嵌入式系统的C语言

译自《C for Embedded Systems》讲稿

刘永重　译

一、C语言基础

1、什么是C?

‘C’程序语言最初是由Dennis Ritchie在1971年为UNIX系统开发并实现的。C的一个最大优点是与任何特定的硬件或系统无关。这使得一个用户写的程序不作任何修改就能运行在几乎所有的机器上。

C通常被称为中级计算机语言，因为它将高级语言的要素与汇编语言的功能结合了在一起。

2、为什么用C?

C非常灵活，而且可随心所欲。这种自由赋予C非常强大的功能，有经验的用户可以掌握；C是一个相对小的语言，但是它经久耐用；C有时被认为是“高级汇编语言”；低级（位操作）编程也容易实现；松类型（不象其它高级语言）；C是结构化编程语言；C允许你创建你脑海中已有的任何任务。

C保留了程序员知道正在做的事情的基本体系；它只需要他们明白地表达其意图。

3、为什么不用C? 文化的问题…

当考虑转到C语言时，我们会遇到一些共同的问题：

产生大而低效的代码；标准IO程序的雍余代码（printf，scanf，strcpy等）；存贮器定位的使用：malloc(),alloc()…；堆栈的使用，在C中不很直接；在RAM和ROM中数据的声明；难于写中断服务程序。

4、8位微控制器的ANSI C

对于嵌入式系统，纯粹的ANSI C并不方便，因为：

嵌入式系统与硬件打交道。ANSI C 提供的在固定存贮空间用寄存器寻址的工具非常拙劣；几乎所有的嵌入式系统使用中断；ANSI C有各种类型的促进规则，对8位机来说绝对是性能杀手；一些微控制器结构没有硬件支持C堆栈；很多微控制器有多个存贮空间。

5、打破一些C范例

当在低端的8位微控制器上用C语言，应想法使代码变小。这意味着打破一些编程规则：

开/关全局中断；使用GOTO语句；全局标号；全局寄存器段；指针支持。

6、嵌入式与桌面编程

嵌入式编程环境的主要特点：

有限的RAM；有限的ROM；有限的栈空间；面向硬件编程；严格的定时（ISR，任务，…）；很多不同种类的指针（far/near/rom/uni/paged/…）；特殊关键字/标识符（@，interrupt，tiny，…）。

7、汇编与C

编译器只是一个能干的优秀汇编程序员。

写能够转换为高效率汇编代码的好的C代码，比手工写高效率的汇编代码容易得多。

C是终极解决办法，但其本身并未终结。

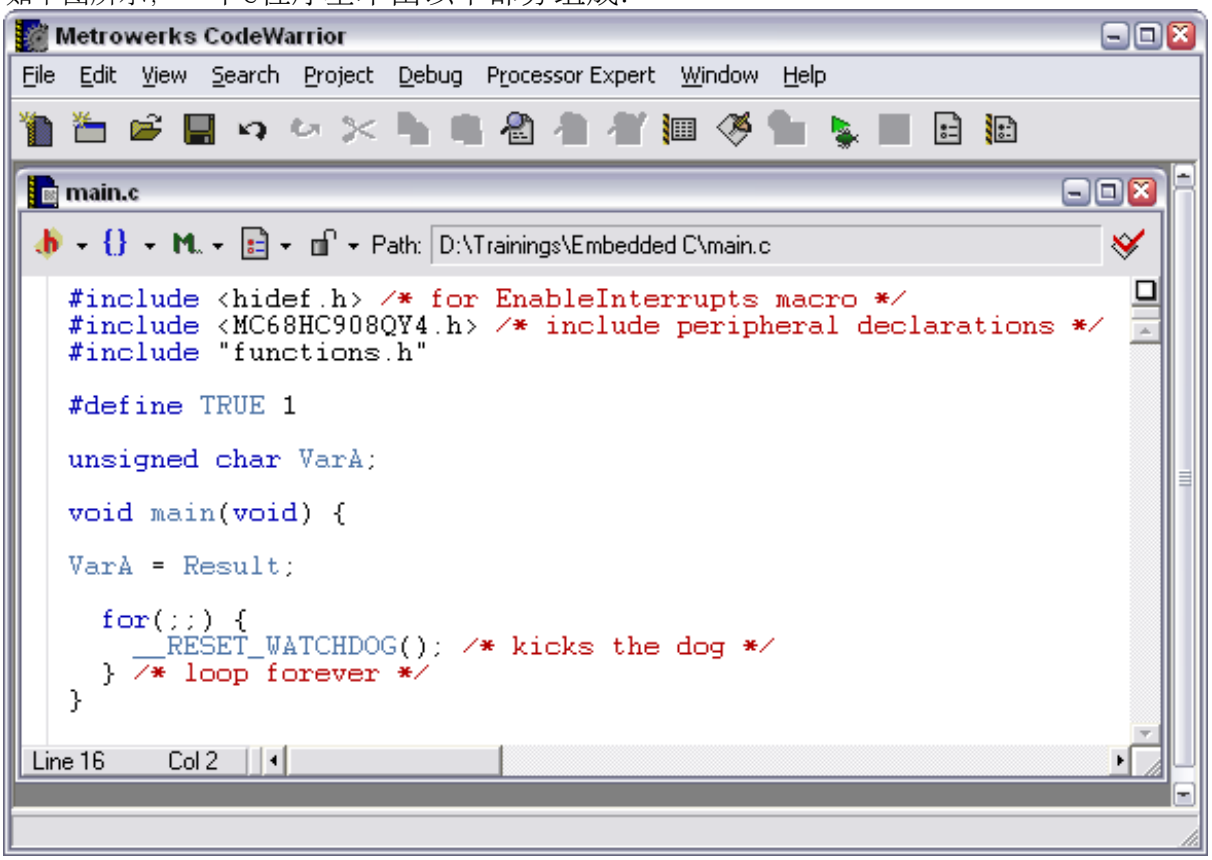
8、为什么改用C?

有很多原因用C语言而不用汇编：

C使我们提高效益；用C写的代码更可靠；C代码更容易升级和扩展；不同平台之间更容易迁移；代码容易维护；文档、书籍、第三方库和程序都可得到。

9、C代码结构

如下图所示，一个C程序基本由以下部分组成：



预处理命令、类型定义、函数原型（声明传给函数的函数类型和变量）、变量和函数。

一个程序必须有一个main()函数，每个命令行必须用分号（；）结束。

10、C函数

一个函数的结构如下：

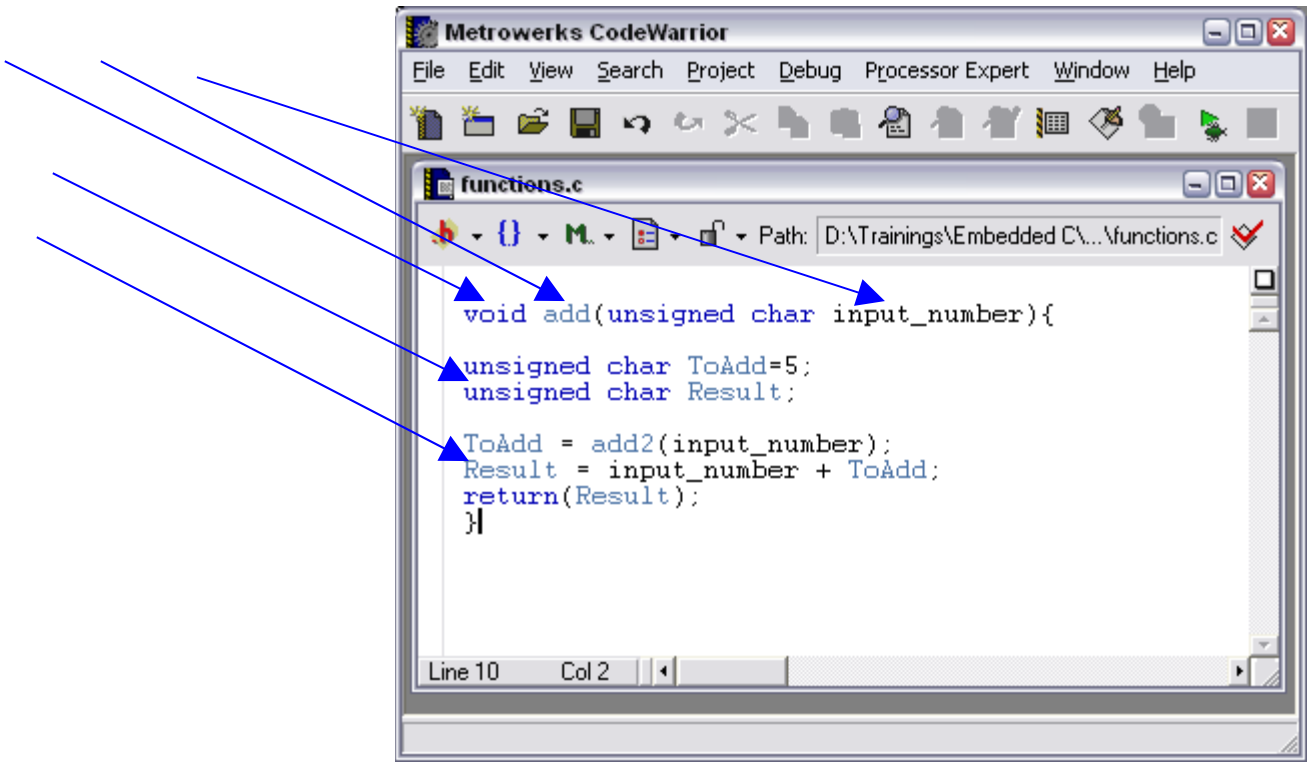
类型 函数名（参数）

{

本地变量

C语句

}



11、C关键字

1）数据类型

char short signed unsigned int float long double

2）修饰符

const static volatile restrict

3）标识符

struct union void enum

4）选择体

if else switch case default

5）存贮指定

register typedef auto extern

6）循环体

do while for

7）跳转

goto continue break return

8）功能指定

inline

9）预处理指示

#include #define #undef #line #error #pragma

10）条件编译

if # ifdef # ifndef # elif # else # endif

12、C操作符

1）基本表达式和后缀操作符

( ) 子表达式和函数调用 [ ] 数组下标 -> 结构指针 . 结构成员

++ 增加(后缀) -- 减少(后缀)

2）一元操作符

! 逻辑非 ~ 取补 ++ 增加(前缀) -- 减少(后缀) - 一元减 + 一元加

(类型) 类型强制 * 间接指针 & 取地址 sizeof 大小

3）赋值符

= 相等赋值 += 加等于 -= 减等于 *= 乘等于 /= 除等于 %= 求余等于

<<= 左移位等于 >>=右移位等于 &=按位与等于 ^= 按位异或等于

|= 按位或等于

4）位操作

& 位与 ^ 位异或 | 位或 < < 位左移 >> 位右移

5）数学运算

* 乘 / 除 %求余 + 加 - 减

6）关系运算

< 小于 <= 小于或等于 > 大于 >=大于或等于 ==相等测试 !=不等测试

7）逻辑运算

&& 逻辑与 || 逻辑或

8）条件运算

?: 条件测试

9）序列

, 逗号

二、嵌入式编程

1、变量

变量的类型决定其可带值的类型。也就是说，为变量选择一个类型与我们使用这个变量的方法直接相关。我们将学习C的基本类型、怎样写常量和声明这些变量。

1．1 选择一个类型

“值集合”是有限的。C的整数类型不能代表所有整数；它的浮点类型也不能代表所有浮点数。当声明一个变量并为它选择一个类型，你应紧记你需要的值和操作。

1．2 C的基本数据类型

ANSI标准并没为本地类型规定尺寸大小，但CodeWarrior规定了。C只有一些基本数据类型：

Type	Default format	Default value range		Formats available with option -T
		min	max	
char (unsigned)	8bit	-128	127	8bit, 16bit, 32bit
signed char	8bit	-128	127	8bit, 16bit, 32bit
unsigned char	8bit	0	255	8bit, 16bit, 32bit
signed short	16bit	-32768	32767	8bit, 16bit, 32bit
unsigned short	16bit	0	65535	8bit, 16bit, 32bit
enum (signed)	16bit	-32768	32767	8bit, 16bit, 32bit
signed int	16bit	-32768	32767	8bit, 16bit, 32bit
unsigned int	16bit	0	65535	8bit, 16bit, 32bit
signed long	32bit	-2147483648	2147483647	8bit, 16bit, 32bit
unsigned long	32bit	0	4294967295	8bit, 16bit, 32bit
signed long long	32bit	-2147483648	2147483647	8bit, 16bit, 32bit
unsigned long long	32bit	0	4294967295	8bit, 16bit, 32bit

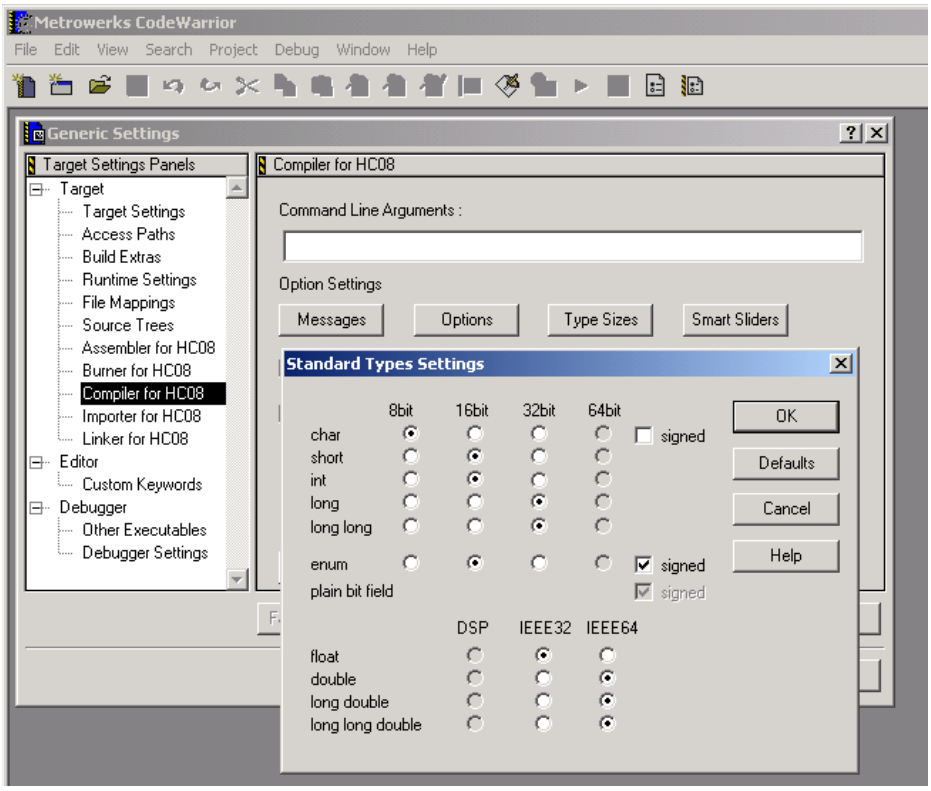
所有数量类型（除了char）缺省都是有符号的，例如：‘int’ = ‘signed int’。

注意：INT型的大小依赖于不同的机器。

1．3 CodeWarrior数据类型

例如,按ALT+F7打开工程的通用设置,选择“Compiler for HC08”面板并点击类型尺寸。这个窗口向你显示CodeWarrior 编译器使用的标准类型设置。

所有基本类型可以改变，尽管这可能不是个好主意。



1．4 数据类型的事实

代码大小和执行时间的最大节约可通过为变量选择最合适的数据类型得到。

8位微控制器内部的数据的长度是8位（一字节），然而C首选的数据类型是‘int‘。

8位机处理8位数据类型比16位类型效率更高。

“int“和大数据类型只有当所描述的数据的大小需要时才使用。

当效率非常重要时，双精度和浮点操作效率低，应当避免。

1．5 选择数据类型

8位微控制器选择数据类型有3个规则：

1）用最可能小的类型来完成工作，大小越小占用存贮空间越少；

2）若可能，用无符号类型；

3）在表达式内声明以将数据类型减到最少需要。

使用类型定义得到固定大小：

1）根据编译器和系统而改变；2）移植到不同的机器代码不变；3）当值需要固定位时使用。

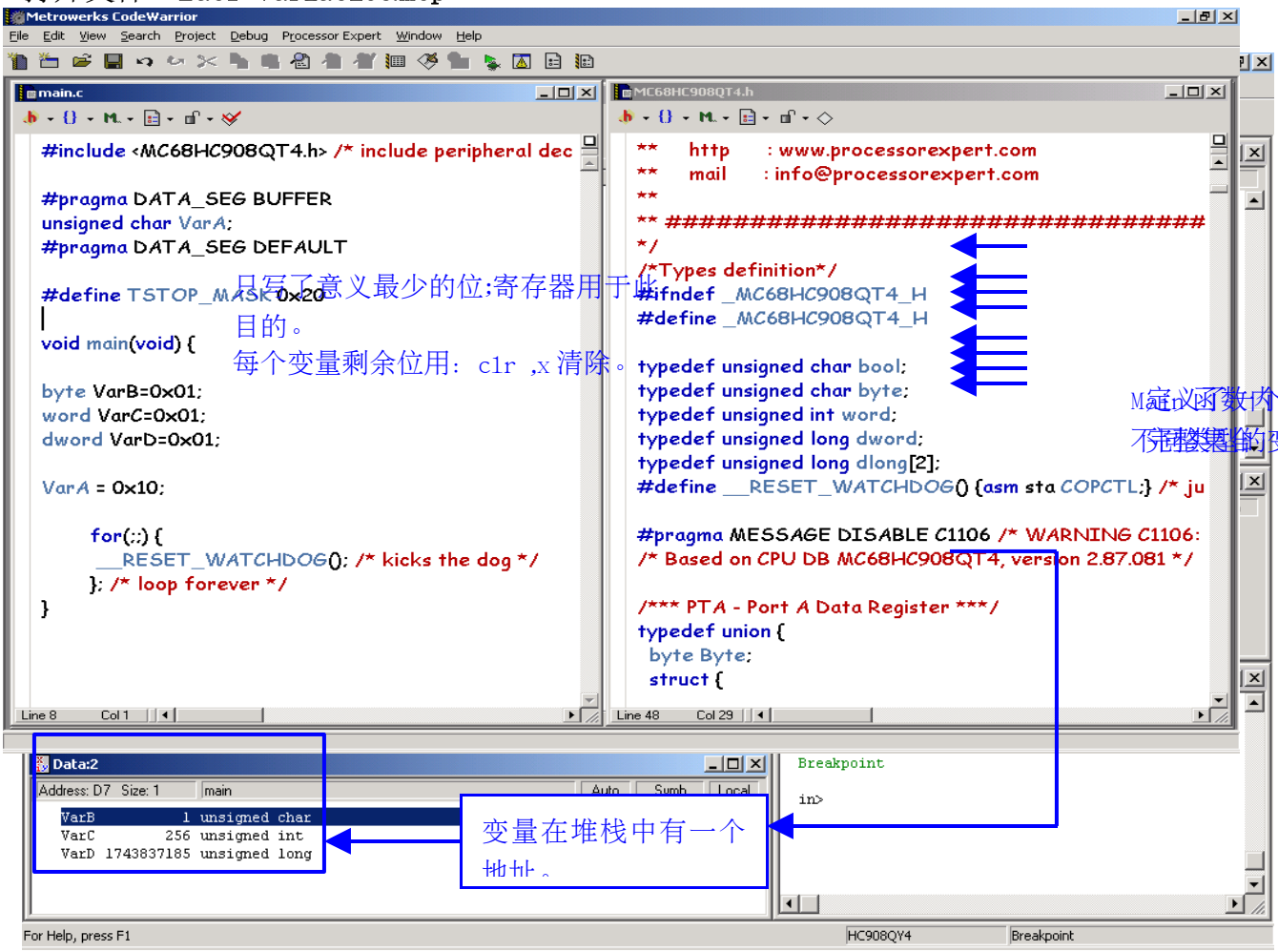
***8-bit machine**

```
/* Fixed size types */
typedef unsigned char uint8_t;
typedef int           int16_t;
typedef unsigned long uint32_t;
```

***32-bit machine**

```
/* Fixed size types */
typedef unsigned char uint8_t;
typedef short         int16_t;
typedef unsigned int  uint32_t;
```

打开文件：Lab1-Variables.mcp

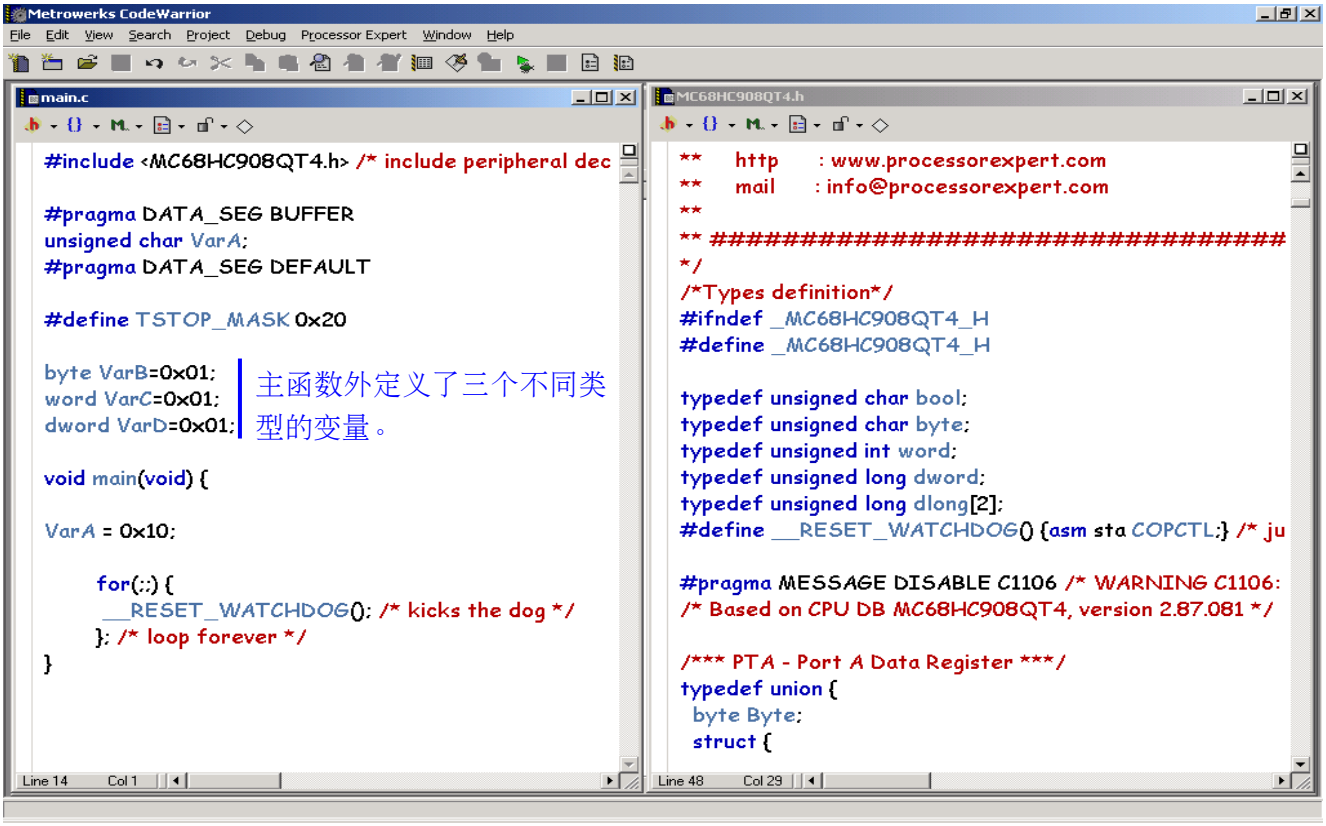


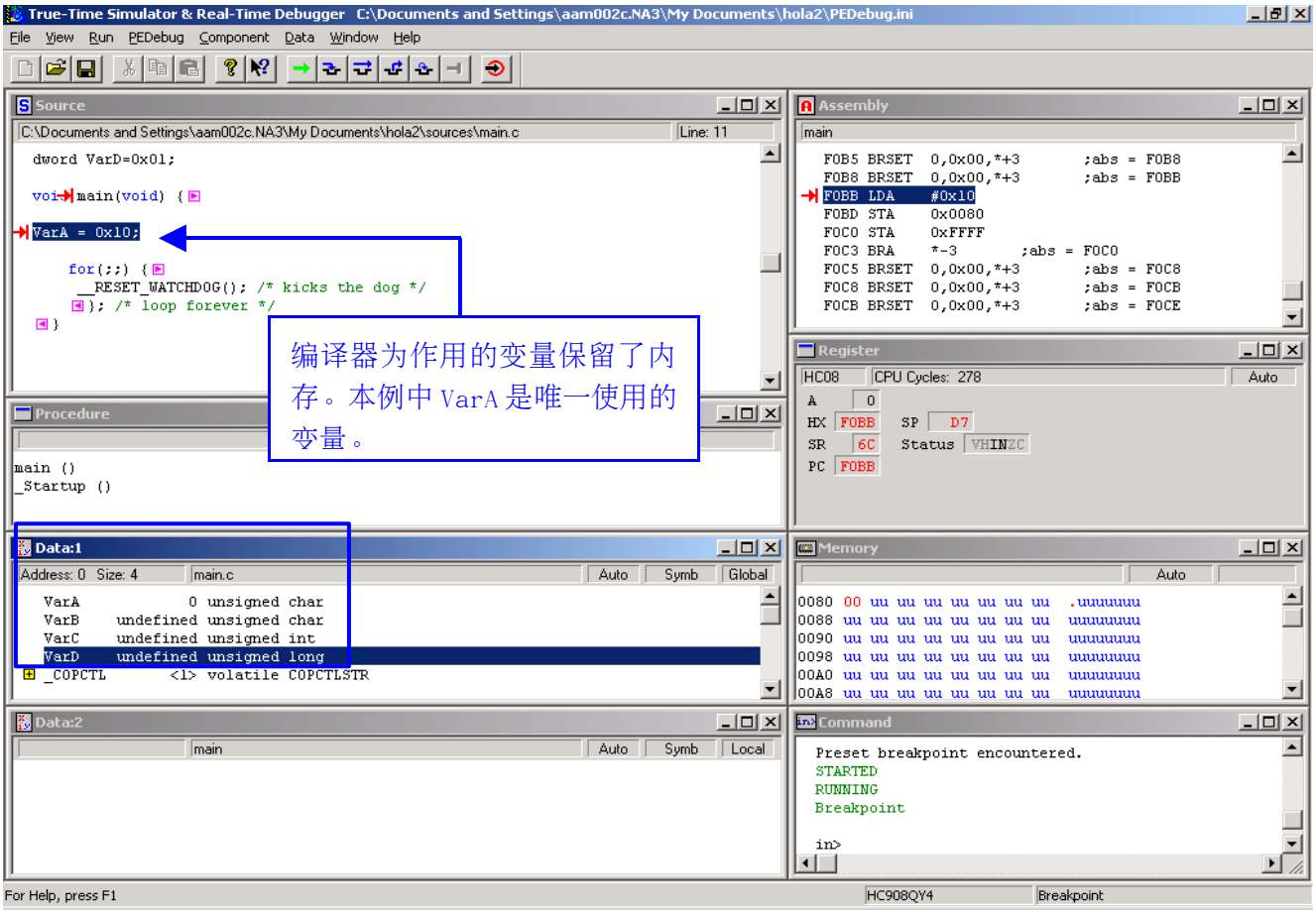
只写了意义最少的位;寄存器用于此目的。

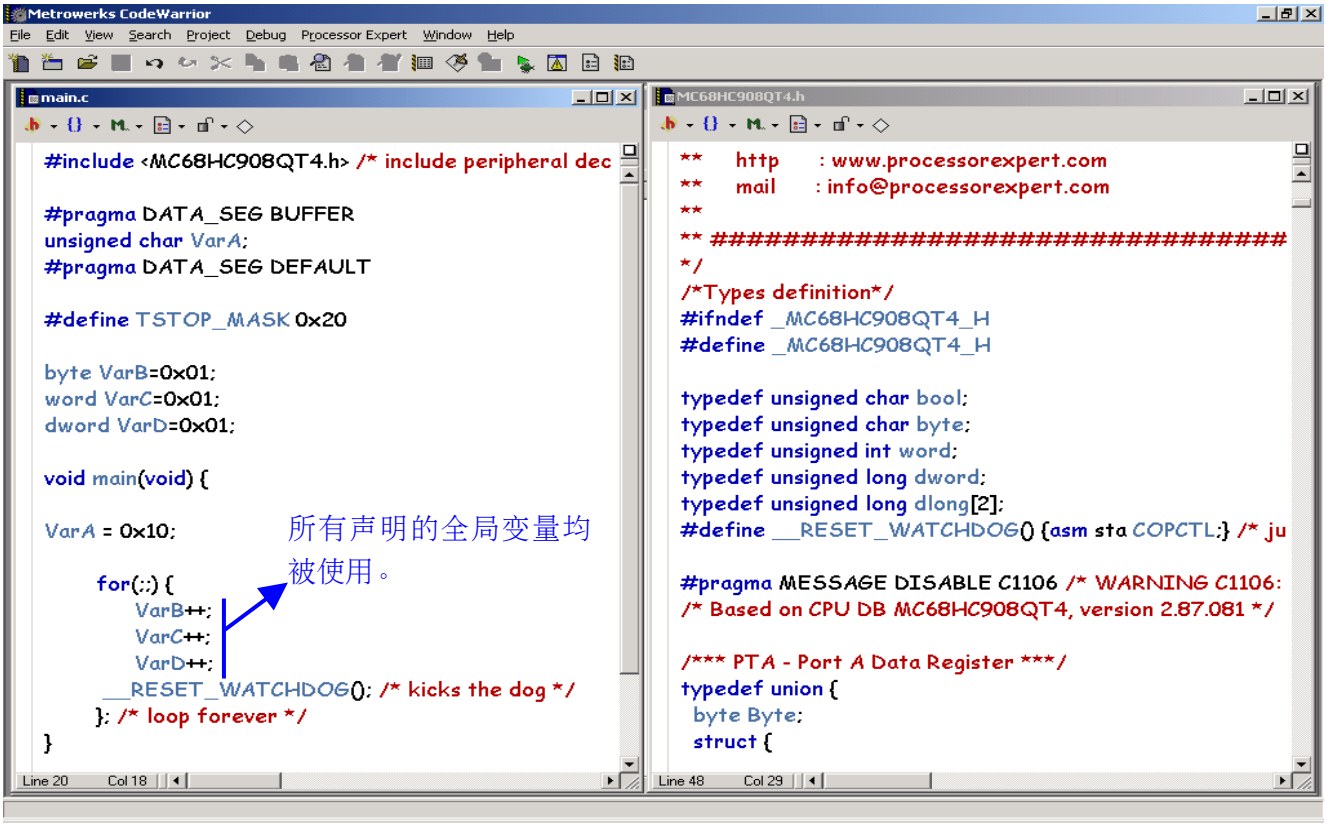
每个变量剩余位用：clr ,x 清除。

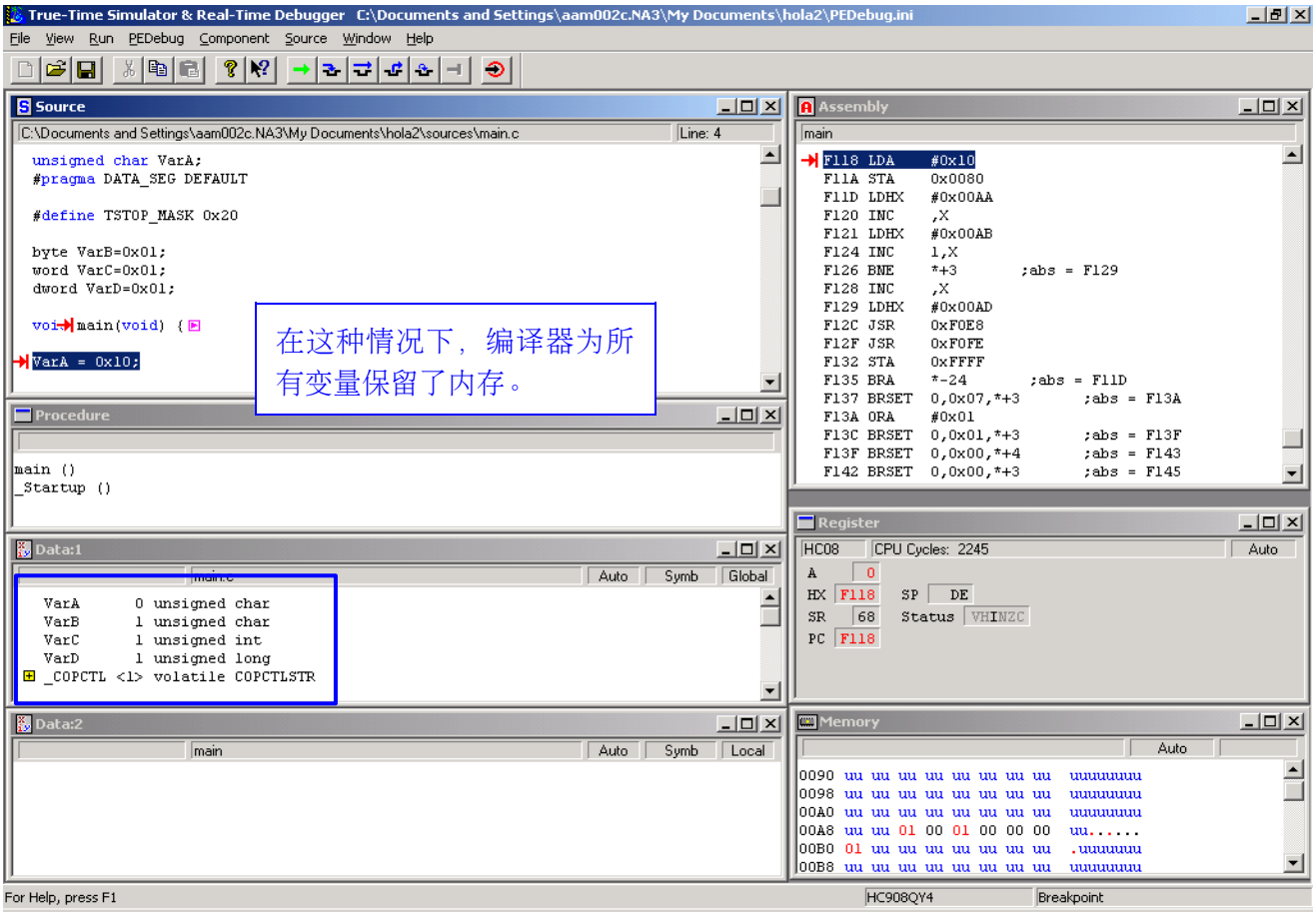
定义函数内数据类型和不指定地址的变量

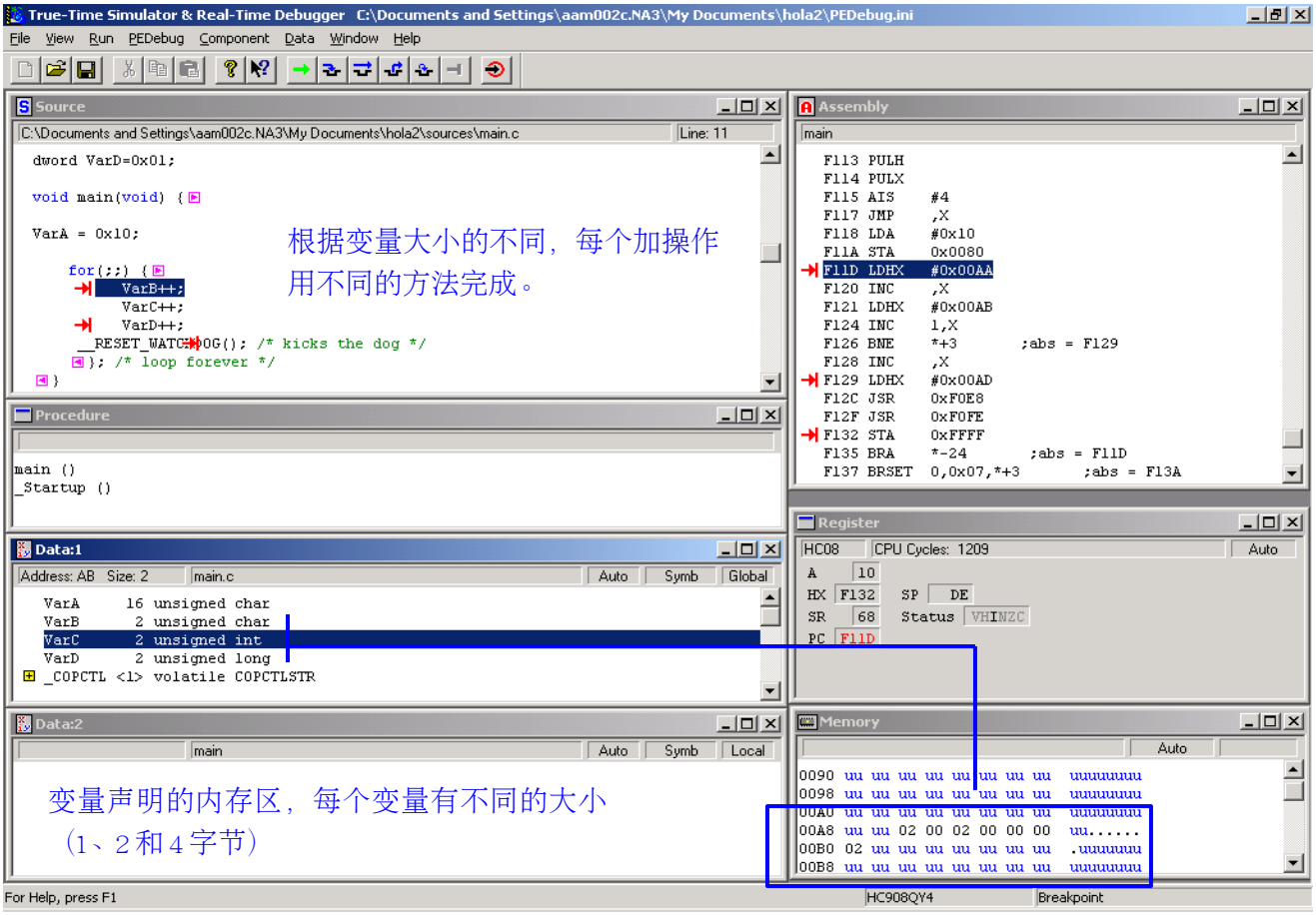
变量在堆栈中有一个地址。


Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help
main.c
#include <MC68HC908QT4.h> /* include peripheral dec
#pragma DATA_SEG BUFFER
unsigned char VarA;
#pragma DATA_SEG DEFAULT
#define TSTOP_MASK 0x20
byte VarB=0x01;
word VarC=0x01;
dword VarD=0x01;
主函数外定义了三个不同类型的变量。
void main(void) {
VarA = 0x10;
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
Line 14 Col 1
MC68HC908QT4.h
** http : www.processorexpert.com
** mail : info@processorexpert.com
**
** ###################################
*/
/*Types definition*/
#ifndef _MC68HC908QT4_H
#define _MC68HC908QT4_H
typedef unsigned char bool;
typedef unsigned char byte;
typedef unsigned int word;
typedef unsigned long dword;
typedef unsigned long dlong[2];
#define __RESET_WATCHDOG() {asm sta COPCTL;} /* ju
#pragma MESSAGE DISABLE C1106 /* WARNING C1106:
/* Based on CPU DB MC68HC908QT4, version 2.87.081 */
/*** PTA - Port A Data Register ***/
typedef union {
byte Byte;
struct {
Line 48 Col 29


True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Data Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 11
dword VarD=0x01;
void main(void) {
VarA = 0x10;
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
编译器为作用的变量保留了内存。本例中 VarA 是唯一使用的变量。
Assembly
main
F0B5 BRSET 0,0x00,*+3 ;abs = F0B8
F0B8 BRSET 0,0x00,*+3 ;abs = F0BB
F0BB LDA #0x10
F0BD STA 0x0080
F0C0 STA 0xFFFF
F0C3 BRA *-3 ;abs = F0C0
F0C5 BRSET 0,0x00,*+3 ;abs = F0C8
F0C8 BRSET 0,0x00,*+3 ;abs = F0CB
F0CB BRSET 0,0x00,*+3 ;abs = F0CE
Register
HC08 CPU Cycles: 278 Auto
A 0
HX F0BB SP D7
SR 6C Status VHINZC
PC F0BB
Procedure
main ()
_Startup ()
Data:1
Address: 0 Size: 4 main.c Auto Symb Global
VarA 0 unsigned char
VarB undefined unsigned char
VarC undefined unsigned int
VarD undefined unsigned long
_COPCTL <1> volatile COPCTLSTR
Memory
Auto
0080 00 uu uu uu uu uu uu uu .uuuuuuu
0088 uu uu uu uu uu uu uu uu uuuuuuuu
0090 uu uu uu uu uu uu uu uu uuuuuuuu
0098 uu uu uu uu uu uu uu uu uuuuuuuu
00A0 uu uu uu uu uu uu uu uu uuuuuuuu
00A8 uu uu uu uu uu uu uu uu uuuuuuuu
Data:2
main Auto Symb Local
Command
Preset breakpoint encountered.
STARTED
RUNNING
Breakpoint
in>
For Help, press F1
HC908QY4
Breakpoint


Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help
main.c
#include <MC68HC908QT4.h> /* include peripheral dec
#pragma DATA_SEG BUFFER
unsigned char VarA;
#pragma DATA_SEG DEFAULT
#define TSTOP_MASK 0x20
byte VarB=0x01;
word VarC=0x01;
dword VarD=0x01;
void main(void) {
VarA = 0x10;
for(;;) {
VarB++;
VarC++;
VarD++;
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
Line 20 Col 18
所有声明的全局变量均被使用。
MC68HC908QT4.h
** http : www.processorexpert.com
** mail : info@processorexpert.com
**
** ###################################
*/
/*Types definition*/
#ifndef _MC68HC908QT4_H
#define _MC68HC908QT4_H
typedef unsigned char bool;
typedef unsigned char byte;
typedef unsigned int word;
typedef unsigned long dword;
typedef unsigned long dlong[2];
#define __RESET_WATCHDOG() {asm sta COPCTL;} /* ju
#pragma MESSAGE DISABLE C1106 /* WARNING C1106:
/* Based on CPU DB MC68HC908QT4, version 2.87.081 */
/*** PTA - Port A Data Register ***/
typedef union {
byte Byte;
struct {
Line 48 Col 29


True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Source Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 4
unsigned char VarA;
#pragma DATA_SEG DEFAULT
#define TSTOP_MASK 0x20
byte VarB=0x01;
word VarC=0x01;
dword VarD=0x01;
void main(void) {
VarA = 0x10;
在这种情况下，编译器为所有变量保留了内存。
Assembly
main
F118 LDA #0x10
F11A STA 0x0080
F11D LDHX #0x00AA
F120 INC ,X
F121 LDHX #0x00AB
F124 INC 1,X
F126 BNE *+3 ;abs = F129
F128 INC ,X
F129 LDHX #0x00AD
F12C JSR 0xF0E8
F12F JSR 0xF0FE
F132 STA 0xFFFF
F135 BRA *-24 ;abs = F11D
F137 BRSET 0,0x07,*+3 ;abs = F13A
F13A ORA #0x01
F13C BRSET 0,0x01,*+3 ;abs = F13F
F13F BRSET 0,0x00,*+4 ;abs = F143
F142 BRSET 0,0x00,*+3 ;abs = F145
Procedure
main ()
_Startup ()
Register
HC08 CPU Cycles: 2245 Auto
A 0
HX F118 SP DE
SR 68 Status VHINZC
PC F118
Data:1
Auto Symb Global
VarA 0 unsigned char
VarB 1 unsigned char
VarC 1 unsigned int
VarD 1 unsigned long
_COPCTL <1> volatile COPCTLSTR
Data:2
main
Auto Symb Local
Memory
Auto
0090 uu uu uu uu uu uu uu uu uuuuuuuu
0098 uu uu uu uu uu uu uu uu uuuuuuuu
00A0 uu uu uu uu uu uu uu uu uuuuuuuu
00A8 uu uu 01 00 01 00 00 00 uu......
00B0 01 uu uu uu uu uu uu uu .uuuuuuu
00B8 uu uu uu uu uu uu uu uu uuuuuuuu
For Help, press F1
HC908QY4
Breakpoint



2、存贮类修饰符

以下关键字用于声明变量，以指定特定需要或内存中变量存贮的相关条件。

static volatile const

这三个关键字，一起让我们不仅可写出好的代码，而且可写出紧凑的代码。

2．1 静态变量

使用静态有二个主要功能:

第一个最常用的用法是定义一个变量，在函数连续调用期间，变量不会消失。

第二个使用静态的用法是限制变量的范围。在模块级定义时，能被整个模块中所有函数访问，不能被其它函数访问。这非常重要，因为当严格限制全局变量众所周知的问题时，它让我们获得所有全局变量执行性能的好处。因此，如果我们有必须被一些函数频繁访问的数据结构，就应当将函数放入同一模块中，并将结构声明为静态。这样所有函数能够访问而不必通过一个访问函数的上层，同时与数据结构无关的代码禁止访问它。这一技术是一种变通方法，立即可访问变量在小的机器上实质上取得了足够的性能。

声明模块级静态变量（与将其设为全局相反）能取得一些其他潜在的益处。静态变量由于定义，只能被一组特定的函数访问。因此，编译器和连接器能够明智地选择变量在存贮空间的放置。例如，对于静态变量，编译器/连接器也许选择将一个模块中所有静态变量放在连续的区域，这样增加了各种优化机会，例如用简单的增加或减少代替重载。相反，全局变量

在存贮空间的位置通常计划于优化编译器的哈稀算法，这排除了可能的优化。

须着重指出，这些变量不会存贮在堆栈中，因为它们必须保存其值。

下面给出一个静态变量怎样工作的例子：

FILE1.c

```
#include <FILE2.h>     //包含文件FILE2.c
                       //中的函数
void main (void){
//第一次进入MyFunction之前，myVar=0。
MyFunction();       //在FILE2.c中
//第二次进入MyFunction之前，myVar=1。
MyFunction();       //在FILE2.c中

}
```

FILE2.c

```
void MyFunction (void){  //FILE2.C中定义
                         //MyFunction函数
static char myVar = 0;  //本地变量
                        //声明为static
myVar = myVar + 1;   //尽管myVar是本地变量，但它保持了自己的值。
}
```

2．2 静态函数

一个静态函数只能被其所在模块中的其它函数调用。使用静态函数是结构化编程的好习惯。你也许惊讶地知道静态函数能产生小/快的代码。这是可能的，因为编译器在编译时确切地知道什么函数能调用一个给定的静态函数。因此，函数的相关内存区域能被调整，以致使用调用的一个短版本或跳转指令。潜在的改进甚至更好，编译器足够聪明地用跳转代替调用。

2．3 关键字“static”的使用

在函数体声明静态的变量，在函数调用期间保持其质；

在模块内声明静态的变量，（但在函数体之外）能被模块内所有函数访问；

在模块内声明静态的函数，只能被模块内其它函数调用。

对于嵌入式系统：封装持续生存的数据（包装）；模块化编码（数据隐藏）；在每个模块中隐藏内部处理。

2．4 可变（volatile）变量

可变变量是其值在正常程序流程以外可能改变的变量。在嵌入式系统中，这种情况通过两种主要途径发生：

通过一个中断服务程序，或作为硬件动作的结果。例如，通过一个串口接收到一个字符，

结果串口状态寄存器更新，这完全在程序流程之外发生。很多程序员知道编译器不会试图优化一个volatile寄存器，而宁可每次重载它。

在嵌入式设备中，将所有外设寄存器声明为*volatile*是一个好习惯。

许多编译器供应商经常炫耀他们的代码优化，它们通常非常好，它们有些根本不明显，但能极大地减少周期和内存。但有时我们不想编译器聪明和优化一个部份，因为我们确实需要代码那样作。

我们怎样才能达到呢？那么，访问定义为volatile的变量从不会被编译器优化。

让我们分析一个例子,看看编译器是怎样处理一个volatile和一个非volatile变量…

```
volatile unsigned char PORTA @0x00;
volatile unsigned char SCS1  @0x16;
unsigned char value;
void main(void){
PORTA = 0x05;      /* PORTA = 00000101 */
PORTA = 0x05;      /* PORTA = 00000101 */
SCS1;
value = 10;
}
```

未使用Volatile关键字，编译器将其编译为：

```
MOV   #5,PORTA
LDA   #10
STA   @value
```

使用Volatile关键字后，编译器将其编译为：

```
MOV   #5,PORTA
MOV   #5,PORTA
LDA   SCS1
LDX   #10
STX   @value
```

这段代码实际上不做任何事,但它很好地表达了优化怎样强烈地影响程序的结果。在main()中连续两次使用语句：PORTA=5，这没有意义，但让我们假设这是正确开发程序所必须的…在这两个语句之后，明显地有一条无意义语句“SCS1;”。让我们看当不使用volatile变量会发生什么…

我们得到了优化过的汇编代码。重复的语句Port A = 5消失了只剩下一句“move #5 to Port A”。语句“SCS1;”似乎什么都不做，因此聪明的编译器将它消去了。最后，将10加载到累加器并作为值存贮。

使用volatile关键字声明PORTA 和SCS1,得到的汇编代码没有优化，连续两次在Port A写入数值5，然后将SCS1加载到累加器。最后由于累加器被使用，于是用X寄存器存贮数值10。

好了,连续两次用数值5写PortA，假设这是需要这样做，但是加载SCS1到累加器有一个很有意义的值。这是串行通信接口SCI需要的，读SCS1寄存器目的是清除任何未决的标志。无意义的语句“SCS1;”被翻译为读寄存器的的汇编语句，这将清除SCI中未决的标志。

前面说过，在嵌入式设备中将所有外设寄存器声明为***volatile***是一个好习惯。在分开的头文件中定义所有外设的名字，能使所写代码更友好并使迁移简化。下面这个例子用volatile变量声明所有寄存器，这样做较妥当，因为任何这些寄存器能在任何时候在程序流程之外被修改。

```
/* MC68HC908GP20/32  Official Peripheral Register Names */
volatile unsigned char PORTA     @0x0000; /* Ports and data direction */
volatile unsigned char PORTB     @0x0001;
volatile unsigned char PORTC     @0x0002;
volatile unsigned char PORTD     @0x0003;
volatile unsigned char PORTE     @0x0008;
volatile unsigned char DDRA      @0x0004; /* Data Direction Registers */
volatile unsigned char DDRB      @0x0005;
volatile unsigned char DDRC      @0x0006;
volatile unsigned char DDRD      @0x0007;
volatile unsigned char DDRE      @0x000C;
volatile unsigned char PTAPUE    @0x000D; /* Port pull-up enables */
volatile unsigned char PTCPUE    @0x000E;
volatile unsigned char PTDPUE    @0x000F;
```

2．5 Const变量

关键字“const”，C语言中命名最差的关键字，并不表示恒量，而是代表“只读”。在嵌入式系统中，有很大的不同，这一会应会明白。

Const声明可用于任何变量，它告诉编译器将其存贮在ROM代码。编译器保留了那个位置程序存贮器地址。由于位于ROM中，其值不能改变。

由于它作为常量工作，必须赋一初值。如：const double PI = 3.14159265;

Const 变量与明显的常数相对，很多原文要求用const变量代替明显的常数。例如：

用const unsigned char channels = 8;代替#define CHANNELS 8 。

本方法的基本原理是在调试器内部，你能检查一个const变量，然而一个明显的常数不可访问。不幸的是，在很多8位机上你将为这一好处付出极大的代价。这两个主要代价是：

- 一些编译器在RAM中创建一个真实的变量来支持cost变量，这是一个极大的惩罚。
- 一些编译器如CodeWarrior，知道变量为const，将把变量存贮在ROM中。无论怎样，变量仍作为变量处理和访问，典型地用某些变址寻址（16位）的方式。与直接寻址（8位）方式相比，这种方法通常很慢。

Const的用法：

const unsigned short a;

```
unsigned short const a;
const unsigned short *a;
unsigned short * const a;
```

2．6 Const volatile 变量

现在讨论一个深奥的问题，一个变量既能是常量，又能是可变量吗？如果是这样，这意味什么，怎样使用？答案是“能”。

这个修饰符应该用于能出乎意料地改变的任何存贮器位置，因此需要volatile限定语，由于const该变量是只读的。

最明显的例子是硬件状态寄存器，象SCI状态寄存器SCS1。这个寄存器包含信号状态标志，如发送空、发送完成、接收满以及其它。这是一个可变寄存器由于这些标志的改变依赖于串行通信的状态，这也是只读，由于标志不能被程序直接改写，它们只对模块的状态作出响应。这个状态寄存器最佳声明方法是：

```
const volatile unsigned char SCS1 @0x0016
```

3、资源映射

3．1 访问固定内存位置

嵌入式系统通常的特点是需要编程者访问一个指定的存贮器位置。

练习：在某个项目中需要将绝对地址0xFFA处整型变量的值设为0xAA55（编译器为纯粹的ANSI编译器）。完成这个任务的代码是：

```
Int * ptr;
ptr = (int *)0x2FFA;
*ptr = 0xAA55;
```

3．2 怎样访问I/O寄存器

在嵌入式领域，设备如微控制器有片上资源，应当被管理和访问。很多I/O和控制寄存器位于直接页，它们应如此声明，因此在可能时编译器能直接寻址。它们有定位的地址，但问题是它们不是存储器，那么怎样访问这些I/O寄存器呢？这是一个非常重要的问题，答案比你想的简单或者复杂。

一个普照通而有用的形式是使用如下的#define指示：

```
#define PortA （ *（ volatile unsigned char * ） 0x0000 ）
```

这构成了I/O寄存器，这种情况下，Port A为地址0x0000处字符型变量。#define实际做的是每次发现PortA时放置一个构件。也就是说在代码中写：PortA = 0x3F，实际做的就是告诉编译器0x0000是一个volatile-unsigned-char类型的指针，它的内容等于0x3F。

糊涂吗?有点…让我们看一些其它选择：

这样做的一个容易的方法是在变量声明中使用符号“@”，创建一个语句读作：在地址0x0000处创建一个volatile-unsigned-char型的变量PortA。

这是一个编译器特定的语法，它可读性高，但失去了兼容性。无论什么时候我们决定使用一个不同的编译器去编译该代码，也许会发现@不被识别。CodeWarrior和Cosmic包含了这个特殊语法。

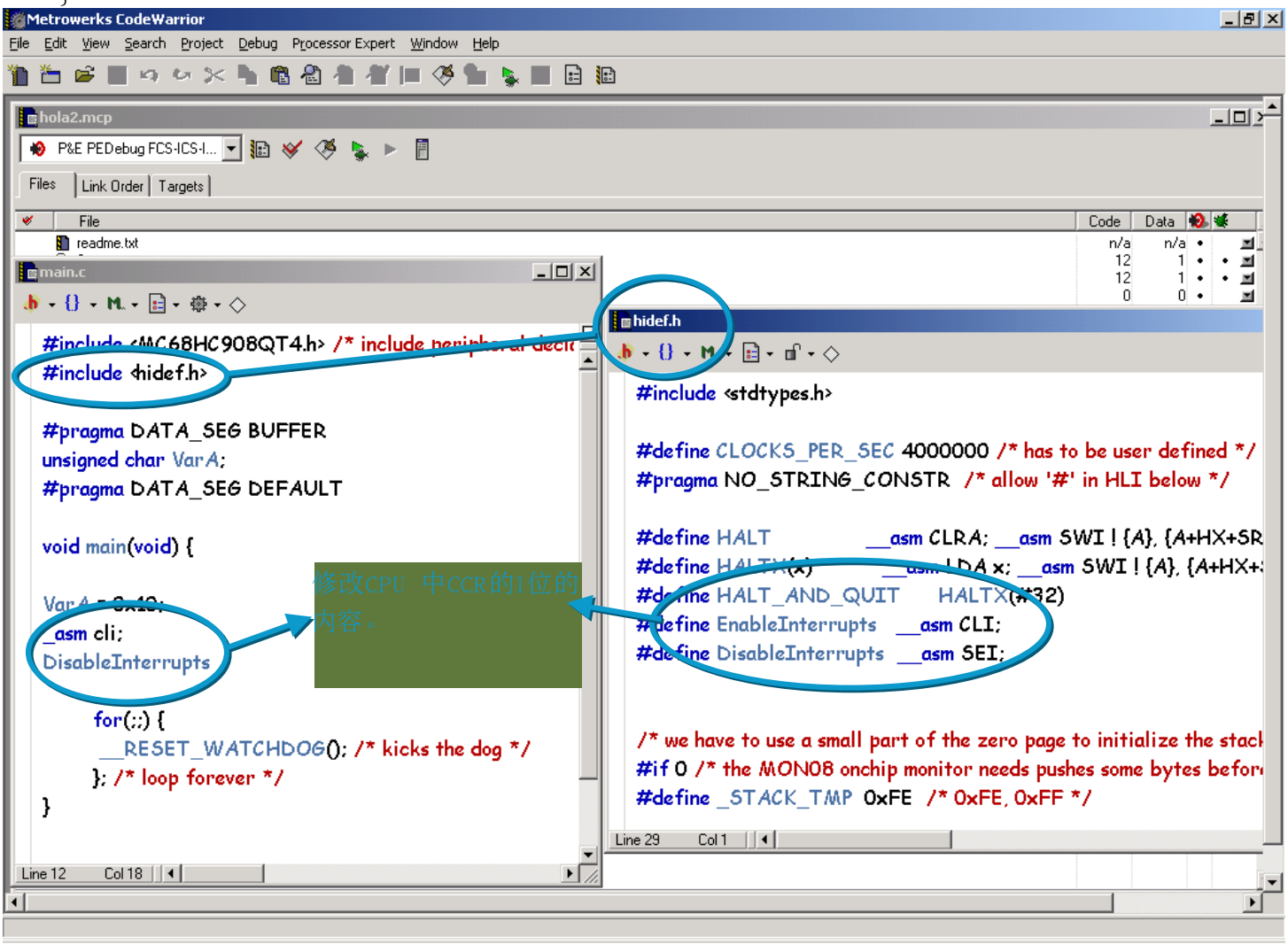
CPU中的寄存器没有内存映射；指令集包含允许它们自修改的子集；C不提供直接访问寄存器的工具；C编译器允许在C代码中使用汇编指令，如：

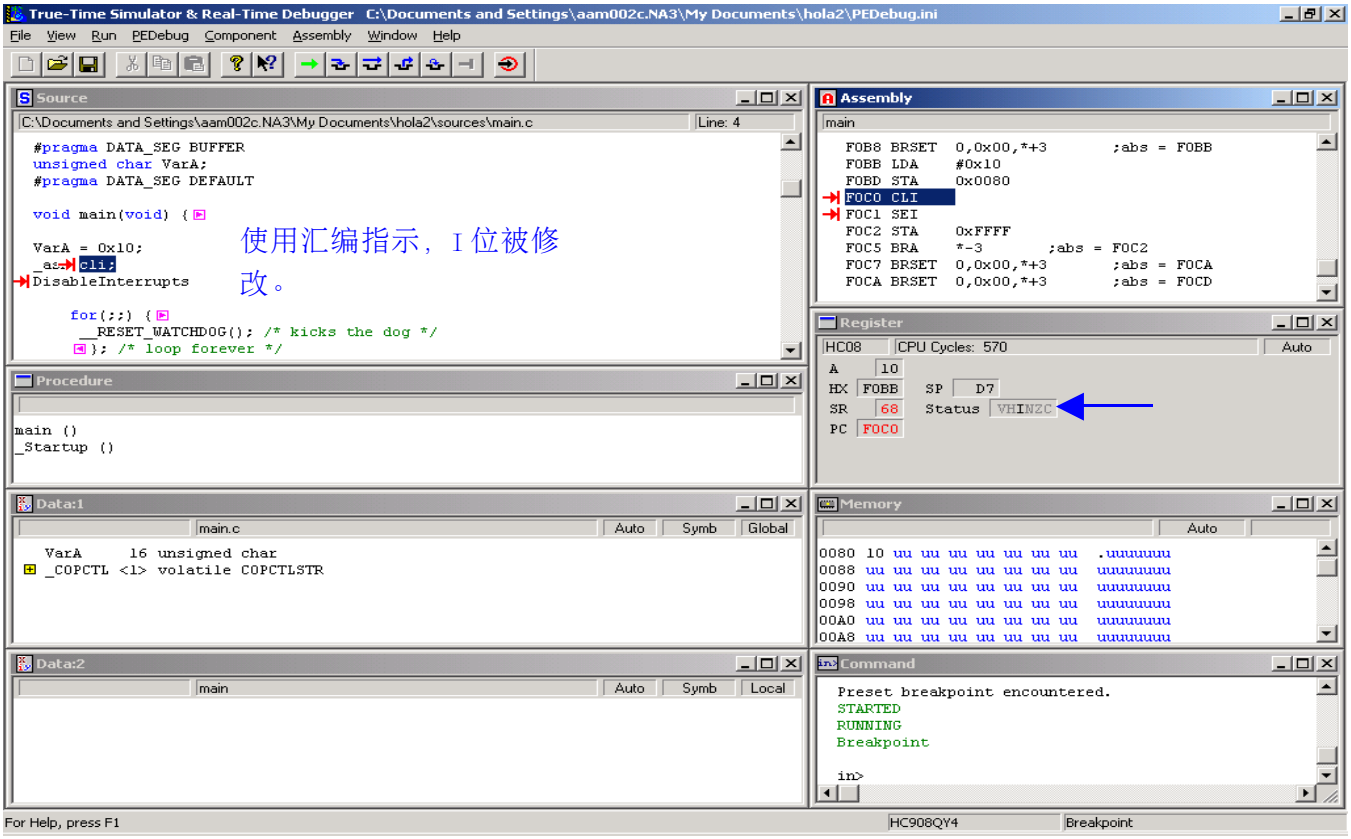
1）_asm AssemblyInstuction;

2）asm (AssemblyInstruction);

3）asm {

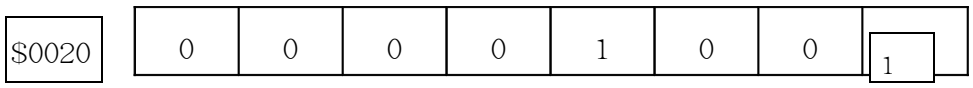
}





3．3 位域

在嵌入系统中,在一个给定的地址,一次能访问和修改一位或几位。



完成这个任务，在C语言中有不同的方法达到和实现。

Address: $0020

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	TOF	TOIE	TSTOP	0	0	PS2	PS1	PS0
Write:	0			TRST				
Reset:	0	0	1	0	0	0	0	0

= Unimplemented

Figure 10-4. TIM Status and Control Register (TSC)

*位结构:

效率随编译器的不同而改变；跨编译器和目标不能移植。

*位类型:

不能移植（标准C语言中没有）；如

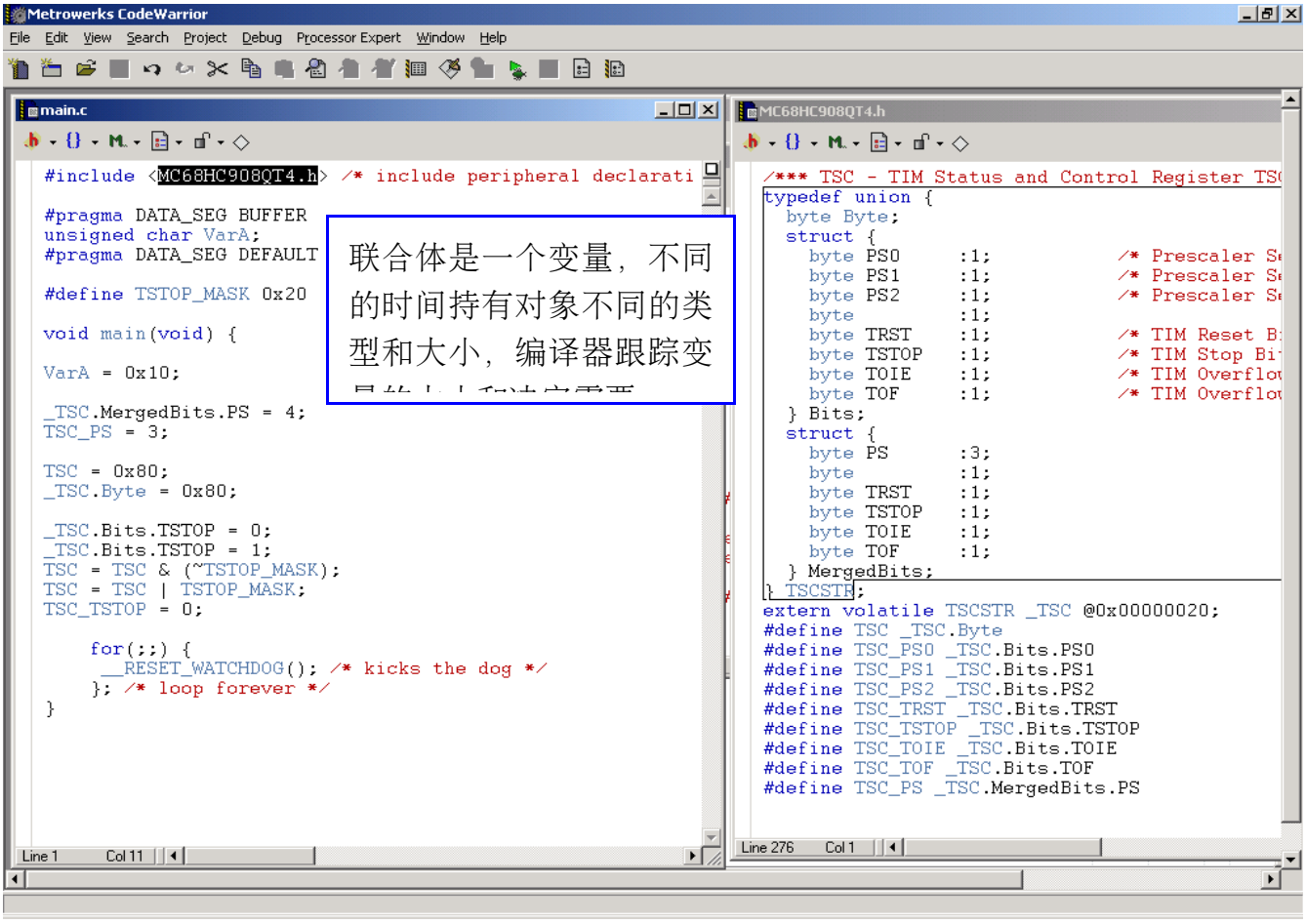
当使用时可提高代码的效率。

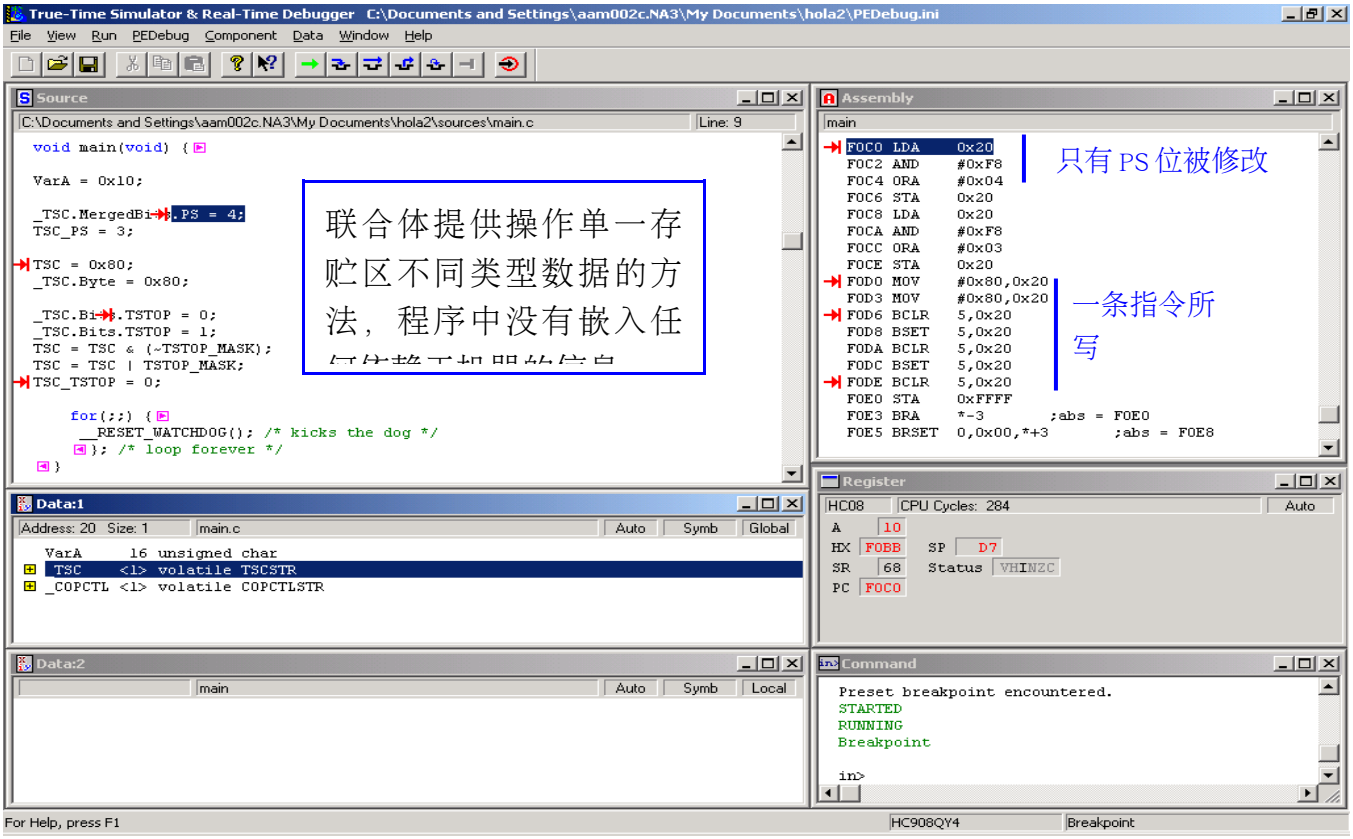
*移位和掩模

可移植，适当的效率；经常优化为位操作。

如果定义一个结构，但所有变量重叠在同一内存的开始位置，你应该使用联合体。联合体允许引用在联合体中定义的以任何形式描述的数据字节。联合体在内存中的尺寸大小为联合体中所列的最大类型的大小。点操作符用于选择需要的成员。

打开文件：Lab2-BitFields.mcp





3．4 数组

C允许程序员用几种不同方法存取数组的内容。

Unsigned char Array[]={0xAA,0xBB,0xCC};

依赖于执行，选择最适合于该应用的需要，将产生快而小的代码。数组访问方法：

1）硬编码：

Array[0]=12*UNIT_VOLTS;

编译时决定地址，执行速度快。

2）变址增加

Array[index++]=12*UNIT_VOLTS;

快速，比硬编码灵活。

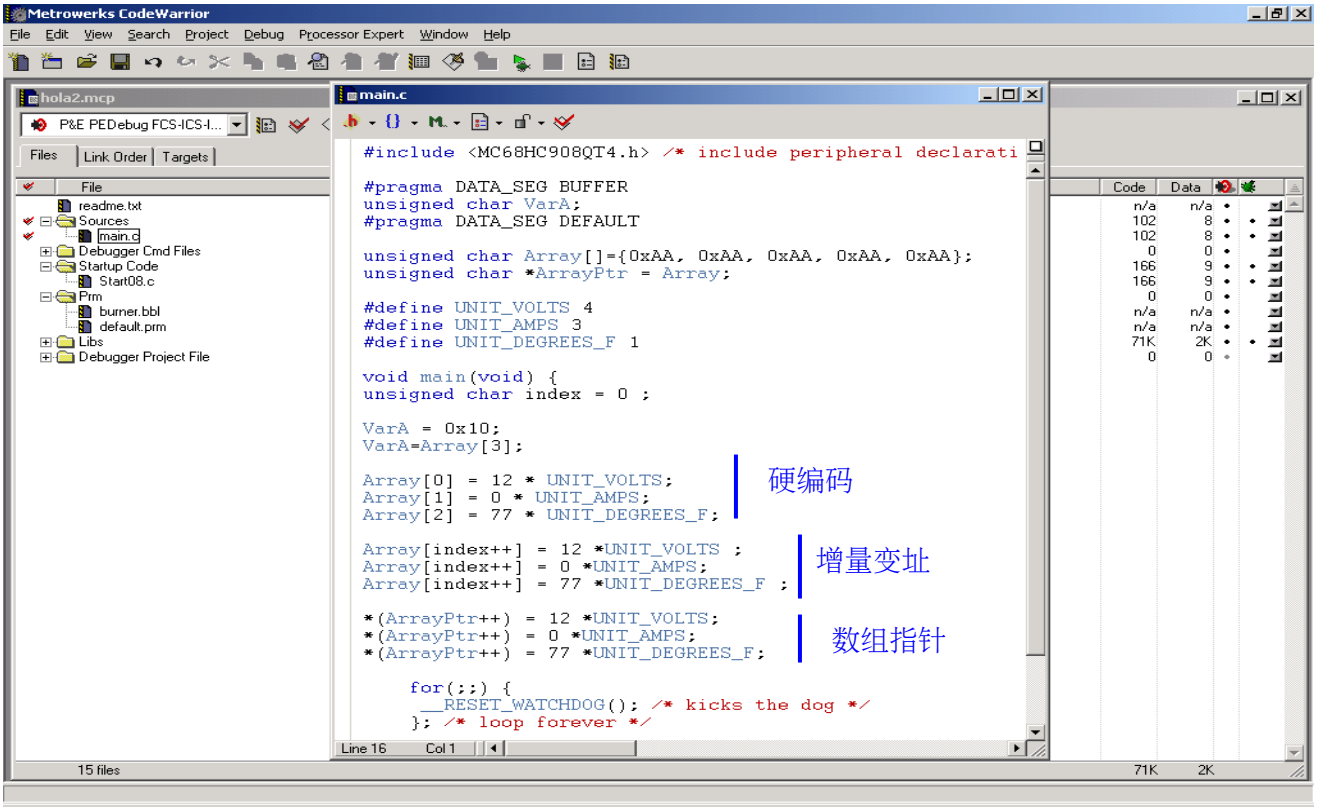
3）数组指针

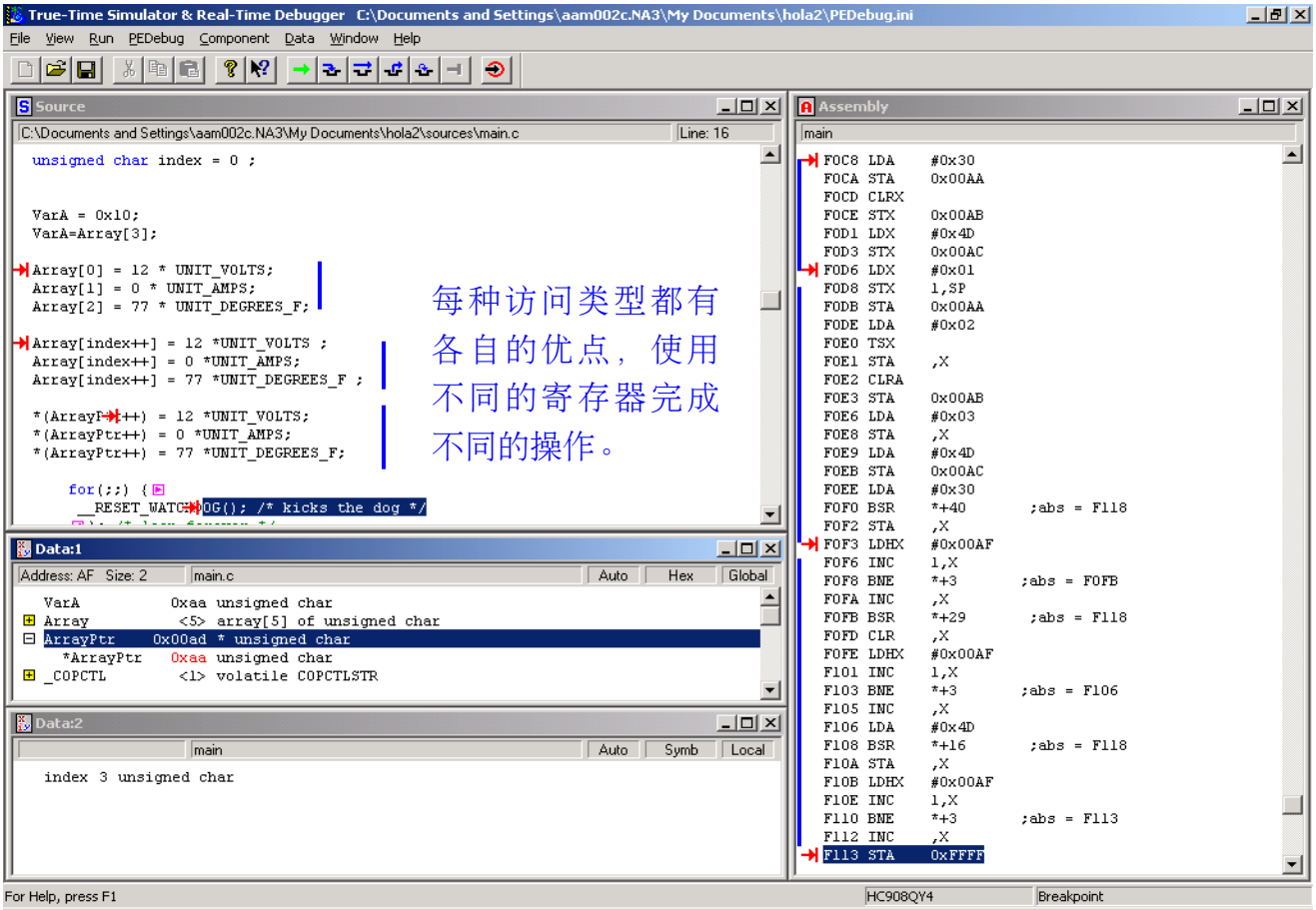
*(ArrayPtr++)=12*UNIT_VOLTS;

执行速度快，可读性差，可和循环一起使用。

如下图所示：

打开文件：Lab3-Arrays.mcp


Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help
hola2.mcp
P&E PEDebug FCS-ICS-I...
Files Link Order Targets
File
readme.txt
Sources
main.c
Debugger Cmd Files
Startup Code
Start08.c
Prm
burner.bbl
default.prm
Libs
Debugger Project File
15 files
main.c
#include <MC68HC908QT4.h> /* include peripheral declarati
#pragma DATA_SEG BUFFER
unsigned char VarA;
#pragma DATA_SEG DEFAULT
unsigned char Array[]={0xAA, 0xAA, 0xAA, 0xAA, 0xAA};
unsigned char *ArrayPtr = Array;
#define UNIT_VOLTS 4
#define UNIT_AMPS 3
#define UNIT_DEGREES_F 1
void main(void) {
unsigned char index = 0 ;
VarA = 0x10;
VarA=Array[3];
Array[0] = 12 * UNIT_VOLTS;
Array[1] = 0 * UNIT_AMPS;
Array[2] = 77 * UNIT_DEGREES_F;
硬编码
Array[index++] = 12 *UNIT_VOLTS ;
Array[index++] = 0 *UNIT_AMPS;
Array[index++] = 77 *UNIT_DEGREES_F ;
增量变址
*(ArrayPtr++) = 12 *UNIT_VOLTS;
*(ArrayPtr++) = 0 *UNIT_AMPS;
*(ArrayPtr++) = 77 *UNIT_DEGREES_F;
数组指针
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
Line 16 Col 1
Code Data
71K 2K


True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Data Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 16
unsigned char index = 0 ;
VarA = 0x10;
VarA=Array[3];
Array[0] = 12 * UNIT_VOLTS;
Array[1] = 0 * UNIT_AMPS;
Array[2] = 77 * UNIT_DEGREES_F;
Array[index++] = 12 *UNIT_VOLTS ;
Array[index++] = 0 *UNIT_AMPS;
Array[index++] = 77 *UNIT_DEGREES_F ;
*(ArrayPtr++) = 12 *UNIT_VOLTS;
*(ArrayPtr++) = 0 *UNIT_AMPS;
*(ArrayPtr++) = 77 *UNIT_DEGREES_F;
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
每种访问类型都有各自的优点，使用不同的寄存器完成不同的操作。
Assembly
main
F0C8 LDA #0x30
F0CA STA 0x00AA
F0CD CLRX
F0CE STX 0x00AB
F0D1 LDX #0x4D
F0D3 STX 0x00AC
F0D6 LDX #0x01
F0D8 STX 1,SP
F0DB STA 0x00AA
F0DE LDA #0x02
F0E0 TSX
F0E1 STA ,X
F0E2 CLRA
F0E3 STA 0x00AB
F0E6 LDA #0x03
F0E8 STA ,X
F0E9 LDA #0x4D
F0EB STA 0x00AC
F0EE LDA #0x30
F0F0 BSR *+40 ;abs = F118
F0F2 STA ,X
F0F3 LDHX #0x00AF
F0F6 INC 1,X
F0F8 BNE *+3 ;abs = F0FB
F0FA INC ,X
F0FB BSR *+29 ;abs = F118
F0FD CLR ,X
F0FE LDHX #0x00AF
F101 INC 1,X
F103 BNE *+3 ;abs = F106
F105 INC ,X
F106 LDA #0x4D
F108 BSR *+16 ;abs = F118
F10A STA ,X
F10B LDHX #0x00AF
F10E INC 1,X
F110 BNE *+3 ;abs = F113
F112 INC ,X
F113 STA 0xFFFF
Data:1
Address: AF Size: 2
main.c
Auto Hex Global
VarA 0xaa unsigned char
Array <5> array[5] of unsigned char
ArrayPtr 0x00ad * unsigned char
*ArrayPtr 0xaa unsigned char
_COPCTL <1> volatile COPCTLSTR
Data:2
main
Auto Symb Local
index 3 unsigned char
For Help, press F1
HC908QY4
Breakpoint

3．5 函数指针

函数指针与数据指针一样有用处，原因如下：

当你想要一个额外级别的间接时；

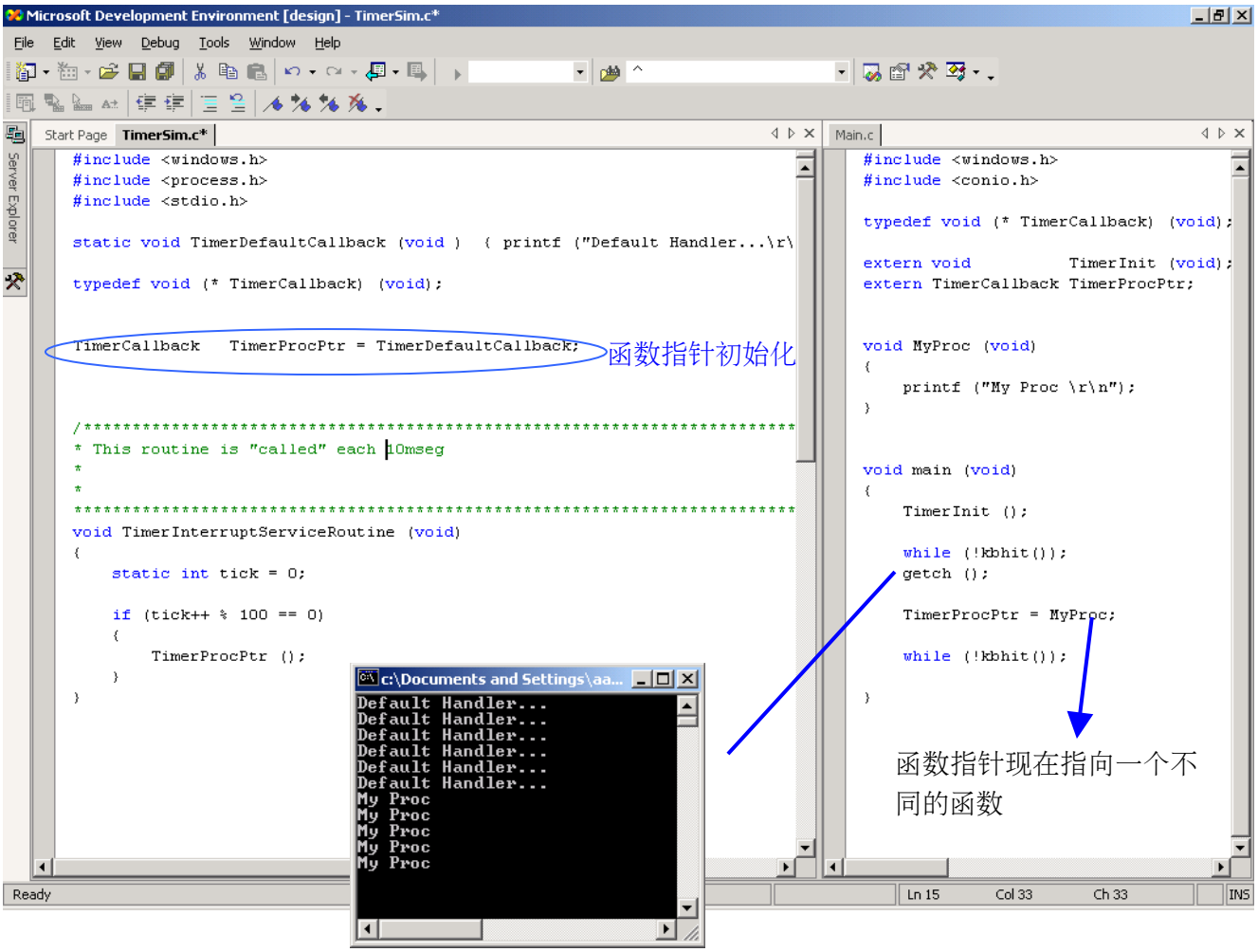
当你想用同一段代码依环境的不同调用不同的函数。

下面的代码定义了一个指向函数的指针，带了一个整型参数并返回一整数：

```
int (*function)(int);
```

(*function)周围的圆括符是必须的，因为定义中的优先关系。没有它们，我们则定义了一个函数返回一个整型指针。

例如：



下面举一个HC08QL的例子：

Table 14-3. Interrupt Sources Summary (BTM = 0)

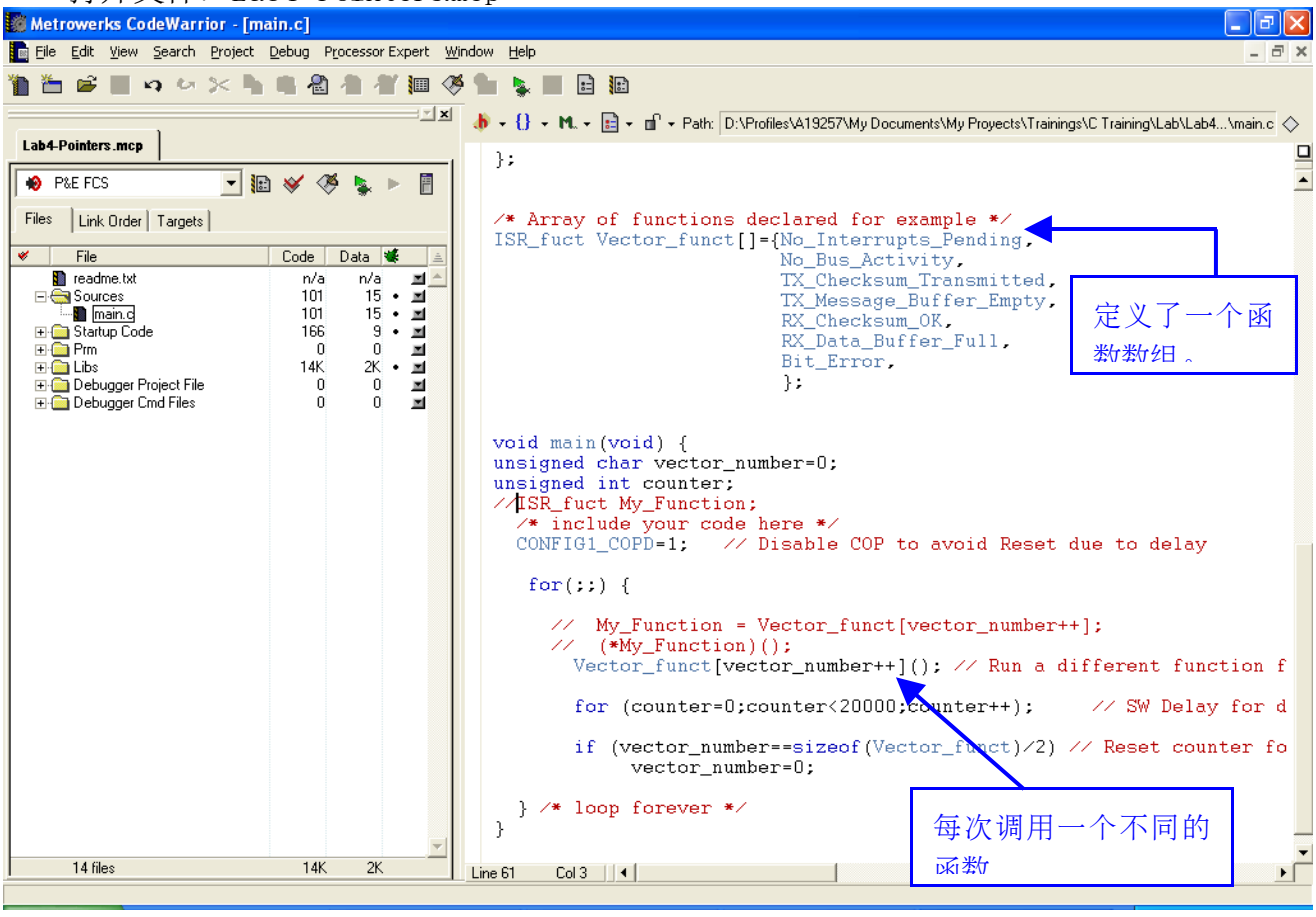
SLCSV	I3	I2	I1	I0	Interrupt Source	Priority
$00	0	0	0	0	No Interrupts Pending	0 (Lowest)
$04	0	0	0	1	No-Bus-Activity	1
$08	0	0	1	0	TX Message Buffer Empty Checksum Transmitted	2
$0C	0	0	1	1	TX Message Buffer Empty	3
$10	0	1	0	0	RX Message Buffer Full Checksum OK	4
$14	0	1	0	1	RX Data Buffer Full No Errors	5
$18	0	1	1	0	Bit-Error	6
$1C	0	1	1	1	Receiver Buffer Overrun	7
$20	1	0	0	0	(RESERVED)	8
$24	1	0	0	1	Checksum Error	9
$28	1	0	1	0	Byte Framing Error	10
$2C	1	0	1	1	Identifier Received Successfully	11
$30	1	1	0	0	Identifier Parity Error	12
$34	1	1	0	1	Inconsistent-Synch-Field-Error	13
$38	1	1	1	0	Reserved	14
$3C	1	1	1	1	Wakeup	15 (Highest)

SLIC模块仅有一个中断；用户必须读SLIC中断向量寄存器（SLCSV）来核实中断源。

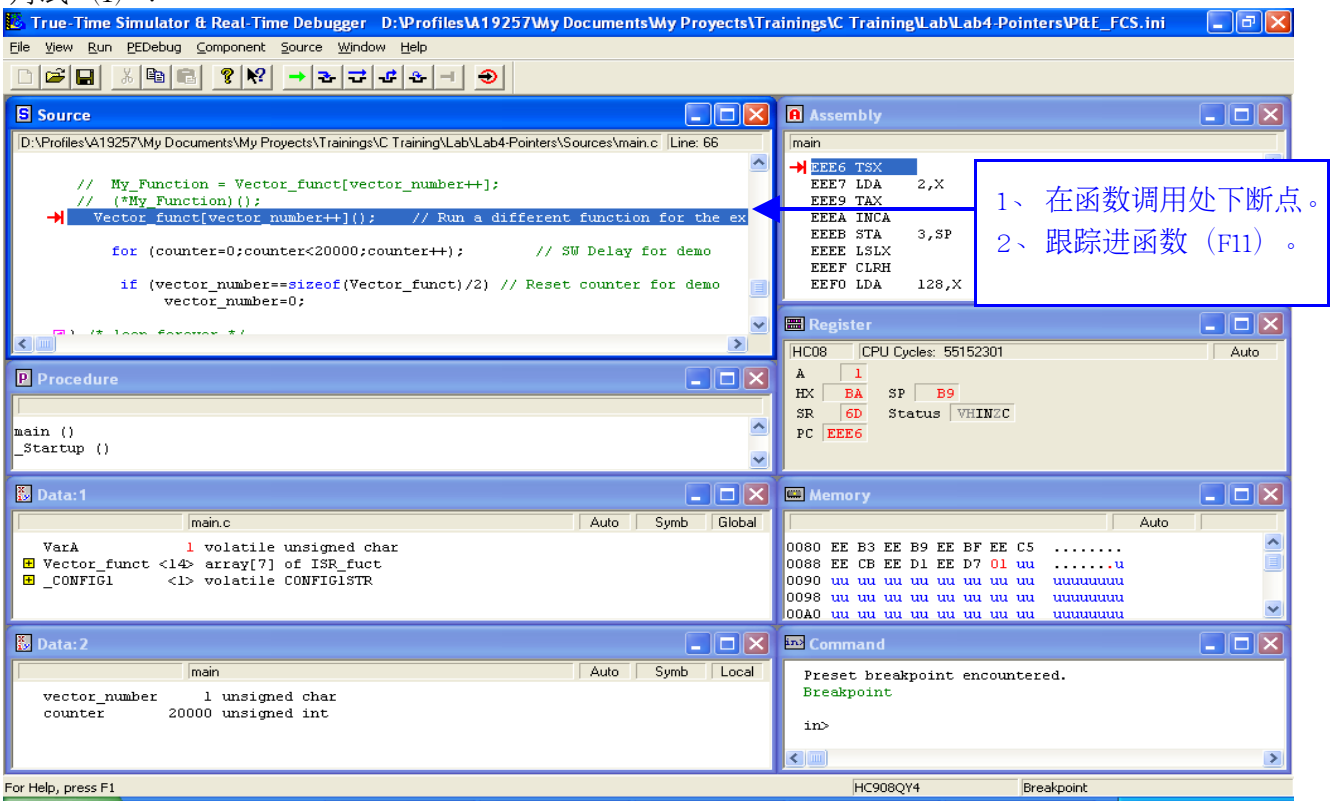
可能的解决方案：

switch 语句；嵌套的if语句；函数指针。

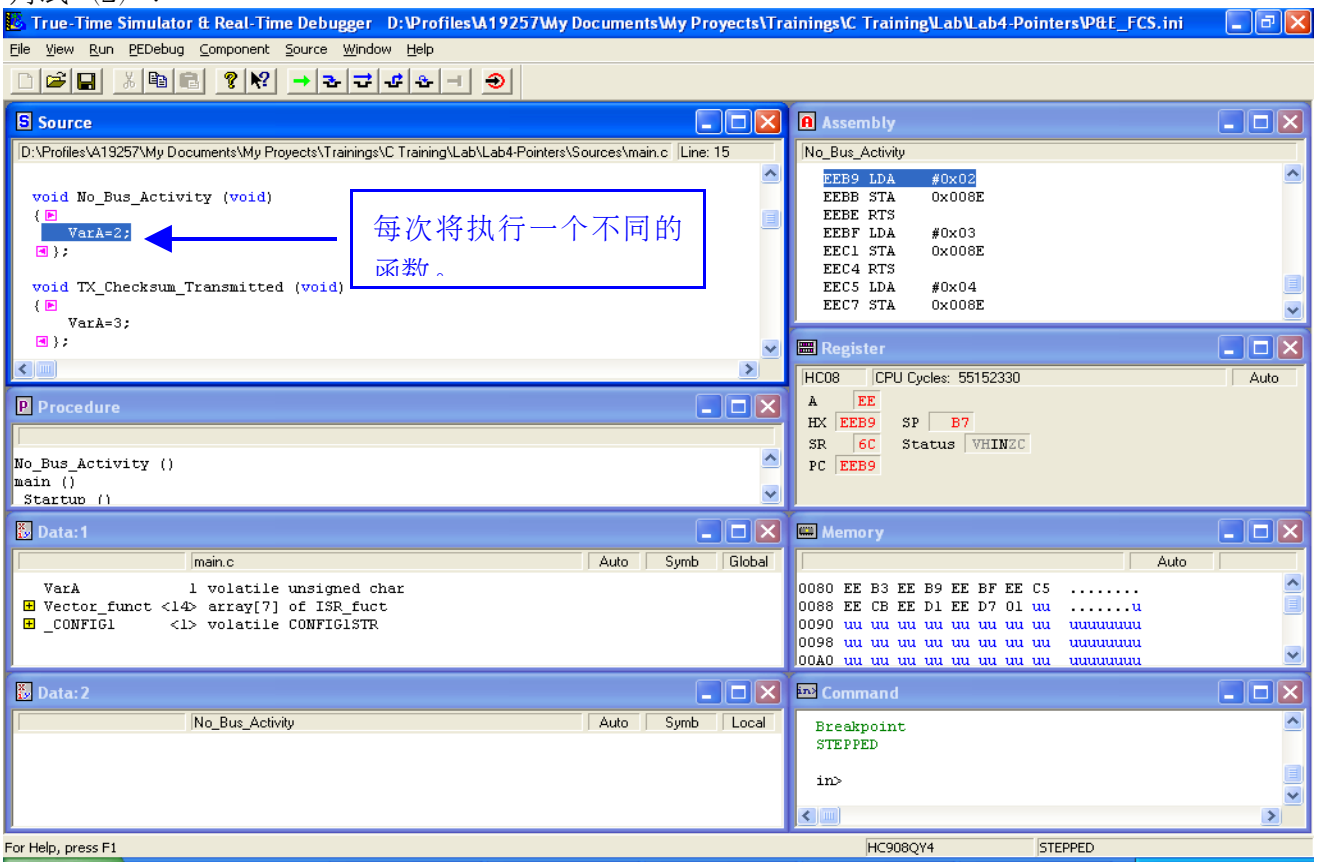
打开文件：Lab4-Pointers.mcp



调试（1）：


True-Time Simulator & Real-Time Debugger D:\Profiles\A19257\My Documents\My Proyects\Trainings\C Training\Lab\Lab4-Pointers\P&E_FCS.ini
File View Run PEDebug Component Source Window Help
Source
D:\Profiles\A19257\My Documents\My Proyects\Trainings\C Training\Lab\Lab4-Pointers\Sources\main.c Line: 66
// My_Function = Vector_funct[vector_number++];
// (*My_Function)();
Vector_funct[vector_number++](); // Run a different function for the ex
for (counter=0;counter<20000;counter++); // SW Delay for demo
if (vector_number==sizeof(Vector_funct)/2) // Reset counter for demo
vector_number=0;
Assembly
main
EEE6 TSX
EEE7 LDA 2,X
EEE9 TAX
EEEA INCA
EEEB STA 3,SP
EEEE LSLX
EEEF CLRH
EEF0 LDA 128,X
1、 在函数调用处下断点。
2、 跟踪进函数（F11）。
Register
HC08 CPU Cycles: 55152301 Auto
A 1
HX BA SP B9
SR 6D Status VHINZC
PC EEE6
Procedure
main ()
_Startup ()
Data:1
main.c Auto Symb Global
VarA 1 volatile unsigned char
Vector_funct <14> array[7] of ISR_fuct
_CONFIG1 <1> volatile CONFIG1STR
Memory
Auto
0080 EE B3 EE B9 EE BF EE C5 ....
0088 EE CB EE D1 EE D7 01 uu ....u
0090 uu uu uu uu uu uu uu uu uuuuuuuu
0098 uu uu uu uu uu uu uu uu uuuuuuuu
00A0 uu uu uu uu uu uu uu uu uuuuuuuu
Data:2
main Auto Symb Local
vector_number 1 unsigned char
counter 20000 unsigned int
Command
Preset breakpoint encountered.
Breakpoint
in>
For Help, press F1
HC908QY4
Breakpoint

调试（2）：

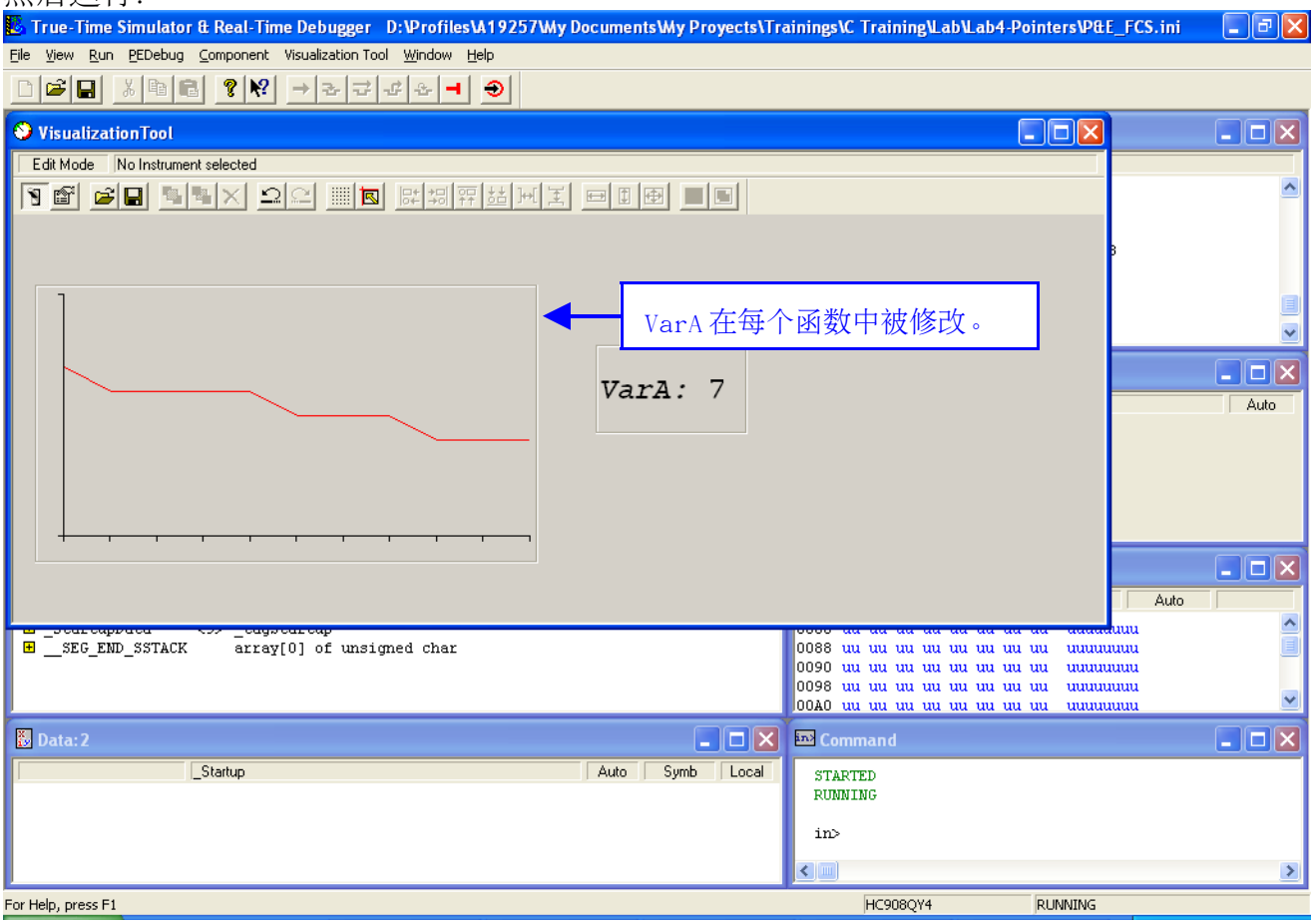

True-Time Simulator & Real-Time Debugger D:\Profiles\A19257\My Documents\My Proyects\Trainings\C Training\Lab\Lab4-Pointers\P&E_FCS.ini
File View Run PEDebug Component Source Window Help
Source
D:\Profiles\A19257\My Documents\My Proyects\Trainings\C Training\Lab\Lab4-Pointers\Sources\main.c Line: 15
void No_Bus_Activity (void)
{
VarA=2;
};
void TX_Checksum_Transmitted (void)
{
VarA=3;
};
每次将执行一个不同的函数。
Assembly
No_Bus_Activity
EEB9 LDA #0x02
EEBB STA 0x008E
EEBE RTS
EEBF LDA #0x03
EEC1 STA 0x008E
EEC4 RTS
EEC5 LDA #0x04
EEC7 STA 0x008E
Register
HC08 CPU Cycles: 55152330 Auto
A EE
HX EEB9 SP B7
SR 6C Status VHINZC
PC EEB9
Procedure
No_Bus_Activity ()
main ()
_Startup ()
Data:1
main.c Auto Symb Global
VarA 1 volatile unsigned char
Vector_funct <14> array[7] of ISR_fuct
_CONFIG1 <1> volatile CONFIG1STR
Memory
Auto
0080 EE B3 EE B9 EE BF EE C5 ....
0088 EE CB EE D1 EE D7 01 uu ....u
0090 uu uu uu uu uu uu uu uu uuuuuuuu
0098 uu uu uu uu uu uu uu uu uuuuuuuu
00A0 uu uu uu uu uu uu uu uu uuuuuuuu
Data:2
No_Bus_Activity Auto Symb Local
Command
Breakpoint
STEPPED
in>
For Help, press F1
HC908QY4
STEPPED

调试（3）：

打开Component->Visuallization Tool

打开Display.vtl

然后运行：



什么时候使用指针：

当使用少量函数时，嵌套的IF语句占用空间少；

Switch语句可读性好，但占空间大；

当很多函数被声明时，指针产生的代码少，但它占用大量的RAM空间。

3．6 栈指针与函数参数

栈指针支持C的关键特性：

在汇编程序和C编译器中，堆栈通常用于给子程序传递变量；

允许使用递归；

是自动变量的基础。

典型地子程序将把需要的操作数放入累加器。堆栈相对寻址允许访问堆栈上的数据，提供直接访问操作数，排除从堆栈压入以及弹出数值所需要的代码和时间。

堆栈指针指令与等份的变址指令相比需要一个额外的字节和一个额外的执行周期。

例如：

```
typedef struct {
```

```
unsigned char        ID;
unsigned short       Time;
} ObjectType;
void foo (unsigned char value) {
volatile ObjectType instance;
instance.ID = value;
}
```

编译后得到:

```
foo:
B00B  A7FB        AIS   #-3
B00D  9EE701      STA   1,SP
B010  A707        AIS   #3
B012  81          RTS
```

3．6．1 堆栈指针寻址

堆栈指针相对寻址进一步增强了C代码的效率。有两种类型:

8位偏移的堆栈指针相对寻址和16位偏移的堆栈指针相对寻址。它们和间址建起方式工作相似，但使用堆栈指针代替H:X变址寄存器。注意当中断不允许时可用堆栈指针作为额外的变址寄存器。

3．6．2 堆栈帧

1) 帧指针

函数通常有一个包含其所有本地数据的堆栈帧。编译器并不设置一个明白的帧指针，但堆栈上的本地数据和参数都根据SP寄存器访问。

2) 入口代码

通常入口代码是一系列为本地变量保留空间的指令:

```
PSHA            ; 仅当有寄存器参数
PSHX            ; 仅当有寄存器参数
AIS  #(-s)      ; 为本地变量保留空间
```

S是函数的本地数据的大小（单位：字节）。没有静态链接，动态链接并没有明白地存储。

3) 出口代码

出口代码从堆栈中移除本地变量，并返回到调用者:

```
AIS    #（t）     ; 移除本地栈空间，包括最终的寄存器参数
RTS               ; 返回调用者
```

3．6．3 HC08返回值

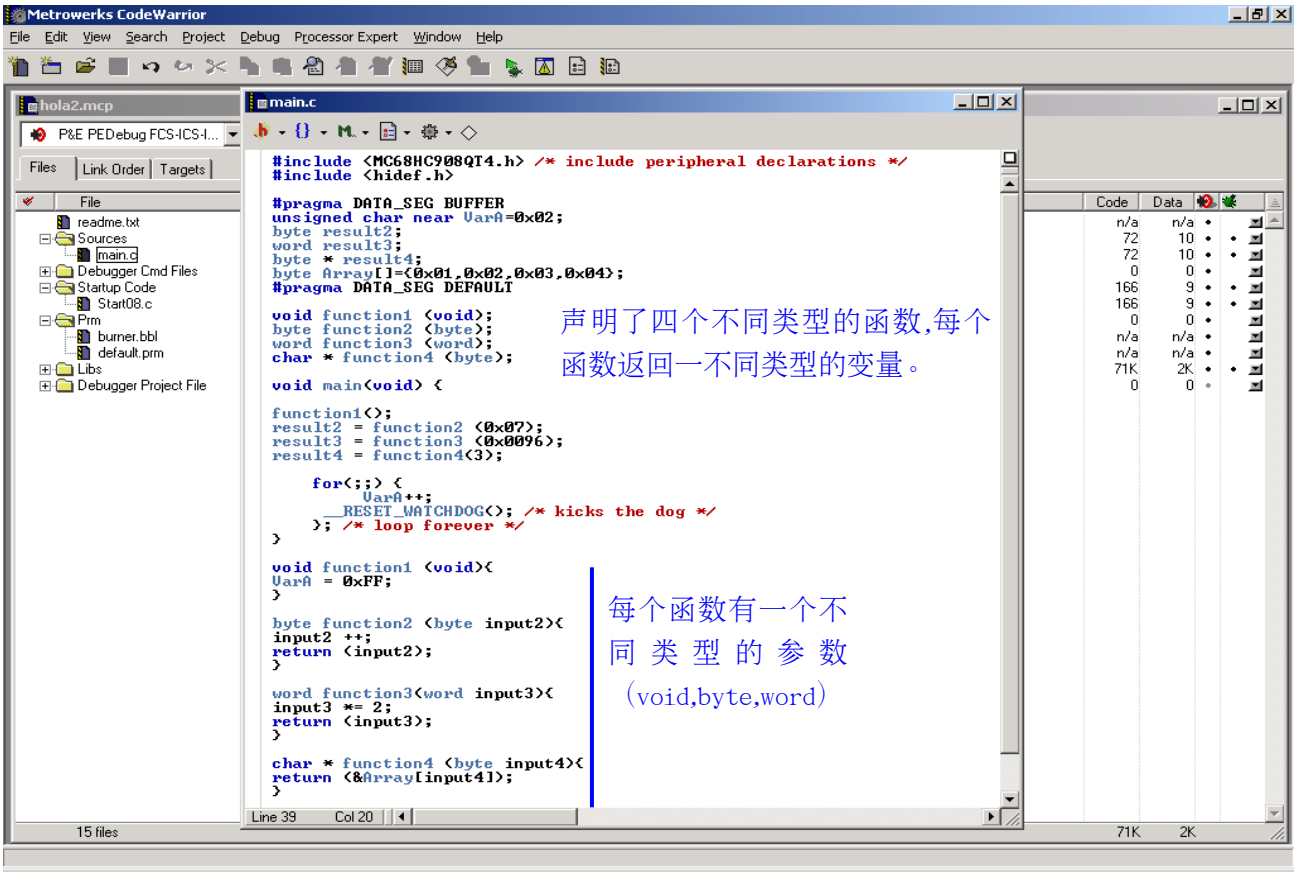
除函数返回一对象大于二字节，函数结果都返回到寄存器中。依据返回类型，使用不同的寄存器。如下表所示:

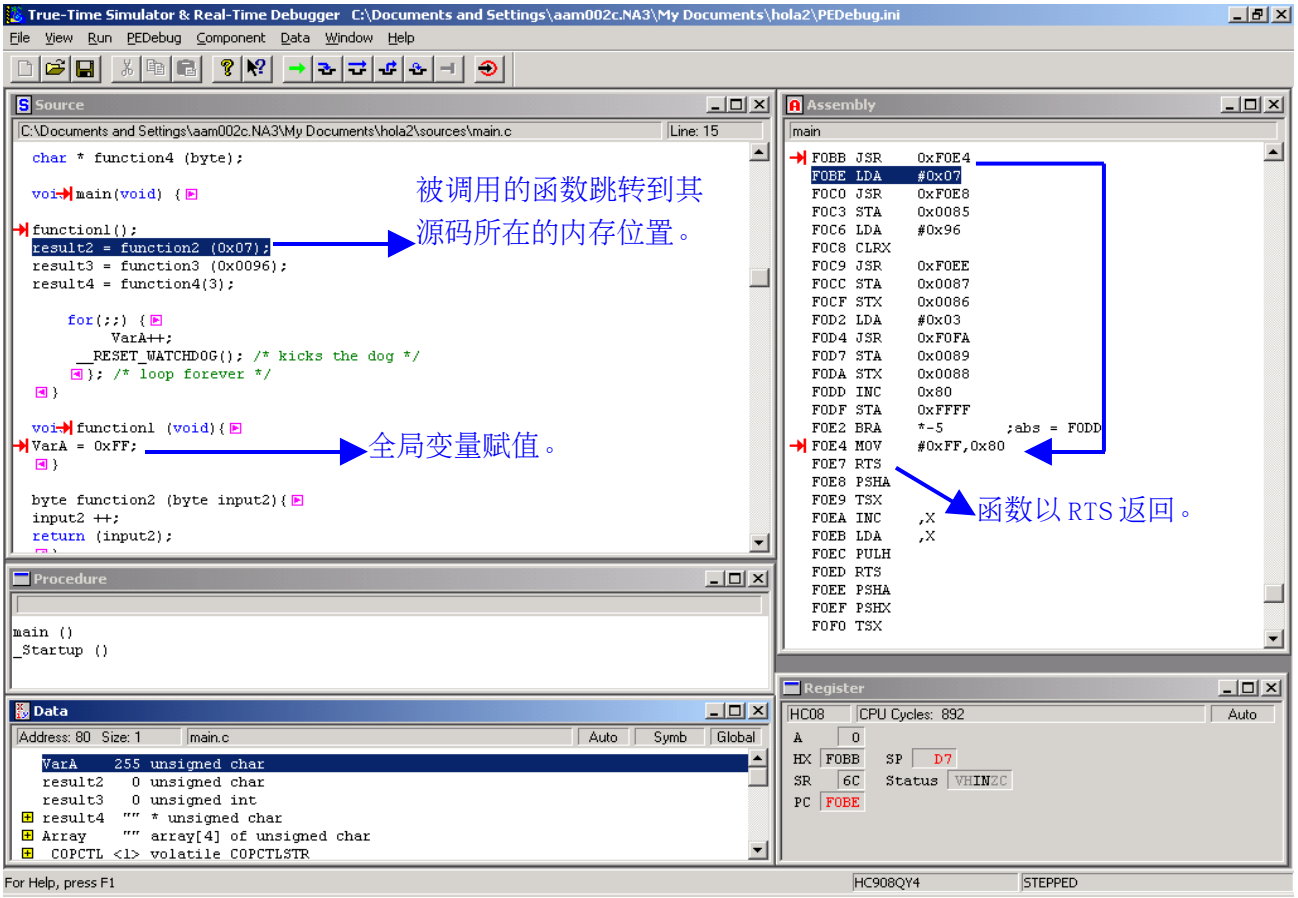
返回类型	寄存器

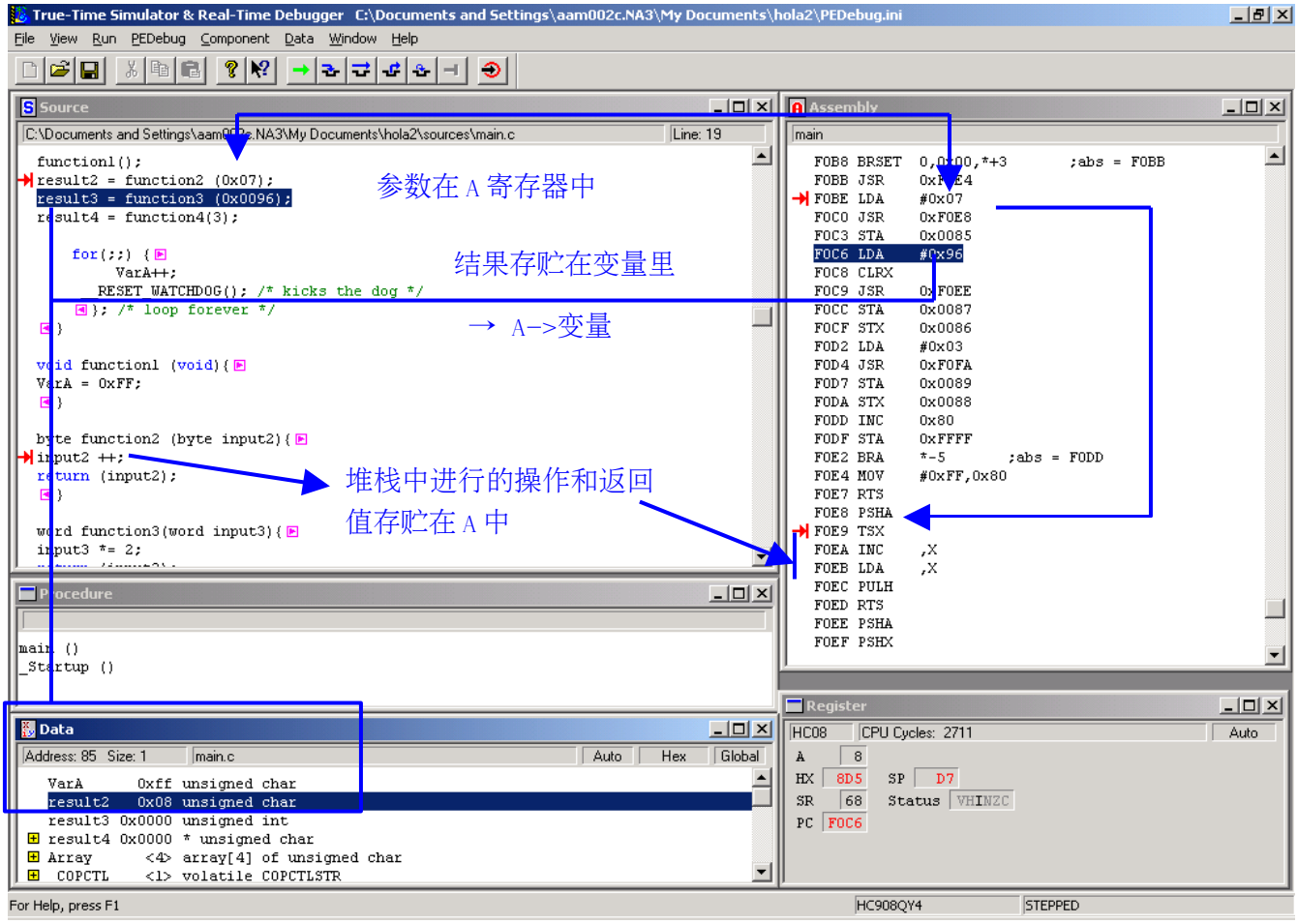
Char(signed或unsigned)	A
int(signed或unsigned)	X：A
指针/数组	X：A
函数指针	X：A

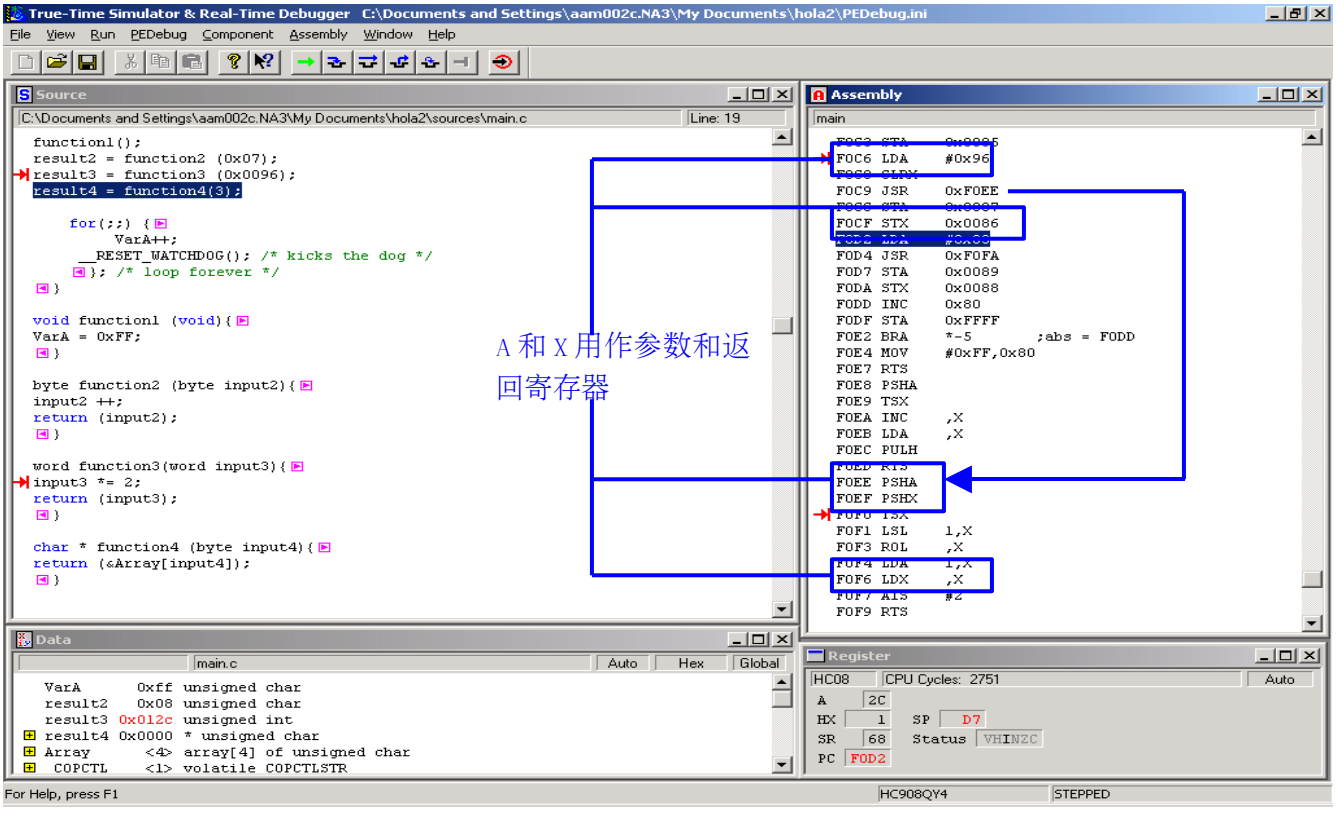
返回大对象：函数返回大于二字节的对象均与一个附加的参数一起调用，它被传到H：X。这个参数是对象应复制到的地址。

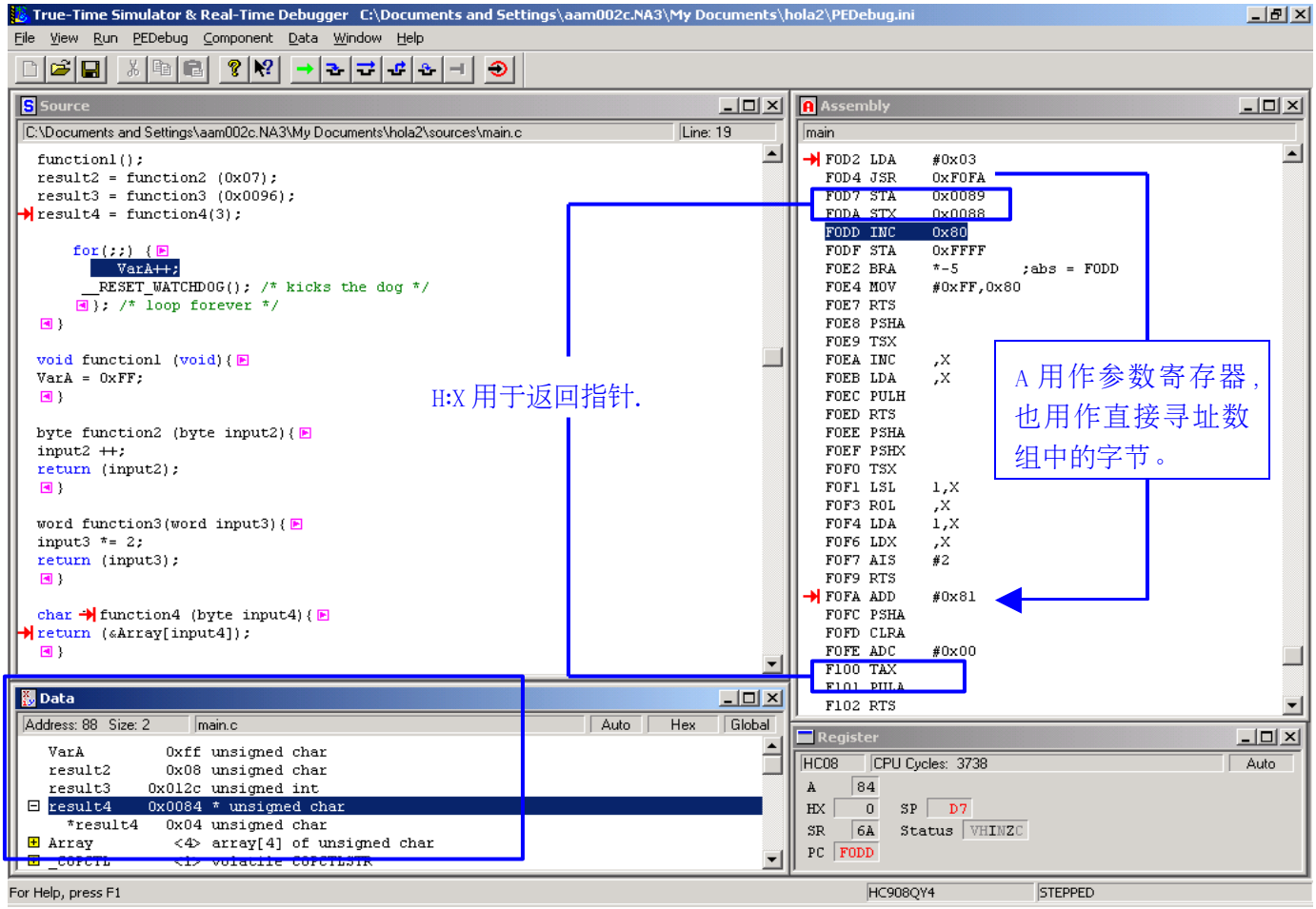
打开文件：Lab5-Arguments.mcp




True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Data Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 15
char * function4 (byte);
voi main(void) {
functionl();
result2 = function2 (0x07);
result3 = function3 (0x0096);
result4 = function4(3);
for(;;) {
VarA++;
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
voi functionl (void){
VarA = 0xFF;
}
byte function2 (byte input2){
input2 ++;
return (input2);
被调用的函数跳转到其源码所在的内存位置。
全局变量赋值。
Assembly
main
F0BB JSR 0xF0E4
F0BE LDA #0x07
F0C0 JSR 0xF0E8
F0C3 STA 0x0085
F0C6 LDA #0x96
F0C8 CLRX
F0C9 JSR 0xF0EE
F0CC STA 0x0087
F0CF STX 0x0086
F0D2 LDA #0x03
F0D4 JSR 0xF0FA
F0D7 STA 0x0089
F0DA STX 0x0088
F0DD INC 0x80
F0DF STA 0xFFFF
F0E2 BRA *-5 ;abs = F0DD
F0E4 MOV #0xFF,0x80
F0E7 RTS
F0E8 PSHA
F0E9 TSX
F0EA INC ,X
F0EB LDA ,X
F0EC PULH
F0ED RTS
F0EE PSHA
F0EF PSHX
F0F0 TSX
函数以 RTS 返回。
Procedure
main ()
_Startup ()
Data
Address: 80 Size: 1
main.c
Auto
Symb
Global
VarA 255 unsigned char
result2 0 unsigned char
result3 0 unsigned int
result4 "" * unsigned char
Array "" array[4] of unsigned char
COPCTL <l> volatile COPCTLSTR
Register
HC08
CPU Cycles: 892
Auto
A 0
HX F0BB SP D7
SR 6C Status VHINZC
PC F0BE
For Help, press F1
HC908QY4
STEPPED


True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Data Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 19
function1();
result2 = function2 (0x07);
result3 = function3 (0x0096);
result4 = function4(3);
for(;;) {
VarA++;
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
void function1 (void){
VarA = 0xFF;
}
byte function2 (byte input2){
input2 ++;
return (input2);
}
word function3(word input3){
input3 *= 2;
参数在 A 寄存器中
结果存贮在变量里
→ A->变量
堆栈中进行的操作和返回
值存贮在 A 中
Assembly
main
F0B8 BRSET 0,0x00,*+3 ;abs = F0BB
F0BB JSR 0xF0E4
F0BE LDA #0x07
F0C0 JSR 0xF0E8
F0C3 STA 0x0085
F0C6 LDA #0x96
F0C8 CLRX
F0C9 JSR 0xF0EE
F0CC STA 0x0087
F0CF STX 0x0086
F0D2 LDA #0x03
F0D4 JSR 0xF0FA
F0D7 STA 0x0089
F0DA STX 0x0088
F0DD INC 0x80
F0DF STA 0xFFFF
F0E2 BRA *-5 ;abs = F0DD
F0E4 MOV #0xFF,0x80
F0E7 RTS
F0E8 PSHA
F0E9 TSX
F0EA INC ,X
F0EB LDA ,X
F0EC PULH
F0ED RTS
F0EE PSHA
F0EF PSHX
Procedure
main ()
_Startup ()
Data
Address: 85 Size: 1
main.c
Auto Hex Global
VarA 0xff unsigned char
result2 0x08 unsigned char
result3 0x0000 unsigned int
result4 0x0000 * unsigned char
Array <4> array[4] of unsigned char
COPCTL <1> volatile COPCTLSTR
Register
HC08
CPU Cycles: 2711
Auto
A 8
HX 8D5 SP D7
SR 68 Status VHINZC
PC F0C6
For Help, press F1
HC908QY4
STEPPED


True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Assembly Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 19
function1();
result2 = function2 (0x07);
result3 = function3 (0x0096);
result4 = function4(3);
for(;;) {
VarA++;
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
void function1 (void){
VarA = 0xFF;
}
byte function2 (byte input2){
input2 ++;
return (input2);
}
word function3(word input3){
input3 *= 2;
return (input3);
}
char * function4 (byte input4){
return (&Array[input4]);
}
A 和 X 用作参数和返
回寄存器
Assembly
main
F0C6 LDA #0x96
F0C9 JSR 0xF0EE
F0CF STX 0x0086
F0D4 JSR 0xF0FA
F0D7 STA 0x0089
F0DA STX 0x0088
F0DD INC 0x80
F0DF STA 0xFFFF
F0E2 BRA *-5 ;abs = F0DD
F0E4 MOV #0xFF,0x80
F0E7 RTS
F0E8 PSHA
F0E9 TSX
F0EA INC ,X
F0EB LDA ,X
F0EC PULH
F0EE PSHA
F0EF PSHX
F0F1 LSL 1,X
F0F3 ROL ,X
F0F6 LDX ,X
F0F9 RTS
Data
main.c
Auto Hex Global
VarA 0xff unsigned char
result2 0x08 unsigned char
result3 0x012c unsigned int
result4 0x0000 * unsigned char
Array <4> array[4] of unsigned char
COPCTL <1> volatile COPCTLSTR
Register
HC08
CPU Cycles: 2751
Auto
A 2C
HX 1 SP D7
SR 68 Status VHINZC
PC F0D2
For Help, press F1
HC908QY4
STEPPED



3．7 中断

好，最后一个棘手的问题深深地困扰嵌入式世界：怎样处理中断?

答案是……简单！

CodeWarrior编译器提供了一个非ANSI的变通的方法，在源码中直接指定中断向量号t。表达式以***interrupt***关键字开始，接着是中断向量号,最后是函数原型。

```
interrupt 17 void TBM_ISR (void){
    /* Timebase Module Handler*/
}
```

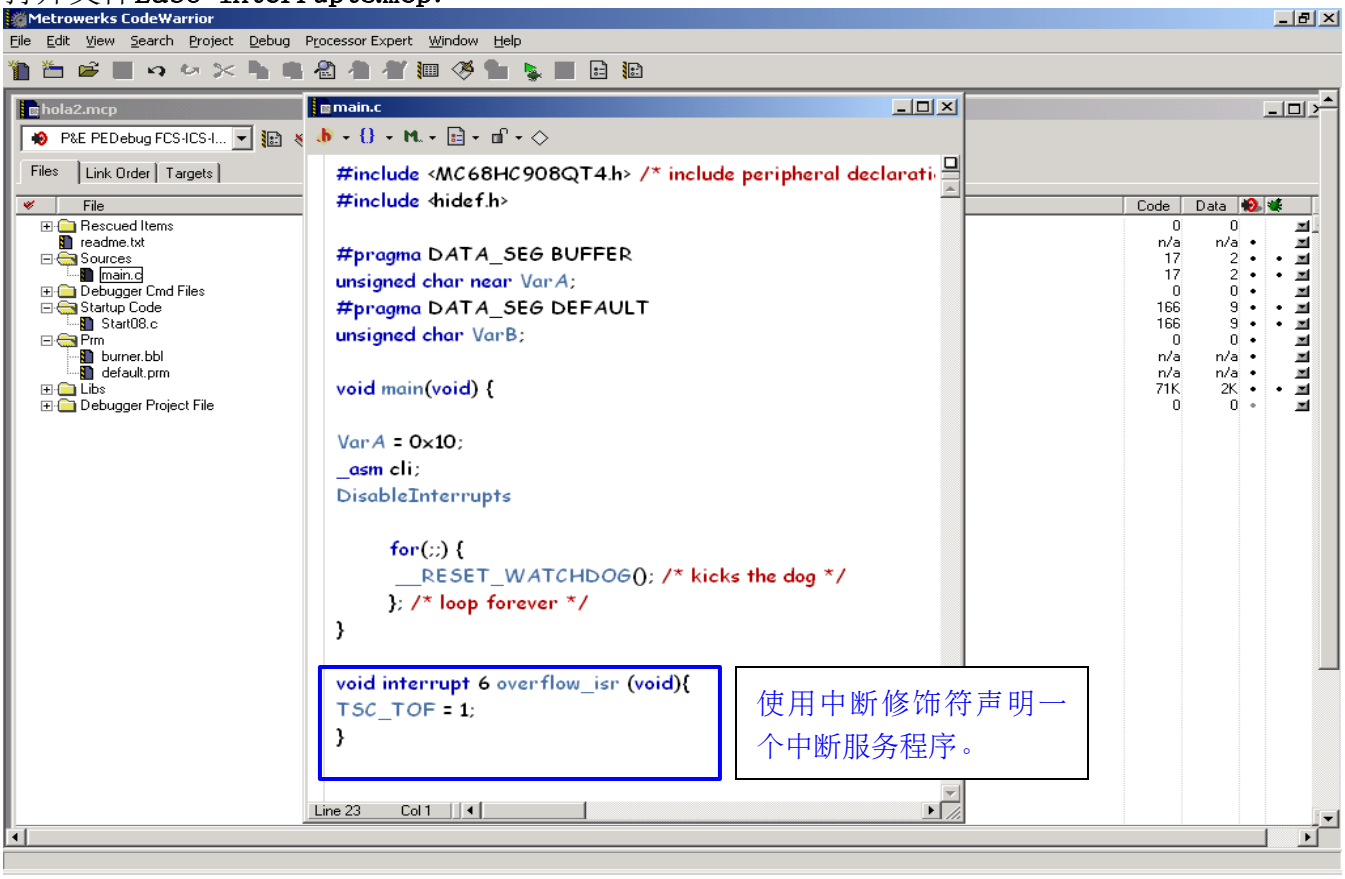
你应查HC08手册，RESET和软中断向处在最高和相同的优先级。这两个是HC08的向量0，但CodeWarrior不得不给每个向量不同的号，因此第一个，RESET，将是向量0，然后SWI，向量1，依次类推……直到最后TBM为向量17，以GP32为例。

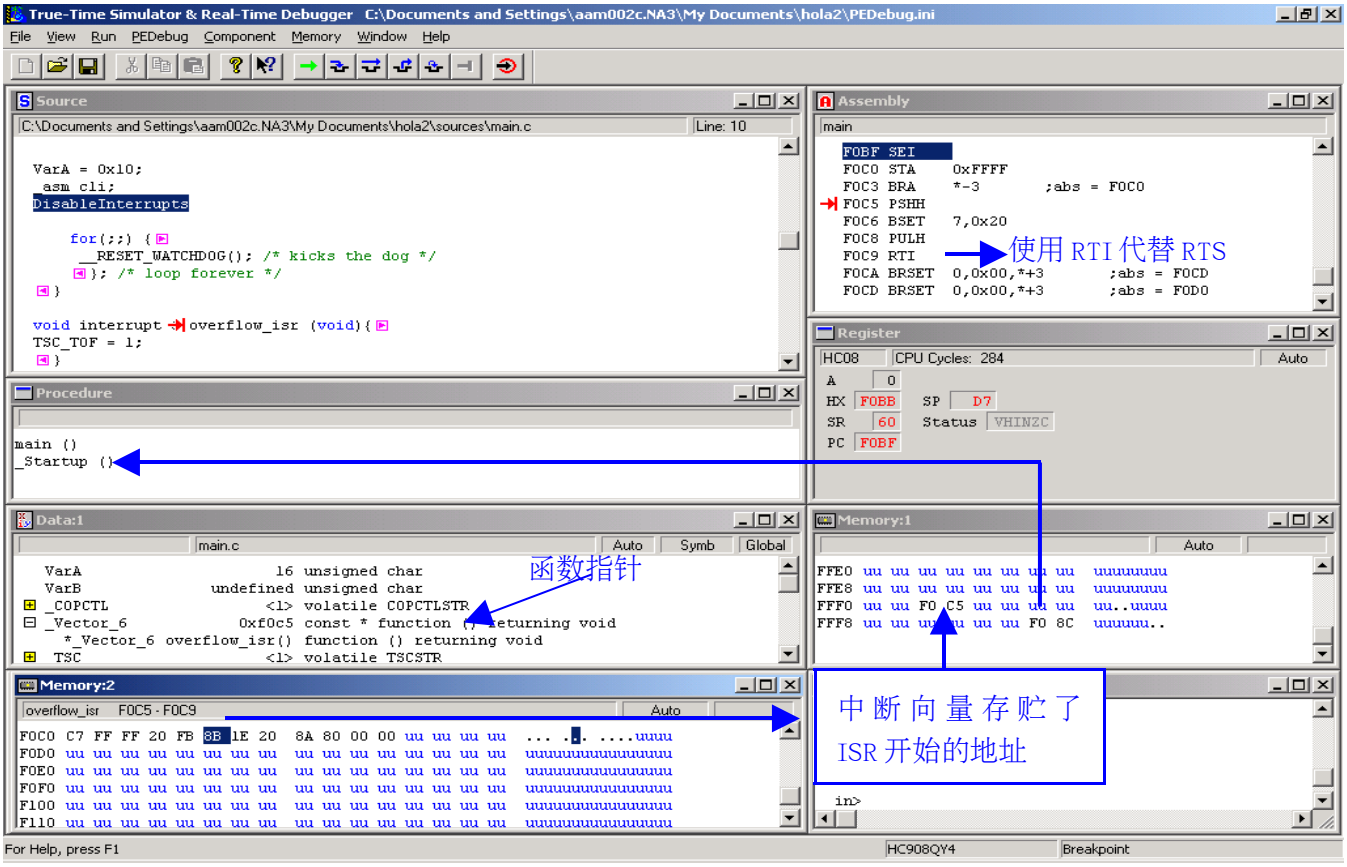
中断向量表定位:

向量号	向量地址	向量地址大小
0	0xFFFE~0xFFFF	2
1	0xFFFC~0xFFFD	2

2	0xFFFA~0xFFFB	2
…	…	…
n	0xFFFF-(n*2)	2

打开文件Lab6-Interrupts.mcp:





3．8 叠代、跳转、循环

执行无限循环：

```
While(1);
For(;;);
Loop:
    goto Loop;
```

对于嵌入式系统：

循环总是被基于MCU的应用所需要。对于应用程序第2种循环最好，因为它不会导致“always true warning”的警告。

3．9 标准C库

标准库如stdio.h通常包含在编译器中。Getchar()、gets()、printf()、putchar()、puts()、scanf()、sprintf()、sscanf()等，都是这些库中的常用函数。

```
#include <stdio.h>
void main(void){
        printf( “Hello World!\n” );
        while(1);
}
```

当给PC机写这段代码， printf()缺省的控制台是显示器，但HC08不需要显示器作为片外外设，如果有，哪个端口用于显示？什么时候我们定义它？在哪儿？

在嵌入式编程中，通常printf()调用putchar()执行打印，这假定控制台缺省为片上串行口（SCI）。

在模拟时，printf()访问片上“虚拟”IO调用一个模拟终端通过片上SCI输出。建议修改基础库函数*putchar()*和*getchar()* t使用任何用户需要的输出/输入控制台。

三、CodeWarrior介绍

1、C编译器

对于现在的嵌入式应用，没必要花大量时间选择一个C/C++交叉编译器，但你应记住详细的难事。稍微详细的资料让我们使用一个特定的交叉编译器容易一些，减少工程中受挫。理想情况下，应从不考虑你的编译器。它只是你用于将系统行为的算法和规则转化为可执行的程序。

但是，世界不是理想的……当比较两个或更多的交叉编译器，以满足你需要的硬件和软件，你应当考虑些什么事情呢？

有一些好的特性形成巨大的不同：

在线汇编: 尽管C语言的发明已超过25年，当开发嵌入式系统时，使用一定数量的汇编仍然是平常的事。

中断函数: 交叉编译器另一个让人满意的特性是指定中断类型。对PC平台，这一非标准关键字是C语言普遍增加的。当用作函数声明的一部份，它告诉编译器这个函数是一个中断服务程序（ISR）。编译器能产生额外的堆栈信息和寄存器保存以及任何ISR需要的恢复。一个好的编译器也会阻止这种方式定义的函数被程序的其它部分调用。应该明白，C/C++中与进出ISR相关的上层与汇编中是没有差别的。

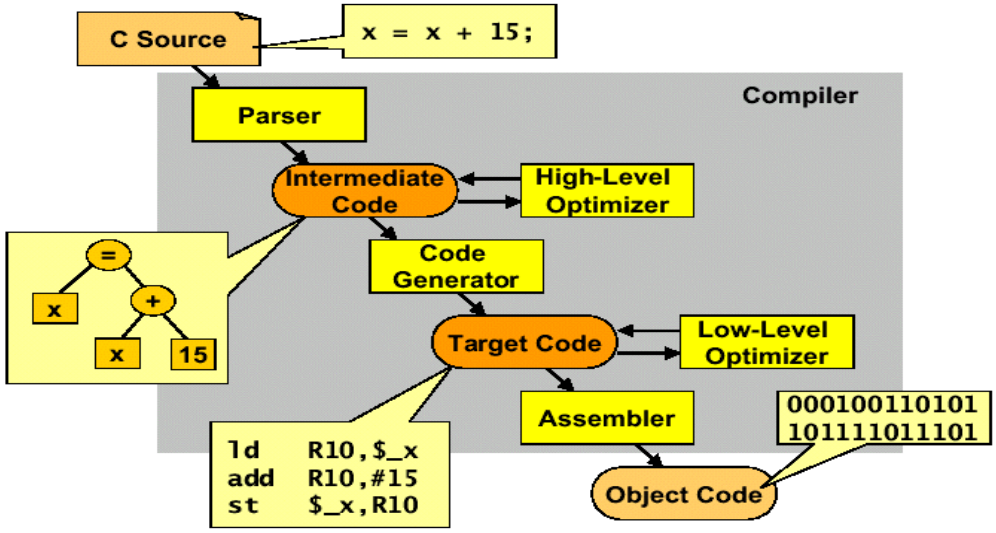
例如CodeWarrior,尽管进一步深入理解一个处理器的中断向表的结构。这种情况，只需简单地向ISR 标记添加中断类型（0x1E)。这使得中断向量表自动生成，并消除了编程人员潜在的误解和错误。

产生汇编语言: 产生汇编语言清单的编译器，作为汇编处理器的一部分是一个令人满意的工具。这一特性对手工优化代码很有帮助，因为你能容易地看到你高级语言程序的每一行产生了什么代码，如果一个特定的函数对于给定的应用太慢，你将能够容易地选择函数最好的部分用汇编重写。

标准库:当你为通用计算机开发应用软件，你希望你的编译器包含一套标准C函数库，数学库和C++类。它们包含各种程序如memcpy()、sin()和cout等。但由于这些库函数不严格地是C或C++语言标准的一部分(库标准是分开的),一个编译器提供商可能省略它，这些省略在嵌入系统程序员使用的交叉编译器提供商中是很普遍的。因些在某些情况下你不得不争取得到标准库的权利。

想一下多少次你花时间重写那些自己的函数。因此花些时间坚持将标准库包含在你购买的编译器中。当然不大可能在多数嵌入式系统应用中使用printf()，但直到太迟了你才意识到你需

要很多其它的函数。
如果提供了标准库，确保它们能再进去。换句话说，那些库中的每个函数能同时执行多次。重入函数能递归调用或多线程执行。对于库程序这意味着不应使用全局变量。其内部所有数据必须在栈上。
起动代码:这是在main()前先执行的额外的一段程序。起动代码通常用汇编写成并和你建立的可执行代码连接在一起。它为用高级语言写的程序的执行铺路。
显然，一个编译器缺少我们提到的一些特性，也仍可是一个好的编译器。毫无疑问，人们会争论非标准特性象 “asm” 和interrupt关键字减少了代码的兼容性。但若你在两个或多个差不多的交叉编译间选择时，你可能会考虑这些事情。正好你被要求完成这些任务，它们会使你的工作容易。它们的确减少编程受挫。
2、编译器必要条件
一个好的编译必须提供的功能有：
ROM定位代码:例如，编译器必须给出容易写入EPROM的的ROM定位代码。
优化的代码:代码大小和执行时间同样必须优化。分枝和窥孔优化自动激活。
重入代码:无全局变量用于存贮中间结果。由于结构和或缺少栈操作（如HC05、ST7……）并不是所有的目标系统有重入代码。HC08是允许重入代码的一个好目标系统。
支持CPU家族的不同成员：对于M68K家族有用，允许激活68000、68020、68332或68881指令集。
支持不同的存储器模型:允许根据存贮器地图的某些简单限制优化产生的代码。已有的存贮器模型与目标处理器结果紧密相关。
非明显的优化:尽管当用汇编写代码时，能用于决定可能的复杂的优化。
编译器工作过程示例：



3、存贮器放置—PRM文件
看了前面的代码，尤其是当一个从桌面编程转到嵌入式编程，脑子里会产生一些基本问题……
我的代码在哪儿?

我的变量在哪儿?

怎样访问I/O寄存器?

怎样处理中断?

当写桌面应用程序时，通常不用考虑代码放在哪里。操作系统负责管理内存分配。处理嵌入式设备，你不得不充当OS决定将代码放在何处，至少决定代码开始之处……事实是连接器为我们作了大量工作，但并是所有的事情。

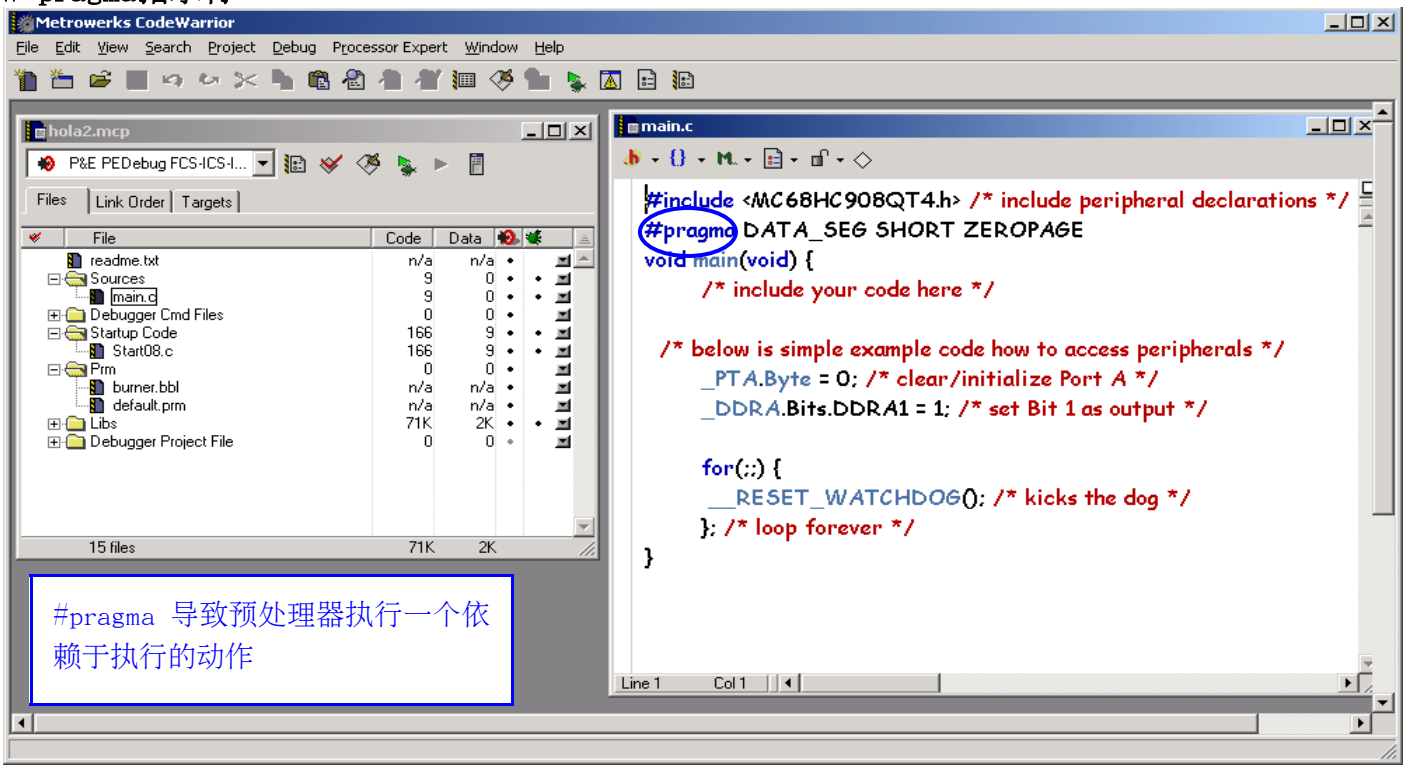
为让连接器知道我们的特定设备或目标的存贮器分布，我们不得不指定它。定义的代码的定位以及内存寻址的数据段由连接器的参数文件控制。这个文件由后缀 “.prm” 识别。连接器的参数文件是一个ASCII文本件。对每个程序你得写一个这样的文件。它包含指导如何连接的命令。

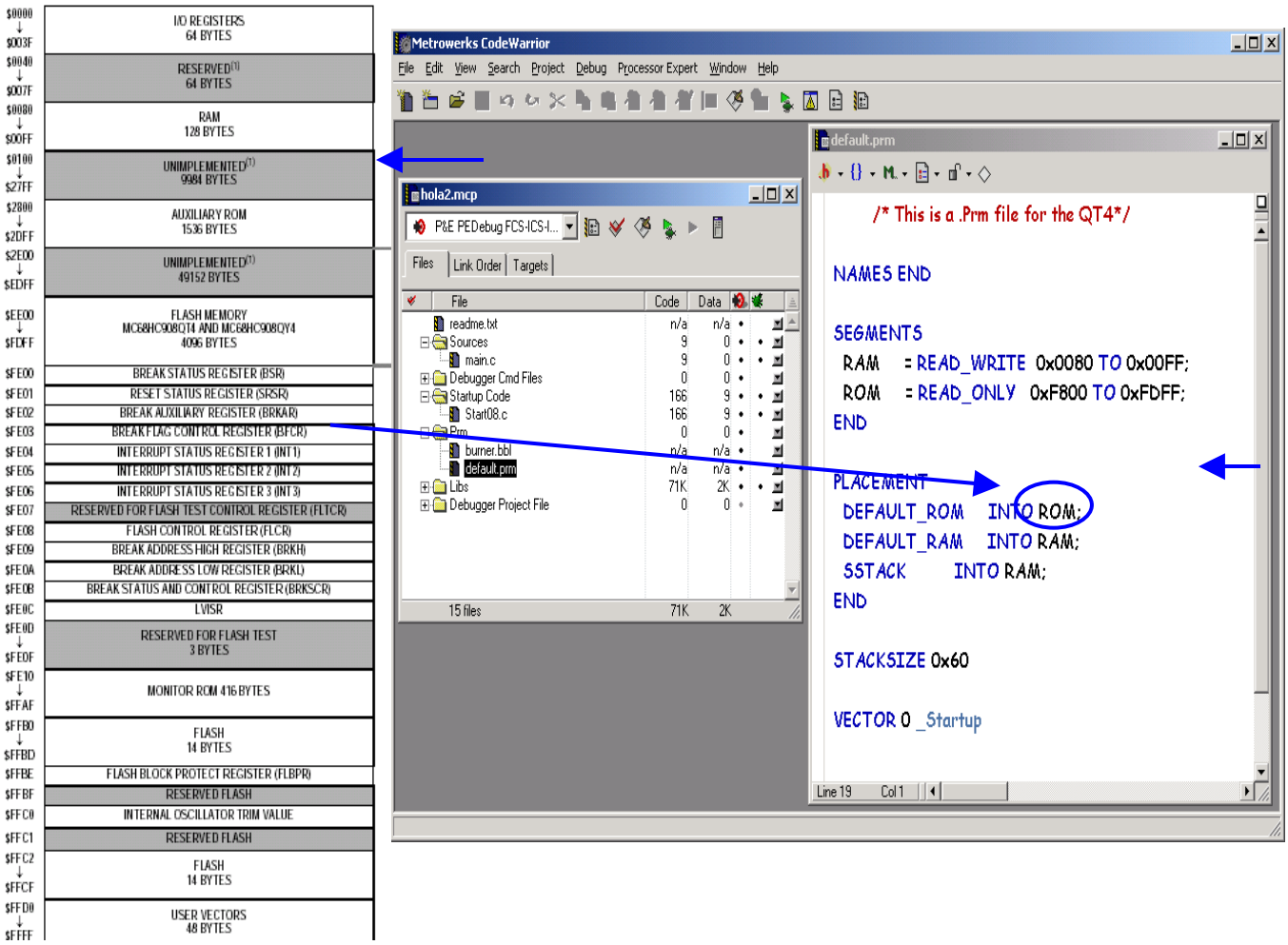
这个文件中，SECTIONS命令块用于定义存贮器的物理区域。在SECTIONS命令块中，每个单独的物理存贮器段用一个名字、一个属性和一个地址范围描述。

一旦定义了SECTIONS，代码和数据段用PLACEMENT命令块定位到存贮器中。PLACEMENT命令块用于将代码和数据段定位到存贮器段。

参数文件中命令的顺序没有关系。你只应确定SEGMENTS块在PLACEMENT块之前指定。若要详细了解连接器的参数文件请参考Metrowerks的手册文件 “SmartLinker.pdf” 。

pragma指示符





除非另规定一个PRAGMA声明外，变量放在Default_RAM 的位置。地址范围$0000至$00FF叫作直接页、基本页或零页。在HC08微控制器家族，在直接页的低地址部分包含I/O和控制寄存器，直接页的高地址部分总是包含RAM。复位后，栈指针总是指向地址$00FF。直接页非常重要，因为许多CPU08指令有一个直接寻址模式（8位寻址模式），其在直接页中访问操作数比扩展寻址模式（16位寻址模式）少一个时钟周期。更有甚者，直接寻址模式指令需要的代码少一字节。一些高效的指令只使用直接页操作数，它们是：BSET、BCLR、BRSET和BRCLR。MOV指令需要一个操作数在直接页。

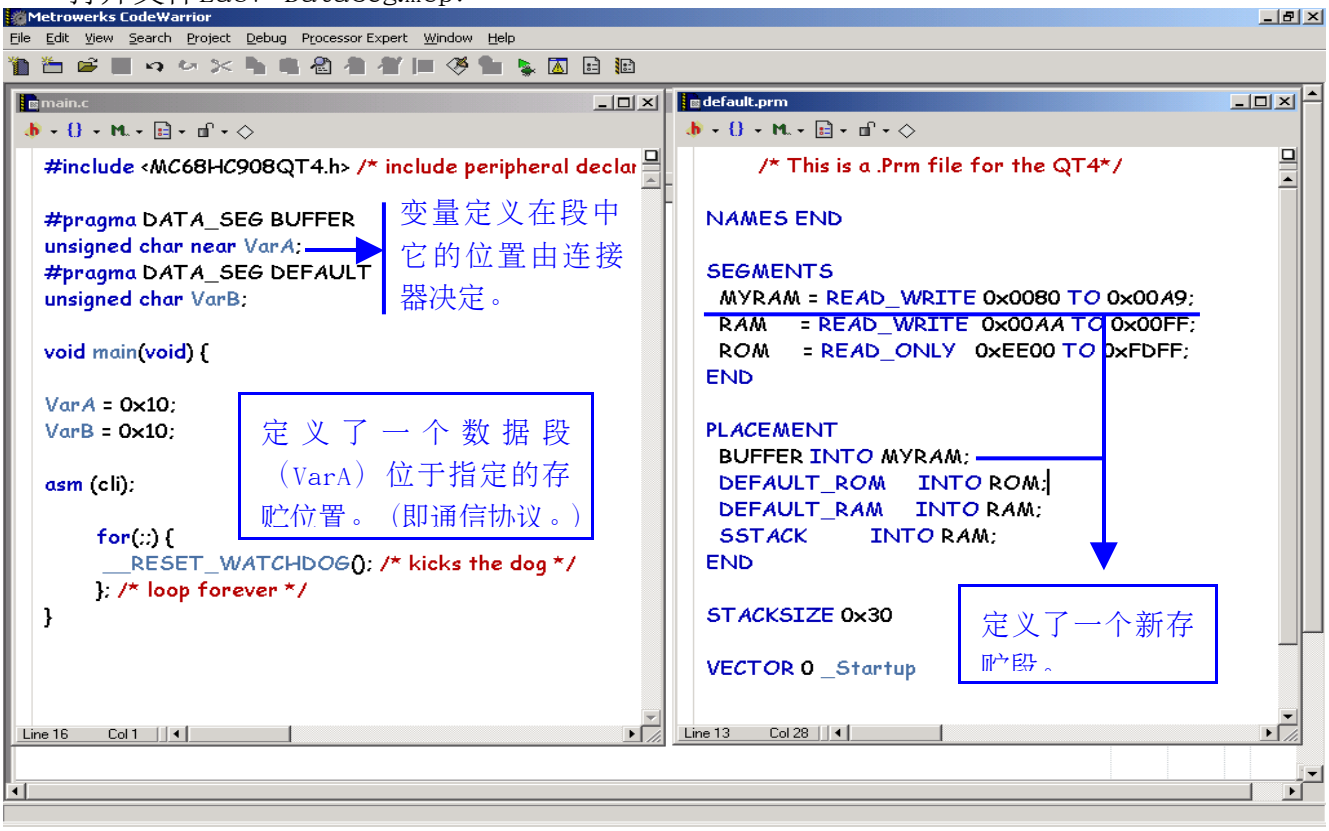
只有变量明确地定义在直接页，编译器才能利用直接寻址模式的效率高优点。ANSI无标准方法做这些，编译器通常提供不同的解决方案。CodeWarrior 使用#pragma声明。

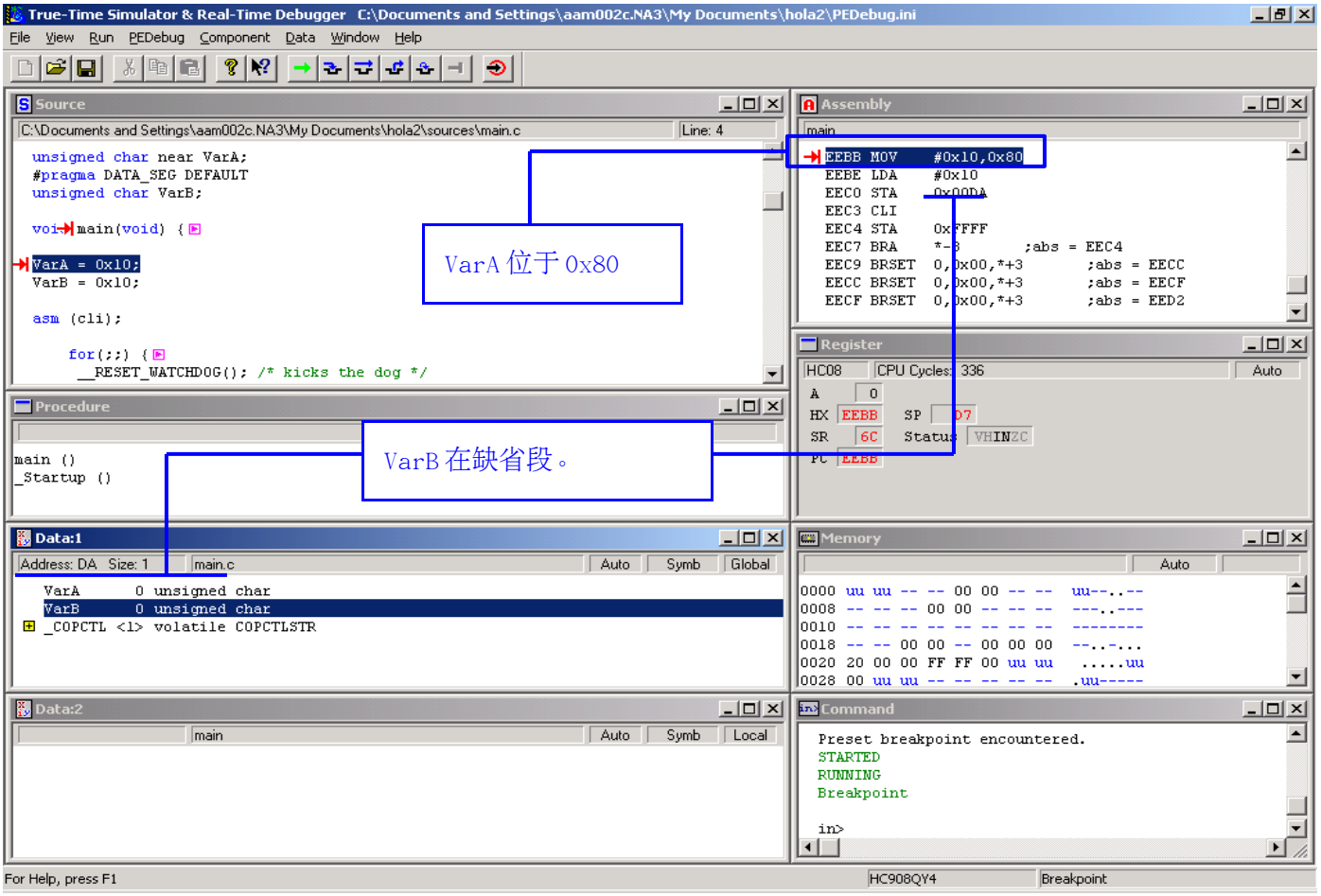
PRAGMA是一个编译器指示。你设置pragma为需要的状态后，那一点以后的所有代码用那个设置编译，直到你改变设置或到达文件的末尾。在每个文件的开头，编译器恢复到工程的设置或缺省设置。

这个pragma将声明的变量放在直接页段，就如前面见到的，程序员必须记住修改PRM文件，使连接器把段放在直接页或零页的一个地址。直接页RAM的数量总是有限的，因此只有频繁使用的变量应放在直接页。如果能用到，要为全局变量释放更多的直接页RAM，堆栈被重定位在直接页RAM之外，这不会影响堆栈指针寻址模式。

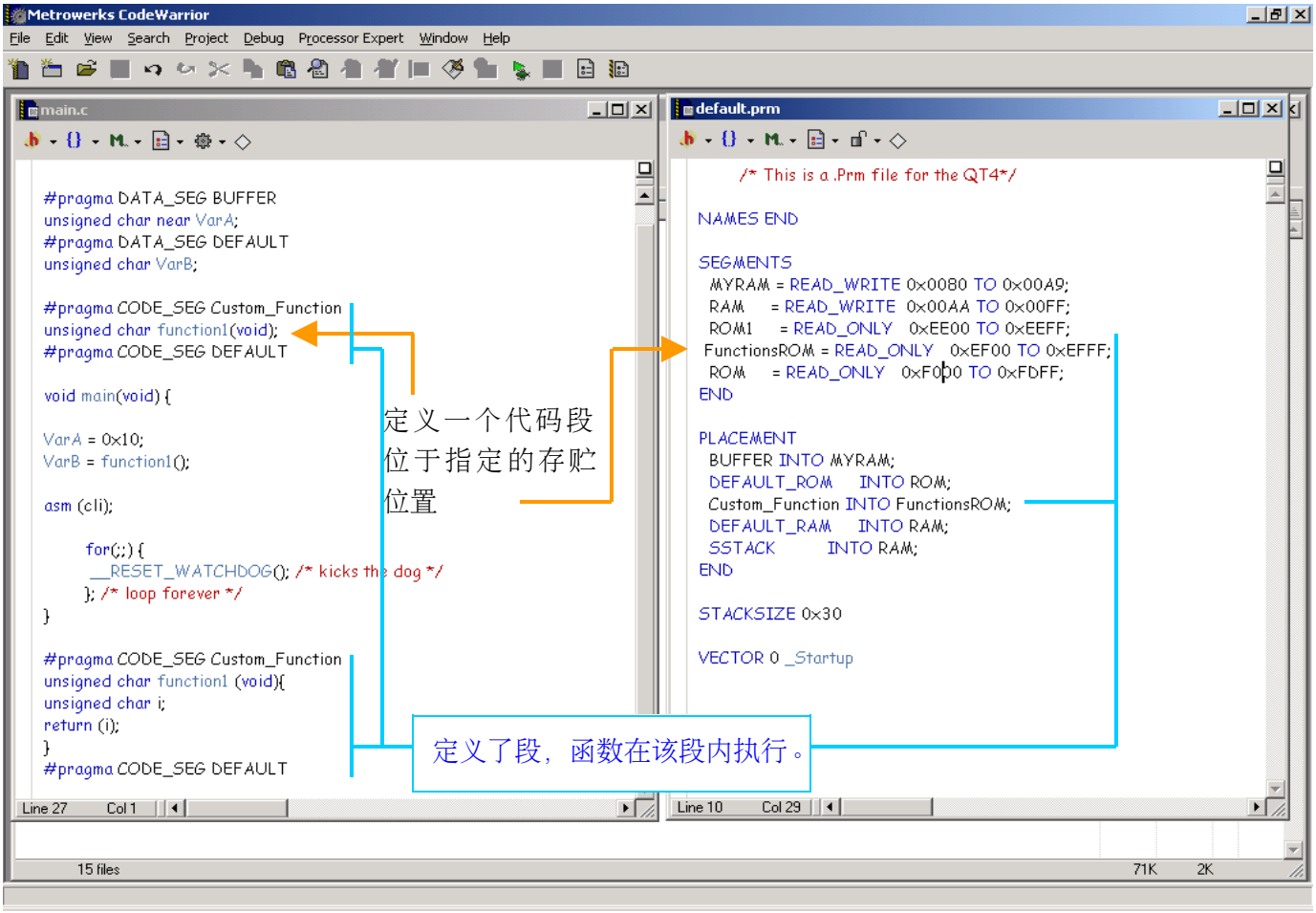
请看下列例子：

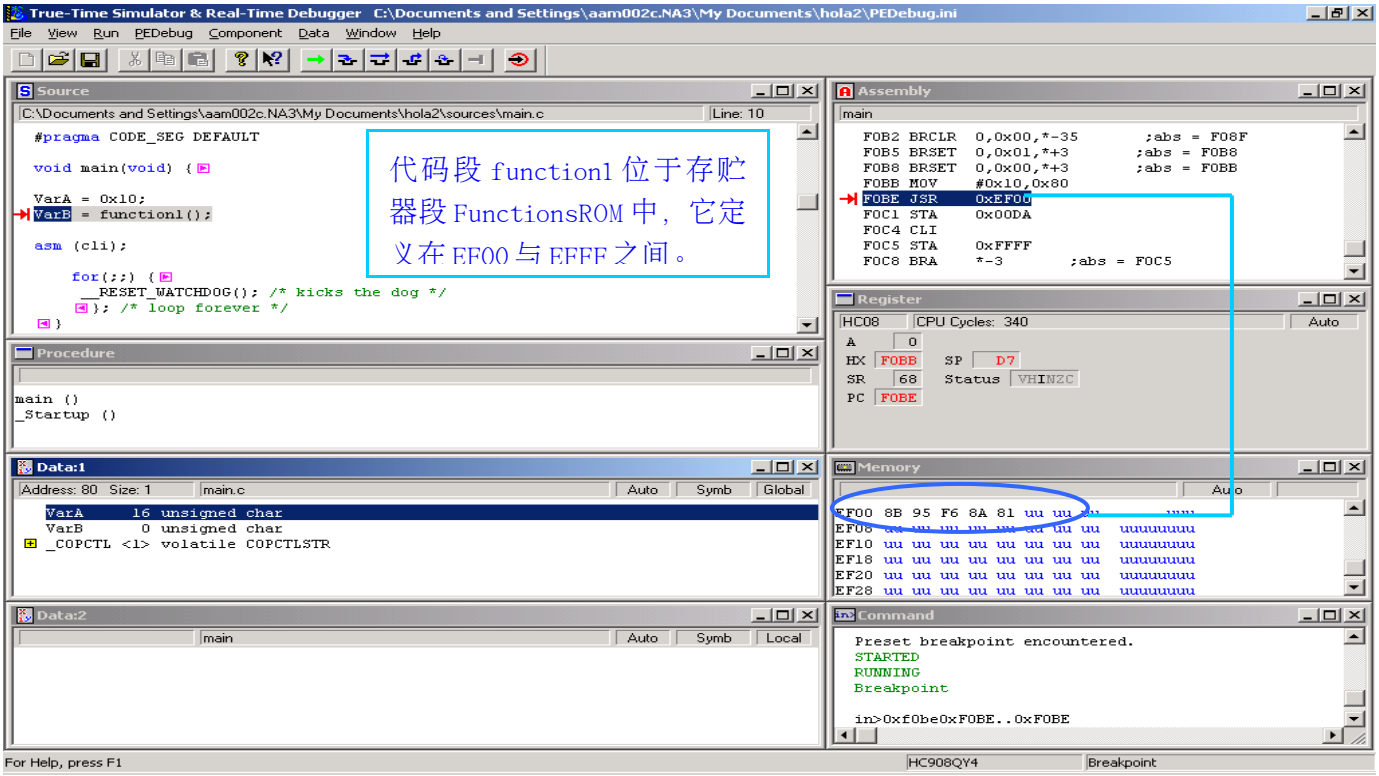
打开文件Lab7-DataSeg.mcp：



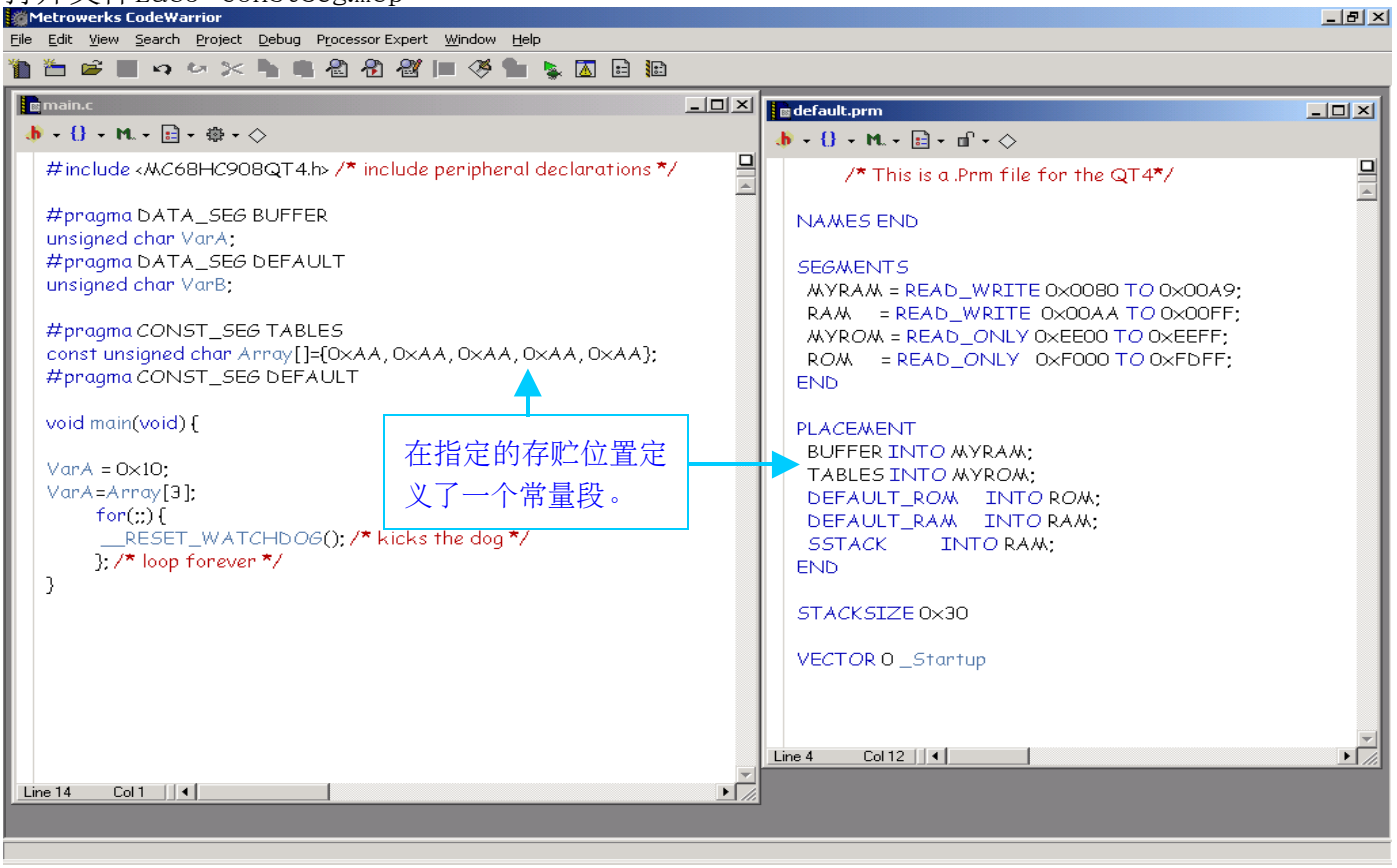


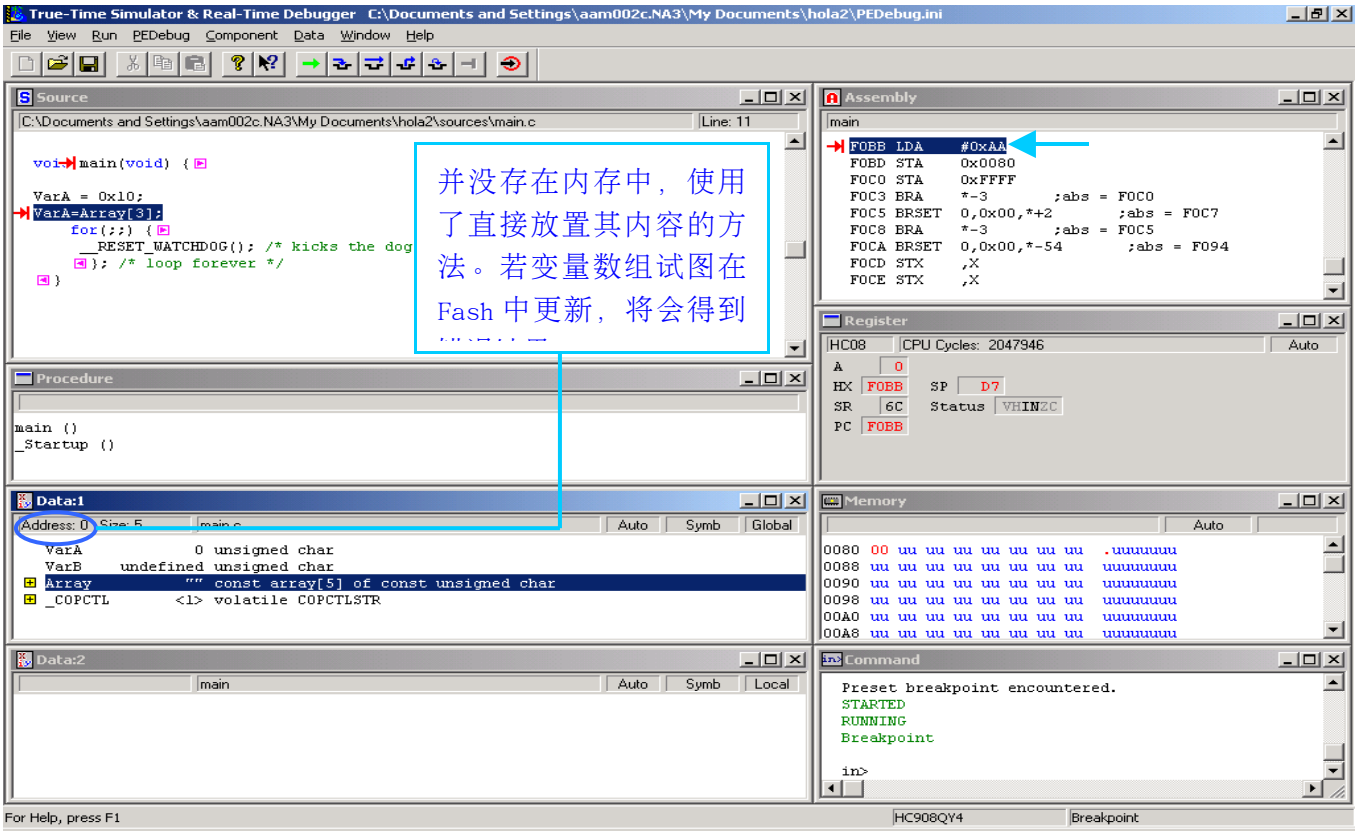
打开文件Lab8-CodeSeg.mcp


Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help
main.c
#pragma DATA_SEG BUFFER
unsigned char near VarA;
#pragma DATA_SEG DEFAULT
unsigned char VarB;
#pragma CODE_SEG Custom_Function
unsigned char function1(void);
#pragma CODE_SEG DEFAULT
void main(void) {
VarA = 0x10;
VarB = function1();
asm (cli);
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
#pragma CODE_SEG Custom_Function
unsigned char function1 (void){
unsigned char i;
return (i);
}
#pragma CODE_SEG DEFAULT
Line 27 Col 1
default.prm
/* This is a .Prm file for the QT4*/
NAMES END
SEGMENTS
MYRAM = READ_WRITE 0x0080 TO 0x00A9;
RAM = READ_WRITE 0x00AA TO 0x00FF;
ROM1 = READ_ONLY 0xEE00 TO 0xEEFF;
FunctionsROM = READ_ONLY 0xEF00 TO 0xEFFF;
ROM = READ_ONLY 0xF000 TO 0xFDFF;
END
PLACEMENT
BUFFER INTO MYRAM;
DEFAULT_ROM INTO ROM;
Custom_Function INTO FunctionsROM;
DEFAULT_RAM INTO RAM;
SSTACK INTO RAM;
END
STACKSIZE 0x30
VECTOR 0 _Startup
Line 10 Col 29
定义一个代码段位于指定的存贮位置
定义了段，函数在该段内执行。
15 files
71K 2K


True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Data Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 10
#pragma CODE_SEG DEFAULT
void main(void) {
VarA = 0x10;
VarB = function1();
asm (cli);
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
Assembly
main
F0B2 BRCLR 0,0x00,*-35 ;abs = F08F
F0B5 BRSET 0,0x01,*+3 ;abs = F0B8
F0B8 BRSET 0,0x00,*+3 ;abs = F0BB
F0BB MOV #0x10,0x80
F0BE JSR 0xEF00
F0C1 STA 0x00DA
F0C4 CLI
F0C5 STA 0xFFFF
F0C8 BRA *-3 ;abs = F0C5
Register
HC08 CPU Cycles: 340 Auto
A 0
HX F0BB SP D7
SR 68 Status VHINZC
PC F0BE
Procedure
main ()
_Startup ()
Data:1
Address: 80 Size: 1 main.c Auto Symb Global
VarA 16 unsigned char
VarB 0 unsigned char
_COPCTL <1> volatile COPCTLSTR
Memory
EF00 8B 95 F6 8A 81 uu uu uu uu uu
EF08 uu uu uu uu uu uu uu uu uuuuuuuu
EF10 uu uu uu uu uu uu uu uu uuuuuuuu
EF18 uu uu uu uu uu uu uu uu uuuuuuuu
EF20 uu uu uu uu uu uu uu uu uuuuuuuu
EF28 uu uu uu uu uu uu uu uu uuuuuuuu
Data:2
main Auto Symb Local
Command
Preset breakpoint encountered.
STARTED
RUNNING
Breakpoint
in>0xf0be0xF0BE..0xF0BE
代码段 function1 位于存贮器段 FunctionsROM 中，它定义在 EF00 与 EFFF 之间。
For Help, press F1
HC908QY4
Breakpoint

打开文件Lab9-ConstSeg.mcp





4、起动程序

你的程序是怎样起动的呢？在工作站或PC上操作系统从磁盘上装入程序并建立环境。在嵌入式系统中没有操作系统。基本上，程序员必须处理程序起动的每个方面。

嵌入式应用，特别是用C或C++写的代码，需要一个起动模块，在起动main()之前配置硬件和代码。通常不可避免地用汇编语言写成，这个模块是处理器离开复位状态第一个执行的代码。

C/C++起动代码通常执行下列动作：

1）关中断；

2）将初始化数据从ROM复制到RAM；

3）将未初始化数据区清零；

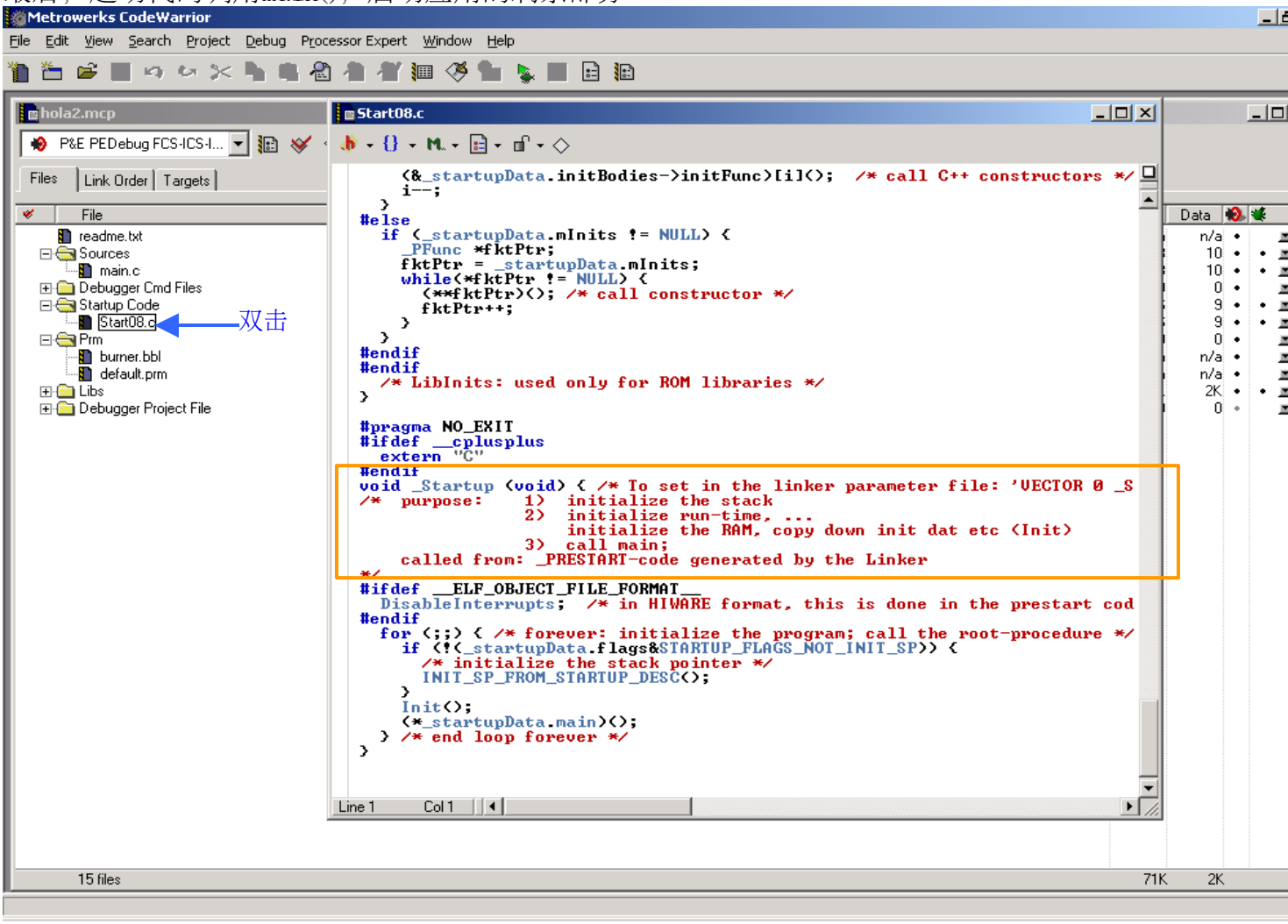
4）为堆栈定位空间以及初始化堆栈；

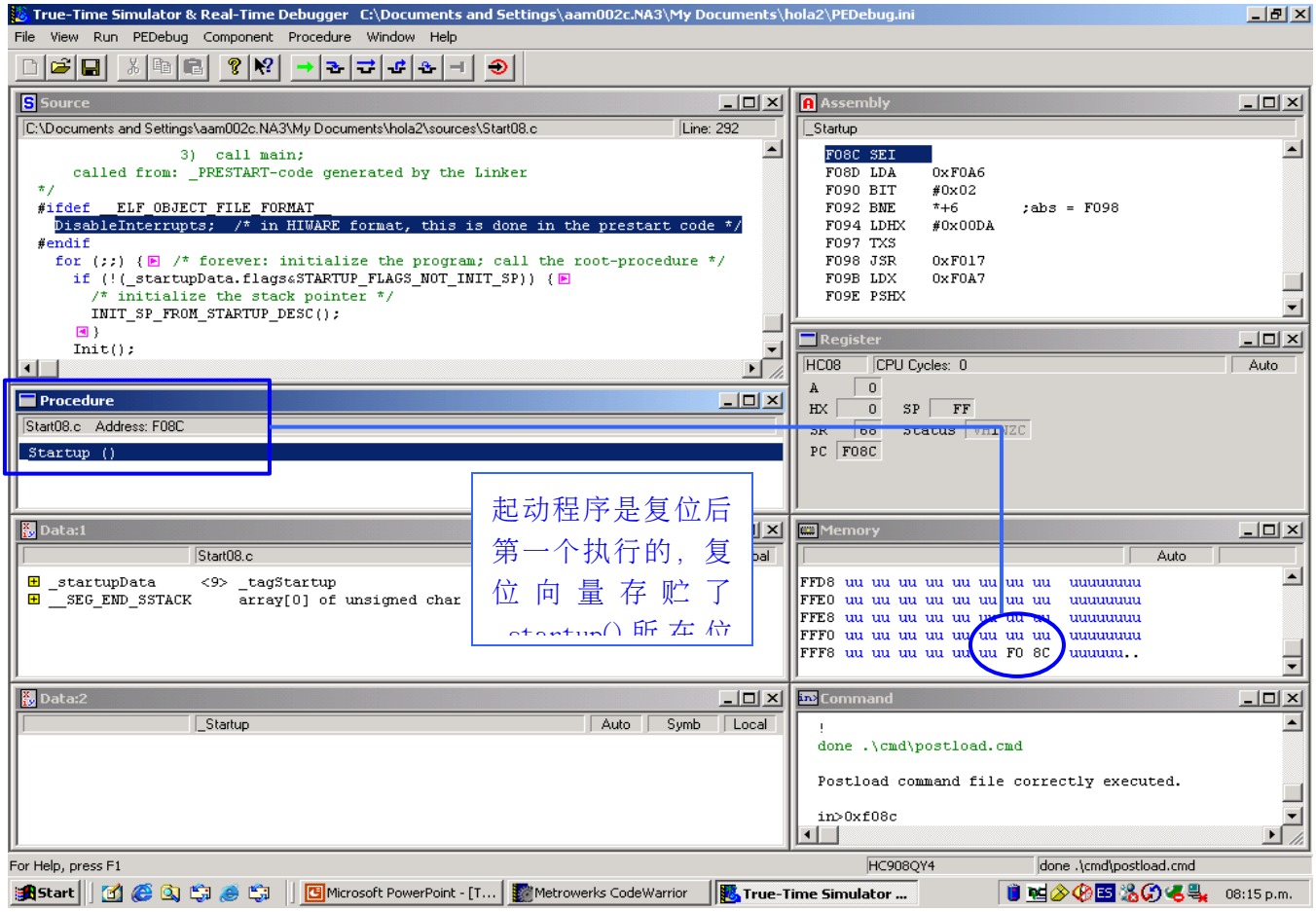
5）创建并初始化堆；

6）执行构造函数并初始化所有全局变量(仅C++)；

7）开中断；

最后，起动代码调用main()，启动应用的剩余部分。





5、编译器优化

CodeWarrior编译器提供了几种从C源代码产生实际汇编代码的优化方法，这些代码被编程到微控制器中。

“Global Optimizations”（全局优化）设置面板设定编译器怎样优化目标代码。所有优化程序重新组织目标代码，不影响其逻辑执行顺序。

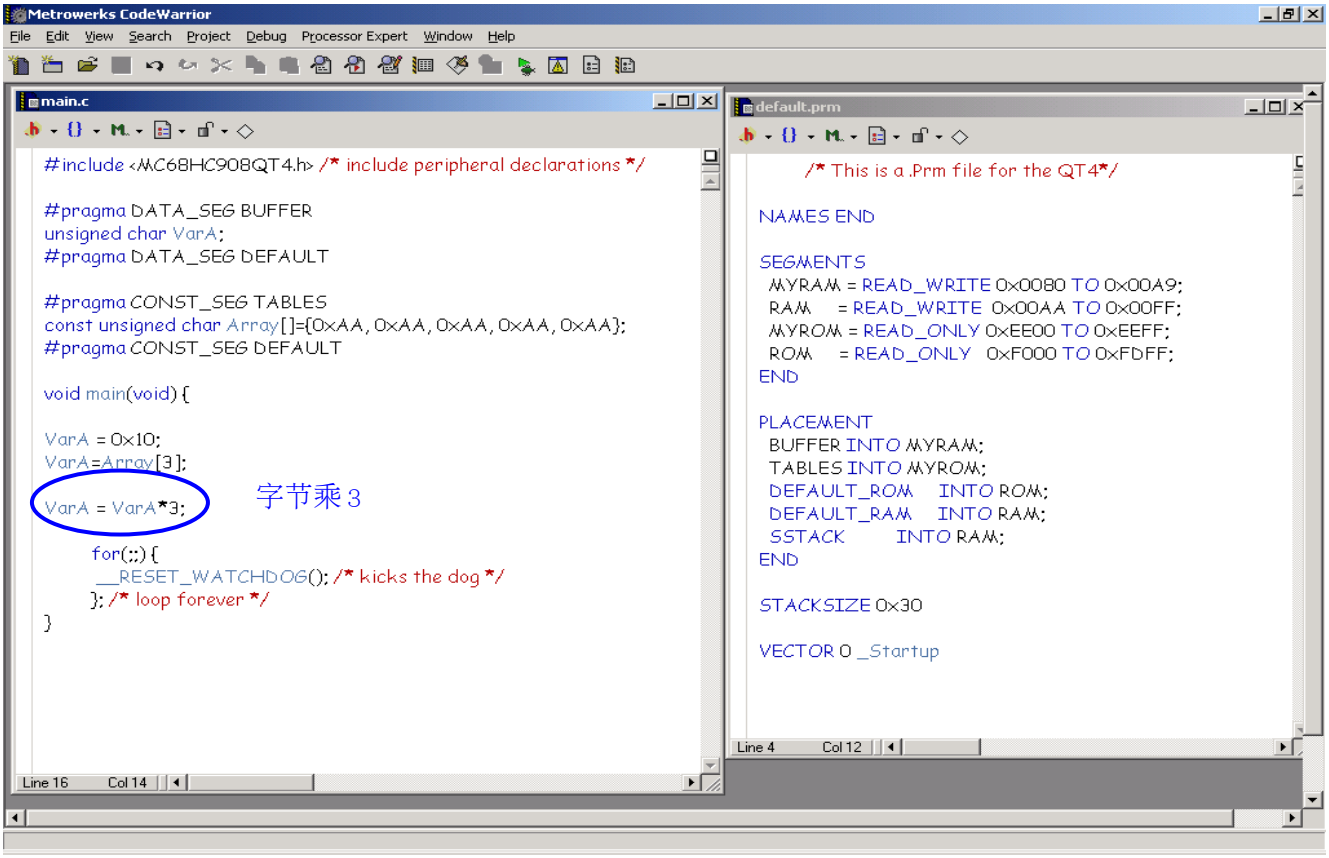
1）强度减弱

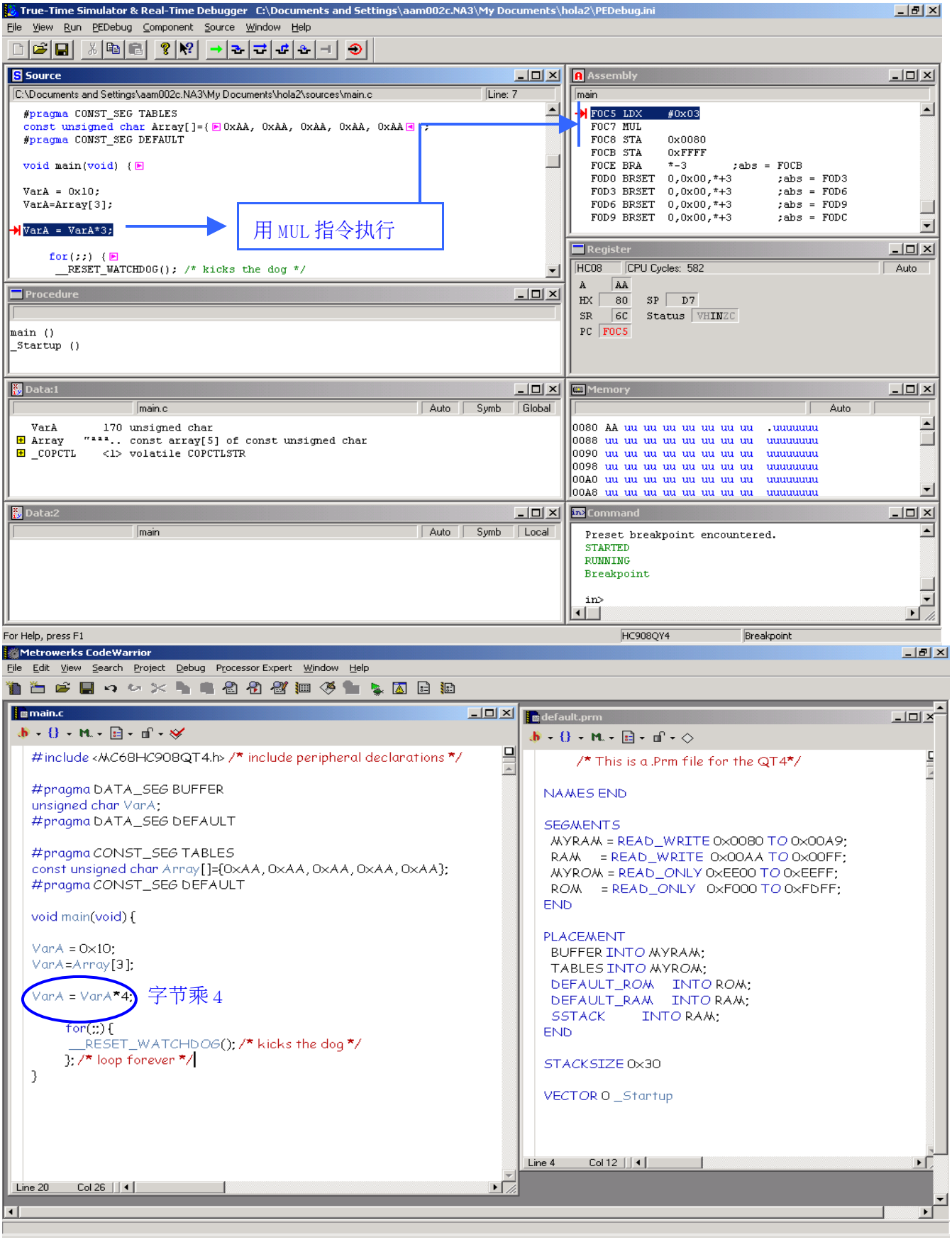
“Strength Reduction”（强度减弱）是一种优化，力争用开销小的操作代替开销大的操作，代价因素是执行时间或代码大小。

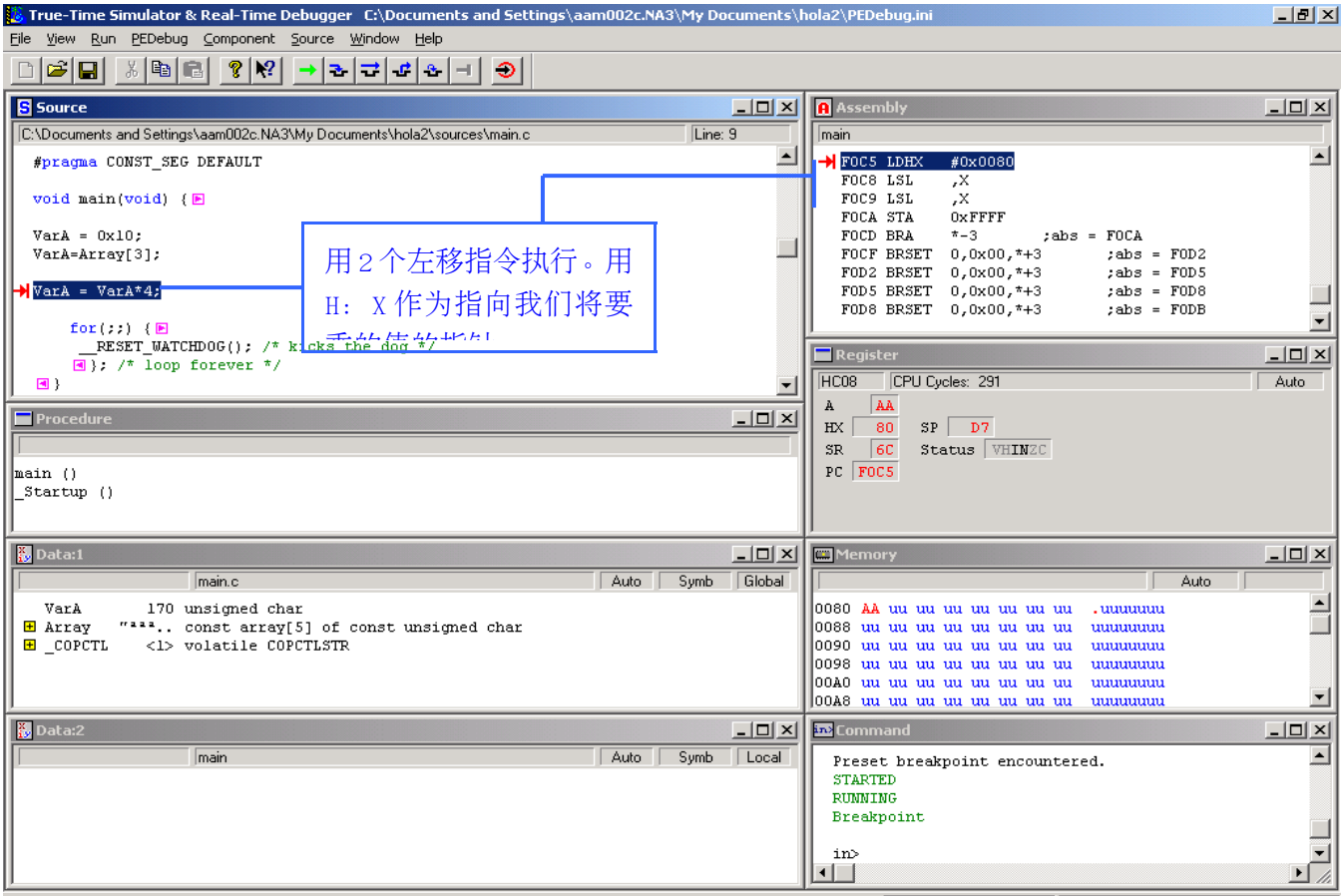
在循环内，用加法指令代替乘法指令。

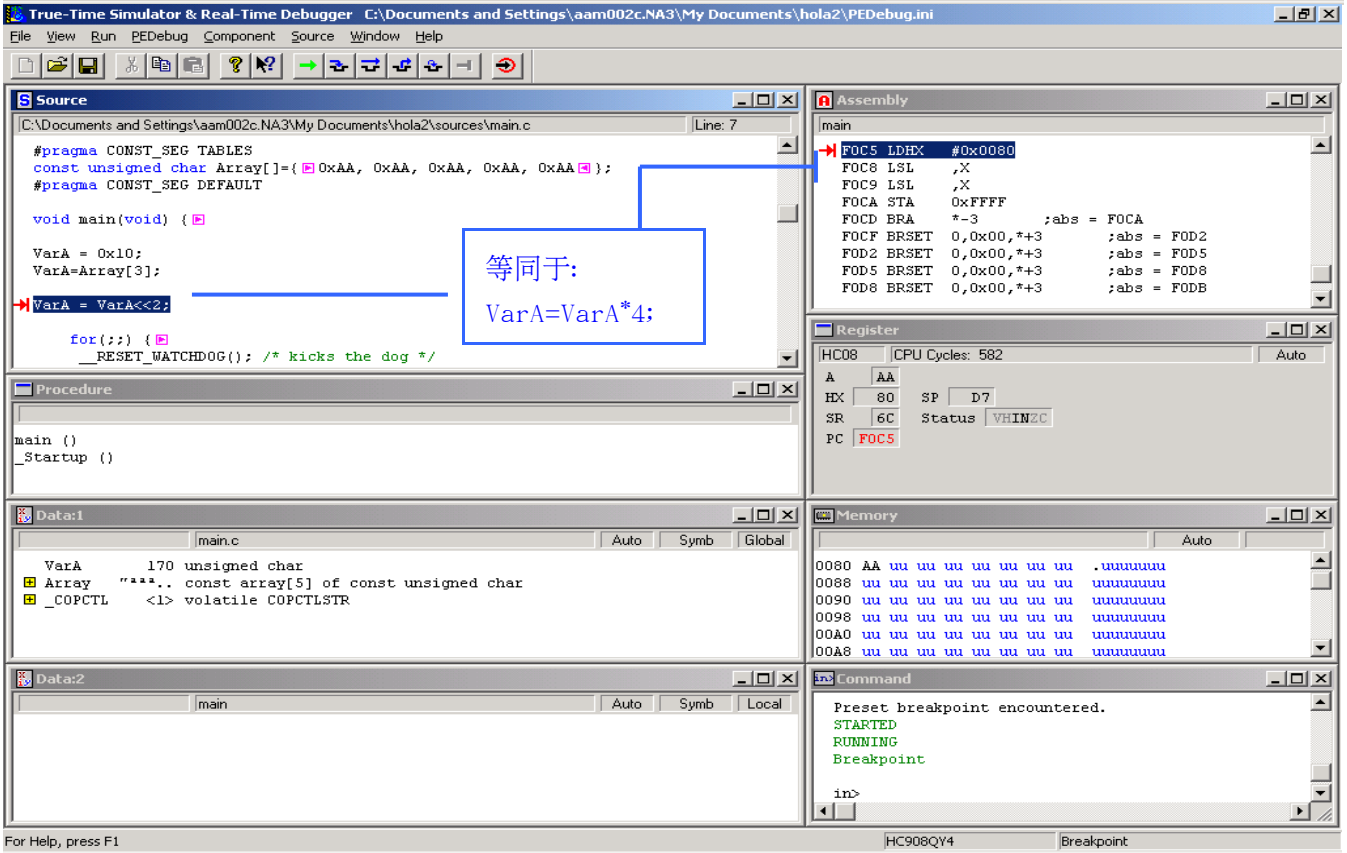
下列例子将演示编译器，基于应用做什么，决定哪一种操作用最少代价达到同样结果。

打开文件Lab10-Optimize1.mcp


Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help
main.c
#include <MC68HC908QT4.h> /* include peripheral declarations */
#pragma DATA_SEG BUFFER
unsigned char VarA;
#pragma DATA_SEG DEFAULT
#pragma CONST_SEG TABLES
const unsigned char Array[]={0xAA, 0xAA, 0xAA, 0xAA, 0xAA};
#pragma CONST_SEG DEFAULT
void main(void) {
VarA = 0x10;
VarA=Array[3];
VarA = VarA*3;
字节乘 3
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
Line 16 Col 14
default.prm
/* This is a .Prm file for the QT4*/
NAMES END
SEGMENTS
MYRAM = READ_WRITE 0x0080 TO 0x00A9;
RAM = READ_WRITE 0x00AA TO 0x00FF;
MYROM = READ_ONLY 0xEE00 TO 0xEEFF;
ROM = READ_ONLY 0xF000 TO 0xFDFF;
END
PLACEMENT
BUFFER INTO MYRAM;
TABLES INTO MYROM;
DEFAULT_ROM INTO ROM;
DEFAULT_RAM INTO RAM;
SSTACK INTO RAM;
END
STACKSIZE 0x30
VECTOR 0 _Startup
Line 4 Col 12

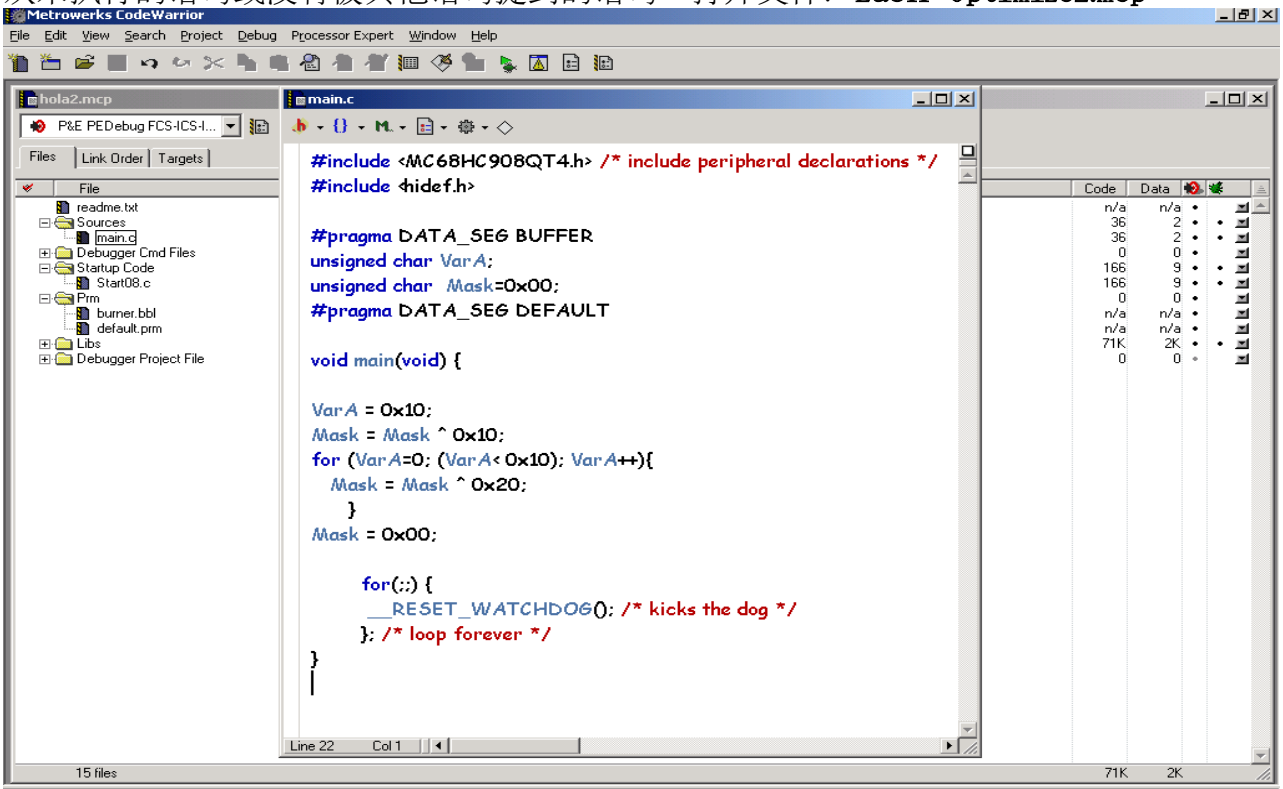

True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Source Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 7
#pragma CONST_SEG TABLES
const unsigned char Array[]={ 0xAA, 0xAA, 0xAA, 0xAA, 0xAA };
#pragma CONST_SEG DEFAULT
void main(void) {
VarA = 0x10;
VarA=Array[3];
VarA = VarA*3;
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
用 MUL 指令执行
Assembly
main
FOC5 LDX #0x03
FOC7 MUL
FOC8 STA 0x0080
FOCB STA 0xFFFF
FOCE BRA *-3 ;abs = FOCB
FOD0 BRSET 0,0x00,*+3 ;abs = FOD3
FOD3 BRSET 0,0x00,*+3 ;abs = FOD6
FOD6 BRSET 0,0x00,*+3 ;abs = FOD9
FOD9 BRSET 0,0x00,*+3 ;abs = FODC
Register
HC08
CPU Cycles: 582
Auto
A AA
HX 80
SP D7
SR 6C
Status VHINZC
PC FOC5
Procedure
main ()
_Startup ()
Data:1
main.c
Auto
Symb
Global
VarA 170 unsigned char
Array "aaa".. const array[5] of const unsigned char
_COPCTL <1> volatile COPCTLSTR
Memory
Auto
0080 AA uu uu uu uu uu uu uu .uuuuuuu
0088 uu uu uu uu uu uu uu uu uuuuuuuu
0090 uu uu uu uu uu uu uu uu uuuuuuuu
0098 uu uu uu uu uu uu uu uu uuuuuuuu
00A0 uu uu uu uu uu uu uu uu uuuuuuuu
00A8 uu uu uu uu uu uu uu uu uuuuuuuu
Data:2
main
Auto
Symb
Local
Command
Preset breakpoint encountered.
STARTED
RUNNING
Breakpoint
in>
For Help, press F1
HC908QY4
Breakpoint
Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help
main.c
#include <MC68HC908QT4.h> /* include peripheral declarations */
#pragma DATA_SEG BUFFER
unsigned char VarA;
#pragma DATA_SEG DEFAULT
#pragma CONST_SEG TABLES
const unsigned char Array[]={0xAA, 0xAA, 0xAA, 0xAA, 0xAA};
#pragma CONST_SEG DEFAULT
void main(void) {
VarA = 0x10;
VarA=Array[3];
VarA = VarA*4;
字节乘 4
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
Line 20 Col 26
default.prm
/* This is a .Prm file for the QT4*/
NAMES END
SEGMENTS
MYRAM = READ_WRITE 0x0080 TO 0x00A9;
RAM = READ_WRITE 0x00AA TO 0x00FF;
MYROM = READ_ONLY 0xEE00 TO 0xEEFF;
ROM = READ_ONLY 0xF000 TO 0xFDFF;
END
PLACEMENT
BUFFER INTO MYRAM;
TABLES INTO MYROM;
DEFAULT_ROM INTO ROM;
DEFAULT_RAM INTO RAM;
SSTACK INTO RAM;
END
STACKSIZE 0x30
VECTOR 0 _Startup
Line 4 Col 12

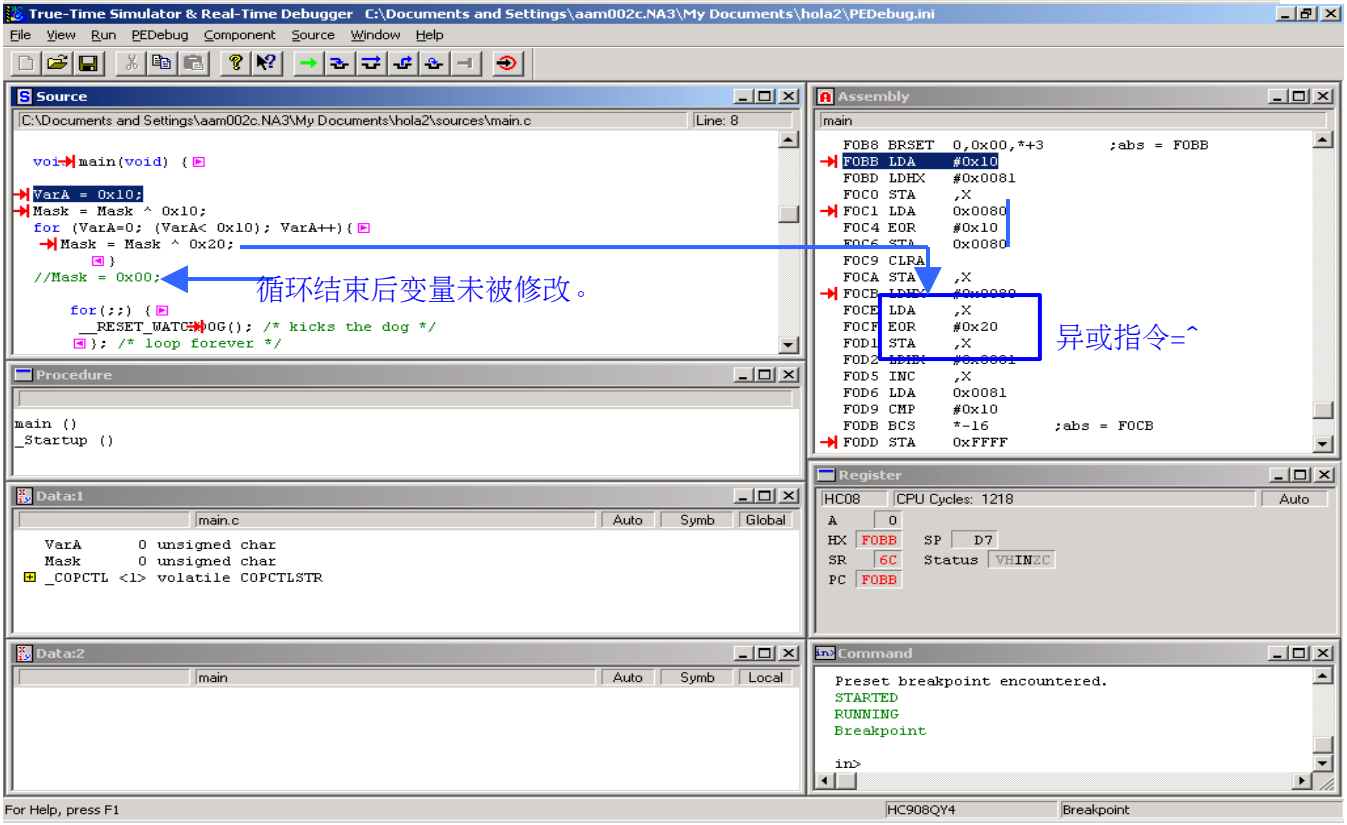

True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Source Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 9
#pragma CONST_SEG DEFAULT
void main(void) {
VarA = 0x10;
VarA=Array[3];
VarA = VarA*4;
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
用2个左移指令执行。用H：X作为指向我们将要
Assembly
main
F0C5 LDHX #0x0080
F0C8 LSL ,X
F0C9 LSL ,X
F0CA STA 0xFFFF
F0CD BRA *-3 ;abs = F0CA
F0CF BRSET 0,0x00,*+3 ;abs = F0D2
F0D2 BRSET 0,0x00,*+3 ;abs = F0D5
F0D5 BRSET 0,0x00,*+3 ;abs = F0D8
F0D8 BRSET 0,0x00,*+3 ;abs = F0DB
Register
HC08
CPU Cycles: 291
Auto
A AA
HX 80
SP D7
SR 6C
Status VHINZC
PC F0C5
Procedure
main ()
_Startup ()
Data:1
main.c
Auto
Symb
Global
VarA 170 unsigned char
Array "aaa.. const array[5] of const unsigned char
_COPCTL <1> volatile COPCTLSTR
Memory
Auto
0080 AA uu uu uu uu uu uu uu .uuuuuuu
0088 uu uu uu uu uu uu uu uu uuuuuuuu
0090 uu uu uu uu uu uu uu uu uuuuuuuu
0098 uu uu uu uu uu uu uu uu uuuuuuuu
00A0 uu uu uu uu uu uu uu uu uuuuuuuu
00A8 uu uu uu uu uu uu uu uu uuuuuuuu
Data:2
main
Auto
Symb
Local
Command
Preset breakpoint encountered.
STARTED
RUNNING
Breakpoint
in>
For Help, press F1
HC908QY4
Breakpoint

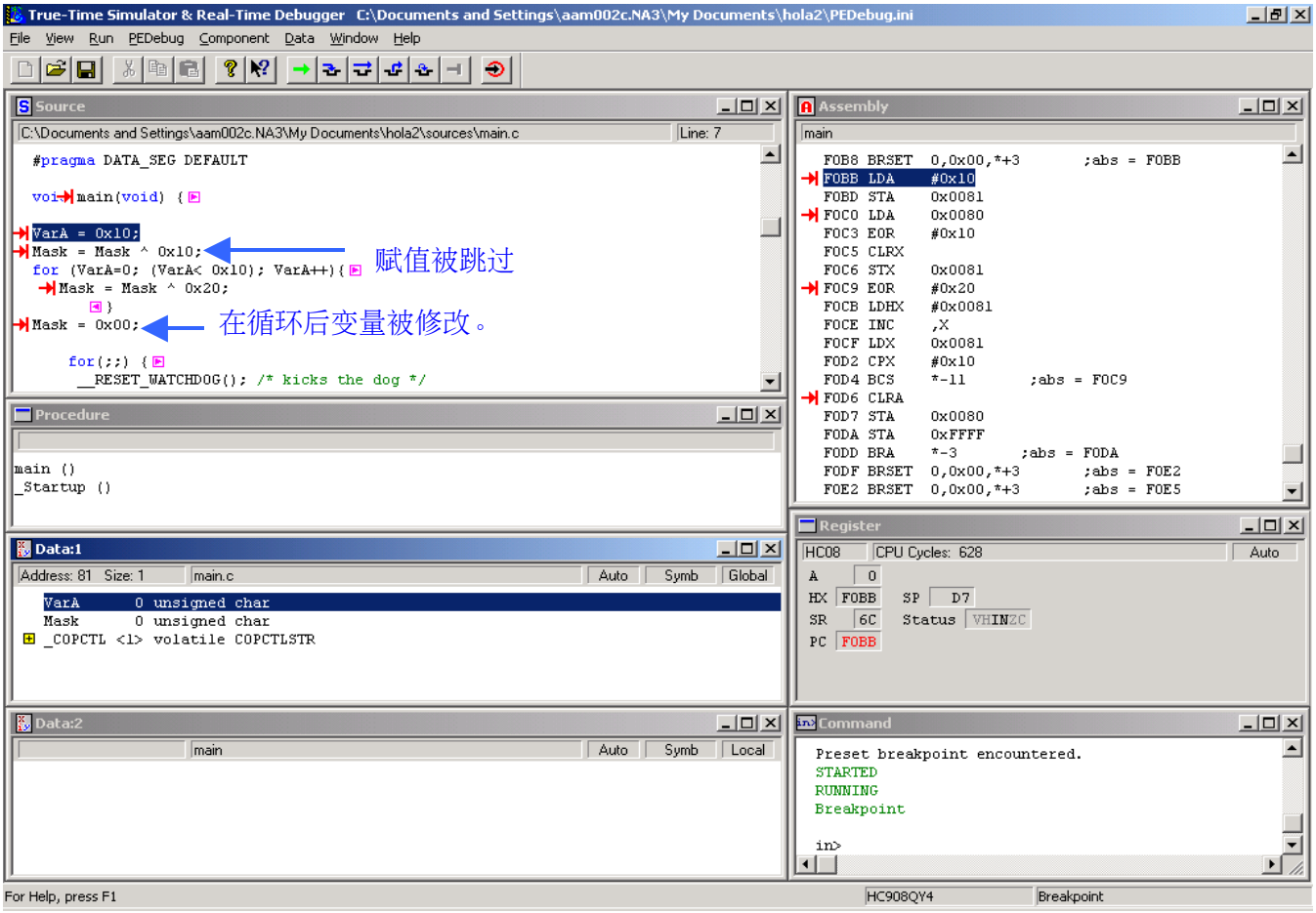

True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Source Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 7
#pragma CONST_SEG TABLES
const unsigned char Array[]={0xAA, 0xAA, 0xAA, 0xAA, 0xAA};
#pragma CONST_SEG DEFAULT
void main(void) {
VarA = 0x10;
VarA=Array[3];
VarA = VarA<<2;
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}
等同于:
VarA=VarA*4;
Assembly
main
F0C5 LDHX #0x0080
F0C8 LSL ,X
F0C9 LSL ,X
F0CA STA 0xFFFF
F0CD BRA *-3 ;abs = F0CA
F0CF BRSET 0,0x00,*+3 ;abs = F0D2
F0D2 BRSET 0,0x00,*+3 ;abs = F0D5
F0D5 BRSET 0,0x00,*+3 ;abs = F0D8
F0D8 BRSET 0,0x00,*+3 ;abs = F0DB
Register
HC08
CPU Cycles: 582
Auto
A AA
HX 80
SP D7
SR 6C
Status VHINZC
PC F0C5
Procedure
main ()
_Startup ()
Data:1
main.c
Auto
Symb
Global
VarA 170 unsigned char
Array "aaa.. const array[5] of const unsigned char
_COPCTL <1> volatile COPCTLSTR
Memory
Auto
0080 AA uu uu uu uu uu uu uu .uuuuuuu
0088 uu uu uu uu uu uu uu uu uuuuuuuu
0090 uu uu uu uu uu uu uu uu uuuuuuuu
0098 uu uu uu uu uu uu uu uu uuuuuuuu
00A0 uu uu uu uu uu uu uu uu uuuuuuuu
00A8 uu uu uu uu uu uu uu uu uuuuuuuu
Data:2
main
Auto
Symb
Local
Command
Preset breakpoint encountered.
STARTED
RUNNING
Breakpoint
in>
For Help, press F1
HC908QY4
Breakpoint

2）死代码消除

对于死代码消除，编译器优化应用程序并不为没被使用的语句产生可执行代码。移除逻辑上从未执行的语句或没有被其他语句提到的语句。打开文件：Lab11-Optimize2.mcp。

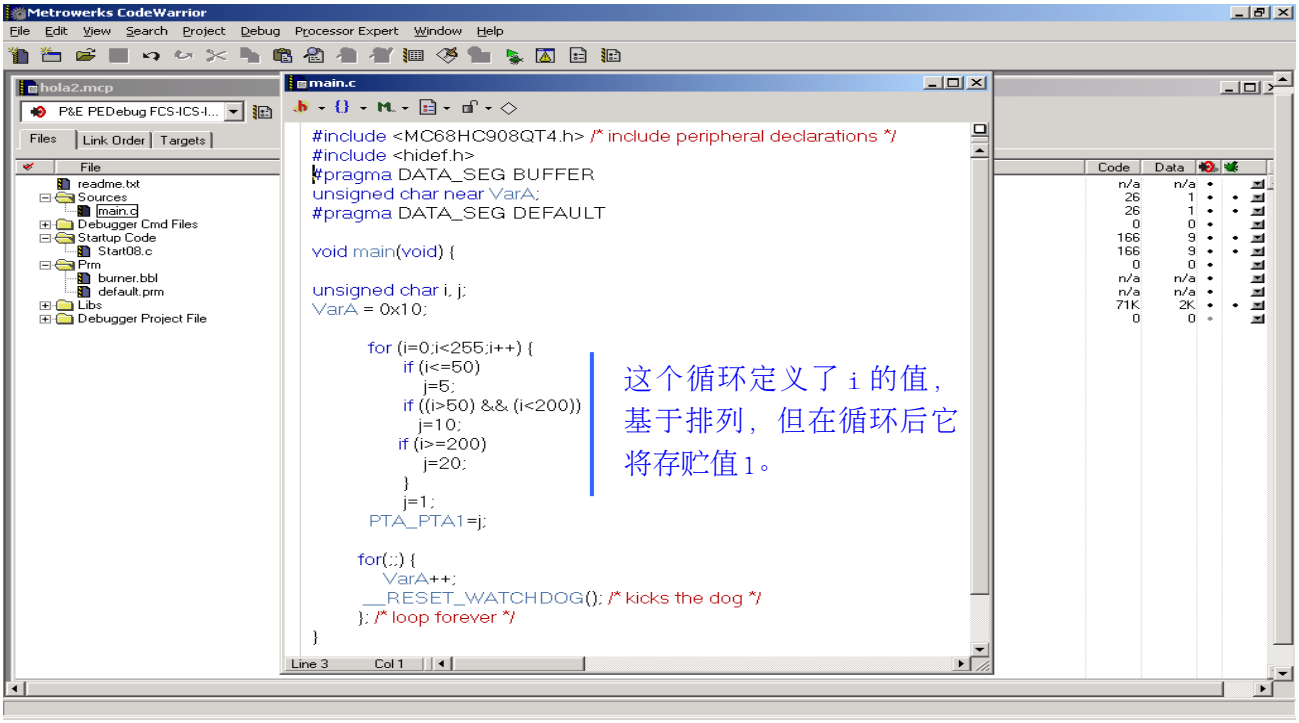


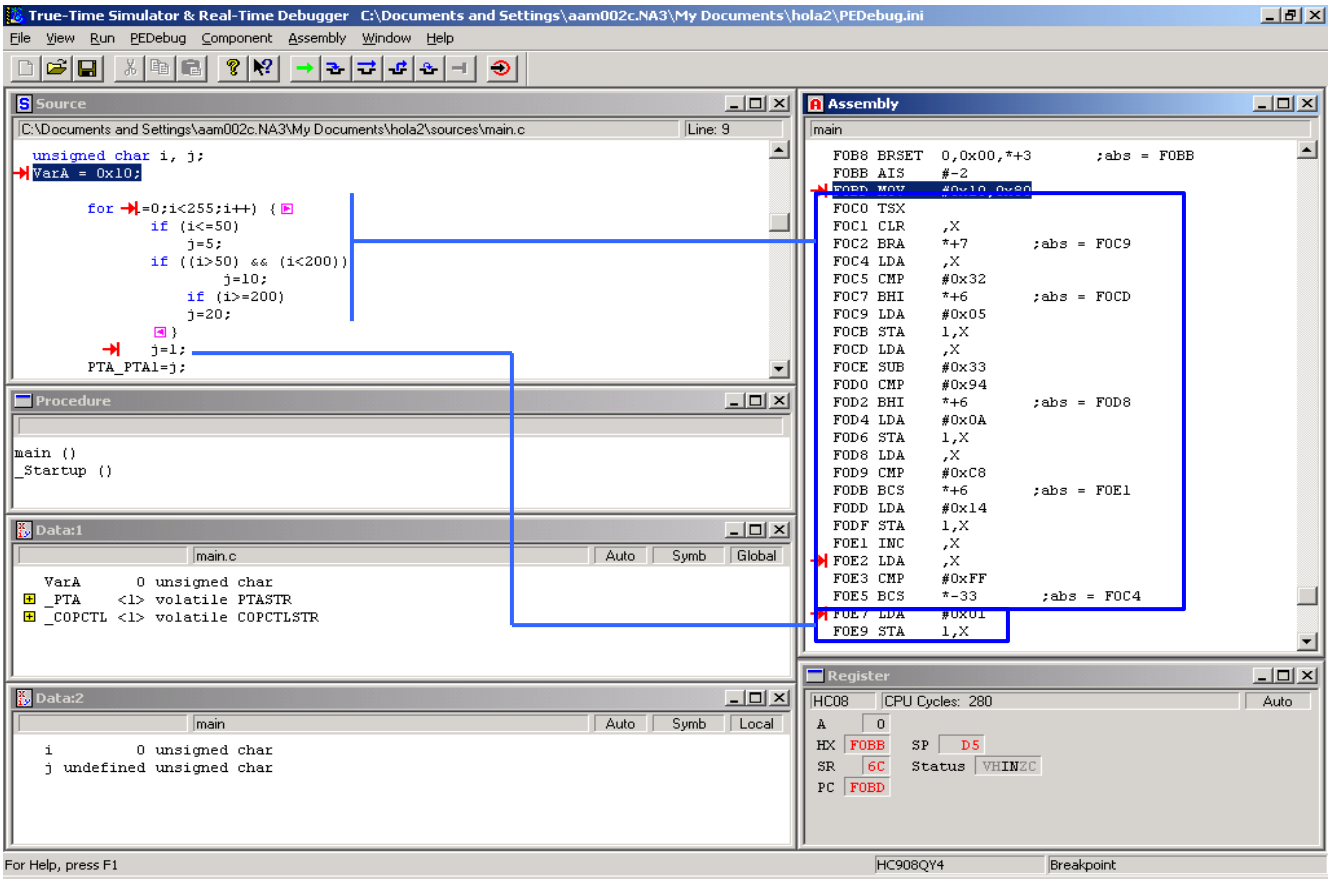




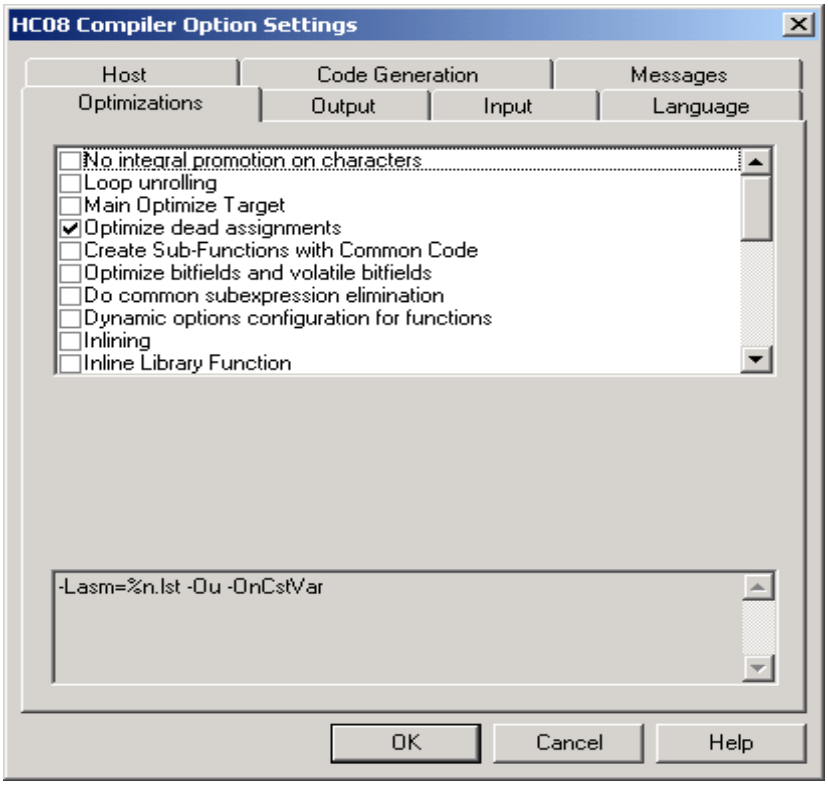
3）死赋值

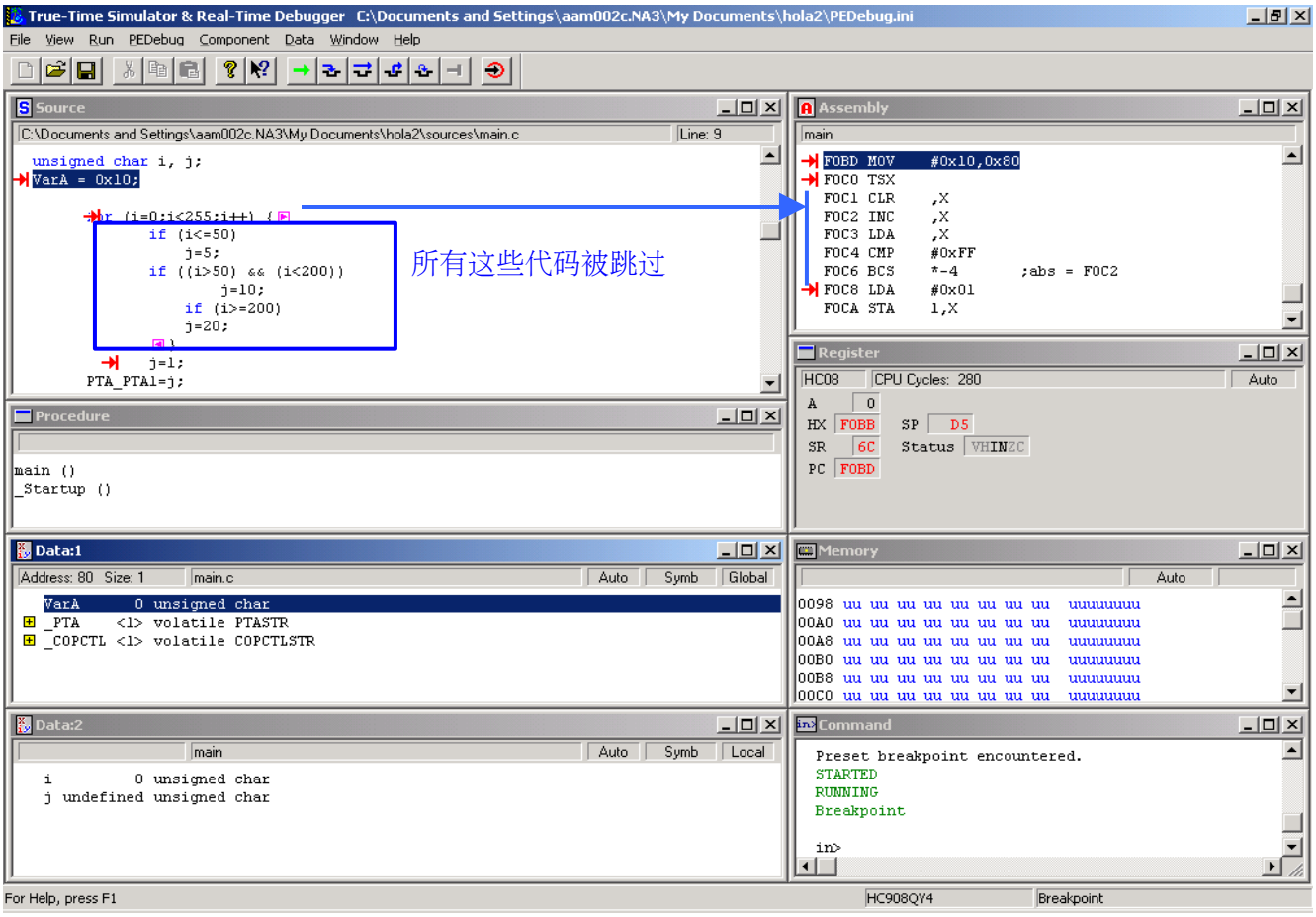
消除死赋值，编译器移去变量在再次赋值之前没有被使用的赋值。在下面编译器优化的例子，我们将演示通过改变编译器的优化设置，达到改变CodeWarrior产生代码的方式。打开文件Lab12-Optimize3.mcp:

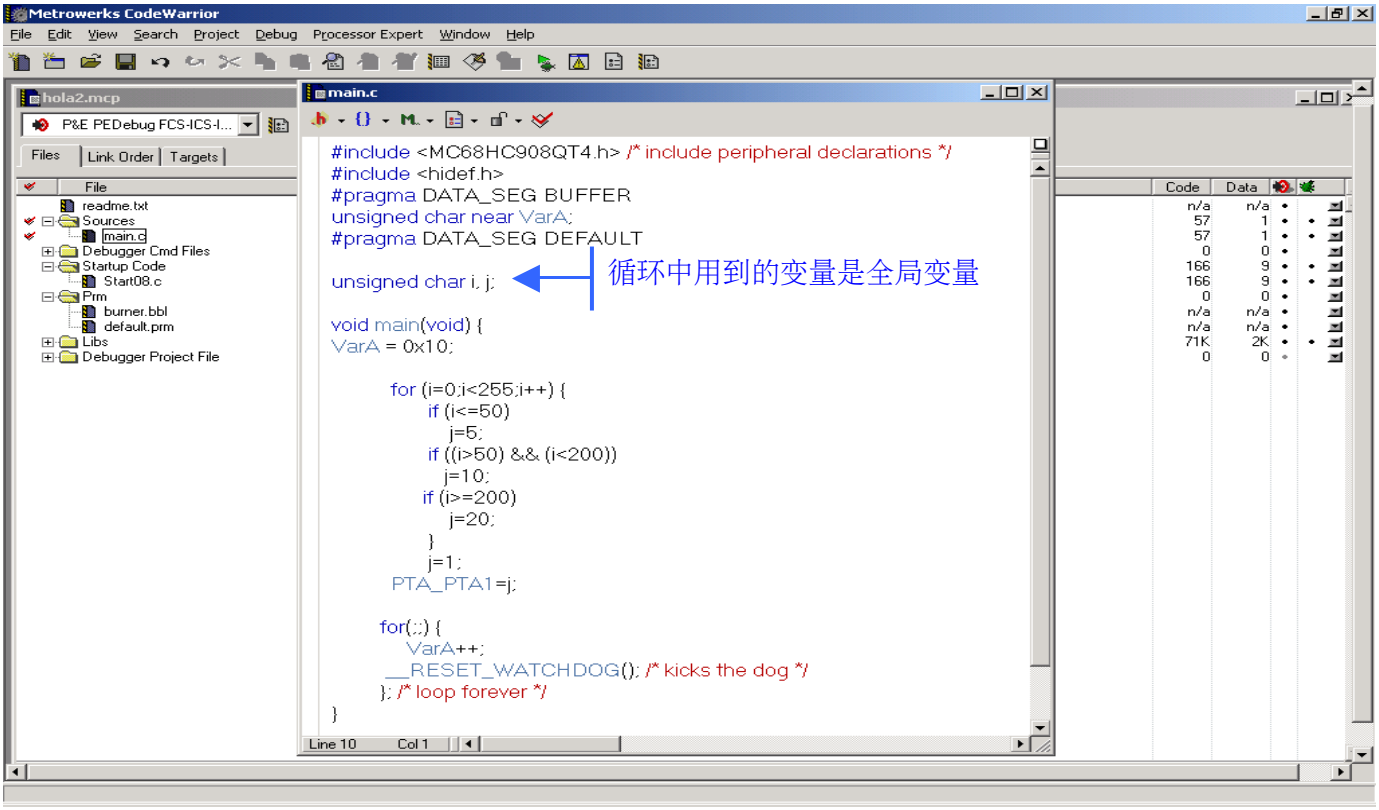


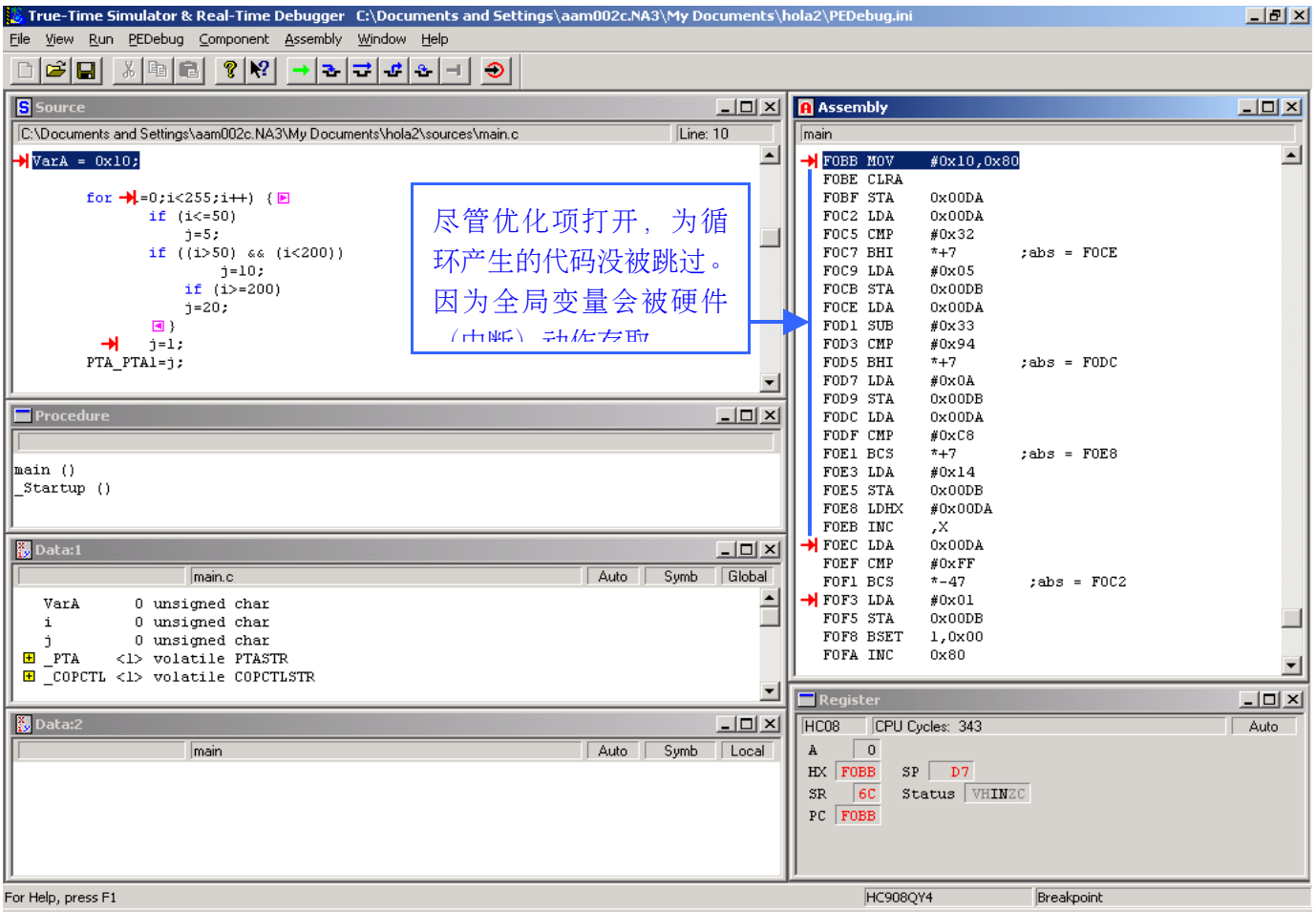


4) Codewarrior HC08 编译器选项设置


HC08 Compiler Option Settings
Host
Code Generation
Messages
Optimizations
Output
Input
Language
No integral promotion on characters
Loop unrolling
Main Optimize Target
Optimize dead assignments
Create Sub-Functions with Common Code
Optimize bitfields and volatile bitfields
Do common subexpression elimination
Dynamic options configuration for functions
Inlining
Inline Library Function
-Lasm=%n.lst -Ou -OnCstVar
OK
Cancel
Help

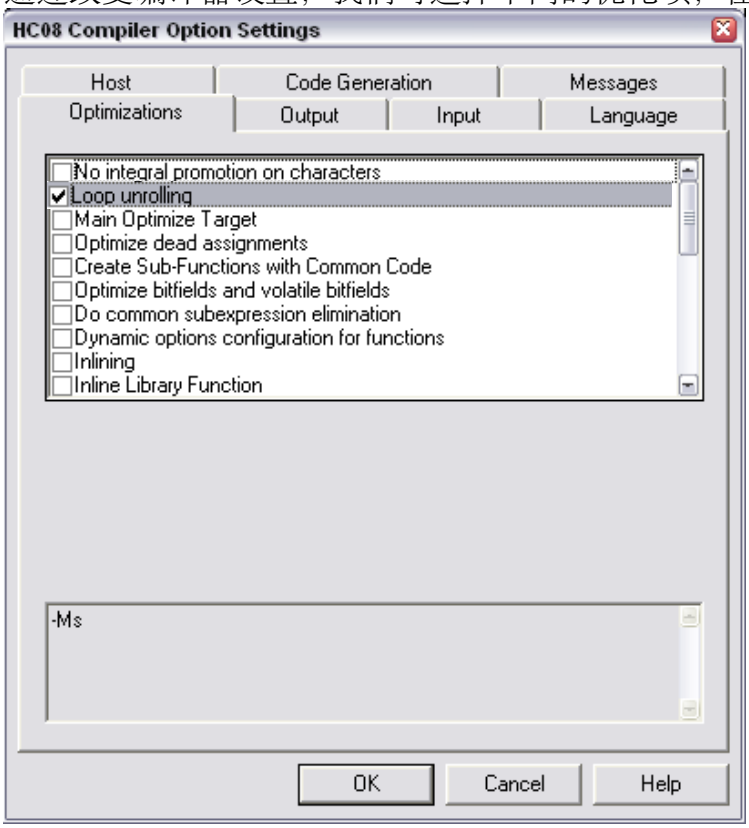

True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Data Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 9
unsigned char i, j;
VarA = 0x10;
if (i<=50)
j=5;
if ((i>50) && (i<200))
j=10;
if (i>=200)
j=20;
j=1;
PTA_PTA1=j;
所有这些代码被跳过
Assembly
main
F0BD MOV #0x10,0x80
F0C0 TSX
F0C1 CLR ,X
F0C2 INC ,X
F0C3 LDA ,X
F0C4 CMP #0xFF
F0C6 BCS *-4 ;abs = F0C2
F0C8 LDA #0x01
F0CA STA 1,X
Register
HC08
CPU Cycles: 280
Auto
A 0
HX F0BB SP D5
SR 6C Status VHINZC
PC F0BD
Procedure
main ()
_Startup ()
Data:1
Address: 80 Size: 1
main.c
Auto
Symb
Global
VarA 0 unsigned char
_PTA <1> volatile PTASTR
_COPCTL <1> volatile COPCTLSTR
Memory
Auto
0098 uu uu uu uu uu uu uu uu uuuuuuuu
00A0 uu uu uu uu uu uu uu uu uuuuuuuu
00A8 uu uu uu uu uu uu uu uu uuuuuuuu
00B0 uu uu uu uu uu uu uu uu uuuuuuuu
00B8 uu uu uu uu uu uu uu uu uuuuuuuu
00C0 uu uu uu uu uu uu uu uu uuuuuuuu
Data:2
main
Auto
Symb
Local
i 0 unsigned char
j undefined unsigned char
Command
Preset breakpoint encountered.
STARTED
RUNNING
Breakpoint
in>
For Help, press F1
HC908QY4
Breakpoint


Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help
hola2.mcp
P&E PEDebug FCS-ICS-I...
Files Link Order Targets
File
readme.txt
Sources
main.c
Debugger Cmd Files
Startup Code
Start08.c
Prm
burner.bbl
default.prm
Libs
Debugger Project File
main.c
#include <MC68HC908QT4.h> /* include peripheral declarations */
#include <hidef.h>
#pragma DATA_SEG BUFFER
unsigned char near VarA;
#pragma DATA_SEG DEFAULT
unsigned char i, j;
循环中用到的变量是全局变量
void main(void) {
VarA = 0x10;
for (i=0;i<255;i++) {
if (i<=50)
j=5;
if ((i>50) && (i<200))
j=10;
if (i>=200)
j=20;
}
j=1;
PTA_PTA1=j;
for(;;) {
VarA++;
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
Line 10 Col 1
Code Data

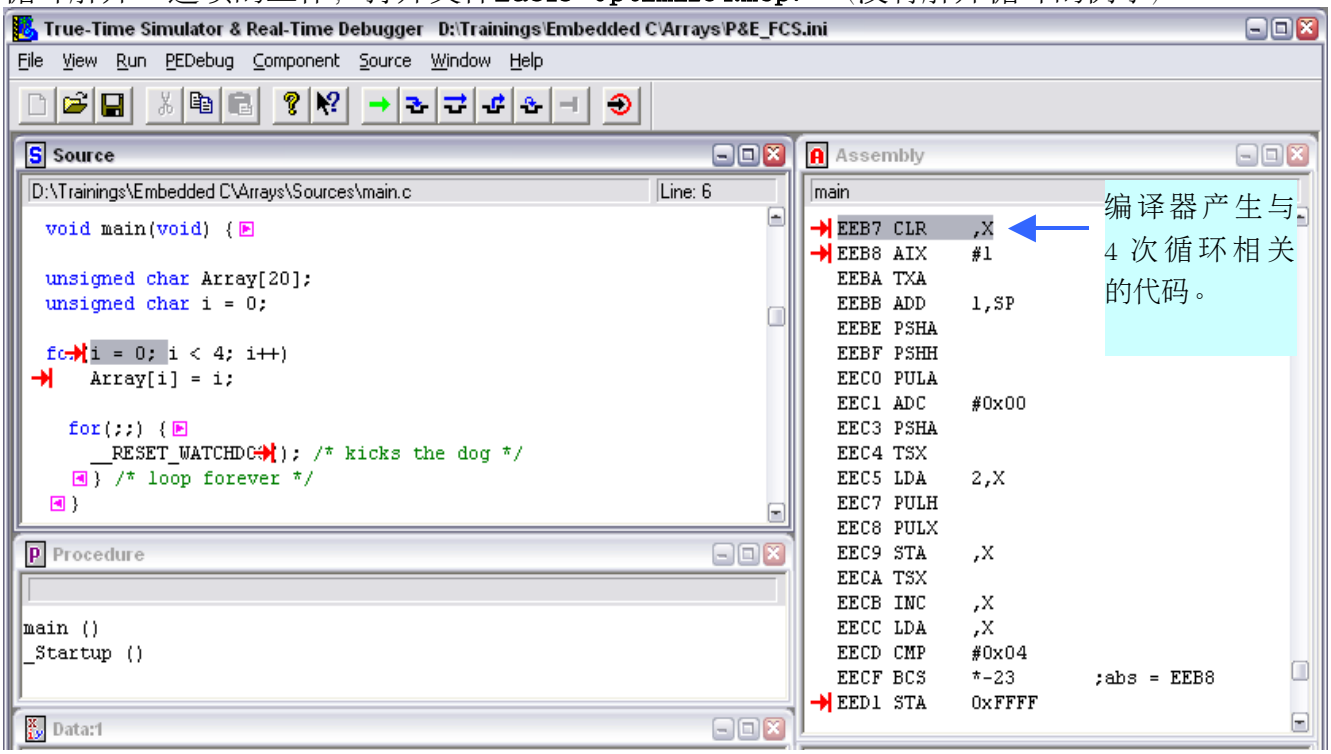

True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Assembly Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 10
VarA = 0x10;
for (i=0;i<255;i++) {
if (i<=50)
j=5;
if ((i>50) && (i<200))
j=10;
if (i>=200)
j=20;
}
j=1;
PTA_PTA1=j;
尽管优化项打开，为循环产生的代码没被跳过。因为全局变量会被硬件
Assembly
main
F0BB MOV #0x10,0x80
F0BE CLRA
F0BF STA 0x00DA
F0C2 LDA 0x00DA
F0C5 CMP #0x32
F0C7 BHI *+7 ;abs = F0CE
F0C9 LDA #0x05
F0CB STA 0x00DB
F0CE LDA 0x00DA
F0D1 SUB #0x33
F0D3 CMP #0x94
F0D5 BHI *+7 ;abs = F0DC
F0D7 LDA #0x0A
F0D9 STA 0x00DB
F0DC LDA 0x00DA
F0DF CMP #0xC8
F0E1 BCS *+7 ;abs = F0E8
F0E3 LDA #0x14
F0E5 STA 0x00DB
F0E8 LDHX #0x00DA
F0EB INC ,X
F0EC LDA 0x00DA
F0EF CMP #0xFF
F0F1 BCS *-47 ;abs = F0C2
F0F3 LDA #0x01
F0F5 STA 0x00DB
F0F8 BSET 1,0x00
F0FA INC 0x80
Procedure
main ()
_Startup ()
Data:1
main.c
Auto Symb Global
VarA 0 unsigned char
i 0 unsigned char
j 0 unsigned char
_PTA <1> volatile PTASTR
_COPCTL <1> volatile COPCTLSTR
Data:2
main
Auto Symb Local
Register
HC08 CPU Cycles: 343
Auto
A 0
HX F0BB SP D7
SR 6C Status VHINZC
PC F0BB
For Help, press F1
HC908QY4
Breakpoint

5）循环解开

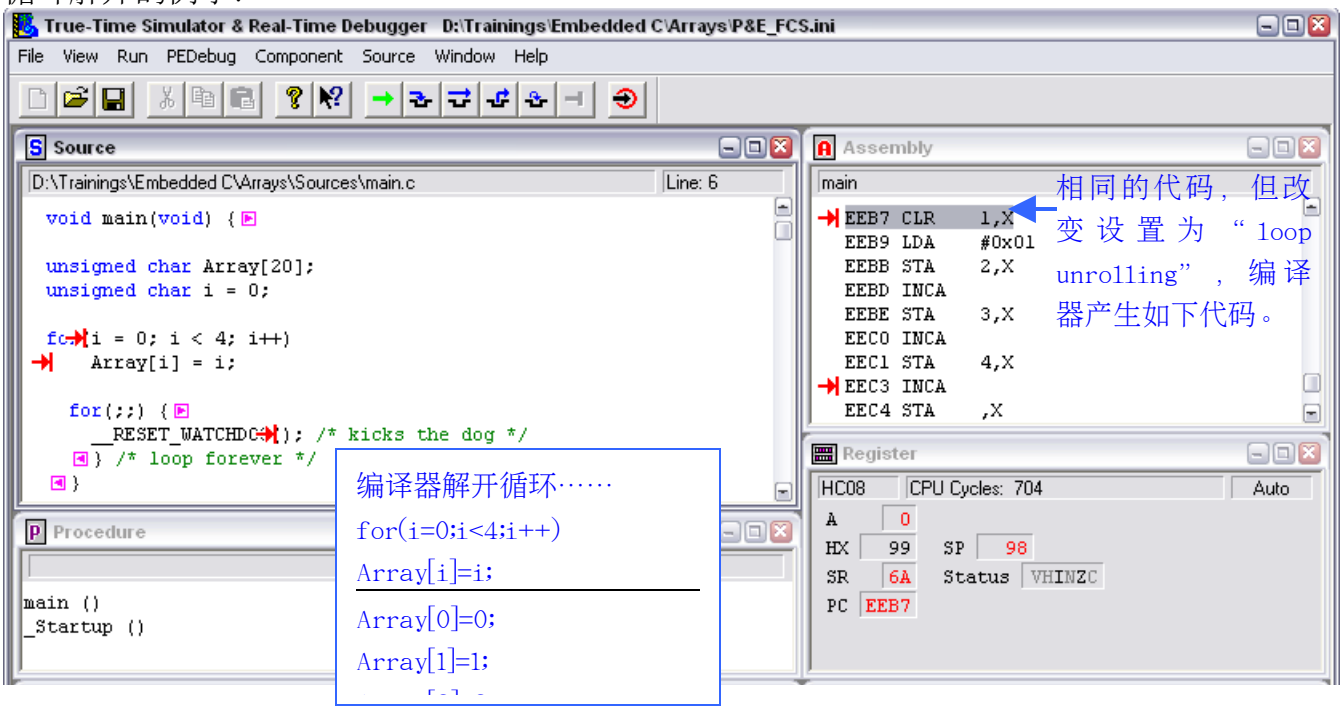
通过改变编译器设置，我们可选择不同的优化项，在生成代码时将会有差异。



循环体内完全相同的代码目的是展开更多操作分支以完成上面的测试。下列例子演示“循环解开”选项的工作，打开文件Lab13-Optimize4.mcp: （没有解开循环的例子）



循环解开的例子：



6）更多编译器优化选项：

全局寄存器定位　将频繁使用的变量的值存贮在寄存器中，而不是存贮器中。

分支优化　为减少分支指令，合并和重建立即代码转换分区。

算法优化　将频繁使用的计算指令替换为快速等效的指令，产生同样的结果。

表达式简化　将复杂算法表达式简化为等效的表达式。

消除公共子表达式　将雍余表达式替换为单一表达式。

多作复制　将一个变量的多个事件替换为单一事件。

孔径优化　将本地优化事务应用到小段代码。

生存范围分割　减小变量生存时间以达到定位优化。短的变量生存时间减少寄存器泄漏。

循环不变量移动　移动静态计算到循环之外。

循环转换　重组循环目标代码以减少设置和测试完成开销。

基于生存时间的寄存器定位　在特定的程序中，只要没有语句同时使用那些变量，用同一个处理器寄存器存贮不同的变量。

指令顺序安排　重组指令顺序以减少寄存器和处理器资源的冲突。

7）条件编译

编译指示符：#if、#else、#elif、#endif

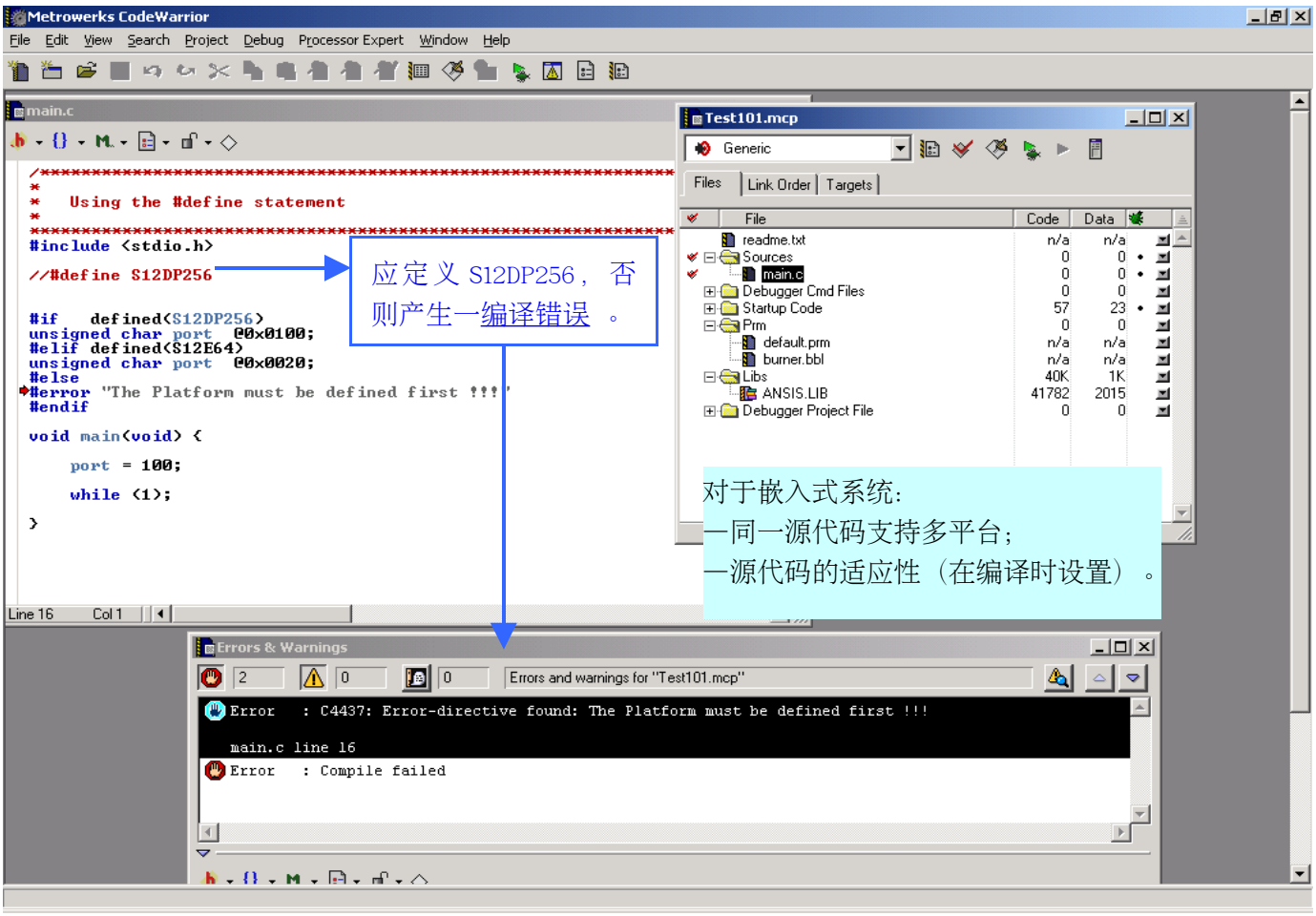
这些指示符均用于条件编译：

```
#if <constant-expression>
#else OR #elif <constant-expression>
```

#endif

只有当条件表达式的值不为零时，才编译跟有#if指示符的行。否则以后的行都被跳过直到遇到匹配的#else或endif。

#error定义一个用于显示的编译错误。



NITRON 培训

一、Nitron 简介

1、68HC908QT/QY系列共公特性

核心：0.5u HC08

总线速度：8MHz（最小指令125ns）

定时器：带IC、OC或PWM的2通道16位定时器

电压：2.7V~5.5V

温度：-40~+85℃

外

额外温度要求请联系市场部

其它特性：LVI COP自动从STOP、KBI中唤醒。

RAM：128字节

SCI/SPI：（软件可编程/已有应用笔记）

外部中断：IRQ、KBI、定时IC

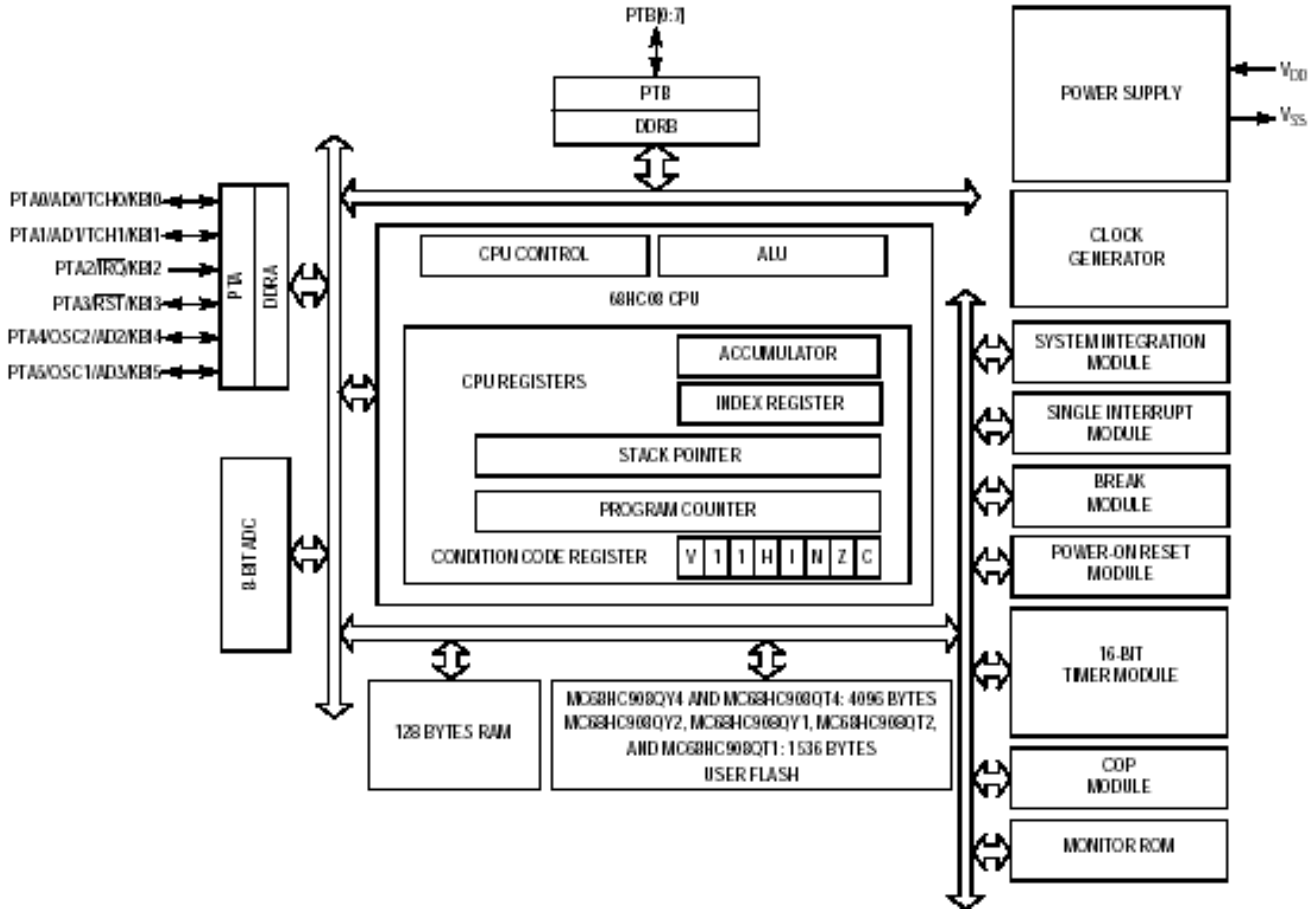
振荡器：可调整（±25%）内部振荡器3.2MHz。

（可调范围±5%到105℃），外接RC，时钟，谐振电路/晶振

2、设备特性

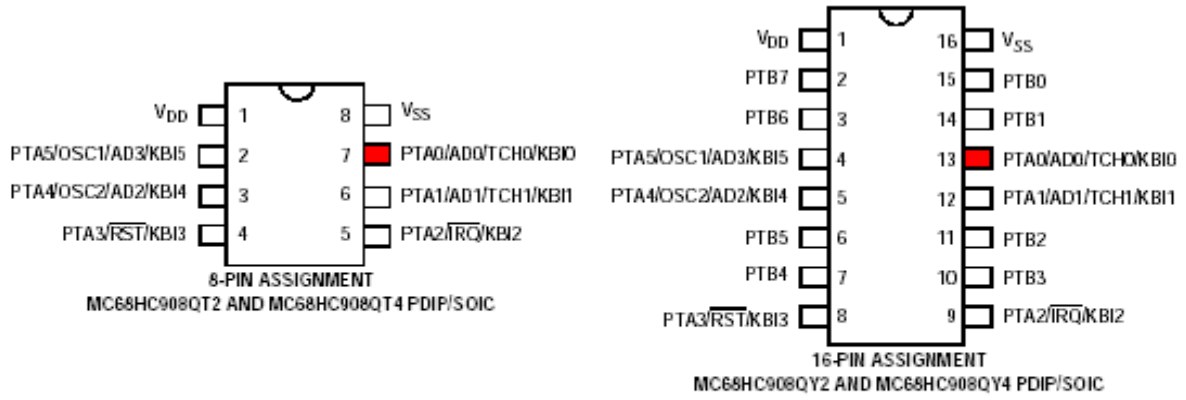
不同点	68HC908QT1	68HC908QT2	68HC908QT4	68HC908QY1	68HC908QT2	68HC908QT4
FLASH	1.5K字节	1.5K字节	4K字节	1.5K字节	1.5K字节	4K字节
ADC	—	4通道8位	4通道8位	—	4通道8位	4通道8位
封装	8脚SOIC/PDIP	8脚SOIC/PDIP	8脚SOIC/PDIP	16脚SOIC/PDIP/TSSOP	16脚SOIC/PDIP/TSSOP	16脚SOIC/PDIP/TSSOP

二、微控制器描述



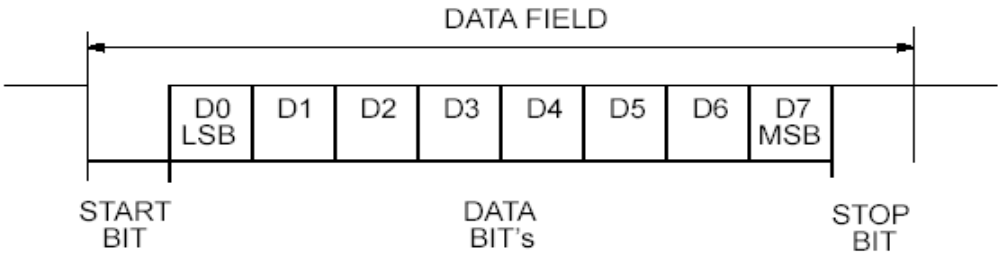
2．1 NITRON实现异步通信

为了使用已存在的板上硬件，必须用定时器模拟异步通信口，可用PTA0/TCH0实现双向异步通信口。

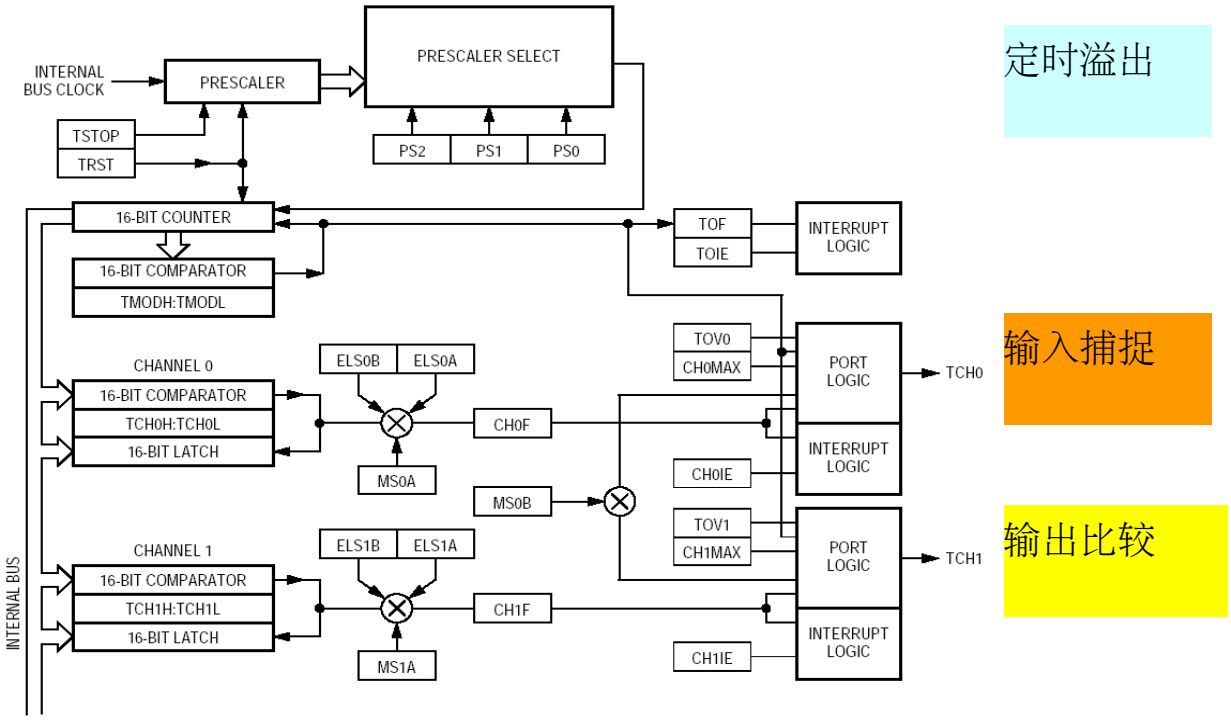


■注意：应当小心，避免充突。

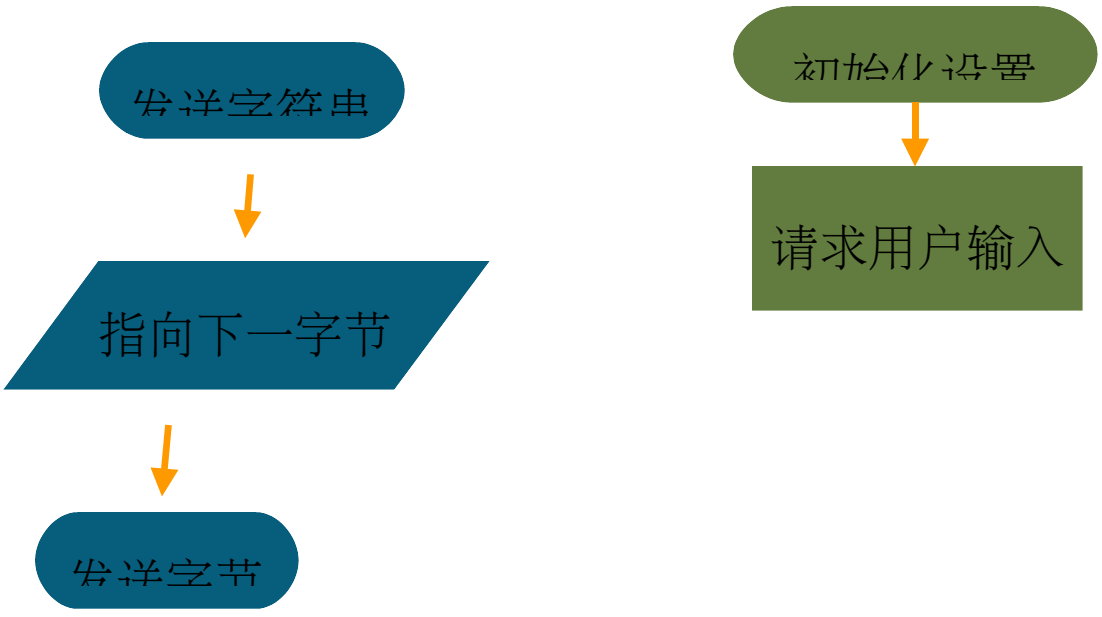
异步通信帧由：一个起动位（支配位）、8位数据、一个停止位（接收状态）构成。



2．2 定时接口模块（TIM）

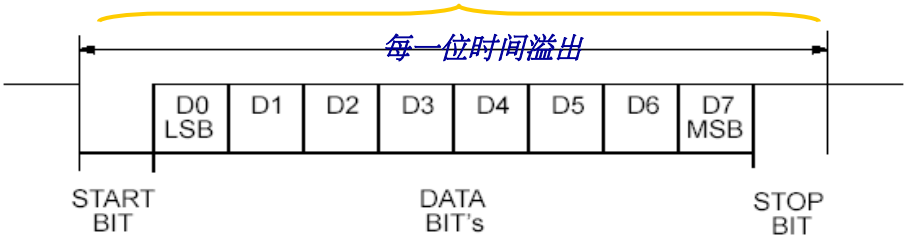


2．3 流程图示例（1）

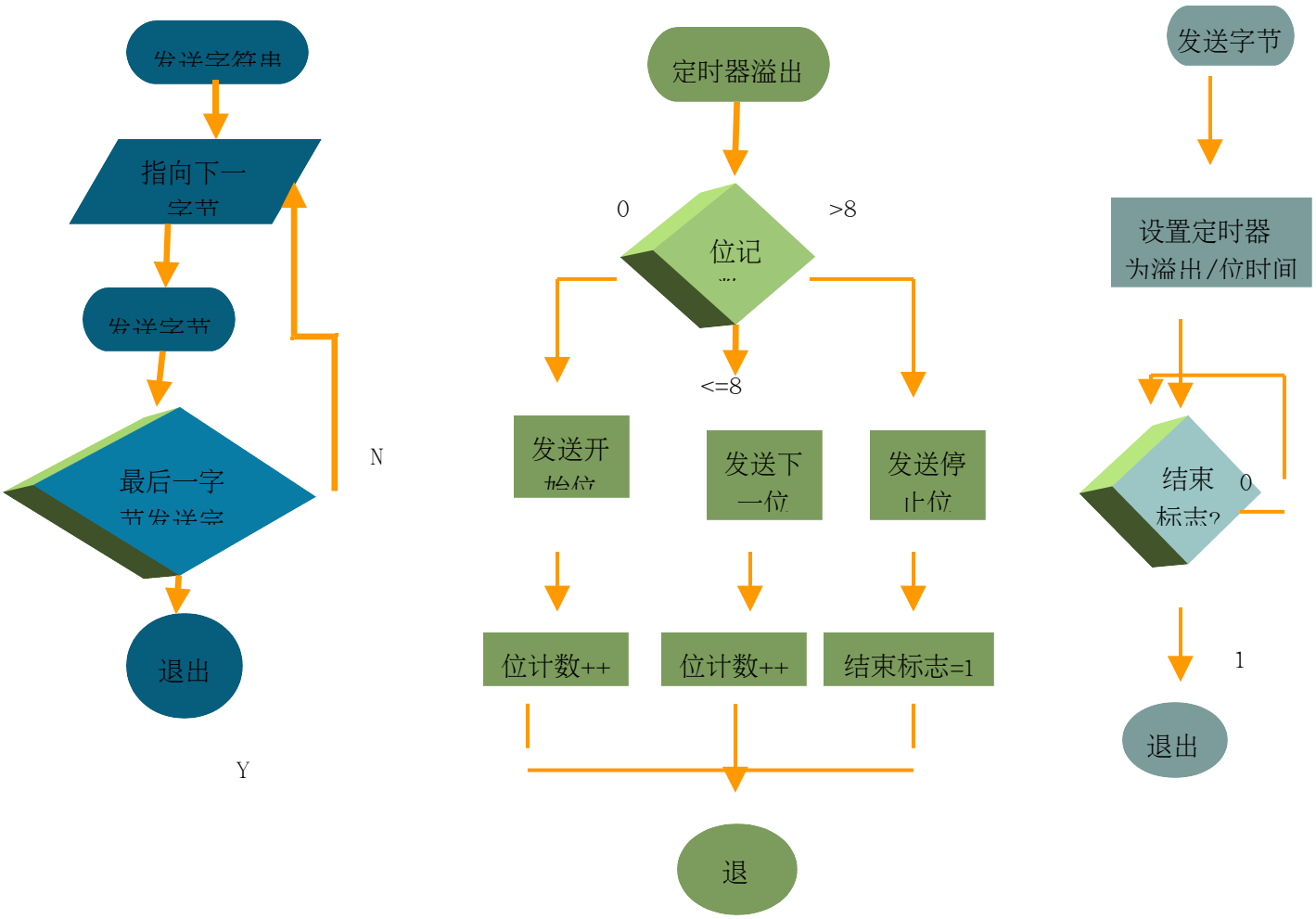


通过异步通信口发送一个字节

定时器必须设为每位时间溢出。每个溢出中断发送一个新位，包括起始和停止位。



2．4 流程图示例（2）



UART_SendByte(Byte)代码

打开文件Lab14-UART.mcp

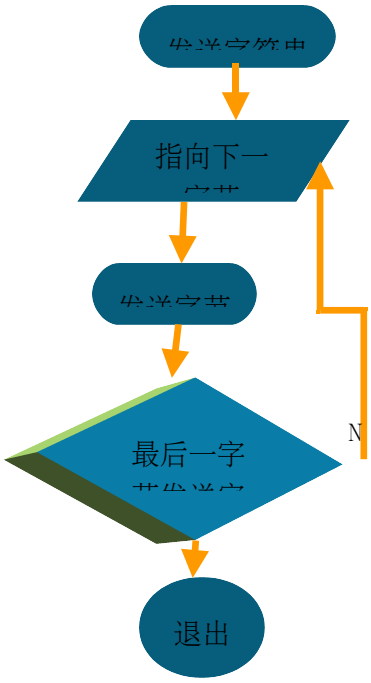
```
void UART_SendByte (unsigned char byte_to_send)
{
Tx_SHR_Reg=byte_to_send;
bitcount=0;
```

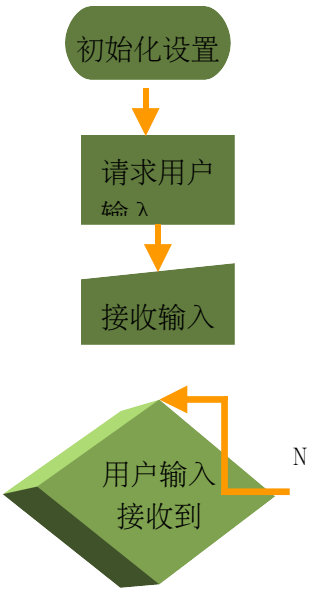
```
overmode=TRANSMITTING;
ChangeTimerConfig(OVERFLOW|_1BITTIME); //设置定时器每位时间溢出出
}
```

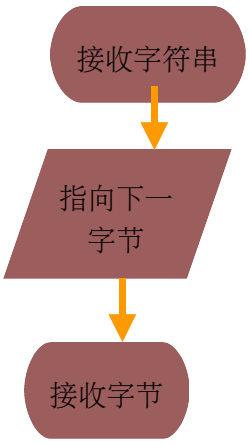
设置开始位、发送字节和每中断停止位，如下例所示：

```
void interrupt 6 overflow_isr (void)
 {…
 DDRA|=0x01;
          if (!bitcount)
         {  SET_TX_LOW;
         bitcount++;  }
          else if (bitcount<=8)
       {
        if (Tx_SHR_Reg&1)
        SET_TX_HIGH;
          else
          SET_TX_LOW;
     Tx_SHR_Reg>>=1;
        bitcount++;
     }
            else
       {  SET_TX_HIGH;
         bitcount=0;
         Flag=1;     }…
```

2．5 流程图示例（3）：







Y

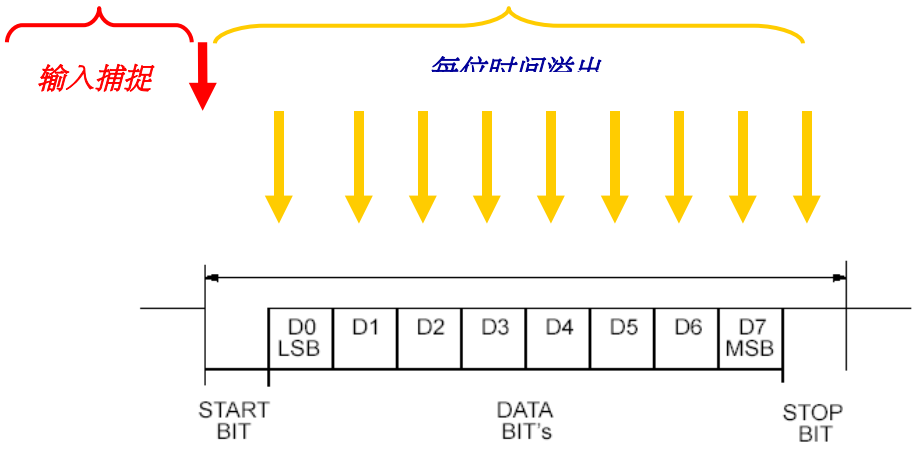
从异步通信口接收一个字节

为了捕捉起始位，定时器设置为输入捕捉。

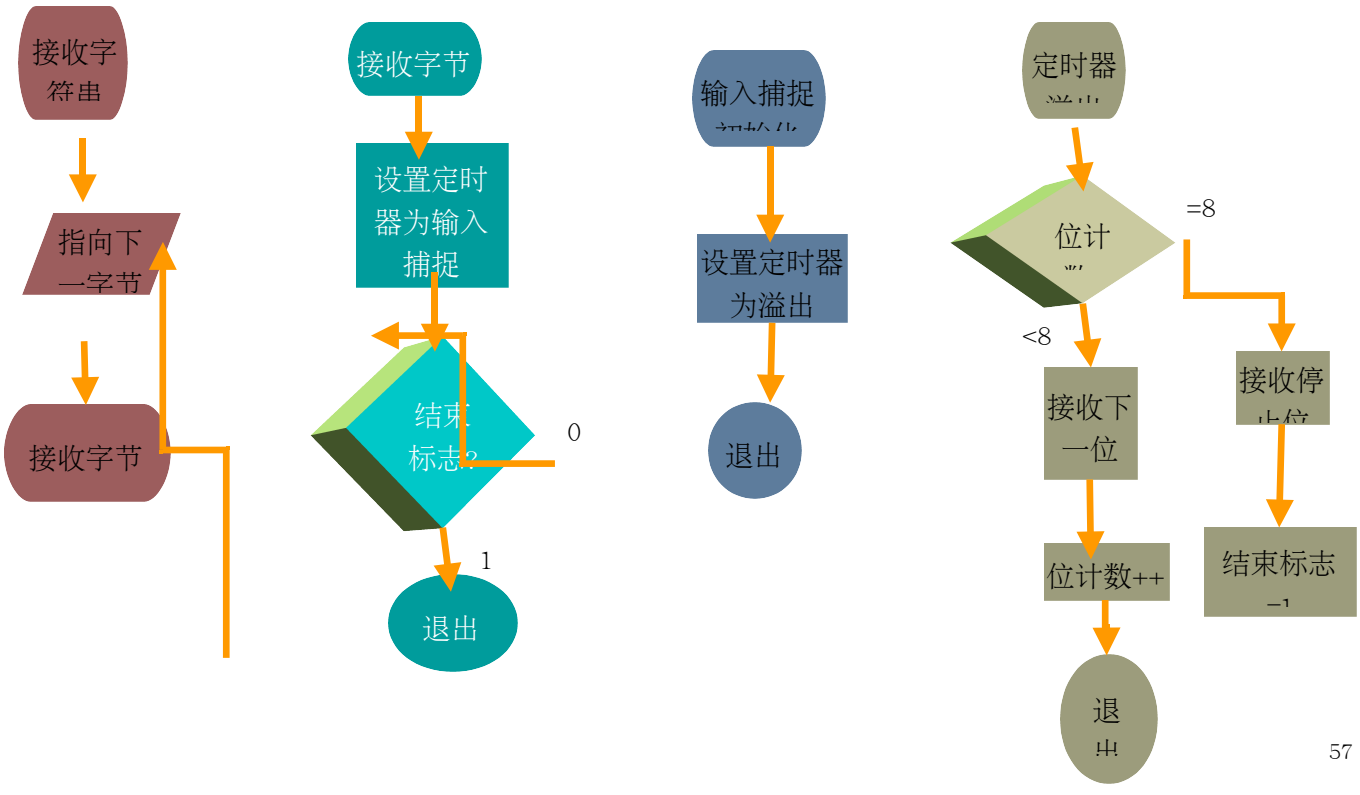
从那时起，定时器设置为每位时间溢出：

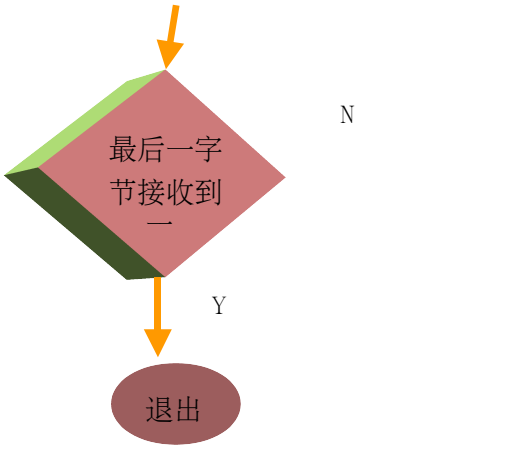
$$\text{位时间}=\frac{\text{总线频率}}{\text{比例因子}*\text{波特率}}$$

每溢出中断引脚状态被检查，并放置一个新位。



2．6流程图示例（4）





UART_GetByte()代码:

设置输入捕捉:

```
unsigned char UART_GetByte (void) {
  ChangeTimerConfig(INPCAPTURE|MAX);
  bitcount=0;
  …
  return Rx_SHR_Reg;
}
```

当检测到开始位，设置定时器为溢出:

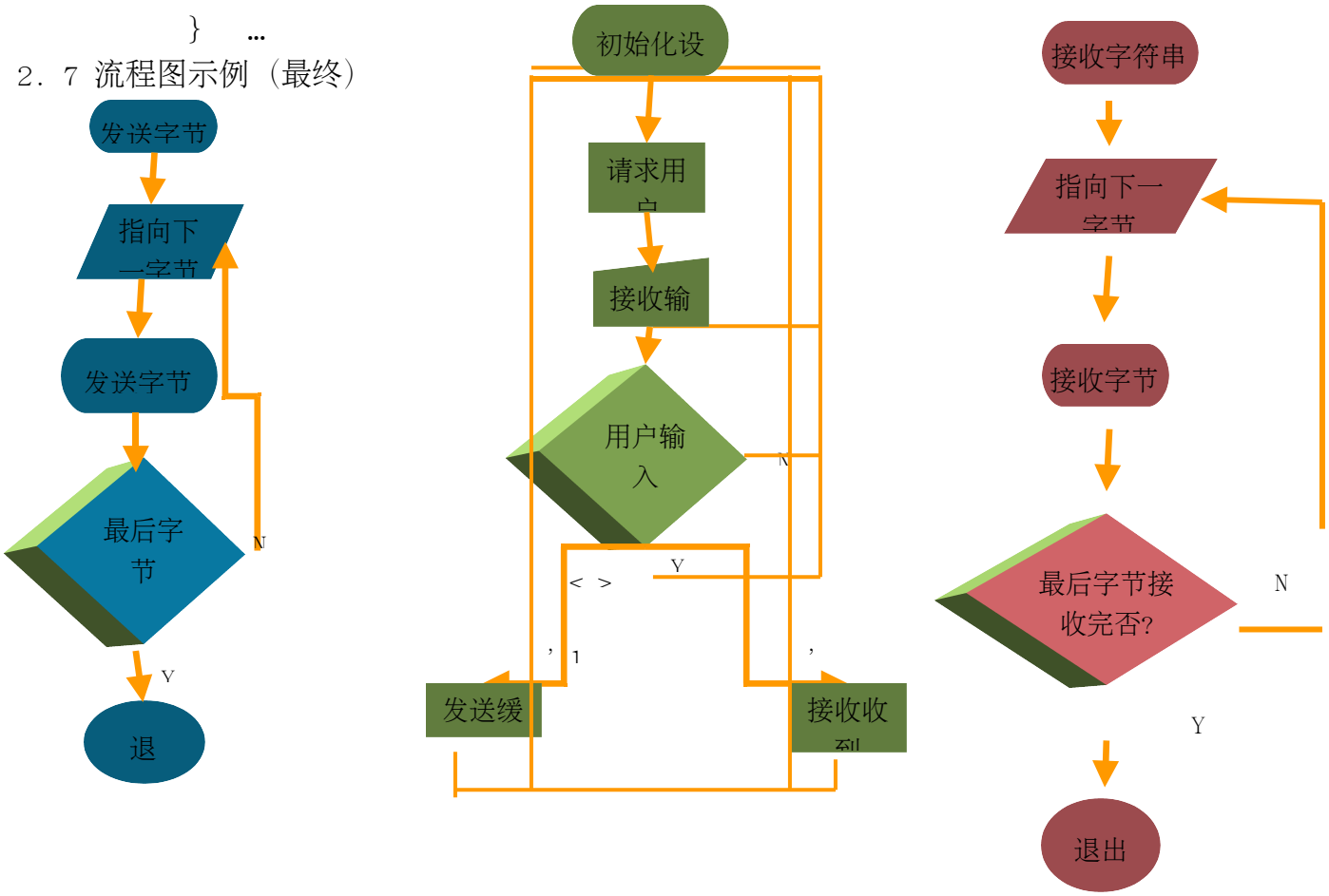
```
void interrupt 4 inpcapture (void)
 {
      ACK_INPCAPTUREINT;
       overmode=RECEIVING_DATA;
       bitcount=0;
       ChangeTimerConfig(OVERFLOW|_1BITTIME);
}
```

读入8位然后退出:

```
void interrupt 6 overflow_isr (void)
 {
   ACK_OVERFLOWINT;
   if (overmode==RECEIVING_DATA)
        {
         DDRA&=0xFE;
             if (bitcount<8)
           {
```

```
            bitcount++;
             Rx_SHR_Reg>>=1;
             if (is_RX_HIGH)
             Rx_SHR_Reg|=0x80;
          }    …
```

2．7 流程图示例（最终）



主程序:

例程发送一个位于FLASH中的消息并等待用户输入:

1) 请缓存（初始化为“FREESCALE”）;

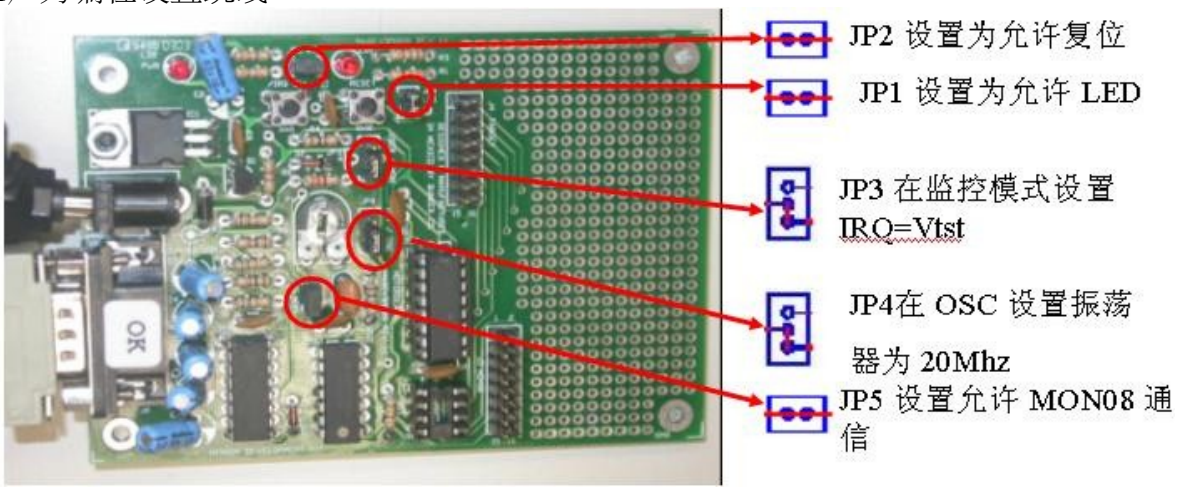
2) 写缓存。

```
temp= UART_GetByte();     //等待用户输入
   switch (temp)
 {
 case '1':
   SEND_BREAK();
   UART_SendString(pSend, SEND_LENGTH);    //如为‘1’，发送缓存内容
   break;
case '2':
```

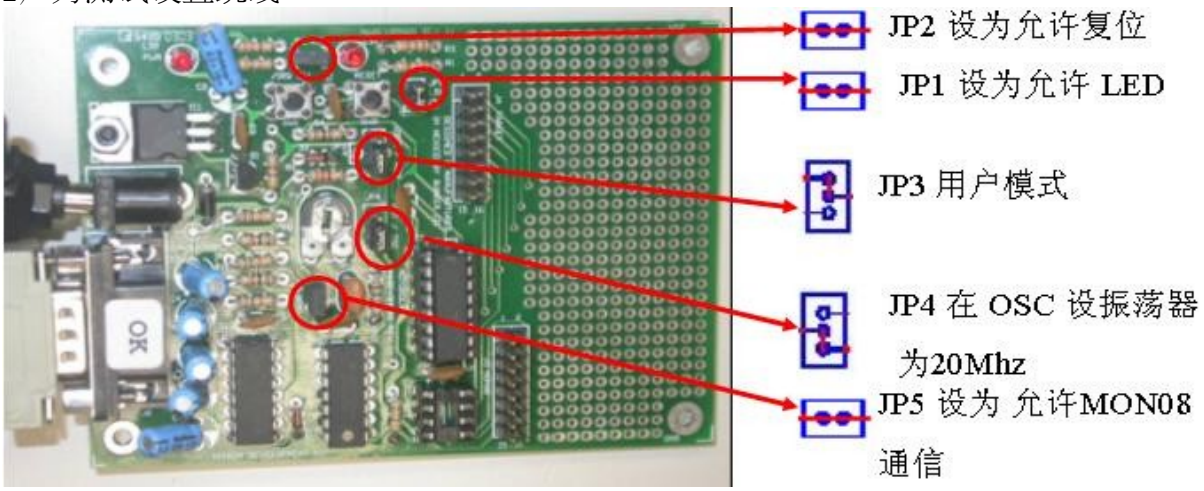
```
    SEND_BREAK();
    UART_GetString(&My_Receive[0],RECEIVE_LENGTH);  //当为‘2’时，接收缓冲[9]
    pSend=&My_Receive[0];
    break;
 }
```

2．8打开试验板电源

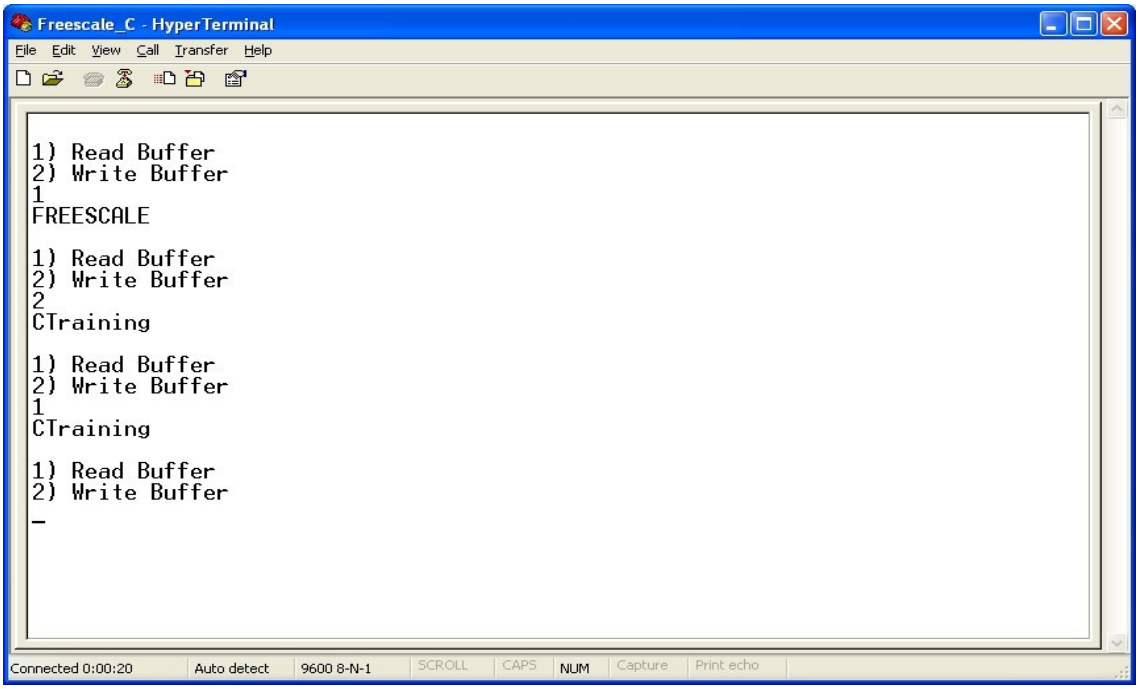
1）为编程设置跳线



2）为测试设置跳线



3）用超级终端测试程序：



记住那个引脚双向使用，因此必须小心以免冲突。

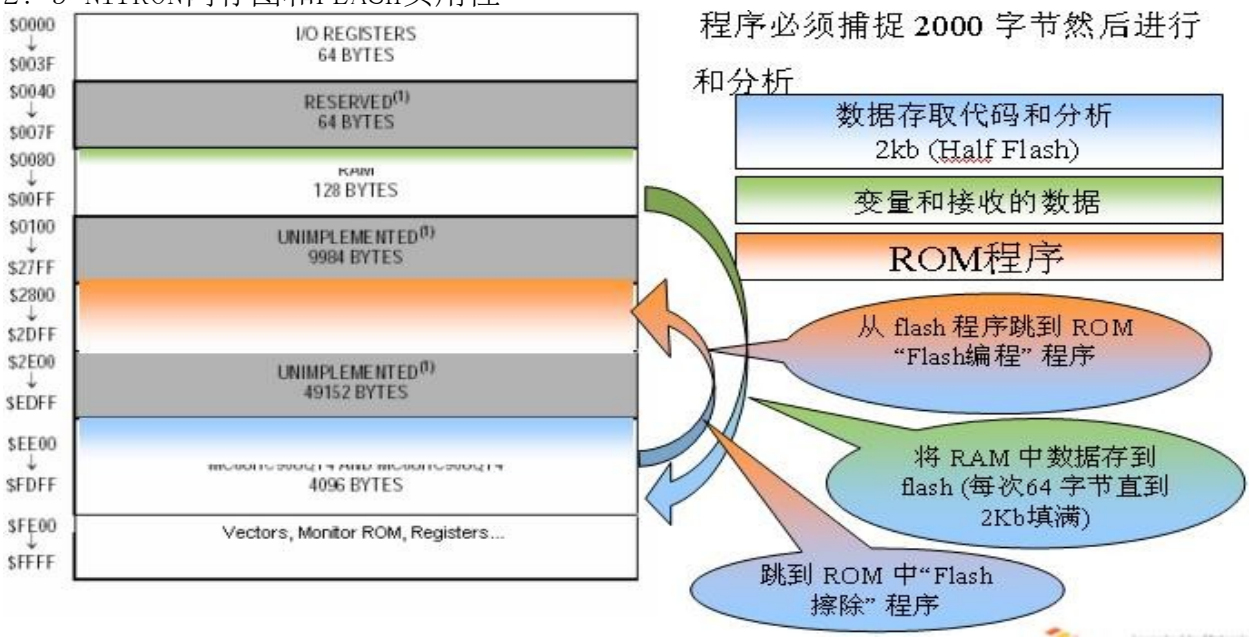
2．8 NITRON FLASH

NITRON系列使用第2代0.5u Flash技术，“基于Flash的系统，提供极其快速的编程，加上最大的灵活性和创造性。在开发期间，由于使用Flash，你的设计能多次重复编程，或最后的制造。升级甚至能现场进行。任何时候可修复缺陷，容易交付新特性或为客户增进性能、安全性和保密性。

Flash保证最少10,000次写/擦（典型100K次）。

Flash存贮器每页64字节。一页包含2行每行32字节。Flash一次编程一行。但是擦除却是一整页。编程速度为每ms10字节(10x EEPROM)。擦除时间为4ms。

2．9 NITRON内存图和FLASH实用性



3．0 NITRON ROM上程序

为减少用户代码、编程和测试目的，公共程序都在ROM中执行。它们是：

名称	地址
GETBYTE	0x2800
RDVRRNG	0x2803
ERARNGE	0x2806
PGRRNGE	0x2809
DELNUS	0x280C

ROM上程序描述：

Routine Name	GETBYTE	RDVRRNG	PRGRNGE	ERARNGE	DELNUS
Routine Description	Gets a byte of data from comm port	Reads and/or verifies a range of locations	Programs a range* of locations	Erases** a page or the entire range	Delays for n x 12 µs
Entry Conditions	Comm port configured as an input	H:X contains first address of range; LADDR contains last address read; Acc is tested to see if read data goes to comm port (Acc = $00) or to DATA;DATA contains data against which to compare read data	H-X contains first address of range; LADDR contains last address to be programmed; DATA contains data used during programming; CPUSPD contains 4 * f_{OP}	H:X contains any address in range to be erased; range size specified by control byte; CPUSPD contains 4 * f_{OP}	X contains time ÷ 12 of delay (in µs); Acc contains 4 times f_{OP}
Exit Conditions	Acc is loaded with byte received	C bit is set if good compare; Acc contains checksum; DATA may contain read FLASH data	H:X contains next address after range just programmed	Preserves contents of H:X (address passed)	
Subroutines Called	Get_Bit		DELNUS	DELNUS	
Variables Read		LADDR, DATA ARRAY	CONTROL BYTE, LADDR, DATA ARRAY, CPUSPD	Control Byte, CPUSPD	
Variables Modified		DATA ARRAY			
Stack Used	4 bytes	6 bytes	7 bytes	5 bytes	3 bytes

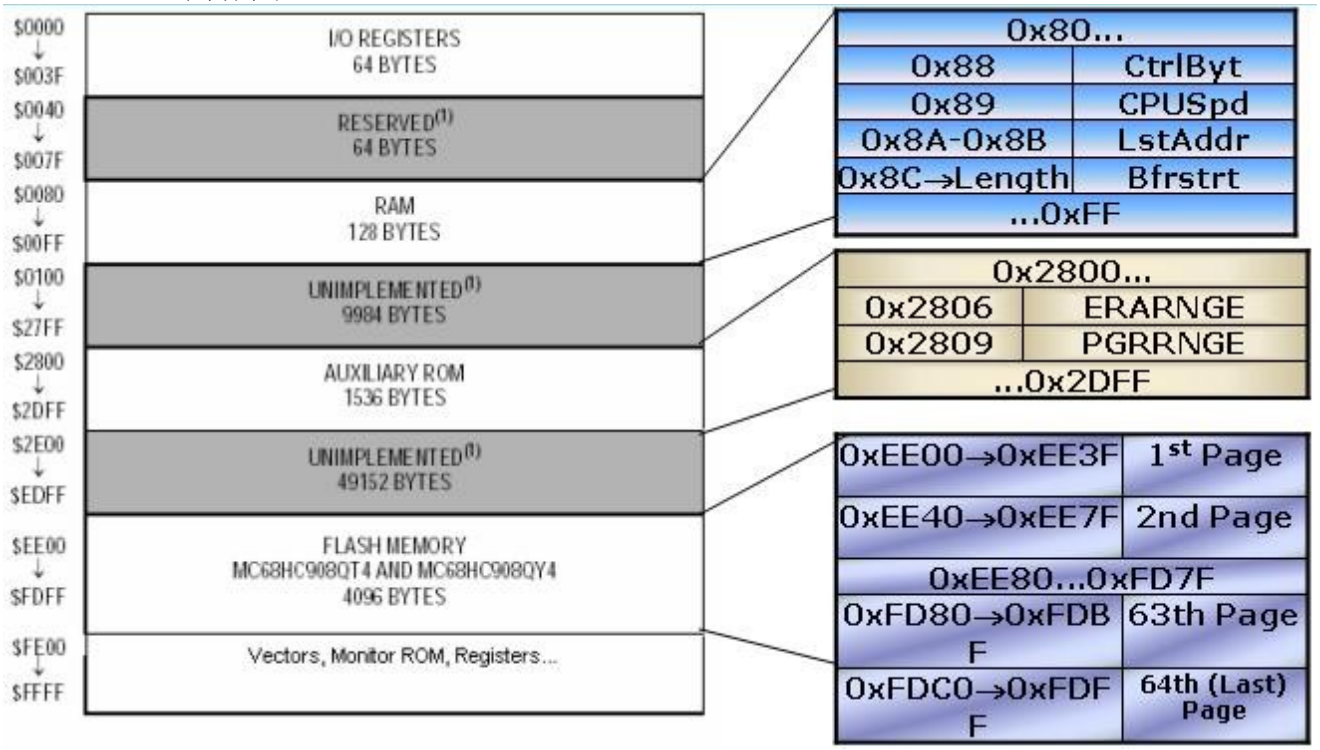
3．1 RAM变量

这些ROM程序使用在RAM中定义的变量去定义编程的地址和数据以及CPU速度和控制字节。它们是：

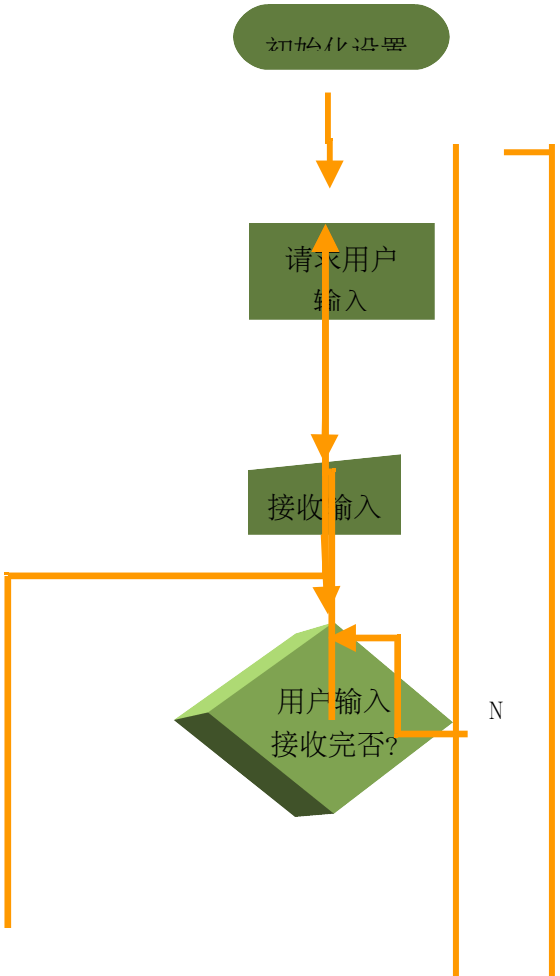
位置	RAM地址	字节	使用
Ctr1Byt	$88	1	控制位
CPUSpd	$89	1	总线速度以0.25MHz为单位

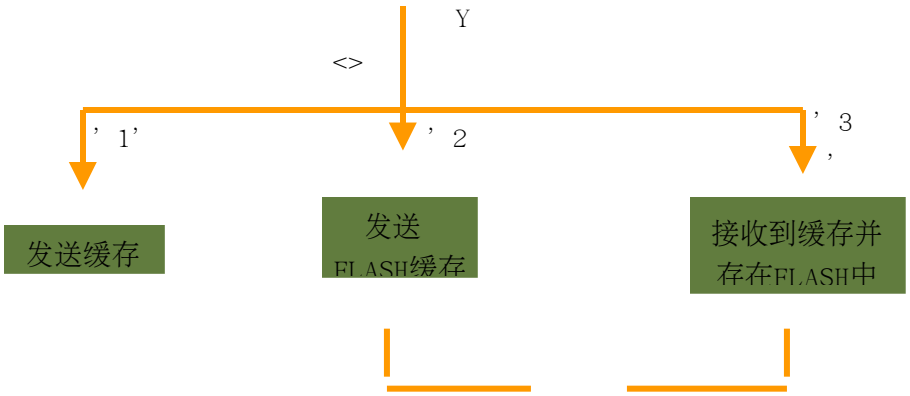
LstAddr	$8A—$8B	2	FLASH块和地址
BfrStrt	$8C=>	块大小	数据缓存

3．2 NITRON内存图

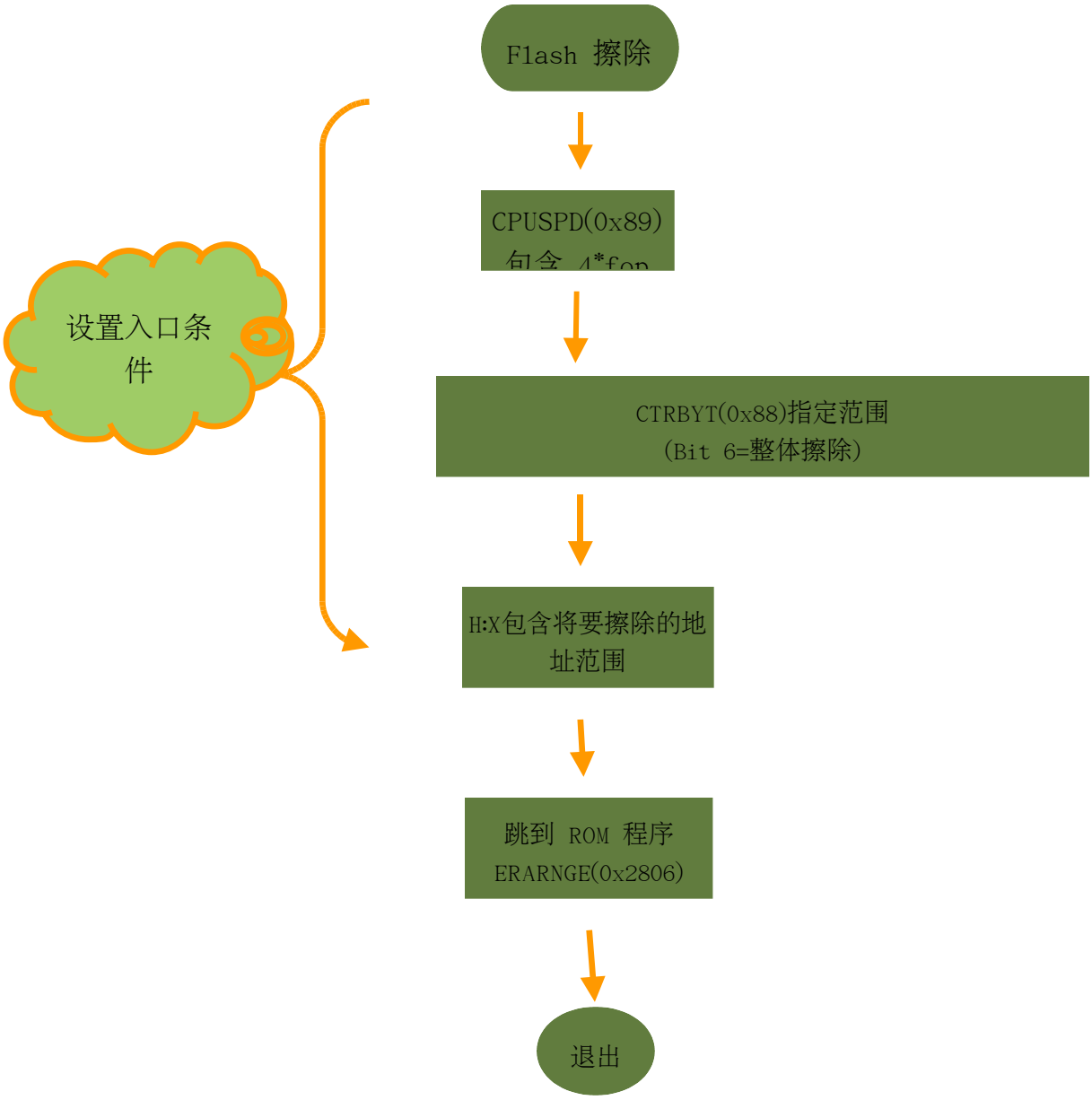


3．3 流程图例2（1）




Y
<>
’ 1’
’ 2
’ 3
’
发送缓存
发送
FLASH缓存
接收到缓存并
存在FLASH中

3．4 流程图示例2（2）

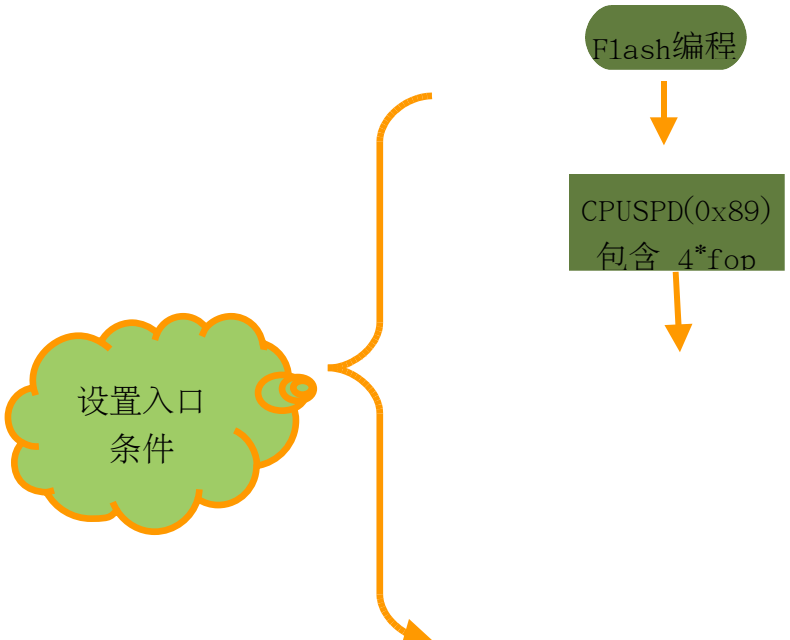

Flash 擦除
CPUSPD(0x89)
设置入口条
件
CTRBYT(0x88)指定范围
(Bit 6=整体擦除)
H:X包含将要擦除的地
址范围
跳到 ROM 程序
ERARNGE(0x2806)
退出

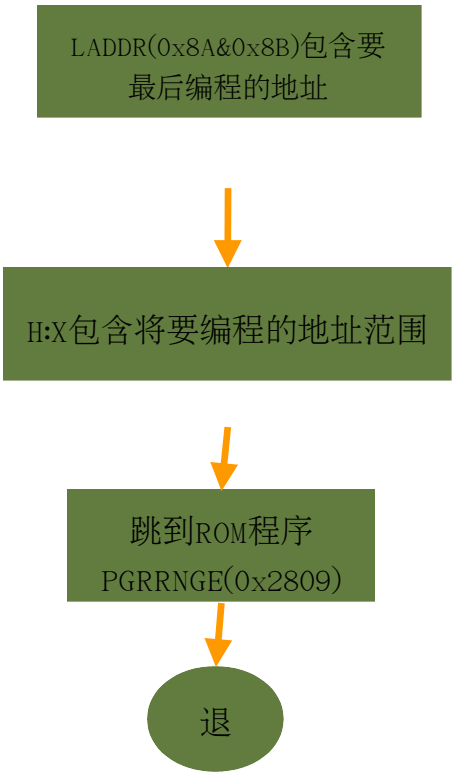
FLASH擦除程序，打开文件Lab15-EEPROM.mcp

```
//Definitions
#define ERARNGE() {__asm jsr 0x2806;}     //跳到0x2806
 #define CTRLBYT  (*(volatile unsigned char*) (0x88))
#define CPUSPD    (*(volatile unsigned char*) (0x89))
#define LADDRH    (*(volatile unsigned char*) (0x8A))
#define LADDRL    (*(volatile unsigned char*) (0x8B))
#define FLASH_TEST_ADDRESS    0xFD40       //定位将要擦除的页
#define OSC_CONST FBUS/250000
 //Function  Call
address = FLASH_TEST_ADDRESS;
EraseRow(&address);
//Function Definition
 void EraseRow(Word *_row){
                Word _address;
                _address = *_row;
                CPUSPD = OSC_CONST;
                CTRLBYT &= 0xBF;                  //清除第6位（非整体擦除）
                __asm ldhx _address;     //H: X必须包含要擦除的地址
                ERARNGE();
                return;
  }
```

RAM 变量

3．5 流程图示例2（3）





FLASH编程程序

```
//Definitions
#define PGRRNGE() {__asm jsr 0x2809;}                    //跳到0x2809
#define CTRLBYT   (*(volatile unsigned char*) (0x88))
#define CPUSPD    (*(volatile unsigned char*) (0x89))
#define LADDRH    (*(volatile unsigned char*) (0x8A))
#define LADDRL    (*(volatile unsigned char*) (0x8B))
unsigned char My_Receive[RECEIVE_LENGTH]@0x8C;  //检查BfrStr位置的缓存
#define FLASH_TEST_ADDRESS      0xFD40          //将要编程的位置
#define OSC_CONST FBUS/250000
 //Function  Call
address = FLASH_TEST_ADDRESS;
ProgramRange (&address, RECEIVE_LENGTH);
//Function Definition
 void ProgramRange(Word *_ini, Byte _num) {
                  Word _first;
                 _first = *_ini;
                 CPUSPD = OSC_CONST;
                  LADDRH = ((_first + _num -1) & 0xFF00) >> 8;    //设置最后编程位
置
```

RAM 变量

```
            LADDRL = ((_first + _num -1) & 0x00FF);
            __asm ldhx _first;      //H: X必须包含将要编程的位置
            PGRRNGE();
            return;
        }
```

主程序

例程使用前面例子中的UART并等待用户输入:

1) 读缓存（初始化为“FREESCALE”）;

2) 读EEPROM位置;

3) 写到缓存和EEPROM。

```
const unsigned char My_ROM[SEND_LENGTH] @FLASH_TEST_ADDRESS;
…case '2':
            SEND_BREAK();
            pSend=&My_ROM[0];                       //从EEPROM发送8位
            UART_SendString(pSend,SEND_LENGTH);
            break;
   case '3':
            address = FLASH_TEST_ADDRESS;
            EraseRow(&address);                     //擦除EEPROM位置
            SEND_BREAK();
            UART_GetString(&My_Receive[0],RECEIVE_LENGTH);
            pSend=&My_Receive[0];
            ProgramRange (&address, RECEIVE_LENGTH);   // 编程EEPROM （数据已在
BufrStrt
                                                    //(My_Receive@0x8C))
            break;          ….
```

改变和改进:

这个例子只是擦除和编程同一Flash位置，这不是已有的最好方法。

Freescale保证Flash存贮器写10,000次（最少)。

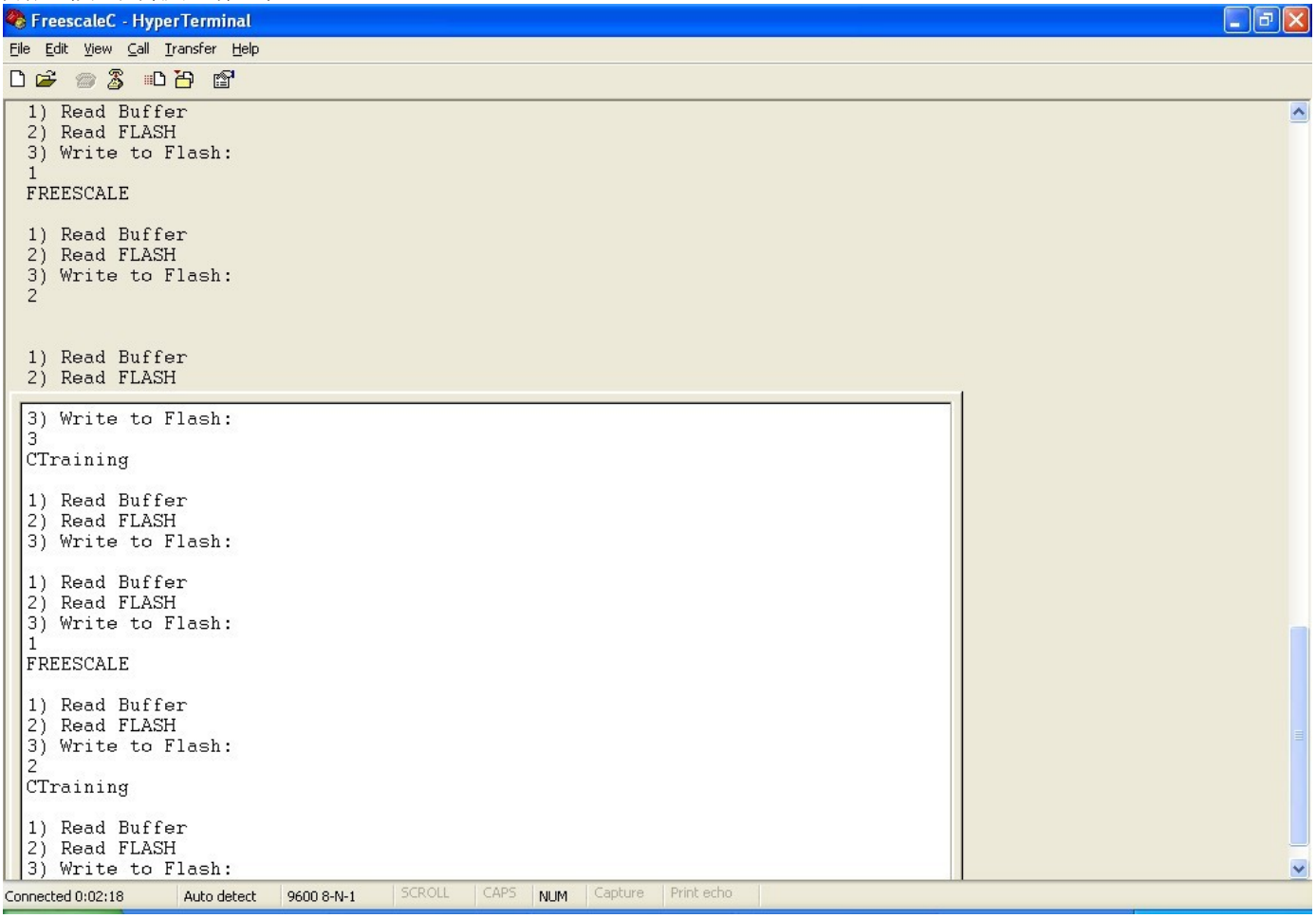
其它方法能用于延长Flash存贮器的寿命。

1) 在写一个新缓冲区前查找$FF（不应将$FF作为一个有效字节， +Code, +Overhead)

2) 在每个数据前使用复制字节(每缓冲区多于1个字节，+前面的同样限制)。

Flash存贮器寿命能延长每块64*100,000次。

用超级终端测试程序：



（全文完）