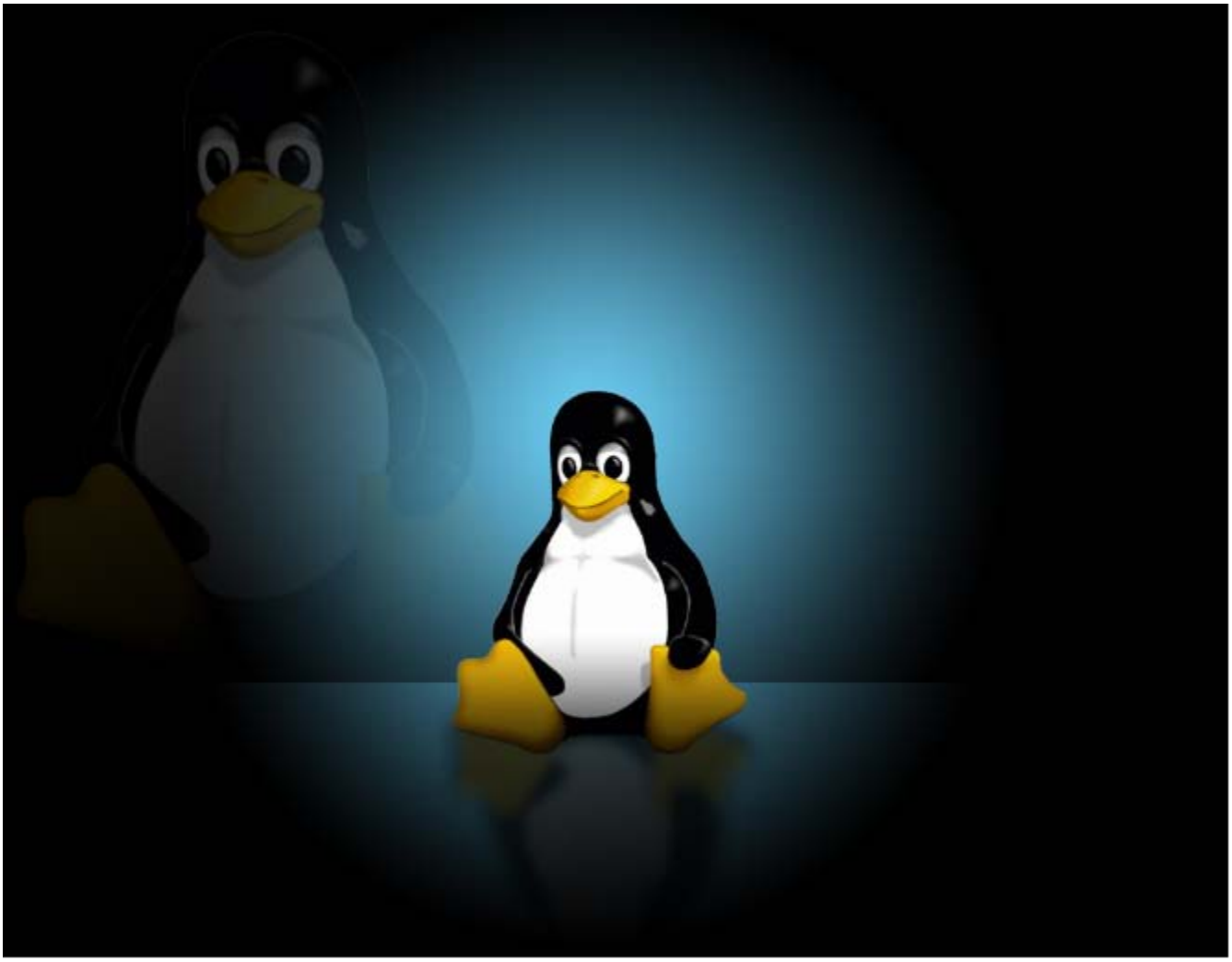
Open Source , Open your life.



<资料整合第二辑>

2009.04.09

大家有好的资料或在网上看到好的文章，请发往 yuanyangliu258@gmail.com ， 我替大家整理
同时也希望有同学愿意加入这样的开源行列，大家一起学习，一起讨论，一起为开源事业奋斗。
有意者邮件联系

申明：本资料均转载网络，有删节，请忽用于商业用途

怎样高效学习嵌入式

1、学习 linux

根据我在论坛的了解，我选择学习嵌入式 linux，刚好我们学校也重视嵌入式 linux，从实验室到课程安排都是关于嵌入式 linux 方面，天时地利！这里我把学习 linux 的经验和教训说说。

可以这样说，在论坛里说道学习 linux 差不多就学习 linux 内核。于是我电脑里安装了 linux 就开始看 linux 内核方面的书了。我记得来学校以前就买到一本陈莉君的讲 linux 内核的第一版，现在有第二版了。我就开始看那本说，大家说 linux 内核情景分析不错，我就买了上下册，后来又买了《深入理解 linux 内核》，最后也买了毛德操的《嵌入式系统》也是分析 linux 内核代码的，主要讲 arm 相关的。

看内核期间是个非常痛苦的过程，看情景分析有种在森林中找出路，其间我组织了一些同学学习内核，几乎没有几个能坚持下来的。我认为我是坚持下来了。情景分析在看第一、第二遍是几乎没有摸到门道，我分析有三个方面的原因：1、自己的基础差，这是最要害的。2、内核本身很难。3、没有交流和高人指点。到了第三遍时我才摸到门，才差不多知道个 linux 的大概脉络，很多细节也是稀里糊涂。

学习 linux 总结，这里声明一下，我指的嵌入式主要是偏向软件的嵌入式。学习嵌入式的重点和难点要害在操作系统，假如没有把握操作系统，我认为很难把握一个嵌入式系统。即使在做嵌入式开发中，作应有层的开发几乎可以不知道操作系统也可以开发，我认为那是浮在表面的。很难深入和提高自己的层次。声明：一孔之见！不可深究！

在学习 linux 内核过程中犯了一个极其严重路线错误：对 linux 几乎不懂就开始学习内核。我个人推荐一个学习路线是：使用 linux—〉linxu 系统编程开发---〉驱动开发和分析 linux 内核。而我差不多相反，实际上你不会使用 linux 也可以学习内核，但是假如你懂了这些东西学习更有效率。

关于要不要学习内核的问题，我的回答如下：不一定。假如你是喜欢钻研的那你进入内核会满足你的欲望。同时对你以后的嵌入式系统的开发有很好的影响。假如你想从事嵌入式 linux 系统开发，最好对内核有所了解。假如仅仅是做应用开发没有必要。我打个比喻：c、c 、java 等语言是武林中的某个武林派别的话，如什么拳法，什么刀法等，那么 linux 内核应该是一个人的内功的反应。

怎么开始学 linux 内核：最好有三件宝物：《深入理解 linux 内核》《情景分析》和源代码。

先看《深》，那主要讲原理，似乎市场上有本讲原理性并且更浅，《linux 内核设计与实现》听说不错。假如没有学习操作系统的，像我这样的，最好先看看操作系统原理的书。看了几遍后，就看情景分析，最好对着《深》看。两本交叉看，《深》是纲，《情》是目。最后深入代

码。

2、学习嵌入式 linux

学习嵌入式，我认为两个重点，cpu 和操作系统，目前市场是比较流行 arm，所以推荐大家学习 arm。操作系统很多，我个人对开始学习的人，非凡不是计算机专业的，推荐学习 ucos。那是开源的，同时很小。学习很好。为什么选 linux，我不想讲太多，网上这方面的太多，但是我在工作中发现，做 linux 的技术路线很难，在 windows 几乎不会有的问题，在 linux 开发中几乎遍地陷阱。一掉进去划很长时间出来，一旦解决自己又长进了！相对来说开发周期长，难度大。现在资料也逐渐丰富起来，难度也降低了些！

至于怎么学习，这是他的特色地方，必须有块开发板，我是同学里最早买学习板的，虽然化钱，我认为值。对我实习和工作产生了很大的影响。

假如没有开发板，那是纸上谈兵。有人说，那要 1000-2000 亚，的确是，兄弟，看长远的，对您的职业和发展那点钱不算什么！有的人说我站着说话不腰痛，好吧，钱这东西对我影响很大，我在大学里扫厕所，扫的不错，奖 5 元/月。兄弟你有过吗？我认为教育投资是效益最有保障的！我实习拿 3k，很多同学拿 1- 2k，当然比我高的也有。虽然我现在没有毕业，但一家公司就签了。从事目前流行的高档的消费电子的研发。对于我两年前一无所知的我，应该有质的变化，我感谢学校收了我这个废品。当然我也非常的努力。说这么多并不是要炫耀我什么，实际上根本不值得炫耀，虽然现在工作环境和待遇比较满足，但是，我发现我很差，非凡是代码能力，我希望平常大家少玩游戏，多编程。编程才是硬道理！

3、 学习板和培训

你可以不接受培训，但是不能不买开发板。假如你想学习嵌入式一定要买块学习板，最好买块 arm9 的，贵些也值。可以跑高级操作系统。这个不要什么品牌公司的，一般不会出什么问题，市场上的板子一般是三星的 2410，基本上是抄的。假如是做产品那要注重些。一般的公司一般要 2000，加 lcd 要 3000 甚至更多。我接触到有的板只要 1200，加 lcd 也就 2000，差别比较大，我认为学习都够啦。 ARM 开发论坛

对于培训，假如有条件最好参加，主要是嵌入式相对别的不同，自己摸索很费时，假如有高手指点非常的好，进步快。非凡是熟悉做这些的朋友。我接受过培训，熟悉个朋友。在以后的工作中帮我很多，在这里谢谢他！我记得哈佛的一个 MBA 的学生这样说，大意是：在哈佛学到

什么不重要，最重要的是这些未来各大公司的高级治理人员是我的同学！真是有远见！

目前市场是培训比较贵，一个星期两三千，真正算起来也要，那些讲师待遇绝对不会低。但是有个致命的问题是连续上课，一连几天，效果不佳。我希望社会上多出些 1000 元左右，并且时间长些，如一个月，一周一到两次课。假如没人做，我来做，哈哈！

没有想到，一下写了这么多，其实还想写的，比如在中科院的一位老兄是怎样学习 linux 内核和看书的，真的很感动。他的为人我很钦佩。也想介绍毛德操的《嵌入式系统》那本书，对学习 arm linux 的很好，也想介绍实习和工作的东西，太长了，耽误大家时间。我也不想检查里面的错别字了，很多！就写到这里吧。

关于嵌入式系统

如何学习嵌入式系统（基于 ARM 平台）

前言

网上看到众多网友都问了关于嵌入式系统方面的很多问题，很多都可在这里找到答案，希望我的这篇文章能给他们以启发。

一、嵌入式系统的概念

着重理解“嵌入”的概念

主要从三个方面上来理解。

1、 从硬件上，将基于 CPU 的处围器件，整合到 CPU 芯片内部，比如早期基于 X86 体系结构下的计算机，CPU 只是有运算器和累加器的功能，一切芯片要造外部 桥路来扩展实现，象串口之类的都是靠外部的 16C550/2 的串口控制器芯片实现，而目前的这种串口控制器芯片早已集成到 CPU 内部，还有 PC 机有显卡， 而多数嵌入式处理器都带有 LCD 控制器，但其种意义上就相当于显卡。比较高端的 ARM 类 Intel Xscale 架构下的 IXP 网络处理器 CPU 内部集成 PCI 控制器（可配成支持 4 个 PCI 从设备或配成自身为 CPI 从设备）；还集成 3 个 NPE 网络处理器引擎，其中两个对应于两个 MAC 地址，可用于网关交换 用，而另外一个 NPE 网络处理器引擎支持 DSL，只要外面再加个 PHY 芯片即可以实现 DSL 上网功能。IXP 系列最高主频可

以达到 1.8G，支持 2G 内 存，1G×10 或 10G×1 的以太网口或 Febre channel 的光通道。IXP 系列应该是目标基于 ARM 体系统结构下由 intel 进行整合后成 Xscale 内核的最高的处理器了。

2、从软件上前，就是在定制操作系统内核里将应用一并选入，编译后将内核下载到 ROM 中。而在定 制操作系统内核时所选择的应用程序组件就是完成了软件的“嵌入”，比如 WinCE 在内核定制时，会有相应选择，其中就是 wordpad,PDF,MediaPlay 等等选择，如果我们选择了，在 CE 启动后，就可以在界面中找到这些东西，如果是以前 PC 上将的 windows 操作系统，多半的东西都需要我们得新再装。

3、把软件内核或应用文件系统等东西烧到嵌入式系统硬件平台中的 ROM 中就实现了一个真正的“嵌入”。

以上的定义是我在 6、7 年前给嵌入式系统下自话侧重于理解型的定义，书上的定义也有很多，但在这个领域范围内，谁都不敢说自己的定义是十分确切的，包括那些专家学者们，历为毕竟嵌入式系统是计算机范畴下的一门综合性学科

二、嵌入式系统的分层与专业的分类。

嵌入式系统分为 4 层，硬件层、驱动层、操作系统层和应用层。

1、硬件层，是整个嵌入式系统的根本，如果现在单片机及接口这块很熟悉，并且能用 C 和汇编语言来编程的话，从嵌入式系统的硬件层走起来相对容易，硬件层 也是驱动层的基础，一个优秀的驱动工程师是要能够看懂硬件的电路图和自行完成 CPLD 的逻辑设计的，同时还要对操作系统内核及其调度性相当的熟悉的。但硬 件平台是基础，增值还要靠软件。硬件层比较适合于，电子、通信、自动化、机电一体、信息工程类专业的人来搞，需要掌握的专业基础知识有，单片机原理及接口技术、微机原理及接口技术、C 语言。

2、 驱动层，这部分比较难，驱动工程师不仅要能看懂电路图还要能对操作系统内核十分的精通，以便其所写的驱动程序在系统调用时，不会独占操作系统时间片，而导 至其它任务不能动行，不懂操作系统内核架构和实时调度性，没有良好的驱动编写风格，按大多数书上所说添加的驱动的方式，很多人都能做到，但可能连个初级的 驱动工程师的水平都达不到，这样所写的驱动在应用调用时就如同 windows 下我们打开一个程序运行后，再打开一个程序时，要不就是中断以前的程序，要不 就是等上一会才能运行后来打开的程序。想做个好的驱动人员没有三、四年功底，操作系统内核不研究上几编，不是太容易成功的，但其工资在嵌入式系统四层中可 是最高的。驱动层比较适合于电子、通信、自动化、机电一体、信息工程类专业尤其

是计算机偏体系结构类专业的人来搞，除硬件层所具备的基础学科外，还要对数据结构与算法、操作系统原理、编译原理都要十分精通了解。

3、 操作系统层，对于操作系统层目前可能只能说是简单的移植，而很少有人来自已写操作系统，或者写出缺胳膊少腿的操作系统来，这部分工作大都由驱动工程师来完 成。操作系统是负责系统任务的调试、磁盘和文件的管理，而嵌入式系统的实时性十分重要。据说，XP 操作系统是微软投入 300 人用两年时间才搞定的，总时工 时是 600 人年，中科院软件所自己的女娲 Hopen 操作系统估计也得花遇几百人年才能搞定。因此这部分工作相对来讲没有太大意义。

4、 应用层，相对来讲较为容易的，如果会在 windows 下如何进行编程接口函数调用，到操作系统下只是编译和开发环境有相应的变化而已。如果涉及 Jave 方 面的编程也是如此的。嵌入式系统中涉及算法的由专业算法的人来处理的，不必归结到嵌入式系统范畴内。但如果涉及嵌入式系统下面嵌入式数据库、基于嵌入式系 统的网络编程和基于某此应用层面的协议应用开发（比如基于 SIP、H.323、Astrisk）方面又较为复杂，并且有难度了。

三、目标与定位。

先有目标，再去定位。

学 ARM，从硬件上讲，一方面就是学习接口电路设计，另一方面就是学习汇编和 C 语言的板级编程。如果从软件上讲，就是要学习基于 ARM 处理器的操作系统层面 的驱动、移植了。这些对于初学都来说必须明确,要么从硬件着手开始学，要么从操作系统的熟悉到应用开始学，但不管学什么，只要不是纯的操作系统级以上基于 API 的应用层的编程，硬件的寄存器类的东西还是要能看懂的，基于板级的汇编和 C 编程还是要会的。因此针对于嵌入式系统的硬件层和驱动程的人，ARM 的接 口电路设计、ARM 的 C 语言和汇编语言编程及调试开发环境还是需要掌握的。

因此对于初学者必然要把握住方向，自己的目标是什么，自己要在那一层面上走。然后再着手学习较好，与 ARM 相关的嵌入式系统的较为实际的两个层面硬件层和驱动层，不管学好了那一层都会很有前途的。

如 果想从嵌入式系统的应用层面的走的话，可能与 ARM 及其它体系相去较远，要着重研究基嵌入式操作系统的环境应用与相应开发工具链，比如 WinCe 操作系统 下的 EVC 应用开发（与 windows 下的 VC 相类似），如果想再有突破就往某些音视频类的协议上靠，比如 VOIP 领域的基于 SIP 或 H.323 协议的应 用层开发，或是基于嵌入式网络数据库的开发等等。

对于初学者来讲，要量力而行，不要认为驱动层工资高就把它当成方向了，要结 合自身

特点，嵌入式系统四个层面上那个层面上来讲都是有高人存在，当然高人也对应的高工资，我是做硬件层的，以前每月工资中个人所得税要被扣上近 3 千大 元，当然我一方面充当工程师的角色，一方面充当主管及人物的角色，两个职位我一个人干，但上班时间就那些。硬件这方面上可能与我 PK 的人很少了，才让我拿 到那么多的工资。

四、开发系统选择。

很多 ARM 初学者都希望有一套自己能用的系统，但他们住住 会产生一种错误认识就是认为处理器版本越高、性能越高越好，就象很多人认为 ARM9 与 ARM7 好，我想对于初学者在此方面以此入门还应该理智，开发系统的 选择最终要看自己往嵌入式系统的那个方向上走，是做驱动开发还是应用，还是做嵌入式系统硬件层设计与板级测试。如果想从操作系统层面或应用层面上走，不管 是驱动还是应用，那当然处理器性能越高越好了，但这个东西自学，有十分大的困难，不是几个月或半年或是一年二年能搞定的事。

在某种意义上请，ARM7 与 9 的差别就是在某些功能指令集上丰富了些，主频提高一些而已，就比如 286 和 386。对于用户来讲可能觉查不到什么，只能是感觉速度有些快而已。

ARM7 比较适合于那些想从硬件层面上走的人，因为 ARM7 系列处理器内部带 MMU 的很少，而且比较好控制，就比如 S3C44B0 来讲，可以很容易将 Cache 关了，而且内部接口寄存器很容易看明白，各种接口对于用硬件程序控制或 AXD 单步命令行指令都可以控制起来，基于 51 单片机的思想很容易能把他 搞懂，就当成个 32 位的单片机，从而消除很多 51 工程师想转为嵌入式系统硬件 ARM 开发工程师的困惑，从而不会被业界某此不是真正懂嵌入式烂公司带到操作 系统层面上去，让他们望而失畏，让业界更加缺少这方面的人才。

而嵌入式系统不管硬件设计还是软件驱动方面都是十分注重接口这 部分的，选择平台还要考察一个处理器的外部资源，你接触外部资源越多，越熟悉他们那你以后就业成功的机率就越高，这就是招聘时所说的有无“相关技能”，因 为一个人不可能在短短几年内把所有的处理器都接触一遍，而招聘单位所用的处理器就可能是我们完全没有见过的，就拿台湾数十家小公司（市价几千万）的公司生 产的 ARM 类处理器，也很好用，但这些东西通用性太差，用这些处理器的公司就只能招有相关工作经验的人了，那什么是相关工作经验，在硬件上讲的是外围接口 设计，在软件上讲是操作系统方面相关接口驱动及应用开发经验。我从业近十年，2000 年 ARM 出现，我一天始做 ARM7,然后直接跑到了 Xscale（这 个板本在 ARM10-11 之间），一做就是五年，招人面试都不下数百人，在这些方面还是深有体会的。

我个人认为三星的 S3C44b0 对初学者来说比较合适，为什么这么说哪？因为接口资源比较丰富，技术成熟，资料较多，应该十分适合于初学者，有问题可能很容易找人帮且解 决，因为大多数人都很熟悉，就如同 51 类的单片机，有 N 多位专家级的人物可以给你帮忙，相关问题得以很快解答，所然业界认为这款 ARM 都做用得烂了，但对 于初学者来，就却是件好事。

因此开发系统的选择，要看自己的未来从来目标方向、要看开发板接口资源、还要看业界的通用性。

五、如何看待培训。

首 先说说我自己，我目前从业近十年，与国内嵌入式系统行业共同起步，一直站在嵌入式系统行业前沿，设计过多款高端嵌入式系统平台产品并为众多公司提供过解决 方案，离职前为从事 VOIP 的美资公司设计 IP-PBX，历任项目经理、项目主管、技术总监、部门经理，积累众多人脉，并集多年经验所得，考虑到学生就业 与公司招人的不相匹配，公司想招人招不到，而学生和刚毕业的工程师想找份工作也不太容易，于此力创知天行科技有限公司，开展嵌入式系统教育培训。

因 一线的科研人员和一线的教师不相接触，导至国内嵌入式人才缺乏，国外高校的技术超前于业界公司，而国内情况是业界公司方面的嵌入式系统技术要远远领先于高 校。为架构业界与高校沟通的桥梁，把先进技能带给高校学子，为学生在就业竞争中打造一张王牌，并为业界工程师快速提升实现自我创造机遇，我就这样辞去了外 企年薪 20 多万的职位，做嵌入式系统方面的培训了。

对于培训来讲，是花钱来买时间，很多工程师都喜欢自己学，认为培训不值，这也是有 可能的，纯为赚钱的培训当然不会太有价值，但对于实力型的培训他们可能就亏大了，有这样一笔帐不知他们算过没有，如果一个一周的培训，能带给他们自学两年 后才能掌握的知识，在培训完后他们用三个月到半年时间消化培训内容，这样他会省约至少一年半的时间来学其它的或重新站在另一个高度上工作，那么他将最迟一 年后会拿到他两年后水平所对应的工资，就是在工资与水平对应的关系上比同批人缩短一年，每月按最少 1 千计，再减去培训费用至少多 1.0 万，同时也省了一年 时间，不管是休闲也好，再继续提高也好，总之是跑到了队伍的前面了。

另一层面上讲，对于新人的培训相当于他们为自己提前买了份失业保险，有师傅会带领他们入道，我今年暑假时班里最年轻的一个学生是大二的，今年才上大三，这学期才刚学单片机，但现在 ARM 方面的编程工作已经搞得有声有色了，再过一年多毕业，他还会失业吗？

再 者通过培训，你可以知道很多业界不为常人所知的事，同时也为自己找了个师傅，就比

如说，两个工程师分别用 S3C2410 和 PXA255 来做手持设备，同样 两人都工作四年，再出去找工作，两人工资可能最多可相差一倍，为什么？这就是业界不为常人所知的规则，2410 属于民品，被业界用烂了，做产品时成本特敏 感，当然也对人才成本敏感了，PXA255 是 intel 的东西，一个 255 CPU 能买三个 2410，一直被业界定义为贵族产品，用的公司都是大公司或为 军方服务的公司，不会在乎成本，只要把东西做好，一切都好说，但这方面做的人也少啊，因为开发系统贵啊。

对于说为自己找了个好师傅， 我想是这样的，因为同级工程师间存在着某此潜在的竞争关系，有很多人不愿意把自己知道的东西教给别人，这意味着他将要失业，就是所说的教会徒弟，饿死师 傅，但对于我们这些人就不存在这样的关系了，我是在嵌入式系统平台设计上走到了一定程序，目前在国内这块的技术上已经是自己很难再突破自己，因此很多东西 我对大家都是 OPEN 的，就比如说下面那部分关于接口设计中所提到的时序接口东西，我要是不讲，却使是高级硬件工程师我想也几乎只有 10%的人能知道吧。

六、成为高级嵌入式系统硬件工程师要具备的技能。

首先我声明，我是基于嵌入式系统 平台级设计的，硬件这个方向我相对来讲比较有发言权，如果是其它方面所要具备的基本技能还要和我们培训中心其它专业级讲师沟通，或去网站看看 www.zt-training.com。他们的方面上我只能说是知道些，但不是太多，初级的问题也可以问我。

对于硬件来讲有几个方 向，就单纯信号来分为数字和模拟，模拟比较难搞，一般需要很长的经验积累，单单一个阻值或容值的精度不够就可能使信号偏差很大。因此年轻人搞的较少，随着 技术的发展，出现了模拟电路数字化，比如手机的 Modem 射频模块，都采用成熟的套片，而当年国际上只有两家公司有此技术，自我感觉模拟功能不太强的人， 不太适合搞这个，如果真能搞定到手机的射频模块，只要达到一般程度可能月薪都在 15K 以上。

另一类就是数字部分了，在大方向上又可分 为 51/ARM 的单片机类，DSP 类，FPGA 类，国内 FPGA 的工程师大多是在 IC 设计公司从事 IP 核的前端验证，这部分不搞到门级，前途不太明朗，即 使做个 IC 前端验证工程师，也要搞上几年才能胜任。DSP 硬件接口比较定型，如果不向驱动或是算法上靠拢，前途也不会太大。而 ARM 单片机类的内容就较 多，业界产品占用量大，应用人群广，因此就业空间极大，而硬件设计最体现水平和水准的就是接口设计这块，这是各个高级硬件工程师相互 PK，判定水平高低的 依据。而接口设计这块最关键的是看时序，而不是简单的连接，比如 PXA255 处理器 I2C 要求速度在 100Kbps，如果把一个 I2C 外

围器件，最高还达 不到 100kbps 的与它相接，必然要导致设计的失败。这样的情况有很多，比如 51 单片机可以在总线接 LCD，但为什么这种 LCD 就不能挂在 ARM 的总线 上，还有 ARM7 总线上可以外接个 Winband 的 SD 卡控制器，但为什么这种控制器接不到 ARM9 或是 Xscale 处理器上，这些都是问题。因此接口并 不是一种简单的连接，要看时序，要看参数。 一个优秀的硬件工程师应该能够在没有参考方案的前提下设计出一个在成本和性能上更加优秀的产品，靠现有的方 案，也要进行适当的可行性裁剪，但不是胡乱的来，我遇到一个工程师把方案中的 5V 变 1.8V 的 DC 芯片，直接更换成 LDO，有时就会把 CPU 烧上几个。前 几天还有人希望我帮忙把他们以前基于 PXA255 平台的手持 GPS 设备做下程序优化，我问了一下情况，地图是存在 SD 卡中的，而 SD 卡与 PXA255 的 MMC 控制器间采用的 SPI 接口，因此导致地图读取速度十分的慢，这种情况是设计中严重的缺陷，而不是程序的问题，因此我提了几条建议，让他们更新试下再 说。因此想成为一个优秀的工程师，需要对系统整体性的把握和对已有电路的理解，换句话说，给你一套电路图你终究能看明白多少，看不明白 80%以上的话，说 明你离优秀的工程师还差得远哪。其次是电路的调试能力和审图能力，但最最基本的能力还是原理图设计 PCB 绘制，逻辑设计这块。这是指的硬件设计工程师，从 上面的硬件设计工程师中还可以分出 ECAD 工程师，就是专业的画 PCB 板的工程师，和 EMC 设计工程师，帮人家解决 EMC 的问题。硬件工程师再往上就是板 级测试工程师，就是 C 语功底很好的硬件工程师，在电路板调试过程中能通过自己编写的测试程序对硬件功能进行验证。然后再交给基于操作系统级的驱动开发人 员。

总之，硬件的内容很多很杂，硬件那方面练成了都会成为一个高手，我时常会给人家做下方案评估，很多高级硬件工程师设计 的东西，经常被我一句话否定，因此工程师做到我这种地步，也会得罪些人，但硬件的确会有很多不为人知的东西，让很多高级硬件工程师也摸不到头脑。

那 么高级硬件件工程师技术技能都要具备那些东西哪，首先要掌握 EDA 设计的辅助工具类如 Protel\ORCAD\PowperPCB\Maplux2 \ISE、VDHL 语言，要能用到这些工具画图画板做逻辑设计，再有就是接口设计审图能力，再者就是调试能力，如果能走到总体方案设计这块，那就基本上快 成为资深工程师了。

硬件是要靠经验，也要靠积累的，十年磨一剑，百年磨一针。

把一个月前想写的东西，今天终于用一上午的进间整理完了，希望对喜爱嵌入式系统开发的工程师和学生们有所帮助。

嵌入式人才的发展方向(zt)

搞嵌入式开发的人有两类。

一类是学电子工程、通信工程等偏硬件专业出身的人，他们主要是搞硬件设计，有时要开发一些与硬件关系最密切的最底层软件，如 BootLoader、Board Support Package(像 PC 的 BIOS 一样，往下驱动硬件，往上支持操作系统），最初级的硬件驱动程序等。他们的优势是对硬件原理非常清楚，不足是他们更擅长定义各种硬件接口，但对复杂软件系统往往力不从心（例如嵌入式操作系统原理和复杂应用软件等）。

另 一类是学软件、计算机专业出身的人，主要从事嵌入式操作系统和应用软件的开发。如果我们学软件的人对硬件原理和接口有较好的掌握，我们完全也可写 BSP 和 硬件驱动程序。嵌入式硬件设计完后，各种功能就全靠软件来实现了，嵌入式设备的增值很大程度上取决于嵌入式软件，这占了嵌入式系统的最主要工作（目前有很 多公司将硬件设计包给了专门的硬件公司，稍复杂的硬件都交给台湾或国外公司设计，国内的硬件设计力量很弱，很多嵌入式公司自己只负责开发软件，因为公司都 知道，嵌入式产品的差异很大程度在软件上，在软件方面是最有“花头“可做的），所以我们搞软件的人完全不用担心我们在嵌入式市场上的用武之地，越是智能设 备越是复杂系统，软件越起关键作用，而且这是目前的趋势。

从事嵌入式软件开发的好处是：

（1）目前国 内外这方面的人都很稀缺。一方面，是因为这一领域入门门槛较高，不仅要懂较底层软件（例如操作系统级、驱动程序级软件），对软件专业水平要求较高（嵌入式 系统对软件设计的时间和空间效率要求较高），而且必须懂得硬件的工作原理，所以非专业 IT 人员很难切入这一领域；另一方面，是因为这一领域较新，目前发展 太快，很多软硬件技术出现时间不长或正在出现（如 ARM 处理器、嵌入式操作系统、MPEG 技术、无线通信协议等），掌握这些新技术的人当然很找。嵌入式人 才稀缺，身价自然就高，越有经验价格就越高。其实嵌入式人才稀少,根本原因可能是大多数人无条件接触,这需要相应的嵌入式开发板和软件,另外需要有经验的 人进行指导开发流程。

（2）与企业计算等应用软件不同，嵌入式领域人才的工作强度通常低一些（但收入不低）。搞企业应用软件的 IT 企业， 这个用户的系统搞完了，又得去搞下一个用户的，而且每个用户的需求和完成时间都得按客户要求改变，往往疲于奔命，重复劳动。相比而言，搞嵌入式系统的

公 司，都有自己的产品计划，按自己的节奏行事。所开发的产品通常是通用的，不会因客户的不同而修改。一个产品型号开发完了，往往有较长一段空闲时间（或只是 对软件进行一些小修补），有时间进行充电和休整。另外，从事嵌入式软件的每个人工作范围相对狭窄，所涉及的专业技术范围就是那些（ARM、RTOS、 MPEG、802.11 等），时间长了这些东西会越搞越有经验，卖卖老本，几句指导也够让那些初入道者琢磨半年的。若搞应用软件，可能下一个客户要换成一 个完全不同的软件开发平台，那就苦了。

（3）哪天若想创业，搞自己的产品，那么嵌入式是一个不错的主意，这可不像应用软件那样容易被盗 版。土木学院有一个叫启明星的公司开发出一个好象叫“工程 e”的掌上 PDA（南校区门口有广告），施工技术人员用该 PDA 可当场进行土木概预算和其它土木 计算，据说销路特好。我认识的某大学老师，他开发的饭馆用的点菜 PDA（WinCE 平台，可无线连网和上网），据他说销路不错，饭馆点点 PDA 让客户点 菜，多显派头档次。我记得 00 级 2+2 班当年有一组同学在学 Windows 程序设计课程时用 VC++设计了一个功能很强的点菜系统做为课程项目，当时真想 建议他们将这个软件做成 PDA，估计会有些销路（上海火车站南广场的 Macdonald 便使用很漂亮的 PDA 给用户点食品，像摸像样的）。这些 PDA 的硬 件设计一般都是请其它公司给订做（这叫“贴牌”：OEM），都是通用的硬件，我们只管设计软件就变成自己的产品了。

从事嵌入式软件开发的缺点是：

（1）入门起点较高，所用到的技术往往都有一定难度，若软硬件基础不好，特别是操作系统级软件功底不深，则可能不适于此行。

（2）这 方面的企业数量要远少于企业计算类企业。特别是从事嵌入式的小企业数量较多（小企业要搞自己的产品创业），知名大公司较少（搞嵌入式的大公司主要有 Intel、Motorola、TI、Philip、Samsung、Sony、Futjtum、Bell-Alcatel、意法半导体、 Microtek、研华、华为、中兴通信、上广电等制造类企业）。这些企业的习惯思维方式是到电子、通信等偏硬专业找人。由于我院以前毕业生以企业计算为 主，所以我院与这些企业联系相对较少。我院正积极努力，目前已与其中部分公司建立了联系，争取今后能有我院同学到这些企业中实习或就业。

（3）有少数公司经常要硕士以上的人搞嵌入式，主要是基于嵌入式的难度。但大多数公司也并无此要求，只要有经验即可。

我院同学若学习嵌入式，显然应偏重于嵌入式软件，特别是嵌入式操作系统方面，应是我们的强项。对于搞嵌入式软件的人，最重要的技术显然是（实际上很多公司的招聘广告上就是这样

写的）：

（1） 掌握主流嵌入式微处理器的结构与原理

（2） 必须掌握一个嵌入式操作系统

（3） 必须熟悉嵌入式软件开发流程并至少做过一个嵌入式软件项目。

嵌入式软件方面最重要的课程包括：

（1）嵌入式微处理器结构与应用：这是一门嵌入式硬件基础课程，我院用这门课取代了传统的“微机原理与接口”课程（目前国内已有少部分高校IT专业这样做了，因为讲x86 微机原理与接口很难找到实际用处，只为教学而已）。我们说过，嵌入式是软硬件结合的技术，搞嵌入式软件的人应对ARM处理器工作原理和接口技术有充分了解，包括ARM的汇编指令系统。若不了解处理器原理，怎么能控制硬件工作，怎么能写出节省内存又运行高速的最优代码（嵌入式软件设计特别讲究时空效率），怎么能写出驱动程序（驱动程序都是与硬件打交道的）？很多公司招聘嵌入式软件人员时都要求熟悉ARM处理器，将来若同学到公司中从事嵌入式软件开发，公司都会给你一本该设备的硬件规格说明书(xxx Specification)， 您必须能看懂其中的内存分布和端口使用等最基本的说明（就像x86 汇编一样），否则怎么设计软件。有些同学觉得嵌入式处理器课程较枯燥，这主要是硬件课程 都较抽象的原因，等我们的嵌入式实验室 10 月份建好后，您做了一些实验后就会觉得看得见摸得着。还有同学对ARM汇编不感兴趣，以为嵌入式开发用C语言就 足够了。其实不应仅是将汇编语言当成一个程序设计语言，学汇编主要是为了掌握处理器工作原理的。一个不熟悉汇编语言的人，怎么能在该处理器写出最优的C语 言代码。在嵌入式开发的一些关键部分，有时还必须写汇编，如Bootloader等（可能还包括BSP）。特别是在对速度有极高要求的场合（如DSP处理器的高速图像采集和图像解压缩），目前主要还要靠汇编写程序（我看到过很多公司是这样做的）。当您在一个嵌入式公司工作时，在查看描述原理的手册时，可能很多都是用汇编描述的（我就遇到过），这是因为很多硬件设计人员只会写或者喜欢用汇编描述，此时您就必须看懂汇编程序，否则软硬件人员可能就无法交流。很多嵌入式职位招聘时都要求熟悉汇编。

(2) 嵌入式操作系统类课程

除了WinCE的实时性稍差外，大多数嵌入式操作系统的实时性都很强,所以也可称为实时操作系

统Real TimeOperating System.从事嵌入式的人至少须掌握一个嵌入式操作系统(当然掌握两个更好),这在嵌入式的所有技术中是最为关键的了。目前最重要的RTOS主要包括：

第 一类、传统的经典RTOS：最主要的便是Vxworks操作系统，以及其Tornado开发平台。Vxworks因出现稍早，实时性很强（据说可在 1ms 内响应外部事件请求），并且内核可极微（据说最小可 8K），可靠性较高等，所以在北美，Vxworks占据了嵌入式系统的多半疆山。特别是在通信设备等实 时性要求较高的系统中，几乎非Vxworks莫属。Vxworks的很多概念和技术都和Linux很类似，主要是C语言开发。像Bell- alcatel、Lucent、华为等通信企业在开发产品时，Vxworks用得很多。但Vxworks因价格很高，所以一些小公司或小产品中往往用不 起。目前很多公司都在往嵌入式Linux转（听说华为目前正在这样转）。但无论如何，Vxworks在一段长时间内仍是不可动摇的。与Vxworks类似 的稍有名的实时操作系统还有pSOS、QNX、Nucleus等RTOS。

第二类、嵌入式Linux操作系统：Linux的前途除作为服务器操作系统外，最成功的便是在嵌入式领域的应用，原因当然是免费、开源、支持软件多、呼拥者众，这样嵌入式产品成本会低。Linux本身不是一个为嵌入式设计的操作系统，不是微内核的，并且实时性不强。目 前应用在嵌入式领域的Linux系统主要有两类：一类是专为嵌入式设计的已被裁减过的Linux系统，最常用的是uClinux（不带MMU功能），目前 占较大应用份额，可在ARM7 上跑；另一类是跑在ARM 9 上的，一般是将Linux 2.4.18 内核移植在其上，可使用更多的Linux功能（当然uClinux更可跑在ARM 9 上）。很多人预测，嵌入式Linux预计将占嵌入式操作系统的 50%以上份额，非常重要。缺点是熟悉Linux的人太 少，开发难度稍大。另外，目前我们能发现很多教材和很多大学都以ucOS/II为教学用实时操作系统，这主要是由于ucOS/II较简单，且开源，非常适 合入门者学习实时操作系统原理，但由于ucOS/II功能有限，实用用得较少，所以我院不将其作为教学重点，要学习就应学直接实用的，比如 uClinux就很实用。况且熟悉了Linux开发，不仅在嵌入式领域有用，对开发Linux应用软件， 对加深操作系统的认识也有帮助，可谓一举多得。据我所知，目前Intel、Philip都在大搞ARM+LINUX的嵌入式开发，Fujitum则是在自 己的处理器上大搞Linux开发。目前在嵌入式Linux领域，以下几个方面的人特别难找，一是能将Linux移植到某个新型号的开发版上；二是 能写Linux驱动程序的人；三是熟悉Linux内核裁减和优化的人。我院在该嵌入式Linux方面的课程系列是：本科生操作系统必修课，然后是 Linux程序设计选修课，最后是嵌入式Linux系统选修课。我院在Linux方面目前已有较强力量，魏老

师和张老师熟悉Linux开发
，金老师和唐老师熟悉Linux系统管理。

第 三类、 Windows CE嵌入式操作系统：Microsoft也看准了嵌入式的巨大市场，MS永远是最厉害的，WinCE出来只有几年时间，但目前已 占据了很大市场份额，特别是在PDA、手机、显示仪表等界面要求较高或者要求快速开发的场合，WinCE目前已很流行（据说有一家卖工控机的公司板子卖得 太好，以至来不及为客户裁减WinCE）。WinCE目前主要为 4.2 版（.NET)，开发平台主要为WinCE Platform Builder，有 时也用EVC环境开发一些较上层的应用，由于WinCE开发都是大家熟悉的VC++环境，所以我院学过Windows程序设计课程的同学都不会有多大难 度，这也是WinCE容易被人们接受的原因，开发环境方便快速，微软的强大技术支持，WinCE开发难度远低于嵌入式Linux。对于急于完成，不想拿嵌 入式Linux冒险的开发场合，WinCE是最合适了（找嵌入式Linux的人可没那么好找的），毕竟公司不能像学生学习那样试试看，保证开发成 功更重要。根据不同的侧重点 ，WinCE还有两个特殊版本，一个是MS PocketPC操作系统专用于PDA上（掌上电脑），另一个是 MS SmartPhone操作系统用于智能手机上（带PDA功能的手机），两者也都属于WinCE平台。在PDA和手机市场上，除WinCE外，著名的 PD
A嵌入式操作系统还有Palm OS（因出现很早，很有名）、Symbian等，但在WinCE的强劲冲击下，Palm和Symbian来日还能有多长？

(3) 嵌入式开发的其它相关软件课程

搞嵌入式若能熟悉嵌入式应用的一些主要领域，这样的人更受企业欢迎。主要的相关领域包括：

A、数字图像压缩技术：这是嵌入式最重要最热门的应用领域之一，主要是应掌握 MPEG 编解码算法和技术，如 DVD、MP3、PDA、高精电视、机顶盒等都涉及 MPEG 高速解码问题。为此，
我院已预订了一位能开设数字图像处理课程的博士。

B、通信协议及编程技术：这包括传统的 TCP/IP 协议和热门的无线通信协议。首先，大多数嵌入式设备都要连入局域网或 Internet，所以首先应掌握 TCP/IP 协议及其编程，这是需首

要掌握的基本技术；其次，无线通信是目前的大趋势，所以掌握无线通信协议及编程也是是很重要的。无结通信协议包括无线局域网通信协议 802.11 系列，Bluetooth，以及移动通信（如 GPRS、GSM、CDMA 等）。

C、网络与信息安全技术：如加密技术，数字证书 CA 等。

D、DSP 技术：DSP 是 Digital Signal
Process 数字信号处理的意思，DSP 处理器通过硬件实现数字信号处理算法，如高速数据采集、压缩、解压缩、通信等。数字信号处理是电子、通信等硬件专业的课程，对于搞软件的人若能了解一下最好。目前 DSP 人才较缺。如果有信号与系统、数字信号处理等课程基础，对于学习 MPEG 编解码原理会有很大帮助。

（4）嵌入式开发的相关硬件基础

对于软件工程专业的学生，从事嵌入式软件开发，像数字电路、计算机组成原理、嵌入式微处理器结构等硬件课程是较重要的。另外，汇编语言、C/C++、数据结构和算法、特别是操作系统等软件基础课也是十分重要的。我们的主要目地是能看懂硬件工作原理，但重点应是在嵌入式软件，特别操作系统级软件，那将是我们的优势。我们的研究生里有些是学电子、通信类专业过来的，有较好的模拟电路和单片机基础，学嵌入式非常合适。嵌入式本身就是从单片机发展过来的，只是单片机不带 OS，而现在很多嵌入式应用越来越复杂，以至不得不引入嵌入式操作系统。另外，为追求更高速的信号处理速度，现在在一些速度要求较高的场合，有不少公司是将一些 DSP 算法，如 MPEG 压缩解压缩算法等用硬件来实现，这就涉及到 HDL 数字电路设计技术及其 FPGA/IP 核实现技术，这方面的人目前市场上也很缺。

题外话
另外，能写驱动程序的人目前是非常紧缺的（驱动程序也可归于嵌入式范畴），包括桌面 Windows 中的 DDK 开发环境和 WDM 驱动程序。公司每时每刻都要推出新产品，每一个新产品出来了，要能被操作系统所使用，是必须写驱动程序的。写驱动程序就必须掌握操作系统(如 Windows 或 Linux)的内部工作原理，还涉及到少量硬件知识，难度较大，所以这方面的人很难找。想成为高手的同学，也可从驱动程序方面获得突破。我可说一下自己的经历，三年前我曾短暂地在一家公司写过 WinCE 驱动程序（正是因为知道这方面的人紧缺，所以才要做这方面的

事），尽管那以前从未做过驱动程序，应聘那个职位时正是看准了公司是很难招聘到这方面的人，既然都找不到人，驱动还得有人做，这正是可能有机会切入这一领域的大好机会。面试时大讲自己写过多少万行汇编程序，对计算机工作原理如何清楚，简历中又写着我曾阅读完两本关于 indows Driver Model 的两本英文原版书，写过几个小型的驱动程序练习程序（其实根本没写过，我们的同学将来千万不要像我这样，早练就些过硬功夫，就不至于沦落到我这等地步，就不用像我那样去“欺骗”公司了，我这是一个典型的反面教材），居然一切都 PASS（当然最重要的是笔试和面试问题还说得过去），这只能说明这一领域找人的困难程度。
搞完一个版本就会空一段时间，只有等公司新的芯片推出或新的 OS 出现后，才需要再去开发新一版驱动，那时有将近一个月时间空闲着在等 WinCE .NET Beta 版推出，准备将驱动程序升级到 CE .NET 上，现在在软件学院工作整日忙，无限怀念那段悠闲时光。

要么走 ARM+WinCE，要么走 ARM+LINUX，要么走 ARM+VXWORKS。每个搞嵌入式的人都可选一条
路，条条大路通罗马

学习linux-unix编程方法的建议

建议学习路径：

首先先学学编辑器，vim, emacs 什么的都行。
然后学 make file 文件，只要知道一点就行，这样就可以准备编程序了。
然后看看《C 程序设计语言》K&R，这样呢，基本上就可以进行一般的编程了，顺便找本数据结构的书来看。

如果想学习 UNIX/LINUX 的编程，《APUE》绝对经典的教材，加深一下功底，学习《UNP》的第二卷。这样基本上系统方面的就可以掌握了。
然后再看 Douglus E. Comer 的《用 TCP/IP 进行网际互连》第一卷，学习一下网络的知识，再看《UNP》的第一卷，不仅学习网络编程，而且对系统编程的一些常用的技巧就很熟悉了，如果继续网络编程，建议看《TCP/IP 进行网际互连》的第三卷，里面有很多关于应用协议 telnet、ftp 等协议的编程。
如果想写设备驱动程序，首先您的系统编程的接口比如文件、IPC 等必须要熟知了，再学习《LDD》2。

对于几本经典教材的评价：

《The C Programing Language》K&R 经典的 C 语言程序设计教材，作者是 C 语言的发明者，教材内容深入浅出。虽然有点老，但是必备的一本手册，现在有时候我还常翻翻。篇幅比较小，但是每看一遍，就有一遍的收获。另外也可用谭浩强的《C 语言程序设计》代替。
《Advanced Programing in Unix Envirement》 W.Richard Stevens：也是非常经典的书（废话，Stevens 的书哪有不经典的！），虽然初学者就可以看，但是事实上它是《Unix Network Programing》的一本辅助资料。国内的翻译的《UNIX 环境高级编程》的水平不怎么样，现在有影印版，直接读英文比读中文来得容易。
《Unix Network Programing》W.Richard Stevens：第一卷讲 BSD Socket 网络编程接口和另外一种网络编程接口的，不过现在一般都用 BSD Socket，所以这本书只要看大约一半多就可以了。第二卷没有设计到网络的东西，主要讲进程间通讯和 Posix 线程。所以看了《APUE》以后，就可以看它了，基本上系统的东西就由《APUE》和《UNP》vol2 概括了。看过《UNP》以后，您就会知道系统编程的绝大部分编程技巧，即使卷一是讲网络编程的。国内是清华翻译得《Unix 网络编程》，翻译者得功底也比较高，翻译地比较好。所以建议还是看中文版。
《TCP/IP 祥解》一共三卷，卷一讲协议，卷二讲实现，卷三讲编程应用。我没有怎么看过。，但是据说也很经典的，因为我没有时间看卷二，所以不便评价。
《用 TCP/IP 进行网际互连》Douglus.E.Comer 一共三卷，卷一讲原理，卷二讲实现，卷三讲高级协议。感觉上这一套要比 Stevens 的那一套要好，就连 Stevens 也不得不承认它的第一卷非常经典。事实上，第一卷即使你没有一点网络的知识，看完以后也会对网络的来龙去脉了如指

掌。第一卷中还有很多习题也设计得经典和实用，因为作者本身就是一位教师，并且卷一是国外研究生的教材。习题并没有答案，留给读者思考，因为问题得答案可以让你成为一个中级的 Hacker，这些问题的答案可以象 Douglus 索取，不过只有他只给教师卷二我没有怎么看，卷三可以作为参考手册，其中地例子也很经典。如果您看过 Qterm 的源代码，就会知道 Qterm 的 telnet 实现部分大多数就是从这本书的源代码过来的。对于网络原理的书，我推荐它，而不是 Stevens 的《TCP/IP 祥解》。

《Operating System - Design and Implement》这个是讲操作系统的书，用 Minix 做的例子。作者母语不是英文，所以英文看起来比较晦涩。国内翻译的是《操作系统设计与实现》，我没看过中文版，因为翻译者是尤晋元，他翻译的《APUE》已经让我失望头顶了。读了这本书，对操作系统的底层怎么工作的就会
有一个清晰的认识。

《Linux Device Driver》2e ，为数不多的关于 Linux 设备驱动程序的好书。不过内容有些杂乱，如果您没有一些写驱动的经验，初次看会有些摸不着南北。国内翻译的是《Linux 设备驱动程序》第二版，第一版，第二版的译者我都有很深的接触，不过总体上来说，虽然第二版翻译的有些不尽人意，但是相比第一版来说已经超出了一大截。要读这一本书，至少应该先找一些《计算机原理》《计算机体系结构》的书来马马虎虎读读，至少应该对硬件和计算机的工作过程有一些了解。

Linux 下线程和进程的经典文章

nux 下进程与线程看过很多文章，我觉的这篇可以说最经典

- -

一.基础知识：线程和进程

按照教科书上的定义，进程是资源管理的最小单位，线程是程序执行的最小单位。在操作系统

设计上，从进程演化出线程，最主要的目的就是更好的支持 SMP 以及减小（进程/线程）上下文切换开销。

无论按照怎样的分法，一个进程至少需要一个线程作为它的指令执行体，进程管理着资源（比如 cpu、内存、文件等等），而将线程分配到某个 cpu 上执行。一个进程当然可以拥有多个线程，此时，如果进程运行在 SMP 机器上，它就可以同时使用多个 cpu 来执行各个线程，达到最大程度的并行，以提高效率；同时，即使是在单 cpu 的机器上，采用多线程模型来设计程序，正如当年采用多进程模型代替单进程模型一样，使设计更简洁、功能更完备，程序的执行效率也更高，例如采用多个线程响应多个输入，而此时多线程模型所实现的功能实际上也可以用多进程模型来实现，而与后者相比，线程的上下文切换开销就比进程要小多了，从语义上来说，同时响应多个输入这样的功能，实际上就是共享了除 cpu 以外的所有资源的。

针对线程模型的两大意义，分别开发出了核心级线程和用户级线程两种线程模型，分类的标准主要是线程的调度者在核内还是在核外。前者更利于并发使用多处理器的资源，而后者则更多考虑的是上下文切换开销。在目前的商用系统中，通常都将两者结合起来使用，既提供核心线程以满足 smp 系统的需要，也支持用线程库的方式在用户态实现另一套线程机制，此时一个核心线程同时成为多个用户态线程的调度者。正如很多技术一样，"混合"通常都能带来更高的效率，但同时也带来更大的实现难度，出于"简单"的设计思路，Linux 从一开始就没有实现混合模型的计划，但它在实现上采用了另一种思路的"混合"。

在线程机制的具体实现上，可以在操作系统内核上实现线程，也可以在核外实现，后者显然要求核内至少实现了进程，而前者则一般要求在核内同时也支持进程。核心级线程模型显然要求前者的支持，而用户级线程模型则不一定基于后者实现。这种差异，正如前所述，是两种分类方式的标准不同带来的。

当核内既支持进程也支持线程时，就可以实现线程-进程的"多对多"模型，即一个进程的某个线程由核内调度，而同时它也可以作为用户级线程池的调度者，选择合适的用户级线程在其空间中运行。这就是前面提到的"混合"线程模型，既可满足多处理机系统的需要，也可以最大限度的减小调度开销。绝大多数商业操作系统（如 Digital Unix、Solaris、Irix）都采用的这种能够完全实现 POSIX1003.1c 标准的线程模型。在核外实现的线程又可以分为"一对一"、"多对一"两种模型，前者用一个核心进程（也许是轻量进程）对应一个线程，将线程调度等同于进程调度，交给核心完成，而后者则完全在核外实现多线程，调度也在用户态完成。后者就是前面提到的单纯的用户级线程模型的实现方式，显然，这种核外的线程调度器实际上只需要完成线程运行栈的切换，调度开销非常小，但同时因为核心信号（无论是同步的还是异步的）都是以进

程为单位的，因而无法定位到线程，所以这种实现方式不能用于多处理器系统，而这个需求正变得越来越大，因此，在现实中，纯用户级线程的实现，除算法研究目的以外，几乎已经消失了。

Linux 内核只提供了轻量进程的支持，限制了更高效的线程模型的实现，但 Linux 着重优化了进程的调度开销，一定程度上也弥补了这一缺陷。目前最流行的线程机制 LinuxThreads 所采用的就是线程-进程"一对一"模型，调度交给核心，而在用户级实现一个包括信号处理在内的线程管理机制。 Linux-LinuxThreads 的运行机制正是本文的描述重点。

二.Linux 2.4 内核中的轻量进程实现

最初的进程定义都包含程序、资源及其执行三部分，其中程序通常指代码，资源在操作系统层面上通常包括内存资源、IO 资源、信号处理等部分，而程序的执行通常理解为执行上下文，包括对 cpu 的占用，后来发展为线程。在线程概念出现以前，为了减小进程切换的开销，操作系统设计者逐渐修正进程的概念，逐渐允许将进程所占有的资源从其主体剥离出来，允许某些进程共享一部分资源，例如文件、信号，数据内存，甚至代码，这就发展出轻量进程的概念。Linux 内核在 2.0.x 版本就已经实现了轻量进程，应用程序可以通过一个统一的 clone()系统调用接口，用不同的参数指定创建轻量进程还是普通进程。在内核中，clone()调用经过参数传递和解释后会调用 do_fork()，这个核内函数同时也是 fork()、vfork()系统调用的最终实现：

<linux-2.4.20/kernel/fork.c>;

int do_fork(unsigned long clone_flags, unsigned long stack_start,

struct pt_regs *regs, unsigned long stack_size)

其中的 clone_flags 取自以下宏的"或"值：

<linux-2.4.20/include/linux/sched.h>;

```
#define CSIGNAL                 0x000000ff      /* signal mask to be sent at exit */
#define CLONE_VM                0x00000100      /* set if VM shared between processes */

#define CLONE_FS                0x00000200      /* set if fs info shared between processes */
#define CLONE_FILES             0x00000400      /* set if open files shared between processes */
#define CLONE_SIGHAND           0x00000800      /* set if signal handlers and bbcked signals shared */
```

```
#define CLONE_PID               0x00001000      /* set if pid shared */
#define CLONE_PTRACE            0x00002000      /* set if we want to let tracing continue on the child too */
#define CLONE_VFORK             0x00004000      /* set if the parent wants the child to wake it up on mm_release */
#define CLONE_PARENT            0x00008000      /* set if we want to have the same parent as the cloner */
#define CLONE_THREAD            0x00010000      /* Same thread group? */
#define CLONE_NEWNS             0x00020000      /* New namespace group? */
#define CLONE_SIGNAL            (CLONE_SIGHAND | CLONE_THREAD)
```

在 do_fork()中，不同的 clone_flags 将导致不同的行为，对于 LinuxThreads，它使用（CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND）参数来调用 clone()创建"线程"，表示共享内存、共享文件系统访问计数、共享文件描述符表，以及共享信号处理方式。本节就针对这几个参数，看看 Linux 内核是如何实现这些资源的共享的。

1.CLONE_VM

do_fork()需要调用 copy_mm()来设置 task_struct 中的 mm 和 active_mm 项，这两个 mm_struct 数据与进程所关联的内存空间相对应。如果 do_fork()时指定了 CLONE_VM 开关，copy_mm()将把新的 task_struct 中的 mm 和 active_mm 设置成与 current 的相同，同时提高该 mm_struct 的使用者数目（mm_struct::mm_users）。也就是说，轻量级进程与父进程共享内存地址空间，由下图示意可以看出 mm_struct 在进程中的地位：

2.CLONE_FS

task_struct 中利用 fs（struct fs_struct *）记录了进程所在文件系统的根目录和当前目录信息，do_fork()时调用 copy_fs()复制了这个结构；而对于轻量级进程则仅增加 fs->;count 计数，与父进程共享相同的 fs_struct。也就是说，轻量级进程没有独立的文件系统相关的信息，进程中任何一个线程改变当前目录、根目录等信息都将直接影响到其他线程。

3.CLONE_FILES

一个进程可能打开了一些文件，在进程结构 task_struct 中利用 files（struct files_struct *）来保存进程打开的文件结构（struct file）信息，do_fork()中调用了 copy_files()来处理这个进程属性；轻量级进程与父进程是共享该结构的，copy_files() 时仅增加 files->;count 计数。这一共享使得任何线程都能访问进程所维护的打开文件，对它们的操作会直接反映到进程中的其他线程。

4.CLONE_SIGHAND

每一个 Linux 进程都可以自行定义对信号的处理方式，在 task_struct 中的 sig（struct signal_struct）中使用一个 struct k_sigaction 结构的数组来保存这个配置信息，do_fork()中的 copy_sighand()负责复制该信息；轻量级进程不进行复制，而仅仅增加 signal_struct::count 计数，与父进程共享该结构。也就是说，子进程与父进程的信号处理方式完全相同，而且可以相互更改。

do_fork()中所做的工作很多，在此不详细描述。对于 SMP 系统，所有的进程 fork 出来后，都被分配到与父进程相同的 cpu 上，一直到该进程被调度时才会进行 cpu 选择。

尽管 Linux 支持轻量级进程，但并不能说它就支持核心级线程，因为 Linux 的"线程"和"进程"实际上处于一个调度层次，共享一个进程标识符空间，这种限制使得不可能在 Linux 上实现完全意义上的 POSIX 线程机制，因此众多的 Linux 线程库实现尝试都只能尽可能实现 POSIX 的绝大部分语义，并在功能上尽可能逼近。

三.LinuxThread 的线程机制

LinuxThreads 是目前 Linux 平台上使用最为广泛的线程库，由 Xavier Leroy (Xavier.Leroy@inria.fr)负责开发完成，并已绑定在 GLIBC 中发行。它所实现的就是基于核心轻量级进程的"一对一"线程模型，一个线程实体对应一个核心轻量级进程，而线程之间的管理在核外函数库中实现。

1.线程描述数据结构及实现限制

LinuxThreads 定义了一个 struct _pthread_descr_struct 数据结构来描述线程，并使用全局数组变量__pthread_handles 来描述和引用进程所辖线程。在 __pthread_handles 中的前两项，LinuxThreads 定义了两个全局的系统线程：__pthread_initial_thread 和 __pthread_manager_thread，并用__pthread_main_thread 表征 __pthread_manager_thread 的父线程（初始为__pthread_initial_thread）。

struct _pthread_descr_struct 是一个双环链表结构，__pthread_manager_thread 所在的链表仅包括它一个元素，实际上，__pthread_manager_thread 是一个特殊线程，LinuxThreads 仅使用了其中

的 errno、p_pid、 p_priority 等三个域。而__pthread_main_thread 所在的链则将进程中所有用户线程串在了一起。经过一系列 pthread_create()之后形成的__pthread_handles 数组将如下图所

新创建的线程将首先在__pthread_handles 数组中占据一项，然后通过数据结构中的链指针连入以__pthread_main_thread 为首指针的链表中。这个链表的使用在介绍线程的创建和释放的时候将提到。

LinuxThreads 遵循 POSIX1003.1c 标准，其中对线程库的实现进行了一些范围限制，比如进程最大线程数，线程私有数据区大小等等。在 LinuxThreads 的实现中，基本遵循这些限制，但也进行了一定的改动，改动的趋势是放松或者说扩大这些限制，使编程更加方便。这些限定宏主要集中在 sysdeps/unix/sysv/linux/bits/local_lim.h（不同平台使用的文件位置不同）中，包括如下几个：

每进程的私有数据 key 数，POSIX 定义_POSIX_THREAD_KEYS_MAX 为 128，LinuxThreads 使用 PTHREAD_KEYS_MAX，1024；私有数据释放时允许执行的操作数，LinuxThreads 与 POSIX 一致，定义 PTHREAD_DESTRUCTOR_ITERATIONS 为 4；每进程的线程数，POSIX 定义为 64，LinuxThreads 增大到 1024（PTHREAD_THREADS_MAX）；线程运行栈最小空间大小，POSIX 未指定，LinuxThreads 使用 PTHREAD_STACK_MIN，16384（字节）。

2.管理线程

"一对一"模型的好处之一是线程的调度由核心完成了，而其他诸如线程取消、线程间的同步等工作，都是在核外线程库中完成的。在 LinuxThreads 中，专门为每一个进程构造了一个管理线程，负责处理线程相关的管理工作。当进程第一次调用 pthread_create()创建一个线程的时候就会创建（__clone()）并启动管理线程。

在一个进程空间内，管理线程与其他线程之间通过一对"管理管道（manager_pipe[2]）"来通讯，该管道在创建管理线程之前创建，在成功启动了管理线程之后，管理管道的读端和写端分别赋给两个全局变量__pthread_manager_reader 和 __pthread_manager_request，之后，每个用户线程都通过__pthread_manager_request 向管理线程发请求，但管理线程本身并没有直接使用__pthread_manager_reader，管道的读端（manager_pipe[0]）是作为 __clone()的参数之一传给管理线程的，管理线程的工作主要就是监听管道读端，并对从中取出的请求作出反应。

创建管理线程的流程如下所示：

（全局变量 pthread_manager_request 初值为-1）

图 3 创建管理线程的流程

初始化结束后，在__pthread_manager_thread 中记录了轻量级进程号以及核外分配和管理的线程 id，2*PTHREAD_THREADS_MAX+1 这个数值不会与任何常规用户线程 id 冲突。管理线程作为 pthread_create()的调用者线程的子线程运行，而 pthread_create()所创建的那个用户线程则是由管理线程来调用 clone()创建，因此实际上是管理线程的子线程。（此处子线程的概念应该当作子进程来理解。）

__pthread_manager()就是管理线程的主循环所在，在进行一系列初始化工作后，进入 while(1)循环。在循环中，线程以 2 秒为 timeout 查询（__poll()）管理管道的读端。在处理请求前，检查其父线程（也就是创建 manager 的主线程）是否已退出，如果已退出就退出整个进程。如果有退出的子线程需要清理，则调用 pthread_reap_children()清理。

然后才是读取管道中的请求，根据请求类型执行相应操作（switch-case）。具体的请求处理，源码中比较清楚，这里就不赘述了。

3.线程栈

在 LinuxThreads 中，管理线程的栈和用户线程的栈是分离的，管理线程在进程堆中通过 malloc()分配一个 THREAD_MANAGER_STACK_SIZE 字节的区域作为自己的运行栈。
用户线程的栈分配办法随着体系结构的不同而不同，主要根据两个宏定义来区分，一个是 NEED_SEPARATE_REGISTER_STACK，这个属性仅在 IA64 平台上使用；另一个是 FLOATING_STACK 宏，在 i386 等少数平台上使用，此时用户线程栈由系统决定具体位置并提供保护。与此同时，用户还可以通过线程属性结构来指定使用用户自定义的栈。因篇幅所限，这里只能分析 i386 平台所使用的两种栈组织方式：FLOATING_STACK 方式和用户自定义方式。

在 FLOATING_STACK 方式下，LinuxThreads 利用 mmap()从内核空间中分配 8MB 空间（i386 系统缺省的最大栈空间大小，如果有运行限制（rlimit），则按照运行限制设置），使用 mprotect()设置其中第一页为非访问区。该 8M 空间的功能分配如下图：

图 4 栈结构示意

低地址被保护的页面用来监测栈溢出。

对于用户指定的栈，在按照指针对界后，设置线程栈顶，并计算出栈底，不做保护，正确性由用户自己保证。

不论哪种组织方式，线程描述结构总是位于栈顶紧邻堆栈的位置。

4.线程 id 和进程 id

每个 LinuxThreads 线程都同时具有线程 id 和进程 id，其中进程 id 就是内核所维护的进程号，而线程 id 则由 LinuxThreads 分配和维护。

__pthread_initial_thread 的线程 id 为 PTHREAD_THREADS_MAX，__pthread_manager_thread 的是 2*PTHREAD_THREADS_MAX+1，第一个用户线程的线程 id 为 PTHREAD_THREADS_MAX+2，此后第 n 个用户线程的线程 id 遵循以下公式：

tid=n*PTHREAD_THREADS_MAX+n+1

这种分配方式保证了进程中所有的线程（包括已经退出）都不会有相同的线程 id，而线程 id 的类型 pthread_t 定义为无符号长整型（unsigned long int），也保证了有理由的运行时间内线程 id 不会重复。

从线程 id 查找线程数据结构是在 pthread_handle()函数中完成的，实际上只是将线程号按 PTHREAD_THREADS_MAX 取模，得到的就是该线程在__pthread_handles 中的索引。

5.线程的创建

在 pthread_create()向管理线程发送 REQ_CREATE 请求之后，管理线程即调用 pthread_handle_create()创建新线程。分配栈、设置 thread 属性后，以 pthread_start_thread()为函数入口调用__clone()创建并启动新线程。 pthread_start_thread()读取自身的进程 id 号存入线程描述结构中，并根据其中记录的调度方法配置调度。一切准备就绪后，再调用真正的线程执行函数，并在此函数返回后调用 pthread_exit()清理现场。

6.LinuxThreads 的不足

由于 Linux 内核的限制以及实现难度等等原因，LinuxThreads 并不是完全 POSIX 兼容的，在它

的发行 README 中有说明。

1)进程 id 问题

这个不足是最关键的不足，引起的原因牵涉到 LinuxThreads 的"一对一"模型。

Linux 内核并不支持真正意义上的线程，LinuxThreads 是用与普通进程具有同样内核调度视图的轻量级进程来实现线程支持的。这些轻量级进程拥有独立的进程 id，在进程调度、信号处理、IO 等方面享有与普通进程一样的能力。在源码阅读者看来，就是 Linux 内核的 clone()没有实现对 CLONE_PID 参数的支持。
在内核 do_fork()中对 CLONE_PID 的处理是这样的：

```
if (clone_flags & CLONE_PID) {
if (current->;pid)
goto fork_out;
}
```

这段代码表明，目前的 Linux 内核仅在 pid 为 0 的时候认可 CLONE_PID 参数，实际上，仅在 SMP 初始化，手工创建进程的时候才会使用 CLONE_PID 参数。
按照 POSIX 定义，同一进程的所有线程应该共享一个进程 id 和父进程 id，这在目前的"一对一"模型下是无法实现的。

2)信号处理问题
如果核心不提供实时信号，LinuxThreads 将使用 SIGUSR1 和 SIGUSR2 作为内部使用的 restart 和 cancel 信号，这样应用程序就不能使用这两个原本为用户保留的信号了。在 Linux kernel 2.1.60 以后的版本都支持扩展的实时信号（从_SIGRTMIN 到_SIGRTMAX），因此不存在这个问题。

某些信号的缺省动作难以在现行体系上实现，比如 SIGSTOP 和 SIGCONT，LinuxThreads 只能将一个线程挂起，而无法挂起整个进程。

3)线程总数问题

LinuxThreads 将每个进程的线程最大数目定义为 1024，但实际上这个数值还受到整个系统的总进程数限制，这又是由于线程其实是核心进程。

在 kernel 2.4.x 中，采用一套全新的总进程数计算方法，使得总进程数基本上仅受限于物理内存的大小，计算公式在 kernel/fork.c 的 fork_init()函数中：

max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 8

在 i386 上，THREAD_SIZE=2*PAGE_SIZE，PAGE_SIZE=2^12（4KB），mempages=物理内存大小 /PAGE_SIZE，对于 256M 的内存的机器，mempages=256*2^20/2^12=256*2^8，此时最大线程数为 4096。

但为了保证每个用户（除了 root）的进程总数不至于占用一半以上物理内存，fork_init()中继续指定：

init_task.rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;

init_task.rlim[RLIMIT_NPROC].rlim_max = max_threads/2;

这些进程数目的检查都在 do_fork()中进行，因此，对于 LinuxThreads 来说，线程总数同时受这三个因素的限制。

4)管理线程问题

管理线程容易成为瓶颈，这是这种结构的通病；同时，管理线程又负责用户线程的清理工作，因此，尽管管理线程已经屏蔽了大部分的信号，但一旦管理线程死亡，用户线程就不得不手工清理了，而且用户线程并不知道管理线程的状态，之后的线程创建等请求将无人处理。

5)同步问题

LinuxThreads 中的线程同步很大程度上是建立在信号基础上的，这种通过内核复杂的信号处理机制的同步方式，效率一直是个问题。

6）其他 POSIX 兼容性问题

Linux 中很多系统调用，按照语义都是与进程相关的，比如 nice、setuid、setrlimit 等，在目前的 LinuxThreads 中，这些调用都仅仅影响调用者线程。

7）实时性问题

线程的引入有一定的实时性考虑，但 LinuxThreads 暂时不支持，比如调度选项，目前还没有实现。不仅 LinuxThreads 如此，标准的 Linux 在实时性上考虑都很少。

写好C程序的 10 + 大要点

要写好 C 语言程序，最重要的当然是把要解决的问题分析清楚，设计好解决问题的方案和通过计算实现求解的过程，对问题的求解过程进行科学的结构化的分解。在此基础上进一步考虑如何写程序的时候，下面的建议可能有所帮助。

这些建议中有些是一般性的，不仅仅对 C 语言程序设计有效；也有些是特别针对 C 语言程序设计的。这个表还会进一步修改和扩充，欢迎提出意见。

1）应该特别注意程序的书写格式，让它的形式反映出其内在的意义结构。

程序是最复杂的东西（虽然你开始写的程序很简单，但它们会逐渐变得复杂起来），是需要用智力去把握的智力产品。良好的格式能使程序结构一目了然，帮助你和别人理解它，帮助你的思维，也帮助你发现程序中不正常的地方，使程序中的错误更容易被发现。

人们常用的格式形式是：逻辑上属于同一个层次的互相对齐；逻辑上属于内部层次的推到下一个对齐位置。请参考本课程的教科书或《C 程序设计语言》（The C Programming Language，Brian W. Kernighan & Dennis M. Rirchie，清华大学出版社，大学计算机教育丛书（影印版，英文），1996。）

利用集成开发环境（IDE）或者其他程序编辑器的功能，可以很方便地维护好程序的良好格式。请注意下面这几个键，在写程序中应该经常用到它们：Enter 键（换一行），Tab 键（将输入光标移到下一个对齐位置 ——进入新的一个层次），Backspace 键（回到前一个对齐位置——退到外面的一个层次）。

2）用最规范的、最清晰的、最容易理解的方式写程序。注意人们在用 C 语言写程序的习惯写法，例如教科书中解决类似问题时所使用的写法，《C 程序设计语言》一书中有许多极好的程序实例。在这里有一个关于程序模式的相关网页，里面也列出了一些常用的模式。

C 语言是一个非常灵活的语言，你可能在这里用许多非常隐晦的方式写程序，但这样写出的程序只能是作为一种玩意儿，就像谜语或者智力游戏。这些东西可以用于消磨时间，但通常与实际无缘。在我们的 C 语言讨论组里提到过这种东西。

3）在编程中，应仔细研究编译程序给出的错误信息和警告信息，弄清楚每条信息的确切根源并予以解决。特别是，不要忽略那些警告信息，许多警告信息源自隐含的严重错误。我们有许多办法去欺骗编译程序，使它不能发现我们程序中的错误，但这样做最终受到伤害的只能是自己。

4）随时注意表达式计算过程和类型。注意运算符的优先级和结合顺序，不同类型的运算对象将怎样转换，运算的结果是什么类型的，等等。在必要的时候加上括号或显式的类型强制转换。

C 语言的运算符很多，优先级定义也不尽合理，很难完全记清楚，因此要特别注意。需要时查一查（不要怕麻烦，相关网页有运算符表），或者直接按照自己的需要加上几个括号。

5）绝不去写依赖于运算对象求值顺序的表达式。对于普通二元运算符的运算对象，函数调用的各个实际参数，C 语言都没有规定特定求值顺序。因此，我们不应该写那种依赖于特定求值顺序的表达式，因为不能保证它一定得到什么结果。例如下面的表达式和函数调用都是不合适的，很可能产生你预料不到的结果：

```
scanf("%d %d", i++, a[i]);
m = n * n++;
```

6）总保证一个函数的定义点和它的所有使用点都能看到同一个完整的函数原型说明。参看《从问题到程序》第 103-107 页。

7）总注意检查数组的界限和字符串（也以数组的方式存放）的结束。C 语言内部根本不检查数组下标表达式的取值是否在合法范围内，也不检查指向数组元素的指针是不是移出了数组的合法区域。写程序的人需要自己保证对数组使用的合法性。越界访问可能造成灾难性的后果。例：在写处理数组的函数时一般应该有一个范围参数；处理字符串时总检查是否遇到空字符'\0'。

8）绝不对空指针或者悬空的指针做间接访问。这种访问的后果不可预料，可能造成系统的破坏，也可能造成操作系统发现这个程序执行非法操作而强制将它终止。

9）对于所有通过返回值报告运行情况或者出错信息的库函数，都应该检查其执行是否正常完成。如果库函数没有完成操作（可能因为各种原因），随后的操作有可能就是非法的。这种错误也可能在程序运行中隐藏很长时间，到很后来才暴露出来，检查错误非常困难。

10）在带参数宏的定义字符串中，一般应该给整个字符串和其中出现的每个参数都加括号。C 语言预处理程序是个简单的文本替换程序，它根本不知道 C 语言的语法结构、优先级规则等。不写括号有时会产生我们不希望的代换结果。

11）所有外部变量名字、所有函数名字，应该只靠前 6 个字符就能够互相区分。因为有些老的编译程序只关注这些名字的前 6 个字符。如果不注意这个问题，就可能引起隐含的连接错误。

有关如何写好程序，如何将自己发展成为一个高水平的计算机工作者（真正的高级程序工作者，而不是那种拿证书的所谓“高级程序员”）还需要进一步学习和实践。如果希望向这个方向努力，我翻译的一本书可能对你有所帮助：

《程序设计实践》，（The Practice Of Programming, Brian W. Kernighan & Bob Pike 1999)。机械工业出版社 2000。这本书讨论了程序设计的许多重要的实践性问题，值得每个学过一个程序设计语言，有些程序设计经验，热爱或梦想在计算机领域工作的人阅读参考。在这里可以看到原书的前言和章节表。还有我为该书中文版写的译者序言。

几个预编译指令的使用

预处理过程扫描源代码，对其进行初步的转换，产生新的源代码提供给编译器。可见预处理过程先于编译器对源代码进行处理。

在 C 语言中，并没有任何内在的机制来完成如下一些功能：在编译时包含其他源文件、定义宏、根据条件决定编译时是否包含某些代码。要完成这些工作，就需要使用预处理程序。尽管在目前绝大多数编译器都包含了预处理程序，但通常认为它们是独立于编译器的。预处理过程读入源代码，检查包含预处理指令的语句和宏定义，并对源代码进行响应的转换。预处理过程还会删除程序中的注释和多余的空白字符。

预处理指令是以#号开头的代码行。#号必须是该行除了任何空白字符外的第一个字符。#后是指令关键字，在关键字和#号之间允许存在任意个数的空白字符。整行语句构成了一条预处理指令，该指令将在编译器进行编译之前对源代码做某些转换。下面是部分预处理指令：

指令	用途
#	空指令，无任何效果
#include	包含一个源代码文件
#define	定义宏
#undef	取消已定义的宏
#if	如果给定条件为真，则编译下面代码
#ifdef	如果宏已经定义，则编译下面代码
#ifndef	如果宏没有定义，则编译下面代码
#elif	如果前面的#if 给定条件不为真，当前条件为真，则编译下面代码
#endif	结束一个#if……#else 条件编译块
#error	停止编译并显示错误信息

一、文件包含

#include 预处理指令的作用是在指令处展开被包含的文件。包含可以是多重的，也就是说一个被包含的文件中还可以包含其他文件。标准 C 编译器至少支持八重嵌套包含。
预处理过程不检查在转换单元中是否已经包含了某个文件并阻止对它的多次包含。这样就可以在多次包含同一个头文件时，通过给定编译时的条件来达到不同的效果。例如：

```
#define AAA
#include "t.c"
#undef AAA
#include "t.c"
```

为了避免那些只能包含一次的头文件被多次包含，可以在头文件中用编译时条件来进行控制。例如：

```
/*my.h*/
#ifndef MY_H
#define MY_H
……
#endif
```

在程序中包含头文件有两种格式：

```
#include <my.h>
#include "my.h"
```

第一种方法是用尖括号把头文件括起来。这种格式告诉预处理程序在编译器自带的或外部库的头文件中搜索被包含的头文件。第二种方法是用双引号把头文件括起来。这种格式告诉预处理程序在当前被编译的应用程序的源代码文件中搜索被包含的头文件，如果找不到，再搜索编译器自带的头文件。

采用两种不同包含格式的理由在于，编译器是安装在公共子目录下的，而被编译的应用程序是在它们自己的私有子目录下的。一个应用程序既包含编译器提供的公共头文件，也包含自定义的私有头文件。采用两种不同的包含格式使得编译器能够在很多头文件中区别出一组公共的头文件。

二、宏

宏定义了一个代表特定内容的标识符。预处理过程会把源代码中出现的宏标识符替换成宏定义时的值。宏最常见的用法是定义代表某个值的全局符号。宏的第二种用法是定义带参数的宏，这样的宏可以象函数一样被调用，但它是在调用语句处展开宏，并用调用时的实际参数来代替定义中的形式参数。

1.#define 指令

#define 预处理指令是用来定义宏的。该指令最简单的格式是：首先神明一个标识符，然后给出这个标识符代表的代码。在后面的源代码中，就用这些代码来替代该标识符。这种宏把程序中要用到的一些全局值提取出来，赋给一些记忆标识符。

```
#define MAX_NUM 10
int array[MAX_NUM];
for(i=0;i<MAX_NUM;i++)    /*……*/
```

在这个例子中，对于阅读该程序的人来说，符号 MAX_NUM 就有特定的含义，它代表的值给出了数组所能容纳的最大元素数目。程序中可以多次使用这个值。作为一种约定，习惯上总是全部用大写字母来定义宏，这样易于把程序红的宏标识符和一般变量标识符区别开来。如果想要改变数组的大小，只需要更改宏定义并重新编译程序即可。

宏表示的值可以是一个常量表达式，其中允许包括前面已经定义的宏标识符。例如：

```
#define ONE 1
#define TWO 2
#define THREE (ONE+TWO)
```

注意上面的宏定义使用了括号。尽管它们并不是必须的。但出于谨慎考虑，还是应该加上括号的。例如：

six=THREE*TWO;

预处理过程把上面的一行代码转换成：

six=(ONE+TWO)*TWO;

如果没有那个括号，就转换成 six=ONE+TWO*TWO;了。

宏还可以代表一个字符串常量，例如：

#define VERSION "Version 1.0 Copyright(c) 2003"

2.带参数的#define 指令

带参数的宏和函数调用看起来有些相似。看一个例子：

```
#define Cube(x) (x)*(x)*(x)
```

可以时任何数字表达式甚至函数调用来代替参数 x。这里再次提醒大家注意括号的使用。宏展开后完全包含在一对括号中，而且参数也包含在括号中，这样就保证了宏和参数的完整性。看一个用法：

```
int num=8+2;
volume=Cube(num);
```

展开后为(8+2)*(8+2)*(8+2);

如果没有那些括号就变为 8+2*8+2*8+2 了。

下面的用法是不安全的：

```
volume=Cube(num++);
```

如果 Cube 是一个函数，上面的写法是可以理解的。但是，因为 Cube 是一个宏，所以会产生副作用。这里的擦书不是简单的表达式，它们将产生意想不到的结果。它们展开后是这样的：

```
volume=(num++)*(num++)*(num++);
```

很显然，结果是 10*11*12,而不是 10*10*10;

那么怎样安全的使用 Cube 宏呢？必须把可能产生副作用的操作移到宏调用的外面进行：

```
int num=8+2;
volume=Cube(num);
num++;
```

3.#运算符

出现在宏定义中的#运算符把跟在其后的参数转换成一个字符串。有时把这种用法的#称为字符串化运算符。例如：

```
#define PASTE(n) "adhfkj"#n
main()
{
printf("%s\n",PASTE(15));
}
```

宏定义中的#运算符告诉预处理程序，把源代码中任何传递给该宏的参数转换成一个字符串。所以输出应该是 adhfkj15。

4.##运算符

##运算符用于把参数连接到一起。预处理程序把出现在##两侧的参数合并成一个符号。看下面

的例子：

```
#define NUM(a,b,c) a##b##c
#define STR(a,b,c) a##b##c
main()
{
printf("%d\n",NUM(1,2,3));
printf("%s\n",STR("aa","bb","cc"));
}
```

最后程序的输出为:

```
123
aabbcc
```

千万别担心，除非需要或者宏的用法恰好和手头的工作相关，否则很少有程序员会知道##运算符。绝大多数程序员从来没用过它。

三、条件编译指令

条件编译指令将决定那些代码被编译，而哪些是不被编译的。可以根据表达式的值或者某个特定的宏是否被定义来确定编译条件。

1.#if 指令

#if 指令检测跟在制造另关键字后的常量表达式。如果表达式为真，则编译后面的代码，知道出现#else、#elif 或#endif 为止；否则就不编译。

2.#endif 指令

#endif 用于终止#if 预处理指令。

```
#define DEBUG 0
main()
{
#if DEBUG
printf("Debugging\n");
#endif
printf("Running\n");
}
```

由于程序定义 DEBUG 宏代表 0，所以#if 条件为假，不编译后面的代码直到#endif，所以程序直接输出 Running。

如果去掉#define 语句，效果是一样的。

3.#ifdef 和#ifndef

```
#define DEBUG
main()
{

#ifdef DEBUG
printf("yes\n");
#endif
#ifndef DEBUG
printf("no\n");
#endif
}
```

#if defined 等价于#ifdef; #if !defined 等价于#ifndef

4.#else 指令

#else 指令用于某个#if 指令之后，当前面的#if 指令的条件不为真时，就编译#else 后面的代码。

#endif 指令将中指上面的条件块。

```
#define DEBUG
main()
{
#ifdef DEBUG
printf("Debugging\n");
#else
printf("Not debugging\n");

#endif
printf("Running\n");
}
```

5.#elif 指令

#elif 预处理指令综合了#else 和#if 指令的作用。

```
#define TWO
main()
{
#ifdef ONE
printf("1\n");
#elif defined TWO
printf("2\n");
```

```
#else
printf("3\n");
#endif
}
```

程序很好理解，最后输出结果是 2。

6.其他一些标准指令

#error 指令将使编译器显示一条错误信息，然后停止编译。

#line 指令可以改变编译器用来指出警告和错误信息的文件号和行号。

#pragma 指令没有正式的定义。编译器可以自定义其用途。典型的用法是禁止或允许某些烦人的警告信息。

嵌入式系统的C语言

译自《C for Embedded Systems》讲稿

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。

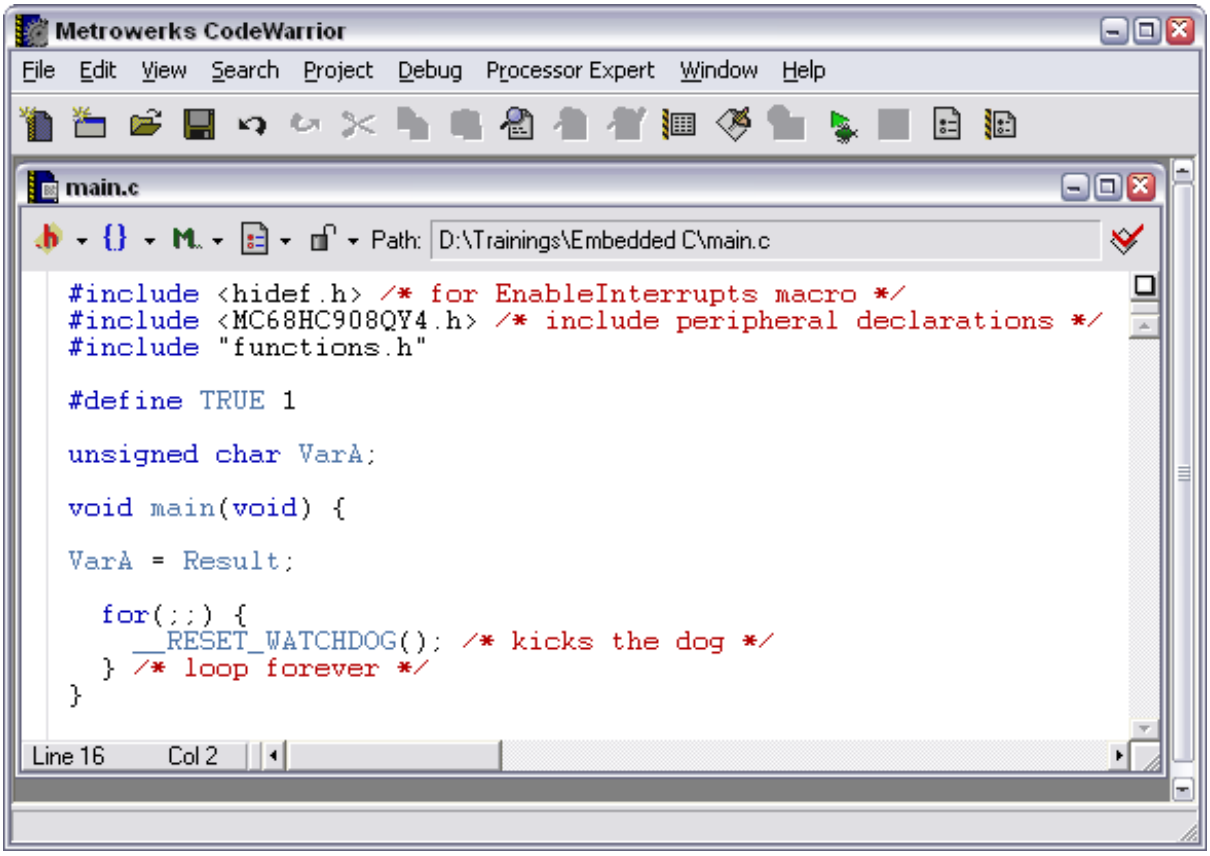
8、为什么改用C？

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：

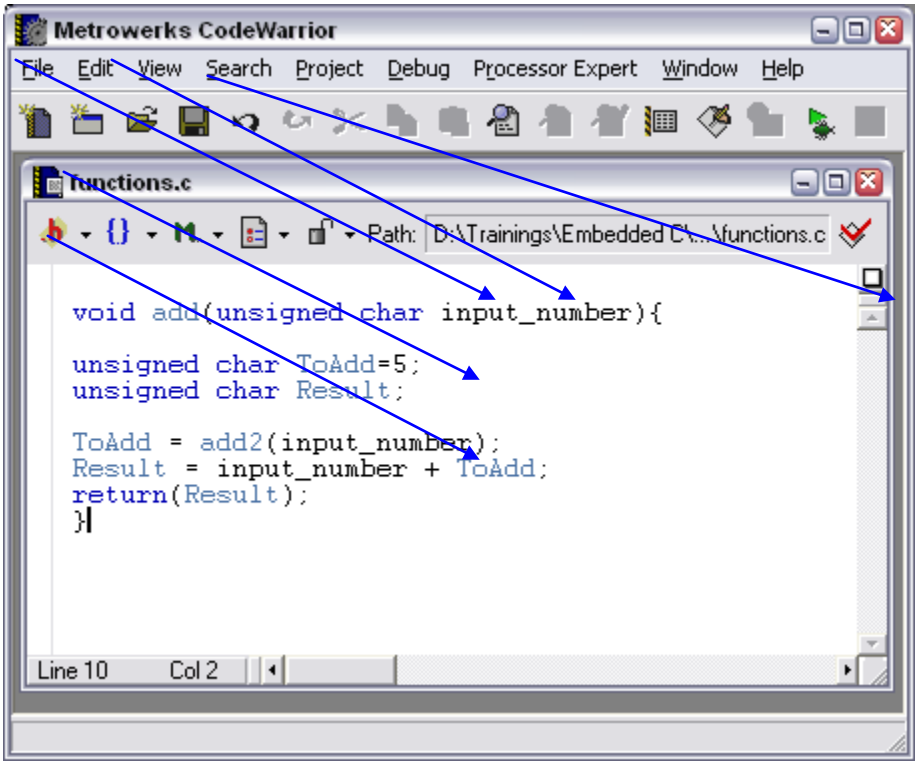
类型 函数名（参数）

{

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

f # fdef # fndef # dif # dse # ndif

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

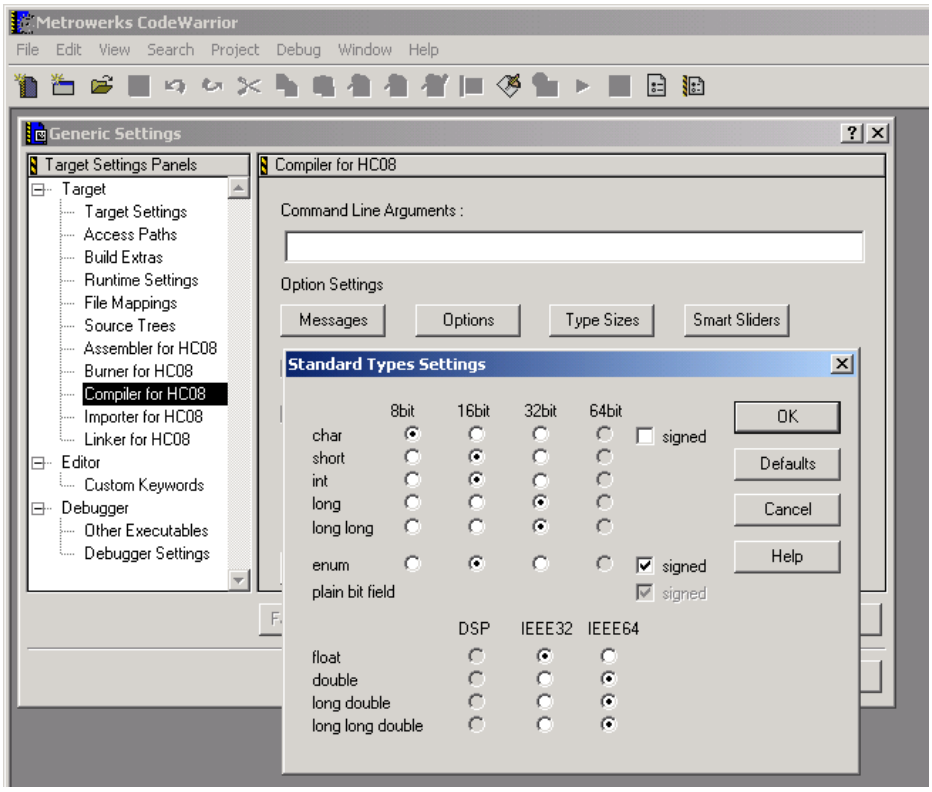
所有数量类型（除了char）缺省都是有符号的，例如：‘int’ = ‘signed int’。

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。

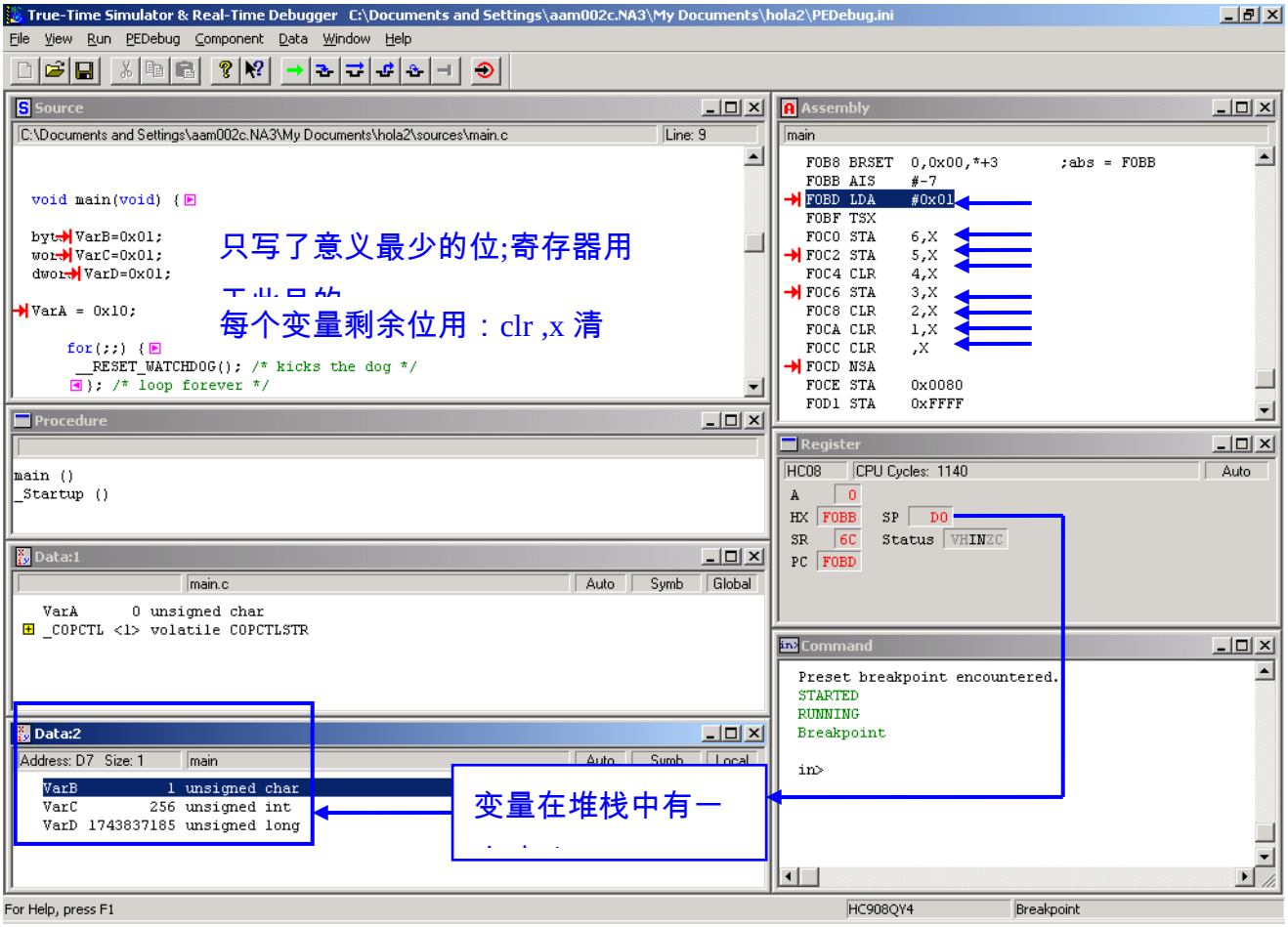
*8-bit machine

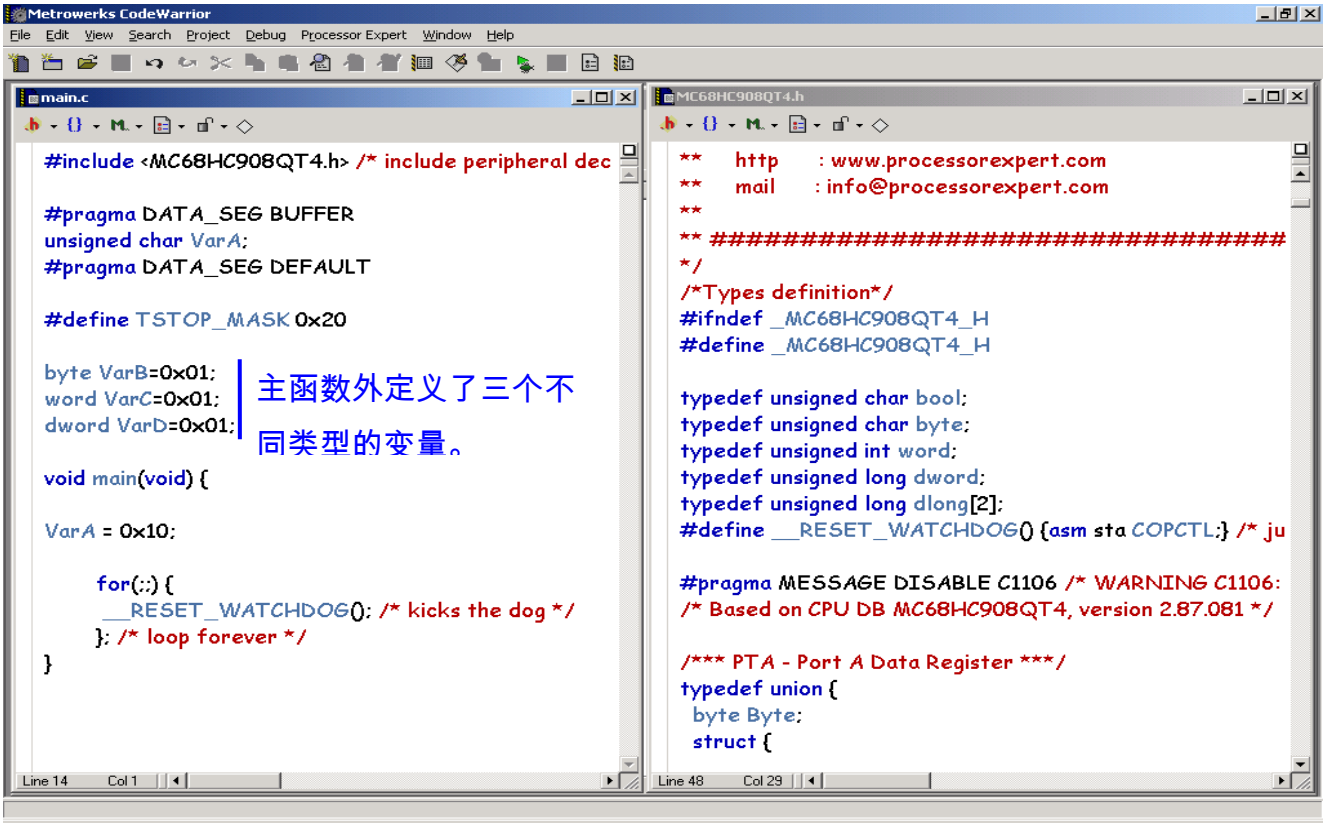
```
/* Fixed size types */
typedef unsigned char  uint8_t;
typedef int            int16_t;
typedef unsigned long  uint32_t;
```

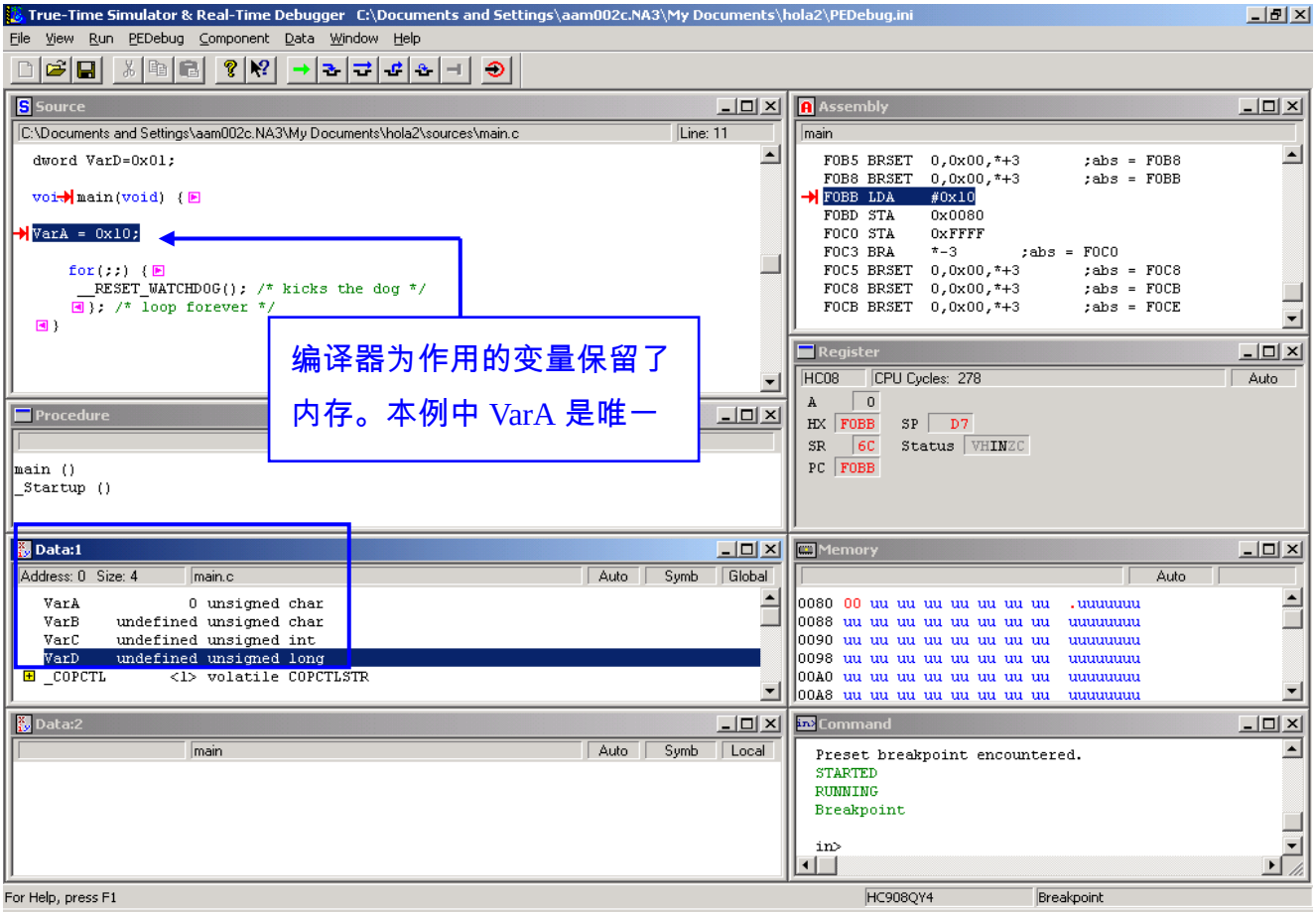
*32-bit machine

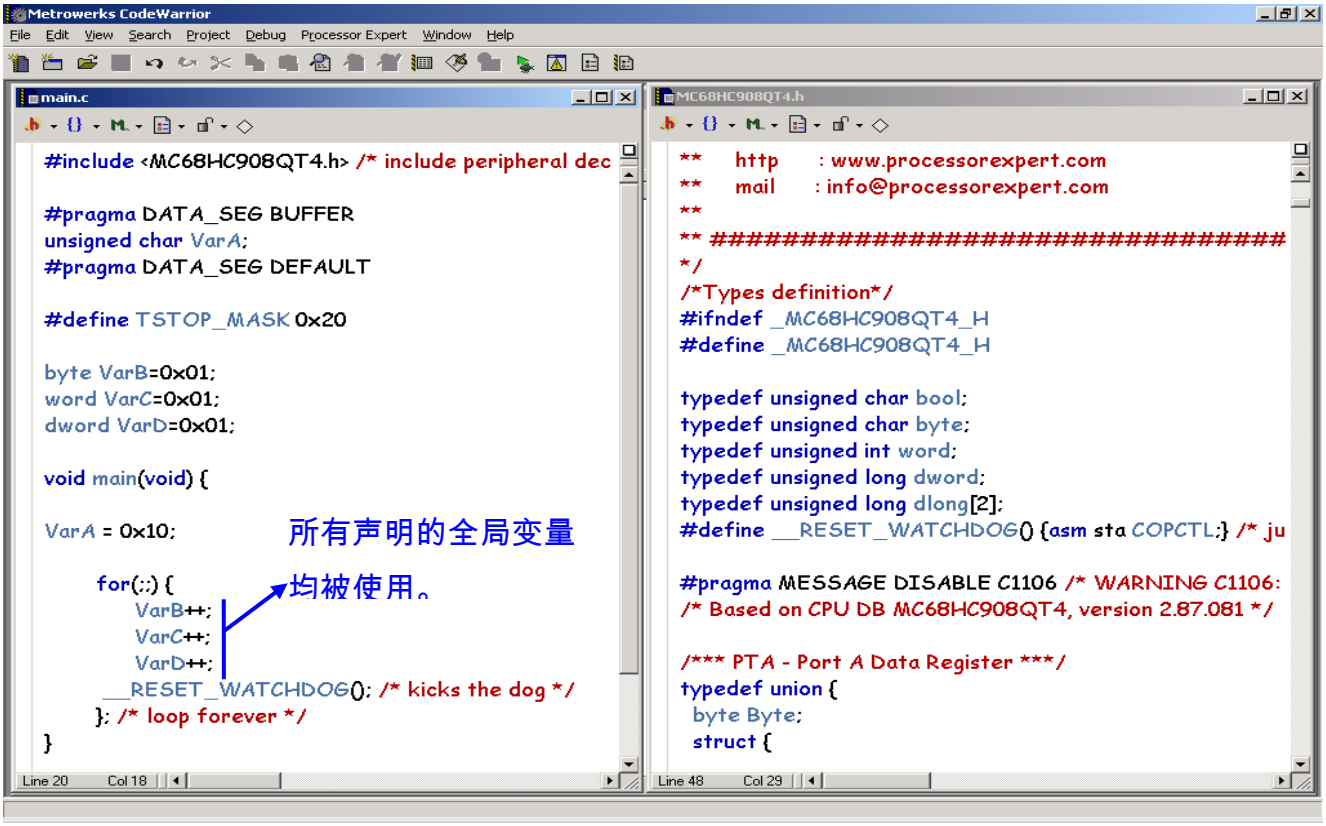
```
/* Fixed size types */
typedef unsigned char  uint8_t;
typedef short          int16_t;
typedef unsigned int   uint32_t;
```

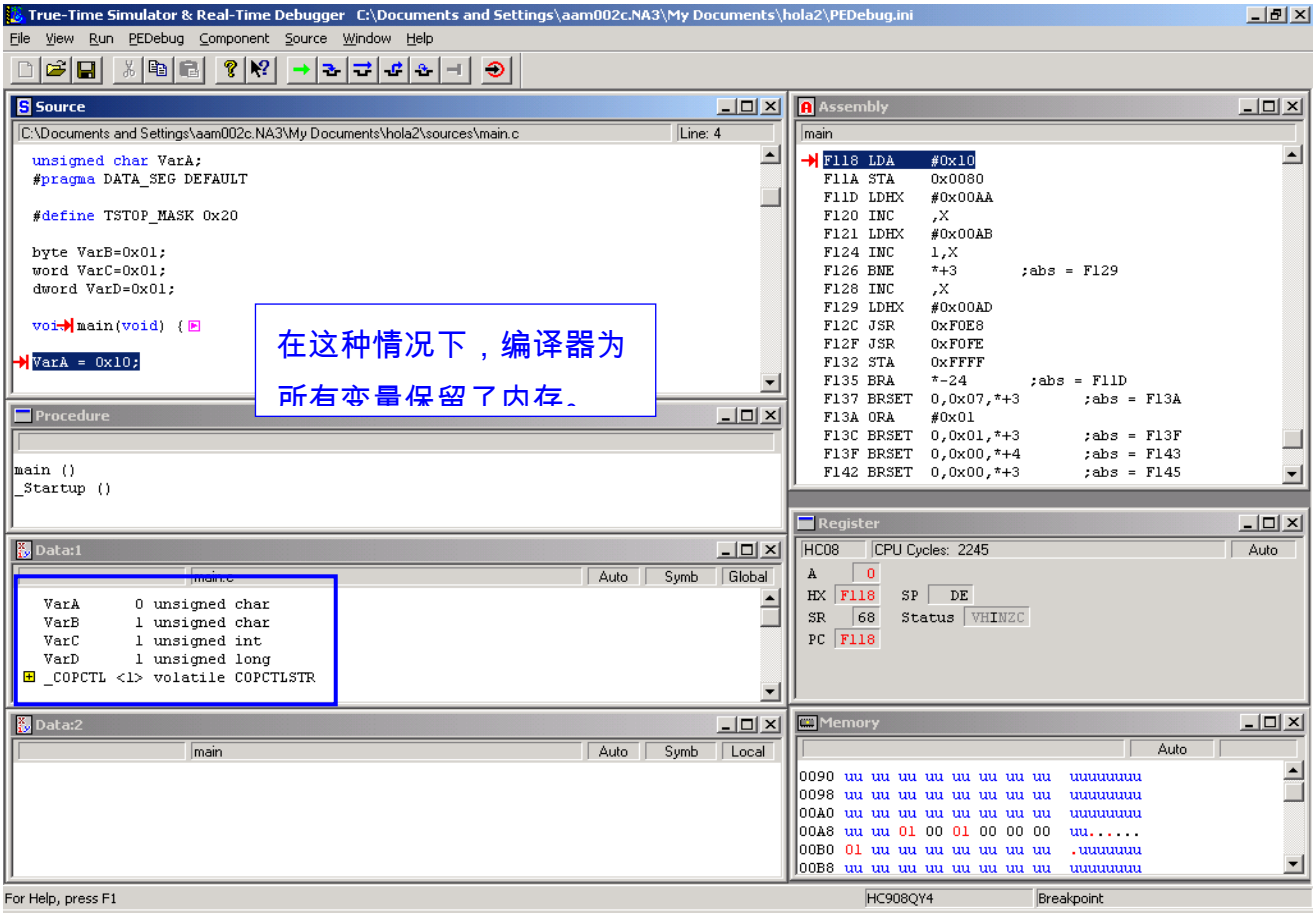
打开文件：**Lab1-Variables.mcp**

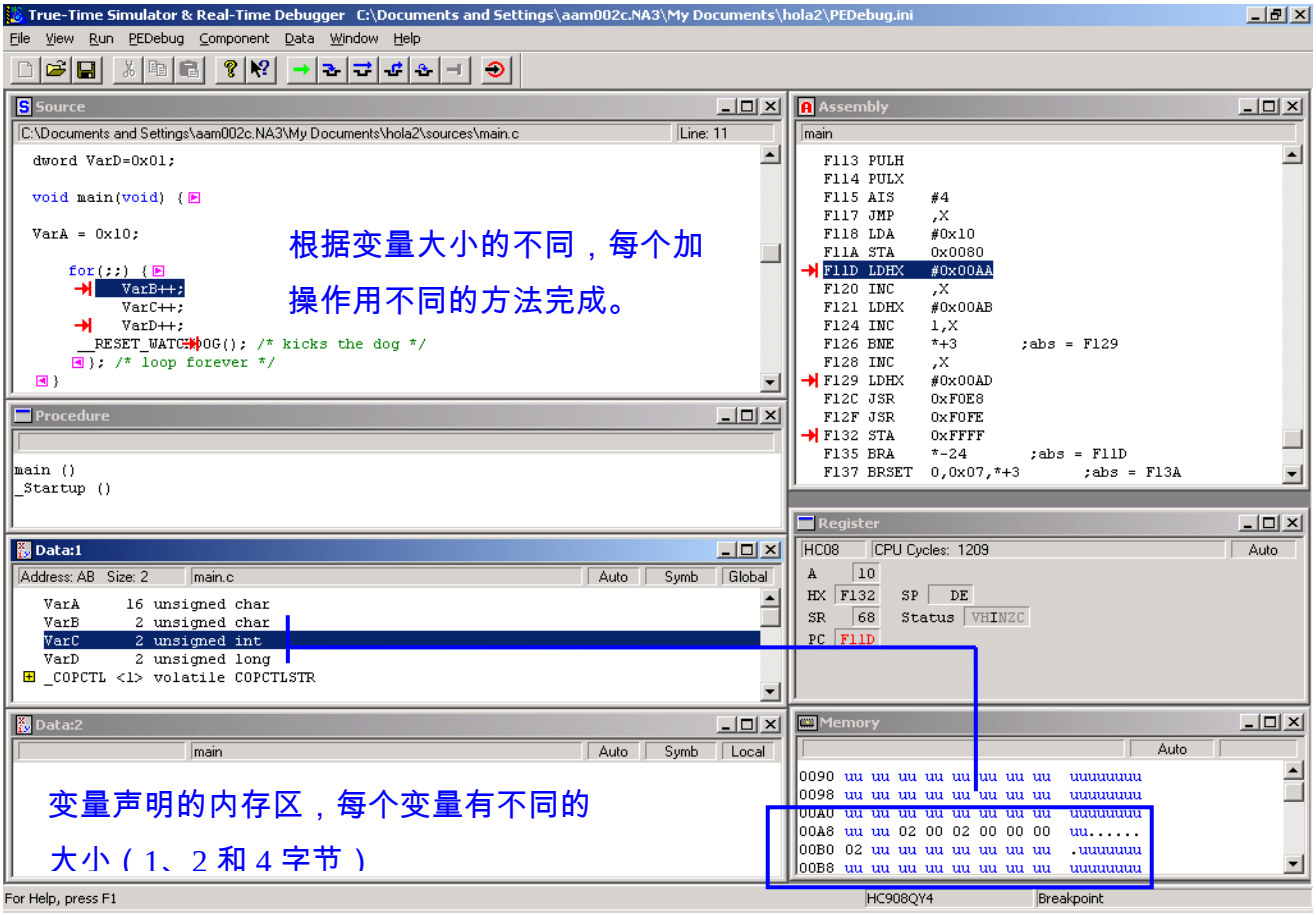

True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Data Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 9
void main(void) {
byt VarB=0x01;
wor VarC=0x01;
dwor VarD=0x01;
VarA = 0x10;
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
只写了意义最少的位;寄存器用
每个变量剩余位用：clr ,x 清
Assembly
main
F0B8 BRSET 0,0x00,*+3 ;abs = F0BB
F0BB AIS #-7
F0BD LDA #0x01
F0BF TSX
F0C0 STA 6,X
F0C2 STA 5,X
F0C4 CLR 4,X
F0C6 STA 3,X
F0C8 CLR 2,X
F0CA CLR 1,X
F0CC CLR ,X
F0CD NSA
F0CE STA 0x0080
F0D1 STA 0xFFFF
Procedure
main ()
_Startup ()
Register
HC08
CPU Cycles: 1140
Auto
A 0
HX F0BB
SP D0
SR 6C
Status VHINZC
PC F0BD
Data:1
main.c
Auto
Symb
Global
VarA 0 unsigned char
_COPCTL <1> volatile COPCTLSTR
Command
Preset breakpoint encountered.
STARTED
RUNNING
Breakpoint
in>
Data:2
Address: D7 Size: 1
main
Auto
Symb
Local
VarB 1 unsigned char
VarC 256 unsigned int
VarD 1743837185 unsigned long
变量在堆栈中有一
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;
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 */
}
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 ()
编译器为作用的变量保留了内存。本例中 VarA 是唯一
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 */
}
所有声明的全局变量
均被使用。
Line 20 Col 18
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 ()
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
Register
HC08 CPU Cycles: 2245 Auto
A 0
HX F118 SP DE
SR 68 Status VHINZC
PC F118
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 pul-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。

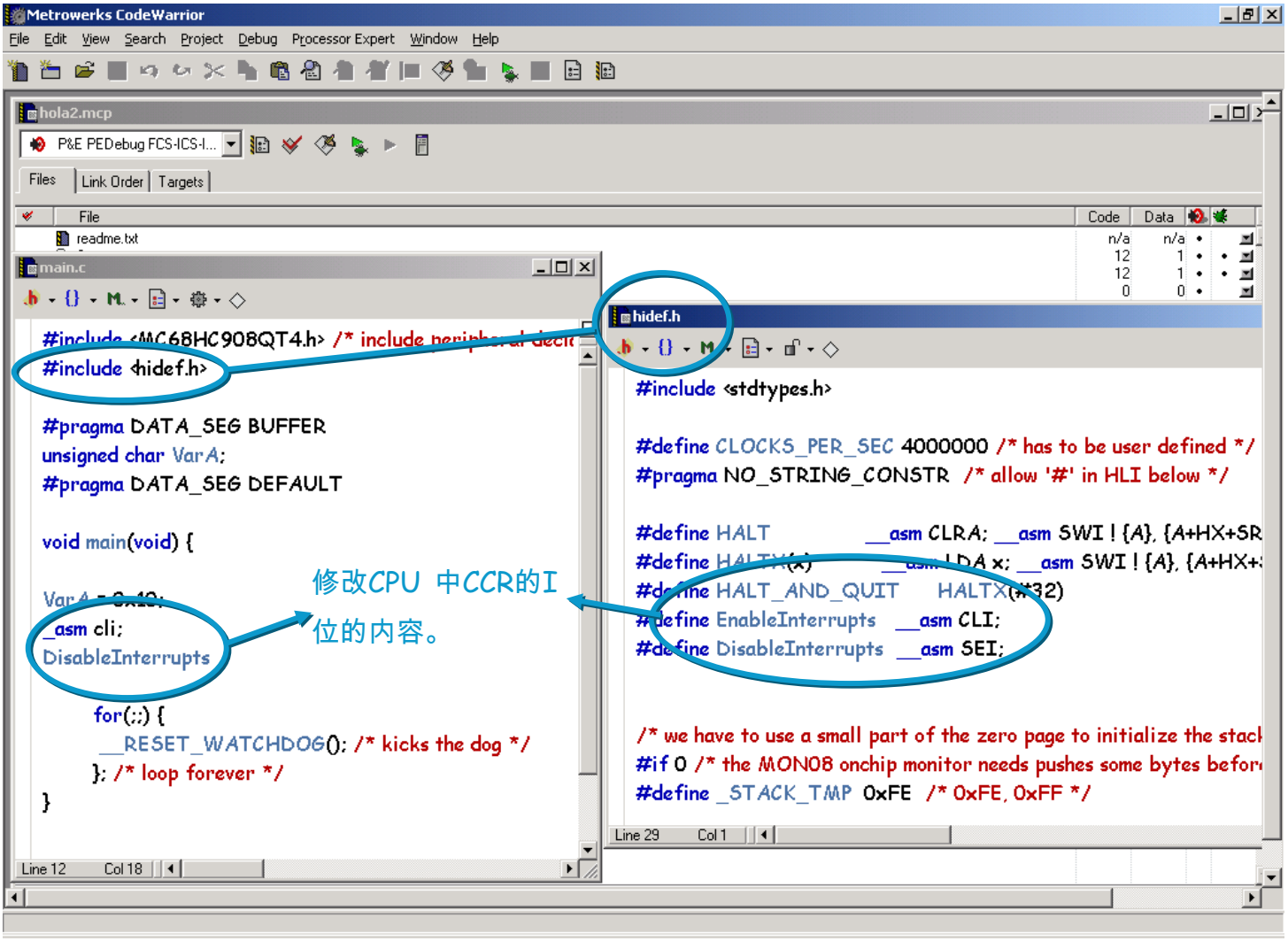
糊涂吗?有点…让我们看一些其它选择：

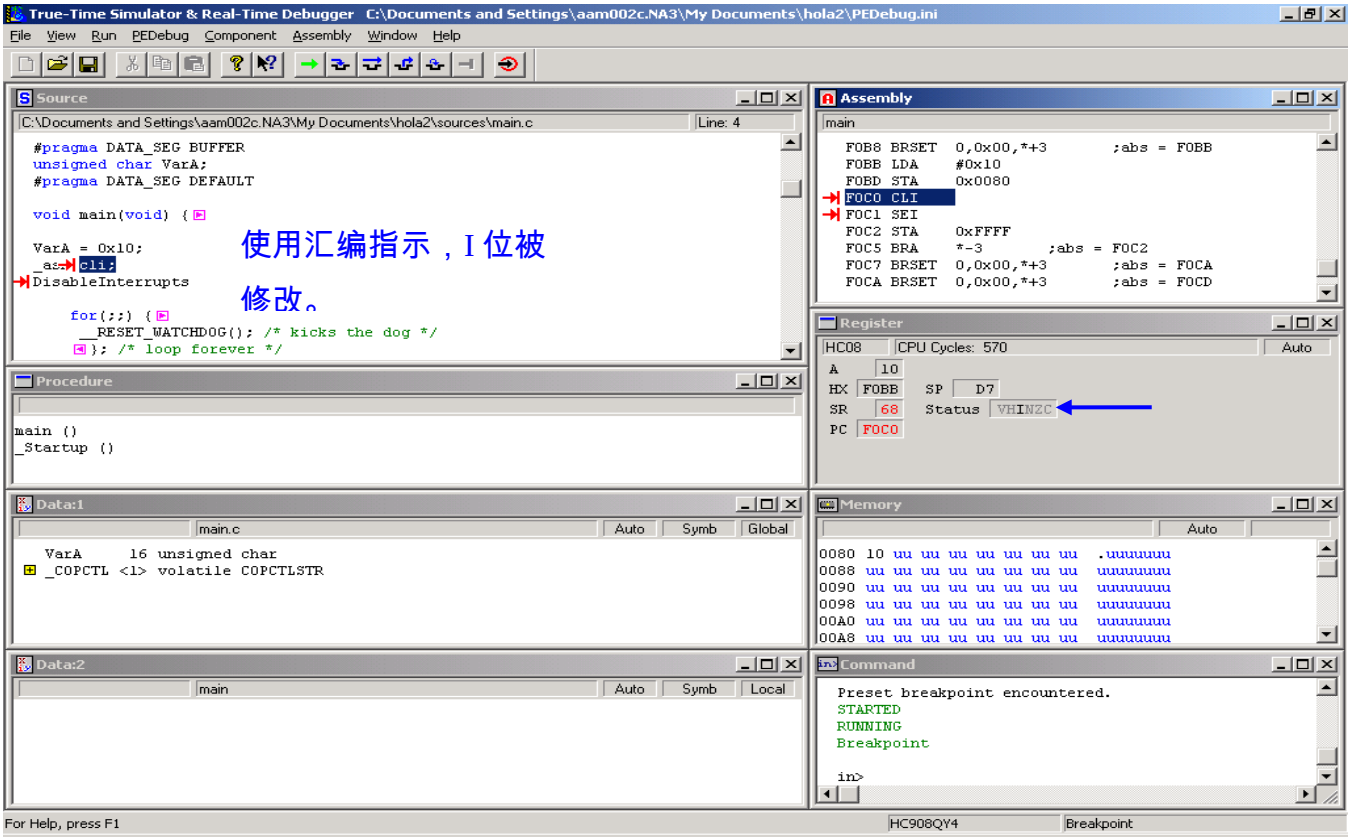
这样做的一个容易的方法是在变量声明中使用符号“@”，创建一个语句读作：在地址 0x0000 处创建一个volatile-unsigned-char型的变量PortA。

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 {
----
----
}
```


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
Code Data
n/a n/a
12 1
12 1
0 0
main.c
#include <MC68HC908QT4.h> /* include peripheral declr
#include <hidef.h>
#pragma DATA_SEG BUFFER
unsigned char VarA;
#pragma DATA_SEG DEFAULT
void main(void) {
VarA = 0x10;
_asm cli;
DisableInterrupts
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
Line 12 Col 18
修改CPU 中CCR的I
位的内容。
hidef.h
#include <stdtypes.h>
#define CLOCKS_PER_SEC 4000000 /* has to be user defined */
#pragma NO_STRING_CONSTR /* allow '#' in HLI below */
#define HALT __asm CLRA; __asm SWI ! {A}, {A+HX+SR
#define HALTX(x) __asm LDA x; __asm SWI ! {A}, {A+HX+
#define HALT_AND_QUIT HALTX(#32)
#define EnableInterrupts __asm CLI;
#define DisableInterrupts __asm SEI;
/* we have to use a small part of the zero page to initialize the stack
#if 0 /* the MON08 onchip monitor needs pushes some bytes before
#define _STACK_TMP 0xFE /* 0xFE, 0xFF */
Line 29 Col 1



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。

$0020	0	0	0	0	1	0	0	1

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

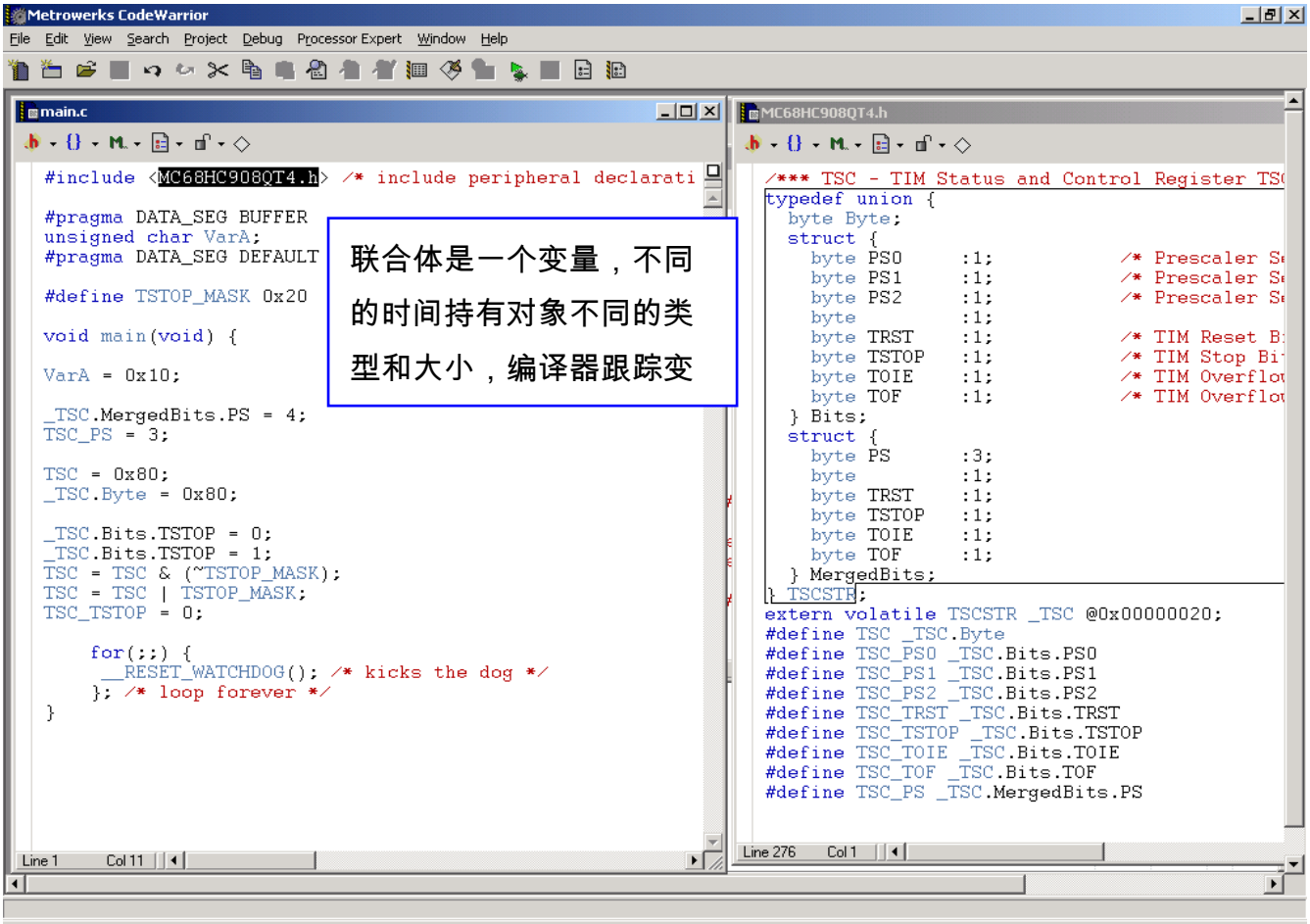
当使用时可提高代码的效率。

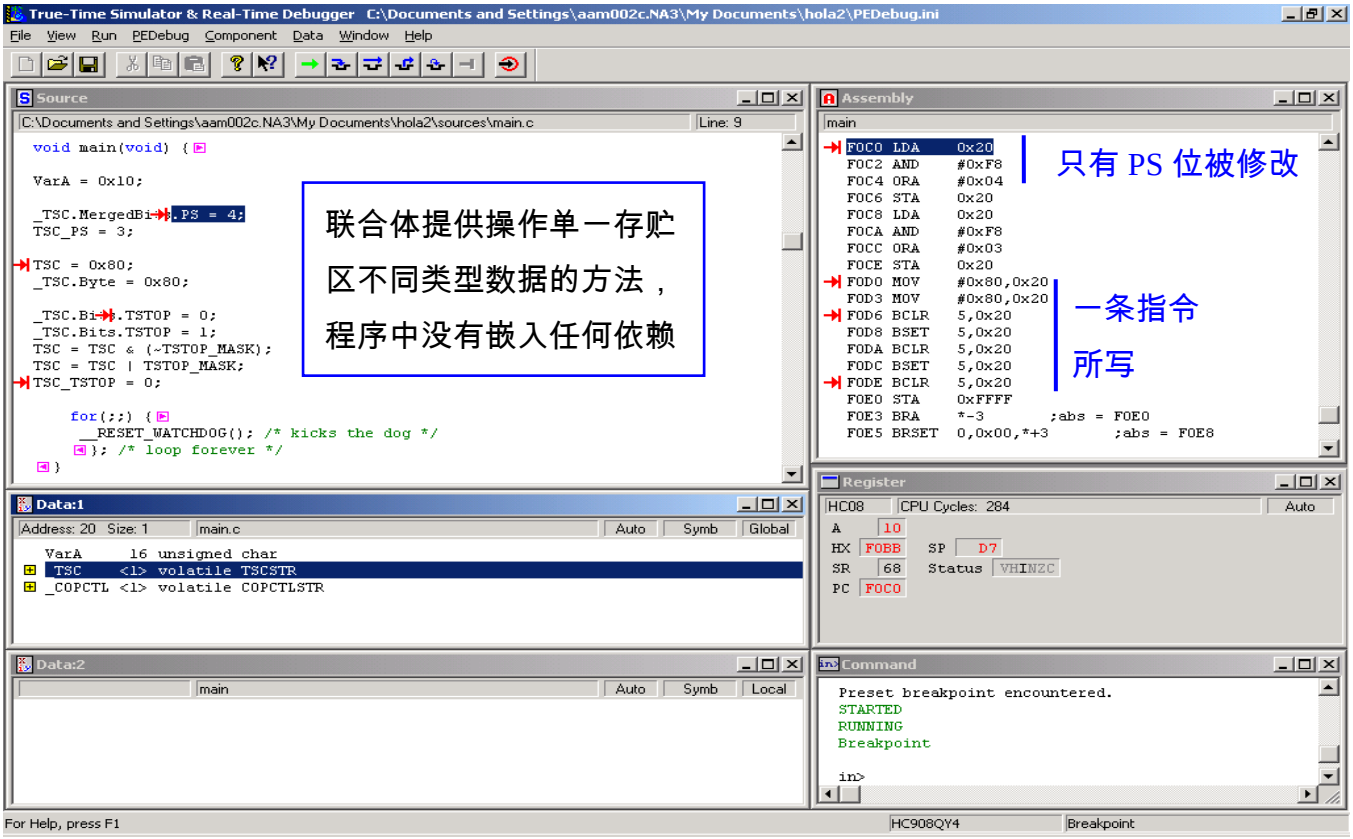
*移位和掩模

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。

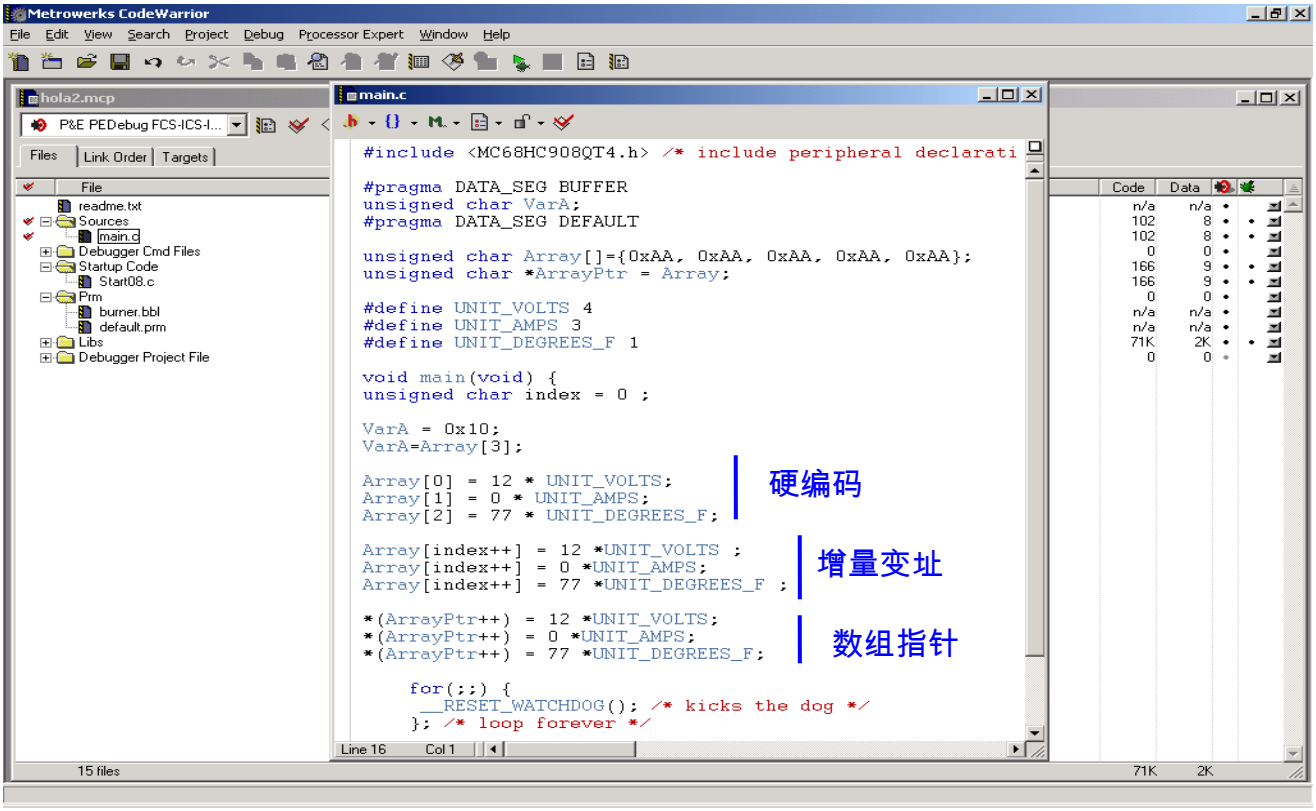
3）数组指针

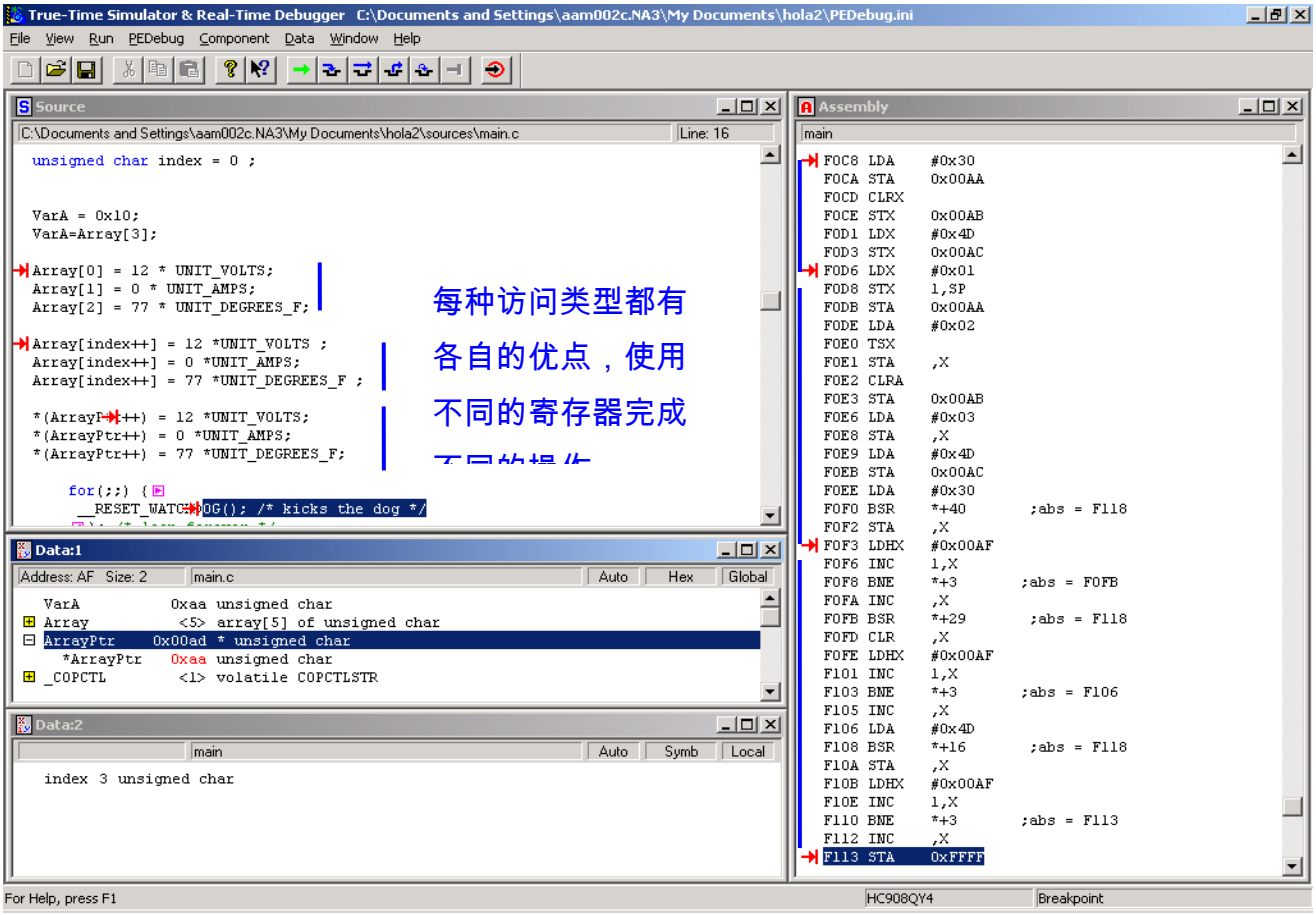
*(ArrayPtr++)=12*UNIT_VOLTS;

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



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；

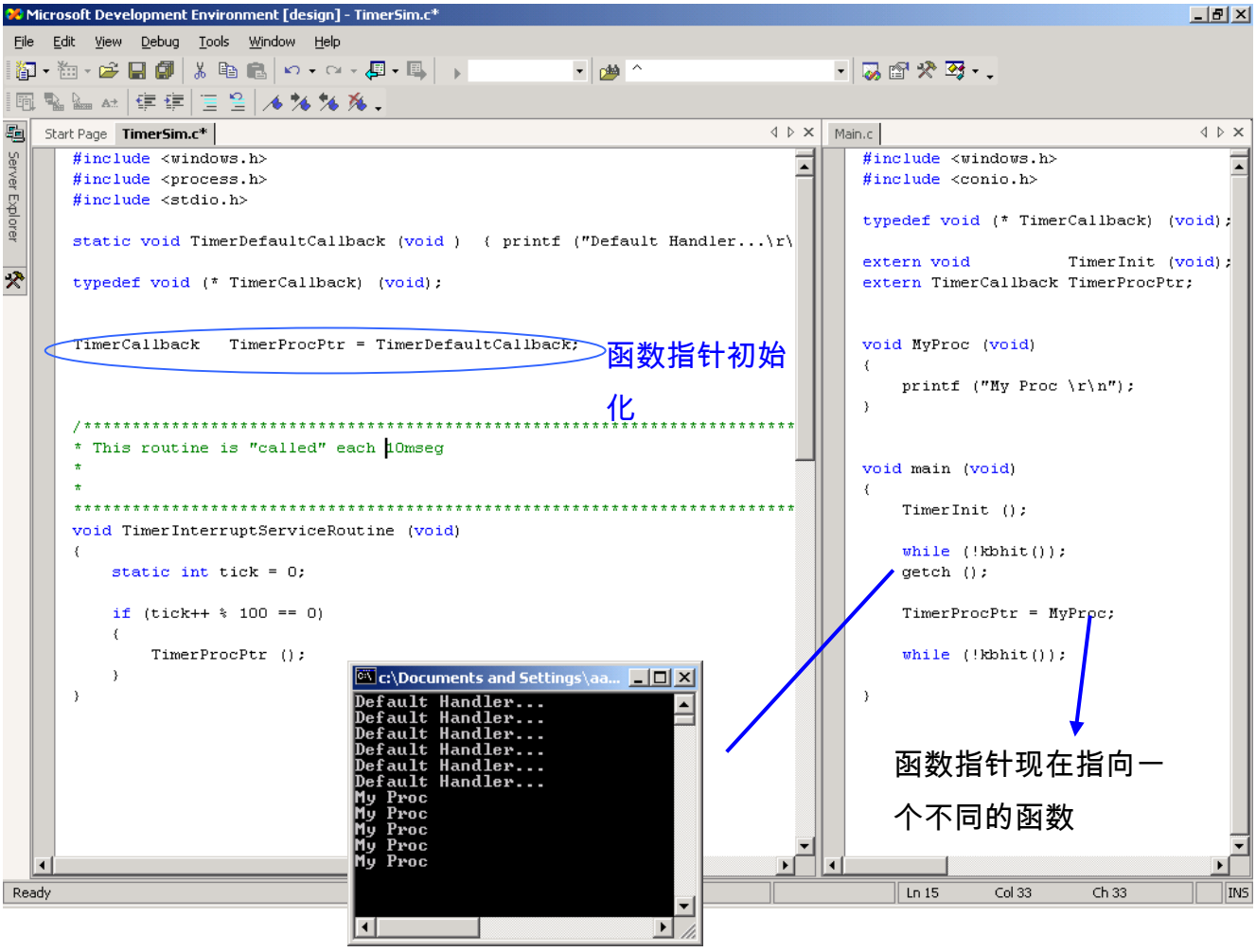
当你想用同一段代码依环境的不同调用不同的函数。

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)

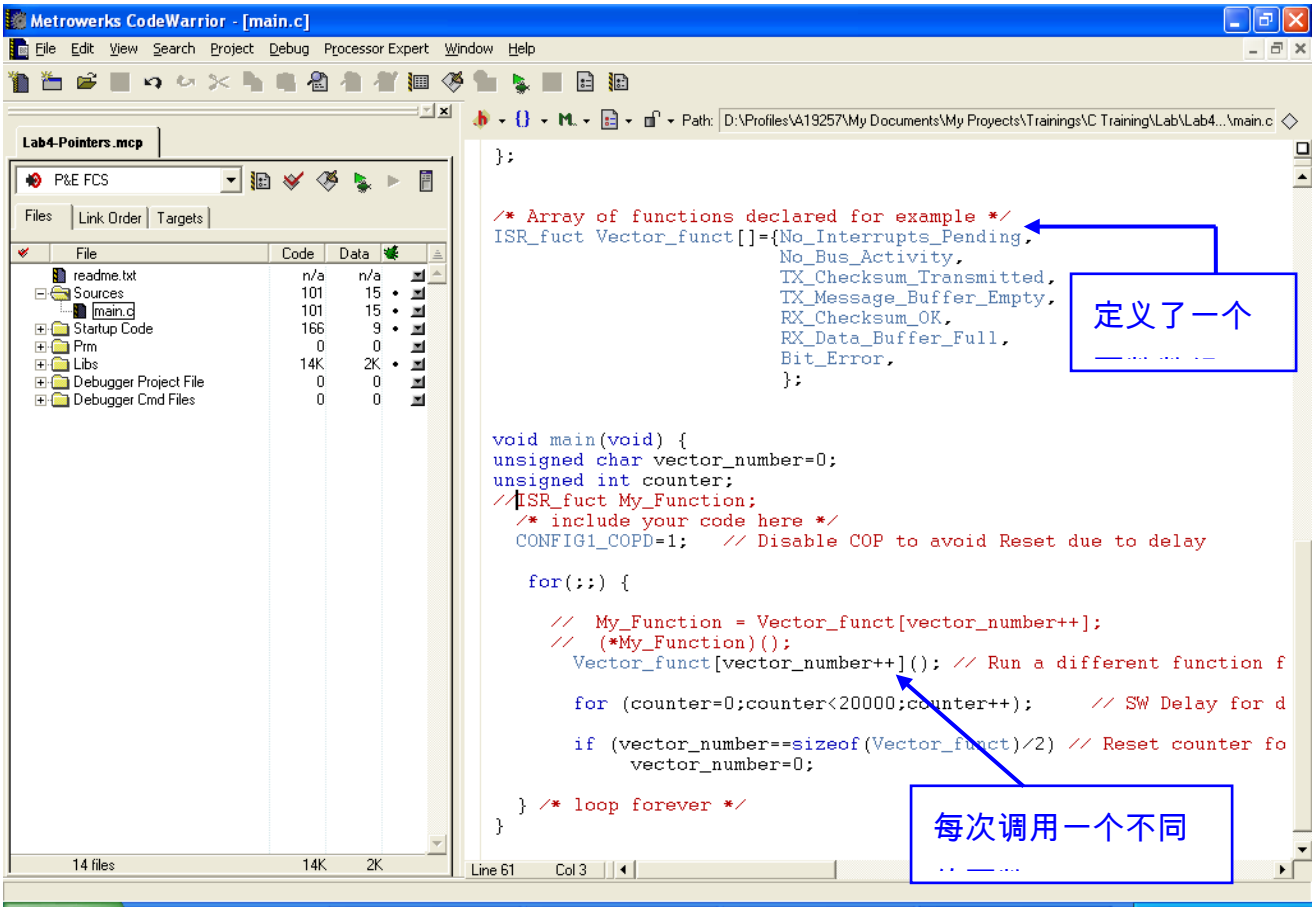
SLCSV	I3	I2	I1	I0	Interrupt Source	Priority
$00	0	0	0	0	No Interrupts Pending	0 (Lowest)
$04	0	0	0	1	No-Bus-Activity	1
$08	0	0	1	0	TX Message Buffer Empty Checksum Transmitted	2
$0C	0	0	1	1	TX Message Buffer Empty	3
$10	0	1	0	0	RX Message Buffer Full Checksum OK	4
$14	0	1	0	1	RX Data Buffer Full No Errors	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。

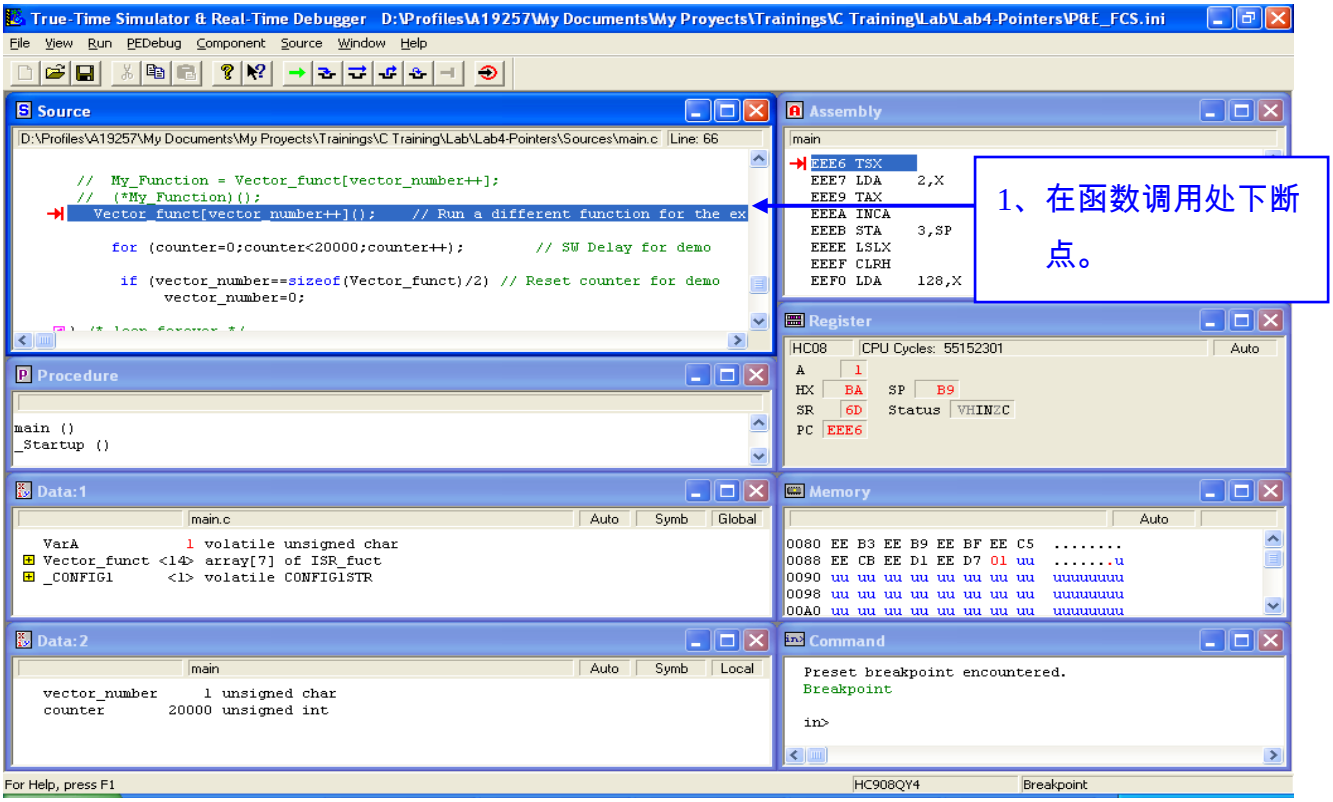
可能的解决方案：

switch 语句；嵌套的if语句；函数指针。

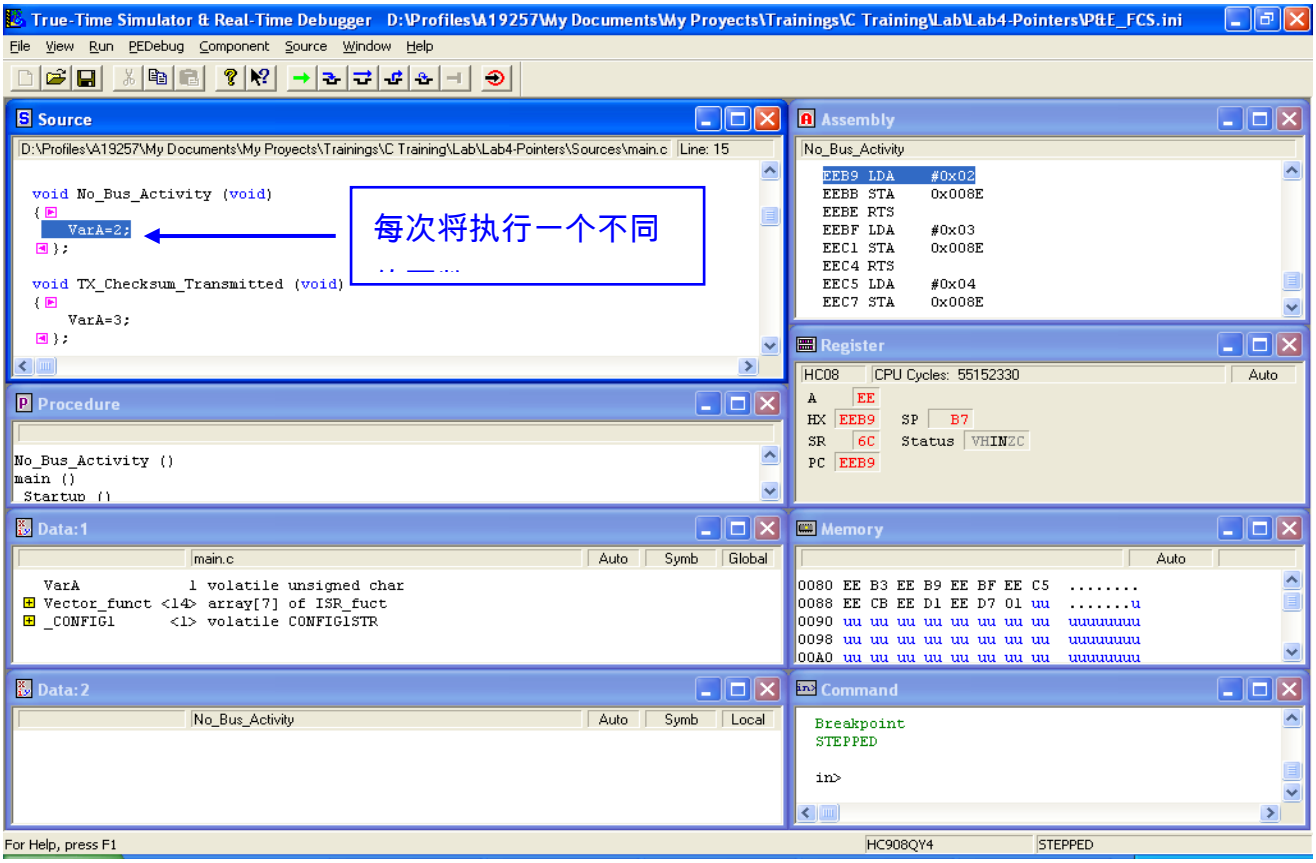
打开文件：Lab4-Pointers.mcp


Metrowerks CodeWarrior - [main.c]
File Edit View Search Project Debug Processor Expert Window Help
Lab4-Pointers.mcp
P&E FCS
Files Link Order Targets
File Code Data
readme.txt n/a n/a
Sources 101 15
main.c 101 15
Startup Code 166 9
Prm 0 0
Libs 14K 2K
Debugger Project File 0 0
Debugger Cmd Files 0 0
14 files 14K 2K
Path: D:\Profiles\A19257\My Documents\My Proyects\Trainings\C Training\Lab\Lab4...\main.c
};
/* Array of functions declared for example */
ISR_fuct Vector_funct[]={No_Interrupts_Pending,
No_Bus_Activity,
TX_Checksum_Transmitted,
TX_Message_Buffer_Empty,
RX_Checksum_OK,
RX_Data_Buffer_Full,
Bit_Error,
};
void main(void) {
unsigned char vector_number=0;
unsigned int counter;
//ISR_fuct My_Function;
/* include your code here */
CONFIG1_COPD=1; // Disable COP to avoid Reset due to delay
for(;;) {
// My_Function = Vector_funct[vector_number++];
// (*My_Function)();
Vector_funct[vector_number++](); // Run a different function f
for (counter=0;counter<20000;counter++); // SW Delay for d
if (vector_number==sizeof(Vector_funct)/2) // Reset counter fo
vector_number=0;
} /* loop forever */
}
Line 61 Col 3
定义了一个
每次调用一个不同

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。
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）：

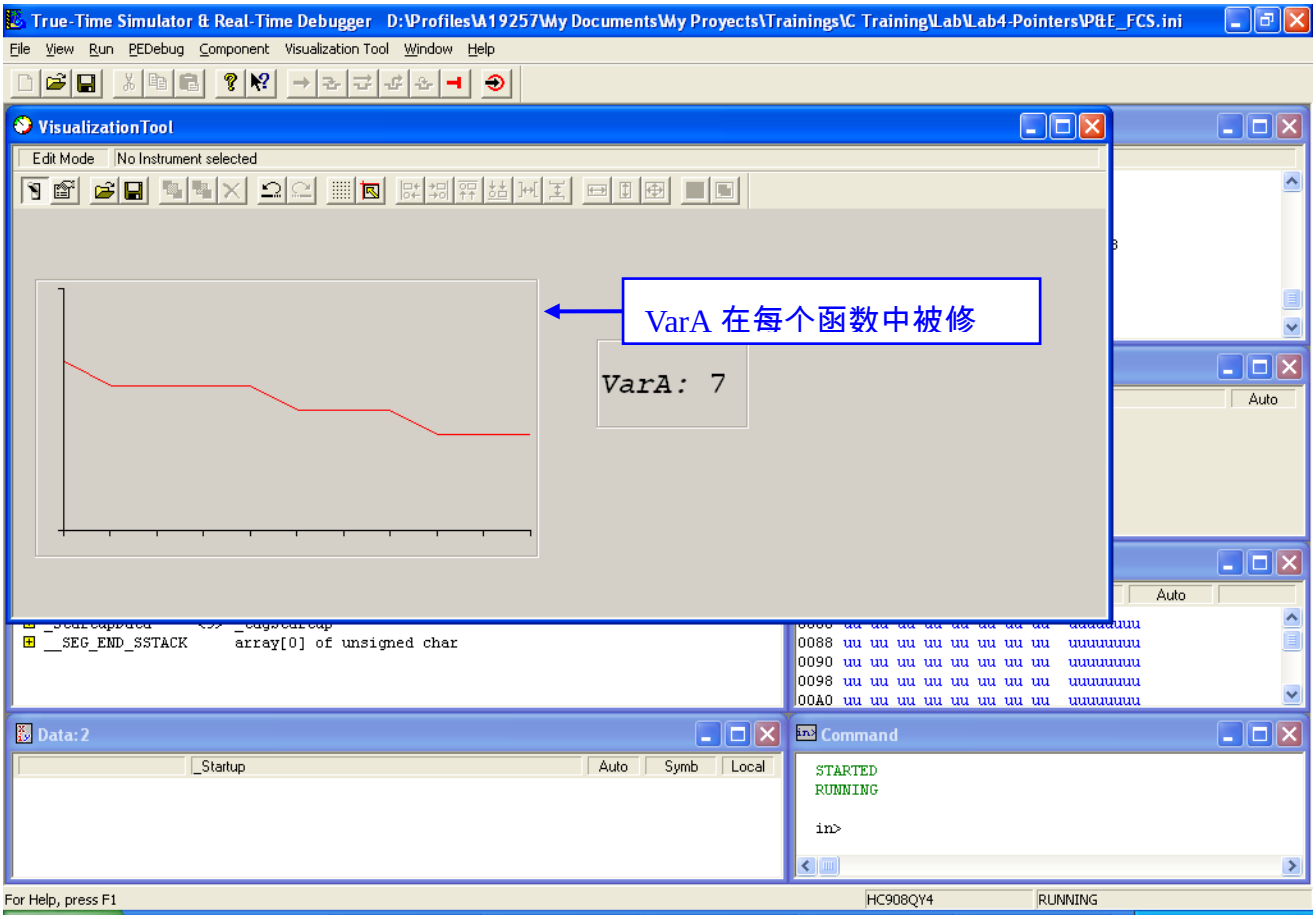


调试（3）：

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
```

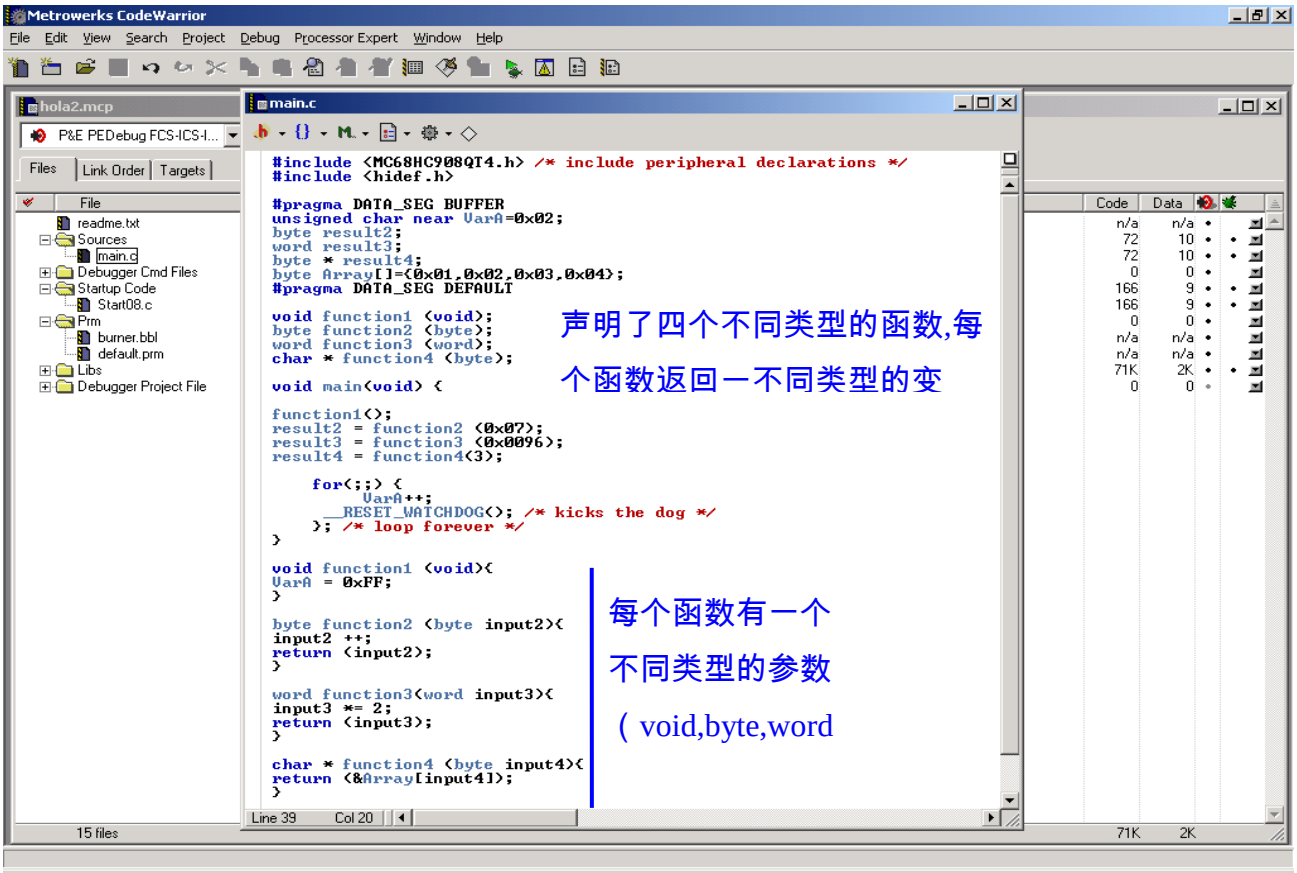
3．6．3 HC08 返回值

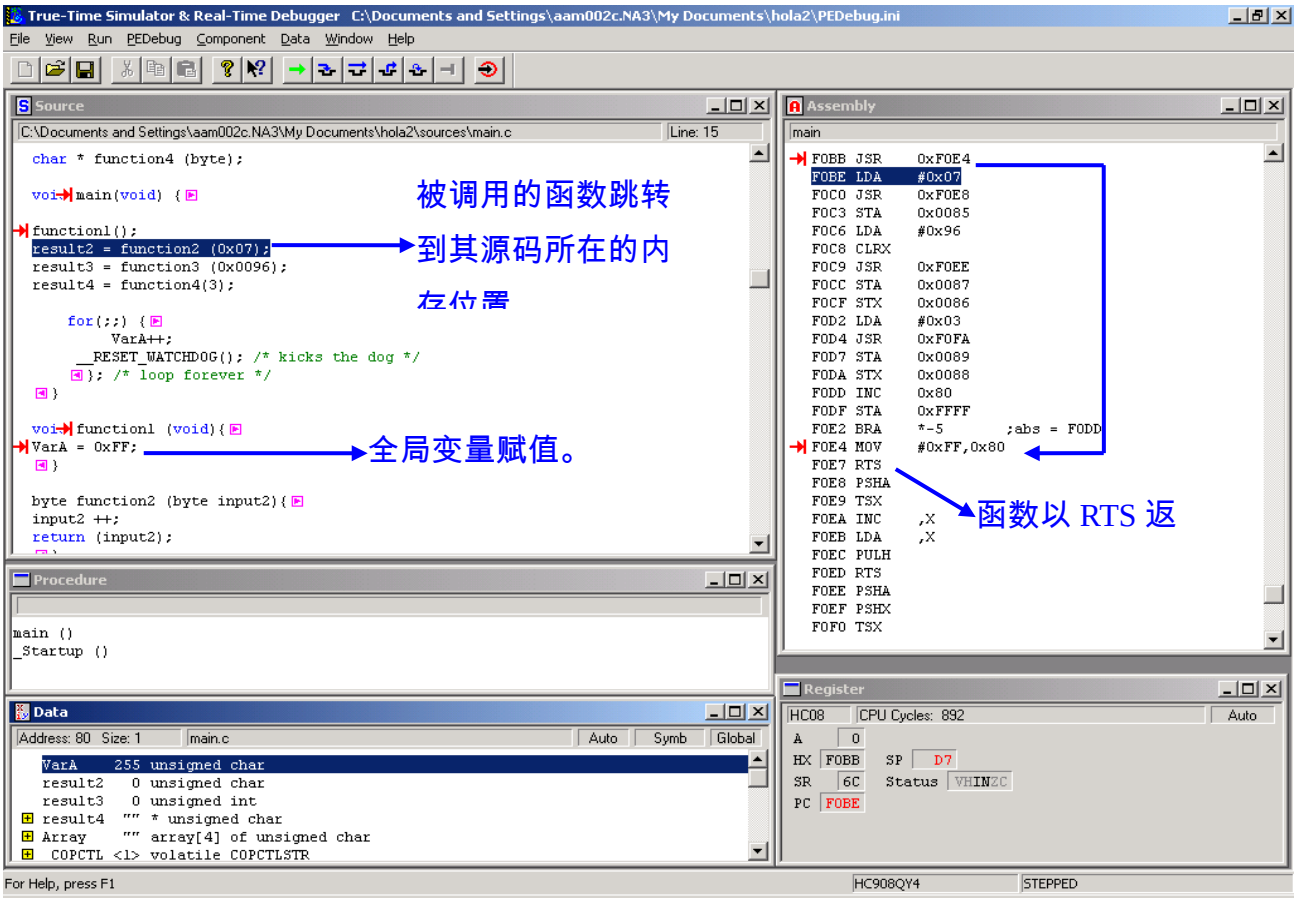
除函数返回一对象大于二字节，函数结果都返回到寄存器中。依据返回类型，使用不同的寄存器。如下表所示：

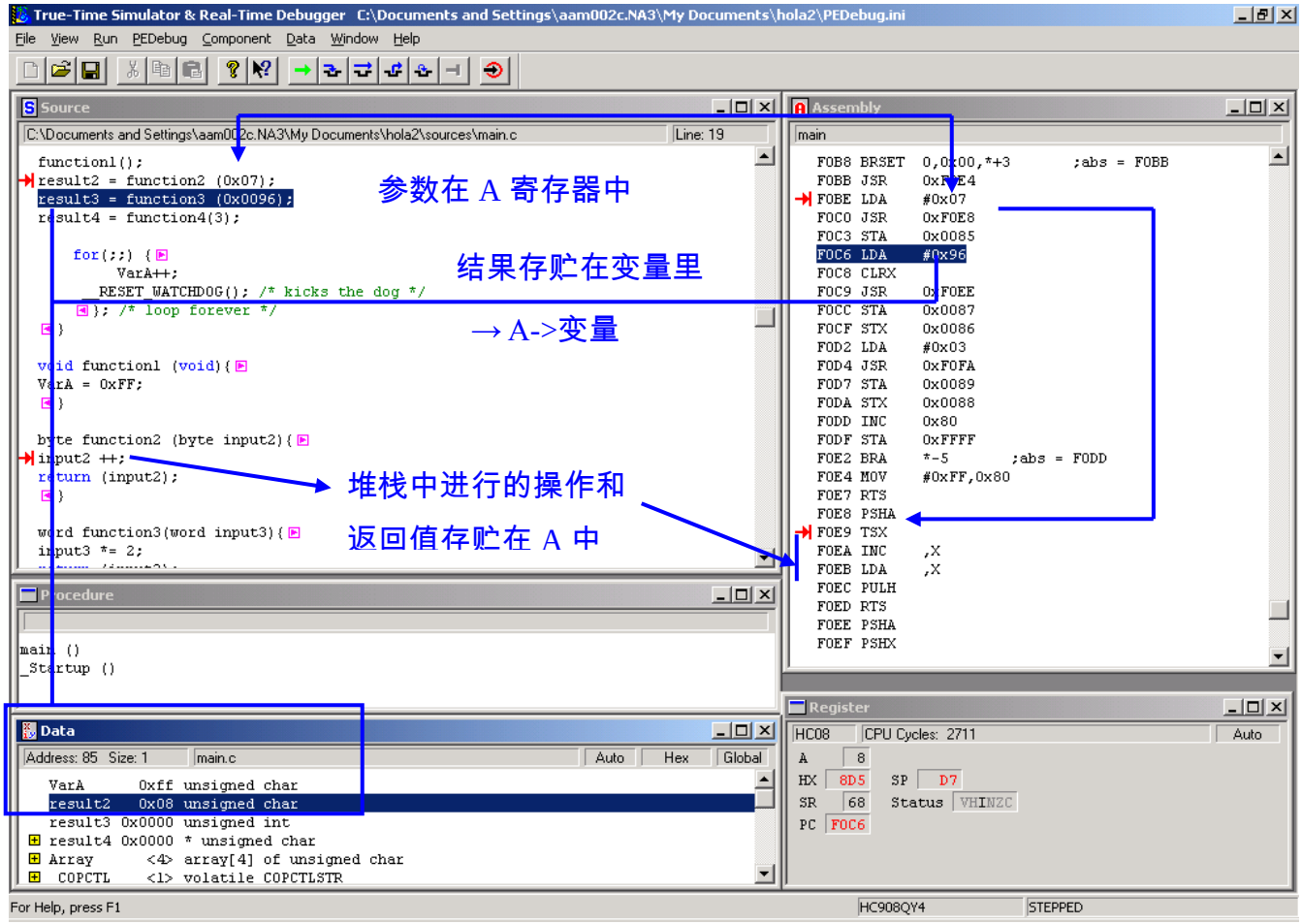
返回类型	寄存器
Char(signed或unsigned)	A
int(signed或unsigned)	X：A
指针/数组	X：A
函数指针	X：A

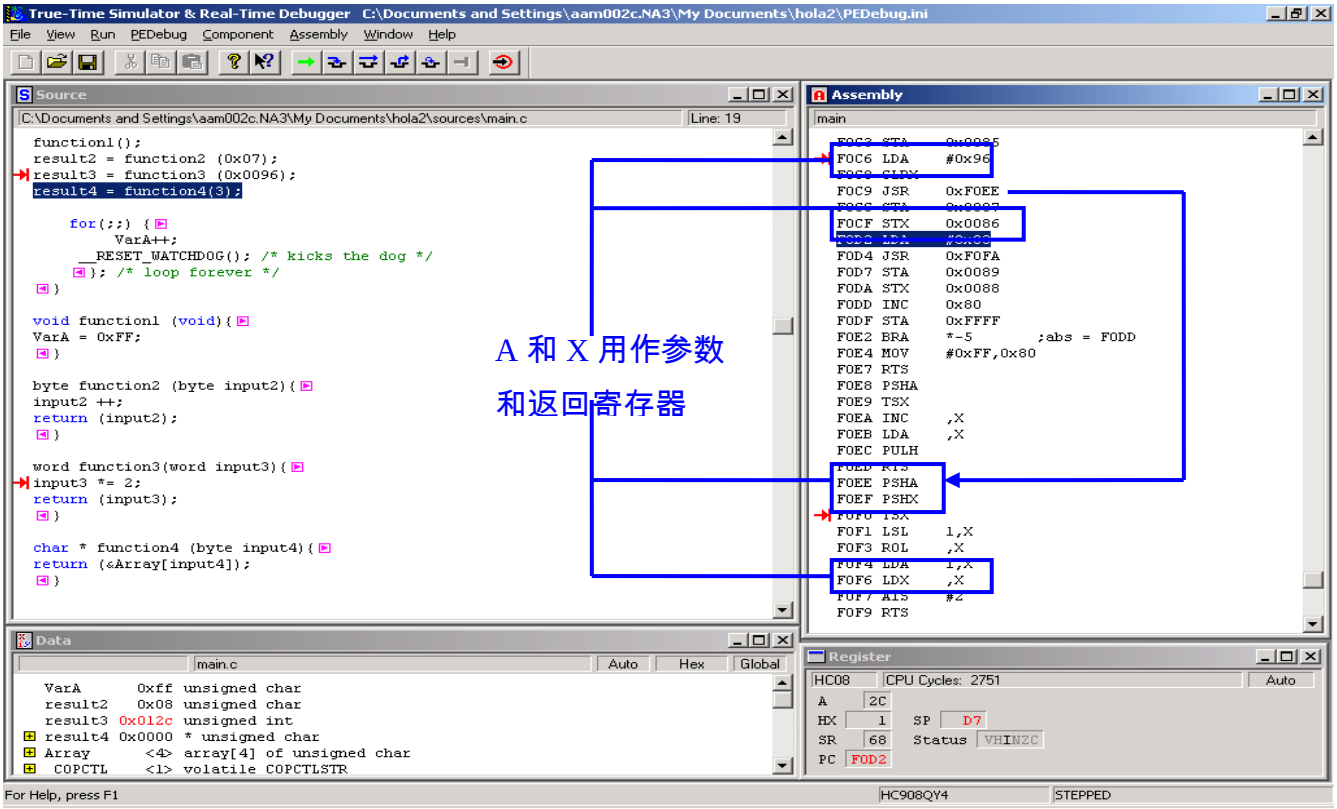
返回大对象：函数返回大于二字节的对象均与一个附加的参数一起调用，它被传到H：X。这个参数是对象应复制到的地址。

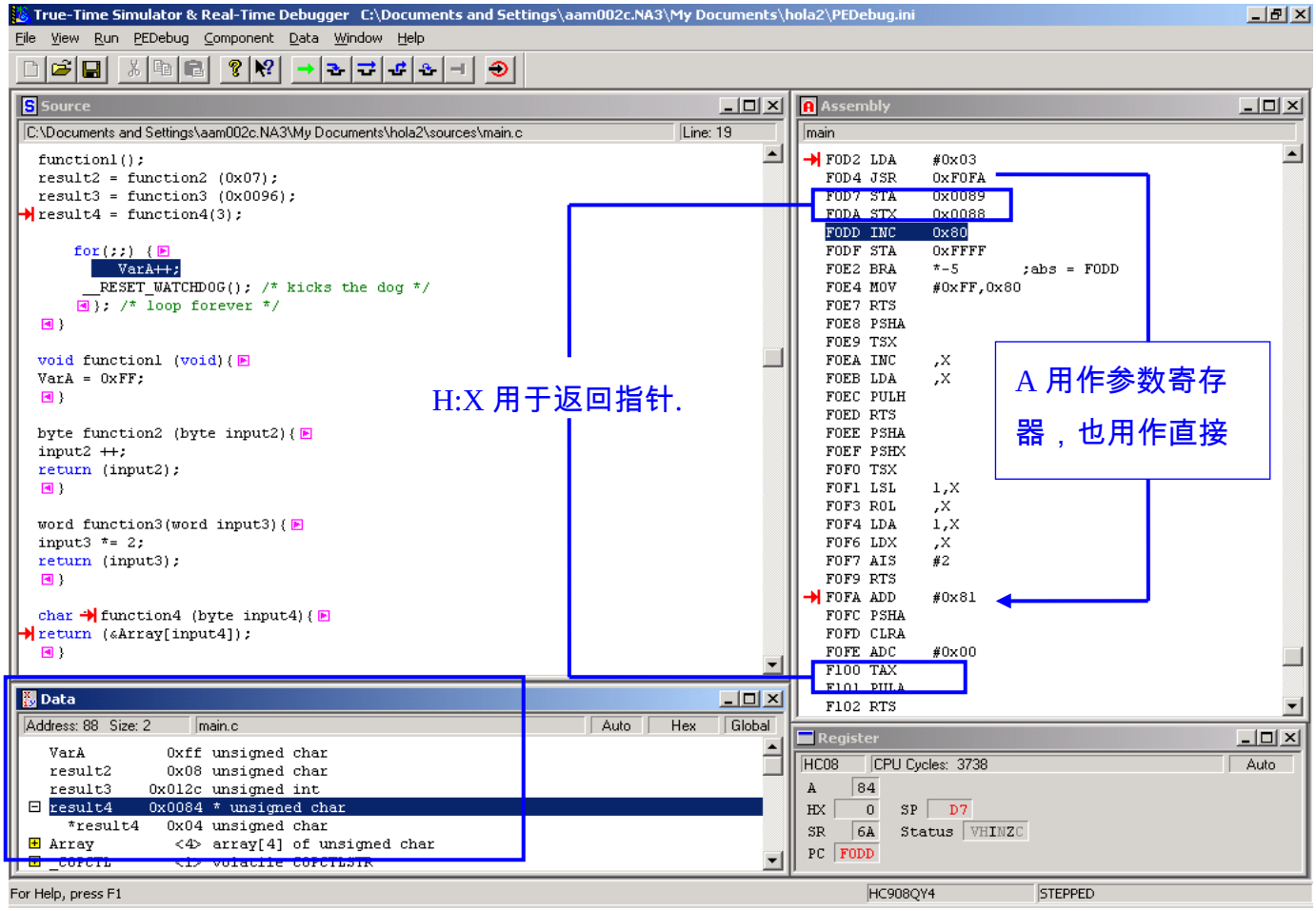
打开文件：Lab5-Arguments.mcp




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);
void main(void) {
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);
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
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 <1> 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]);
}
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
F0ED RTS
F0EE PSHA
F0EF PSHX
F0F0 TSX
F0F1 LSL 1,X
F0F3 ROL ,X
F0F4 LDA 1,X
F0F6 LDX ,X
F0F7 AIS #2
F0F9 RTS
A 和 X 用作参数
和返回寄存器
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。

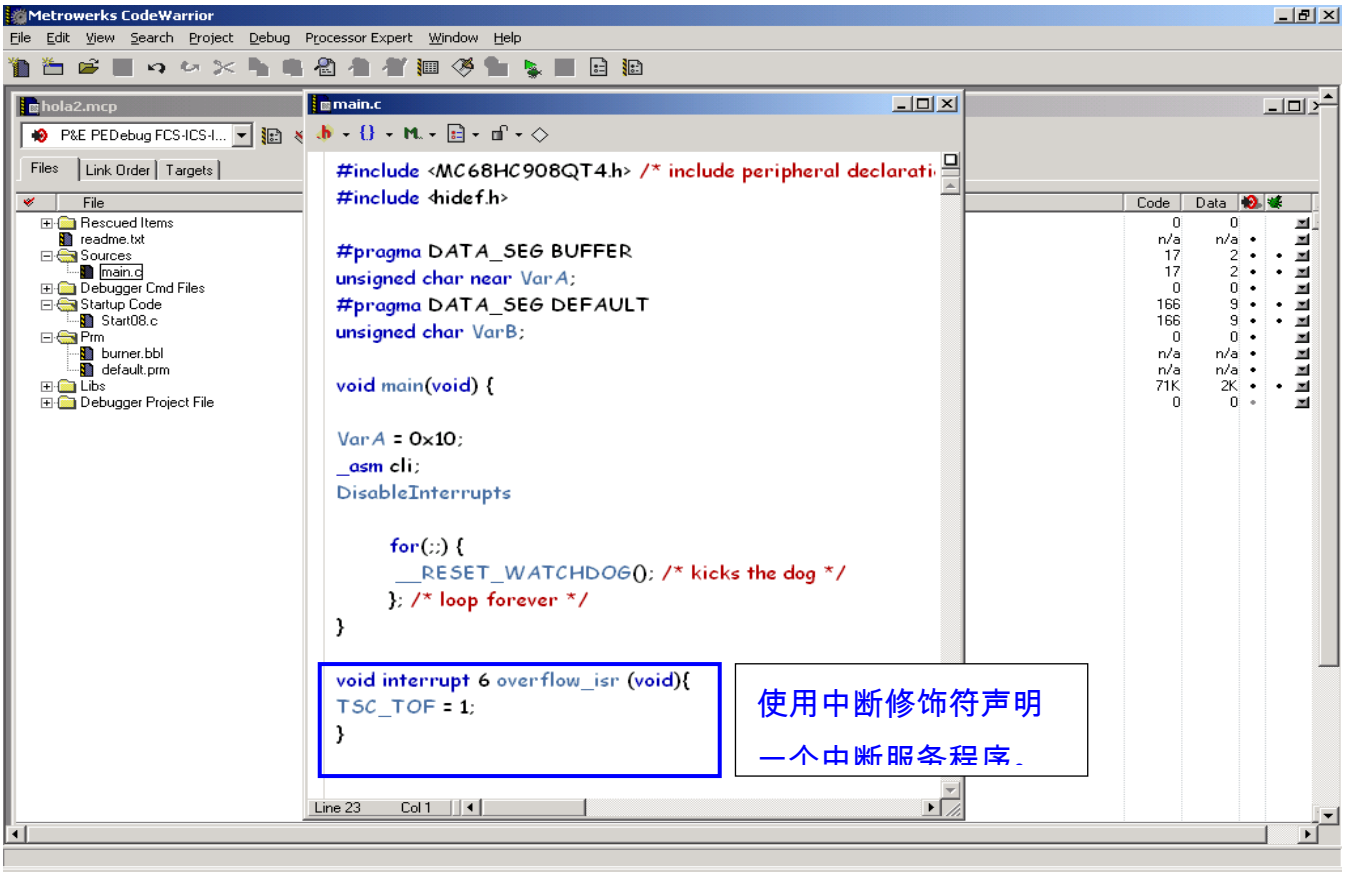
interrupt 17 void TBM_ISR (void){
/* Timebase Module Handler*/
}

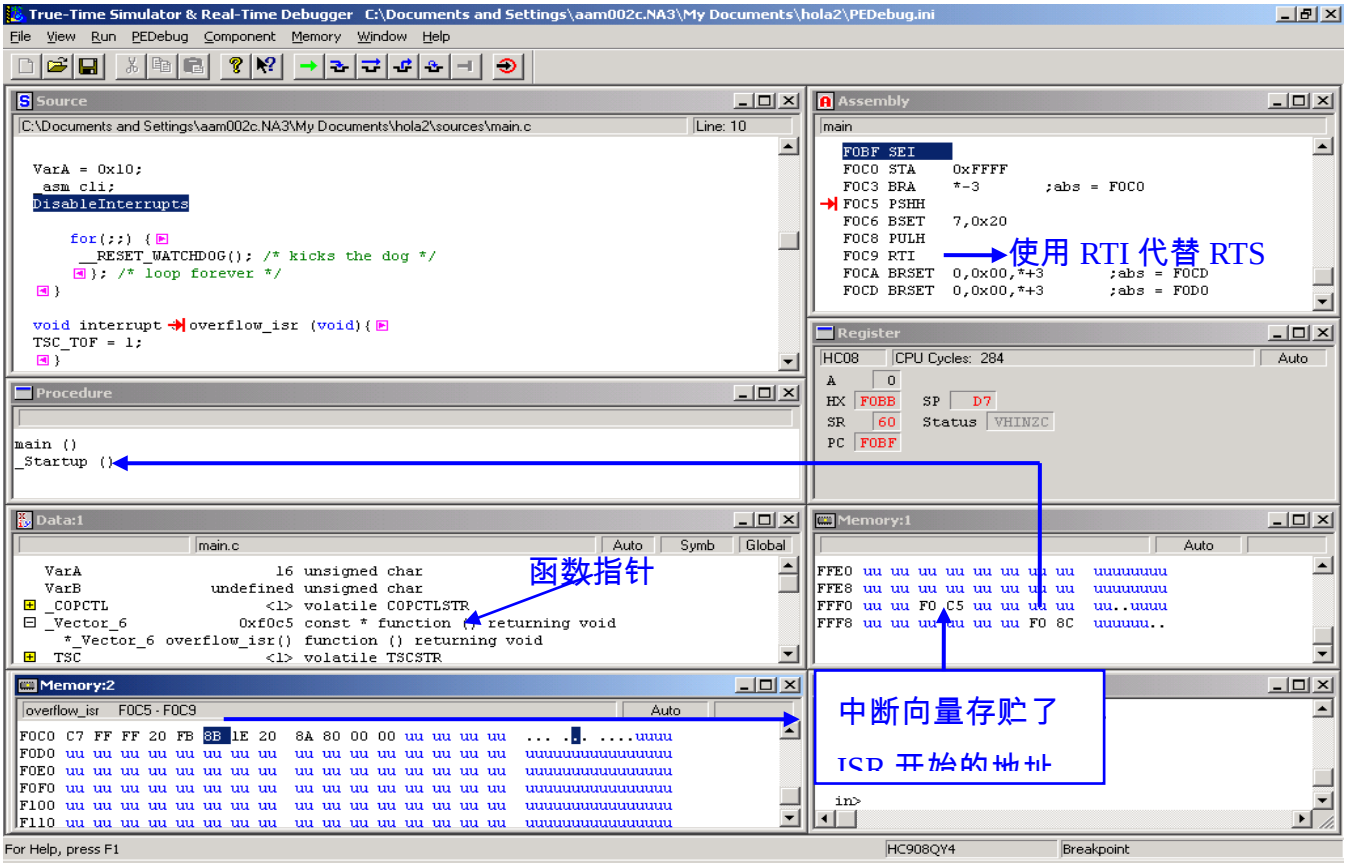
你应查HC08 手册，RESET和软中断向处在最高和相同的优先级。这两个是HC08 的向量 0，但CodeWarrior不得不给每个向量不同的号，因此第一个，RESET，将是向量 0，然后SWI，向量 1，依次类推……直到最后TBM为向量 17，以GP32 为例。

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。

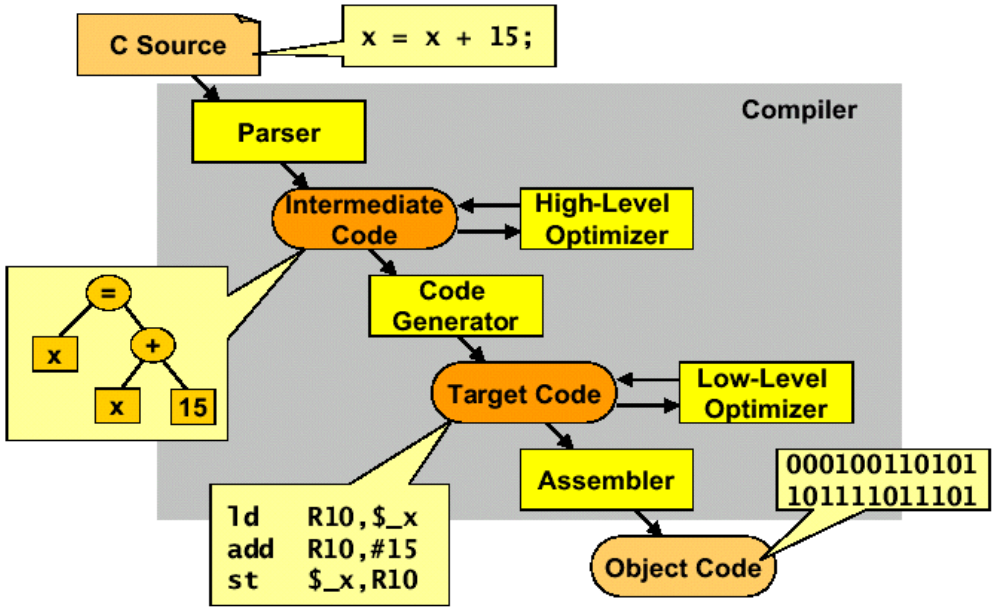
重入代码:无全局变量用于存贮中间结果。由于结构和或缺少栈操作（如HC05、ST7……）并不是所有的目标系统有重入代码。HC08 是允许重入代码的一个好目标系统。

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。

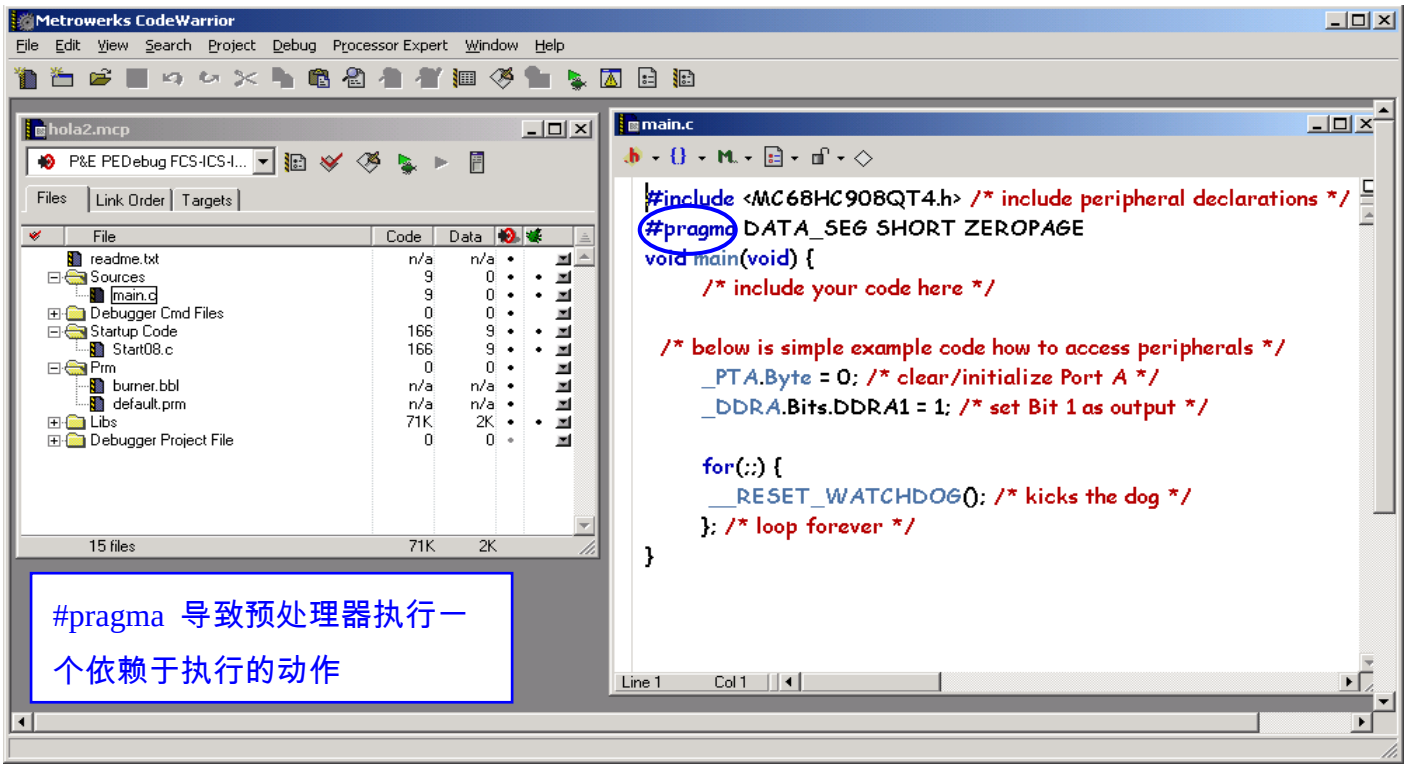
为让连接器知道我们的特定设备或目标的存贮器分布，我们不得不指定它。定义的代码的定位以及内存寻址的数据段由连接器的参数文件控制。这个文件由后缀 ".prm"识别。连接器的参数文件是一个ASCII文本件。对每个程序你得写一个这样的文件。它包含指导如何连接的命令。

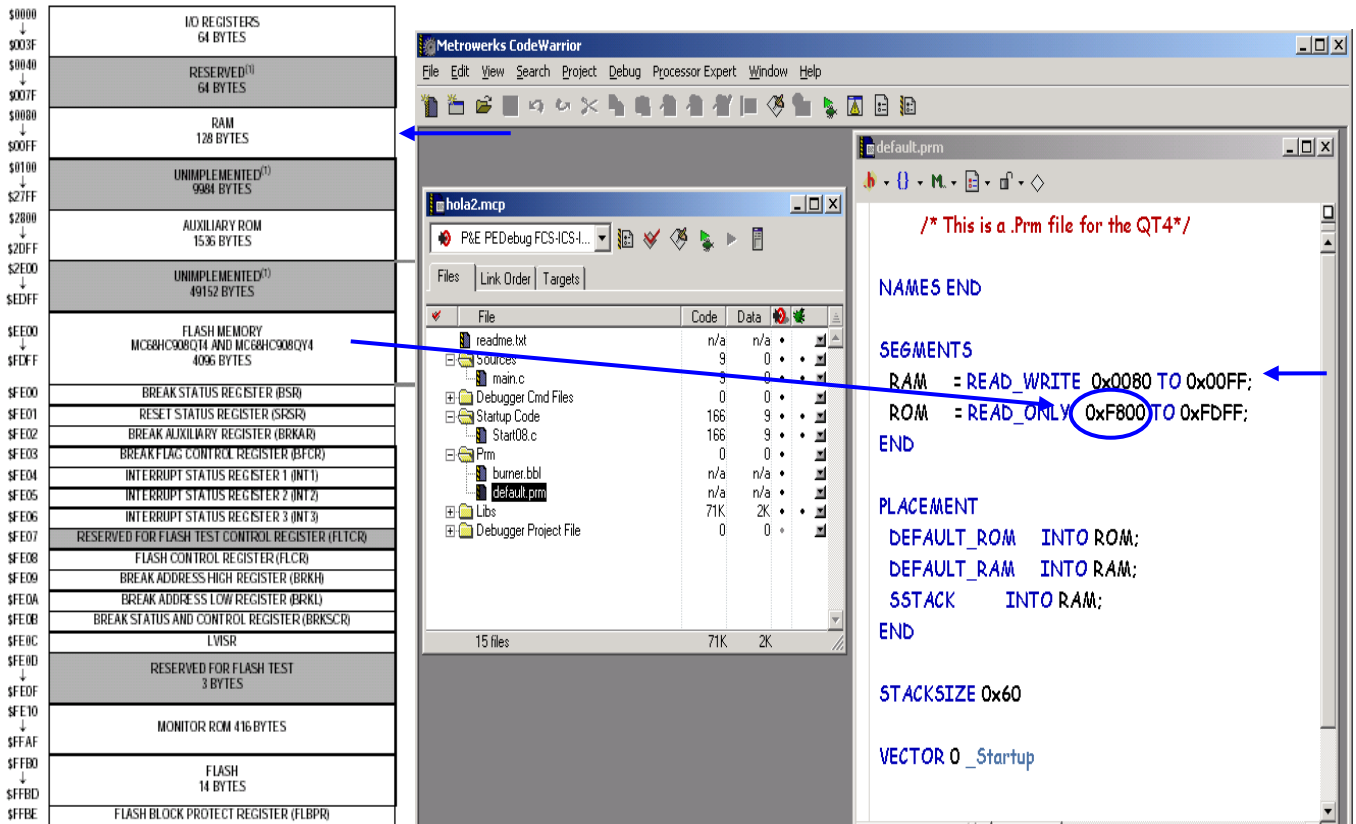
这个文件中，SECTIONS命令块用于定义存贮器的物理区域。在SECTIONS命令块中，每个单独的物理存贮器段用一个名字、一个属性和一个地址范围描述。

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。

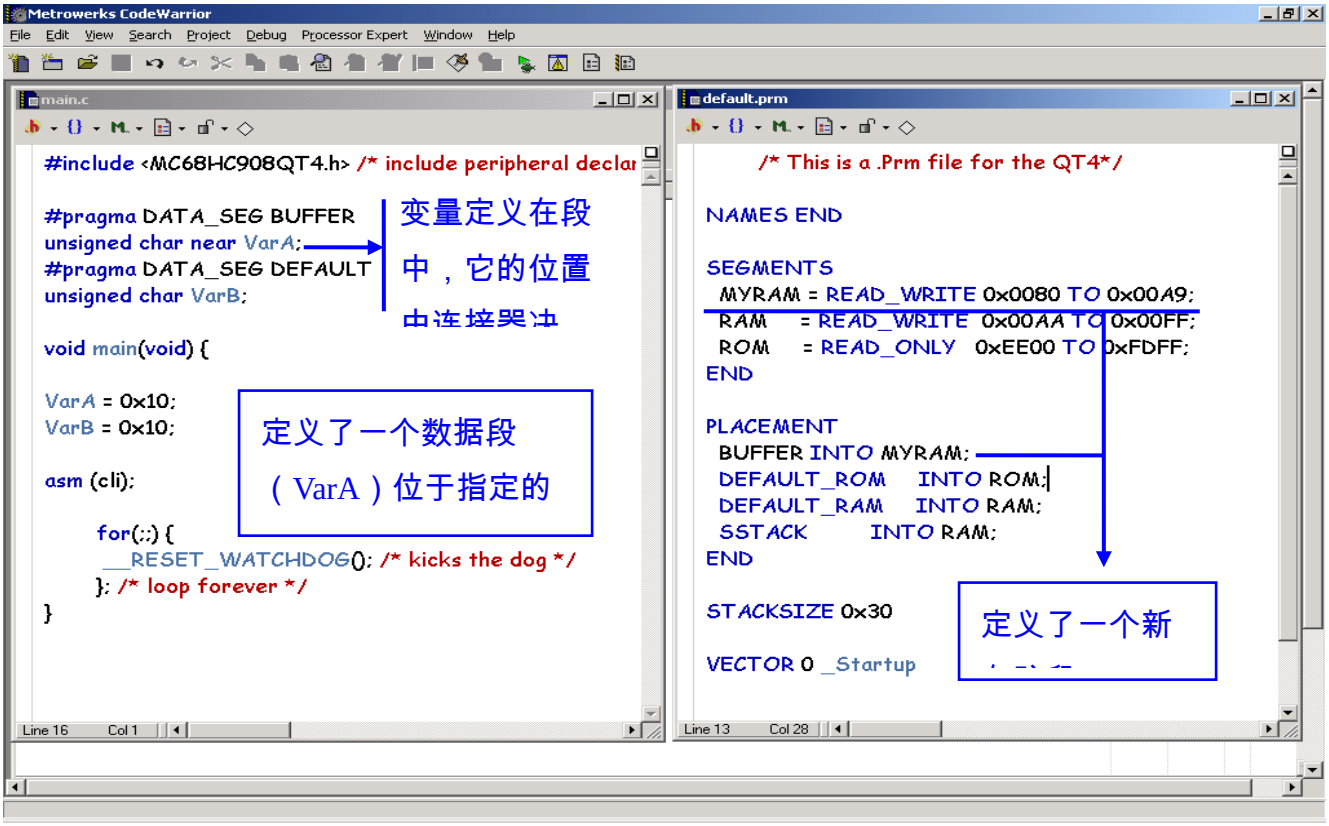
只有变量明确地定义在直接页，编译器才能利用直接寻址模式的效率高优点。ANSI无标准方法做这些，编译器通常提供不同的解决方案。CodeWarrior 使用#pragma声明。

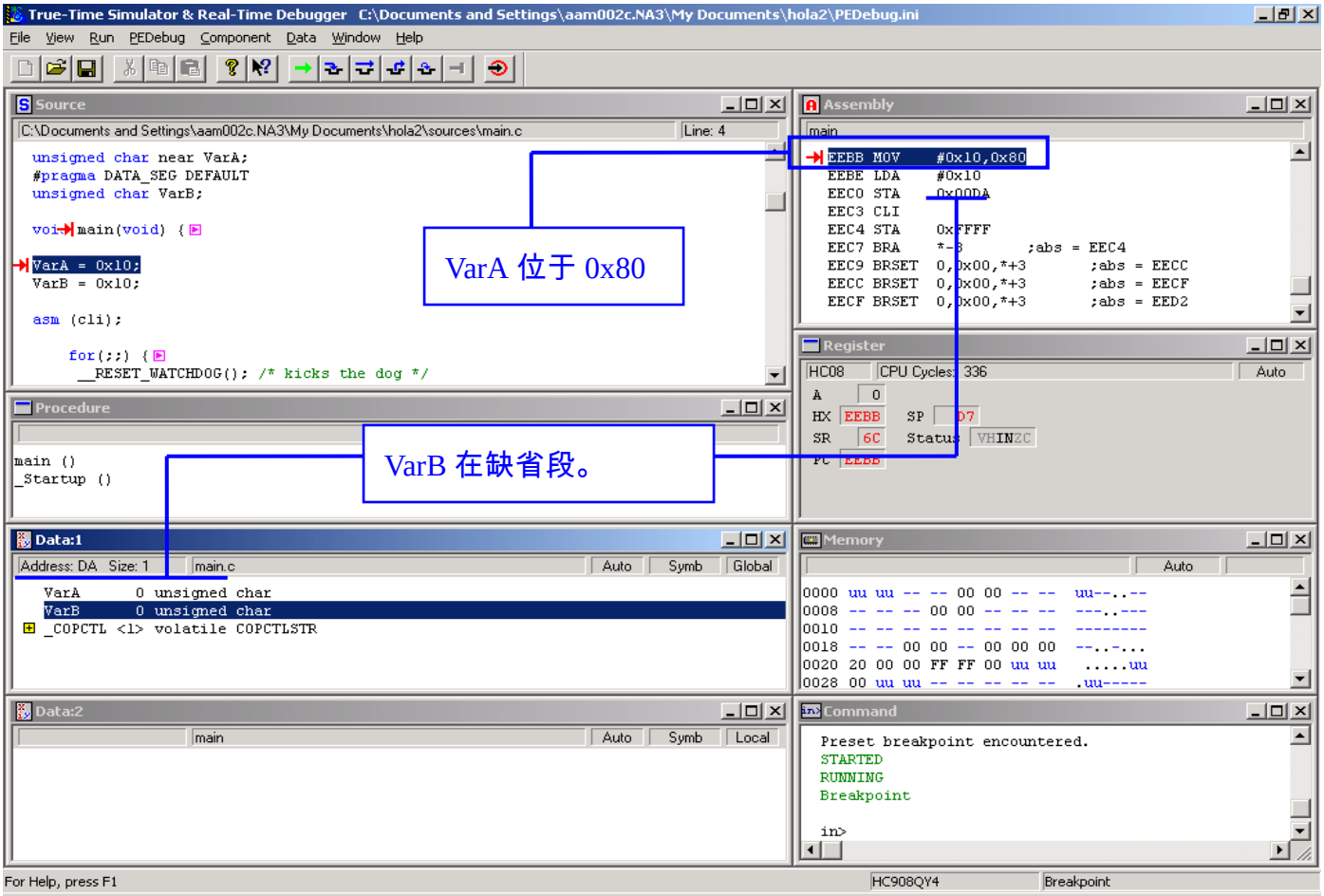
PRAGMA是一个编译器指示。你设置pragma为需要的状态后，那一点以后的所有代码用那个设置编译，直到你改变设置或到达文件的末尾。在每个文件的开头，编译器恢复到工程的设置或缺省设置。

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。

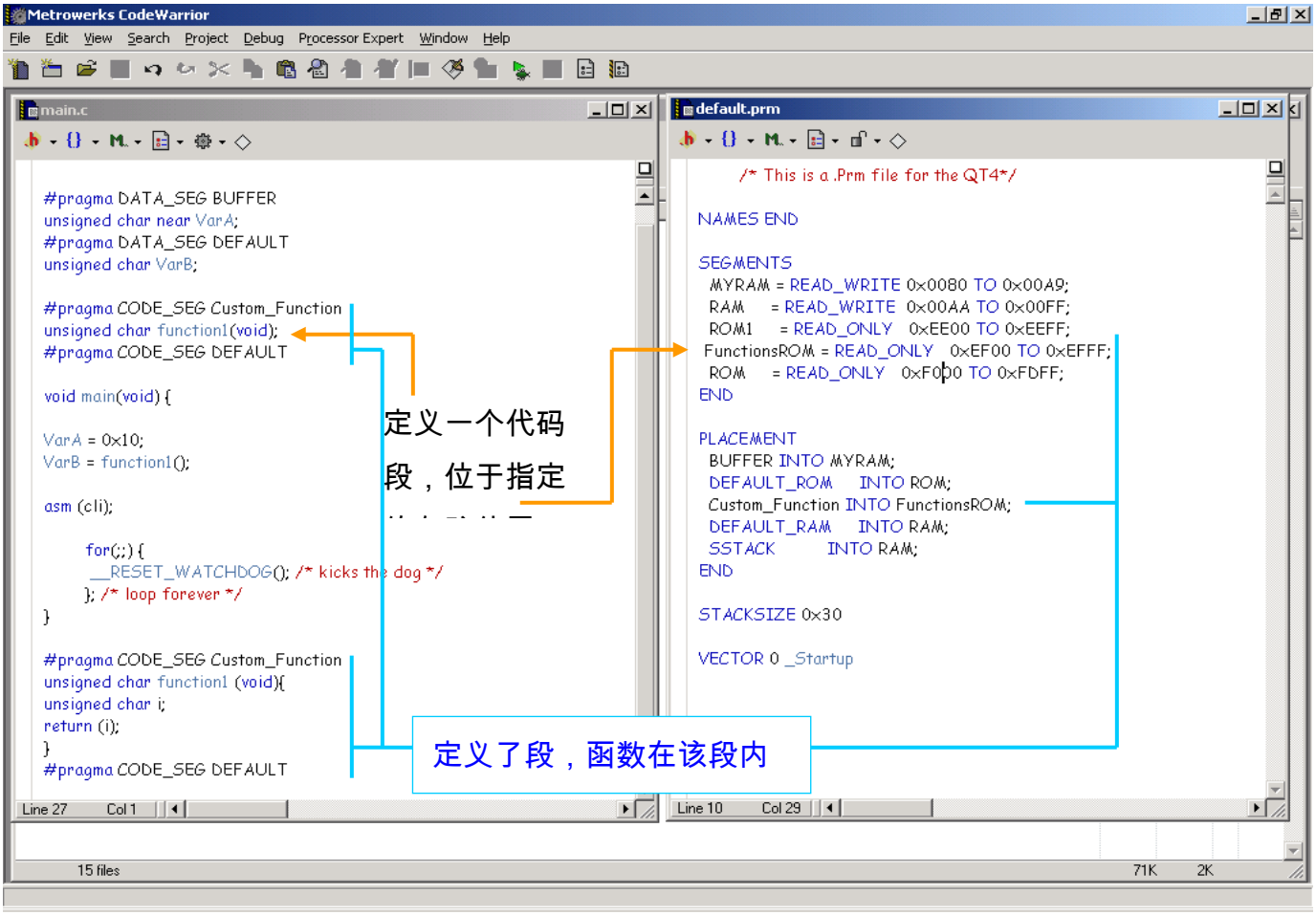
请看下列例子：

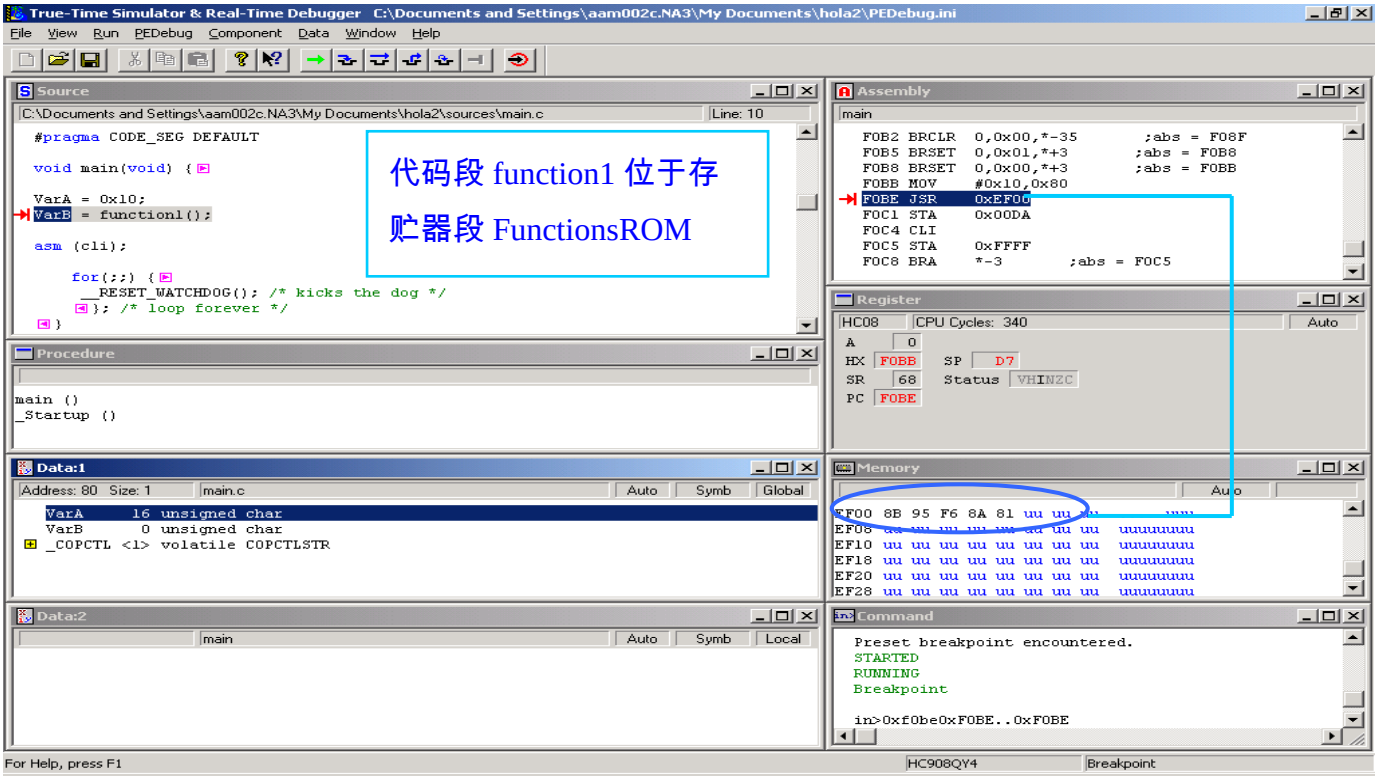
打开文件Lab7-DataSeg.mcp：


Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help
main.c
#include <MC68HC908QT4.h> /* include peripheral declar
#pragma DATA_SEG BUFFER
unsigned char near VarA;
#pragma DATA_SEG DEFAULT
unsigned char VarB;
变量定义在段
中，它的位置
由连接器决
void main(void) {
VarA = 0x10;
VarB = 0x10;
asm (cli);
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
定义了一个数据段
（VarA）位于指定的
Line 16 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;
ROM = READ_ONLY 0xEE00 TO 0xFDFF;
END
PLACEMENT
BUFFER INTO MYRAM;
DEFAULT_ROM INTO ROM;
DEFAULT_RAM INTO RAM;
SSTACK INTO RAM;
END
STACKSIZE 0x30
VECTOR 0 _Startup
定义了一个新
Line 13 Col 28

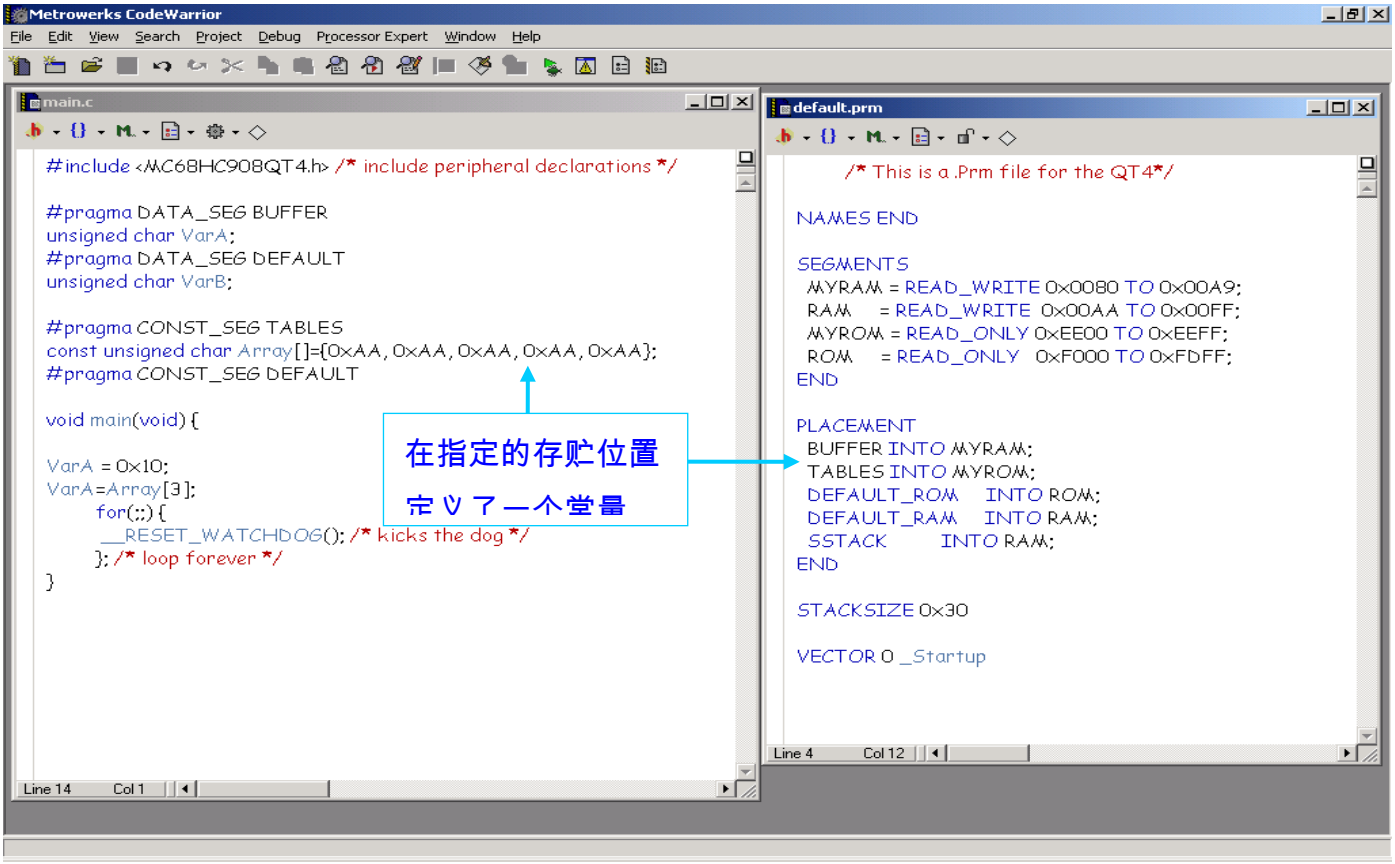


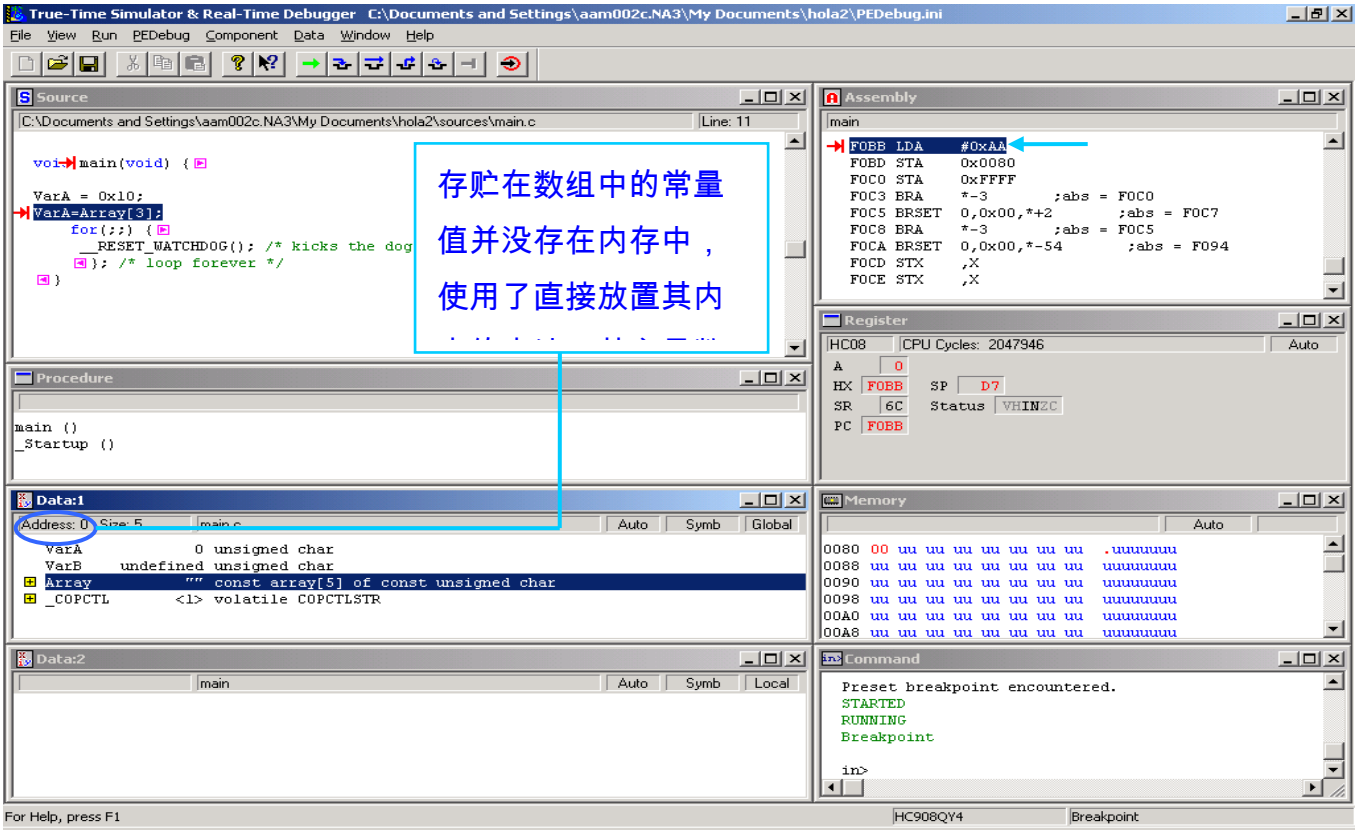
打开文件Lab8-CodeSeg.mcp


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
定义了段，函数在该段内
15 files
71K 2K



打开文件Lab9-ConstSeg.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
unsigned char VarB;
#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];
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
Line 14 Col 1
在指定的存贮位置
定义了一个常量
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



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；

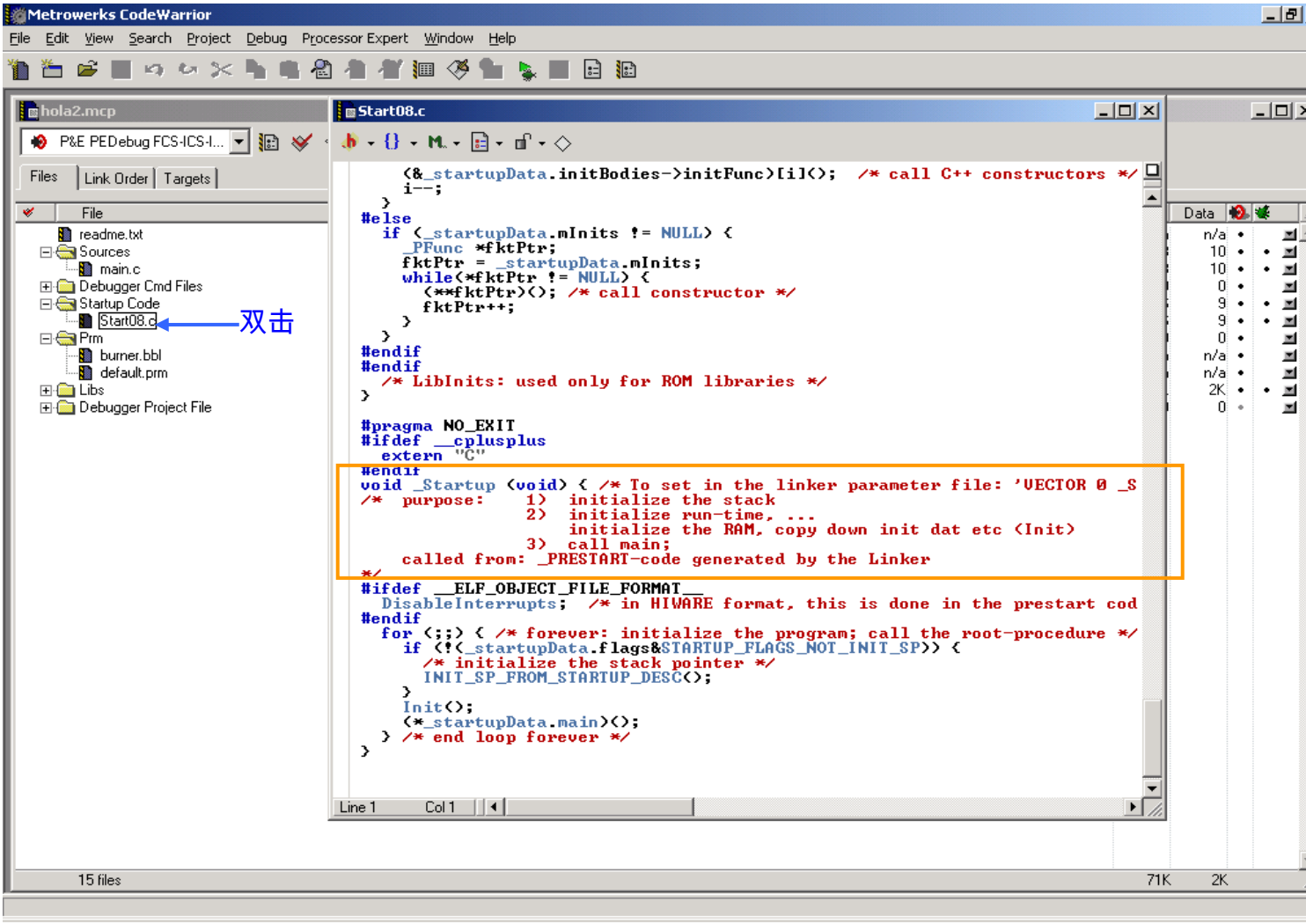
4）为堆栈定位空间以及初始化堆栈；

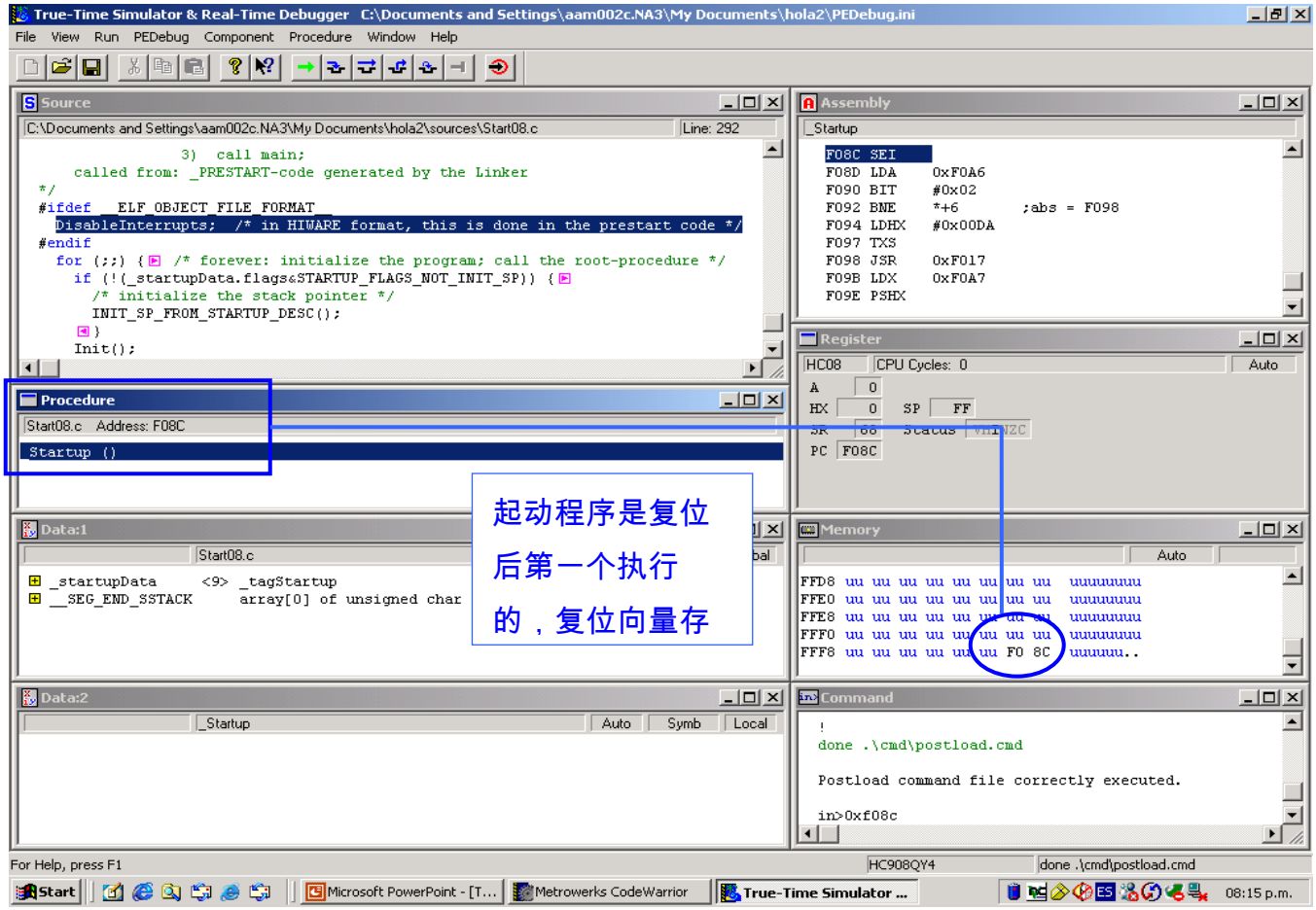
5）创建并初始化堆；

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。


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
双击
Prm
burner.bbl
default.prm
Libs
Debugger Project File
Start08.c
(&_startupData.initBodies->initFunc)[i](); /* call C++ constructors */
i--;
}
#else
if (_startupData.mInits != NULL) {
_PFunc *fktPtr;
fktPtr = _startupData.mInits;
while(*fktPtr != NULL) {
(**fktPtr)(); /* call constructor */
fktPtr++;
}
}
#endif
#endif
/* LibInits: used only for ROM libraries */
}
#pragma NO_EXIT
#ifdef __cplusplus
extern "C"
#endif
void _Startup (void) { /* To set in the linker parameter file: 'VECTOR 0 _S
/* purpose: 1) initialize the stack
2) initialize run-time, ...
initialize the RAM, copy down init dat etc (Init)
3) call main;
called from: _PRESTART-code generated by the Linker
*/
#ifdef __ELF_OBJECT_FILE_FORMAT__
DisableInterrupts; /* in HIWARE format, this is done in the prestart cod
#endif
for (;;) { /* forever: initialize the program; call the root-procedure */
if (!(_startupData.flags&STARTUP_FLAGS_NOT_INIT_SP)) {
/* initialize the stack pointer */
INIT_SP_FROM_STARTUP_DESC();
}
Init();
(*_startupData.main)();
} /* end loop forever */
}
Line 1 Col 1
Data
15 files
71K 2K



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。

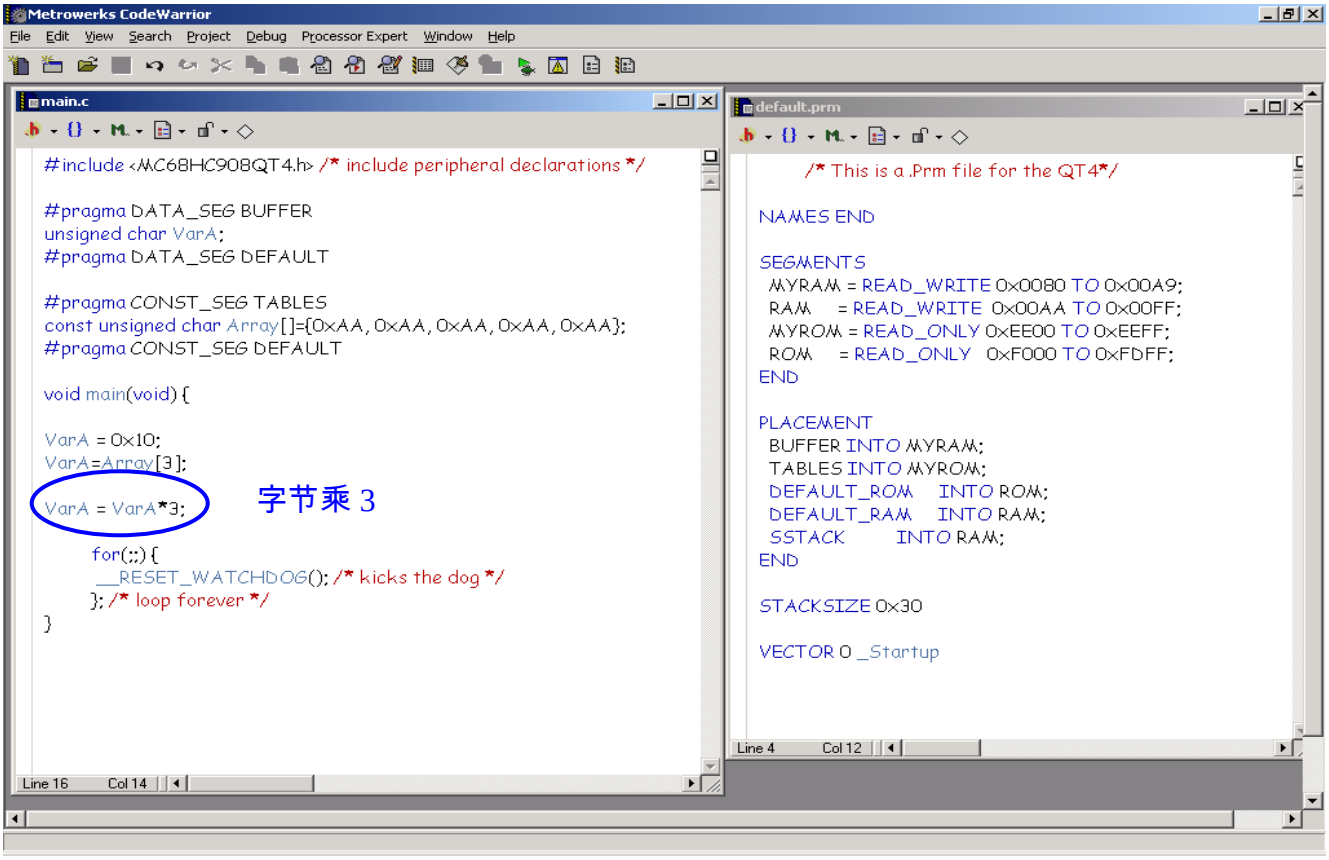
1）强度减弱

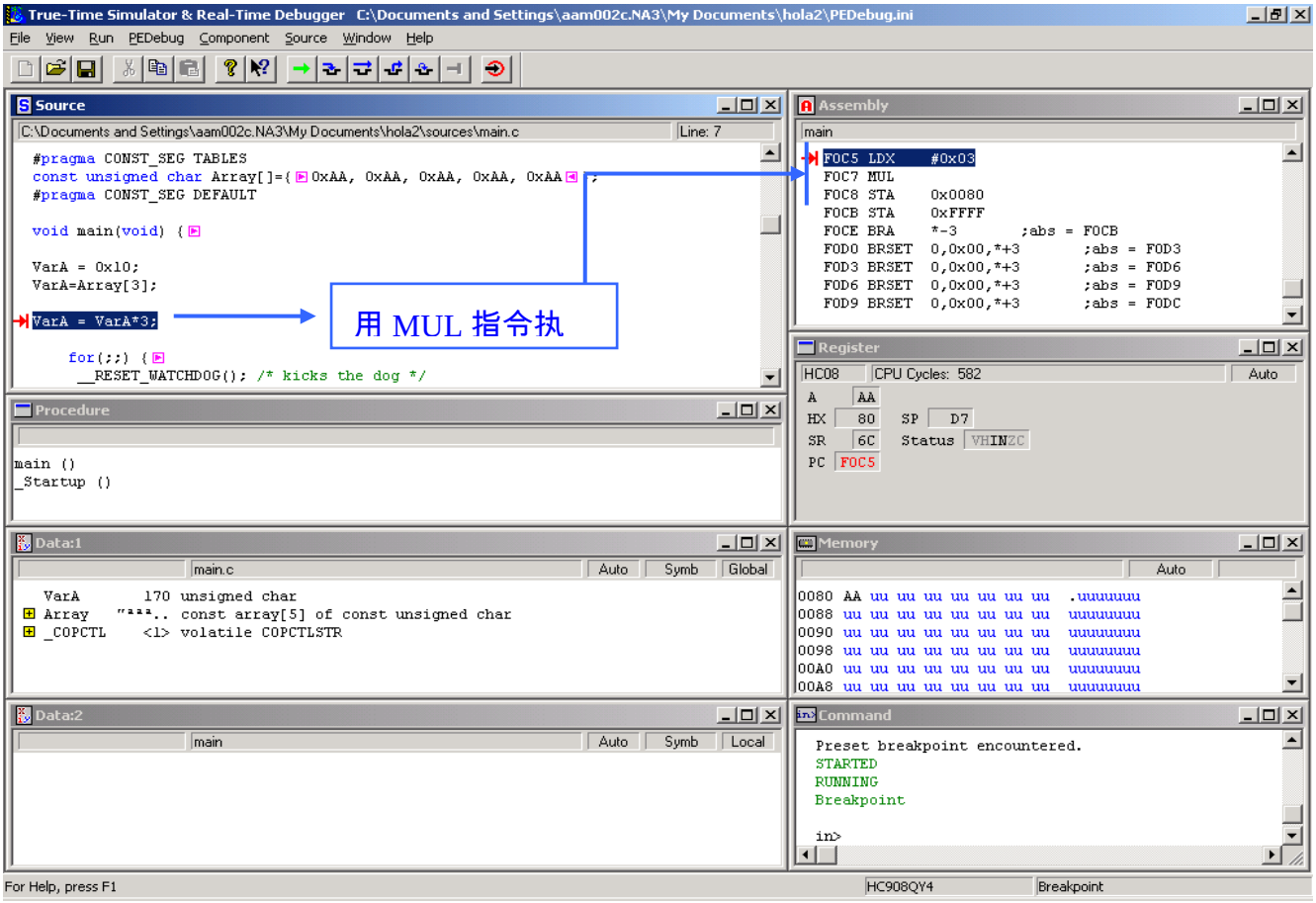
“Strength Reduction”（强度减弱）是一种优化，力争用开销小的操作代替开销大的操作，代价因素是执行时间或代码大小。

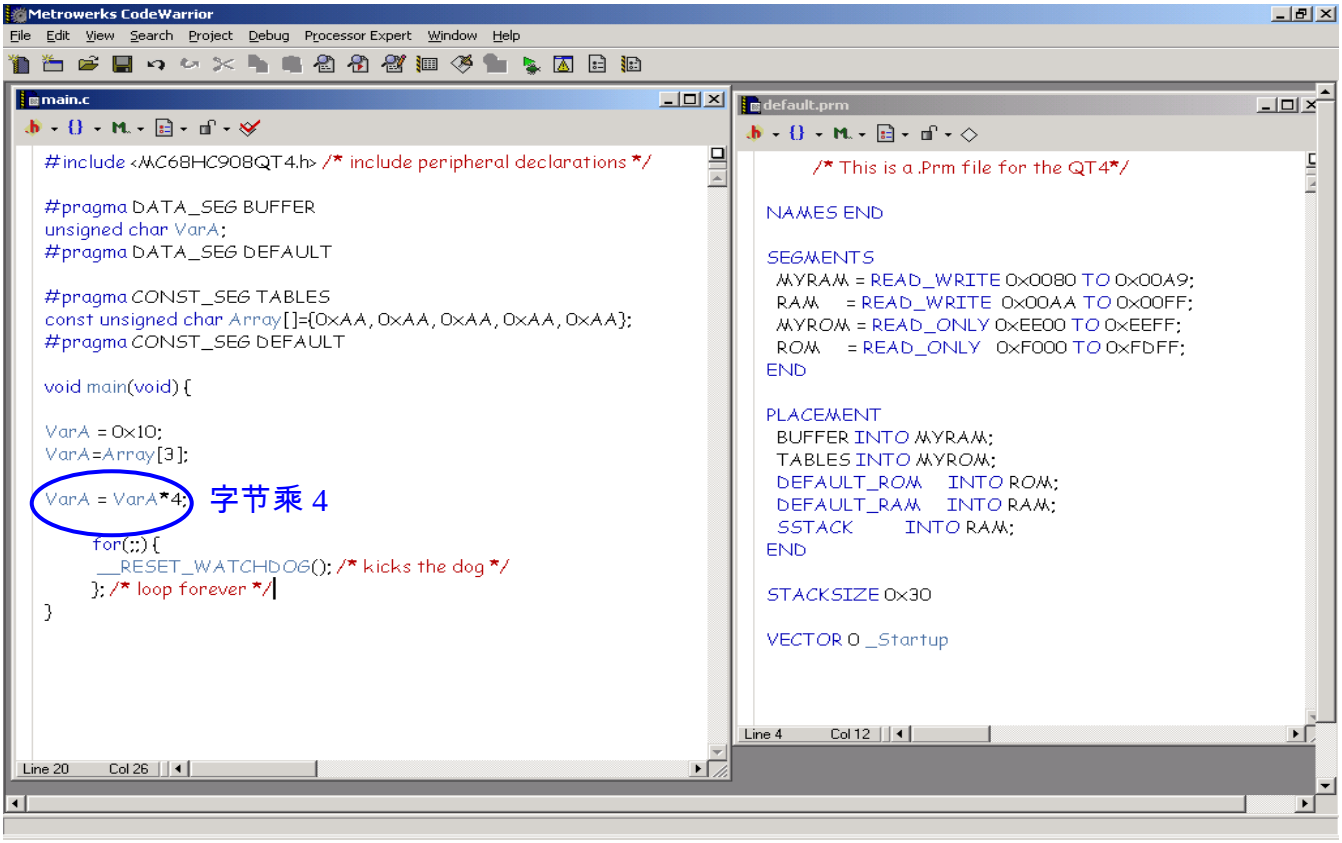
在循环内，用加法指令代替乘法指令。

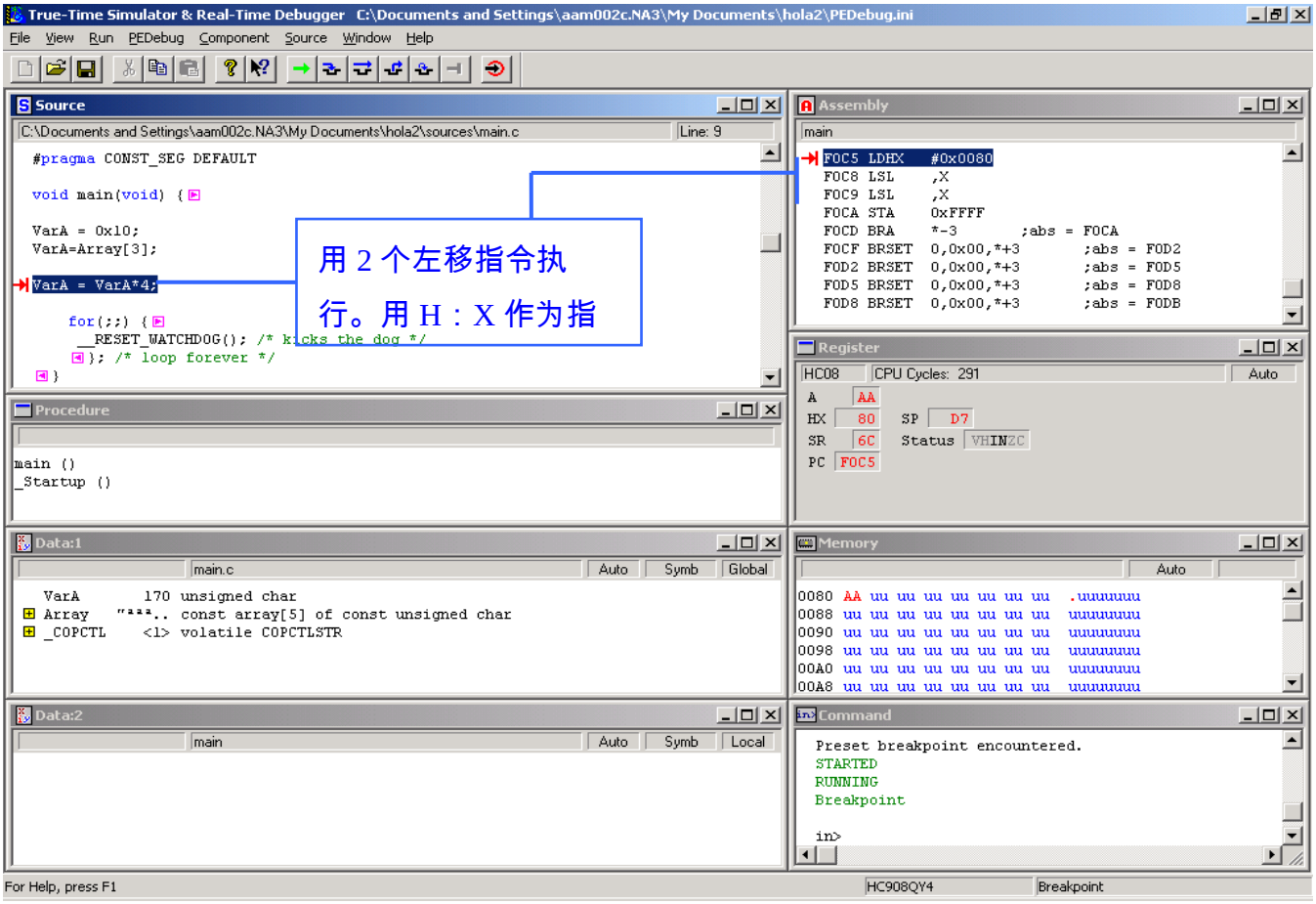
下列例子将演示编译器，基于应用做什么，决定哪一种操作用最少代价达到同样结果。

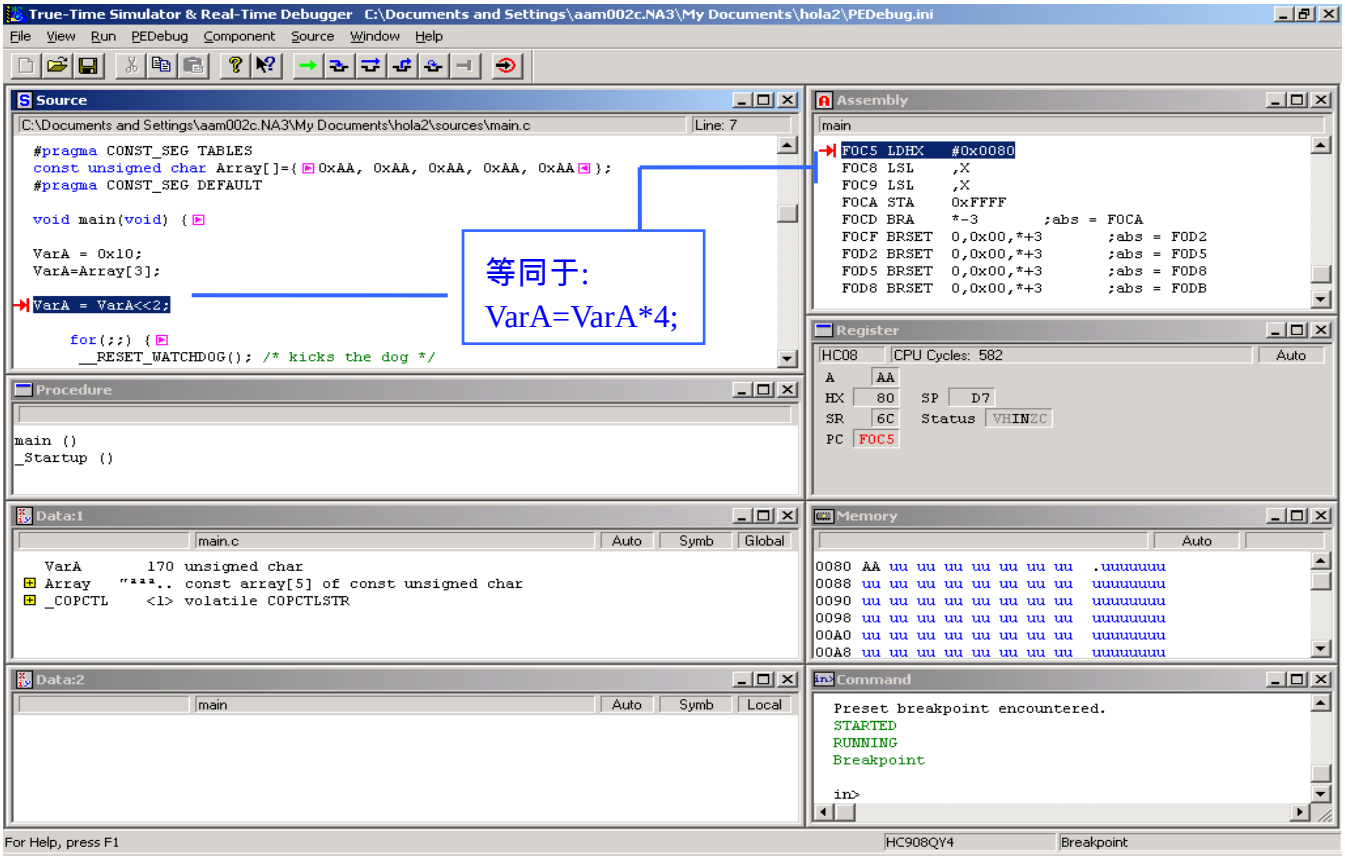
打开文件Lab10-Optimize1.mcp


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
Assembly
main
F0C5 LDX #0x03
F0C7 MUL
F0C8 STA 0x0080
F0CB STA 0xFFFF
F0CE BRA *-3 ;abs = F0CB
F0D0 BRSET 0,0x00,*+3 ;abs = F0D3
F0D3 BRSET 0,0x00,*+3 ;abs = F0D6
F0D6 BRSET 0,0x00,*+3 ;abs = F0D9
F0D9 BRSET 0,0x00,*+3 ;abs = F0DC
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

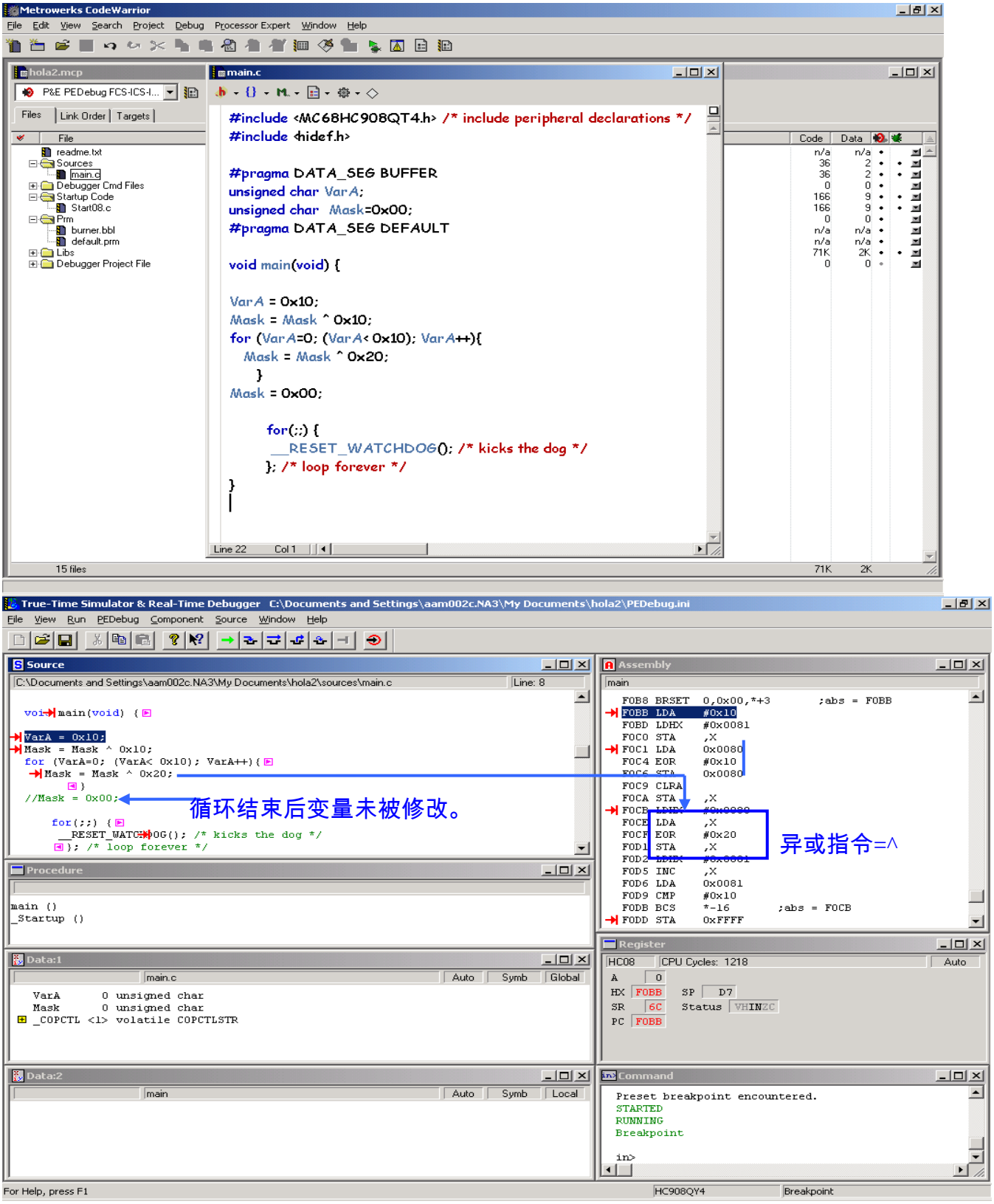

Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help
main.c
#include <MC68HC908QT4.h> /* include peripheral declarations */
#pragma DATA_SEG BUFFER
unsigned char VarA;
#pragma DATA_SEG DEFAULT
#pragma CONST_SEG TABLES
const unsigned char Array[]={0xAA, 0xAA, 0xAA, 0xAA, 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

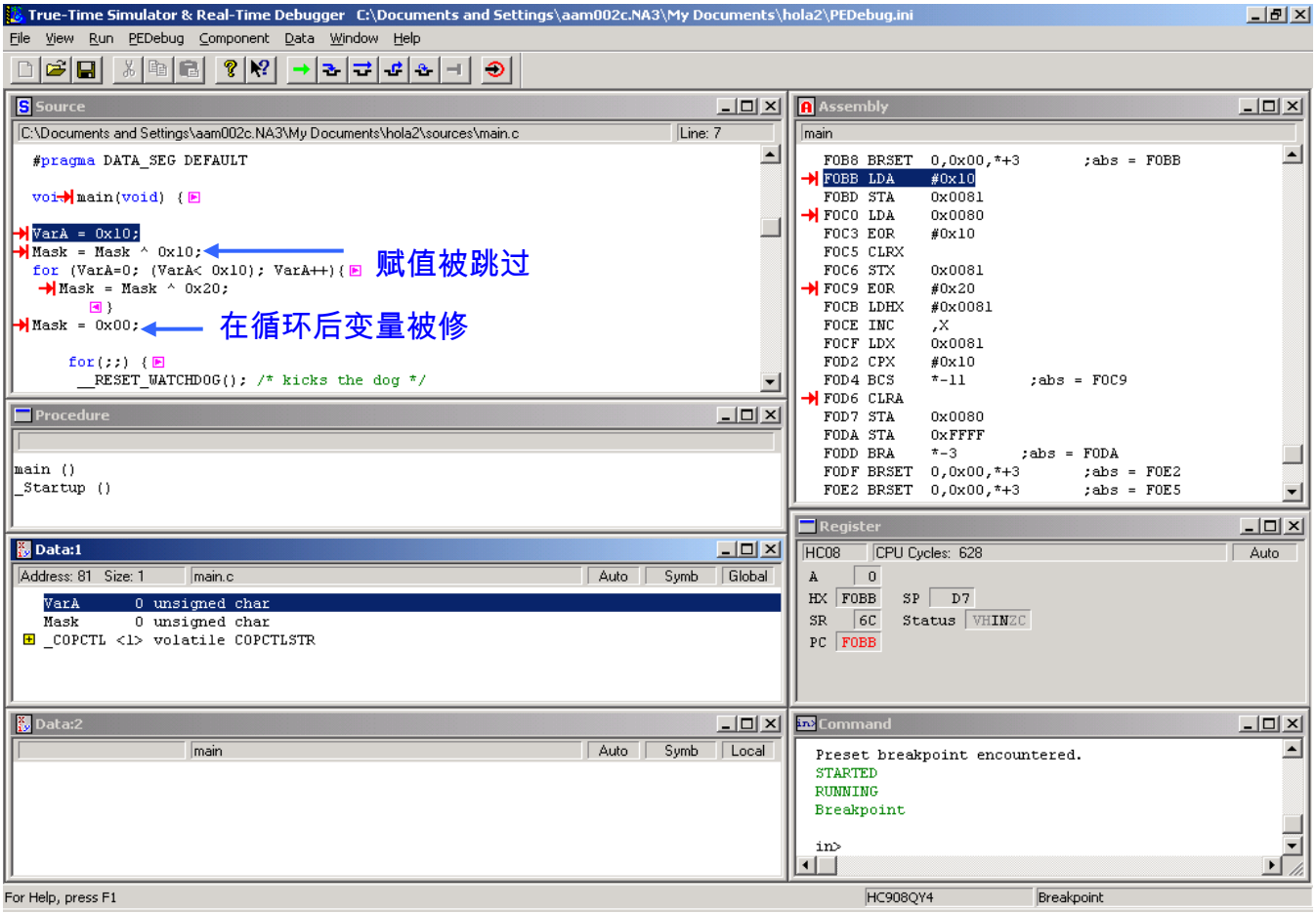

True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Source Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 9
#pragma CONST_SEG DEFAULT
void main(void) {
VarA = 0x10;
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
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 <l> 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

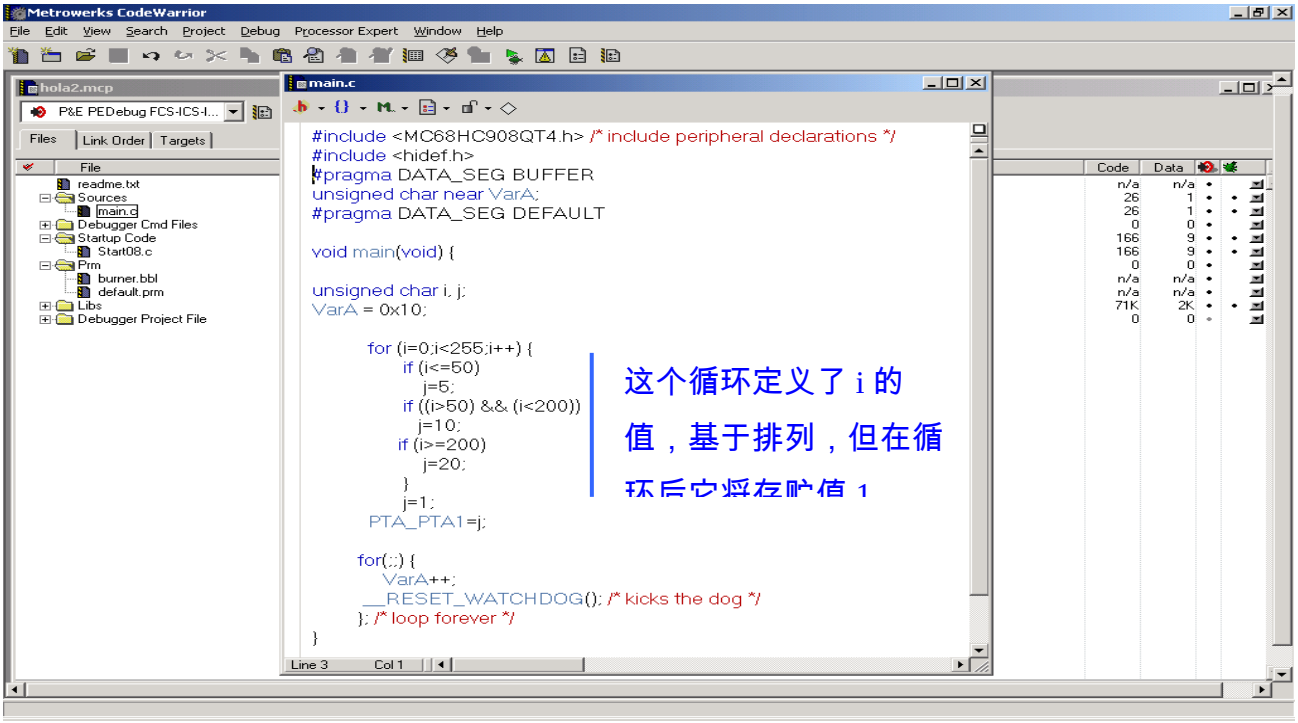
对于死代码消除，编译器优化应用程序并不为没被使用的语句产生可执行代码。移除逻辑上从未执行的语句或没有被其他语句提到的语句。打开文件：**Lab11-Optimize2.mcp**。

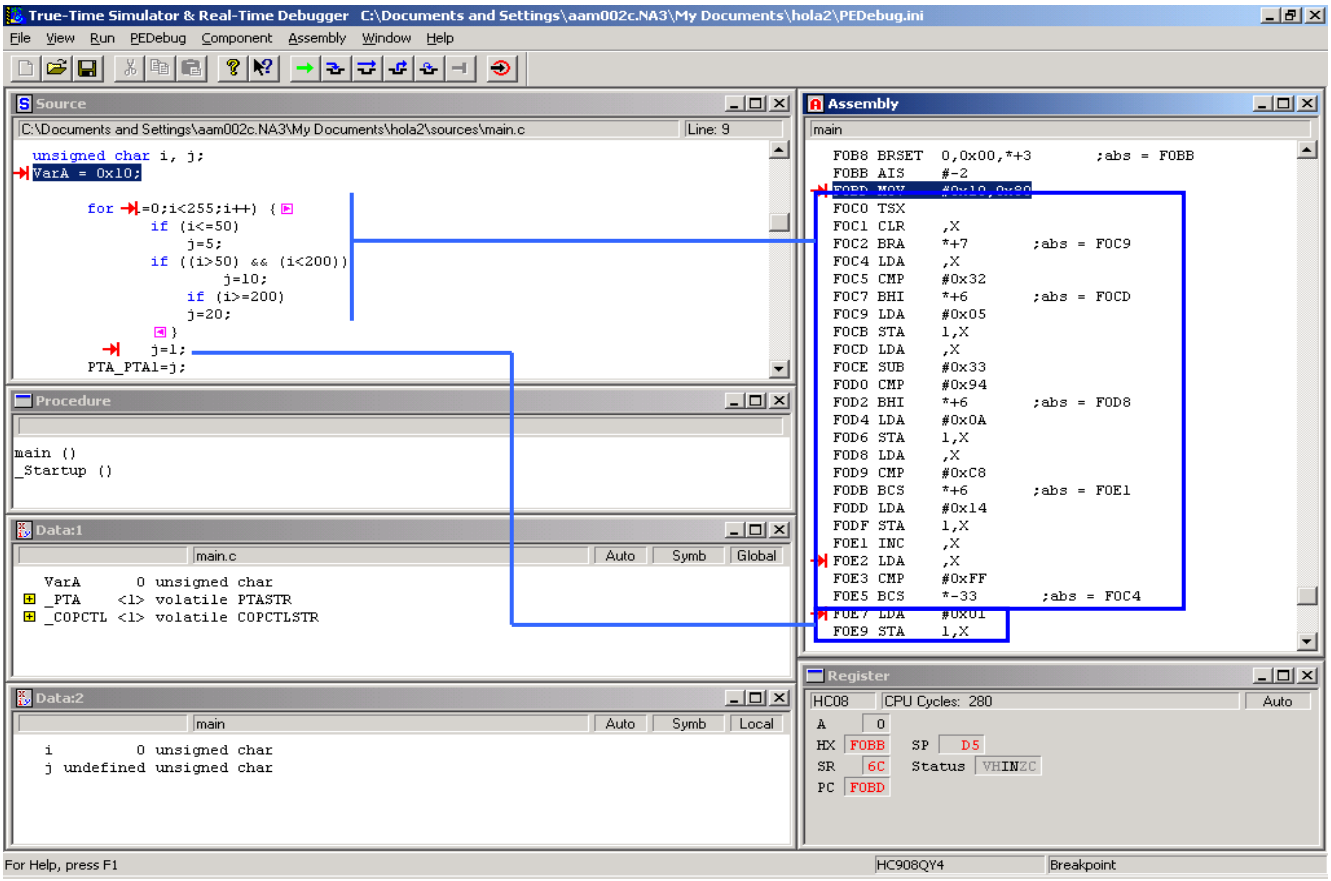

Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help
hola2.mcp
P&E PEDebug FCS-ICS-I...
Files Link Order Targets
File
readme.txt
Sources
main.c
Debugger Cmd Files
Startup Code
Start08.c
Prm
burner.bbl
default.prm
Libs
Debugger Project File
15 files
main.c
#include <MC68HC908QT4.h> /* include peripheral declarations */
#include <hidef.h>
#pragma DATA_SEG BUFFER
unsigned char VarA;
unsigned char Mask=0x00;
#pragma DATA_SEG DEFAULT
void main(void) {
VarA = 0x10;
Mask = Mask ^ 0x10;
for (VarA=0; (VarA< 0x10); VarA++){
Mask = Mask ^ 0x20;
}
Mask = 0x00;
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
}
Line 22 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 Source Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 8
VarA = 0x10;
Mask = Mask ^ 0x10;
for (VarA=0; (VarA< 0x10); VarA++){
Mask = Mask ^ 0x20;
}
//Mask = 0x00;
循环结束后变量未被修改。
for(;;) {
__RESET_WATCHDOG(); /* kicks the dog */
}; /* loop forever */
Assembly
main
F0B8 BRSET 0,0x00,*+3 ;abs = F0BB
F0BB LDA #0x10
F0BD LDHX #0x0081
F0C0 STA ,X
F0C1 LDA 0x0080
F0C4 EOR #0x10
F0C6 STA 0x0080
F0C9 CLRA
F0CA STA ,X
F0CB LDHX #0x0080
F0CE LDA ,X
F0CF EOR #0x20
F0D1 STA ,X
F0D2 LDHX #0x0081
F0D5 INC ,X
F0D6 LDA 0x0081
F0D9 CMP #0x10
F0DB BCS *-16 ;abs = F0CB
F0DD STA 0xFFFF
异或指令=^
Procedure
main ()
_Startup ()
Register
HC08 CPU Cycles: 1218 Auto
A 0
HX F0BB SP D7
SR 6C Status VHINZC
PC F0BB
Data:1
main.c Auto Symb Global
VarA 0 unsigned char
Mask 0 unsigned char
_COPCTL <1> volatile COPCTLSTR
Data:2
main Auto Symb Local
Command
Preset breakpoint encountered.
STARTED
RUNNING
Breakpoint
in>
For Help, press F1
HC908QY4
Breakpoint



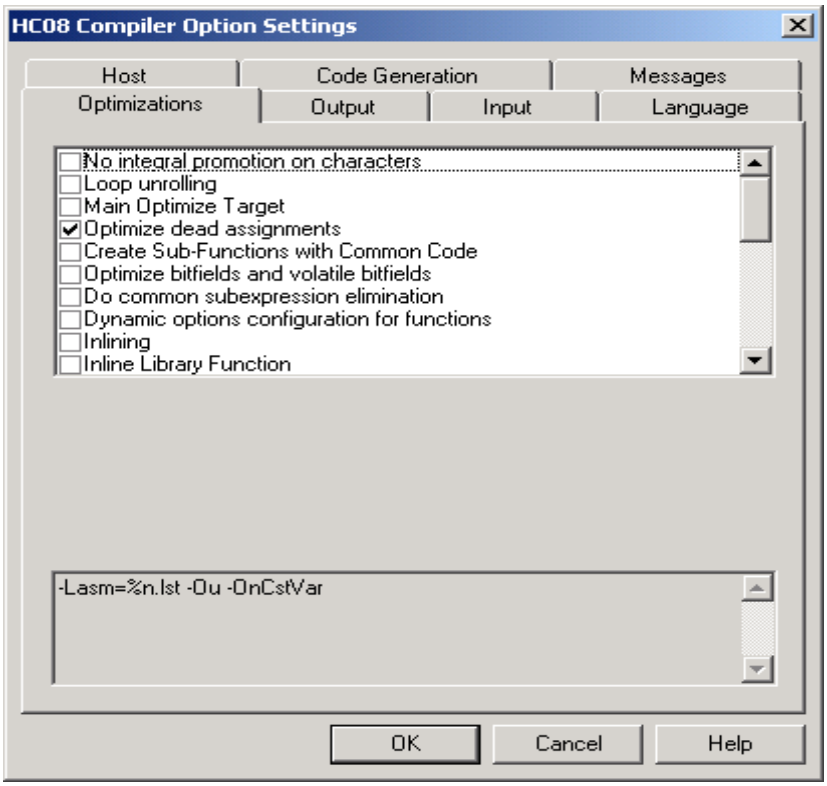
3）死赋值

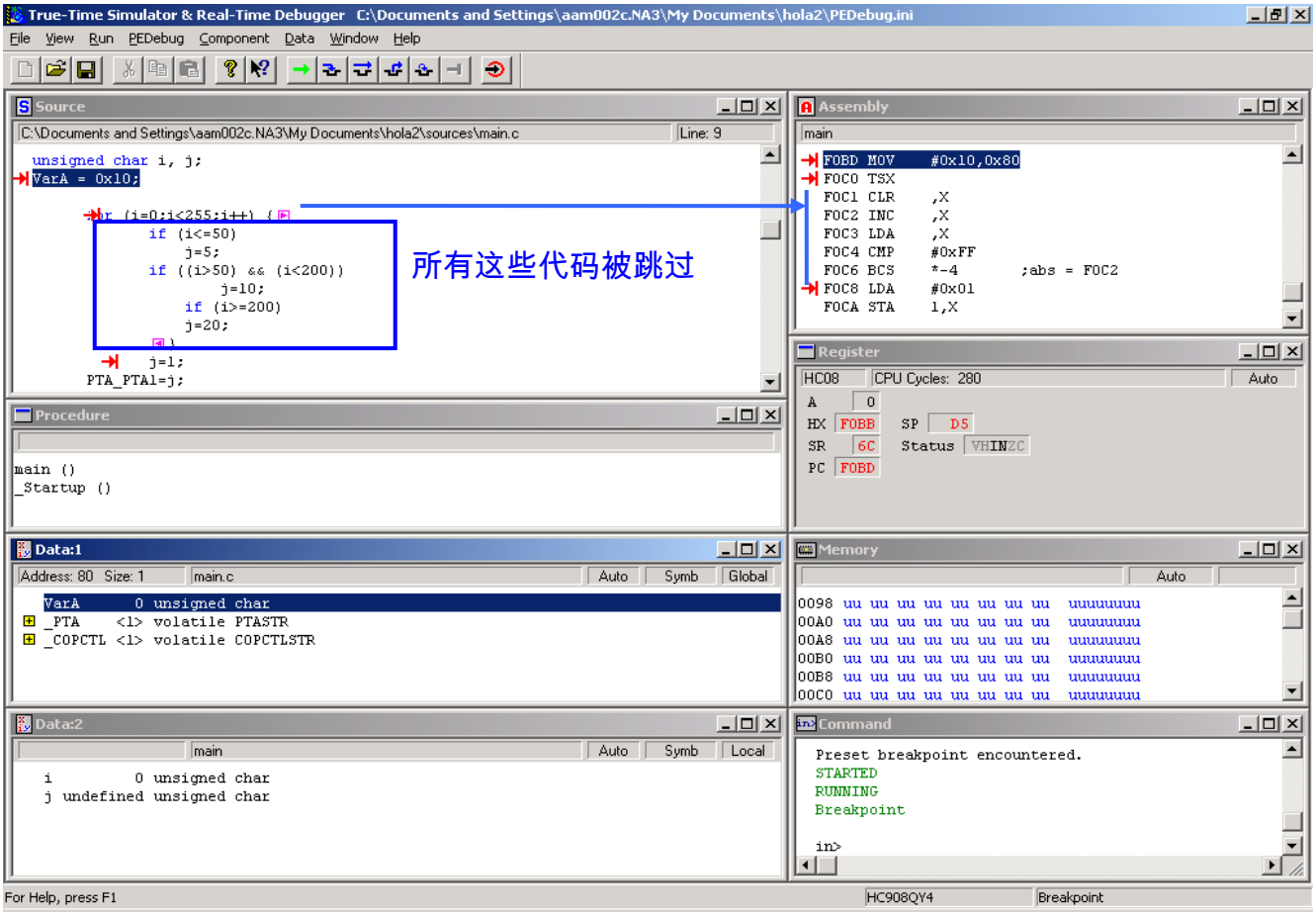
消除死赋值，编译器移去变量在再次赋值之前没有被使用的赋值。在下面编译器优化的例子，我们将演示通过改变编译器的优化设置，达到改变CodeWarrior产生代码的方式。打开文件 **Lab12-Optimize3.mcp**：

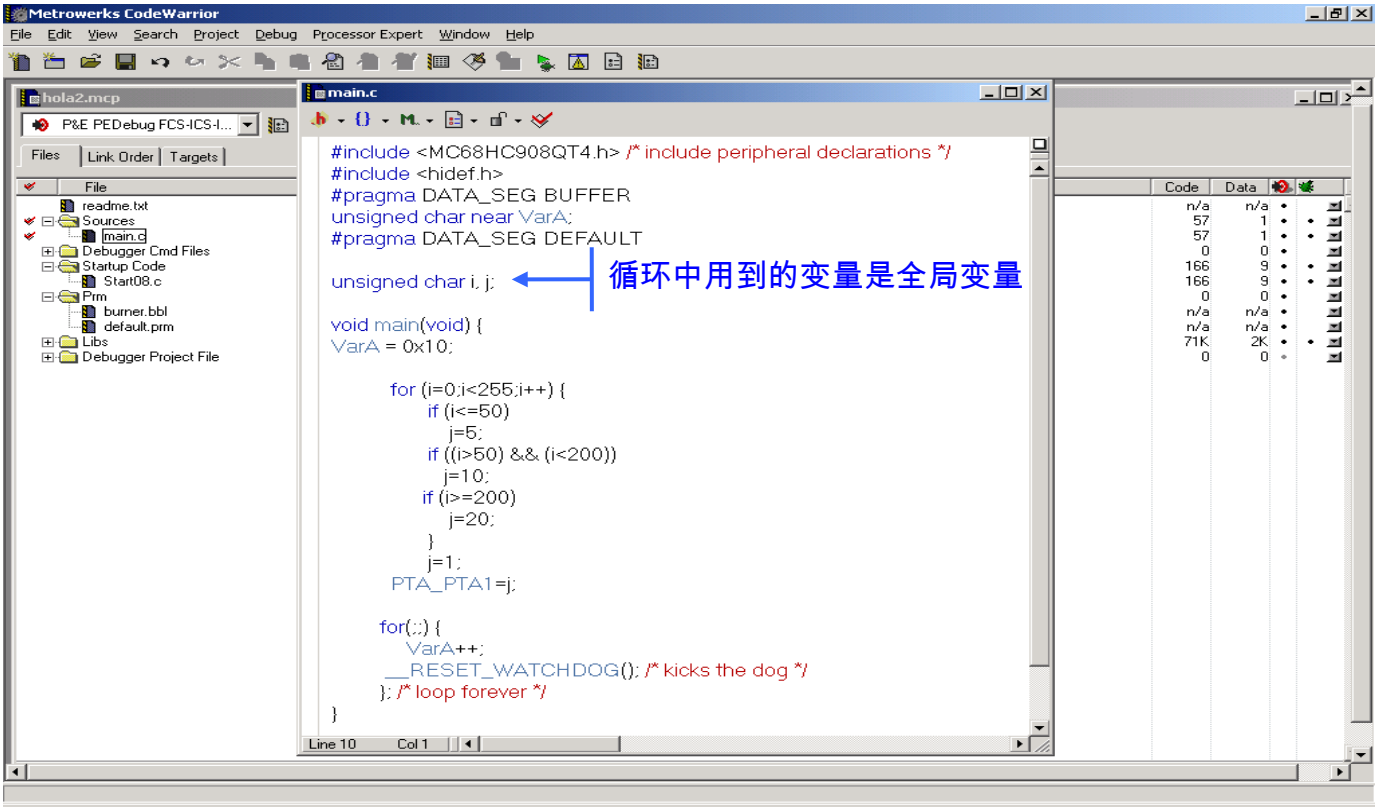

Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help
hola2.mcp
P&E PEDebug FCS-ICS-I...
Files Link Order Targets
File
readme.txt
Sources
main.c
Debugger Cmd Files
Startup Code
Start08.c
Prm
burner.bbl
default.prm
Libs
Debugger Project File
main.c
#include <MC68HC908QT4.h> /* include peripheral declarations */
#include <hidef.h>
#pragma DATA_SEG BUFFER
unsigned char near VarA;
#pragma DATA_SEG DEFAULT
void main(void) {
unsigned char i, j;
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 3 Col 1
Code Data
这个循环定义了 i 的
值，基于排列，但在循
环后它将存贮值 1

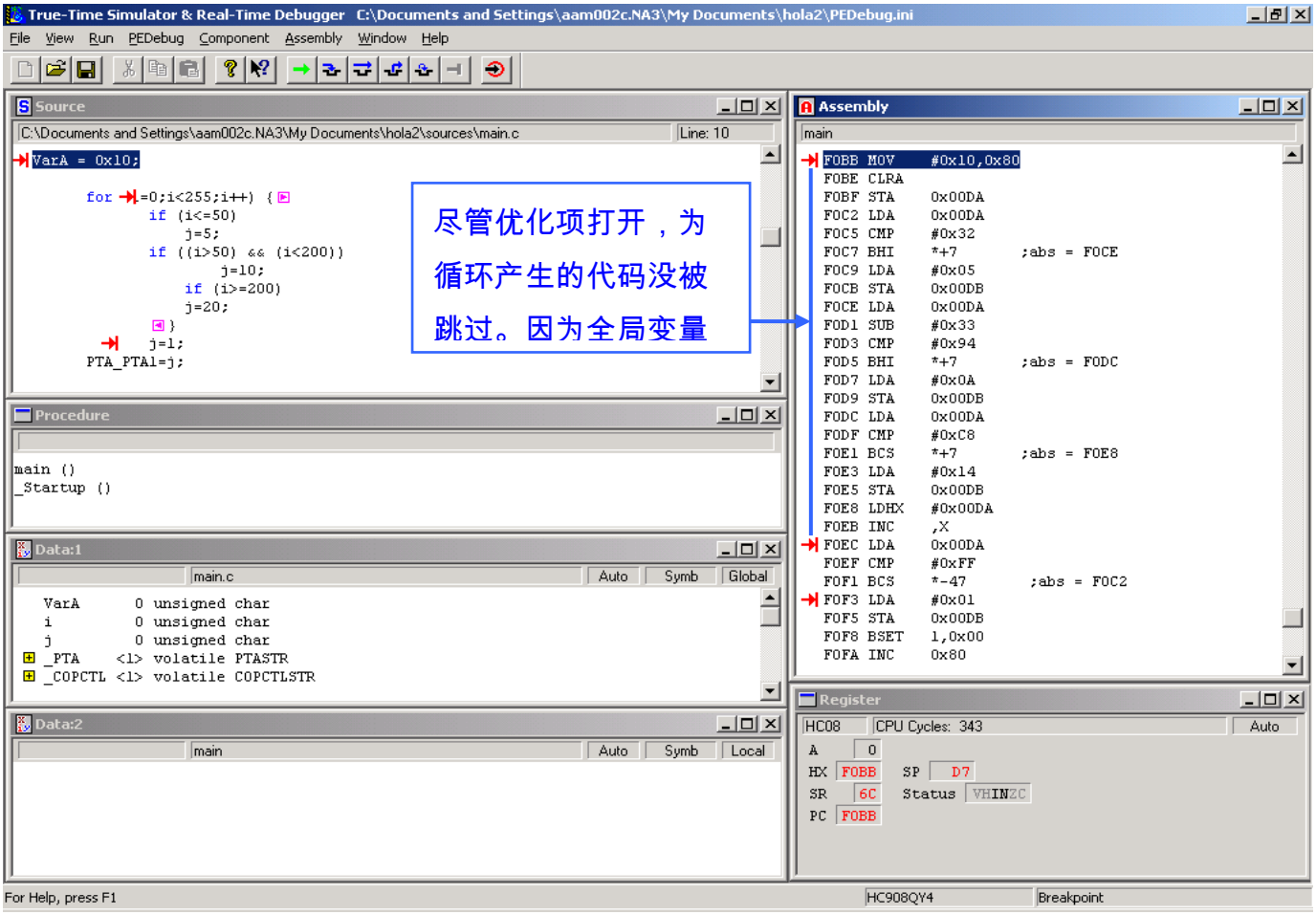

True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Assembly Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 9
unsigned char i, j;
VarA = 0x10;
for (i=0;i<255;i++) {
if (i<=50)
j=5;
if ((i>50) && (i<200))
j=10;
if (i>=200)
j=20;
}
j=1;
PTA_PTA1=j;
Assembly
main
F0B8 BRSET 0,0x00,*+3 ;abs = F0BB
F0BB AIS #-2
F0BD MOV #0x10,0x80
F0C0 TSX
F0C1 CLR ,X
F0C2 BRA *+7 ;abs = F0C9
F0C4 LDA ,X
F0C5 CMP #0x32
F0C7 BHI *+6 ;abs = F0CD
F0C9 LDA #0x05
F0CB STA 1,X
F0CD LDA ,X
F0CE SUB #0x33
F0D0 CMP #0x94
F0D2 BHI *+6 ;abs = F0D8
F0D4 LDA #0x0A
F0D6 STA 1,X
F0D8 LDA ,X
F0D9 CMP #0xC8
F0DB BCS *+6 ;abs = F0E1
F0DD LDA #0x14
F0DF STA 1,X
F0E1 INC ,X
F0E2 LDA ,X
F0E3 CMP #0xFF
F0E5 BCS *-33 ;abs = F0C4
F0E7 LDA #0x01
F0E9 STA 1,X
Procedure
main ()
_Startup ()
Data:1
main.c
Auto Symb Global
VarA 0 unsigned char
_PTA <1> volatile PTASTR
_COPCTL <1> volatile COPCTLSTR
Data:2
main
Auto Symb Local
i 0 unsigned char
j undefined unsigned char
Register
HC08 CPU Cycles: 280 Auto
A 0
HX F0BB SP D5
SR 6C Status VHINZC
PC F0BD
For Help, press F1
HC908QY4
Breakpoint

4）Codewarrior HC08 编译器选项设置



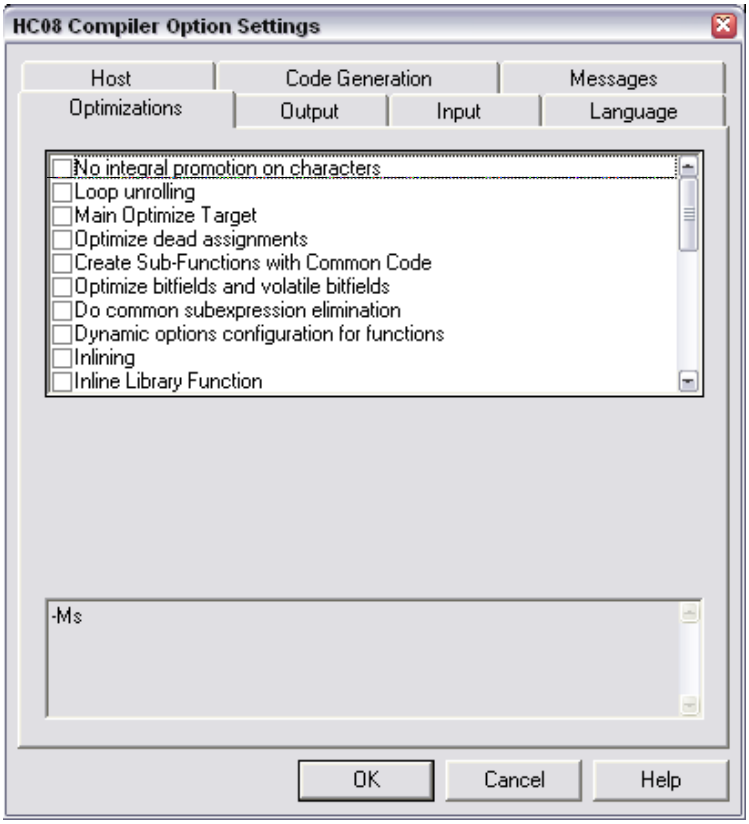

True-Time Simulator & Real-Time Debugger C:\Documents and Settings\aam002c.NA3\My Documents\hola2\PEDebug.ini
File View Run PEDebug Component Data Window Help
Source
C:\Documents and Settings\aam002c.NA3\My Documents\hola2\sources\main.c
Line: 9
unsigned char i, j;
VarA = 0x10;
for (i=0;i<255;i++) {
if (i<=50)
j=5;
if ((i>50) && (i<200))
j=10;
if (i>=200)
j=20;
}
j=1;
PTA_PTA1=j;
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

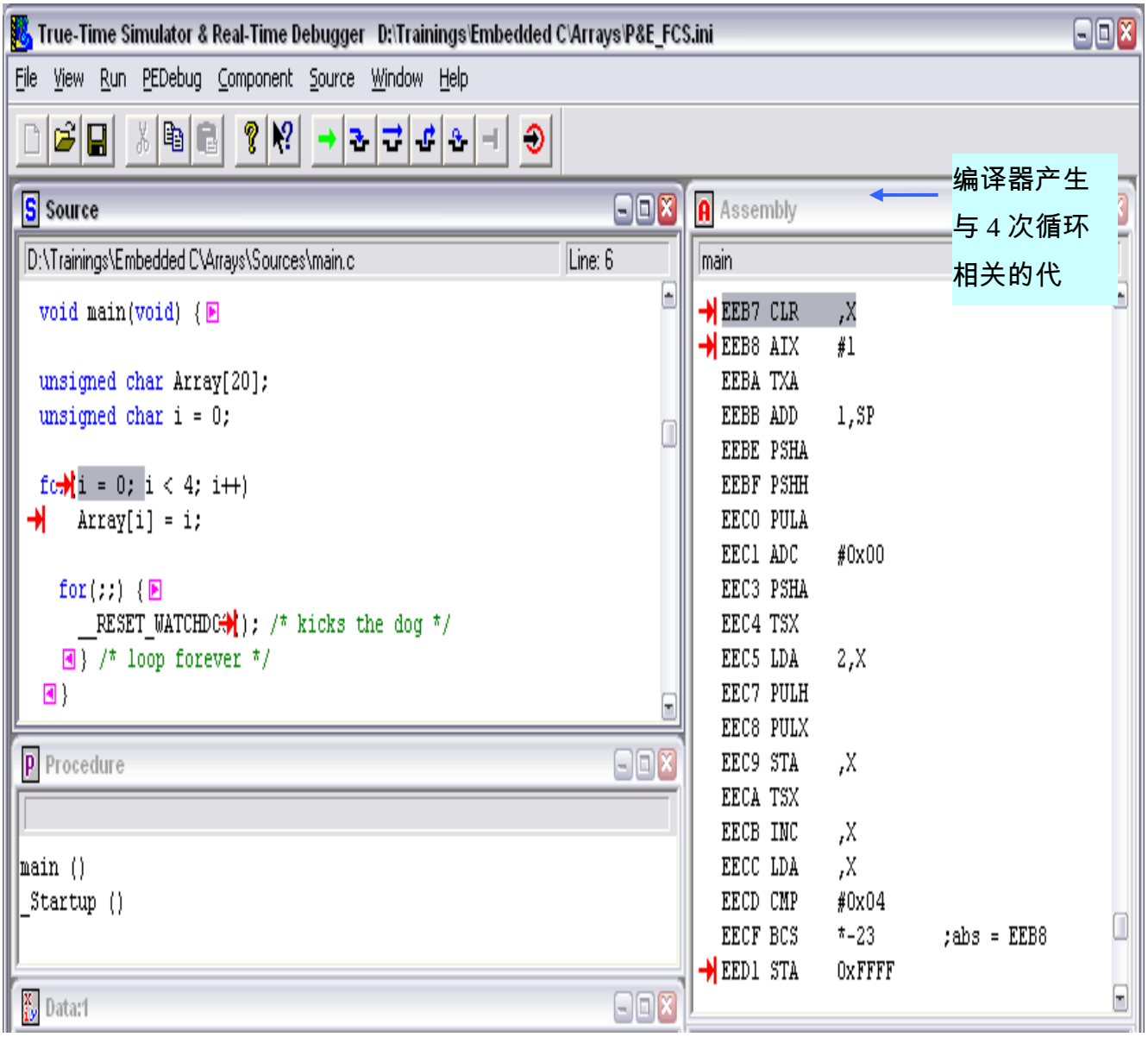


5）循环解开

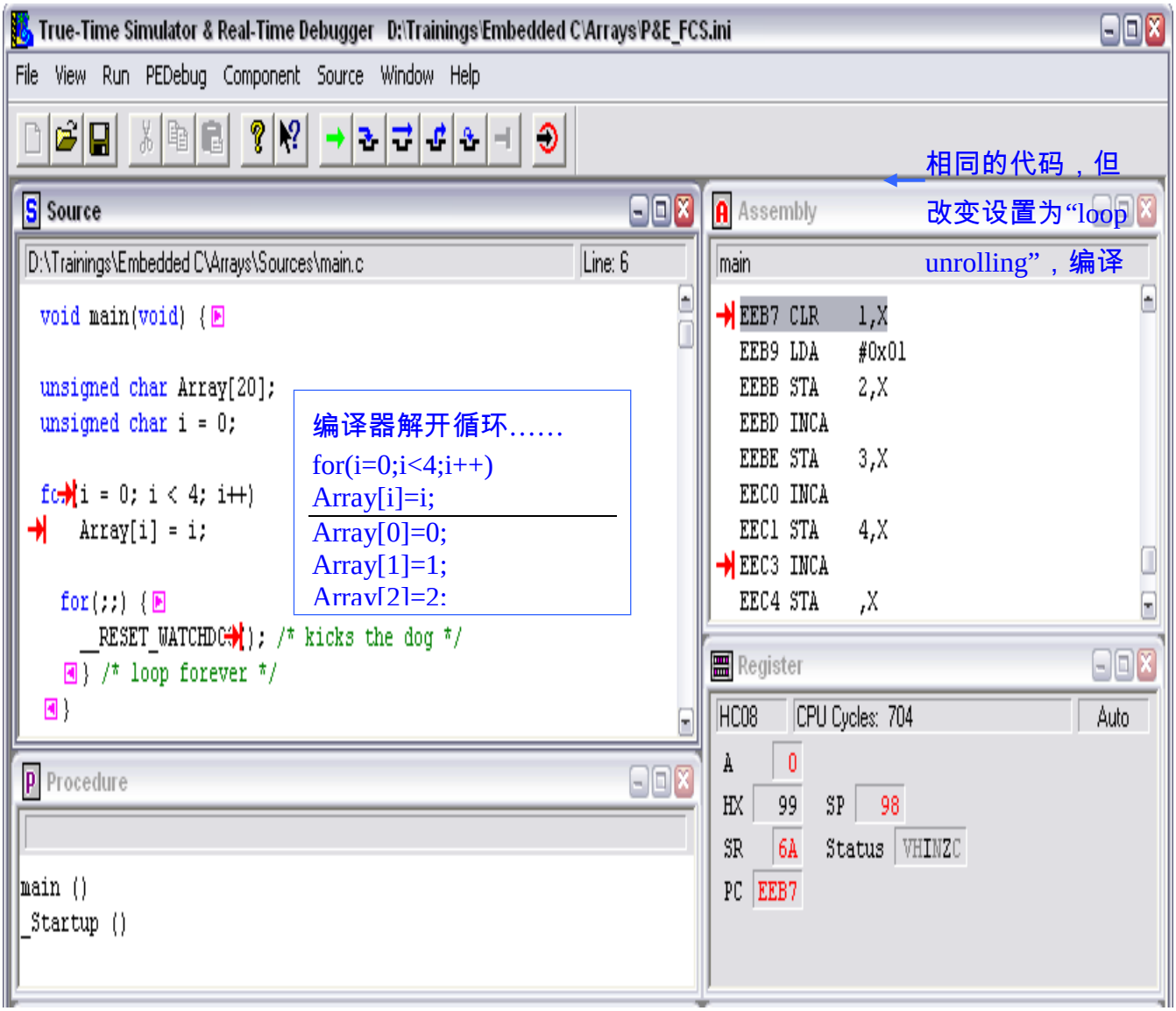
通过改变编译器设置，我们可选择不同的优化项，在生成代码时将会有差异。



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）


True-Time Simulator & Real-Time Debugger D:\Trainings\Embedded C\Arrays\P&E_FCS.ini
File View Run PEDebug Component Source Window Help
Source
D:\Trainings\Embedded C\Arrays\Sources\main.c
Line: 6
void main(void) {
unsigned char Array[20];
unsigned char i = 0;
for(i = 0; i < 4; i++)
Array[i] = i;
for(;;) {
_RESET_WATCHDOG(); /* kicks the dog */
} /* loop forever */
}
Procedure
main ()
_Startup ()
Data:1
Assembly
main
EEB7 CLR ,X
EEB8 AIX #1
EEBA TXA
EEBB ADD 1,SP
EEBE PSHA
EEBF PSHH
EEC0 PULA
EEC1 ADC #0x00
EEC3 PSHA
EEC4 TSX
EEC5 LDA 2,X
EEC7 PULH
EEC8 PULX
EEC9 STA ,X
EECA TSX
EECB INC ,X
EECC LDA ,X
EECD CMP #0x04
EECF BCS *-23 ;abs = EEB8
EED1 STA 0xFFFF
编译器产生
与 4 次循环
相关的代

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

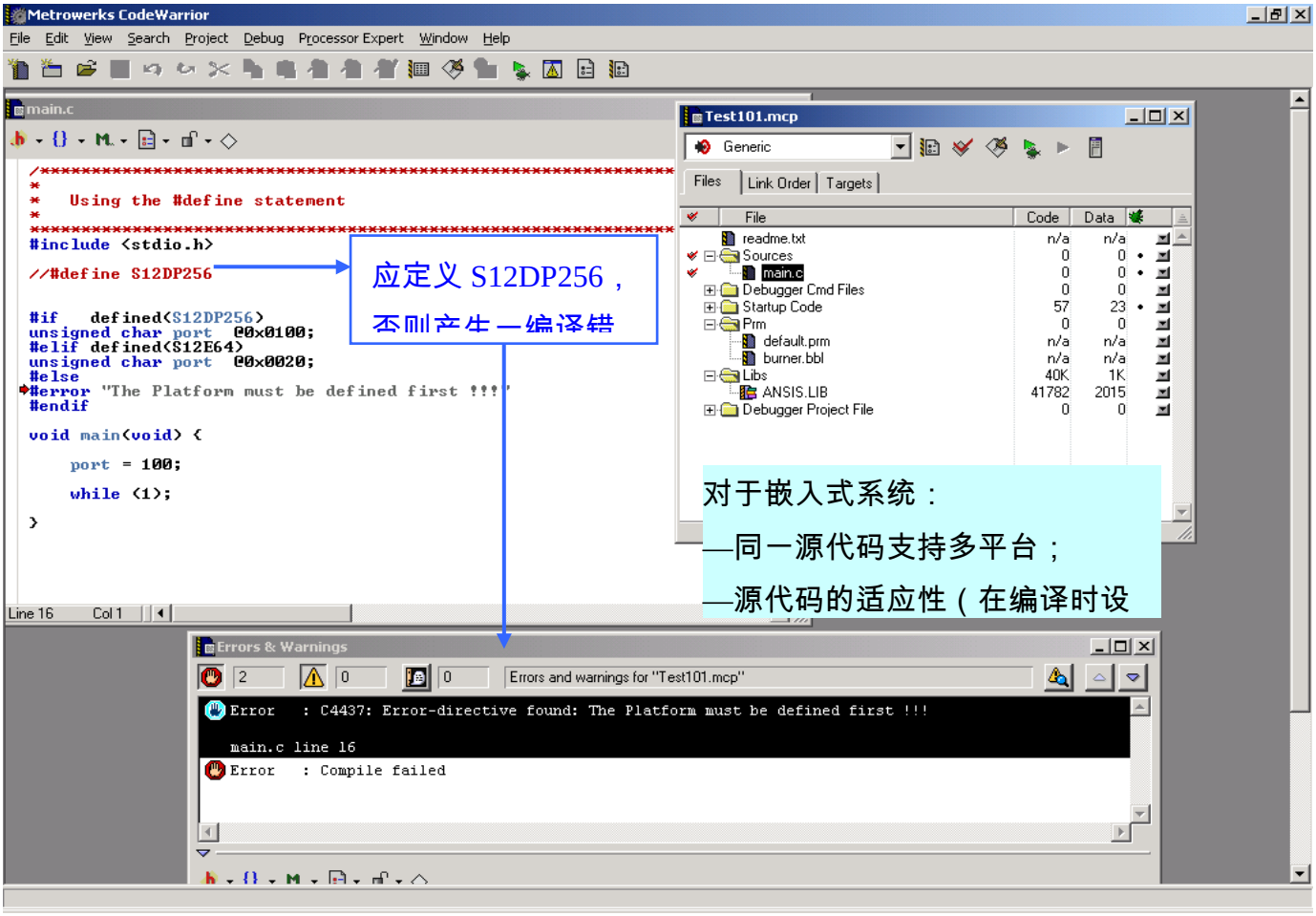
编译指示符：#if、#else、#elif、#endif

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。



嵌入式开发入门经验

你好,我是一名嵌入式开发爱好者.近来总是看到很多初学者（多数是在校大学生）由于没有条件，想学习却不知道如何下手。

本人绝对能体会到学习的艰辛，而且视任何对知识技术有强烈追求的人（不管目前水平高低）为同路人，所以整理短文一篇写出一些学习感受，替所有渴望知识的人企盼高手指点学习之路，分享经验。

爱因斯坦说过，”我是站在巨人的肩膀上“

实践当然是最锻炼人的方式，但是我想在校生很少有这样的机会，别说本科生，硕士生也未必有条件。所以我想学习嵌入式要从个人的知识背景和现实条件出发。订立合适的阶段目标，在允许的条件下多动手多思考。

一般情况下对于硬件设备是比较短缺的。但是可以从软件方面和嵌入式系统开发模式上下功夫，提醒大家一点，嵌入式系统开发设计的内容知识很多，所以大家不要乱，在了解嵌入式系统开发的体系结构后，一步一步的下手，最容易上手的是linux下的C，比如ucos（有开放源代码），虽然可能无法在硬件上仿真，但也不必着急。wince，palmos上手都很容易。无论对于初学者还是自以为是高手的人来说，编程水平（这可不受硬件条件限制）绝对是没有止境的，有了较高的编程水平（嵌入式主要是C,当然OO的几种语言c++,java是发展趋势），等到有机会的时候及时的补充硬件知识，会很快的成为高手。

还有，一定记住，学习嵌入式，”要想办法，不要找理由“。当年在dos下用tc编程时的条件，现在回想起来建筑就是奇迹。

我推荐一条学习之路吧，仅供参考，

1、C开发经验

条件：linux（这都有吧）

方法：随便，主要是掌握ansiC编程（不包括gtk,qt等图形可视化开发）

2、网络、操作系统、体系结构

条件：linux，各种书，算法、例程。

方法：通过C编程实现简单的网络等知识的算法和过程。

3、嵌入式系统概念

条件：各个嵌入式网站，讨论组，书籍

方法：少提问（尤其是等着天上掉馅饼，这主要是防止增长惰性，也解决不了实际问题），多思考。

4、嵌入式开发实践

条件：各种嵌入式系统开发工具的demo版（或者D版，如果有的话），包括编译器，仿真器。可以找高手们要，也可以下载。

方法：这里有两个分支，一个是基于mcu/dsp的嵌入式系统开发，一个是象palmos,wince,ucos等rtos下的应用软件开发。对硬件感兴趣，想成为真正高手的由第一个分支入手，以后进入第二个分支；如果十分厌烦硬件，只想停留在软件开发上的，可以只由第二个分支入手，以后就和pc上的开发没有什么本质上的区别了，找份不错的工作应该没问题，可以不用进行下面的步骤了，感兴趣可以参考第 6 条。

5、硬件开发

条件：各种嵌入式芯片、存储器等电路器件，protel99 等电路设计软件，电路板制作。

方法：这时候该有开发条件了，最起码是 51 系列，这个比较方便。电路的设计内容较多，不过看起来吓人，实际上比软件要简单的多。只要下功夫，实践会告诉你一切。

6、硬件工程/软件工程/项目管理

条件： 各个芯片详细资料和使用经验，软件工程知识，项目管理知识，培训，大型项目参与经验

方法：已经是高手了，但是学无止境，沾沾自喜于已有的知识是致命的。那个下一步。。。，你该是管理者了。

~~~~~~~~~~~~~~~~~~~~~~~~~~留个爪印~~~~~~~~~~~~~~~~~~~~~~~~~

嵌入式学习的路径

嵌入式学习的路径有哪些?

这是个初学者常问的问题,也是初学者问嵌入式该如何入门的根源..我感觉有两个方面,偏硬和偏软.我不认为嵌入式研发软件占绝对比重,相反,软件和硬件都懂,才是嵌入式高手所应该追求的,也是高手的必由之路.

硬件道路:

第一步: pcb设计,一般为研发板的电路裁减和扩充,由研发板原理图为基础,画出PCB和封装库,设计自己的电路.
第二步: SOPC技术,一般为FPGA,CPLD研发,利用VHDL等硬件描述语言做专用芯片研发,写出自己的逻辑电路,基于ALTER或XILINUX的FPGA做研发.
第三步: SOC设计,分前端,后端实现,这是硬件设计的核心技术:芯片设计.能做到这步,已不属于平凡的技术人员.

软件道路:

第一步:bootloader的编写，修改, 通过这步熟悉ARM硬件结构,学习ARM汇编语言,阅读ARM的芯片手册,感觉就是像操作 51 单片机相同操作ARM芯片.这一步最好的两个参考资料就是:芯片手册和bootloader源代码.
第二步:系统移植, 驱动研发,
我只做过linux方向,所以也推荐学习嵌入式linux系统,作为标准体系,他开源而且能够获得大量学习资料.操作系统是
~~~~~~~~~~~~~~~~~~~~~~~~~~

整个电脑科学的核心,熟悉 kernel实属不易,kernel, 驱动研发的学习,没有什么捷径,只有多读代码,多写代码,熟悉系统API.. understanding linux kernel , linux device driver 都是不可多得的好书，值得一看.
第三步:应用程式的编写,各种GUI的移植,qt , minigui都被大量采用,两种思想都类似,熟悉一种就能够.
软件道路中,驱动,系统应该是最深入的部分,不是短时间能够掌控的,需要有勇气和耐心..
嵌入式研发,软硬结合,因为硬件条件比PC差很多,所以肯定会遇见不少问题,因此实践的勇气更加重要.有问题就解决问题，无数次的实验,也许是解决问题的必由之路.

Linux文件查找技术大全

每一种操作系统都是由成千上万个不同种类的文件所组成的。其中有系统本身自带的文件，用户自己的文件，还有共享文件等等。我们有时候经常忘记某份文件放在硬盘中的哪个地方。在微软的WINDOWS操作系统中要查找一份文件是相当简单的事情，只要在桌面上点击“开始” - “搜索”中就能按照各种方式在本地硬盘上，局域网络，甚至在INTERNET上查找各种文件，文档。

可是使用Linux的用户就没有那么幸运了，在Linux上查找某个文件确实是一件比较麻烦的事情。毕竟在Linux中需要我们使用专用的“查找”命令来寻找在硬盘上的文件。Linux下的文件表达格式非常复杂，不象WINDOWS,DOS下都是统一的AAAAAAA.BBB格式那么方便查找，在 WINDOWS中，只要知道要查找的文件的文件名或者后缀就非常容易查找到。Linux中查找文件的命令通常为“find”命令，“find”命令能帮助我们在使用,管理Linux的日常事务中方便的查找出我们需要的文件。对于Linux新手来说，“find”命令也是了解和学习Linux文件特点的方法。因为Linux发行版本繁多，版本升级很快，在Linux书籍上往往写明某个配置文件的所在位置，往往Linux新手按图索骥还是不能找到。比如说 REDHAT Linux 7.O和REDHAT Linux 7.1 中有些重要的配置文件所在的硬盘位置和文件目录就有了很大的改变，如果不学会使用“find”命令，那么在成千上万的Linux文件中要找到其中的一个配置文件是相当困难的，笔者在没有精通“find”命令之前就吃过这样的苦头。好，下面就详细为大家介绍强大的“find”命令的全部使用方法和用途。

通过文件名查找法：

这个方法说起来就和在WINDOWS下查找文件一样容易理解了。如果你把这个文件放在单个的文件夹里面，只要使用常见的“ls"命令就能方便的查找出来，那么使用“find”命令来查找它就不能给你留下深刻的印象，毕竟“find”命令的强大功能不止这个。如果知道了某个文件的文件名，而不知道这个文件放到哪个文件夹，甚至是层层套嵌的文件夹里。举例说明，假设你忘记了httpd.conf这个文件在系统的哪个目录下，甚至在系统的某个地方也不知道，则这是可以使用如下命令：

find / -name httpd.conf

这个命令语法看起来很容易就明白了，就是直接在find后面写上 -name，表明要求系统按照文件名查找，最后写上httpd.conf这个目标文件名即可。稍等一会系统会在计算机屏幕上显示出查找结果列表：
etc/httpd/conf/httpd.conf

这就是httpd.conf这个文件在Linux系统中的完整路径。查找成功。
如果输入以上查找命令后系统并没有显示出结果，那么不要以为系统没有执行find/ -name httpd.conf命令，而可能是你的系统中没有安装Apache服务器，这时只要你安装了Apache Web服务器，然后再使用find / -name httpd.conf就能找到这个配置文件了。

无错误查找技巧：

在Linux系统中“find”命令是大多数系统用户都可以使用的命令，并不是ROOT系统管理员的专利。但是普通用户使用“find”命令时也有可能遇到这样的问题，那就是Linux系统中系统管理员ROOT可以把某些文件目录设置成禁止访问模式。这样普通用户就没有权限用“find”命令来查询这些目录或者文件。当普通用户使用“find”命令来查询这些文件目录是，往往会出现"ermissiondenied."（禁止访问）字样。系统将无法查询到你想要的文件。为了避免这样的错误，我们可是使用转移错误提示的方法尝试着查找文件，输入
find / -name access_log 2>;/dev/null

这个方法是把查找错误提示转移到特定的目录中去。系统执行这个命令后，遇到错误的信息就直接输送到stderrstream 2 中，access_log 2 就是表明系统将把错误信息输送到stderrstream 2 中，/dev/null是一个特殊的文件，表明空的或者错误的信息，这样查询到的错误信息将被转移了，不会再显示了。

在Linux系统查找文件也会遇到这样一个实际问题。如果我们在整个硬盘，这个系统中查找某个文件就要花费相当长的一段时间，特别是大型Linux系统和容量较大的硬盘，文件放在套嵌很深的目录中的时候。如果我们知道了这个文件存放在某个大的目录中，那么只要在这个目录中往下找就能节省很多时间了。使用 find /etc -name httpd.conf就可以解决这个问题。上面的命令就是表示在etc目录中查询httpd.conf这个文件。这里再说明一下“/ ”这个函数符号的含义，如果输入 “find/ ”就是表示要求Linux系统在整个ROOT目录下查找文件，也就是在整个硬盘上查找文件，而“find/etc”就是只在 etc目录下查找文件。因为“find/etc”表示只在etc目录下查找文件，所以查找的速度就相应要快很多了。

根据部分文件名查找方法：

这个方法和在WINDOWS中查找已知的文件名方法是一样的。不过在Linux中根据部分文件名查找文件的方法要比在WINDOWS中的同类查找方法要强大得多。例如我们知道某个文件包含有srm这 3 个字母，那么要找到系统中所

有包含有这 3 个字母的文件是可以实现的，输入：
find /etc -name '*srm*'

这个命令表明了Linux系统将在/etc整个目录中查找所有的包含有srm这 3 个字母的文件，比如 absrmyz， tibc.srm等等符合条件的文件都能显示出来。如果你还知道这个文件是由srm 这 3 个字母打头的，那么我们还可以省略最前面的星号，命令如下：
find/etc -name 'srm*'

这是只有像srmyz 这样的文件才被查找出来，象absrmyz或者 absrm这样的文件都不符合要求，不被显示，这样查找文件的效率和可靠性就大大增强了。

根据文件的特征查询方法：

如果只知道某个文件的大小，修改日期等特征也可以使用“find”命令查找出来，这和WINDOWS系统中的"搜索"功能是基本相同的。在微软的"搜索" 中WINDOWS中的"搜索助理"使得搜索文件和文件夹、打印机、用户以及网络中的其他计算机更加容易。它甚至使在Internet 上搜索更加容易。"搜索助理"还包括一个索引服务，该服务维护了计算机中所有文件的索引，使得搜索速度更快。使用"搜索助理"时，用户可以指定多个搜索标准。例如，用户可以按名称、类型及大小搜索文件和文件夹。用户甚至可以搜索包含特定文本的文件。如果用户正使用 Active Directory，这时还可以搜索带有特定名称或位置的打印机。

例如我们知道一个Linux文件大小为 1,500 bytes，那么我们可是使用如下命令来查询find / -size 1500c，字符 c 表明这个要查找的文件的大小是以bytes为单位。如果我们连这个文件的具体大小都不知道，那么在Linux中还可以进行模糊查找方式来解决。例如我们输入find/ -size +10000000c 这个命令，则标明我们指定系统在根目录中查找出大于 10000000 字节的文件并显示出来。命令中的“+”是表示要求系统只列出大于指定大小的文件，而使用“-”则表示要求系统列出小于指定大小的文件。下面的列表就是在Linux使用不同“ find"命令后系统所要作出的查找动作，从中我们很容易看出在Linux中使用“find"命令的方式是很多的，“ find"命令查找文件只要灵活应用，丝毫不必在WINDOWS中查找能力差。

find / -amin -10 # 查找在系统中最后 10 分钟访问的文件
find / -atime -2 # 查找在系统中最后 48 小时访问的文件
find / -empty # 查找在系统中为空的文件或者文件夹
find / -group cat # 查找在系统中属于 groupcat的文件
find / -mmin -5 # 查找在系统中最后 5 分钟里修改过的文件
find / -mtime -1 #查找在系统中最后 24 小时里修改过的文件
find / -nouser #查找在系统中属于作废用户的文件
find / -user fred #查找在系统中属于FRED这个用户的文件

下面的列表就是对find命令所可以指定文件的特征进行查找的部分条件。在这里并没有列举所有的查找条件，参考有关Linux有关书籍可以知道所有find命令的查找函数。
-amin n
查找系统中最后N分钟访问的文件
-atime n
查找系统中最后n*24 小时访问的文件
-cmin n
查找系统中最后N分钟被改变状态的文件
-ctime n
查找系统中最后n*24 小时被改变状态的文件
-empty
查找系统中空白的文件，或空白的文件目录，或目录中没有子目录的文件夹
-false
查找系统中总是错误的文件
-fstype type
查找系统中存在于指定文件系统的文件，例如：ext2 .
-gid n
查找系统中文件数字组 ID 为 n的文件
-group gname
查找系统中文件属于gnam文件组，并且指定组和ID的文件

Find命令的控制选项说明：

Find命令也提供给用户一些特有的选项来控制查找操作。下表就是我们总结出的最基本，最常用的find命令的控制选项及其用法。
选项
用途描述
-daystart
.测试系统从今天开始 24 小时以内的文件，用法类似-amin
-depth
使用深度级别的查找过程方式,在某层指定目录中优先查找文件内容
-follow
遵循通配符链接方式查找; 另外，也可忽略通配符链接方式查询
-help
显示命令摘要
-maxdepth levels
在某个层次的目录中按照递减方法查找
-mount

不在文件系统目录中查找， 用法类似 -xdev.
-noleaf
禁止在非UNUX文件系统，MS-DOS系统，CD-ROM文件系统中进行最优化查找
-version
打印版本数字

使用-follow选项后，find命令则遵循通配符链接方式进行查找，除非你指定这个选项，否则一般情况下find命令将忽略通配符链接方式进行文件查找。

-maxdepth选项的作用就是限制find命令在目录中按照递减方式查找文件的时候搜索文件超过某个级别或者搜索过多的目录，这样导致查找速度变慢，查找花费的时间过多。例如，我们要在当前(.)目录技巧子目录中查找一个名叫fred的文件，我们可以使用如下命令
find . -maxdepth 2 -name fred

假如这个fred文件在./sub1/fred目录中，那么这个命令就会直接定位这个文件，查找很容易成功。假如，这个文件在./sub1/sub2 /fred目录中，那么这个命令就无法查找到。因为前面已经给find命令在目录中最大的查询目录级别为2，只能查找 2 层目录下的文件。这样做的目的就是为了让find命令更加精确的定位文件，如果你已经知道了某个文件大概所在的文件目录级数，那么加入-maxdepth n 就很快的能在指定目录中查找成功。

使用混合查找方式查找文件
find命令可以使用混合查找的方法，例如我们想在/tmp目录中查找大于 100000000 字节并且在 48 小时内修改的某个文件，我们可以使用-and 来把两个查找选项链接起来组合成一个混合的查找方式。
find /tmp -size +10000000c -and -mtime +2

学习过计算机语言的朋友都知道，在计算机语言里，使用and ,or 分别表示“与”和“或”的关系。在Linux系统的查找命令中一样通用。
还有这样的例子，
find / -user fred -or -user george

我们可以解释为在/tmp目录中查找属于fred或者george这两个用户的文件。
在find命令中还可以使用“非”的关系来查找文件，如果我们要在/tmp目录中查找所有不属于panda的文件，使用一个简单的
find /tmp ! -user panda
命令就可以解决了。很简单。

查找并显示文件未找到索引项。的方法

查找到某个文件是我们的目的，我们更想知道查找到的文件的详细信息和属性，如果我们采取现查找文件，在使用LS命令来查看文件信息是相当繁琐的，现在我们也可以把这两个命令结合起来使用。
find / -name "httpd.conf" -ls

系统查找到httpd.conf文件后立即在屏幕上显示httpd.conf文件信息。
12063 34 -rw-r--r-- 1 root root 33545 Dec 30 15:36 /etc/httpd/conf/httpd.conf

下面的表格就是一些常用的查找文件并显示文件信息的参数和使用方法

选项

用途描述

-exec command;

查找并执行命令

-fprint file

打印文件完整文件名

-fprint0 file

打印文件完整文件名包括空的文件

-fprintf file format

打印文件格式

-ok command;

给用户命令执行操作，根据用户的Y 确认输入执行

-printf format

打印文件格式

-ls

打印同种文件格式的文件.

总结：到这里为止我们已经学习了这名多关于find命令的使用方法，也列出了很多常用的find命令的选项，如果我们能熟练掌握在Linux中find命令的使用方法，那么在Linux中查找文件也不是一件困难的事情。