第1章 简 介

1.1 本书的作用

本书提供了使用C编程语言进行微控制器程序设计的一个完整的中级讨论，覆盖了设计嵌入式环境所需对C的改编，以及一个成功开发工程的通用组成部分的全部内容。

C是编写基于32位内核的较大微控制器（MCU）所选择的语言。这些微控制器通常由它们的通用微控制器衍生而来，并且同通用微控制器一样，既复杂又功能丰富。因此，对于这些MCU，C（和C++）编译器是必需的，也是很容易得到的。

相反，选择采用8位控制器的设计者经常求助于汇编语言的手工编码。虽然用于精确控制的手工汇编程序设计从来都不会过时，但也不会推动降低成本。因此，即使在8位MCU的有限资源里，编译高级C语言仍然有许多优势。

- 对如16位或更长的数据类型的算法之类的重复编码任务能自动生成代码。
- 硬件特殊性的直观处理。对一个串行闪存设备的读或写能用C语言表达为一个简单的赋值语句，尽管存储操作需要一些编码。
- 平台独立性。C带给桌面计算的跨平台能力对目前市场上的8位微控制器领域也是同样适用的。

本书将展示怎样用C语言编写8位嵌入式MCU程序。我们希望您不仅熟悉C，同时还具备有关微控制器程序设计更深层次的知识。

本书的主要样例工程是计算机控制的自动调温器。从一个最初规范开始，我们用与其他任何消费品或控制产品相同的方式逐步求精和增加设备。软件开发是我们关注的焦点，我们将做出任何设计者将要做出的选择和权衡。

1.2 嵌入式系统中使用C语言的好处

在嵌入式系统中使用C语言的好处如下：

不会困于细节。8位微控制器不仅仅是外形小：微控制器只包括执行其限定的任务所必需的逻辑，而程序员的付出是“轻松”的。通过一个C编译器与这些有限的资源协调工作，有助于抽象体系结构和避免坠入操作代码序列和硅片故障的云里雾里。

学习可移植性的原理。嵌入式应用程序是成本敏感的。为减少每个单元的成本，改变部件（或者甚至体系结构）可能是最大的激励。然而，修改汇编代码使得针对一种微处理器编写的程序能够在不同的处理器上运行，这样降低成本可能打消做出这种改变的任何念头。

通过传统编程降低成本。本书强调C语言代码能够概括微控制器特征。与特定硬件实施相关的细节能够被放置在不同的库函数和头文件中。使用C库函数和头文件可保证应用程序源代码能

够针对不同微控制器目标重新编译。

花费在算法设计上的时间更多而在实现上的时间更少。 C是一种高级语言，使用它能够快速而轻松地编写应用程序。C的表达范围是简明而强大的。因此，用C语言编写的每行代码能够代替多行汇编语言代码。调试和维护C语言代码将比汇编代码容易得多。

1.3 本书概览

确定软件开发目标是第1步，将在第2章中论述。它包括有关对高效软件开发至关重要的预设计文档规则的嵌入式注解。

第3章为以前没有涉及过8位微控制器的读者提供一个由浅入深的介绍。

有了一个好的计划和关于中央控制器的深入知识后，设计过程（在第4章论述）把以前的评估都最后确定下来。与实现自动调温器有关的处理器细节也在第4章介绍。

第5章详细描述了硬件的C语言表达。它汇集了编写程序源码所必需的所有设置。

第6章提供对嵌入式数据的深刻剖析。变量存储修饰符near和far在运行微软视窗的Intel PC上和在运行您的代码的嵌入式处理器上将代表不同的事物。

第7章讲述C语句，提供关于嵌入式的函数、语句和操作符的信息。

第8章介绍函数库。即使在只有很少ROM和有极其特定工作要做的环境里，预先编写的函数库也会带来很大的帮助。

第9章提供关于代码优化方面的深入知识，并帮助您彻底测试您创造的产品。

第10章总结了样例工程的更多信息。尽管某些信息已经在本章前面出现过，但它包括以前没有讨论的内容。

1.4 修改和补充信息

如果需要查找关于自动调温器工程的更多信息，请通过web咨询我们的补充信息：http://www.bytecraft.com/embedded_C/。

第2章 问 题 规 范

问题规范是设备和软件将要解决的问题的最初文档，它不应当包括任何具体设计难题或者产品解决方案，其主要的目的是详细解释程序将要做什么。

当然，正如在这个星球上有许多工作场所一样，处理工程计划的方法也有许多种。即使在那些最标准化了的阶段，也体现出不同的风格或者不同的次序。下面各节之所以被包含进来是因为它们增添了嵌入式领域的信息，有的则特别地适合样例工程。

2.1 产品需求

通常，这个文档是从用户的视角来写的，由一系列的用户需求组成。就针对一单任务设计的嵌入式系统的情况而言，应相当明确和肯定产品预期的功能范围。

有关硬件的总体决策是问题规范的组成部分，尤其在硬件将要受到充分控制的嵌入式工程中。

结果

- 程序将测量和显示当前的温度。
- 程序将按12或24小时制计数实际时间，并在一个数字显示屏上显示小时和分钟。
- 程序将接受时间设置并设置时钟。
- 程序将为三个日常使用周期接受和存储时间设置。
- 程序将控制制冷和制热之间的切换。注意，某些HVAC（heating ventilating and air conditioning，供热通风及空调）专家可能会想到偶尔在同一时间里既操作制冷又操作制热的需要，但这里的需求更接近于传统的自动调温器操作。
- 程序将比较当前温度与当前时间周期的设置，进而按需要打开或关闭外部的制冷或制热单元。
- 程序将抑制在一个短周期时间里两次改变外部单元的状态，这是为了使HVAC设备运行良好。
- 程序将在任何时间接受人工干预，进而立即关闭制热或制冷单元。

2.2 硬件管理

除了样例工程，本书并不直接涉及硬件。尽管如此，目标平台影响到产品的每个方面。它决定由编译器产生代码的容易程度，并且决定某些整体软件的设计决策。

如果软件开发者非常幸运，已经参与了硬件开发过程，那么影响设计的机会就太重要了且不要错过。请求的意向清单项目如下所示：

内置调试接口　现场可编程（field-programmability）的另一种方法也能满足需要。当一个设备必须在现场安装、定制或修补时，选用Flash-ROM芯片将比选用EEPROM或ROM设备更明智。

此为试读，需要完整PDF请访问：www.ertongbook.com

ROM代码保护　嵌入式处理器往往提供ROM代码保护，以防止偶然的检查。一个配置位禁止通过程序设计接口读ROM。尽管存在几个破坏这项保护的方法，但只有某个致命的对手才能够成功地读到您的程序。

合理的外设接口　电子线路设计时因贪图方便而影响I/O组织时，可能会迅速地降低软件性能。首选处理器有独立改变端口比特的位操作指令吗？多路复用接口将需要很多数据方向切换吗？

通过采用通用I/O端口线路和驱动器软件，某些外设能够被（软）复制。这节省了钱但增加了程序设计任务的复杂性。为模仿专用外设电路的逻辑信号，软件必须快速而重复地写比特序列到端口输出线路，这被典型地描述为比特风暴（bit-banging）。

标准函数库也许不会产生特别优化的硬件方案，但能够用减少的软件成本来补偿增加的硬件成本。

硬件设计的中心决策是处理器选择。处理器的选定是一个协商的决定，权衡的因素有：预期应用所必需的资源、芯片供应的价格和可用性以及可得到的开发工具等。针对微控制器的深入探讨请参见下一章。内存估计是问题规范的组成部分，因此RAM和ROM大小的估计在2.3.5节的资源管理中讨论。

结果

本书不涉及硬件管理，但本书包括了实现信息设置的硬件的某些样例产品规范信息。

表2-1　初始硬件规范

管理因素	评　估
操作环境	• 局部环境 • 中等功耗、中等噪声电子线路 • 偶尔断电
接口	• 一个针对切换HVAC的多位端口：也许只有3个针脚是必需的 • 一个针对显示屏的多位I/O接口 • 一个针对键盘的多位I/O接口 • 一个针对温度传感的A/D设备 • 实时时钟源：1秒粒度
内存大小	（参见下文）
特殊功能	• 时钟/计数器或者实时时钟 • NVRAM的使用依赖于处理器是否和怎样睡眠 • 监视定时器也许是有用的
开发工具	• C编译器 • 模拟器或仿真器 • 开发板

2.3 软件计划

软件计划应该对程序设计语言的选择做某些说明。对于嵌入式系统，通常的开发语言有三种选择：机器语言、C语言或者像BASIC那样的某种高级语言。三种选择当中，C平衡了两个对抗的需求。

- 比起像BASIC语言那样的解释系统， C有接近手工编码的机器语言的性能。如果一个BASIC系统通过暴露指针或通过预编译源代码变得更加复杂，则在测试时的困难将与C不相上下。
- C提供了机器语言没有提供的设备独立性。如果用汇编语言手工编码一个程序，则会遇到随微控制器的改变而废弃全部代码的风险。用C编程在改变处理器时只需要付出很小的努力，在软件模块里修改一个头文件即可。

软件计划中的第1步是选择解决在问题规范中所描述问题的算法。各种不同的算法应当被考虑到，并且依照它们的代码大小、速度、难度以及维护成本作出比较。

一旦一个基本的算法被选定，整个问题应该被分解为多个更小的问题。家庭自动调温器工程很自然地细分为对应于下面每个设备的模块以及每个设备的每一种函数：

- HVAC 接口
- 键盘
- LCD
- 温度传感器

从模块开始工作，可以写传统的伪代码。这有助于产生将要在代码中实现的标识符和逻辑段落。

从流程图着手把自然语言伪代码翻译为实际代码。在流程图里可以开始重点考虑函数将接受和提供的数据。最重要的是可以开始计划函数库的使用了。即使这里没有预写的外设或数据转换库存在，我们也能够以库的形式编写原始代码并在将来更加易于重用。

把不同的状态引入到计划中是可能的。状态框图映射了这种状态转换，它可以作为流程图的一种补充工具。

从伪代码开始，能够建立一个变量清单并对RAM和ROM的需要作出评估。内存资源的限制将对某些程序员是一个打击。用现代桌面环境工作的程序员在巨大的内存空间里很舒适，大量的RAM可用来创建可能永远也不会被初始化或用到的大数据结构或数组。

相反，微控制器只能在特定类目标应用所需要的那么多RAM和ROM里运转。制造商力求提供一系列类似的部件，每个变种的贡献只是增加少量的单片资源。

结果

2.3.1 软件体系结构

自动调温器设备的程序设计语言是C。主要体系结构的难以决断之处涉及采用中断还是轮询（或查询）。决断当中部分将随芯片选择而自动决定：一些处理器变种根本就不包括中断。其他选择包括对中断驱动的键盘的显式支持，或者在超时时产生中断的定时器。

一个基于中断方案的重要方面是在中断和主程序代码之间的通信协议。因为中断和主程序是尽可能独立的（一个中断可能在任何主程序指令期间出现），所以竞争条件是出现的一个结果。

我们已经选定几个替代算法中最简单的那个：一个时钟/计数器中断将计算时间，请求显示屏修改和设置目标温度。主程序将循环查询键盘，采样环境温度，修改显示屏和切换HVAC机械。这一切只需要一个精确的计时中断，它必须24小时不间断工作。

2.3.2 伪代码

伪代码用自然语言表达程序的必要步骤。它在嵌入式程序设计里显得特别有用，因为执行的每个方面能够一起规划，而没有必要考虑操作系统的古怪特性。

在下面的示例里，假定时间是用计数器和软件跟踪的。

1. 初始化

1) 设置时钟计数器为0。

2) 设置时间和温度目标变量为默认值。

3) 启用时间中断。

2. 时钟/计数器每秒触发中断一次。

1) 增加时钟计数器。

2) 请求显示屏修改。

3) 循环遍历预置周期。如果时钟正好在或过了指明的周期时间，设置目标温度为那个周期的温度。

3. 主循环

1) 采样环境温度。

(a) 如果环境温度在目标温度范围之外，打开制冷或制热开关。

(b) 如果环境温度在目标温度范围之内，关闭制冷或制热开关。

2) 写时间、环境温度和状态到LCD。

3) 等待按键

如果按键被按下，等待弹起周期并再次检查。

4)分析按键。

(a) 如果关闭命令被发送，则立即关闭操作单元。

(b) 如果周期选择命令被发送，则改换到下一周期记录。

(c) 如果时间设置命令被发送，则调整在当前周期记录里的时间。

(d) 如果温度设置命令被发送，则调整在当前周期记录里的温度。

2.3.3 流程图

图2-1基本上是嵌入式软件里主要和次要任务之间关系的一种表达。此流程图帮助决定：

- 什么功能出现在哪个逻辑模块里。
- 希望由函数库来支持哪些功能

也能够对流程图的重要结构给出标识符。

2.3.4 状态图

软件可能用来表达在处理外部交互或内部事件后在状态间转换的不同状态。图2-2显示了这些状态和使它们变化的相应激励条件。

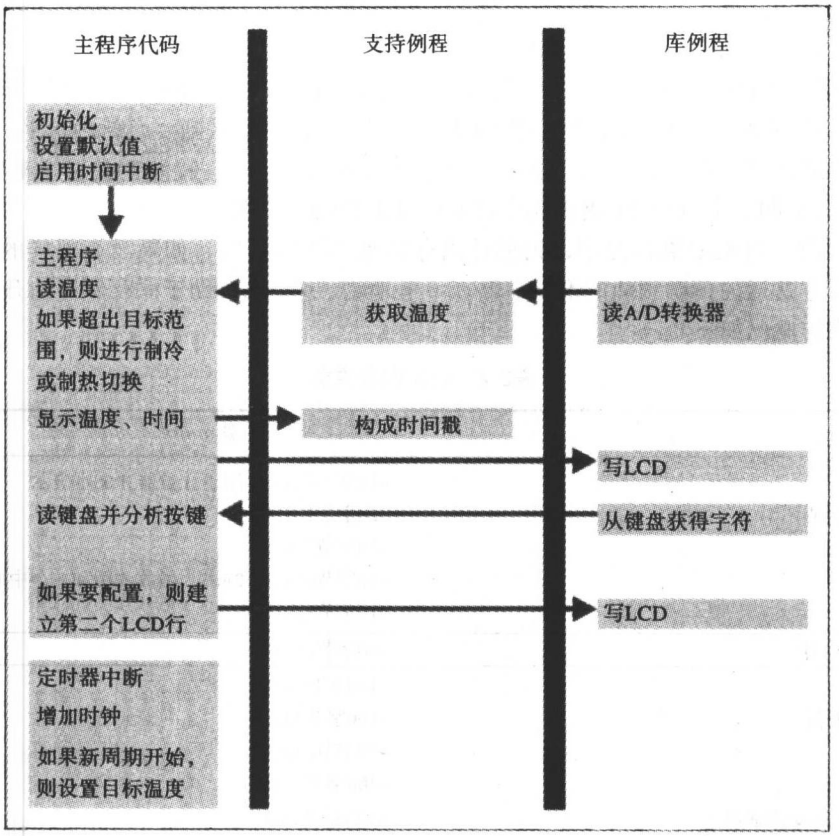

图2-1　关于算法的数据流程

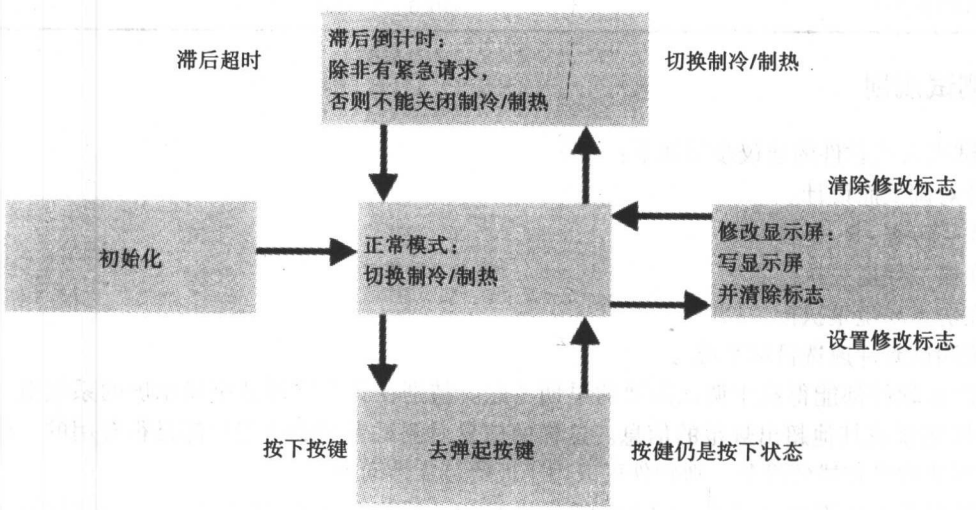

图2-2　关于算法的状态图

此为试读，需要完整PDF请访问：www.ertongbook.com

2.3.5 资源管理

在微控制器的受限环境里，一个多余的变量或常量能够改变被选择芯片的内存需求，进而影响到价格。像多语言支持这样的特征能够迅速地使资源需求膨胀到一个新的水平。

显式地规划出必需的资源是明智之举。这不是极其幼稚的——我们还在这里谈论通用变量，而不是像页面0访问、串行ROM 或者其他技术选择那样的具体细节。

如果您以前写过汇编语言程序，则估计内存需求会更容易些。如果没有那样的经验，写样本代码并编译它是通向精确预估的惟一道路。幸运地是，采用C有助于保存所有的开发成果。

一个粗略的概况如下。

表2-2 估计内存需求

变量/模块	资　源
实时时钟	~10字节RAM，用于计数器和文本表示
日常周期记录	~20字节RAM
用户设置	~10字节RAM
栈	~10字节RAM：2或3个函数调用和一个中断
局部变量	~10字节RAM
RAM估计合计	~60字节RAM
常量	~100字节ROM
中断服务例程	~100字节ROM
初始化	~50字节ROM
主程序	~300字节ROM
A/D转换（温度传感器）	~50字节ROM
LCD	~300字节ROM，依赖于接口类型而具有较大变化
键盘解码	~100字节ROM
ROM估计合计	~1000字节ROM

2.4 测试规划

调试嵌入式软件的建议步骤如下：

- 针对调试而设计。
- 代码审查。
- 在模拟环境里执行。
- 在仿真环境里执行。
- 按测试套件遴选目标系统。

硬件和软件都能得益于调试需要的早期考虑。特别在具有字母数字显示屏的系统里，软件能够通知错误或其他超出规范的信息。这样的信息对测试者和最终用户都是很有用的，但是如果市场不能容忍有错的设备，则软件可以用于证明可靠性问题。

在没有显示面板的情况下，LED能够发送有意义的状态或者事件信号。运行时诊断反馈信息、应该在伪代码和资源管理中体现。

调试的第1步，需要检查由编译器产生的汇编代码。在8位CPU上的嵌入式控制应用程序是足够小的，体系结构也足够简单，开发者能够很容易地审查产生的全部汇编语言代码。按行排列C源代码片段与它们产生的汇编代码的清单文件提供了最容易浏览的形式。

然而，经过这一步之后，测试变成一个挑战：当正测试中的代码实现了机器的最基本行为时，深入系统里的调试变得更加困难。一个故障可能彻底阻止来自嵌入式系统的任何有意义的响应，而桌面操作系统却能够提供内核卸载或其他的诊断帮助。

为使深入系统里的调试成为可能，要将模拟器和仿真器与嵌入式系统相匹配。每种工具尝试模拟目标环境的不同部分，从而可以彻底和容易地检测软件性能。在不需要或不关心硬件的条件下，纯软件的模拟器最适合用于检查算法性能和正确性。在真实世界里，仿真器更加集中于I/O和内部的外围设备操作。您将需要访问至少一个仿真器。因为工具选择是与硬件设计过程和处理器选择紧密联系的，所以对它做了讨论。

最后，在测试套件里放入一个原型设备能提供工作软件的最准确证明。

结果

我们的设计将有一个LCD面板。由于这项能力，系统能够写调试消息到显示屏。这些消息可能包括打开电源时的一个回应按键或者显示状态信息的“耀眼屏幕”。

编译器必须支持调试。产生的汇编代码必须是可用于检查的。

选择的产品应该为模拟器执行源代码级的调试提供方便，能匹配当前正执行的机器代码与原始C代码。对于自动调温器而言，模拟速度不是一个关键因素，惟一的时间依赖的功能是实时时钟。

测试套件由一个灯泡和风扇组成，它们由控制器切换开关并指向热敏电阻。这是一个最简单有效的方案。

第3章 微 控 制 器

本章回顾微控制器特征并概述在8位微控制器市场里的可能选择。某些特征可能在中央处理器里看到过的，如图形增强或浮点支持等，在嵌入式系统里是不存在的。

计算机硬件设计最令人注意和具有超凡魅力的部分是中央处理单元的选择。在桌面世界里，处理器选择围绕着同 Intel x86产品线的兼容性展开：与Intel兼容的处理器，接近兼容的处理器和完全不兼容的处理器。

在嵌入式世界里，没有这样的连续性，特别在谈论一个新的设计时。 8位控制器市场竞争非常激烈，主要是因为它在容量上的焦点。它通常没有商标名称识别，且消费品生产厂商不愿意让用户知道技术细节。如果用户关心驱动他们产品的芯片，他们则可能要去了解超越产品预期应用的新应用领域。

8位微控制器不像32位处理器那样是程序员友好的。32位处理器高度优化的体系结构的后天增强，如额外的ROM地址空间，能够快速超出8位体系结构的限制。这反过来推动处理器设计者在组装机里增加如存储体切换或者寻址限制等技术以作为补偿。

最后，如体系结构的预期寿命等因素也应该考虑到。首选处理器达到它的产品生命周期的末期的时候，采用产生设备程序的C编译器可以减少不断改换处理器的成本。

8位微控制器具有计算机的所有传统功能部件。

中央处理单元（CPU）微控制器的算法和逻辑单元针对存在于这么小的体系结构里的有限资源受到限制和优化。多路复用和分路操作非常罕见，并且浮点也不存在。寻址模式有时受到令人恼怒的限制。

ROM和RAM 8位微控制器很少有ROM和RAM寻址超过16线的（64 Kb）。如果某个芯片封装暴露所有地址或数据总线，则它们提供的寻址空间只有几千字节。 MCU （微控制器单元）包含少量的内部RAM和ROM阵列。因为程序化单个芯片的需要， ROM往往像电子可编程（或电子可擦写）内存一样常见。

定时器 有两种常见类型：计数器和监视定时器。简单的计数器能够对一个时钟周期或者输入信号作出响应。在到达一个零点或者一预置的阈值时，它们能触发一个中断。

中断电路 在通用微处理器有多个通用化的中断输入或者级别的场合，一个微控制器有多个专用于特殊任务的中断信号：计数器超时，或者在某个输入针脚上的信号变化。

那就是说，上述情况是在控制器根本有中断的条件下；如果预期的应用足够简单而不需要它们，就不能保证设计者将一定包含它们。

输入与输出 大多数芯片提供能够切换外围设备的一些I/O线。偶然地，这些针脚能够减弱大电流以减少外部组件。某些变种提供A/D和D/A转换器或者驱动某些设备（如红外LED）的特殊逻辑。

外设总线 并行外设总线将减少“单片机”的优势，因此它们的应用是受到妨碍的。因为速

度在嵌入式系统设计里不是位于前列的功能要素，几个有竞争力的串行外设总线标准已经开发出来。仅仅使用1~3根线，这些总线可使外围设备芯片，例如ROM，与微控制器交互而不独占现有的接口线路。

微控制器的微小尺寸的主要后果是，它的资源相对于桌面个人计算机按比例限制了。尽管具备了计算机的所有特征——RAM、ROM、I/O以及微处理器——但是，开发者不能指望具有其上有8个并行位的I/O端口。

在决定首选处理器之前，必须考虑针对目标可得到的外部开发工具。一个嵌入式系统不像一台PC机那样是自运行的。为开发嵌入式软件，开发工具必须运行在一个桌面计算机上，并至少使用某些非常特殊的硬件。

3.1 中央处理单元

各种不同微控制器中，寄存器的数量和名称可能不同。它们有时出现在某个内存地址空间里，有时又完全各自独立。尽管名称可能不同，但是某些寄存器对大部分微控制器是通用的。

- 累加器
- 变址寄存器
- 栈指针
- 程序计数器
- 处理器状态寄存器

用C直接访问累加器和变址寄存器只能偶尔令人满意。 C的register数据类型修饰符等于向一个寄存器发出的要直接访问的“请求”：如果编译器不能令人满意地做到这点的话，实际上则不使用寄存器。

然而，当使用寄存器是最合适的或者是必须的时候，另一类型声明能够链接变量名和一个寄存器本身。Byte Craft编译器提供了registera类型（以及其他寄存器的相应类型）。给一个registera变量赋值产生进入到累加器寄存器的一个数据装载，但是不产生到内存的一个存储。对该标识符的求值返回在寄存器里的值，而不是来自内存的值。

```
reg stera important_variable = 0x55 ;
```

直接访问栈指针或程序计数器是不太明智的。采用C语言的目的是从直接机器语言引用中抽象程序逻辑。函数调用和循环将使设备依赖的栈操作和分支操作平衡，是结构化编程的最好方法。如果有必要，可以对带标号的目标使用C关键词goto：编译器将插入合适的跳转指令，并且，最重要地是，自动地考虑任何分页或设置问题。

3.1.1 指令集

期待出现在通用微处理器单元MPU（microprocessor unit）里的机器指令有：乘法、除法、表查询或者乘法-加法，而在8位处理器里的相应指令并不总是出现在控制器家族的每个变种里。

一个#pragma语句能够通知编译器目标芯片确实有一个可选指令功能，因此编译器能够利用该指令提供的便利条件而优化代码。这样的例子出现在MC68HC05C8的头文件中。

此为试读，需要完整PDF请访问：www.ertongbook.com

清单3-1 指令集配置

```
#pragma has MUL,
#pragma has WAIT,
#pragma has STOP,
```

312 栈

如果处理器在通用内存里支持栈，则记录该栈所必需的空间是从RAM分配的，否则该空间用于全局变量。不是所有的栈都记录在主（或者数据）内存： Microchip PIC和Scenix SX体系结构使用在用户RAM外的栈空间。

检查由函数调用和中断而存储的返回信息的深度是很重要的。编译器可能报告栈溢出（意味着栈太小了），但是栈声明也可能比必需的要大。

除了声明一个内存区域作为栈的预留区外，没有任何其他需要担心的了。考虑下面Motorola MC68HC705C8栈。该栈从地址00C0到00FF有64字节。

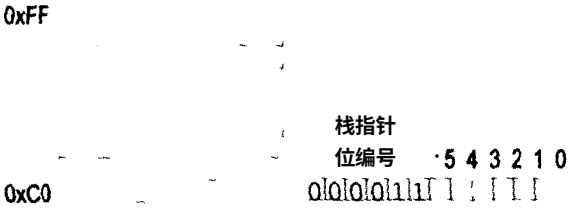

图3-1 MC68HC705C8栈

在C中，必须做如下声明：

```
#pragma memory stack [0x40] @0xFF;
```

因为栈大小和配置将是随处理器族的不同而该改变的（或者甚至在相同族里的变种之间也是不同的），该声明使编译器精确地意识到有多少可以得到的空间。如果不需要64字节，则可减少其大小，把0x40变为更小的数即可。

编译器能够提供函数调用的更深层次的信息，参见9.6节的CALLMAP选项。

3.2 内存寻址和类型

多数小的微控制器几乎不提供RAM。对于桌面应用程序员而言，由完全消耗掉RAM或者ROM引发的那种惧怕的感觉是新奇的。除了对失败的内存分配进行粗略的检查，程序员依靠成兆的RAM和交换文件几乎能够完全避免内存不足的错误。

C编译器支持重用内存，无论什么情况下都可能支持。编译器为了在多个局部范围里重用内存，不断判定哪个时间里哪些位置是空闲的。当然，“空闲”意味着只有被下一个函数调用重新初始化，它才是被该子例程可读的。

因为内存问题，某些经典的程序设计技术超出了8位微控制器的能力。重入的或者递归函

数，即那些在桌面系统程序设计里的精华，是假定有足够的栈空间，但在嵌入式环境里却是不可能的。

3.2 1 RAM和 ROM

RAM和ROM在微控制器里是永久性地被划分的。它们可能是不同地址空间的部分。

控制器的所有存储空间减去RAM或者ROM所占有的空间，余下的是未实现的内存地址空间部分。指令取或读或写到这些区域能够导致不可预测的或者错误的结果。

声明可用的RAM和ROM，指示编译器哪里能够安全地放置程序代码或者数据。 Byte Craft编译器需要所有的内存资源都被声明。这些声明能够简单地声明可得到的内存类型、大小和位置，或者可选地给区域指派一个符号名称。

命名地址空间在优化处理之外带给程序员一些控制。如果处理器具有更快访问部分内存的能力（例如，在680x上的page 0），并且头脑中有一个特别的规划，那么声明将存留于那个内存区域的变量。

清单3-2 在命名地址空间里声明

```
#pragma memory ROM [0x4000] @ 0xA000;
#pragma memory RAM page0 [0xFF] @ 0x00;
#pragma memory RAM page1 [0xFF] @ 0x100;

/*  .. */

/* my_variable将出现在page0。如果处理器有访问page0的特殊指令。编译器将产生它们以
 便赋值和后续引用。 */

int page0 my_variable = 0x55;
```

3.2.2 ROM和程序设计

在制作一个封装的ROM产品之前，可编程ROM或PROM开始是作为一个昂贵的工具来原型化和测试应用代码。近几年来，PROM已相当流行，许多开发者考虑将它作为大规模生产芯片中封装ROM的出众替代品。

随着微控制器应用变得越来越个性化和越来越复杂，开始需要对它们进行维护和支持。许多开发者采用PROM设备为客户提供软件修改，而免除了发布新硬件的成本。

可编程ROM的分类在下面描述。

Fused ROM 是传统的PROM，具有已编程的ROM单元，采用的方法是在一个存储体阵列（即ROM单元）里，按照比特模式有选择地烧断熔丝。可编程只有在外部设备支持下才能实现。

EPROM（可擦写可编程ROM）是非易失性的并且是只读的。它必须通过暴露在紫外线里来擦除。

EEPROM （电子可擦写可编程ROM ）设备有超过EPROM设备的重要优势，它们允许有选

择地擦写内存段。EEPROM 最平常的使用是记录和维护对应用至关重要的配置数据。例如，调制解调器采用EEPROM存储体记录当前配置设置。

闪存是EEPROM和EPROM技术之间的一个经济的折中选择。产品有一个基于ROM 的配置内核，以及写入闪存的应用程序代码。当想要为客户提供附加功能或者维护升级时，硬件能够在现场重新编程而不用安装新的物理芯片。硬件已被置入到配置模式，配置模式传递控制到写入ROM的内核。然后，该内核处理为删除和重写闪存内容所必需的软件步骤。

依赖于目标芯片，EEPROM和闪存在程序控制下是可编程的。电子必须等待电荷传输并慢速工作以免设备过热，因此，这种编程过程要占用一些时间。

3.2.3 冯·诺依曼与哈佛体系结构

冯·诺依曼体系结构有一个单一的公用的内存空间，程序指令和数据都存放在这个空间里。提取指令和数据是通过一个单一的内部数据总线进行的。

哈佛体系结构计算机针对程序指令和数据有分离的内存空间。有两个或更多的数据总线，这就允许同时访问指令和数据。CPU利用程序内存总线获取程序指令。

程序员不必关心他们针对什么体系结构编写程序。C编译器能够补偿他们各自缺点和特异的大部分。以下是某些更通用的特征的解释，用以分析由编译器产生的代码。

- 冯·诺依曼体系结构的机器的代码生成常利用了处理器能够执行RAM外的程序这样一个事实。某些数据类型的操作可能实际上预先准备了与操作代码相关的RAM位置，然后针对它们（RAM地址）进行分支操作！
- 因为哈佛体系结构机器有针对数据的明确内存空间，所以为数据存储而使用程序内存是更加复杂的。例如，一个声明为C常量的数据值必须作为一个常量值存储在ROM。一些芯片有允许从程序内存空间获取信息的特殊指令，这些指令总是比从数据内存获取数据的等价指令更复杂或成本更高。其他简单芯片没有这些指令，数据必须被装载，例如，通过一条返回指令的副作用完成。

3.3 定时器

定时器是按照固定速率的时钟脉冲增加或减少的计数器。通常，一个固定时间间隔引发一个中断：定时器计数到0，溢出到0，或者达到一个目标计数。

在微控制器里，定时器是标志产品竞争性的特征。提高了精度和智能的定时器或者计时单元是很容易得到的。目前，定时器的不同类型给工程师带来了许多选择的空间。

编码预定标器并启动时钟是软件程序员的任务。只要知道处理器时钟频率，并选择了正确的预置数值，程序员就能够得到正确的定时器时钟周期。

程序员与定时器的接口是几个命名控制寄存器，用`#pragma port`语句声明并作为变量读或者写。

如果定时器中断得到支持，则它能够用一个`#pragma vector`语句声明，进而通过一个作为函数编写的关联的中断服务例程来得到服务。

清单3-3 定时器寄存器和中断处理程序

```
#pragma portr TIMER_LSB @ 0x24;
#pragma portr TIMER_MSB @ 0x25,

#pragma vector TIMER_IRQ @ 0xFFE0;

void TIMER_IRQ(void) {
        /*  IRQ处理程序代码  */
}
```

3.3.1 监视定时器

COP （计算机正常操作）或者监视定时器检查代码执行是否失控。通常条件下，监视定时器必须在重置后的头几个循环周期里被打开一次。然后，在执行期间，软件必须周期性地重置监视。

如果处理器执行离开了正常轨道，监视不可能被可靠地重置。这正是那种需要被修补的状态：间接跳转到一个意想不到的地址可能是其原因，发送从来不会被收到的外部信号的循环也是一个可能的原因。

监视超时能够使处理器进入到一个已知的状态，通常是RESET状态，或者执行一个中断。监视定时器的硬件实现在不同处理器间变化相当大。某些监视定时器能够针对不同超时延时编程。

在C语言里，重置监视的序列同赋值给一个端口一样简单。

清单3-4 重置监视定时器

```
#pragma portw WATCHDOG @ 0x26;
#define RESET_WATCHDOG() WATCHDOG = 0xFF

void main(void) {
  while( ) {
    /* . . */
    RESE _WATCHDOG();
  }
}
```

3.3.2 实例

下面是一些样例配置：

- National Semiconductor 公司的 COP8SAA7有一个称作T1的16位定时器，一个称作T0的16位空闲定时器，以及一个监视定时器。空闲定时器T0帮助维护空闲模式期间的实时和低功耗。定时器T1用于与3种用户可选的模式相关的实时控制任务。

- Motorola公司的MC68HC705C8 有一个16位计数器和一个COP监视定时器。COP监视定时器是用户启用的，提供可选择的超时周期，并且可以用两条向 COPCR寄存器写的指令来重置它。有意思的是， COP监视是依赖于系统时钟的。如果时钟停止，则时钟监视电路重置MCU，从而致使COP监视无用。
- Microchip公司的PIC17C42a有4个定时器模块，分别称作TMR0、TMR1、TMR2和TMR3，以及一个监视定时器。TMR0是一个具有可编程预置数值的16位定时器，TMR1和TMR2是8位定时器，TMR3是16位定时器。

3.4 中断电路

微控制器通常提供硬件（信号）中断源，有时提供软件（指令）源。在有有限针脚数的封装里，IRQ信号可能没有被暴露或者可能是与其他I/O信号多路复用的。

能够被禁止的中断是可屏蔽中断，不能被禁止的中断是非屏蔽中断。例如， RESET是非屏蔽中断。不管当前正执行的代码，CPU必须立即服务于RESET中断。

中断信号是异步的：它们是能够发生在一个指令周期期间、之后或之前的事件。处理器能够使用两种方法之一确认中断：同步或者异步确认。

大部分处理器同步确认中断：它们在处理中断之前处理当前的指令。相反，异步确认的处理器停止当前指令的执行转而服务于该中断。虽然异步确认比同步确认更及时，但它带来了中断代码干扰已进行中的指令的可能性。

例如，一个中断例程要修改一个主程序代码正读取的各字节数值。如果主程序代码在一个多字节提取操作里读该值时被半道中断，则已装载的值在没有任何提示下变成无意义的值。

在中断和主程序代码之间单程读和写变量的代码应遵从我们的建议（参见4.4.2节）。为提供完整性保护，编译器需要采用不可分割指令，或者临时禁止中断，以保护主程序代码。

同步确认也不是完美的方案。当一个多字节操作需要几个指令时，以上同样的问题也会影响同步确认的处理器。

3.4.1 向量和非向量仲裁

微处理器服务于中断有两种竞争的方法。向量仲裁需要指向中断服务例程（ISR）的指针表。非向量仲裁期待ISR的第一条指令位于一个预定义的入口点。多数8位微控制器采用向量仲裁中断。

当编译器为ISR产生代码时，它把开始地址放到ROM映像里特定的中断向量里，或者重定位ROM里入口点处的代码。编译器也可能自动产生仲裁代码：在评估ROM使用时，记住检查它。

当一个中断发生时，处理器将禁止中断以防止服务例程被其自身中断。然后，一个向量机器读取包含在特定中断向量里的地址，它跳转到那个地址并开始执行ISR代码。

相反，一个非向量系统简单地跳转到已知的起始位置并执行那里的代码。为了实现优先级，ISR可能不得不按顺序测试每个中断源，或者简单跳转到ISR驻留的主体所在的不同位置。

因为在非向量系统里需要额外的处理，所以向量中断更快一些。总之，非向量ISR在少于5个中断的微控制器里是可行的。

表3-1显示了8位微控制器的主流族的仲裁方法。

表3-1 中断仲裁方法

体系结构	仲 裁	注 释
Motorola 6805/08	向量	向量位于支持内存的顶端
National COP8	混合	参见本表之后的文字
Microchip PIC	非向量	某些模式没有中断，而有些为中断组提供向量派遣
Zilog Z8	向量	必须设置优先级
Scenix SX	非向量	无优先级
Intel 8051	非向量	每个中断跳到一个不同的 固定的ISR入口占
Cypress M8	非向量	对于每个中断，处理器跳到一个不同的、固定的ISR入口点。它们被称作“向量”且有两字节长。为正确跳转到ISR，在那些位置必须有一条JMP指令

National Semiconductor公司的COP8使用了一种混合方法，所有中断按照非向量方式转移到一个公共位置。在那个位置，代码要么执行VIS指令，该指令在多个活动中断源里仲裁并跳转到从向量表里得到的地址处；要么显式地投送中断条件给系统并按照用户定义的方式处理它。后一种方法也许是有用的，但是也有许多缺点。

表3-2显示了COP8向量表，它是COP8SAA7设备所必需的。排列顺序是VIS指令的实施顺序。

表3-2 COP8向量 中断

序 号	中 断源	描 述	向量地址①
1	软件	INTR 指令	0bFE 0bFF
2	保留	为将来保留	0bFC - 0bFD
3	外部	G0	0bFA 0bFB
4	定时器 T0	下溢	0bF8 - 0bF9
5	定时器 T1	T1A/下溢	0bF6 0bF7
6	定时器 T1	T1B	0bF4 - 0bF5
7	MICROW-IRE/PLUS	BUSY Low	0bF2 0bF3
8	保留	为将来保留	0bF0 - 0bF1
9	保留	为将来保留	0bEE 0bEF
10	保留	为将来保留	0bEC - 0bED
11	保留	为将来保留	0bEA - 0bEB
12	保留	为将来保留	0bE8 - 0bE9
13	保留	为将来保留	0bE6 - 0bE7
14	保留	为将来保留	0bE4 - 0bE5
15	端口 L/唤醒	端口 L边界	0bE2 - 0bE3
16	默认	无中断时，执行VIS指令	0bE0 - 0bE1

① b代表.VIS块。VIS和向量表必须在相同的256字节的块里。如果.VIS是一个块的最后地址，则该表必须在下一块里

3.4.2 中断期间保存状态

对所有芯片而言，中断处理都保存机器的一个最小处理器状态，通常为当前程序计数器。

此为试读，需要完整PDF请访问：www.ertongbook.com