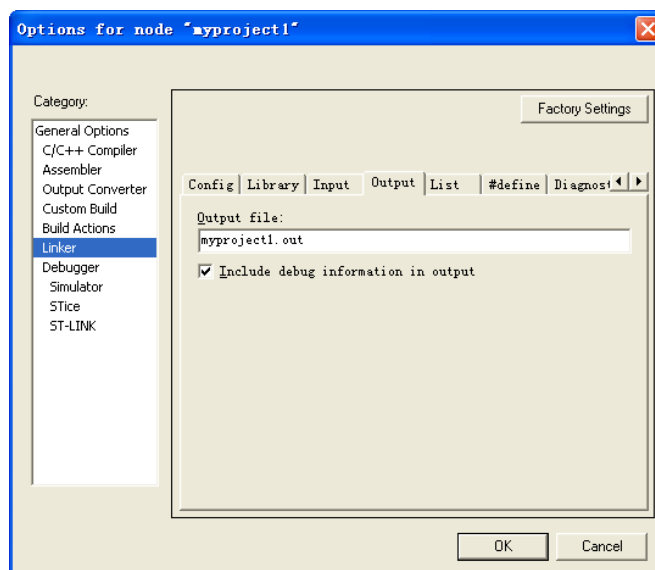


- 4) 在 Category 中，选择 Output Converter，显示 Output Converter 选项页

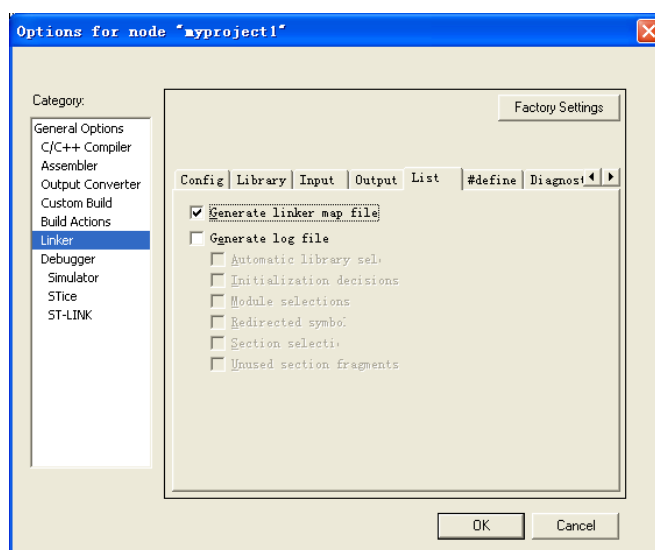


在 Debug\Exe 目录中，产生目标文件。IAR 默认的是.out 文件。此界面可选择不同的格式目标文件。本例选择 Intel extended 格式的目标文件。

5) 在 Category 中，选择 Linker，显示 Linker 选项页

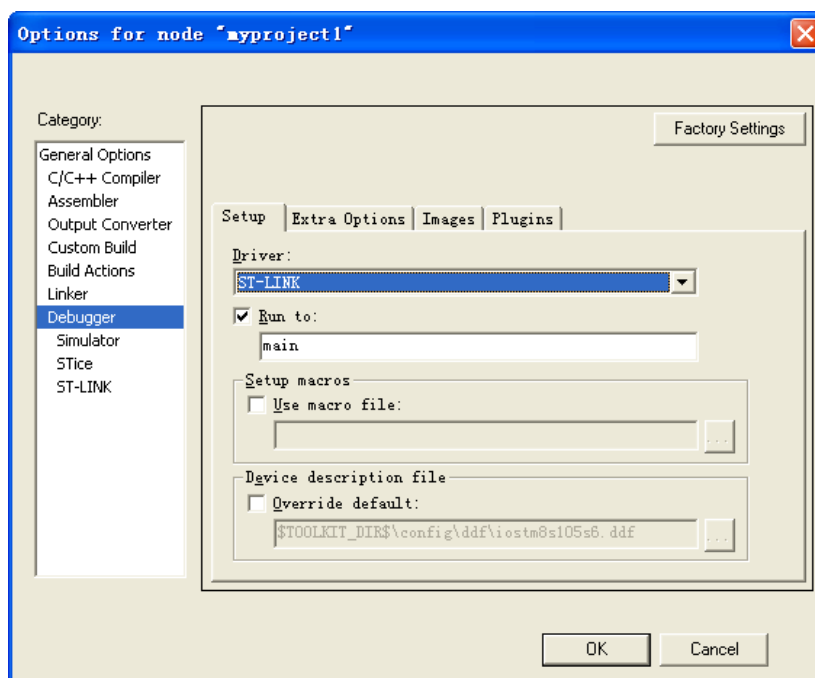


默认输出文件是：工程名字.out



选中 Generate linker map file,输出工程的 map 文件。

6) 在 Category 中，选择 Debugger，显示 Debugger 选项页

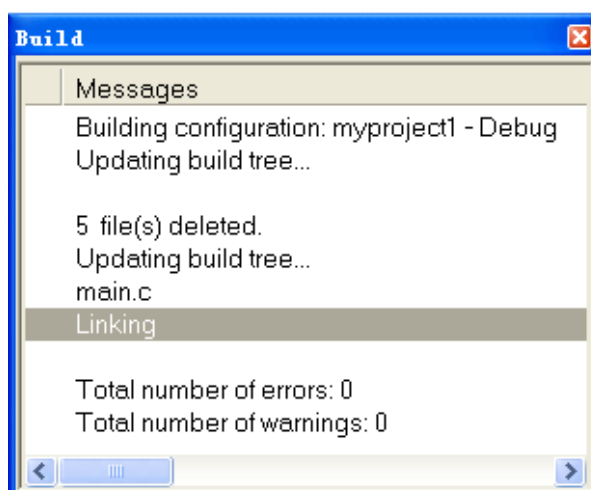


在 Driver 中，选择 ST-LINK。IAR 目前还支持软件仿真模拟和 STice 工具

### 3.3.2 编译

通过 Project>Compile 或者 Project>Rebuild All 来编译某个源文件或者全部重新编译。

编译创建信息显示窗口如下：

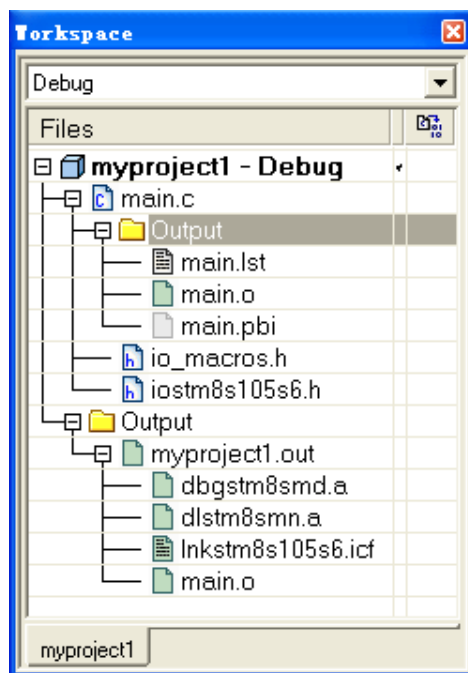


IDE 将创建 List, Obj 和 Exe 目录。



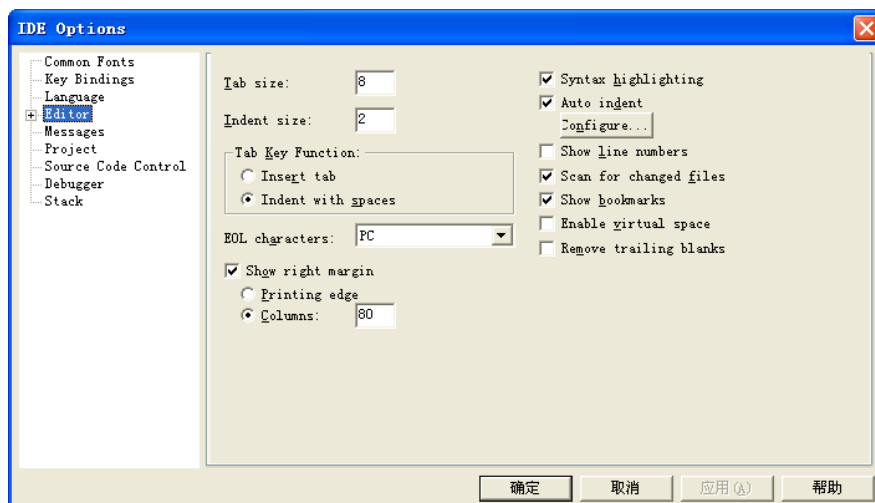
- List 目录是列表文件的目录。其扩展名为.lst
- Objm 目录是目标文件目录。ILINK 连接器的扩展名为.o
- Exe 目录是可执行文件目录。

可通过 workspace 中的 output 文件夹，来查看所有输出的文件。



### 查看列表文件

List 文件可查看汇编代码和每个段的二进制代码。还显示变量如何分配。List 文件的结尾现实了堆栈，代码和数据存储器的大小。可选择 Tools>Options 来配置更新改变的文件



若要看代码的大小，可找到工程目录下的 \Debug\List，在 list 下有.map 文件。用记事本打开.map 文件，在.map 的末尾，可查看程序代码大小。

如本例中的.map 文件。


```
...
[1] = D:\Tutorial\EWSTM8S\Debug\Obj
[2] = command line
[3] = dbgstm8smd.a
[4] = dlstm8smn.a

    294 bytes of readonly code memory
    132 bytes of readonly data memory
    274 bytes of readwrite data memory

Errors: none
Warnings: none
...
```

本例的程序代码是：294 个字节

### 3.3.3 调试

选择 Project>Download and Debug. 进入调试状态。或者选择工具栏 

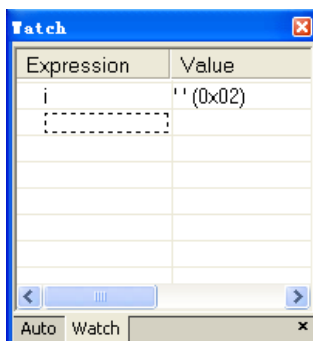
#### 使用 Auto 窗口

在调试状态下，选择 View>Auto 打开 Auto 窗口。Auto 窗口自动地显示当前的修改变量。如图：



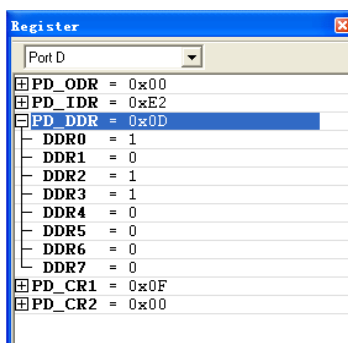
#### 观察窗口

选择 View>Watch，打开观察窗口，如图：



#### 寄存器窗口

选择 View>Register，打开寄存器窗口。

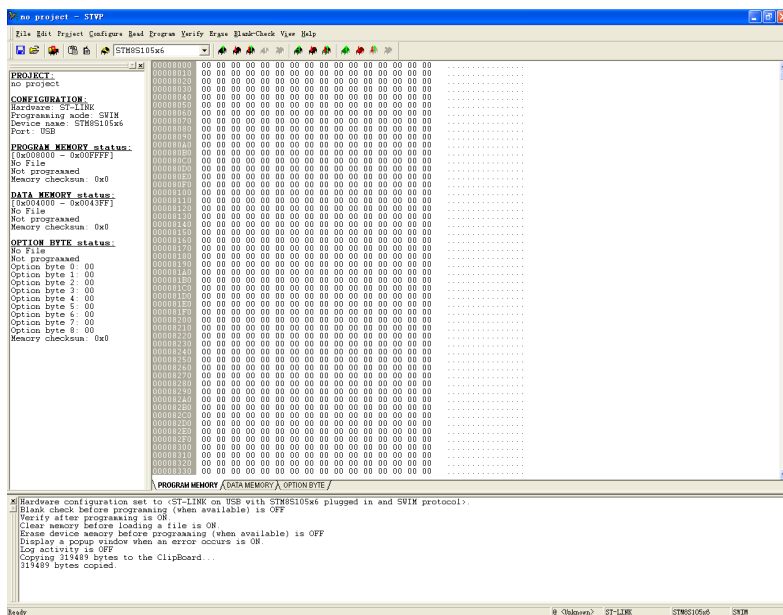


### 3.3.4 烧录

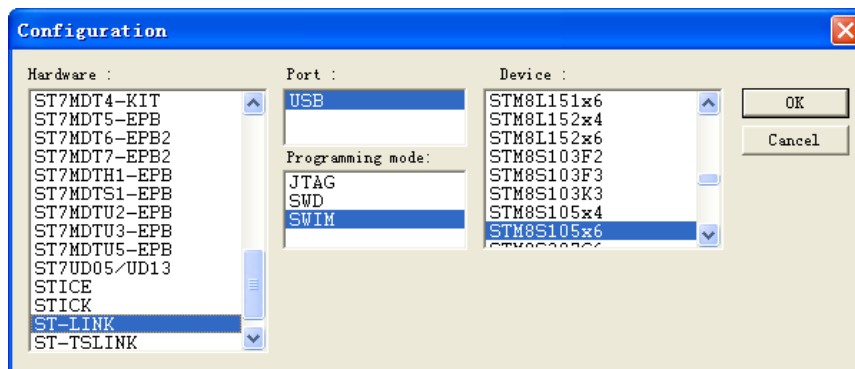
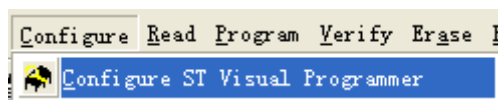
IAR 在调试时，可直接把代码下载到 STM8 里。

也可以通过 STVP 来烧录。在 IAR 工程当前目录下，有“Debug\Exe”，Exe 下面有个 .hex 文件。这个就是目标文件。在 STVP 中调入 HEX 文件。配置好 OPTION BYTE 进行程序的烧录。

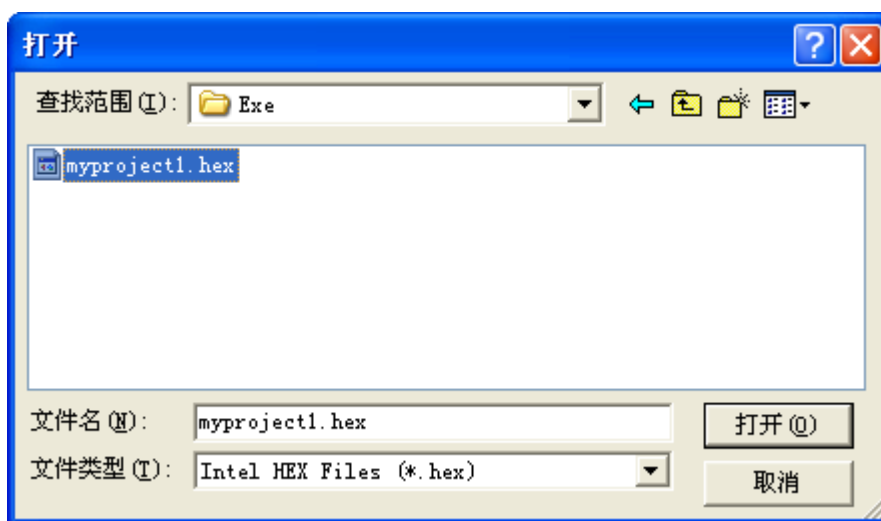
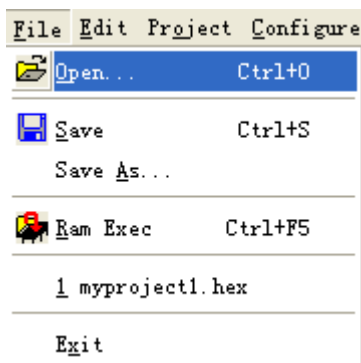
运行 STVP 如图所示：



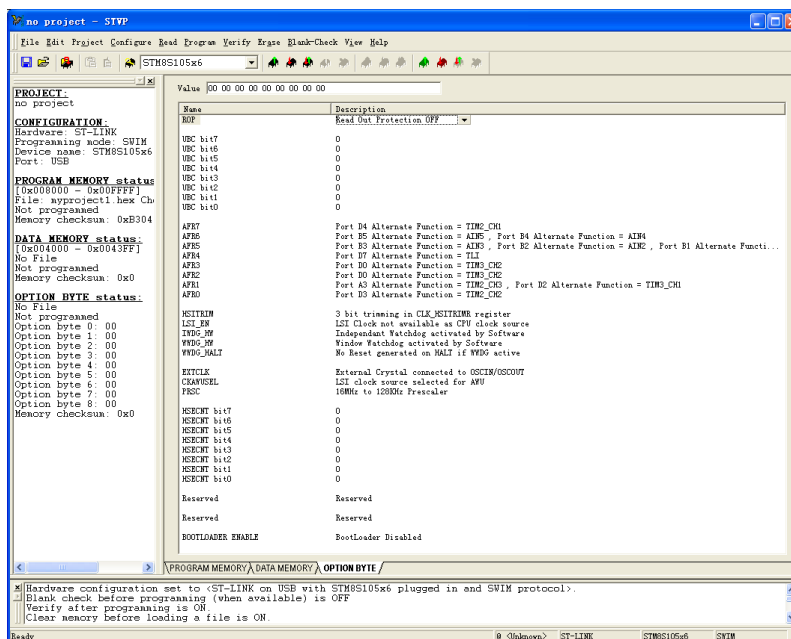
选择编程器，“Configure>Configure ST Visual Programmer”



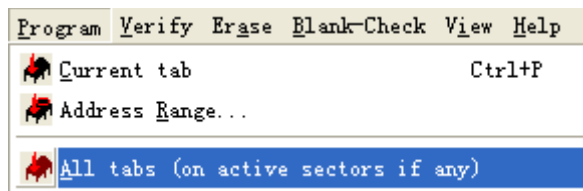
打开 HEX 文件。 “File>Open...”



配置 OPTION BYTE



选择 “Program>All tabs(on action sectors if any)”。此选项可将 “PROGRAMM MEMORY,DATA MEMORY 和 OPTION BYTE”一起烧录到 STM8 里面去。



## 3.3.5 IAR C 语言相关说明

### 3.3.5.1 嵌入汇编语言

```
asm("nop");
```

# 4 STM8 程序设计

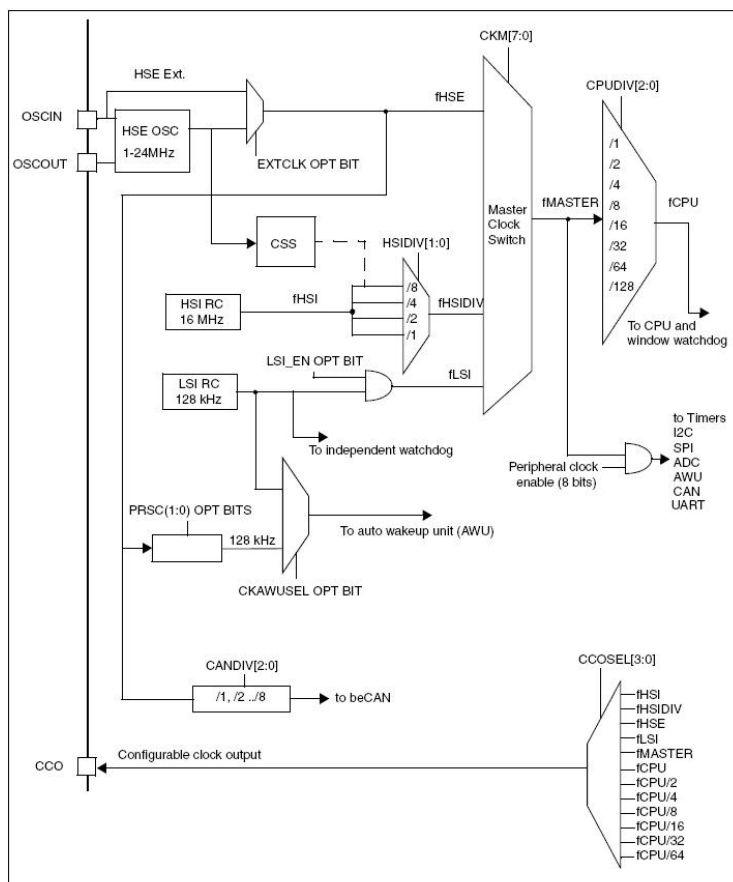
## 本章提示：

- ◇ 在开始写代码之前，需要准备两份资料
  - ✓ 一份是 datasheet：有引脚图及引脚功能说明、OPTION BYTE 说明
  - ✓ 另一份是参考手册：详细描述了各个外设的功能
- ◇ 本章介绍了 STM8 的时钟配置、GPIO、ADC、TIMER2、EEPROM 等的应用

## 4.1 STM8S 应用例程

### 4.1.1 时钟控制

STM8 的钟控制器功能强大而且灵活易。现以 STM8S 单片机的时钟树为例，时钟树如下图所示：



1. Legend: HSE = High speed external clock signal; HSI = High speed internal clock signal; LSI = Low Speed internal clock signal.

从时钟树来看，fCPU 的时钟来源是由 CPUDIV 分频 fMASTER 时钟而获得的；fMASTER 的时钟源有三个可以选择：fHSE、fHSIDIV、fLSI。fHSE 来自于外部的时钟；fHSIDIV 来自于内部 16MHz RC 的时钟源；fLSI 来自于内部 128KHz RC 时钟源。

在开始设计前，先确定时钟为外部或内部。主频要跑多少频率的。建议使用内部 16MHz 的 RC 振荡器作为主时钟源。

STM8 时钟控制部分可以在程序的任何时候，都可以切换。灵活地控制 CPU 的功耗与运行。

配置时钟控制 fCPU=fMASTER = 16MHz 的例子：

```
/* fCPU=fMASTER = 16MHz */
CLK_CKDIVR = 0x00;    // HSIDIV[1:0]=0x00, CPUDIV[2:0]=0x00
```

配置时钟控制 fCPU=fMASTER = 2MHz 的例子：

```
/* fCPU=fMASTER = 2MHz */
CLK_CKDIVR = 0x18;    // HSIDIV[1:0]=0x11, CPUDIV[2:0]=0x00
                        // HSIDIV[1:0]= 11: fHSI = fHSI RC 输出/8
                        // CPUDIV[2:0]= 000: fCPU = fMASTER
```

### 4.1.2 GPIO

复用功能的映射是通过选项字节控制的。但是在同一时刻仅有一个复用功能可以映射到引脚上。

每个端口都分配有一个输出数据寄存器，一个输入引脚寄存器，一个数据方向寄存器，一个选择寄存器，和一个配置寄存器。一个 I/O 口工作在输入还是输出是取决于该口的数据方向寄存器的状态

GPIO 配置表如下所示：



Mode	DDR bit	CR1 bit	CR2 bit	Function	Pull-up	P-buffer	Diodes	
							to V <sub>DD</sub>	to V <sub>SS</sub>
Input	0	0	0	Floating without interrupt	Off	Off	On	On
	0	1	0	Pull-up without interrupt	On			
	0	0	1	Floating with interrupt	Off			
	0	1	1	Pull-up with interrupt	On			
Output	1	0	0	Open drain output	Off	Off	On	On
	1	1	0	Push pull output		On		
	1	x	1	Output speed limited to 10 MHz		Depends on CR1 bit		
	1	x	x	True open drain (on specific pins)	Not Implemented	Not implemented (see note)		

例子：

```

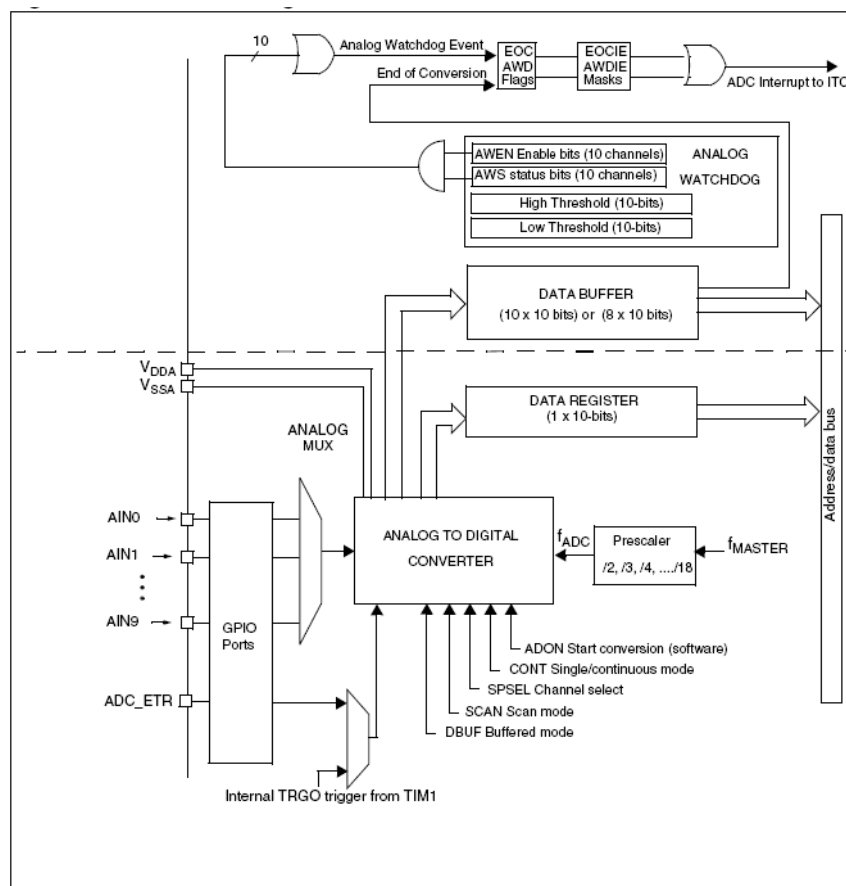
/* PD0 控制一个 LED 灯的亮/灭 */
PD_DDR |= 0x01;          /* PD0 设置为输出          */
PD_CR1 |= 0x01;         /* 推挽输出          */
PD_CR2 = 0x00;          /* 在输出模式 DDR=1 时，速度 2MHz */

PD_ODR ^= 0x01;        /* LED 亮 */
PD_ODR ^= 0x01;        /* LED 灭 */

```

### 4.1.3 ADC

ADC1 框图，如下图所示：



本文以 STM8S105S4T6C 的 PB3/AIN3 为模拟输入通道，单次转换模式为例。

在单次转换模式中，ADC 操作步骤：

- ADC\_CR1 的 CONT=1 将 ADC 设置为单次转换模式
- 通过 ADC\_CSR 寄存器的 CH[3:0] 选定通道
- 设置 ADC\_CR1 寄存器的 ADON=1 来启动 ADC

一旦转换完成，转换后的数据存储在 ADC\_DR 寄存器中，EOC(转换结束)标志被置位，如果 EOCIE 被置位将产生一个中断。

例子：

```
unsigned int AD_Value;

PB_DDR &=~0x04;      /* 设置 PB3 为输入 */
PB_CR1 &=~0x04;     /* 悬空输入 */
PB_CR1 &=~0x04;     /* 中断禁止 */

ADC_CR1 = 0x00;    /* 预分频 fADC = fMASTER/2, 单次转换模式 */
ADC_CR2 = 0x00;    /* 数据左对齐 */
```

```

ADC_CSR = 0x03; /* 选择 AIN3 作为输入通道 */

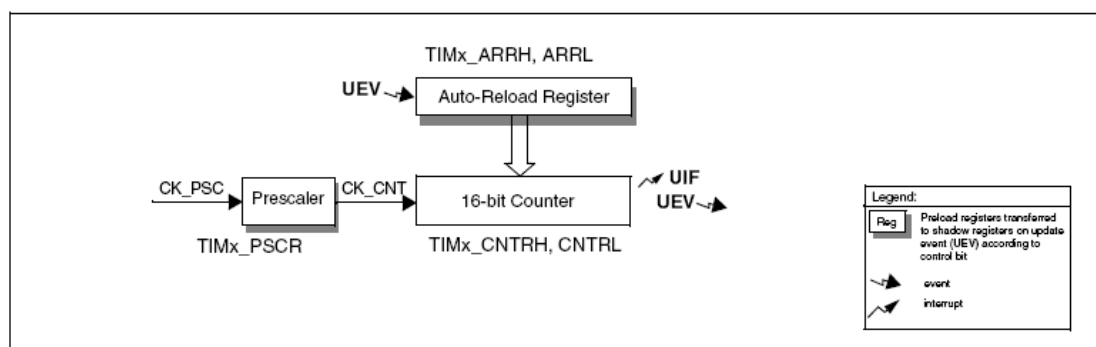
ADC_CR1 |= 0x01; /* 启动 ADC */
/* 如果是第一次启动 ADC, 则需要等待时间>7us, 确保 ADC 电源稳定*/
ADC_CR1 |= 0x01; /* ADON=1, 启动 ADC */
while(!(ADC_CSR & 0x80)); /* 等待转换完成 (EOC=1) */

/* 读取 ADC 的结果到 AD_Value 变量 */
AD_Value = (((unsigned int)ADC_DRH)<<2)+ADC_DRL);

```

## 4.1.4 TIMER2

TIM 时基单元，如下图所示：



计数器使用内部时钟( $f_{\text{MASTER}}$ )，由CK\_PSC提供，并经过预分频器分频产生计数器时钟CK\_CNT。

计数器时钟频率的计算公式：

$$f_{\text{CK\_CNT}} = f_{\text{CK\_PSC}} / 2^{(\text{PSCR}[3:0])}$$

本例中，PSC=0， $f_{\text{CK\_CNT}}=16\text{MHz}$ ，每次计数时间为  $0.0000625\text{ms}$ ，记时  $1\text{ms}$  需要 16000，计数初值为： $65535-16000=49535=0xC17F$

例子：

```

/* TIM2 时基配置 */
unsigned int ms_count; /* 定义一个 ms_count 计数变量*/

void TIM2_Init(void)

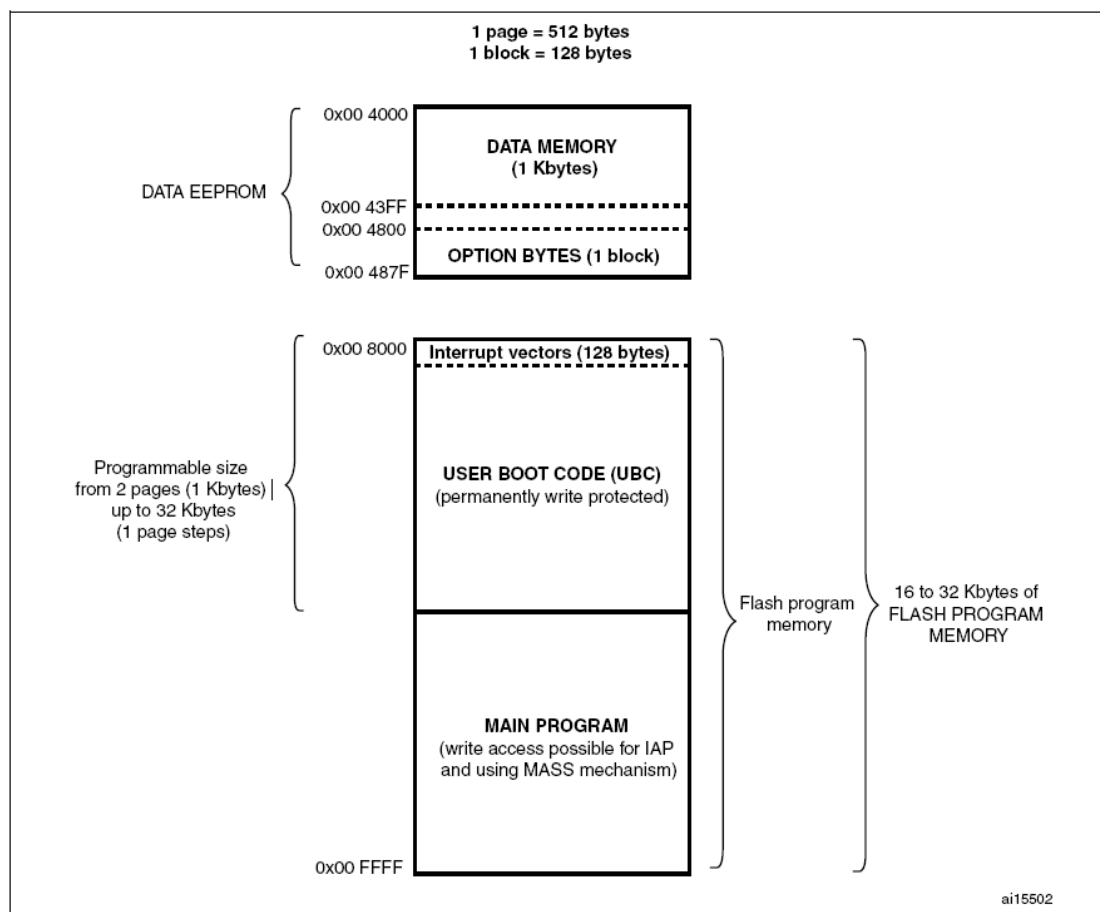
```

```
{
    TIM2_PSCR = 0x00;      /* 配置 TIM2 预分频 =16 */
    TIM2_ARRH = 0xC1;     /* 配置 TIM2 周期, 1ms 中断 */
    TIM2_ARRL = 0x7F;
    TIM2_CNTRH = 0xC1;
    TIM2_CNTRL = 0x7F;
    TIM2_CR1 |= 0x81;     /* 使能 TIM2. */
    TIM2_IER |= 0x01;    /* 使能 TIM2 溢出中断 */
}

/* TIM2 溢出中断 */
TIM2_SR1 &=~(0x01);    /* 清中断标志 */
ms_count++;
if(ms_count == 500)    /* 0.5 秒中断 */
{
    PD_ODR ^= 0x01;     /* PD0 翻转 */
    ms_count = 0;
}
```

### 4.1.5 EEPROM

下面是中容量STM8S的FLASH存储器和数据EEPROM组织机构



### EEPROM 编程模式：

- 字节编程和自动快速字节编程(没有擦除操作)
- 字编程
- 块编程和快速块编程(没有擦除操作)

#### 4.1.5.1 EEPROM 字节编程

EEPROM 在字节编程模式中，可对 EEPROM 的数据区域进行逐字地编程。应用程序直接向目标地址写入数据。

#### EEPROM 字节操作步骤：

1. 设定编程时间。FIX=1 为标准编程时间（一般一次编程时间为 6ms）。如果 EEPROM 被擦除过并且 FIX=0，那么变成时间为标准编程时间的一般（一般为 3ms）
2. 向 FLASH\_DUKR 寄存器连续写入两个 MASS 密钥值来解除 DATA 区域的写保护

第一个硬件密钥：0b1010 1110 (0xAE)

第二个硬件密钥：0b0101 0110 (0x56)

3. 编程成功后，EOP 位置 1。如果 FLASH\_CR1 中的 IE 位预先使能，只要标志位 EOP/WP\_PG\_DIS 中一位置 1，就会产生一个中断

```
volatile unsigned char eeprom0    @0x4000; /* 定义固定地址变量 */
unsigned char eeprom_value;      /* 定义一个显示 EEPROM 值的变量 */
unsigned int eeprom_address;     /* 定义 eeprom_address 变量 */

//定义 EEPROM 字节写 函数
void EEPROM_WRITE_BYTE(unsigned int eeaddress, unsigned char eedata)
{
    eeprom_address = eeaddress;

    /* 设置编程时间，FIX =1: 编程时间固定为标准编程时间 tprog。 */
    FLASH_CR1 &= (unsigned char) (~0x01);
    FLASH_CR1 |= 0x01;

    /* MASS 密钥，解除 EEPROM 的保护 */
    FLASH_DUKR = 0xAE;
    FLASH_DUKR = 0x56;

    *((unsigned char*) eeprom_address) = eedata;

    //EOP=1, EEPROM 编程结束
    while((FLASH_IAPSR & 0x04) != 0x00);
}

main{
    EEPROM_WRITE_BYTE(0x4000,0x77);
    EEPROM_WRITE_BYTE(0x4001,0x88);
    EEPROM_WRITE_BYTE(0x4002,0x99);
    eeprom_address = 0x4000;
    eeprom_value = *((unsigned char*) eeprom_address);
    while(1)
    {_asm("nop");}
}
```

### 4.1.5.2 EEPROM 字编程

EEPROM 允许字编程（一次编程 4 个字节），从而缩短 EEPROM 的编程时间

#### EEPROM 字编程步骤：

1. 设置字编程模式：  
通过设置 FLASH\_CR2 和 FLASH\_NCR2 中的 WPRG/NWPRG 置位/清零来设置字编程模式
2. 向 FLASH\_DUKR 寄存器连续写入两个 MASS 密钥值来解除 DATA 区域的写保护  
第一个硬件密钥：0b0101 0110 (0x56)  
第二个硬件密钥：0b1010 1110 (0xAE)
3. 编程成功后，EOP 位置 1。

```
void EEPROM_WRITE_WORD(unsigned int eeaddress, unsigned char
eedata)
{
    /* 设置编程时间 */
    FLASH_CR1 &= (unsigned char) (~0x01);
    FLASH_CR1 |= 0x01;

    /* MASS 密钥，解除 EEPROM 的保护 */
    FLASH_DUKR = 0xAE;
    FLASH_DUKR = 0x56;

    /* 设置字编程模式 */
    FLASH_CR2 |= 0x40;
    FLASH_NCR2 &= (unsigned char) (~0x40);

    /* 从低地址开始写入 4 个字节数据 */
    *((unsigned char *)eeaddress) = (eedata);
    *((unsigned char *)eeaddress) + 1) = (eedata+1);
    *((unsigned char *)eeaddress) + 2) = (eedata+2);
    *((unsigned char *)eeaddress) + 3) = (eedata+3);
    while((FLASH_IAPSR & 0x04) != 0x00); //EOP=1, EEPROM 编程结束
}

main{
    EEPROM_WRITE_WORD(0x4000,0x11);
    while(1)
    {_asm("nop");}
}
```

### 4.1.5.3 EEPROM 块编程

EEPROM块编程操作允许一次对整个块(128个字节)进行编程，整个块在编程前被自动擦除。但块编程操作一定要在RAM中运行。

#### EEPROM块操作步骤：

1. 设置块操作编程时间
2. 向 FLASH\_DUKR 寄存器连续写入两个 MASS 密钥值来解除 DATA 区域的写保护  
第一个硬件密钥：0b1010 1110 (0xAE)  
第二个硬件密钥：0b0101 0110 (0x56)
3. 设置块编程模式：  
通过设置 FLASH\_CR2 和 FLASH\_NCR2 中的 PRG/NPRG 位预先置位/清零来使能标准块编程
4. 块编程  
向EEPROM的目标地址依次写入要编程的数据，这样数据会被锁存在内部缓存中。为编程整个块，每个块中的所有字节都需要被写入数据。所有被写入缓存的数据必须位于同一个块中，当目标块的最后一个字节被装载到缓存后，编程就自动开始了。
5. 编程成功后，可通过 HVOFF 和 EOP 位来判断是否完成编程操作。STM8 的块大小是不同的。如下表：

STM8 系列单片机	块大小
低密度	64 个字节
中密度	128 个字节
高密度	128 个字节

本例程以中高密度的STM8为例，写第一块(128字节/块)。

```

/* 在RAM中定义一个数组，用于存放EEPROM的写操作代码 */
unsigned char eeprom_write_block_in_ram[100];

void EEPROM_WRITE_BLOCK(void)
{
    unsigned char count;
    unsigned int eeaddress_start;

```



```
eeaddress_start = 0x4000;

/* 设置编程时间 */
FLASH_CR1 &= (unsigned char)(~0x01);
FLASH_CR1 |= 0x01;

/* MASS 密钥, 解除 EEPROM 的保护 */
FLASH_DUKR = 0xAE;
FLASH_DUKR = 0x56;

/* 设置块编程模式 */
FLASH_CR2 |= 0x01;
FLASH_NCR2 &= (unsigned char)(~0x01);

for (count = 0; count < 128; count++)
{
    /* 本例子中在第一个 128 块中, 写入 0x99 数据. 可根据需要写入需要的数据 */
    *((unsigned char *) (eeaddress_start + count)) = 0x99;
}

//判断 EEPROM 块操作是否完成

/* STM8S103, STM8S903 属于低容量, 其 BLOCK 的大小为 64 字节 */
//while((FLASH_IAPSR & 0x04) != 0x00); //EOP=1, EEPROM 编程结束

/* STM8S208, STM8S207, STM8S105 是中大容量, 其 BLOCK 大小为 128 个字节 */
while ((FLASH_IAPSR & 0x40) != 0x00 ); //HVOFF=1 高压结束
}

//-----

/* 将块写函数 EEPROM_WRITE_BLOCK, 拷贝到 RAM 中 */
void COPY_EEPROMWRITEBLOCK_INTO_RAM(void)
{
    unsigned char eerom_count;
    eerom_count=0;
    while ( *((unsigned char*)EEPROM_WRITE_BLOCK + eerom_count) !=
0x81 )
    {
        eeprom_write_block_in_ram[eerom_count] = *((unsigned
char*)EEPROM_WRITE_BLOCK + eerom_count);
    }
}
```

```
        eeprom_count++;
    }
    eeprom_write_block_in_ram[eeprom_count] = 0x81;//RET 指令 0x81
}

main()
{
    /* 调用将 EEPROM 写操作代码拷贝到 RAM 中 */
    COPY_EEPROMWRITEBLOCK_INTO_RAM();

    /*调用 RAM 中的 EEPROM 写操作函数*/
    //_asm("call _eeprom_write_block_in_ram"); //使用 COSMIC C 编译器
    asm("call eeprom_write_block_in_ram"); //使用 IAR 编译器

    _asm("nop");

    while (1)
    {
        _asm("nop");
        _asm("nop");
        _asm("nop");
    }
}
```

**注意：**

- 对 EEPROM 的块操作（一次写入 128 字节），其块写操作代码必须在 RAM 中运行
- 需要将 EEPROM 块写操作代码，拷贝到 RAM 中运行
- 在调用 RAM 调用块写操作代码，完成块写操作。但要注意 IAR 和 COSMIC 运行 RAM 中的代码的方式不同：

在 COSMIC C 编译器中（STVD+COSMIC），调用语句如下：

```
asm("call _eeprom_write_block_in_ram");
```

在 IAR C 编译器中，调用语句如下：

```
asm("call eeprom_write_block_in_ram");
```

## 5 STM8 开发工具

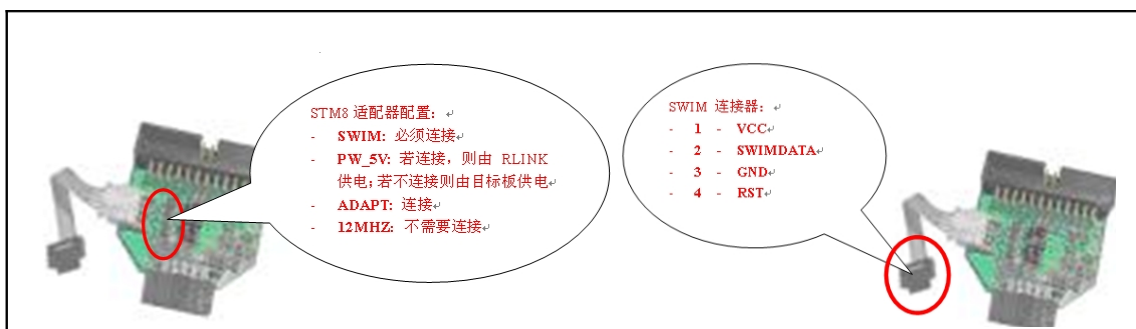
目前，ST 有两款开发工具可以支持 STM8 的开发。工具有：STX-RLINK 和 ST-LINK。其中 ST-LINK 是 ST 的开发工具，其也支持 STM32 和 STM8 两个产品系列。

### 5.1 STX-RLINK

STX-RLINK 是 Raisonance 公司提供的第三方开发工具。STX-RLINK 是一个低成本的调试器/编程器，可以支持 STM32, STR9, STR7, STM8, ST7 和 uPSD。STM8 使用 SWIM 接口调试/编程 ( STM8: 4-pin SWIM ) 。



### STX-RLINK 连接说明



SWIM 连接器			
信号	4PINs	10PINs	24PINs
VCC	1	7	
SWIMDATA	2	2	6, 12, 15 (All three pins must be connected.)
GND	3	1,3,5,10	3,4,10,17,19,21,22
RST	4	4	9

注意：

- 如果在目标板上没有上拉电阻，SWIMDATA 上需要增加一个 2K2 的上拉电阻。
- 需要在目标板上外加 5V 电源

## 5.2 ST-LINK

ST-LINK 是在线调试器和编程器，可用于 STM8 系列和 STM32 系列的设计开发生产。可满足用户大部分的应用开发和生产。



ST-LINK 提供的接口

接口方式	描述
SWIM	用于开发 STM8 系列产品

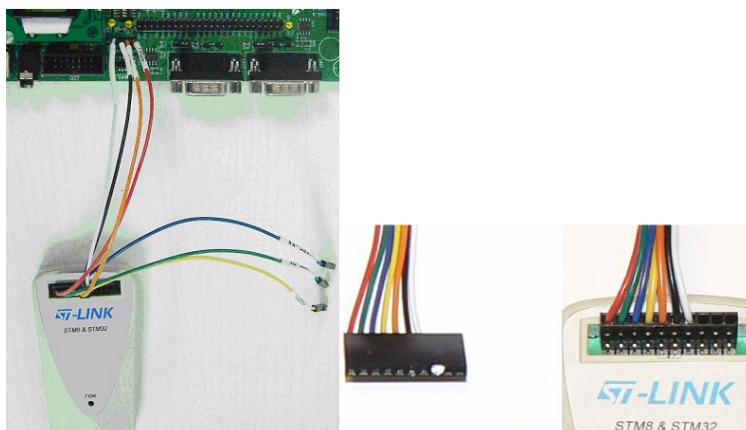
ST-LINK 目前支持的开发环境

开发环境	描述
ST Visual Develop (STVD)	用于开发 STM8 系列产品
IAR EWSTM8	用于开发 STM8 系列产品
COSMIC	用于开发 STM8 系列产品

ST-LINK 与 STM8 系列对应的引脚连接

ST-LINK 引线	STM8 的引脚
TVCC 线	MCU VCC 电源引脚
SWIM 线	MCU SWIM 引脚
GND 线	MCU 的 GND 电源地
SWIM-RST	MCU 复位引脚

ST-LINK 与 STM8 目标板连接如下图所示：



**注意：**

**在调试或编程时，要给目标板提供外部的电源**

### 5.3 ST-LINK/V2

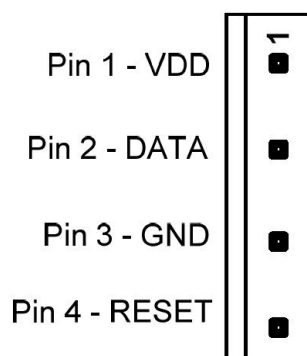


ST-LINK/V2 是一款用于开发 STM8/STM32 的在线调试器/编程器，提供 SWIM 和 JTAG 接口，可调试开发 STM8 和 STM32 全系列的产品。

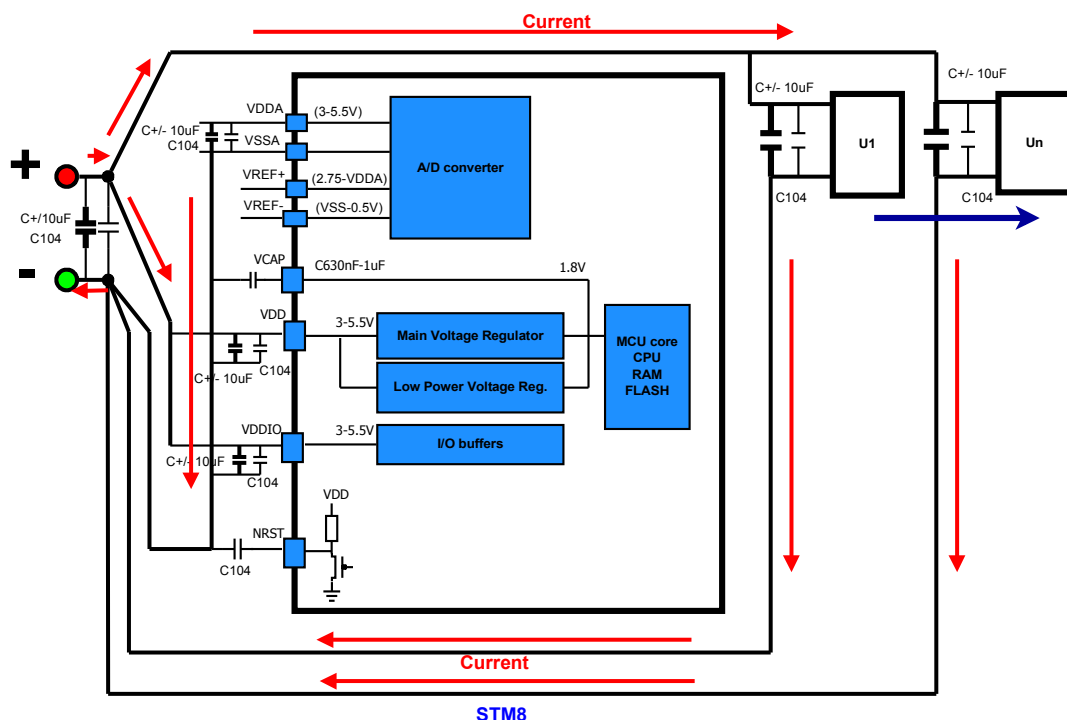
ST-LINK/V2 可支持如下软件：

- STM8: ST Visual Develop(STVD) 或 ST Visual Program(STVP)软件
- STM32: Attolic, IAR, Keil 和 TASKING

SWIM 连接器接口定义：



## 6 STM8 EMC 设计注意事项



### 1) VCAP 电容

STM8S 的 VCAP 电容是一个很特别的电容。它是 STM8S 的内核的工作电压。需要在外部加一个外部的电容，以保证内核工作电压的稳定。一般推荐 1uF 的瓷片电容。1uF 的瓷片电容在进行 PCB 布线时，必须要尽可能地靠近 VCAP 引脚，一直靠近到不能再靠近为止。这一点非常非常重要（STM8L 系列上没有此电容）

### 2) 电源

- VDD 和 VSS 电源引脚上，建议加上退耦电容(10uF 点解电容和 0.1uF 瓷片电容)
- 在电源对 VDDIO\_x 和 VSSIO\_x 的引脚上，建议加上退耦电容(10uF 点解电容和 0.1uF 瓷片电容)。或者至少加上一个 0.1uF 瓷片电容。
- 若在电路中，有用到外部的设备（如 FLASH, 24Co2 等），建议在其电源上加上退耦电容(10uF 点解电容和 0.1uF 瓷片电容)。或者至少加上一个 1uF 瓷片电容。最好不要使其与 MCU 共地。

### 3) 地线

在开始 PCB 布线前，需要全局考虑 GND 的走向。在设计中注意电流回路，特别是 MCU 电流回路要与其他大电流的回路分开。过孔在线路中有阻抗，容易造成的 GND 电势不同，尽可能 GND 布在 PCB 的一面。不建议 GND 走过孔。不建议在 GND 线上加跳线连接。

### 4) 复位

- 对于 STM8 的应用，NRST 复位脚，因内部有一个弱上拉电阻。在应用时可复位电路可只用一个外部的瓷片电容就(一般在 100nF-0.1uF)就可以。也可按照通常的方式加一个

上拉电阻 ( 4.7K-10K )。

### 5) SWIM 调试接口

建议在 SWIM 引脚上，接一个上拉电阻，以保持其数据可靠稳定 ( 4.7K-10K )。

### 6) STM8 时钟

建议采用内部的 RC 时钟作为主时钟。针对一些 STM8S 产品，使用外部时钟，MCU 的抗干扰性能稍弱。又对时钟精度有特别的要求，可用外部的时钟作为一个参考时钟去校验内部的 RC 时钟，仍使用内部 RC 作为主时钟，可避免使用外部时钟抗干扰的问题。

### 7) I2C 接口

STM8 是真正开路输出的接口。在作为一般 GPIO 时，需加上拉电阻。