1. CodeWarrior 中建立新项目

运行 CodeWarrior（CW）集成开发平台，如图 1-1 所示在 File 菜单下点击 New，弹出建立新项目的模板对话框，见图 1-2。

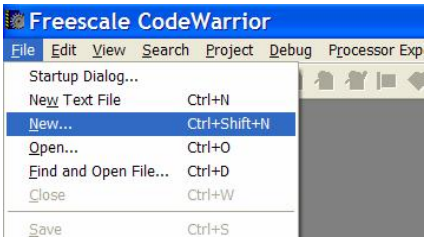

图 1-1

一般的简便做法是在图 1-2 对话框左面的选择列表中选择“HC(S)08 New Project Wizard”，然后在右面的项目名“Project Name”输入条中，输入你要建立的新项目名字，再在“Location”一栏中用 Set... 确定项目存放的文件夹路经，完成后按“OK”进入下一步。

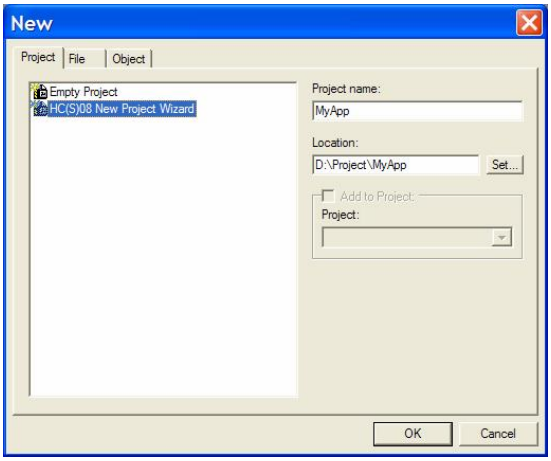

图 1-2

你也可以在图 1-2 对话框左侧列表中选择“Empty Project”，这样生成的项目不包含任何文件，你必须在 CodeWarrior 中自己添加所有相关的文件内容。我想除非有特殊理由，实际项目开发过程中很少采用这种麻烦的方式来建立自己的项目。

接下去是选择项目开发所用的编程语言，见图 1-3。最常用的当然是 C 语言编程。有时因具体项目要求，除了 C 编程外还需要编写独立的汇编语言模块，那就再加选汇编工具（Assembly）。C++编程在免费版和标准版 CW 下都不支持，只有在专业版下才可以使用。编程语言选择完毕后按“Next”。

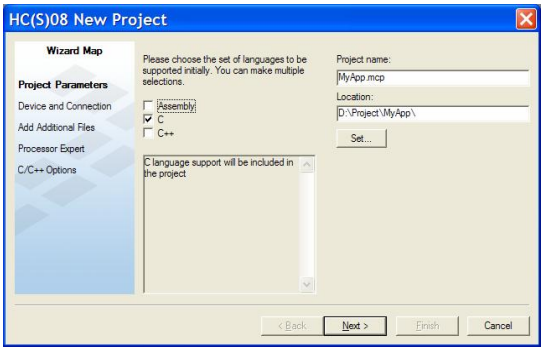

图 1-3

这时将出现如图 1-4 的对话框，让你选择项目开发对应的 MCU 型号。在 CW5.x 版本下支持几乎所有的 HC08 和大部分 HCS08 单片机型号。在最新的 CW6.x 中，增加了飞思卡尔最低端的 8 位机（RS08 系列）和低端 32 位处理器（Coldfire V1 系列）的支持，但 HC08 系列的有些型号没有被包含在内。由于 HC08 为比较老的产品系列，已经不推荐在新项目设计中选用，因此影响不会太大。对于新用户来说，请尽量直接安装 CW6.x 或以后推出的更新版本。

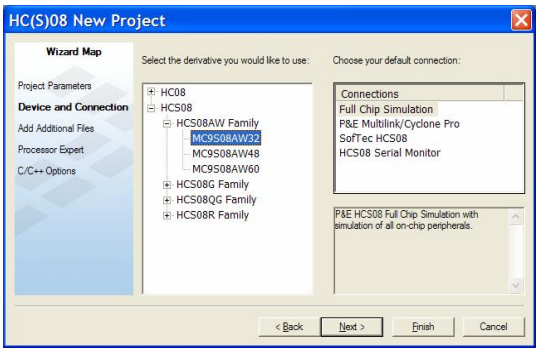

图 1-4

以典型的 9S08 系列为例，当你选择了一个 MCU 型号后，在图 1-4 右侧会显示出所有针对该型号芯片可用的项目调试场景。其中：

- “Full Chip Simulator”是芯片全功能模拟仿真，即无需任何目标系统的硬件资源，直接在你的 PC 机上模拟运行单片机的程序，在模拟运行过程中可以观察调试程序的各项控制和运行流程，分析代码运行的时间，观察各种变量，等等。CW 提供了功能强大的模拟激励功能，可以在模拟运行时模拟一些外部事件的输入，配合程序调试；
- “P&E Multilink/Cyclone Pro”是基于 P&E 公司的硬件调试工具实现实时在线硬件调试。实际就是我们经常说的 BDM 调试。BDM 调试是基于芯片本身内含的在线调试功能，可实现程序下载，单步 / 全速运行，可以设若干个断点，可以观察和修改任意寄存器或 RAM 内存空间。BDM 几乎是开发飞思卡尔 8 位（9S08 和 RS08 系列）、16 位（9S12 系列）和 32 位（Coldfire V1 系列）单片机的标准调试模式，运用最为广泛；
- “SofTec HCS08”是另外一家 SofTec 公司提供的硬件调试工具，国内使用较少；
- “HCS08 Serial Monitor”是基于芯片串口的监控调试开发模式。由于开发效率较低，现在几乎无人使用。

注意不同系列，不同型号的芯片，或不同版本的 CW，其所对应或支持的开发场景可能不同，在图 1-4 的项目建立模板中都可以显现出来。用户点击选择某一项场景后，该场景将在项目建立完成后作为首选配置。你可以在稍后调试过程中随意切换开发场景，不必太在意在这里的选择。

到此你如果按“Finish”，整个项目建立过程将完成，剩下的一些项目设定将自动用缺省配置。如果你要自己选择调整，则按“Next”进入下一步，往项目中添加现成的文件，见图 1-5。

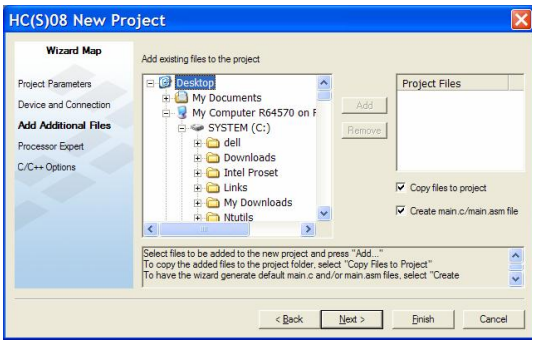

图 1-5

如果你以前编写了很多代码文件现在想重复利用，那么可以通过图 1-5 对话框左面的文件树选择对应的文件，按中间的“Add”逐个添加到右侧的“Project Files”列表中。若加错了就用“Remove”把列表中的文件移除。注意此列表下方的两个选项：“Copy files to project”选择是否将所选的文件拷贝到现在的项目文件夹中。如果你准备在新的项目中修改这些文件，就选择拷贝，以免把原始的文件改变后而影响先前的一些项目；“Create main.c/main.asm file”选择是否在本项目中生成全新的 main.c 或 main.asm 文件，一般的项目开发都需要生成新的 main 文件。按用户自己的要求和目的自由选取。建议大家保留默认的选择状态。如果没有什么现成的文件需要加入，就直接按“Next”进入下一步，选择处理器专家（Processor Expert 或简称 PE）。

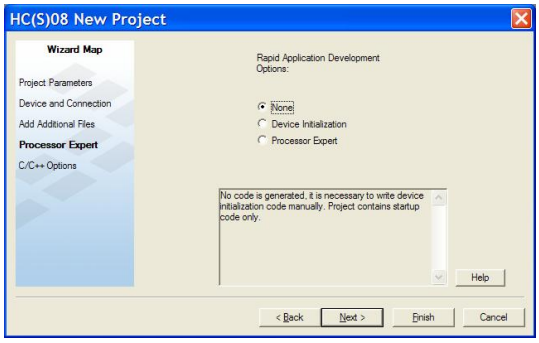

图 1-6

PE 是 CW 集成开发平台内带的可实现芯片内部各种资源模块配置并自动生成相关代码的一个软件工具。不过只有专业版的 CW 才支持该功能。通过 PE，用户可以快速实现芯片初始化代码的自动生成工作，而且 PE 还提供了大量的软件库可供用户开发时嵌入或调用。因为 8 位单片机结构和功能相对简单，实现的控制项目复杂度也不是很高，故一般情况下 8 位机开发我们都不需要 PE 的介入，自己直接编写程序代码即可。关于 PE 的详细介绍将耗费大量的文字，这里按下不提。所以在图 1-6 的对话框中选择“None”,并直接按“Next”进入下一步。

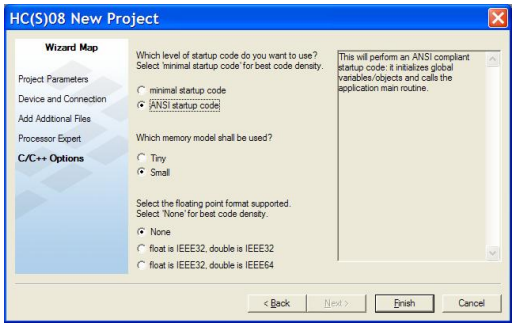

图 1-7

这是项目建立模板的最后一步。在这一步你可以决定有关 C/C++的一些编译和代码生成模式，见图 1-7。

- 启动代码选择。所有 C 编译器会自动生成一些启动代码。单片机复位后的指令运行将首先执行这些启动代码，然后再进入到你自己的程序模块 main 函数。这些启动代码主要完成堆栈指针初始化、全局和静态变量自动清零

或赋初值、调用 main 函数等。ANSI 标准初始化“ANSI startup code”即完成上述工作，是项目开发的标准选择；最小初始化“Minimal startup code”除了初始化堆栈指针外就直接调用 main 函数，代码最少，进入 main 函数最快，但变量的清零和赋初值必须由用户自己编写代码实现。在这里请大家特别注意，即最小初始化将不会对全局或静态变量自动清零，这一点在单片机编程中有时非常重要。在实际产品中当单片机出现异常复位程序重新开始运行时，我们往往希望原先的控制过程得以延续，因此一些关键变量的内容要在复位后保留而不能不分青红皂白地一概清零。选择最小初始化代码可以实现这一特殊要求，但还有更合理更高级的方法，将在后面介绍 prm 文件时详细说明。

- 编译内存模式选择。“Tiny”模式是指所有程序不超过 64KB，RAM 变量不超过内存地址最前面的 256 字节（有时也被称作第 0 页）；“Small”模式程序空间一样不超过 64KB，但 RAM 不限于第 0 页，可以覆盖整个 64K 地址空间。如果你选择的芯片有超过第 0 页空间的 RAM 并想在设计中充分利用，就应该选择该缺省的“Small”模式。
- 浮点运算库选择。当你的程序设计决定用浮点运算时就应该选择加入浮点运算库。浮点运算库有两种：一是标准浮点 float 和双精度浮点 double 都用 32 位精度表示，换句话说 float 和 double 都看成是 float。这样做的目的是减少代码量，提高运算速度；另一种是 double 用 64 位精度表示，毋庸置疑其运算精度将增加，但代码量也将增加，运算时间也会更长。用户可以按实际计算需求酌情选取。如果设计中无需浮点运算，就选择“None”。

全部选择完成并确认后，按“Finish”，恭喜你：你的项目已经成功建立，可以开始编写你自己的代码，调试你的目标系统了。完成后的项目范例如图 1-8，其中：

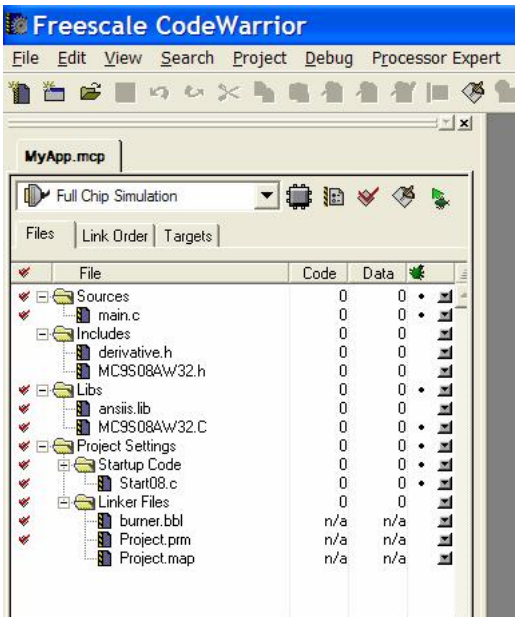

图 1-8

- Sources 栏目下包含所有你的原程序文件，可以是 C，也可以是 asm，或 C++。你可以在此栏下点击鼠标右键在弹出菜单中选择“Add Files”添加其他源程序文件；
- Includes 栏目下包含本项目所有被引用的头文件。你可以自己编写项目相关的头文件并添加到本栏目下；
- Libs 栏目所包含的是本项目开发用到的代码库，可以是目标代码型式或 C 源程序型式；
- Project Setting 下放的全是项目的配置文件。Startup Code 下是刚才建项目时自动生成的启动文件，你可以打开观察具体的程序代码，也可以在必要时自己添加或修改这些启动代码；Link Files 下的三个文件分别是：用于编程器下载的代码文件格式配置（bbl 文件）、机器代码连接定位用的内存说明和配置文件

（prm 文件）、生成的目标代码在内存中的映射文件（map 文件）。其中 prm 文件最为关键，将在稍后重点专门介绍。

2. CodeWarrior 中项目的基本管理和设定

现在项目已经成功建立，应该可以开始编写自己的代码了。但在写代码之前，先了解一下 CW 中最基本的项目管理和设定的方法还是很有必要。

在图 1-8 中项目窗口的右上角有一些小图标，这些图标代表了项目开发管理的最基本功能：

- 该图标可以即时改变目标单片机型号和开发调试场景。按下这一图标，将弹出图 1-4 所示的对话框，可以按照前面针对新项目建立模板的介绍，改变目标单片机的型号，或设定不同的当前目标开发调试场景。对于调试场景的改变，也可以直接点击当前场景右边的下拉菜单按纽，直接用鼠标点击选择所需的新场景，如图 2-1；

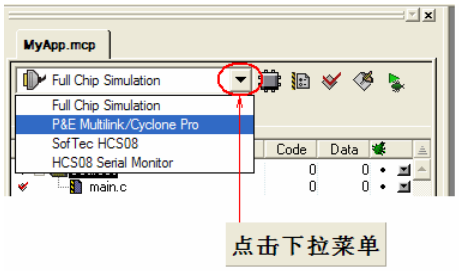

图 2-1

- 该图标完成项目配置选项设定。点击该图标会弹出一个对话框，里面所含的内容非常繁杂，这里只解释几个日常使用时最常用的选项配置：

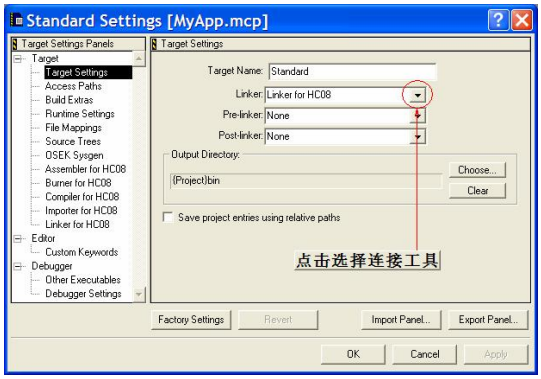

图 2-2

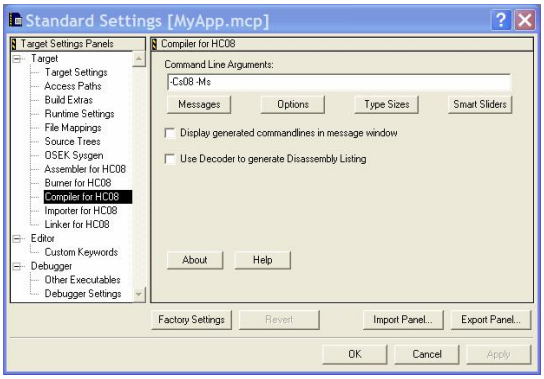

图 2-3

1. 最终目标代码类型设定（Target Setting）。在这里你可以选择最终编译连接生成的代码直接用于单片机程序运行（Link for HC08），或将各个源代码文件编译连接生成一个库文件（Libmaker for HC08）。这些选择可以通过图 2-2 所示的下拉菜单选择；

2. C 编译选项设定（Complier for HC08），对话框内容如图 2-3。在这里你可以完成针对 C 编译器的所有配置设定。几个选项按纽解释如下：

Messages 选择配置编译时产生的各种信息，其中包括普通一般信息（Information）、告警信息（Warning）、错误信息（Error）和致命信息（Fatal）。出现一般或告警信息时编译能顺利完成，所以你可以有选择地将某些你不希望太关注的信息屏蔽掉（Disable）；但如果有任何错误或致命信息出现，当前源程序的编译将立即终止，你必须按给出的信息提示解决这些错误，然后才能继续编译。

Options 完成编译过程中代码生成的各类选项设定，所含内容也很多，最需要关注的是优化栏“Optimization”。你可以按实际需要打开或关闭某些特定的优化选项，但我们一般通过下面介绍的“Smart Sliders”做综合的优化设定。

Type Sizes 显示编译器当前设定的各类变量的长度和符号特性（针对字符和枚举型变量），无特殊原因一般都不用对这些变量长度做任何修改。按常规，在满足功能要求的前提下，变量长度尽可能短，字符型变量尽量选择无符号型，以便提高代码编译效率和程序运行速度。

Smart Sliders 可以非常方便地实现代码优化时的综合考虑。你可以用上面介绍的“Option”对话框可以设定特定的优化选项，但用此“Smart Sliders”对话框则可以针对不同的优化侧重面由系统自动配置具体的优化选项。具体的对话框见图 2-4，你只需用鼠标拉动各项的滑块对其进行优化级别的设定：

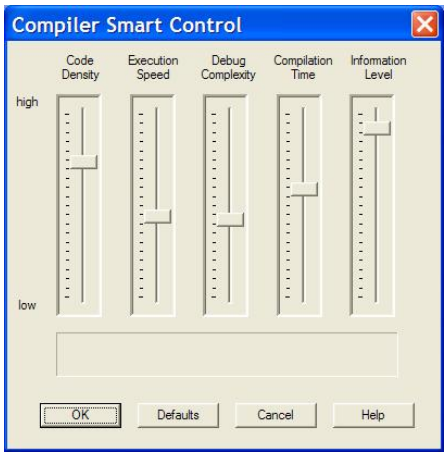

图 2-4

“Code Density”针对生成代码的长度进行优化，设定越高，生成的代码长度越短，代码越高效紧凑；

“Execution Speed”针对代码运行速度进行优化，设定越高，代码执行速度越快；

“Debug Complexity”针对调试复杂度进行优化，设定越高，生成的的调试信息越丰富，调试越方便；

“Compilation Time”针对编译时间进行优化，设定越高，表明编译过程所需时间越长（对应其他各项所做的优化程度越高）；

“Information Level”针对编译信息进行优化，设定越高，表明编译过程中产生的各类信息越丰富。

所有这些编译优化项目都是相互关联的。你移动任意一项滑块的位置，其他各项也会随之自动发生变化。你自己必须有针对性地改变某一项或两项的优化级别，从中作出平衡，无法将所有的优化级别都提到最高。最常用的优化侧重面是代码长度和代码速度这两项。

3. 连接器选项设定（Linker for HC08），对话框内容见图 2-5，值得一提的是其中对 prm 文件的选择。通过项目模板建立的项目其中必含有本项目专用的一个 prm 文件。缺省设置是利用此 prm 文件进行内存分配和连接定位。但你也可以通过此对话框选择使用其他 prm 文件。当你的项目用的是纯汇编单一文件且为绝对定位的编程模式，则不能选择任何 prm 文件，必须设定成“Absolute, Single File Assembly Project”。

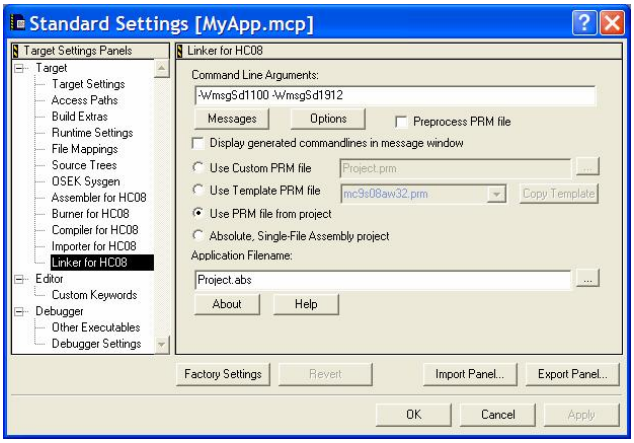

图 2-5

- 该图标检查项目文件是否被更新。当你在 CW 环境中编辑项目中的各个文件时，只要文件内容发生变化，项目列表窗内该文件的左侧会出现此小图标，表明此文件已经被更新，它们在代码生成过程会被重新编译。有时你会用其它你熟悉或喜欢的文本编辑器编辑修改项目中各类文件，当编辑完成文件被保存后，在 CW 环境下按一下这个图标，所有被更新的文件在项目栏中都会得到显现。如果文件左侧没有出现此小图标，表明该文件最近没有被修改过，代码生成时可能不会对它进行重新编译，以节约时间。在任何时候你都可以用鼠标点击源文件左侧该小图标的位置以显现此图标（如果原本没有显现的话），让编译器在代码生成过程中无条件重新编译此文件。

- 该图标进行代码生成（make），鼠标点击该图标后进行源程序的编译和目标代码的连接定位。如果编译连接成功，最后将生成用于源程序符号调试的 abs 文件、用于芯片烧写的 s19 文件、所有变量和函数模块在内存中的映射 map 文件。另外通过 CW 菜单“Project→Make”或键盘快捷键“F7”也可以实现相同功能。

- 该图标用于打开并进入代码调试窗口。鼠标点击该图标后，如果你的项目文件中有最新更新，CW 会自动调用 make 功能进行编译和连接。然后将利用最新生成的 abs 文件，激活一个独立的代码调试窗口，进行源程序级的代码调试。CW 菜单“Project→Debug”或键盘快捷键“F5”同效。因为关系到以后调试程序的方便，在这里还要特别提到编译过程中调试信息的打开和关闭控制。

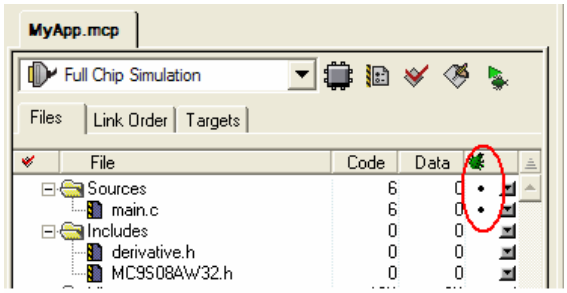

图 2-6

请注意图 2-6 中的黑点，该黑点表明编译此文件时将产生调试信息，如果没有此黑点，生成的 abs 文件中将没有对应的源程序调试信息，你就无法在调试窗口中进行源代码级调试，只能进行汇编代码级调试。你可以用鼠标点击此黑点位置打开或关闭调试信息。所以当你在编译连接结束后发现 L1923 号信息“xxx.o has no DWARF debug info”，请检查对应的文件调试信息有没有打开。

现在，请你在所建立的项目中体会一下上面介绍的这些基本配置及其设定方法。最后按下 图标，看是否能进入调试窗口？应该没问题！

好了，该编写你自己的代码了。

3. C 语言编程要点

CW 中针对 Freescale 的 8 位单片 C 语言编程基本符合 ANSI 规范，因此关于标准 C 语言编程的话题就不再重复。这里主要介绍和单片机资源密切相关的一些编程要点。

3.1 变量类型和定义

CW 中 08 系列单片机 C 编译器支持的基本变量类型及其缺省的长度位数由表 3.1 所示。有些变量的长度可以按实际项目需要而改变，见对话框图 3-1。此对话框经由上面的图 2-3 中“Type Size”配置按纽打开。

表 3.1

长度 / 类型	8 位	16 位	32 位	64 位
char	√			
short		√		
int		√		
long			√	
long long			√	
enum		√		
float			√	
double				√
long double				√
long long double				√

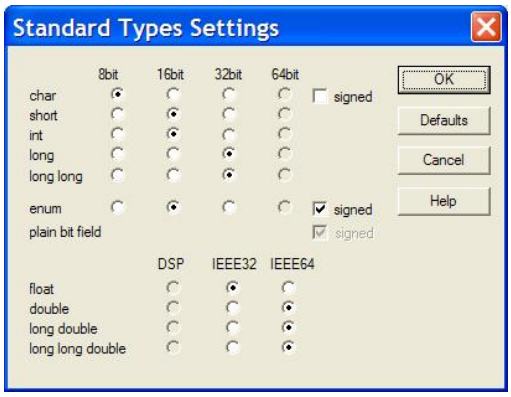

图 3-1

表 3.1 中所示的整形数变量 char、short、int、long 等都有对应的无符号型式（前面冠以 unsigned）。CW 给定的头文件已经将最常用的一些无符号变量类型做了类型

名简化替换，例如用“byte”代替“unsigned char”，用“word”代替“ unsigned int”。这样在程序编写是可以节约点敲键盘的时间。

在单片机程序设计中对于变量类型的选择确认有两条最基本的原则须遵循：能用短的变量就不用长的；能用无符号数就不用有符号数。这两条基本原则将在很大程度上决定你代码的长度和效率。因此请多多使用 byte 或 word 类型变量。

由于 08 系列单片机内部硬件寄存器定义的特点，对于多字节组成的变量，例如 int、long 等，C 编译器缺省的变量内存排列方式是“big endian”模式，即高位字节放在低地址，低位字节放在高地址，又俗称“大头朝上”。这一点相比普通 Intel 格式，例如 51 系列和 PIC 系列正好相反，它们都是“little endian”模式，即“大头朝下”，在程序跨平台移植时请特别注意。当然 CW 编译器本身可以设定改变成“little endian”模式，但单片机内部寄存器地址排列顺序无法改变，故这样做将使最终的程序代码效率降低，特别是在存取一些 16 位长的寄存器组时，它们在硬件上都是由顺序排列的两个 8 位寄存器组成，高字节在前，低字节在后。

单片机程序设计中经常会用到的位变量作为一些标志。CW 中没有特别的位变量定义关键词，位变量必须由位域结构体的型式来定义。例如:

```
struct {
  unsigned powerOn      : 1;
  unsigned alarmOn      : 1;
  unsigned commActive   : 1;
  unsigned sysError     : 1;
} myFlag;
```

若引用某个位变量，只需

```
myFlag.alarmOn  = 1;
myFlag.sysError = 0;
```

这样定义的各个位变量将被顺序排放在一起，以字节为基本单位，字节的第 0 位放第一个位变量，一个字节含 8 个位变量。因此如果位域结构中定义的位变量数目很多，在最后内存分配上将占居多个字节。

有时为了编程方便，位变量需单独定义和操作但又希望一次整个字节一起初始化（清零或赋值），这时我们可以定义字节（或字）和位域结构的联合体：

```
union {
    byte flagByte;
    struct {
      unsigned powerOn      : 1;
      unsigned alarmOn      : 1;
      unsigned commActive   : 1;
      unsigned sysError     : 1;
    } bits;
} myFlag;
```

整字节操作可以

```
myFlag.flagByte = 0;
```

单独某一个位操作可以可以

```
myFlag.bits.powerOn = 1;
myFlag.bits.commActive = !myFlag.bits.commActive;
```

若嫌这样的位变量名称太长，大可以在你自己的头文件里用“#define”预定义，用更简洁易懂的名称进行替换。

最后要提醒的是在定义位变量时尽量将它们指定分配到内存空间的第 0 页（地址范围 0x00-0xff），这样对位变量操作的 C 代码将直接被编译成对应的汇编位操作指令，代码效率最高。具体的定位方法将在介绍“#pragma”时说明。

3.2 变量的特殊修饰

上面介绍的各类基本变量和由其合成的高级变量如数组、结构和联合，将满足95%以上的单片机程序设计工作。由于单片机资源的有限性和特殊型，还有一小部分因素需要在定义变量时加以考虑：

3.2.1 变量的绝对定位

变量绝对定位是特别针对芯片内部的硬件寄存器定义的。所有的硬件寄存器在编写 C 程序时均被视为变量，它们都已在 CW 给定的头文件中预先定义。由于是硬件资源，其地址是唯一且不可改的，所以在头文件中定义这些寄存器时都采用绝对定位的方式，如定义 9S08AW32 的 PORTA：

```
/*** PTAD - Port A Data Register; 0x00000000 ***/
typedef union {
  byte Byte;
  struct {
    byte PTAD0       :1;                /* Port A Data Register Bit 0 */
    byte PTAD1       :1;                /* Port A Data Register Bit 1 */
    byte PTAD2       :1;                /* Port A Data Register Bit 2 */
    byte PTAD3       :1;                /* Port A Data Register Bit 3 */
    byte PTAD4       :1;                /* Port A Data Register Bit 4 */
    byte PTAD5       :1;                /* Port A Data Register Bit 5 */
    byte PTAD6       :1;                /* Port A Data Register Bit 6 */
    byte PTAD7       :1;                /* Port A Data Register Bit 7 */
  } Bits;
} PTADSTR;
extern volatile PTADSTR _PTAD @ 0x00000000;
#define PTAD                           _PTAD.Byte
#define PTAD_PTAD0                     _PTAD.Bits.PTAD0
#define PTAD_PTAD1                     _PTAD.Bits.PTAD1
#define PTAD_PTAD2                     _PTAD.Bits.PTAD2
#define PTAD_PTAD3                     _PTAD.Bits.PTAD3
#define PTAD_PTAD4                     _PTAD.Bits.PTAD4
#define PTAD_PTAD5                     _PTAD.Bits.PTAD5
#define PTAD_PTAD6                     _PTAD.Bits.PTAD6
#define PTAD_PTAD7                     _PTAD.Bits.PTAD7
```

在定义端口寄存器时用“@”给出其绝对地址为 0x00。

理论上用户自己定义的变量也可以用这种方式对其分配一个固定地址来绝对定位。但这样定义的变量其地址不被保留，完全可能被其他变量覆盖。例如用绝对定位的方式定义一个变量 k 在地址 0x70，同时还有其他变量定义，在最后连接定位后的内存映射文件中我们可以看到变量 i 和 k 的地址是重叠的。

```
- VARIABLES:
     prjName                 80B9      F      15      1   .rodata
     x1                      80C8      4       4      1   .rodata
     i                         70      1       1      3   MY_ZEROPAGE
     j                         71      1       1      1   MY_ZEROPAGE
     k                         70      1       1      0   .abs_section_70
```

所以你可以采用上面介绍的方法来绝对定位你自己的变量，但千万千万慎用。我自己实在找不到合适的理由去绝对定位程序中的各类变量。

3.2.2 变量 volatile 声明

声明方法：

```
volatile byte msCounter;
volatile byte uartBuff[16];
volatile word adValue;
```

“volatile”型变量顾名思义就是这些变量是易变的，其值是不随你的程序代码运行而随意改变的。可见，基本所有的单片机片内硬件寄存器其性质是易变的，因为其值变化是由内部硬件模块运作或外部信号输入决定而不受程序代码的控制；你自己定义的变量如果在中断服务程序中被修改，对正常的代码运行流程来说它们也是易变的。“volatile”类型定义在单片机的 C 语言编程中是如此的重要，是因为它可以告诉编译器的优化处理器这些变量是实实在在存在的，在优化过程中不能无故消除。假定你的程序定义了一个变量并对其作了一次赋值，但随后就再也没有对其进行任何读写操作，如果是非 volatile 型变量，优化后的结果是这个变量将有可能被彻底删除以节约存储空间。另外一种情形是在使用某一个变量进行连续的运算操作时，这个变量的值将在第一次操作时被复制到中间临时变量中，如果它是非 volatile 型变量，则紧接其后的其它操作将有可能直接从临时变量中取数以提高运行效率，显然这样做后对于那些随机变化的参数就会出问题。只要将其定义成 volatile 类型后，编译后的代码就可以保证每次操作时直接从变量地址处取数。

任何类型的变量，都可以冠以“volatile”声明。

3.2.3 const 声明

const 用以声明变量为永不变化的常数。一般来说这些变量都应该被放在 ROM 区（也就是 Flash 程序空间）以节约宝贵的 RAM 内存。但简单的一个 const 声明并不能保证变量最后会被分配到 ROM 区，安全的做法必须配合#pragma 声明的“CONST_SEG”数据段或“INTO_ROM”一起实现，这将在稍后介绍。下面为 const 声明的一个范例：

```
const byte prjName[]="This is a demo";
```

任何类型的变量，都可以冠以“const”声明。

3.3 重要的**#pragma** 声明

#pragma 声明是基于单片机开发的特点而对标准 C 语法的一个扩充。它对充分利用单片机内各类有限的资源起到不可或缺的关键作用。下面简单介绍几个最常用的 #pragma 声明。

3.3.1 #pragma DATA_SEG

定义变量所处的数据段。其语法型式为：

```
#pragma DATA_SEG <属性> 名称
```

数据段名称可以自己任意命名，但习惯上有些约定的名称，其作用分别为：

- **DEFAULT** — 缺省的数据段，在 08 系列单片机中的地址为 0x100 以上。一般的变量定义可以放在这一区域。
- **MY_ZEROPAGE** — 特指第 0 页数据段，地址范围 0x00-0xff，但实际用户可用的空间不到 256 字节，因为前面的一些地址空间已经分配给了片内寄存器。需要频繁或快速存取的变量应该指定放在这一特殊区域，特别是位变量。

数据段名称必须和 prm 文件中的数据段配置说明相关连才能真正发挥其定位作用。如果你自己命名的数据段在 prm 文件中没有特别说明，那此数据段的性质等同于“DEFAULT”。

数据段的“属性”可以缺省，它主要的目的是告诉编译器此段数据可适用的寻址模式。不同的寻址模式所花的指令数量和运行时间都不同。对于 08 系列单片机，关键的是第 0 页数据段可以用 8 位地址进行直接快速寻址，故对应此数据段应尽量指明其属性为“__SHORT_SEG”。对于一般数据段没有属性描述，其缺省是“__FAR_SEG”，将用 16 位地址间接寻址。

举几个数据段定义的例子加以进一步说明。

```
#pragma DATA_SEG __SHORT_SEG MY_ZEROPAGE      //开始 0 页数据定义
volatile struct {
   unsigned powerOn      : 1;
   unsigned alarmOn      : 1;
   unsigned commActive   : 1;
   unsigned sysError     : 1;
} myFlag;
volatile word msCounter;
byte i,j,k;

#pragma DATA_SEG DEFAULT                      //开始普通数据段定义（结束 0 页数据段）
byte tmpBuff[16];
```

3.3.2 #pragma CONST_SEG

定义一个常数数据段，必须和变量的 const 修饰关键词配合使用。其语法型式为：

```
#pragma CONST_SEG 名称
```

该数据段下定义的所有数据将被放置在程序只读的 ROM 区，也就是 08 系列单片机内的 Flash 程序空间区。常数段名称可以用户自由定义，但一般都用“DEFAULT”，让连接器按可用的 ROM 区域自由分配变量位置。举例如下：

```
#pragma CONST_SEG DEFAULT
const byte prjName[]="This is a demo";
```

```
const word version  = 0x0301;

#pragma CONST_SEG DEFAULT
word version  = 0x0301;       //没有 const 该变量将被放置在 RAM 区!

#pragma DATA_SEG DEFAULT
const word version  = 0x0301; //尽管有 const 但该变量将被放置在 RAM 区!
```

3.3.3 #pragma INTO_ROM

功能类似于“CONST_SEG”，和变量修饰词“const”配合使用。但它只定义一个常数变量到 ROM 区，且只作用于紧接着的下一行定义。例如：

```
#pragma INTO_ROM
const byte prjName[]="This is a demo";       //变量将被放置在 ROM 区
word verData = 0x0301;                       //变量将被放置在缺省 RAM 区
```

3.3.4 #pragma CODE_SEG

用以定义程序段并赋以特定的段名，语法型式如下：

```
#pragma CODE_SEG <属性> 名称
```

一般的程序设计是无需对代码段做特殊处理的。因为所有传统的 08 系列单片机其程序空间都不超过 64KB（16 位寻址最大范围）且在内存地址中呈线性连续分布。对于项目中所有的代码文件或库文件，连接器会在最后按程序模块出现的先后顺序挨个自动安排所有程序函数在内存中所处的实际位置，用户不必太关心某一个函数的具体位置。但最新推出的几款 8 位机程序将超过 64KB，这样必须在内存空间中以页面型式映射到首 64KB 地址范围，其对应的程序段属性要特殊声明。

某些特殊的设计需要将不同部分的程序分别定位到不同的地址空间，例如实现程序代码下载自动更新。这样的设计需要把负责应用程序下载更新的驱动代码固定放置在一个保留区域内，而把一般的应用程序放置在另外一个区域以便在需要时整体擦除后更新。这时就需要用“CODE_SEG”来分别指明不同的程序段，但还必须配合 prm 文件对程序空间进行分配和指派。

代码段的属性一般都用缺省的“__FAR_SEG”，表明所有的函数调用都是长调用（对应汇编指令为 JSR）。但 C08 和 S08 系列单片机支持效率更高的函数短调用（对应汇编指令为 BSR），如果你的某一个功能模块含有多个相互调用的小函数且函数调用间距不超过+127 或-128 字节，则可以将这部分代码段声明为短调用属性“__NEAR_SEG”。但实际编程时由于 C 代码对应的汇编指令长度不是很容易就能估测得到，所以短调用属性很少使用。

下面以几个实例进一步说明：

```
//定义缺省的代码段，缺省属性为远调用
#pragma CODE_SEG DEFAULT
void main(void)
```

```
{
   ...
}

//定义名字为 FUNC_CODE 的代码段，缺省属性为远调用
#pragma CODE_SEG FUNC_CODE
void MyApp(void)
{
}

//定义远调用的程序段，段名为 BOOTLOAD
#pragma CODE_SEG __FAR_SEG BOOTLOAD
void BootLoader(void)
{
}

//定义近调用的程序段，段名为 KEYBOARD
#pragma CODE_SEG __NEAR_SEG KEYBOARD
void KeyDebounce(void)
{
   ...
}
byte KeyCheck(void)
{
   ...
}

void KeyBoard(void)
{
   if (keyCheck()) {
     KeyDebounce();
     ...
    }

}
```

3.3.5 #pragma TRAP_PROC

用于定义一个函数为中断服务类型。此类型的函数编译器在将 C 代码编译成汇编指令时会在代码前后增加必要的现场保护和恢复汇编代码，同时函数的最后返回用汇编指令“RTI”而不是针对普通函数的“RTS”。例如：

```
#pragma TRAP_PROC
void SCI1_Int(void) {  //定义SCI1的中断服务程序
...
}
```

注意用“TRAP_PROC”定义的中断服务函数其实际中断矢量地址必须通过 prm 文件指派。

3.3.6 #pragma MESSAGE

这个声明用以控制编译信息的显示。一般情况下这些编译信息都是有用的，特别是告警和错误信息。但有时我们会按单片机的工作特性编写一些代码，但正常程序编写时这些代码会产生一些告警信息，例如

```
#pragma MESSAGE DISABLE C4002 //忽略“Result-not-used”告警

//=================================================================
// sci1 1 data receive interrupt service routine
// Assigned for full-duplex communication with Main board
//=================================================================
void interrupt 17 sci11_Receive_ISR(void)
```

```
{
   SCI1S1;      //读一次状态寄存器清除中断标志，会产生 C4002 告警

   sci1RxFifo[sci1FifoPut] = SCI1D;
   sci1FifoPut++;
   sci1FifoPut &= (SCI1_RXFIFO_SIZE-1);
}
```

如果你不想每次都看见编译器给出的这一类信息，可以先确认这一信息的编号，然后用“#pragma MESSAGE”加上“DISABLE” 关键词和信息号将它屏蔽。如果你想特别关注某类信息，可以用“ENABLE”让其永远显示出来。

3.4 编写中断服务函数

编写中断函数几乎是每一个单片机项目开发必需的一个内容。CW 针对 08 系列单片机的中断函数编写有三种方式可以实现。

3.4.1 用关键词 interrupt 和中断矢量编号定义中断函数

这种方式最直观也最简单。缺点是程序的可移植性稍差。在上面的 3.3.6 节中已经给出了实现的范例。关键词“interrupt”告诉编译器此函数为中断服务函数，数字“17”告诉连接器该中断矢量的偏移位置（以复位矢量偏移为 0 计）。某一个中断响应对应的矢量入口编号可以在该芯片的数据手册中查到。

3.4.2 用关键词 interrupt 定义中断函数，中断矢量入口由 prm 文件指定

仍以上面的中断服务函数为例，这是函数的定义方式为

```
void interrupt sci11_Receive_ISR(void)
{
   ...
}
```

然后在项目对应的 prm 文件中添加一行矢量位置定义：

```
VECTOR  0 _Startup            //系统缺省的复位矢量入口
VECTOR 17 sci11_Receive_ISR   //指定的中断服务矢量入口
```

3.4.3 用#pragma TRAP_PROC 定义中断函数，中断矢量入口由 prm 文件指定

实际上就是用前面介绍的“#pragma TRAP_PROC”定义中断函数，再按照和“interrupt”相同的方法在 prm 文件中指定矢量入口，不再重复。

4. prm 文件内容释疑

prm 文件已经在前面反复多次提到。我们在这里仔细看看此文件内包含的详细内容，理解在项目开发过程中起到什么关键作用。

通过项目模板建立的新项目中都有一个名字为“project.prm”的文件，位于图 1-8 所示项目文件列表的“Linker Files”一栏。一个标准的 prm 文件起始内容如下：

例 **4.1 prm** 文件内容实例

```
/* This is a linker parameter file for the AW32 */

NAMES END /* CodeWarrior will pass all the needed files to the linker by command line. But here you may add your own
files too. */

SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in PLACEMENT below. */
    ROM                     =  READ_ONLY    0x8000 TO 0xFFAF;
    Z_RAM                   =  READ_WRITE   0x0070 TO 0x00FF;
    RAM                     =  READ_WRITE   0x0100 TO 0x086F;
    ROM1                    =  READ_ONLY    0xFFC0 TO 0xFFCB;
END

PLACEMENT /* Here all predefined and user segments are placed into the SEGMENTS defined above. */
    DEFAULT_RAM                         INTO  RAM;
    DEFAULT_ROM, ROM_VAR, STRINGS       INTO  ROM;
    _DATA_ZEROPAGE, MY_ZEROPAGE         INTO  Z_RAM;
END


STACKSIZE 0x50

VECTOR 0 _Startup /* Reset vector: this is the default entry point for an application. */
```

4.1 prm 文件组成结构

按所含的信息 prm 文件有五个组成部分构成：

- “**NAMES – END**”部分用以指定在连接时加入除本项目文件列表之外的额外的目标代码模块文件，这些文件都是事先经 C 编译器或汇编器编译好的机器码目标文件而不是源代码文件。不过这种用法比较少见，因为我们可以在图 1-8 所示项目文件列表的“Libs”一栏中添加这些目标代码文件来实现同样的任务，而且由项目列表管理这些模块文件比较直观方便。
- “**SEGMENTS – END**”部分定义和划分芯片所有可用的内存资源，包括程序空间和数据空间。一般我们将程序空间定义成“ROM”，把数据空间划分成第 0 页的“Z_RAM”和普通区域的“RAM”，但实际上这些名字都不是系统保留的关键词，可以由用户随意修改。用户也可以把内存空间按地址和属性随意分割成大小不同的块，每块可以自由命名。关于内存划分的具体方法在后面详解。
- “**PLACEMENT – END**”部分将指派源程序中所定义的各种段，例如数据段 DATA_SEG、CONST_SEG 和代码段 CODE_SEG 被具体放置到哪一个内存块中。它是将源程序中的定义描述和实际物理内存挂钩的桥梁。
- “**STACKSIZE**”定义系统堆栈长度，其后给出的长度字节数可以根据实际应用需要进行修改。堆栈的实际定位取决于 RAM 内存的划分和使用情况。

在常见的 RAM 线性划分变量连续分配的情况下，堆栈将紧挨在用户所定义的所有变量区域的高端。但如果你将 RAM 区分成几个不同的块，请确保其中至少有一个块能容纳已经定义的堆栈长度。

- “**VECTOR**”定义所有矢量入口地址。模板在生成 prm 文件时已经定义了复位矢量的入口地址。对于各类中断矢量用户必须自己按矢量编号和中断服务函数名相关联，请参考 3.4.2 中代码范例。如果中断函数的定义是用“interrupt”加上矢量号，则无需在这里重复定义。

prm 文件中可以添加注释，语法和 C 语言相同，可以是“/*…*/”或“//”。

4.2 内存划分的具体方式

由“**SEGMENTS**”开始到“**END**”为止，中间可以添加任意多行内存划分的定义，每一行用分号“;”结尾。定义行的语法型式为：

[块名] = [属性] [起始地址] TO [结束地址];

其中，

- “块名”的定义和 C 语言变量定义相同，是以英文字母开头的一个字符串。
- “属性”可以有三种不同的类型。对于只读的 Flash-ROM 区属性一定是“READ_ONLY”，对于可读写的 RAM 区属性可以是“READ_WRITE”，也可以是“NO_INIT”。它们两者的关键区别是 ANSI-C 的初始化代码会把定位在“READ_WRITE”块中的所有全局和静态变量自动清零，而“NO_INIT”块中的变量将不会被自动清零。对于单片机系统，变量在复位时不被自动清零这一特性有时是很关键的。
- 起始地址和结束地址决定了一内存块的物理位置，用 16 进制表示。

下面举几个例子来进一步说明：

例 **4.2** 划分 **Flash-ROM** 区，定义 **512** 字节 **EEPROM** 模拟区

```
SEGMENTS
    EEPROM    =  READ_ONLY    0x8000 TO 0x81FF;
    ROM       =  READ_ONLY    0x8200 TO 0xFFAF;
    Z_RAM     =  READ_WRITE   0x0070 TO 0x00FF;
    RAM       =  READ_WRITE   0x0100 TO 0x086F;
END
```

例 **4.3** 划分 **RAM** 区，定义 **16** 字节非自动清零的数据保留区

```
SEGMENTS
    ROM       =  READ_ONLY    0x8000 TO 0xFFAF;
    Z_RAM     =  READ_WRITE   0x0070 TO 0x00FF;
    RAM_KEEP  =  NO_INIT      0x0100 TO 0x010F;
    RAM       =  READ_WRITE   0x0110 TO 0x086F;
END
```

用“SEGMENTS”只是从单片机的物理内存这一角度对其进行空间划分。源程序本身并不知道物理内存被分割和属性定义的这些细节。它们两者之间必须通过下面的“PLACEMENT”建立联系。

4.3 程序段和数据段的放置

“PLACEMENT … END”内所描述的信息是告诉连接器源程序中所定义的各类段应该被具体放置到哪一个内存块中去。其语法型式为：

[段名 1], [段名 2], . . . [段名 n] INTO [内存块名];

其中

- 段名就是在源程序中用“#pragma”声明的数据段、常数段或代码段的名字。如果用缺省名“DEFAULT”，则默认的数据段名为“DEFAULT_RAM”，代码段和常数段名为“DEFAULT_ROM”。若程序中定义的段名没有在 PLACEMENT 中提及，则将被视同为 DEFAULT。几个相同性质但不同名字的段可以被放置到同一个内存块中，相互之间用逗号“,”分隔。
- INTO 是系统保留的关键词，在这里为“放入”的意思。
- 内存块名就是前面介绍的用“SEGMENTS”划分好的不同的内存块名字。

利用这样直观的定位描述文本可以方便灵活的将你的数据或代码定位到芯片内存任意可能的位置，实现某些特殊目的的应用。下面举几个例子，注意各种段名、PLACEMENT 和 SEGMENTS 之间的对应关系。

例 **4.4** 定义并保留 **512** 字节 **Flash** 字节作为 **EEPROM** 模拟

```
//prm 文件 SEGMENTS 定义:
SEGMENTS
    EEPROM    =  READ_ONLY    0x8000 TO 0x81FF;
    ROM       =  READ_ONLY    0x8200 TO 0xFFAF;
    Z_RAM     =  READ_WRITE   0x0070 TO 0x00FF;
    RAM       =  READ_WRITE   0x0100 TO 0x086F;
END

//prm 文件 PLACEMENT 定义:
PLACEMENT
    DEFAULT_RAM                         INTO  RAM;
    DEFAULT_ROM, ROM_VAR, STRINGS       INTO  ROM;
    _DATA_ZEROPAGE, MY_ZEROPAGE         INTO  Z_RAM;
    EE_DATA                             INTO  EEPROM;
END

//源程序编写:
#pragma CONST_SEG EE_DATA
const byte eeDataBuff[512]="123456";
```

例 4.5 将不同的代码段分别放置于不同的程序区

```
//prm 文件 SEGMENTS 定义:
SEGMENTS
    BOOT_SECTOR     =  READ_ONLY     0x8000 TO 0x87FF; //2KB 作为加载引导专用区
    ROM             =  READ_ONLY     0x8800 TO 0xFFAF;
    Z_RAM           =  READ_WRITE    0x0070 TO 0x00FF;
    RAM             =  READ_WRITE    0x0100 TO 0x086F;
END

//prm 文件 PLACEMENT 定义:
PLACEMENT
    DEFAULT_RAM                          INTO  RAM;
    DEFAULT_ROM, ROM_VAR, STRINGS        INTO  ROM;
    _DATA_ZEROPAGE, MY_ZEROPAGE          INTO  Z_RAM;
    BOOT_LOADER                          INTO  BOOT_SECTOR;
END

//源程序编写:
#pragma CODE_SEG BOOT_LOADER  //定义专用的加载引导代码段
void CodeLoader(void)
{
   ...
}


#pragma CODE_SEG DEFAULT      //普通代码段
void main(void)
{
   ...
}
```

例 4.6 定义非自动清零的数据段

```
//prm 文件 SEGMENTS 定义:
SEGMENTS
    ROM             =  READ_ONLY     0x8000 TO 0xFFAF;
    Z_RAM           =  READ_WRITE    0x0070 TO 0x00FF;
    RAM_KEEP        =  NO_INIT       0x0100 TO 0x011F; //32 字节非自动清零数据段
    RAM             =  READ_WRITE    0x0120 TO 0x086F;
END

//prm 文件 PLACEMENT 定义:
PLACEMENT
    DEFAULT_RAM                          INTO  RAM;
    DEFAULT_ROM, ROM_VAR, STRINGS        INTO  ROM;
    _DATA_ZEROPAGE, MY_ZEROPAGE          INTO  Z_RAM;
    DATA_PERSISTENT                      INTO  RAM_KEEP;
END

//源程序编写:
#pragma DATA_SEG DATA_PERSISTENT      //定义复位时非自定清零数据段
byte sysState;
word pulseCounter;
```

建议阅读文档（英文版）：
C 编译器使用手册：Compiler_HC08.pdf
汇编器使用手册：Assembler_HC08.pdf
连接器使用手册：Build_Tools_Utilities.pdf