Programming Embedded Systems in C and C++

C/C++

嵌入式系统编程

O'REILLY®

中国电力出版社

Michael Barr 著

于志宏 译

C/C++嵌入式系统编程

Micbael Barr 著

于志宏 译

作者简介

Michael Barr 是 **Netrino** 公司（一个嵌入式系统共享软件和软件工程服务提供商）的创始人兼总裁。**Netrino** 公司鼓励所有职员通过为杂志撰稿和在业界会议演讲来分享自己的专业知识。这些资料可以在公司的网站 **http://www.netrino.com** 找到。

Michael 拥有马里兰大学的电机工程学士和硕士学位。他的大部分时间都用在嵌入式软件、设备驱动和实时操作系统的开发上了。他还喜欢写作、教书，并期待着开始下一部著作的创作。目前他有好几个计划，其中包括一部小说。

前言

首先需要弄清楚，你为什么希望你的学生学习某个主题，
以及你希望他们学到什么，那么一般来说，
你授课的方法或多或少就有了。

－Richard Feynman

今天，几乎所有电子设备里面部包含了嵌人式的软件系统。这些软件隐藏在我们的手表里、录像机里、蜂窝电话里，甚至可能在烤面包机里面。军事上会使用嵌入式软件来引导导弹。侦测敌方的飞行物。外太空探测器和许多医疗仪器离开嵌人式软件几乎不可能工作。

设计人员不得不写所有的代码，实际上，成千上万的电子工程师 计算机科学家和其他专业人员正在这样做。我也是其中的一员，从我的个人经验来说，我很清楚掌握这门技术是多么的困难。学校军从未开设有关嵌入式系统的课程。而我也没能从哪个图书馆里找到一本有关这个题目的像样的书。

每一个嵌入式系统都是独特的，其硬件部分对它的应用目标来说是高度专用的。这就导致了嵌入式系统编程的涉及面很广，而且可能会需要很多年才能掌握它。不过，几乎所有的嵌入式软件开发都使用了 C 语言。这本书就是要教你怎样在嵌入式系统中使用 C 和 C 的派生语言，C++。

即使你已经知道如问编写嵌入式软件，你还是可以从这本书里学到很多东西。除了了解如何更有效地使用 C 和 C++你还将会从本书中对常见的嵌人式软件问题的详细解释，并从本书所提供的源代码中得到益处。本书中包含的高级主题有存储器检测和验证、设备驱动程序的设计和实现．实时操作系统的内部机理，还有代码优化技术。

我为什么写这本书

我曾经听到一个统计数字，在美国，平均下来大概每个人拥有八个微处理器。我当时很惊讶，怎么可能呢？难道我们周围真的有这么多计算机吗？后来．当我有更多时间来想这个问题的时候，我开始把我用过的并且可能含有一个微处理器的东西逐一列出来。短短三分钟内，我的清单已经包含了十样物品了 它们是：电视机、录音机、咖啡机、报时闹钟、录像机、微波炉、洗碗机、遥控器、烤面包机、还有数字式手表。这还只是我的个人物品——我很快就可以拿出我工作中用到的另外十样东西。

进一步的发现是很自然的。那些产品里的每一个都不仅仅包含一个处理器。还有软件在里面。最终，我知道在我一生里我想做些什么了。我希望能用我的编程技能来开发这种嵌入式的什算机系统。但是我如问能得到必要的知识呢？当时我正在该大学的最后一年，而学校里迄今为止没有关干嵌入式系统编程的课程。

幸运的是、虽然我那时还处在学习的过程中 但当我毕业的时候我还是找到了一家公司，从事编写嵌人式软件的工作。不过在这里我必须要靠自己的努力，因为为数不多的了解嵌人式软件的几个人通常都非常的忙，以至于很少有时间来解答我的问题，所以找到处找能给我教益的书、最后。才发现我必须自学所有的东西因为我从没有找到这么一本书，并且我很奇怪为什么会没有人来写这么一本书。

现在我决定自己来写这样一本书了。在此过程中 我也发现了为什么以前没有人做这件事。关于这个题目最困难的是，决定什么时候可以收笔封稿了。每一个嵌八式系统都是独一无二的，并且就我所知，每一条法则同时都会存在例外倩况。不过，我已经尝试着提取出这个主题的本质的东西，并且仅仅讲述嵌人式系统程序员们必须要了解的那些部分。

面向的读者

这是一本关于使用 C 和 C++来进行嵌人式系统编程的书。同样，这里假定读者已经有了一些编程经验，并且至少熟悉这两种语言的语法。如果你比较熟

悉基本的数据结构例如链表等，也会有些帮助。这本书并不要求你在计算机硬件方面了解很多，但是希望你愿意由这本书而学一点有关硬件的知识。这毕竟是一个嵌入式程序员工作的一部分。

写这本书的时候，在我的脑海里有两类读者。第一类是初学者——正像我刚从大学毕业的时候那样。她会何一些计算机科学或工程的背景，并有几年编程经验。初学者感兴趣的是如何为一个既有的设备写嵌人式程序，却不能肯定该如问着手去做。看完前五章后，她就能够用她的编程技术来开发简单的嵌入式程序了。本书的其他部分可以作为她在以后的职业生涯里遇到更高级的主题时的参考。

第二类读者已经是嵌入式系统程序员了。她熟悉嵌入式硬件，并目知道怎样来为此编写软件。但是她正在寻找一本参考书来解释一些关键问题。出许这位嵌入式系统用序员一直在用汇编语言编程，并且刚接触 C 和 C++不久。这样的话，这本书会教给她如问在嵌入式系统里使用这些语言。后面的章节还会提供她所需要的更高级的材料。

不论你是否属丁上述两种读者之一，我还是希望这本书能够以一种友好和方便的形式给你一些帮助。

本书的组织

本书包括十章、一个俘虏、一个词汇表，还有一个带注释的参考书目列表。这十章恰好可以分为两个部分。第一部分包含第一到第五章，主要面向嵌人式系统的初学者。这些章节应该按照它们出现的次序完整地读一下，这将快速地带给你有关嵌入式软件开发的基础知识。结束了第五章之后 你就可以独立开发一些小的嵌入式软件了。

第二部分包括第六到第十章，讨论了不论有没有经验的嵌入式程序员都很感兴趣的一些高级主题。这些章节基本上各自独立，可以按照随意的次序来读。另外，第六到第九章包含的示例程序可能会对你将来的嵌人式系统项目有所帮助。

- 第一章“引言”。介绍嵌入式系统。其中定义了若干术语，给出了一些例子并且说明了为什么选择 C 和 C++来作为本书的编程语言。

- 第二章“你的第一个嵌人式程序”。引导你尝试用 C 语言编写一个简单的嵌入式程序的全过程。这比较类似于其他很多编程书籍里的“Hello，World”的例子。
- 第三章“编译、链接和定址”。介绍了一些软件工具。你将用它们来为一个嵌人式处理器生成可执行文件。
- 第四章“下载和调试”。介绍将可执行程序调人一个嵌入式系统的各种技术手段，同时也描述了你可以使用的调试工具和技术。
- 第五章“接触硬件”。描述了学习一个不熟悉的硬件平台的简单过程。结束本章后，你已经能够书写和调试简单的嵌人式程序了。
- 第六章“存储器”。讲解了关于嵌人式系统内的存储器作所需要知道的全部知识。这一章还包括了存储器测试和闪速存储器驱动程序的源代码实现。
- 第七章“外围设备”。说明了设备驱动程序的设计和实现技术，同时包含了一个通用外围设备（定时器）的示范驱动程序。
- 第八章“操作系统”。包含了一个可以用在任何嵌入式系统中的很简单的操作系统。这有助于你决定你是否需要这么一个操作系统，如果需要的话，是买一个还是干脆自己写一个。
- 第九章“合成一个整体’。进一步拓展前面章节学到的关于设备驱动程序和操作系统的知识。本章讲解了如何控制更复杂的外设，同时引入了一个完整的示范应用来把你学过的东西综合到一起。
- 第十章“优化你的代码”。描述了如何在增加代码运行速度的同时，减少你的嵌入式软件对存储器的需求。这包括使用一些技巧来刊用最有效的 C++ 特性，而不导致显著的性能损失。

在整本书里，我一直在努力在特定的例子和通用的知识之间保持平衡，也就是尽可能地消除微小的细节，使这本书更加易读。像我一样，通过阅读示例你会从这本书里得到最大的收获，但是应该只把它们作为理解重要概念的工具。记住不要的在任问一个电路板或芯片的细节里面。在理解了全面的概念以后，你将能够把它应用在你所碰到的任何嵌人式系统中。

在排版和其他方面的约定

本书使用了如下的一些印刷约定：

斜体{*italic*}

当文件、函数、程序、方法、例程和选项出现在段落中的时候，用来表示它们的名字。斜体也用来强调或引人新的术语。

等宽(constant width)

用来显示文件的内容和命令的输出。在段落体中．这种字体用来表示关键字、变量名、类、对象、参数和其他代码片断。

等宽祖体**(constant width bold)**

用来在示例里表示你输人的命令和选项。

其他约定是和性别与角色有关的。关于性别，我有意在全书区分使用了“他”和“她”。“他”代表奇数章节而“她”代表偶数章节。

关于角色我偶尔会在我的讨论中区对一下硬件工程师、嵌人式软件工程师和应用程序员的不同任务。但是这些称谓只是工程师个体的角色，需要注意到的是一个人充当多个角色是常有的事。

在线取得示例

这本书包含很多示例源计码，除了最小的单行代码以外都可以在线获得。这些示例按照章节来组织、并包含了 **build** 指令（**makefile**）来帮助你重建每个可执行文件、完整的文件可以通过 **FTP** 得到，在 **ftp://ftp.oreilly.com/examples/nutshell/embedded_c/**。

建议与评论

我们已尽全力保证本书内容的正确性，但你仍可能发现有些内容不对（甚至可能是我们出了错误!）。你的建议将帮助我们使下一版更加完美，请告诉我们你找到的错误以及你的建议，写信到：

美国：

O'Reilly & Associates,Inc.
101 Morris Street
Sebastopol.CA 95472

中国：

100080 北京市海淀区知春路 **49** 号希格玛公寓 **B** 座 **809** 室
奥莱理软件（北京）有限公司

询问技术问题或对本书的评论，请发电子邮件到：

info@mail.oreilly.com.cn

最后，你可以在 **WWW** 找到我们：

http://www.oreilly.com
http://www.oreilly.com.cn

个人说明和致谢

我曾经很想写一两本书，但是现在我这么做了以后，我必须承认我开始的时候非常的天真。我甚至不知道要做多少工作、另外还会有多少人被牵扯进来。不过令我吃惊的是，找一个愿意出版我的书的出版社是如此容易，我原以为这会是很困难的一件事。

从提议写书到出版，这个计划花了两年才完成。这是因为我一直在全时工作并且希望尽可能保持我的社会生活所致。要是我早知道我会熬夜到很晚来为最后的底稿而苦恼的话，我也许会放弃我的工作来早点交付这本书。但过继续工作对这本书很有好处（同时对我的银行账户也有好处）。这使我有机会对以和很多嵌入式硬件和软件的专家进行广泛的讨论他们巾的很多人通过审阅部分成全部

的书稿为这本书做出了直接的贡献。

我非常感或以下诸位，与我分享了他们的知识并且一直在帮助我的工作：**Tony Bennet**、**Paul Cabler**（和其他来自 **Arcom** 的很棒的人们），**Mike Corish**、**Kevin D'Souza**、**Don Davis**、**Steve Edwards**、**Mike Ficco**、**Barbara Flanngan**、**Jack Ganssle**、**Stephen Harpster**（他在看完早期书稿后管我叫“断句王”）、**Jonathan Harris**、**Jim Jesen**、**Mark Kohler**、**Andy Kollegger**、**Jeff Mallory**、**Ian Miller**、**Henry Neugauss**、**Chris Schanck**、**Brian Silverman**、**John Snyder**、**Jason Steinhorn**（正是他的流畅的语法素养和技术批评的眼光使这个书值得一读）、**Ian Taylar**、**Lindsey Wereen**、**Jeff Whipple** 和 **Greg Young**。

我还要感谢我的编辑 **Andy Oram**。要是没有对我最初建议的巨大热情、超乎寻常的耐心和持续的鼓励，这本书将永远不会完成。

最后，我要感谢 **Apla Dharla**，感谢她在这个漫长的过程中给予我的支持和鼓励。

Michael Barr
mbarr@netrino.com

本章内容:

- 什么是嵌入式系统
- 各种实现间的差异
- C: 最基本的必需品
- 关于硬件的一些说明

第一章

引言

我想全世界计算机市场也许会有五台。

——*Thomas Watson（托马斯·沃森），IBM 公司主席，1943*

没有人会想在家里放一台计算机。

——*Ken Olson（肯·奥尔森），DEC 公司总裁，1977*

最近几十年里最令人惊讶的事，莫过于计算机逐渐占据了人类生活的主要地位。今天在我们的家里和办公室里，计算机的数量要比使用它们来生活和工作的人还要多，只是这些计算机里有很大一部分我们没有意识到它们的存在罢了。在这一章里，我将说明什么是嵌人式系统，以及可以在哪里找到它们。同附会介绍一下嵌人式编程的主题，说明一下为什么本书采用 C 和 C++语言讲述，另外简单介绍一下示例中所用到的硬件环境。

什么是嵌入式系统

一个嵌入式系统（*embedded system*）就是一个计算机硬件和软件的集合体，也许还包括其他一些机械部件，它是为完成某种特定的功能而设计的。一个很好的例子就是微波炉。几乎每个家庭都有一台，并且每天都有上千万台微波炉在被人们使用着，但是很少有人意识到有处理器和软件在帮助他们做饭。

这和家里的个人计算机形成了鲜明的对比。同样是由计算机硬件和软件，还有机械部件（比如硬盘）组成的，个人计算机却不是用来完成某个特定功能的。相反它可以做各种不同的事情。很多人用通用计算机（*general-purpose computer*）来区分这一点。在发货的时候，通用计算机就像一块没有字的黑板，制造商并不知道用户要拿它来做什么。一个用户可能会用它来做文件服务器。另一个只用来玩游戏，还有一位可能会用它来写下一部伟大的美国小说。

而嵌入式系统常常是一些更大的系统中的一个组成部分。比如，现代的轿车或卡车里就包含了很多嵌人式系统。一个嵌人式系统会被用来控制防刹车锁死，另一个监控车辆的气体排放情况，还有一个用来在仪表板上显示信息。虽然不是必需的，但在某些情况下，这些嵌人式系统会通过某种通信网络互相连起来。

为了不至于混淆你的思路，有必要指出，通用计算机本身就是由很多嵌入式系统组成的。比如，我的电脑包含了键盘、鼠标、显示卡、调制解调器、硬盘、软盘和声卡，它们中的每一样都是一个嵌入式系统。每个设备都包含处理器和相应的软件来完成特定的功能。比如凋制解调器就是用来在模拟电话线上收发数字信号用的。正是如此，所有其他的设备也都能归纳出这么一句话来。

如果一个嵌入式系统设计得很完善，那么它的使用者完全可以忽略它内部的处理器和软件的存在。微波炉、录像机和报时闹钟就是很好的例子。在某些情况下，用同样的功能的定制集成电路硬件来代替上面所说的处理器和软件，也能做出具有同样功能的设备来。不过，如果真是这样用纯粹的硬件来设计的话，在灵活性上就会丧失不少了，改几行软件怎么说也要比重新设计一块硬件电路来得方便和便宜。

过去和将来

本章开头定义的嵌人式系统的第一个产品直到 **1971** 年以后才出现。这一年，**Intel** 发布了世界上第一块微处理器，**4004**，主要被日本的 **Busicom** 公司用来生产商用计算器。**1969** 年，**Busicom** 请 **Intel** 为他们的每一种新式计算器分别设计一种定制的集成电路，**Intel** 则拿出了 **4004**。**Intel** 没有为每一种计算器分别进行设计，而是设计了一种可以用在所有型号上的通用电路。这个通用处理器被设计来读取存在外部存储芯片里的一系列指令（软件）。**Intel** 的想法是通过软件的设计可以为每一种计算器提供各自的特性。

这种微处理器在一夜之间就成功了，并且在以后的十年中获得了广泛的应用。早期的嵌入式应用包括无人空间探测器、计算机控制的交通信号灯以及航空灯光控制系统。在整个 **80** 年代，嵌人式系统静悄悄地统治着微处理器时代，并把微处理器带人了我们个人和职业生活的每一个角落。装有嵌人式系统的电子设备已经充斥了我们的厨房（烤面包机、食物处理机、微波炉）、卧室（电视、音响、遥控器）和工作场所（传真机、寻呼机、激光打印机、点钞机和信用卡读卡机）。

嵌入式系统的数量看起来肯定会继续迅速增长。已经有很多具有巨大市场潜力的新的嵌入式设备了：可以被中央计算机控制的调光器和恒温器、当小孩子或矮个子的人在的时候不会充气的智能气囊、掌上电子记事簿和个人数字助理（**PDA**）、数码照相机和仪表导航系统。很明显，掌握一定技能并且愿意从事下一代嵌入式系统设计的人将会获得很多的机会。

实时系统

现在很有必要介绍一下嵌入式系统的一个子集。按照通常的定义，实时系统（***real-time system***）就是有一定时间约束的计算机系统。换句话说，实时系统可以部分地从及时完成计算或判断的能力来辨别。这些重要的计算有完成的明确期限，并且，对实际应用来说，一个延期的反应就像一个错误的结果一样糟糕。

如果一旦延期会产生什么结果，是至关重要的问题。例如，如果一个实时系统是飞机飞行控制系统的一部分，那么一个延期的计算就可能会使乘客和机组人员的生命受到威胁。而把这个系统用在卫星通信环境下，危害也许可以限制在仅仅一个损坏的数据包。在更严格的情况下，很可能这个时间期限是“硬性”需求的，也就是说，这个系统是个“硬”实时系统，和它对应的就有“软”实时系统了。

本书中所有的主题和示例都可以应用到实时系统中。不过 一个实时系统的设计者必须更加细心，他必须保证软件和硬件在所有可能的情况下都能可靠工作。同时，根据人们生活对该系统可靠执行的依赖程度，这种保证一定要有工程计算和描述性的论文加以支持。

各种实现间的差异

与为通用计算机设计的软件不同，嵌人式软件通常无法在不做显著修改的情况下在其他嵌入式系统中运行。这主要是由底层硬件之间的明显不同所致。每个嵌人式系统的硬件都是为特定的应用专门调整过的，这样才能使系统的成本保持很低。所以，不必要的电路就被省去了，硬件资源也尽可能地共享使用。在这一节里你会学到哪些硬件特性是所有嵌人式系统共有的以及其他方面为什么又会有如此多的不同之处。

通过定义我们知道所有的嵌入式系统都包含处理器和软件，那么还有哪些特性是它们共有的呢？当然，要想执行软件，就一定要有存储执行代码的地方和管理运行时数据的临时存储区，这就分别要用到 **ROM** 和 **RAM**；任何嵌入式系统都会有一些存储区。如果只要求很少的存储量，也许就使用与处理器在同一芯片里的存储器，否则就需要使用外部存储芯片来实现。

所有嵌人式系统都包含其种输入和输出。例如，一个微波炉的输人就是前面板上的按钮和温度探测器，输出就是人可阅读的显示信息和微波射线。嵌入式系统的输出几乎总是它的输入和其他一些因素的函数、包括花费的时间、当前的温度等等。输入常见的形式有传感器和探测器。通信信号或物理世界的某些变化。图 **1-1** 给出了嵌入式系统的一个常见的例子。

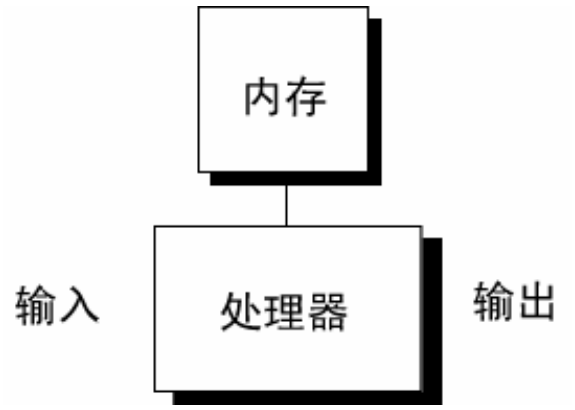

图 1-1　一个基本的嵌入式系统

除了上述几个共同点，嵌入式系统的其他部分通常是互不相同的。实现之间的差异是由不同的设计侧重导致的。每个系统都是面向完全不同的一整套需求，这些需求的折中考虑直接影响了产品的开发过程。例如，如果一个系统要求成本低于 **10** 美元,那么就有可能要牺牲一些处理性能或可靠性才能达到要求。

当然，生产成本只是嵌入式硬件开发人员需要考虑的一个可能的限制而已。其他要考虑的设计需要还包括：

处理能力

要完成目标所需的运算能力。一个常用来衡量运算能力的指标是 **MIPS**（以百万计算的每秒可执行的指令数量）。如果两个处理器的指标分别是 **25MIPS** 和 **40MIPS**，那么就说后者的运算能力更强一些。但是，还需要考虑处理器的其他一些重要特性。其中之一是寄存器字长，一般会是 **8** 到 **64** 位。现在的通用计算

机一般使用 32 位或 64 位的处理器，但是嵌入式系统通常仍使用更老、更便宜的 8 位和 16 位处理器。

存储器

用来保存执行代码和操作数据的存储器的容量。硬件设计人员必须事先做出估计并且在软件开发完成之后增加或减少实际的容量。存储容量也会影响处理器的选择，通常寄存器的字长构成了处理器可存取的存储容量的限制，例如，一个 8 位的寻址寄存器可以确定 256 个存储位置之一（注 1）。

开发费用

硬件软件开发过程所需的费用。这是一个确定的、一次性的花费，所以这也许无关紧要（通常对于大批量产品），也许需要仔细衡量（在只生产少量产品的情况下）。

批量

生产费用和开发费用的折中考虑主要由期望的生产批量和销量所决定。例如，通常不会选择为一个小批量产品开发自己的专用硬件模块。

预计的生命周期

系统必须延续多久（平均估算）？一个月、一年、或者十年？这影响到从硬件的选择到开发和生产费用方面的各种设计决策。

可靠性

最终产品应具有什么程度的可靠性？如果只是一个儿童玩具，那么不需要总是工作正常，但是如果是航天飞机或小轿车的一部分，那就最好在任何时间都要工作正常。

除了这些常见的要求之外，系统还有自己详细的功能要求。正是这些要求赋予了嵌人式系统不同的特性，比如微波炉、起搏器或寻呼机。

注 1：当然，寄存器的字长越小，处理器就更可能需要采取一些策略，如多个地址空间，以支持更大的内存。几百字节是不足以做太多事情的。即使对 8 位处理器而言，几千个字节也可能只是最低要求。

表 1-1 说明了前面谈到的设计要求的可能的取值范围。这些只是估计数字，并不需要严格采用。在某些情况下，几个标准是联系在一起的。比如，处理能力的增加也会导致产品成本的增加。同时，我们也可以设想同样是增加处理能力也会通过减少硬件和软件设计的复杂性来降低开发成本。所以每一列的数值并不是一定要同时满足。

表 1-1　嵌入式系统常见的设计需求

分类	低	中	高
处理器	4 或 8 位	16 位	32 或 64 位
存储器	<16KB	64KB－1MB	>1MB
开发费用	<$100000	$100,000～$1,000,000	>$1000000
生产成本	<$10	$10～$1000	>$1000
批量	<100	100～10000	>10000
预计的生命周期	几日、几周或几月	几年	十年
可靠性	可以偶尔故障	必须可靠工作	必须无故障运行

为了同时说明两个嵌人式系统之间的差异，以及这些设计需求对开发过程的影响，我会比较详细地介绍三个嵌入式系统。我的想法是在具体讨论嵌人式软件开发之前，先从系统设计人员的角度考虑一下问题。

数字手表

计时工具从日◎、滴漏、沙漏一路发展而来就是数字化手表。它的特性包括显示日期和时间（通常精确到秒），以百分秒计时，还有在每个整点发出烦人的响声。正如它所表现的那样，这些都是非常简单的功能，并不需要很多的处理能力或存储器。实际上，采用处理器的唯一原因，只是为了使硬件设计可以支持一系列的型号。

典型的数字表包含一片简单、便宜的 8 位处理器。因为这种处理器不能寻址较多的存储器，所以这类处理器一般都自带了片上 ROM。如果有足够的寄存器的话，那这个产品连 RAM 也用不着了。实际上，所有电子部件——处理器、存储器、计数器和实时时钟，几乎都做在同一个芯片上，这块表还剩下的硬件就

包括输人（按钮）和输出（LCD 或扬声器）了。

数字手表的设计者的目标是用超低的生产成本来提供一个相对可靠的产品。如果在生产后发现部分手表比其他大多数要更精确些，那么这些手表就会被冠以某个品牌以更高的价格出售。或者也可以通过折扣分销渠道来获得利润。对于低价品种，则可以把停止按钮和杨声器去掉。这虽然会失去一些功能却几乎不需要改动软件。当然，所有这些开发的花费可能会相当高，但随着成千上万只表卖出去，收入会源源不断地增加。

视频游戏机

当你从娱乐中心取出任天堂 Nintendo-64 或者 SONY PlayStation（PS）的时候，你就将要使用一个嵌人式的系统。有时候这些机器比同级别的个人计算机的性能还要好，不过面向家用市场的视频游戏机同个人计算机比起来还是要便宜一些。正是高的处理能力和低的生产成本这两个相抵触的要求，使得视频游戏机的设计师们经常熬夜工作（当然他们的孩子可就过得不错喽）。

只要最终产品的生产成本能比较低——一般在 100 美元左右。生产视频游戏机的公司一般不去关心系统的开发费用。他们甚至鼓励他们的工程师们设计专用的处理器，因而每一次开发费用都比较高昂。所以，尽管在你的视频游戏机里会有一个 64 位的处理器，它和一个 64 位个人计算机里的处理器可不一定是一样的。一般来说，这个处理器是专门用来满足它要运行的视频游戏的要求的。

因为在家用视频游戏市场上生产成本是如此重要，设计人员也会用一些手段来分摊成本。比如，一个常用的技巧是尽可能把存储器和其他外围电路从主电路板上挪到游戏上。这样就会在降低游戏机的成本同时增加了每一个游戏的价格。这样，一个系统也许会配备一个强劲的 64 位处理器但主板却只带了几兆内存。这些内存只够启动机器让它可以存取游戏卡上的存储器。

火星探测器

1976 年，两个无人飞船抵达火星。它们的任务是采集火星表面的岩石样本，

并在分析其化学成分后把结果传回给地球上的科学家们。那个“海盗船”的任务使我感到颇为吃惊。因为我现在被一些几乎每天都要重新启动的个人计算机包围着，所以我发现对多年前的这些科学家和工程师真是很伟大，他们成功地设计了两台计算机并且使它们在五年里经过了 **3400** 万英里的旅程依然工作正常。很明显，在这些系统中，可靠性是最重要的要求。

如果存储芯片损坏或者软件存在缺陷以至于导致运行崩溃，或者一个电连接在碰撞之下断开，结果会如何呢？根本没有办法防止这些问题的发生。所以必须通过增加冗余电路或额外的功能来消除这些隐患：使用额外的处理器、特殊的存储器检验、当软件死锁后用一个硬件定时器来复位系统等等，各种手段，不一而足。

最近，美国宇航局启动了“探路者”计划，主要的目标就是论证一下以有限的预算到达火星的可行性。当然，随着 **70** 年代中期以来技术的极大发展，设计者并不需要为这个目标费太多脑筋了。他们可以在给予“探路者”比“海盗船”更强大的处理能力和更多的存储量的同时，减少相当一部分冗余设计。“火星探路者”实际包含两个嵌入式系统：着陆艇和漫游车。着陆艇有一个 **32** 位处理器和 **128MB** 的 **RAM**；漫游车只有一个 **8** 位处理器和 **512KB** 的存储量。这种选择也许反映出了两个系统不同的功能需求的考虑，不过我可以保证生产成本不是问题。

C：最基本的必需品

这些系统下多的几个共同点之一是都使用了 **C** 语言。和其他语言相比，**C** 已经成为嵌人式程序员的语言了，情况当然不全总是这样，事情总会变的。不过，起码现在 **C** 是嵌入式世界里最接近标准的东西。这一节里，我会说明为什么 **C** 会变得如此普遍，我又为什么选择 **C** 和 **C++**作为这本书的主要语言。

因为对于一个给定的项目来说，选择一种语言对成功的开发是如此的重要，所以当一种语言被证明同时适合于 **8** 位和 **64** 位处理器，适用于字节、千字节甚至兆字节的系统，适用于从一个人到很多人的开发团队。是很令人吃惊的。而 **C** 语言做到了。

当然，**C** 是有很多优势的。它小而易学，今天每一种处理器都有 **C** 的编译器，

同时有相当多的有经验的 C 程序员。另外，C 是和处理器无关的，这就让程序员可以着眼于算法和应用而不用考虑特定处理器结构的细节。可是，很多其他的高级语言也具备这些优点，为什么只有 C 语言取得了成功呢？

也许 C 语言最具威力的地方——也正是把它和其他语言比如 Pascal 和 FORTRAN 区别开的地方——是，它是一个非常“低级”的高级语言。正如我们将在整本书里看到的，C 给予嵌入式程序员很大程度的直接控制硬件的能力，却不会失去高级语言带来的好处。“低级”的内在本质是这个语言的创建者的明显目的。实际上。Kernighan 和 Ritchie 在他们的书《C Programming Language》的开头有这么一段话：

> C 是一种相对“低级”的语言。这个特征并没有什么不好的含义；它只是说明 C 语言可以处理大多数计算机可以处理的事情。这些事情通常和实际机器实现的技学和逻辑运算结合在一起。

很少有其他高级语言可以像 C 一样，为几乎所有处理器生成紧凑的、高效的代码。同时，只有 C 允许程序员方便地和底层硬件打交道。

其他嵌入式语言

当然 C 并不是嵌人式程序员使用的唯一语言。至少还有其他三种值得详细说一下．即汇编语言、C++语言和 Ada 语言。

在早期的时候，嵌人式软件只用目标处理器的汇编语言来书写。这样做使程序员可以完全控制处理器和其他硬件，当然也是有代价的。除了更高的软件开发费用和缺乏可移植性，汇编语言还有很多缺点，同时，最近几年找一个有经验的汇编语言程序员也变得越来越难。汇编语言现在只用作高级语言的附件，通常只用在那些必须要求极高效率或非常紧凑，或其他方式无法编写的小段代码里面。

C++是 C 语言的面向对象的超集，正在嵌入式程序员中变得越来越流行。它的核心语言特性和 C 完全一样，但是 C++提供了更好的数据抽象和面向对象形式的编程功能。这些新的特性对软件开发人员非常有帮助，但是部分特性会降低可执行程序的性能，所以 C++在大的开发队伍里用的最为普遍，在那里只

程序员的帮助要比程序效率的损失更为重要。

Ada 也是一种面向对象的语言。不过和 C++完全不同。Ada 开始是美国国防部为了开发面向任务的军用软件而设计的。尽管它曾两次被接纳为国际标准（Ada 83 和 Ada 95），但 Ada 从没有在防务和航空工业领域之外获得足够的应用。即使是这些领地这几年也在逐渐丧失，这是很不幸的事，因为与 C++比起来 Ada 有很多特性可以简化嵌人式软件的开发工作。

为这本书选择一种语言

类似本书的同类书的作者面临的主要问题是采用哪一种语言来开展讨论。同时使用太多的语言只会使读者犯晕或者偏离更重要的问题。另一方面，着眼点太窄又会使讨论变得不必要的学术化，或者（对作者和出版商都很糟糕）限制了这本书的潜在市场。

很明显，C 是所有关于嵌入式编程的书的核心，这本书也不例外。超过一半的例子是用 C 编写的，同时讨论也主要集中在和 C 有关的编程问题上。当然，所有关于 C 编程的问题同样适用于 C++。另外，我会在后面的例子里使用那些对嵌人式软件开发最有用的 C++特性。汇编语言在特定的环境下会加以讨论，但是会尽量避免。换句话说，我只在用别的方法无法完成一个特定的编程任务时，才会考虑用汇编语言。

我觉得这种混合使用 C、C++和汇编语言的安排方式，更能反映现在的嵌入式软件开发过程，并且在不久的将来还会是这样。我希望这种选择会使讨论能比较清晰，可以提供给开发实际系统的人有用的信息，并尽可能地适合更多的潜在的读者。

关于硬件的一些说明

关于编程的书籍必须要给出实际的例子。通常，这些例子要能很容易地被感兴趣的读者试验。这就是说读者必须可以接触和作者完全一样的软件开发了具和硬件平台。很不幸，在嵌入式编程的情况下，这是不现实的。在大多数读者

的平台上，比如 **PC**、**Mac** 和 **Unix** 工作站上来运行任何示范程序都是没意义的。

即使要选择一个标准的嵌入式平台地是很困难的。正如你已经知道的，沿有“典型的”嵌人式系统这么一种东西。不管选了哪种硬件，大多数读者都没办法接触到。但是尽管有这个相当重要的问题．我还是觉得选择一个参考平台来使用示例是很重要的。通过这样做，我希望可以使所有的例子保持一致性，以此来使整个讨论更加清楚。

为了只使用一个硬件来说明尽可能多的问题，我发现有必要选择一个中档的平台。这个硬件包含一个 **16** 位处理器（**Intel** 的 **80188EB**，注 **2**）、适量的存储器（**128KB** 的 **RAM** 和 **256KB** 的 **ROM**），还有一些常见的输入、输出和外设部件。我选用的电路板是 **Arcom** 控制系统公司制造的 **Target188EB**。关于这块电路板和如何获取的信息可以参看附录“**Arcom** 的 **Target188EB**”。

如果你可以接触到这个参考硬件的话。你将能原封不动地使用本书里的例子。否则，你需要把示例代码移植到你能用到的嵌人式平台上面。为了这个目的，我尽可能地使示例程序易于移植。可是读者必须要知道，每一种嵌入式系统的硬件都是不一样的，可能一些例子对地的硬件来说一点意义也没有，比如，把第六章“存储器”里提到的快闪存储器驱动程序，移植到一个不带闪存的板子上就很没意义。

不管怎样，在第五章“接触硬件”里面我还会讲很多东西。但是首先我们还有很多软件问题需要讨论，这就开始吧。

注 2：Intel 的 80188EB 处理器是专门为嵌入式系统修改了设计的 80186 的特殊版本，原来的 80I86 是 IBM 的第一台个人计算机（PC/XT）使用的 8086 处理器的一个继承者。它从来没有被实际使用。因为当 IBM 设计下一个型号（PC/AT）的时候选择的是 80286。尽管早期是失败的，近几年来自 Intel 和 AMD 的 80186 却在嵌入式系统里面取得了巨大的成功。

第二章

你的第一个嵌入式程序

本章内容:

- Hello World!
- 闪烁程序
- 无限循环的作用

注意！此机器不能摸也不能拿。它的内部在飞速地转动，
而且不断发出火花。它不是傻瓜摆弄的玩意儿。请把手
放在口袋里，站得远远地，放松些，看那闪烁的火花。
——Ken Olson（肯·奥尔森），DECC 公司总裁，1977

在这一章里我们将通过一个例子直接进入嵌入式编程。这个例子看起来和其他大多数编程书籍开头的“**Hello，world!**”例子差不多。在讨论代码的时候，我会说明选择特定代码段的理由，并会指出依赖目标硬件的部分。本章只包含这第一个程序的源码，在接下来的两章里我们会讨论如何创建可执行代码并运行它。

Hello World!

好像所有讲述编程的书都用同一个例子来开始，就是在用户的屏幕上显示出“**Hello，World!**”。总是使用这个例子可能有一点叫人厌烦，可是它确实可以帮助读者迅速地接触到在编程环境中书写简单程序时的简便方法和可能的困难。就这个意义来说，“**Hello，World!**”可以作为检验编程语言和计算机平台的一个基准。

不幸的是，如果按照这个标准来说，嵌入式系统可能是程序员工作中碰到的最难的计算机平台了。甚至在某些嵌入式系统中，根本无法实现“**Hello，World!**”程序。即使在那些可以实现这个程序的嵌入式系统里面，文本字符串的输出也更像是目标的一部分而不是开始的一部分。

你看，“Hello，World!”示例隐含的假设，就是有一个可以打印字符串的输出设备。通常使用的是用户显示器上的一个窗口来完成这个功能。但是大多数的嵌入式系统并没有一个显示器或者类似的输出设备。即使是对那些有显示器的系统，通常也需要用一小段嵌入式程序，通过调用显示驱动程序来实现这个功能。这对一个嵌入式编程者来说绝对是一个相当具有挑战性的开端。

看起来我们还是最好以一个小的，容易实现并且高度可移植的联人式程序来开始，这样的程序也不太会有编程错误。归根到底，我这本书继续选用“Hello，World!”。这个例子的原因是，实现这个程序实在太简单了。这起码在读者的程序第一次就运行不起来的时候，会去掉一个可能的原因，即：错误不是因为代码里的缺陷：相反，问题出在开发工具或者创建可执行程序的过程里面。

嵌人式程序员在很大程度上必须要依靠自己的力量来工作。在开始一个新项目的时候，除了他所熟悉的编程语言的语法，他必须首先假定什么东西都没有运转起来，甚至连标准库都没有，就是类似 printf()和 scanf()的那些程序员常常依赖的辅助函数。实际上，库例程常常作为编程语言的基本语法出现。可是这部分标准很难支持所有可能的计算平台，并且常常被嵌入式系统编译器的制造商们所忽略。

所以在这一章里你实际上将找不到一个真正的”Hello，World!”程序，相反，我们假定在第一个例子中只可以使用最基本的 C 语言语法。随着本书的进一步深人，我们会逐步向我们的指令系统里添加 C++的语法、标准库例程和一个等效的字符输出设备。然后，在第九章“综合所学的知识”里面。我们才最终实现一个“Hello，World!”程序。到那时候你将顺利地走上成为一个嵌入式系统编程专家的道路。

闪烁程序（译注 1）

在我的职业生涯中所进到的嵌入式系统都至少有一个可以被软件控制的

注 1：当然，闪烁的频率的选择完全是任意的、我选择 1Hz 的原因是这可以根容易地用一个秒表来核对。简单地启动秒表，计几次闪烁，燃后停下秒表看嵌闪烁的次数是不是和经过的秒数相同，如果需要更精确的话，简单地多计几次闪烁就行了。

译注 1：原文为德语。

LED（发光二极管）。所以我用一个以 1Hz（注 1）频率闪烁 LED（发光二极管）的程序来替代“Hello，World!”。1Hz 就是每秒完整地开关一次。典型的情况是，用来开关一个 LED 的代码通常只有几行 C 或汇编代码，所以发生错误的机会也就很少。同时因为几平所有的嵌入式系统都有 LED，所以潜在的概念是非常容易移植的。

LED 闪烁程序的高层部分如下所示。这部分程序是与硬件无关的。不过，它还要依赖分别使用和硬件有关的 toggleLed()和 delay()来改变 LED 的状态和控制计时。

```
/*****************************************************************/
* Function main()
* Description: Blink the green LED once a second
* Notes: This outer loop is hardware-independent. However.
*        it depends on two hardware-dependent functions.
* Returns: This routine contains an infinite loop.
/*****************************************************************/
void
main(void)
{
    while(1)
    {
        toggleLed(LED_GREEN);  /*Change the state of the LED.*/
        delay(500);            /*Pause for 500 millisenconds.*/
    }
}   /*main()*/
```

toggleLed

在 Arcom 的电路板上，有两个 LED：一红一绿。每个 LED 的状态都被一个叫做端口 2I/O 锁存寄存器（缩写是 P2LTCH）的一个位来控制。这个寄存器和 CPU 在同一个芯片里，它实际上包含了芯片外围的 8 个 I/O 引脚的锁存状态。这 8 个引脚合在一起叫做端口 2。P2LTCH 寄存器里的每一位都和相应的 I/O 引脚的电压联系到一起。比如，第 6 位控制送到绿色 LED 的电压：

```
    #define LED_GREEN 0X40   /* The green LED is controlled by bit 6.*/
```

通过修改这一位，就可以改变相应外部引脚的电压从而改变了绿色 LED 的状态。如图 2-1 所示，当 P2LTCH 的第 6 位是 1 的时候 LED 关，第 6 位是 0 则 LED 打开。

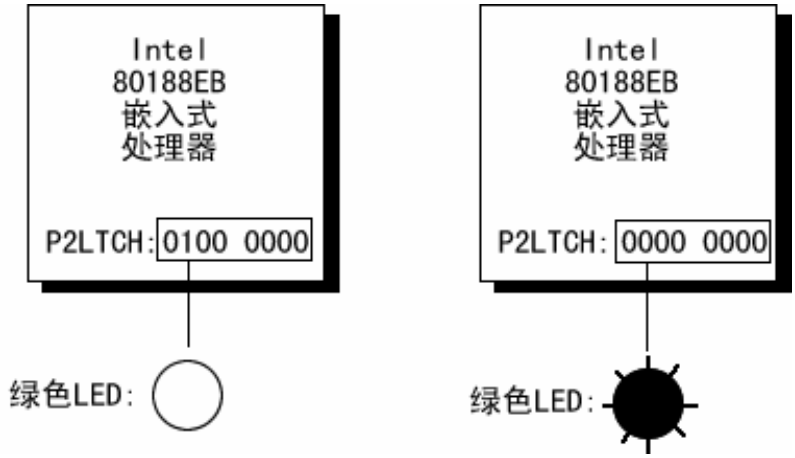

图 2-1 Arcom 电路板上的 LED

P2LTCH 寄存器位于 **I/O** 空间的一块特定内存里，偏移为 **OxFF5E**。不幸的是，**80x86** 处理器的 **I/O** 空间里的寄存器只能使用汇编语言指令 **in** 和 **out** 来操作。**C** 语言没有内嵌的类似操作。最接近的替换函数是定义在面向 **PC** 平台的头文件 ***dos.h*** 里的 ***inport()***和 ***outport()***。理想情况下，我们可以包含这个头文件并从我们的嵌人式程序里调用这两个库函数。不过，因为它们是 **DOS** 编程库的一部分，我们必须要考虑到最坏的情况：它们在我们的系统上不工作。最起码的是，我们在第一个程序里不应该依赖它们。

下面列出了面向 **Arcom** 电路板并且不依赖库例程的 **toggleLed** 例程的实现。实际的算法是很简单的：读 **P2LTCH** 寄存器的内容，切换要控制的 **LED** 的相应位，再把新的值写回寄存器。你会注意到尽管这个例程是用 **C** 书写的，而实际的控制部分是用汇编语言实现的。这种简便的方法叫内嵌汇编语言（**inline assembly**）。它一方面使程序员避开了复杂的 **C** 函数调用和参数的传递和转换过程，同时使她可以随意地使用汇编语言来工作（注 2）。

```
#define P2LTCH 0xFE5E  /*The offset the P2LTCH regiser.*/
```

注 2：不幸的是，各种编译器的内嵌汇编语法是不一样的。我在示例中使用的是 Borland C++编译器的格式。Borland 的内嵌汇编格式非常好，它主持在汇编行里引用用 C 代码定义的变量和常数。

```
/*****************************************************************/
* Function toggleLed()
* Description: Toggle the state of one or both LEDs.
* Notes: This function is specific to Arcom's Target188EB board.
* Returns: None defined.
/*****************************************************************/
void
toggleLed(unsigned char ledMask)
{
    asm {
        mov dx, P2LTCH  /*Load the address of the register.*/
        in  al, dx      /*Read the content of the register.*/
        mov ah, ledMask /*Move the ledMask into a register.*/
        xor a1, ah      /*Toggle the requested bits.*/

        out dx, al      /*Write the new register contents.*/
    }
}   /*toggleLed()*/
```

delay()

我们也需要在切换 **LED** 的动作之间实现一个半秒（**500ms**）的延时。这是通过在如下所示的 ***delay*** 例程里使用忙等待技术实现的。这个例程接受以毫秒计的参数作为请求的延迟时间，然后用这个参数和常数 **CYCLES_PRE_MS** 相乘来得到为了延迟制定时间需要的 **while** 循环重复次数。

```
/*****************************************************************
* Function delay()
* Description: Busy-wait for the requested number of milliseconds.
* Notes: The number of decrement-and-test cycles per millisecond
*        was determined through trial and error. This value is
*        dependent upon the processor type and speed.
* Returns: None defined.
/*****************************************************************
```

```
void
delay(unsigned int nMilliseconds)
{
    #define CYCLES_PER_MS 260 /*Number of decrement-and-test cycles.*/
    unsigned long nCycles = nMilliseconds * CYCLES_PER_MS;
    while(--nCycles);
    }
}   /*delay()*/
```

与硬件相关的常数 **CYCLES_PER_MS**，代表了处理器在 **1** 毫秒里可以执行的“减测试”**(nCycles-- != 0)**周期的次数。为了确定这个值我使用了尝试和排错的方法。我做了一个大概的估算（我想可能在 **200** 左右），然后写程序的其余部分，编译并运行。**LED** 确实闪了，不过频率要比 **1Hz** 快，然后我用一个精确的秒表来对 **CYCLES_PER_MS** 作了一系列小的调整直到闪烁的频率很接近 **1Hz** 为止。

就是这样，这就是闪烁 **LED** 程序的所有内容了。有三个函数来完成整个工作： ***main()***、***toggleLed()***和 ***delay()***。如果你想把这个程序移植到别的嵌人式系统的话，你应仔细阅读你的硬件的文档，必要时重写 ***toggleLed()***，并修改 **CYCLES_PER_MS** 的值。当然，我们还得创建和执行这个程序，我会在下两章里讲这些。但是首先，我得花一点时间说说无限循环和它在嵌入式系统里的作用。

无限循环的作用

在为嵌入式系统和其他计算机平台写程序时有一个最基本的区别，就是嵌人式程序总是以一个无限循环作为结束。典型地，这个无限循坏包含了程序功能的一个重要组成部分，就像在闪烁 **LED** 程序里那样。无限循环是必要的，因为嵌入式软件的工作永不结束。它一般要运行到世界末日到来或者电路板复位。

另外，大多数嵌入式系统只运行一块程序。并且尽管硬件是重要的，可是没有了嵌入式软件它怎么也成不了一个数字手表、蜂窝电话或者微波炉。如果软件停止运行了，那硬件也就没用了。所以一个嵌入式程序的功能体总是被一个无限循环来包含着以使它们可以永远运行下去。

这种情形是如此普遍以至于不值一提。但是我不会这样，因为我曾经看到相当多的嵌入式程序员新手们对这个微妙的区别感到很困惑。所以如果你的第一个程序看起来运行了，可是 **LED** 没有闪烁而是就改变了一次状态，那也许就是你忘记了把对 ***toggleLed()***和 ***delay()***的调用包在一个无限循环里面。

本章内容:

- Hello World!
- 闪烁程序
- 无限循环的作用

第三章

编译、链接和定址

如果我喜欢一个程序那么我就应该和同样喜欢它的人分享它，
我认为这是一条黄金法则。软件销售商们意图分化和征服用户，
使每一个用户同意不和别人共享软件。我拒绝用这种方式打破其他用户的团结。
我不认为签署一份软件授权许可协议或者不公开协议是有良知的事。
因此我能够继续光荣地使用电脑。我决定要汇集足够的自由软件，
这样我就不会使用任何非由的软件了。
——*Richard Stallman（理查德·斯多曼），GNU 工程创始人。《GNU 宣言》*

本章中，我们逐步学习使你的程序可以在嵌入式系统上运行的每个步骤，我们也会讨论相关的开发工具，并了解如问创建在第二章“你的第一个嵌人式程序”里讲述的闪烁 LED 程序。在我们开始之前，我将讲清楚一件事，嵌入式系统编程和你以前从事的编程工作实质上并无区别。唯一改变的是每一个硬件平台都是独特的。不幸的是，一个不同点就会导致许多附加的软件复杂性。这也是你必须要比以前格外注意软件创建过程的原因。

创建过程

当目标平台（**target platform**，注 **1**）选定之后软件开发工具可以自动做很多的事情。这个自动过程是可能的，因为这些工具可以发掘程序运行的硬件和操作系统平台的特性。例如，如果你的所有程序将执行在运行 **DOS** 的 **IBM** 兼容 **PC** 上，那么你的编译器就可以自动处理（因此也使你无法得知）软件创建过

注 **1**：在这种方式下，术语“目标平台”最好理解为构成运行你的软件的基本运行环境的硬件和操作系统的统一体。在某些情况下，嵌入式系统并没有操作系统，那么你的目标平台就只是运行你的程序的处理器。

程的某些方面。而在另一方面，嵌入式软件开发工具很少时目标平台做出假定。相反，用户必须给出更清晰的指令来告知这些工具有关系统的具体知识。

把你的嵌入式软件的源代码表述转换为可执行的二进制映像的过程，包括三个截然不同的步骤。首先，每一个源文件都必须被编译或汇编到一个目标文件（**object file**）。然后．第一步产生的所有目标文件要被链接成一个目标文件，它叫做可重定位程序（**relocatable program**）。最后，在一个称为重定址（**relocation**）的过程中，要把物理存储器地址指定给可重定位程序里的每个相对偏移处。第三步的结果就是一个可以运行在嵌入式系统上的包含可执行二进制映像的文件。

图 **3-1** 说明了上述的嵌人式软件开发过程。在图中，三个步骤是由上至下表示的，在圆角矩形框里说明了执行该步骤所用到的工具。每一个开发工具都以一个或多个文件作为输人共产生一个输出文件。本章接下来的部分会说明关干这些工具和文件的更详细的内容。

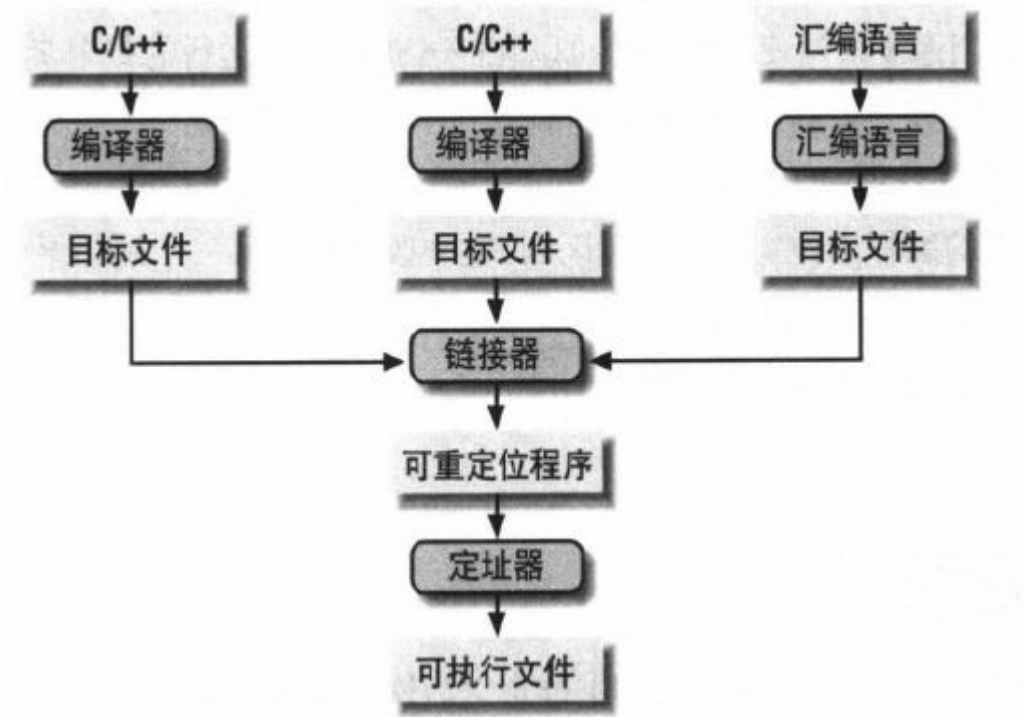

图 3-1 嵌入式软件开发过程

嵌入式软件开发过程的每一个步骤都是在一个通用计算机上执行的软件的转换过程。为了区别这台开发计算机（通常会是一台 **PC** 或 **Unix** 工作站）和目标嵌入式系统，我们称它作主机。换句话说，编译器、汇编器，链接器和定址器都是运行在主机上的软件。而不是在嵌入式系统上运行。可是，尽管它们事

实上在不同的计算机平台上运行，这些工具综合作用的结果是产生了可以在目标嵌人式系统上正确运行的可执行二进制映像。图 3-2 显示了这种功能的划分。

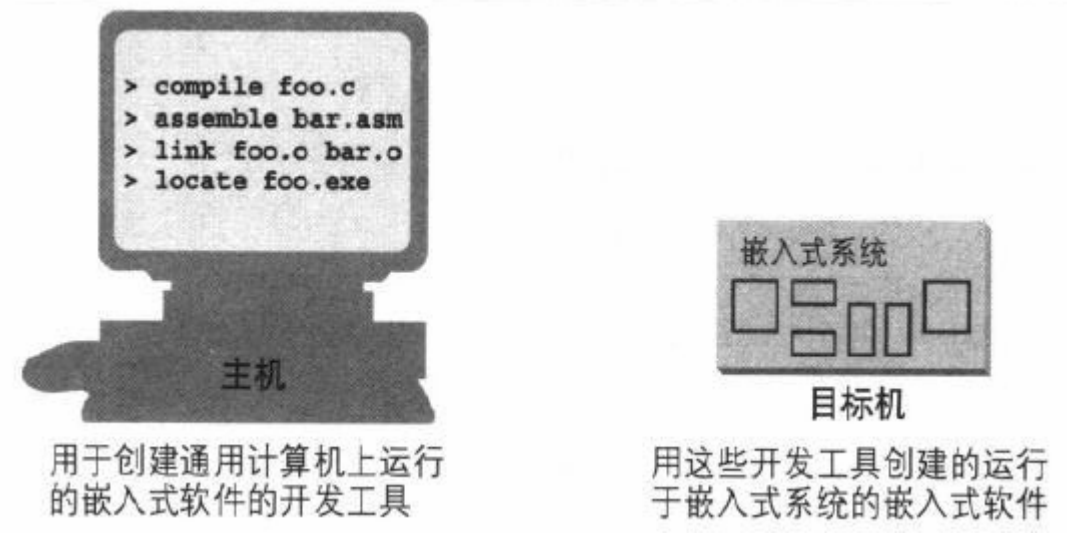

图 3-2 主机和目标机的划分

在本章和下一章里我将使用 GNU 工具（编译器、汇编器、链接器和定址器）作为示范。这些工具在嵌入式软件开发人员中使用极为普遍，因为可以免费得到它们（甚至源代码也是免费的），而且它们支持大多数最流行的嵌人式处理器。我会用这些特定工具的特性来说明一些讨论到的基本概念。一旦你领悟以后，同样的概念就可以应用到任何相类似的开发工具上。

编译

编译器的工作主要是把用人可读的语言所书写的程序，翻译为特定的处理器上等效的一系列操作码。在这种意义上，一个汇编器也是编译器（你可以称之为“汇编语言编译器”），但是它只执行了一个简单地逐行把人可阅读的助记符翻译到对应操作码的过程。这一节里的所有内容都同样适用于编译器和汇编器。这些工具综合在一起形成了嵌入式软件开发过程的第一个步骤。

当然，每一种处理器都有它独特的机器语言，所以你需要选择一个可以为你的目标处理器产生程序的编译器。在嵌人式系统的情况下，编译器几乎总是在主机上运行，在嵌入式系统本身运行编译器也没什么意义。一个像这样运行在一个计算机下台上并为另一个平台产生代码的编译器叫做交叉编译器

（cross-compiler）。使用交叉编译器是嵌人式软件开发的固定特征。GNU C/C++编译器（gcc）和汇编器（as）可以被配置为本地编译器或交叉编译器。用作交叉编译器的时候这些工具支持非常多的主机•目标机组合。表 3-1 列出了最常见的一些得到支持的主权和目标机。当然，主机和目标机的选择是相互独立的，这些工具可以被配置成住意的组合。

表 3-1 GNU 编译器所支持的主机和目标机

主机平台	目标机
DEC Alpha Digital Unix HP 9000/700 HP-UX IBM Power PC AIX IBM RS6000 AIX SGI Iris IRIX Sun SPARC Solaris Sun SPARC SunOS X86 Windows 95/NT X86 Red Hat Linux	AMD/Intel x86 (仅为 32 位) Fujitsu SPARClite Hitachi H8/300, H8/300H, H8/S Hitachi SH IBM/Motorola PowerPC Intel i960 MIPS R3xxx, R4xx0 Mitsubishi D10V, M32R/D Motorola 68k Sun SPARC, MicroSPARC Toshiba TX39

不管输入文件是 C/C++，汇编还是什么别的，交叉编译器的输出总是一个目标文件。这是语言翻译过程产生的包含指令集和数据的特定格式的二进制文件。尽管目标文件包含了一部分可执行代码，它却是不能直接运行的。实际上，目标文件的内部结构正强调了更大的程序的不完备性。

目标文件的内容可以想象成一个很大的、灵活的数据结构。这个文件的结构通常是按照标准格式定义的，比如“通用对象文件格式”（COFF）和“扩展的链接器格式”（ELF）。如果你计划使用不止一个编译器（就是说你用不同的源代码语言写你的程序的各个部分），那么你应该确定它们产生相同的目标文件格式。尽管很多编译器（特别是那些运行在 Unix 平台上的）支持类似 COFF 和 ELF 的标难格式（gcc 两者都支持），还是有一些编译器只产生专有格式的目标文件。如果你使用后一类编译器，你也许会发现你将不得不从同一个供货商处购买所有其他的开发工具。

大多数目标文件以一个描述后续段的头部开始每一段包含一块或几块源于源文

件的代码或数据，不过，这些块被编译器重新组合到相关的段中。比如，所有代码块都被收集到叫做 **text** 的段中，已初始化的全局数据（包括它们的初始值）被收集到叫做 **data** 的段中，未初始化的全局变量被收集到叫做 **bss** 的段里面。

通常在目标文件里还有一个符号表，记录了源文件引用的所有变量和函数的名字和位置。这个表的部分内容可能不完整，因为不是所有的变量和函数都总在同一个文件里定义。这些符号就是对别的源文件定义的变量和函数的引用，要一直到链接器的时候才会解决这些不完整的引用。

链接

在程序能被执行前，所有第一步产生的目标文件都要以一种特殊的方式组合起来。目标文件分开来看还是不完整的，特别是那些有未解决的内部变量和函数引用的目标文件。链接器的工作就是把这些目标文件组合到一起，同时解决所有未解决的符号问题。

链接器的输出是同样格式的一个目标文件，其中包含了来自输人目标文件的所有代码和数据。它通过合对输人文件里的 **text**、**data** 和 **bss** 段来完成这一点。这样，当链接器运行结束以后，所有输入目标文件里的机器语言代码将出现在新文件的 **text** 段里，所有初始化变量和未初始化变量分别在 **data** 和 **bss** 段里面。

在链接器台并各段内容的过程中，它也监视没解决的符号。例如，如果一个目标文件包含一个对变量 **foo** 的未解决的引用同时一个叫 **foo** 的变量在另外的一个目标文件里被声明，那么链接器将匹配它们。那个没解决的引用就会被一个到实际变量的引用所取代。换句话说，如果 **foo** 位于输出的数据段的偏移 **14** 的位置，它在符号表中的人口将包含这个地址。

GNU 链接器（***ld***）运行在和 **GNU** 编译器一样的所有主机平台上。它本质上是一个命令行工具，用来把参数中列出来的所有目标文件链接到一起。对于嵌入式开发来说，一个特殊的包含编译过的启动代码的目标文件也必须包括在参数列表里面。（参看本章后面的选读部分“启动代码”。）**GNU** 链接器也有一种脚本语言，可以用来对输出的目标文件施加更严格的控制。

启动代码

传统软件开发工具自动做的一件事是插入启动代码（**startup code**）。启动代码是用来位高级语言写的软件做好运行前准备的一小段汇编语言。每一种高级语言都有其希望的运行环境。比如 **C** 和 **C++**都使用了一个固定的堆栈。在任何用此两种语言写的软件可以正确运行之前，必须为堆栈分配空间并进行初始化。这还只是 **C/C++**程序启动代码的一个职责而已。

大多数供嵌入式系统使用的交叉编译器包括一个叫 **startup.asm**，**crt0.s**（“**C** 运行时”的所写）或者类似的一个汇编语言文件。随编译器提供的文档通常会说明该文件的位置和内容。

C/C++程序的启动代码通常包含以下行为，并且按照所列的次序执行:

1、禁止所有中断。
2、从 **ROM** 里复制所有初始化数据到 **RAM** 里。
3、把未初始化数据区清零。
4、未堆栈分配空间并初始化。
5、初始化处理器堆栈指针。
6、创建并初始化堆。
7、（只对 **C++**有效）对所有全局变量执行构造函数和初始化函数。
8、允许中断。
9、调用 **main**。

典型地，启动代码在调用 **main** 之后也包含一些指令。这些指令只在高级语言程序退出地情况下运行（即：从对 **main** 的调用返回）。根据嵌入式系统的种类，你也许会希望利用这些指令来暂停处理器，复位整个系统或者把控制传到一个调试工具。

因为启动代码不是自动插入的，程序员通常必须亲自汇编这段代码并把得到的目标文件包括在链接器的输入文件列表里。他甚至需要给链接器一个特殊的命令行选项以阻止它插入通常的启动代码。适用于多种目标处理器的启动代码可以在 **GNU** 软件包 **libgloss** 中找到。

如果在超过一个目标文件里都声明了同一个标号，链结器就无法继续了。它可能会通过显示一条错误信息来通知编程人员并退出。然而，如果所有目标文件都合并之后还有符号没有解决，链结器会尝试自己来解决引用问题。这个引用也许会指向标准库的一个函数，那么链接器按照命令行指示的顺序打开每一个库并检查它们的符号表。如果它找到具有引用名字的函数，就会把有关的代码和数据包含进输出目标文件以解决引用（注 2）。

不幸的是，标准库例程在可以用到嵌入式系统之前经常需要做一些改动。这里的问题是随大多数软件开发工具仅以目标文件格式提供标准库，所以你很少能自己来修改库的源代码。令人感激的是，一个叫 **Cygnus** 的公司提供了一个可用在嵌入式系统中的自由软件版的标准 **C** 库。这个软件包叫 **newlib**。你只需要从 **Cygnus** 的 **Web** 站点下载这个库的源代码，实现一些面向目标系统的功能，然后编译即可。然后这个库就可以和你的嵌人式软件链接到一起来解决任何以前没有解决的标准库调用。

在合并了所有代码和数据段并且解决了所有符号引用之后，链接器产生程序的一个特殊的“可重定位 的拷贝。换句话说，程序要说完整还差一件事：还没有给其内部的代码和数据指定存储区地址。如果你不是在为了一个嵌入式系统而工作，那么你现在就可以结束软件创建过程了。

但是嵌入式程序员一般在这个时候还没有结束整个创建过程。即使你的嵌入式系统包括一个操作系统，你可能仍然需要一个绝对定址的二进制映像。实际上，如果有一个操作系统，它包含的代码和数据很可能也在可重定位的程序里。整个嵌入式应用——包括操作系统——几乎总是静态地链接在一起并作为一个二进制映像来运行。

定址

把可重定址程序转换到可执行二进制映像的工具叫定址器。它负责三个步骤中最容易的部分。实际上，这一步你将不得不自己做大部分工作来为定址器提供关于目标电路板上的存储器的信息。定址器将用这个信息来为可重定址程序

注 2：需要注意我这里谈的只是静态链接。在非嵌入式环境里，动态链接库是很普遍的。在那种情况下，和库例程关联的代码和数据并不直接插入到程序里。

里的每一个代码和数据段指定物理内存地址。然后它将产生一个包含二进制内存映像的输出文件。这个文件就可以被调人目标 ROM 中执行。

在很多情况下，定址器是一个独立的开发工具。但是在 GNU 工具的情况下，这个功能建立在链接器里。试着不要被这个个别的实现所迷惑。不管你是为一个通用计算机还是一个嵌人式系统编写软件，你的可重定址程序里的段中的某些地方必须要被赋予实际地址。在第一种情况下，操作系统在调用程序的时候为你做这些事，在后一种情况下，你必须用一个独立的工具执行这个步骤。在定址器是链接器的一部分的情况下也是这样，就像在 ld 的情况下。

GNU 链接器需要的存储器信息可以通过一个链接器脚本来传递。这种脚本有时用来控制可重定址程序里代码和数据区的精确顺序。但是在这里，我们希望做出控制次序更多的事：我们希望建立每一般在存储器里的位置。

下面是为一个假设的有 512K RAM 和 512K ROM 的嵌入式目标板提供的链接器脚本的例子：

```
MEMORY
{
    ram : ORGIN = 0x00000, LENGTH = 512K
    rom : ORGIN = 0x80000, LENGTH = 512K
}

SECTIONS
{
    data ram :  /* Initialized data. */
    {
        _DataStart = .;
        *(.data)
        _DataEnd = .;
    } >rom

    bss :
    {
        _BssStart = .;
        *(.bss)
```

```
        _BssEnd = .;
    }

    _BottomOfHeap = . ; /* The heap starts here. */
    _TopOfStack = 0x80000;  /* The stack ends here. */

    text rom :  /* The actual instructions. */
    {
        *(.text)
    }
}
```

这段脚本告知 GNU 链接器的内置定址器有关目标板上存储器的信息，并指导它把 data 和 bss 段定位在 RAM 中（从地址 0X00000 开始），把 text 段放在 ROM 中（从 0x80000 开始）。不过，通过在 data 段的定义后面添加>rom 可以把 data 段中的变量的初始值作为 ROM 映像的一部分。

所有以下划线开始的名字（比如_TopOfStack）是可以从你的源代码内部引用的变量。链接器将用这些符号来解决输入的目标文件里的引用。这样，比方说，可能会有嵌人式软件的某一部分（通常在启动代码里面）把 ROM 可初始化变量的初始值拷贝到 RAM 的数据段中。这个操作的开始和停止地址可以通过引用整型变量_DataStart 和_DataEnd 符号化地建立。

创建过程的最后一步的结果是一个绝对定址的二进制映像，它可以被下载到嵌入式系统内或写人到只读式存储设备中。在前面的例子里，内存映像正好是 1MB 大小。无比如何，因为初始化数据段的初始值存放在 ROM 里，这个映像的低 512K 字节将只包含 0，所以只有映像较高的一半是有意义的。你将在下一章里看到如何下载并执行这个内存映像。

创建闪烁程序

不幸的是，因为我们使用 Arcom 的板子作为我们的参考平台，我们不能使用 GNU 工具来创建示例代码。作为替换我们使用 Borland 的 C++编译器和 Turbo Assembler 汇编器。这些工具可以在任何基于 DOS 和 Windows 的 PC 上运行（注

3)。如果你有一个 **Arcom** 的板子来做实验，现在就可以把它安装起来并在你的主机上安装 **Borland** 的开发工具。(参看附录“**Arcom** 的 **Targetl99EB**”来取得订货信息。）我用的是 **3.1** 版的编译器，运行在一台基于 **Windows 95** 的 **PC** 上。不过，任问可以为 **80186** 处理器生成代码的 **Borland** 工具都可以使用。

正如我所实现的那样，闪烁 **LED** 示例包含三个源文件模块：***led.c*** 和 ***blink.c*** 和 ***startup,asm***。创建过程的第一步是编译这两个文件。我们需要使用的命令行选项有：***-c*** 说明“编译，但是不要链接”，***-v*** 说明“在输出文件里包含符号调试信息”***-ml*** 说明“使用大内存模式”。还有***-l*** 说明“目标是 **80186** 处理器”。这里是实际命令：

```
bcc -c -v -ml -l led.c
bcc -c -v -ml -l blink.c
```

当然，要执行这些命令，***bcc.exe*** 必须在你的 **PATH** 路径里并且这两个源文件在当前目录下。换句话说，你应该在 ***Chapter2*** 子目录下。每个命令的结果都创建了一个和**.c** 文件有同样前缀而后缀是***.obj*** 的文件。所以如果一切顺利的话，在工作目录里会出现两个文件——***led.obj*** 和 ***blink.obj***。

尽管看起来在我们的例子里只有两个目标文件需要链接，实际上是有三个。这是因为我们必须给 **C** 程序链接某个启动代码。(参看本章前面的选读”启动代码”。）**Arcom** 电路板使用的示例启动代码在 ***Chapter3*** 子目录的文件 ***startup.asm*** 里。为了把这段代码编成目标文件，进入这个目录并发出如下命令：

```
tasm /nx startup.asm
```

结果是这个目录里多了一个 ***startup.obj*** 文件。实际把三个目标文件链接在一起的命令如下。需要注意在这个例子中命令行里目标文件出现的次序是有讲究的：启动代码必须放在第一个才能正确链接。

注 3: 应该注意 Borland 的 C++编译器不是专门为嵌入式软件开发人员设计的。相反它是设计来为使用 80x86 处理器的 PC 生成基于 DOS 和 Windows 的程序的。不过，通过使用特定的命令行选项可以允许我们指定特定的 80x86 处理器——比如 80186，这样就可以用这个工具作为类似 Arcom 的电路板的嵌入式系统的交叉编译器。

tlink /m /v /s ..\Chapter3\startup.obj led.obj blink.obj
blink.exe, blink.map

作为执行 ***tlink*** 命令的结果，**Borland** 的 **Turbo Linker** 链接器产生两个新文件：***blink.exe*** 和 ***blink.map***。第一个文件包含了可重定址的程序，第二个文件包含了人可阅读的程序映像。如果你以前从来没有看过一个映像文件，记着在往下读之前看一看这个文件。它提供了类似前面讲的链接器脚本所包含的信息。只不过这里是结果，所以包含了每一段的长度和在可重定址程序里的公共符号的名字和位置。

还有一个工具要用来使闪烁 **LED** 程序可以运行，它就是定址器。我们要用的定址器是 **Arcom** 随板子附带的 **SourceView** 开发和调试程序包的一部分提供的。因为这个工具是为这个特定的嵌入式平台设计的，所以它没有更通用的定址器带的很多选项（注 4）。实际上，只有三个参数：可重定址二进制映像的名字、**ROM** 的起始地址（以十六进制形式提供）和目标 **RAM** 的总长度（以千字节单位提供）：

tlink /m /v /s ..\Chapter3\startup.obj led.obj blink.obj
blink.exe, blink.map

作为执行 ***tlink*** 命令的结果，**Borland** 的 **Turbo Linker** 链接器产生两个新文件：***blink.exe*** 和 ***blink.map***。第一个文件包含了可重定址的程序，第二个文件包含了人可阅读的程序映像。如果你以前从来没有看过一个映像文件，记着在往下读之前看一看这个文件。它提供了类似前面讲的链接器脚本所包含的信息。只不过这里是结果，所以包含了每一段的长度和在可重定址程序里的公共符号的名字和位置。

还有一个工具要用来使闪烁 **LED** 程序可以运行，它就是定址器。我们要用的定址器是 **Arcom** 随板子附带的 **SourceView** 开发和调试程序包的一部分提供的。因为这个工具是为这个特定的嵌入式平台设计的，所以它没有更通用的定址器带的很多选项（注 4）。实际上，只有三个参数：可重定址二进制映像的名字、**ROM** 的起始地址（以十六进制形式提供）和目标 **RAM** 的总长度（以千字节单位提供）：

tcrom blink.exe C000 128

SourceVIEW Borland C ROM Relocator v1.06

Copyright (C) Arcom Control Systems Ltd 1994

Relocating code to ROM segment C000H, data to RAM segment 100H

Changing target RAM size to 1I8 Kbytes

Opening 'blink.exe'...

Startup stack at 0102:0402

PSP Program size 550H bytes (2K)

Target RAM size 20000H bytes (128K)

Target data size 20H bytes (1K)

Creating 'blink.rom'...

ROM image size 55HH bytes (2K)

tcrom 定址器给出了给每个段指定了基地址的可重定址输人文件的内容，并产生文件 ***blink.rom***。这个文件包含了一个已经绝对定址的二进制映像，它可以直接调入到 **ROM** 里。不过我们不是用一个设备编程器把它写人到 **ROM** 里，相反我们将生成这个二进制映像的一个 **ASCII** 版本并通过一个串行口把它下我到 **ROM** 型。要做到这一点我们还是使用 **Arcom** 提供的一个工具，叫做 **bin2hex**。下面是此命令的语法：

bin2hex blink.rom /A=1000

这个额外的步骤生成一个新的文件 **blink.hex**，它包合和 **blink.rom** 一样的内容，不过是以一种叫做 **Intel** 十六进制格式（**Intel Hex Format**）的 **ASCII** 格式表示的。

注 4 ： 不管怎样，它是免费的，这可比更通用的定址器便宜多了。

本章内容:

- 在 ROM 中的时候...
- 远程调试器
- 仿真器
- 模拟器和其他工具

第四章

下载和调试

我现在还清楚地记得那一刹那，我明白了从此以后我生活的
很大部分将用来找我自己程序的错误，
——*Maurice Wilkes，剑桥大学计算机实验室主任，1949*

当你已经在主机上有了一个可执行二进制映像文件的时候，你就需要有一种途径来把这个映像文件下载到嵌入式系统里来运行了。可执行二进制映像一般是要下载到目标板上的存储设备里并在那里执行。并且如果你配备了适当的工具的话，还可以在程序到设置断点或以一种不干扰它的方式来观察运行情况。本章介绍了可用于下载、运行和调试嵌入式软件的各种技术。

在 ROM 中的时候……

下载嵌入式软件的最明显的方式，是把二进制映像载人一片存储芯片并把它插在目标板上。虽然一个真正的只读存储芯片是不能再覆盖写人的，不过你会在第六章“存储器”里看到，嵌人式系统通常使用了一种特殊的只读存储器，这种存储器可以用特殊的编程器来编程（或重新写人编程）。编程器是一种计算机系统，它上面有各种形状和大小的芯片插座，可以用来为各种存储芯片编程。

在一个理想的开发条件下，设备编程器应该和主机接在同一个网络上。这样，可执行二进制映像文件就很容易传给它来对 ROM 芯片编程。首先把映像文件传到编程器然后把存储芯片插入大小形状合适的插座里并从编程器屏幕上的菜单里选择芯片的型号。实际的编程过程可能需要几秒钟到几分钟，这要看二进制映像文件的大小和你所用的芯片型号来定。

编程结束以后，你就可以把 ROM 插进板上的插座了。当然，不能在嵌入式

系统还加电的时候做这件事。应该在插入芯片之前关掉电源，插入之后再打开。

一旦加电，处理器就开始从 **ROM** 里取出代码并执行。不过，要注意到每一种处理器对第一条指令的位置都有自己的要求。例如，当 **Intel 80188EB** 处理器复位以后，它就会取位于物理地址 **FFFF0h** 的指令来执行。这个地址叫复位地址，位于那里的指令就叫复位代码。

如果你的程序看起来像是没有正确运行，那可能是你的复位代码出了点问题。你必须保证 **ROM** 里你的二进制映像格式要遵从目标处理器的复位要求。在开发过程中，我发现在复位代码执行完后打开板子上的一个 **LED** 非常有用，这样我一眼就知道我的新 **ROM** 程序是不是满足了处理器的基本要求。

注意：调试技巧#1：一个最简单的调试技巧就是利用 LED 来相示成功或者失败。基本的思路是慢慢地从 LED 驱动代码过渡到更大的程序。换句话说，你先从启动地址处的 LED 驱动代码开始。如果 LED 亮了，你就可以编辑程序，把 LED 驱动代码挪到下一个运行标记的地方。这个方式最适合像启动代码那样的简单以线性执行的程序。如果你没有本章后面提到的远程调试器或者任何其他调试工具的话，这也许是你唯一的调试办法了。

Arcom 电路板包含一个特殊的在线可编程存储器，叫做快闪存储器（简称闪存），它可以在不从板上移走的情况下编程。实际上，板上的另外一块存储器中已经包含了可以对这个快闪存储器编程的功能。你知道吗，**Arcom** 电路板上实际带了两个只读存储器，一个是真正的 **ROM**，其中包含了可以让用户对另外一片（即快闪存储器）在线编程的简单程序。主机只需通过一个串行通信口和一个终端程序就可以和这个监控程序沟通了。随板提供的“**Target188EB Monitor User's Manual**”包含了把一个 **Intel** 十六进制格式文件，比如 ***blink.hex***，载入到闪存里的指令。

这种下载技术的最大缺点是没有一种简单的方法来调试运行在 **ROM** 外面的软件。处理器以一种很高的速度提取指令并执行，并没有提供任何使你观察程序内部状态的手段。这在你已经知道你的软件工作正常并且你正计划分发这个系统的时候看起来是不错的，不过对于正在开发的软件是一点用都没有。当然，你还是可以检查 **LED** 的状态和其他外部可视的硬件指示，但这永远不会比一个

调试器提供更多的信息和反馈。

远程调试器

如果可能的话，一个远程调试器（**remote debugger**）可以通过主机和目标机之间的串行网络连接来下载、执行和调试嵌入式软件。一个远程调试器的前端和你可能用过的其他调试器都一样，通常有一个基于文本或 **GUI**（图形用户界面）的主窗口和几个小一点的窗口来显示正在运行的程序的源代码、寄存器内容和其他相关信息。所不同的是，在嵌人式系统的情况下，调试器和被调试的软件分别运行在两个不同的计算机系统上。

一个远程调试器实际上包含两部分软件。前端运行在主机上并提供前述的人机界面。但还有一个运行在目标处理器上的隐藏的后端软件来负责通过某种通信链路和前端通信。后端一般被称作调试监控器（**debug monitor**），它提供了对目标处理器的低层控制。图 **4-1** 显示了这两个部分是如何一起工作的。

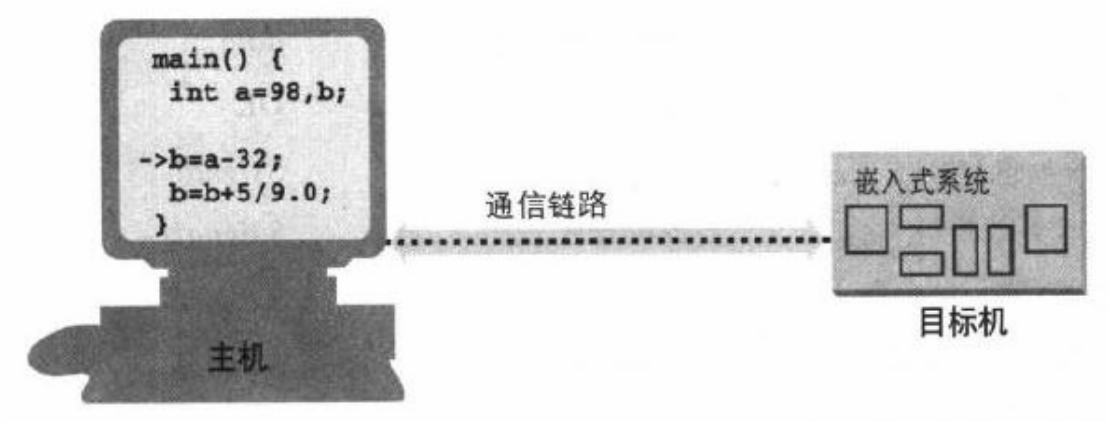

图 4-1　一个远程调试会话

调试监控器通常是由你或生产厂以前面讲过的方式放置在 **ROM** 的，它在目标处理器复位的时候会自动启动。它监控和主机的通信链路并对远程调试器的请求做出回应。当然，这些请求和监控器的响应必须符合某种预先定义好的通信协议，而且这些协议通常是很底层的。远程调试器的请求的一些示例就如“读寄存器 ***x***”、“修改寄存器 ***y***”、“读从 ***address*** 开始的内存的 ***n*** 字节”还有”修改位于 ***address*** 的数据”等等。远程调试器通过组合利用这些低层命令来完成诸如下载程序、单步执行和设置断点等高级调试任务。

GNU 调试器（*gdb*）就是这样的一个调试器。像其他 GNU 具一样，它一开始是被设计用来完成本机调试，后来才具有了跨平台调试的能力。所以你可以创建一个运行在任问被支持的主机上的 GDB 前端，它就会理解任何被支持的目标机上的操作码和寄存器名称。一个兼容的调试监控器的源代码包含在 GDB 软件包里面，并需要被移植到目标平台上。不过，要知道这个移植可能需要一些技巧，特别是如果你的配置里只能通过 LED 来调试的话（参见调试技巧#1）。

GDB 前端和调试监控器之间的通信专门被设计来通过串行连接进行字节传输。表 4-1 显示了命令格式和一些主要的命令。这些命令示范了发生在典型的远程调试器前端和调试监控器之间的交互类型。

表 4-1 GDB 调试监控器命令

命令	请求格式	响应格式
读寄存器	**G**	***data***
写寄存器	**G*data***	**OK**
读某地址数据	**m*address, length***	***data***
写某地址数据	**M*address, lenth:data***	**OK**
启动/重启执行	**c**	**S*signal***
从某地址开始执行	***caddress***	**S*signal***
单步执行	**s**	**S*signal***
从某地址开始单步执行	***saddress***	**S*signal***
重置/中止程序	**k**	***no response***

远程调试器是嵌入式软件开发里最常用到的下载和测试工具。这主要是因为它们一般比较便宜。嵌入式软件开发人员已经有了所需的主机了，何况一个远程调试器的价格并不会在全套跨平台开发工具（编译器、链接器、定址器等等）的价格上增加多少。还有，远程调试器的供应商们通常会提供他们的调试监控器的源代码，以增加他们的用户群。

Arcom 电路板在交付的时候在快闪存储器里包含了一个免费的调试监控器。和 Arcom 提供的主机软件一起使用，这个调试监控器就可以把程序直接下载到目标权的 RAM 里并运行。你可以用 tload 工具来完成这一工作。按照“SourceVIEW for Target188EB User's Manual”的指示简单地把 SourceVIEW 用行通信适配器接到目标位和主机上，然后在主机 PC 上执行下述命令：

tload -g blink.exe

SourceView Target Loader v1.4

Copyright (c) Arcom Control Systems Ltd 1994

Opening 'blink.exe' ... download size 750H bytes (2K)

Checking COM1 (press ESC key to exit) ...

Remote ident: TDR188EB version 1.02

Download successful

Sending 'GO' command to target system

-g 选项告诉调试监控器程序下载一结束就马上开始运行。这样一来，运行的就是和 **ROM** 里的程序完全对应的 **RAM** 里的程序了。在这种情况下，我们也许会以可重定址程序来开始，那么 **tload** 工具也会自动地在 **RAM** 里第一个可利用的地址处为我们的程序重新定址。

对于远程调试的目的，**Arcom** 的调试监控器可以用 **Borland** 的 **Turho Debugger** 做前端。然后 **Turbo Debugger** 就可以单步执行你的 **C/C++**和汇编语言程序、在程序里设置断点，并可以在程序运行时监控变量、寄存器和堆栈（注 **1**）。下面是你可能用来启动一个对闪烁 **LED** 程序的调试会话的命令：

tdr blink.exe

tver -3.1

Target Debugger Version Changer v1.2

Copyright (c) Arcom Control Systems Ltd 1994

Checking COM1 (press ESC key to exit) ...

Remote ident: TDR188EB version 1.02

TDR88 set for TD version 3.1

td -rp1 -rs3 blink.exe

Turbo Debugger Version 3.1 Copyright (c) 1988,92 Borland International

Waiting for handshake from remote driver (Ctrl-Break to quit)

tdr 命令实际是调用另外两个命令的一个批处理文件。第一个命令告诉板上的调试监控器你用的是哪个版本的 **Turbo Debugger**，第二个才实际调用了 **TurboDebugger**。每一次用 **Arcom** 板启动一个调试会话的时候都要发出这两条命令，

注 **1**：实际的交互过程和作用 **Turbo Debugger** 调试一个 **DOS** 或 **Windows** 程序没有什么不同。

tdr.bat 批处理文件只是用来把它们组合成一个单一的命令。这里我们再一次使用了程序的可重定址版本，因为我们要把程序下载到 RAM 里并在那里执行它。

调试器启动选项-rpl 和-rp3 设置了到调试监控器的通信链路的参数。-rpl 代表“remote-port（远程端口）=1”（COM1）．-rp3 代表“remote-speed（远程速率）=3”（38,400 波特率），这些是同 Arcom 调试监控器通信所要求的参数，在建立了和调试监控器的联系之后，Turbo Debugger 就可以开始运行了。如果没成功的话可能是串行连接出了问题。把你的安装过程和 SourceView 用户手册中的描述对照一下吧。

一旦进人 Turbo Debugger，你就会看到一个对话框显示“Program out of date on remote，send over link？（远程的程序已过期，是否通过链路发送？）”，选择“Yes”后，blink.exe 的内容就会被下载到目标 RAM 里。然后调试器会在 main 处设置第一个断点并指示调试监控器运行程序到此处。所以你现在看到的就是 main 的 C 源代码，一个光标指示着嵌入式处理器的指令指针正指向这个例程的人口点。

使用标准的 Turbo Debugger 命令，你可以单步执行程序、设置断点、监视变量和寄存器的值、做凋试器允许的任何事情、或者可以按下 F9 立即运行程序的剩下部分。这样做了以后。你就能看见板上的绿色 LED 开始闪烁了。确认程序和调试器都工作正常之后，按下 Arcom 板上的复位开关来复位嵌人或处理器，然后 LED 会停止闪烁。Turbo Debugger 又可以响应你的命令了。

仿真器

远程调试器用来监视和控制嵌人式软件的状态是很有用，不过只有用在线仿真器（In-Circuit Emulator，ICE）才能检查运行程序的处理器的状态。实际上，ICE 取代了（或者仿真了）目标板上的处理器。它自己就是一个嵌入式系统，有它自己的目标处理器、RAM、ROM 和自己的嵌入式软件。结果在线仿真器一段非常贵，往往要比目标硬件还贵。但是这是一种强有力的工具，在某些严格的调试环境下可以帮你很大的忙。

同调试监控器一样，仿真器也有一个远程调试器作为用户界面。某些情况下，甚至能使用相同的前端调试器。但是因为仿真器有自己的目标处理器，所以就

有可能实时地监视和控制处理器的状态。这就允许仿真器在调试监控器提供的功能外支持一些高级的调试特性，如：硬件断点和实时跟踪。

使用调试监控器，你可以在你的程序里设置断点。不过，这些软断点只能到指令提取级别，也就是相当于“在提取该指令前停止运行”。相比之下，仿真器同时支持硬件断点。硬件断点允许响应多种事件来停止运行。这些事件不仅包括指令提取，还有内存和 I/O 读写以及中断。例如，你可以对事件“当变量 foo 等下 15 同时 AX 寄存器等于 0”设置一个硬件断点。

在线仿真器的另一个有用的特性是实时跟踪。典型地，仿真器包含了大块的专用 RAM，专门用来存储执行过的每个指令周期的信息。这个功能使你可以得知事件发生的精确次序，这就能帮助你回答诸如计时器中断是发生在变量 bar 变成 94 之前还是之后这类问题。另外，通常可以限制存储的信息或者在察看之前预处理数据以精简要检查的数据的数量。

ROM 仿真器

另外一种仿真器也值得在这里提一下。ROM 仿真器被用来仿真一个只读存储芯片。和 ICE 一样，它是一个独立的嵌入式系统并和主机与目标板相连。不过，这次是通过 ROM 芯片插座来和目标板连接的。对于嵌人式处理器，它就像一个只读存储芯片，而对于远程调试器，它又像一个调试监控器。

ROM 仿真器相比调试监控器有如下几个优点。首先，任问人都不需要为你的专有目标硬件移植调试监控器代码。其次，ROM 仿真器通常自带了连接主机的串行或网络连接，所以不必占用主机自己的通常很有限的资源。最后，ROM 仿真器完全替代了原来的 ROM，所以不会占用目标板的存储空间来容纳调试监控器代码。

模拟器和其他工具

当然，还可以使用另外很多种调试工具，比如模拟器（simulator）、逻辑分析仪和示波器。模拟器是一个完全基于主机的程序，它模拟了目标处理器的功

能和指令集，它的用户用面通常和远程调试器的一样或比较类似。实际上，可以为后端模拟器使用一个调试器来做前端，就像图 4-2 显示的那样。尽管模拟器有很多不足，它在项目的早期特别是还没有任何实际的硬件可以用来试验程序的时候是相当有用的。

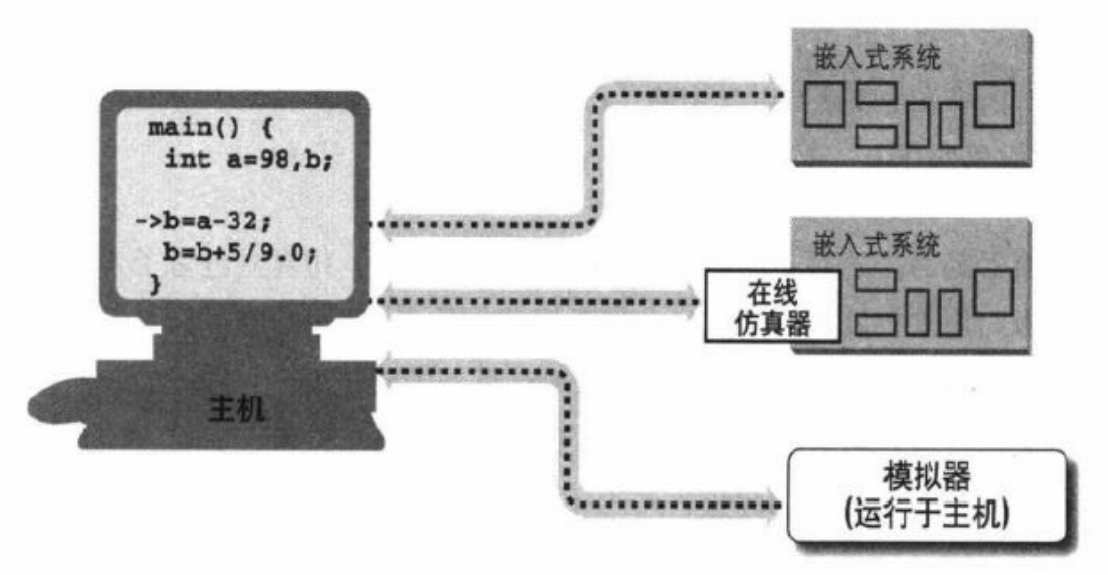

图 4-2 理想的环境：通用的前端调试器

注意：调试技巧#2：如果你曾经碰到目标处理器的行为和你阅读数据手册后所想的不一样的话，可以试着用模拟器试验一下程序。如果你的程序在这里运行良好，那你就知道发生了某种硬件问题。但是如果模拟器产生了和实际芯片同样的问题，那么你一定自始至终错误地理解了处理器的文档了。

到目前为止，模拟器最大的缺点是它仅能模拟处理器，而嵌人式系统经常包含一个或更多重要的外围设备。和这些设备的交互有时会限制模拟器的脚本或其他工作内容，而这些用模拟器很难产生的工作内容又会是很重要的。所以一旦你有了实际的嵌入式硬件以后就很可能用不着模拟器了。

一旦你开始接触你的目标硬件，特别是在硬件调试的时候，逻辑分析仪和示波器是绝对必要的调试工具。它们是调试处理器和电路板上其他芯片的交互过程的最有用的工具。不过，因为它们只能观察处理器外部的信号，所以它们不能像调试器或仿真器那样控制作的软件的执行过程。但是和一个软件调试工具比如远程调试器或仿真器结合起来，它们就是非常有用的。

逻辑分析仪是专门用来调试数字电路硬件的一种实验室设备。它会有几十个甚至上百个输入。它们分别只用来做一件事：它所连接的电信号的逻辑电子是 1 还是 0。你选择的任问输人子集都可以以时间坐标显示出来，如图 4-3 所示。大多数逻辑分析仪也允许你以特定的模式捕捉数据或“触发器”。例如，你可能发出如下请求：“显示输人信号 1 到 10 的值，但是直到输入 2 和 5 同时变为 0 时才开始记录”。

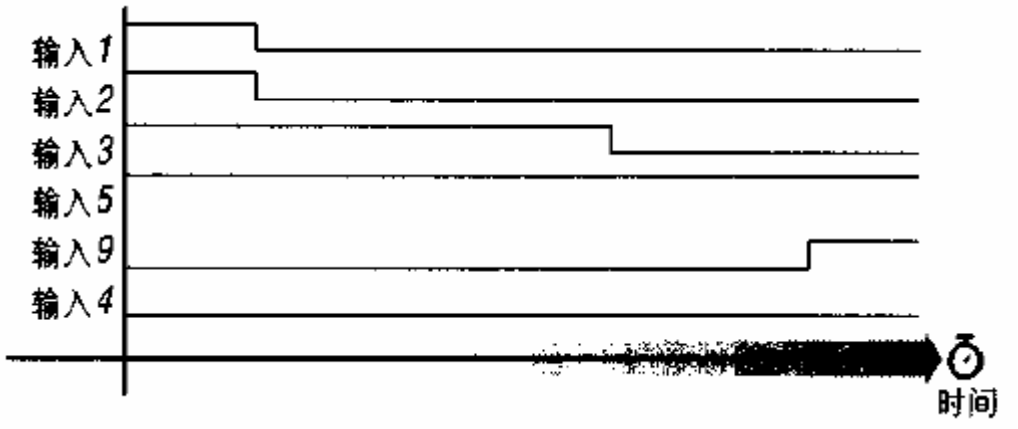

图 4-3　一个典型的逻辑分析仪的显示结果

注意：调试技巧#3：有时可能需要同时观察运行着嵌入式软件的目标板上电信号的一个子集。例如：你可能想观察处理器和它所连的一个外设的总线交互信号。一个技巧是在你感兴趣的交互的前面加上一个输出语句。这个输出语句会在处理器的一个或多个引脚上产生特定的逻辑电平。例如，你可以使一个空闲的 I/O 引脚从 0 变到 1，然后逻辑分析仪就可以设置成响应这个事件的触发器并开始捕捉后续的所有情况。

示波器是用于硬件调试的另一种实验室设备，不过它可以在任何硬件上检查任何电信号，不管是模拟的还是数字的。在手头没有逻辑为析仪的情况下，示波器可以迅速观察特定引脚上的电压，也可以做一些更复杂的事情。不过，它的输入很少（通常有四个）而且通常没有高级的触发逻辑。结果，只有在没有软件调试工具的情况下它才会对你有用。

本章讲述的大多数调试工具都会在每个嵌入式项目或多或少地用到。示波器和逻辑分析仪常用来调试硬件问题，模拟器用在软件开发的早期，调试监控器和仿真器用在实际的软件调试过程中。为了最有效地利用它们，你应该明白每一个工具用在什么地方，以及什么时候和什么地方使用它才会有最好的效果。

第五章

接触硬件

本章内容:

- 理解全貌
- 检查一下环境
- 了解通信过程
- 接触处理器
- 研究扩展的外围设备
- 初始化硬件

硬件[名] 计算机系统里可以被你踢上一脚的部分。

作为一个嵌入式软件工程师，你在以后的职业生涯重将会遇到很多不同的硬件。本章里我会介绍一些我用过的使自己熟悉一种新的电路板的简单过程。我将引导你创建一个秒数电路板最重要特性的头文件和把硬件初始化到某一已知状态的一部分软件代码。

理解全貌

在为一个嵌入式系统写软件之前，你必须先熟悉将要使用的硬件环境。首先，你需要了解系统的一般操作。你并不需要了解很小的细节，这些只是现在还用不到，慢慢就会碰到了。

无论何时你拿到一块新的电路板，都应该阅读一下附带的所有文档。如果这块班子使货价上拿来的标准产品，那么很可能会附带着面向软件开发人员的“用户手册”或“程序员手册”，如果板子是你为你的项目专门开发的，文档就可能写得更不清楚或主要是为硬件涉及人员做参考用的，不管怎样，这都是你唯一的最好起点。

再看文档的时候先把板子放在一边。这会有助你着眼于全局。等看完资料以后有得是时间来仔细检查电路板。在拿起这块板子之前，你应该能回答如下两个基本问题：

- 这块板子主要目标是什么？
- 数据是如何在里面流动的？

例如，假设你是一个调制解调器涉及队伍的软件开发人员，并且刚从硬件设计人员那里拿到一块早期的原型电路板。因为你已经对调制解调器比较熟悉，所以这块板子的主要目标和其中的数据流行你可能相当熟悉。这块板子的目的是通过模拟电话线发送和接收数据。硬件从一组电连接上读取数字信号然后转换程模拟信号传到相连的电话线上。当从电话线上读到模拟数据并输出数字信号的时候数据也会反方向流动。

尽管大多数系统的目的会很明显，可是数据流就可能不会是这样。我发现一份数据流图可以帮助你快速理解。如果你幸运的话，硬件所带的文档重会有你所需要的一整套方框图，而且你会发现它会帮助你创建你自己的数据流图。这样，你就可以不去理会那些和系统数据流无关的硬件元器件了。

对于 **Arcom** 电路板，硬件不是面向某一特殊应用设计的，所以为了本章后面的讨论，我们必须想象它有一个设计目的。我们可以假定这个板子是为了一个打印共享器而设计的。打印共享器可以允许两台计算机共用一台打印机。这个设备通过串口来连接每台计算机，通过并行口连接打印机，然后这两台计算机就可以向打印机发送文件了，不过同一时刻只能有一台计算机使用。

为了说明打印共享器的数据流向，我画了图 **5-1**。（只画出了 **Arcom** 板上和这个应用相关的部分。）通过看这张图，你可以很快地想象以下这个系统地数据流。从任一串行口接收要打印地数据并保存在 **RAM** 里，直到打印机可以接受更多地数据，然后通过并行口把数据送给打印机。**ROM** 里放了控制这一切地软件。

既然画好了方框图，就别急着把它揉一揉扔掉了，相反在整个项目进行中应已知留着以备参考。我建议使用一个项目记录本或装订夹，并把这张数据流图作为第一页。随着你使用这块硬件的工作的进展，把你了解到的所有东西都记到记录本上，你也许还会希望保持有关软件设计和实现的记录。一个项目记录本不仅在你开发软件时有用，而且在项目结束后也一样。当你需要对你的软件做一些改动，或者在几个月或几年后做类似工作的时候，你就会很感谢自己当年为保持一份记录而做的额外努力。

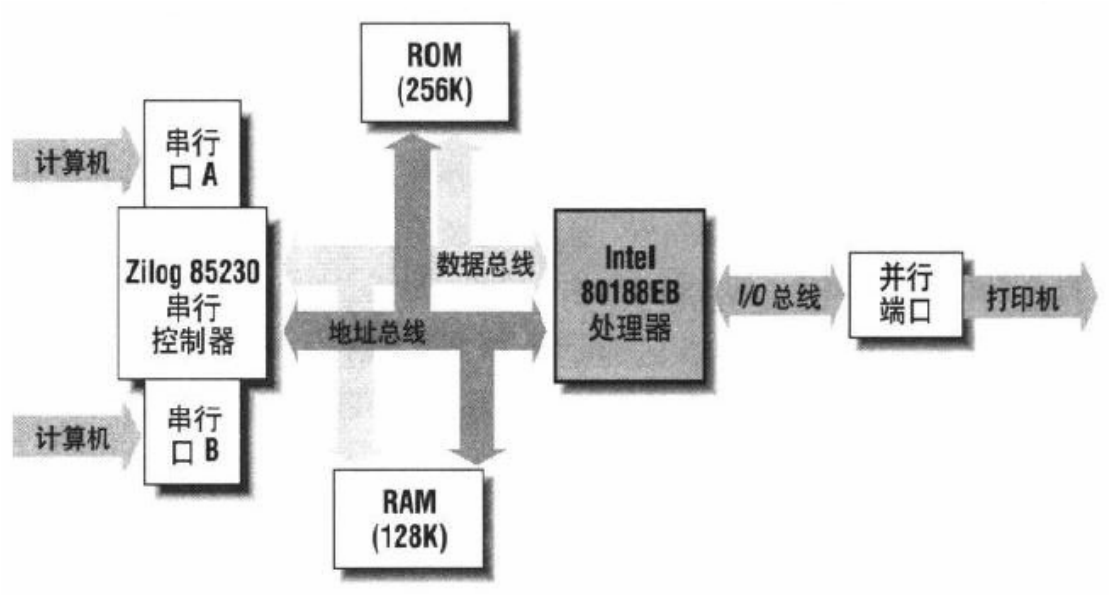

图 5-1 打印共享器的数据流向

如果你在读完硬件文档后，对整体情况还有什么疑问的话，你可以去询问一位硬件工程师来取得帮助。如果你还不认识这个硬件的设计人员的话，先花几分钟把你自己介绍一下，要是有时间的话，可以请他吃午饭或在工作以后送给他一只玩具熊（你可以不必讲这个项目!）。我发现很多软件工程师和硬件工程师沟通起来有困难，反过来也一样。在嵌入式系统开发中，软件人员和硬件人员的交流是特别重要的。

检查一下环境

经常把自己设想成处理器往往是很有帮助的，因为处理器是最终需要你指示来运行你的软件的部件。想象一下处理器可能是什么样子：处理器看起来会像是什么？如果你从这个角度想的话，你很快就会发现处理器有很多伙伴，在电路板上还有很多硬件部件，而处理器直接和它们进行通信。这一节里你将学习认识并找到它们。

首先要知道有两种基本类型：存储器和外设。很明显，存储器是用来存取数据和代码的。但是你也要考虑外设是什么。外设是和外部进行交互（I/O）或者完成某一特定硬件功能的特殊硬件设备。例如，嵌入式系统里最常见的两种外设是串行口和计时器。前者是一个 I/O 设备，后者基本上是一个计数器。

Intel 80x86 和其他一些处理器家族通过两个独立的地址空间来和这些存储器和外设打交道。第一个地址空间叫存储空间，是用来在取存储器设备的；第二个只为外设保留，叫做 **I/O** 空间。不过，硬件工程师也可以把外设放在存储空间里，这时，我们把这些外设称作存储器映像的外设。

从处理器的角度看，存储器映像的外设和存储设备非常相像。不过，一个外设和一个存储器的功能有明显的不同。外设下是简单地存储传给它的数据，而是把它翻译成一条命令或者要以某种方式处理的数据。如果外设位于存储空间的话，我们就说这个系统有存储器映像 **I/O**。

嵌入式硬件的设计人员往往喜欢只采用存储器映像 **I/O**，因为这样做对硬件和软件开发人员都很方便。它对硬件开发人员有吸引力是因为他可以因此而省去 **I/O** 空间，同时省去了相关的连线。这也许不会显著地降低电路板的生产成本，但是会降低硬件设计的复杂性。存储器映像外设对程序员也很有用，他可以更方便、更有效地使用指针、数据结构和联合来和外设进行交互（注 1）。

存储器映射

所有的处理器都在存储器里存放它们的程序和数据。有时存储器和处理器在同一个芯片里，不过更常见的是存储器会位于外部的存储芯片中。这些芯片位于处理器的存储空间里，处理器通过叫做地址总线和数据总线的两组电子线路来和它们通信。要读或写存储器取的某个位置，处理器先把希望的地址写到地址总线上，然后数据就会被传送到数据总线。

在你了解一块新的电路板的时候，可以创建一张表来显示在储空间里每个存储设备和外设的名字和地址范围。组织一下这张表，让最低的地址位于底部，最高的地址位于顶端。每次往存储器映射里添加一个设备的时候，按照它在内存里的大概位置来放进表内并用十六进制标出起始和结束地址。在往存储器映射图里插入所有的设备以后，记着用同样的方式把没利用的存储区域也标记出来。

注 1：如果 P2LTCH 寄存器是存储器映像的话，toggleLed 函数就连一行汇编代码都用不着。

回过头来再看一下图 5-1 所示的 Arcom 电路板的方框图，你会发现有三个设备连在地址和数据总线上。它们分别是 RAM、ROM 和一个标着“Zilog 85230 串行控制器”的神秘设备。Arcom 提供的文档说 RAM 位于存储器映射的底端并向上占据了 128KB 的存储空间。ROM 则位于存储器映射的顶部，并向下拓展了 256KB 的存储空间。但是这块区域实际上包含两片 ROM：一个 EPROM 和一个闪速存储器，并且分别具有 128KR 的容量、第三个设备，Zilog 85230 串行控制器，是一个存储器映射的外设，它的寄存器的寻址范围在 70000h 和 72000h 之间。

图 5-2 所示的存储器映射显示了对处理器而言这些设备的寻址范围。在某种意义上，这就是处理器的”通讯录”。就像你在生活中要维护一个名字和地址的列表一样，你也要为处理器维护一张类似的表。存储器映射图里包含了可以从处理器的存储空间访问的每个存储芯片和外设的人口。可以证明这张表是关于系统的信息里最重要的部分，并且应该随时更新并作为项目的永久记录的一部分。

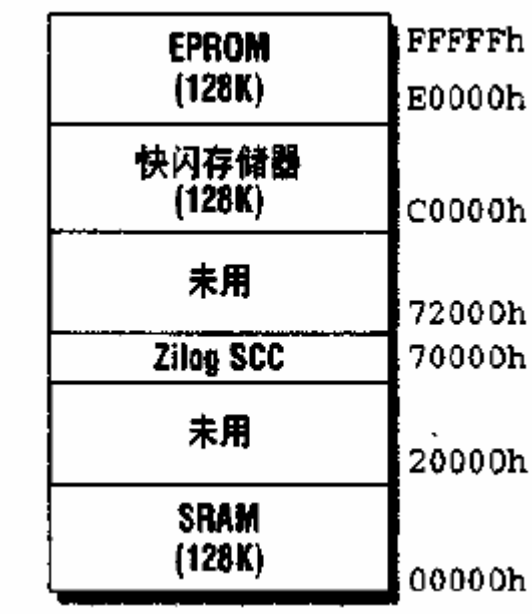

图 5-2 Arcom 电路板的存储器映射

对于每一块新的电路板来说，你应该创建一个头文件来描述它的特性，这个文件提供了硬件的一个抽象接口。在实际中，它使你可以通过名字而不是地址来引用板子上的各种设备。这样做带来的一个额外的好处，是使你的应用软件更加容易移植。如果存储器映射发生了变化——例如 128KB 的 RAM 被移走了——你只需要改变面向电路板头文件中相关的几行，然而重新编译你的程序。

随着本章的进行，我会告诉你如何来为 Arcom 电路板创建一个头文件。这个文件的第一部分就列在下面，这部分内容描述了存储器映射。它和图 5-2 中所表示的最大的区别是地址的格式。选读部分“指针和地址”解释了原因。

```
/*******************************************************
* Memory Map
*   Base Address    Size    Description
*   ------------    ----    -----------
*   0000:0000h      128K    SRAM
*   2000:0000h              Unused
*   7000:0000h              Zilog SCC Registers
*   7000:1000h              Zilog SCC Interrupt Acknowledge
*   7000:2000h              Unused
*   C000:0000h      128K    Flash
*   E000:0000h      128K    EPROM
*******************************************************/

#define SRAM_BASE   (void*) 0x00000000
#define SCC_BASE    (void*) 0x70000000
#define SCC_INTACK  (void*) 0x70000000
#define FLASH_BASE  (void*) 0xC0000000
#define EPROM_BASE  (void*) 0xE0000000
```

I/O 映射

如果存在独立的 I/O 空间的话，那就需要像完成存储器映射一样也要为电路板创建一个 I/O 映射。这个过程完全一样。简单地创建一张包含外设名字和地址范围的表，并组织一下把低地址放在底端。典型地，I/O 空间里的大部分是未利用的因为大多动外设只有几个寄存器。

图 5-3 显示了 Arcom 电路板的 I/O 映射。它包含三个设备：外设控制快（PCB）、并行口和调试口。PCB 是 80188EB 里的一组寄存器，用来控制片上的外设。控制并行口和调试口的芯片在处理器外面。这些端口分别是用来和打印机与基于主机的调试器通信用的。

指针和地址

在 C 和 C++里，指针的值就是地址。所以当我们说我们有一个指向数据的指针的时候，实际就是说我们有这个数据的地址。但是程序员通常不去直接设置或检查这些地址。这条规则的例外是操作系统、设备驱动程序和嵌入式软件的开发人员，他们有时需要在代码里明确地设置一个指针的值。

不幸的是，对地址的确切表示会因处理器而不同，甚至还会依赖编译器的实现。这就是说一个像 12345h 那样的地址可能不会精确地以那个格式被保存，甚至会因编译器不同而保存在不同的地方（注）。这就导致了一个问题，程序员该怎样明确地设置一个指针的值以使它指向存储器映射中希望的位置。

大多数 80/86 处理器的 C/C++编译器使用 32 位的指针。不过，比较老的处理器没有一个完全线性的 32 位地址空间。例知，Intel 80188EB 就只有 20 位的地址空间。另外，它没有一个内部处理器可以存放超过 16 位的数据。所以在这个处理器上，是通过两个 16 位处理器（段寄存器和偏移寄存器）来组合形成 20 位物理地址。（物理地址的计算是把段寄存器左移 4 位再加上偏移动寄存器，任何溢出到第 21 位的数据均被忽略。）

为了声明并初始化一个指向位于物理地址 12345h 处的寄存器的指针，我们如下书写：

```
int * pRegister = (int *) 0x10002345;
```

左边 16 位是段地址，右边 16 位是偏移地址。

为了方便，80x86 的程序员们经常以段地址：偏移地址对来书写地址，使用这种记法，物理地址 12345h 可以被写做 0x1000:2345。这就是上面我们用来初始化指针的不带冒号的值。不过，对任一可能的物理地址会有 4096 个不同的段地址：编移地址对。例如，地址 0x1200:0345 和 0x1234:0005（还有另外 4093 个）都引用了物理地址 12345h。

注：这种情况在你考虑了某些处理器提供的不同内存模式时变得更为复杂。本书所有的例子都假设使用 80188 的大内存模式。在这种内存模式下，我告诉你的所有细节都包含了整个指针的值，而在别的内存模式下，存储在指针里的地址的格式会因其指向的代码和数据的类型而不同。

图 5-3　Arcom 电路板的 I/O 映射

在为你的板子创建头文件的时候 I/O 映射也很有用。I/O 空间的每个区域直接映射到一个叫基地址的常数。上面的 I/O 映射到常数的翻译如下表所示：

```
/*********************************************
*
* I/O Map
*
*   Base Address    Description
*   ------------    --------------------------
*   0000h           Unused
*   FC00h           SourceVIEW Debugger Port (SVIEW)
*   FD00h           Parallel I/O Port (PIO)
*   FE00h           Unused
*   FF00h           Peripheral Control Block (PCB)
*
*********************************************/

#define SVIEW_BASE  0xFC00
#define PIO_BASE    0xFD00
#define PCB_BASE    0xFF00
```

了解通信过程

既然现在已经知道了与处理器相连的存储器和外设的名称和地址，那就接着学习如何与它们通信。有两种基本的通信技术：轮询（Polling）和中断（interrupt）。在每一种情况下，处理器都会通过存储或 I/O 空间向设备发出一些命令．然后等待设备完成指定的任务。例如，处理器通知一个定时器从 1000 倒计数到 0，一旦倒计数开始，处理器就只关心一件事：定时器记完数了吗？

如果使用轮询技术的话，处理器就反复地检查看任务完成没有。这就像在一个漫长的旅途中一个小孩子不停地问“我们到那儿了吗？”。就像那个小孩子一样，处理器花费了很多宝贵的时间来问这个问题而不停地得到否定的回答。要用软件实现轮询，只需要写一个循环语句来读有疑问的设备的状态寄存器即可。下面是一个例子：

```
do
{
    // Play games, read, listen to music, etc.
    // Poll to see if we're there yet.
    status = areWeThereYet();
} while (status == NO);
```

第二种通信技术用到了中断。中断是外设发给处理器的一个异步的电信号。在使用中断的情况下，处理器和与前面完全一样的方式向外设发命令，不过在处理器等待中断到达的时候，它可以继续做其他的事倩，当中断信号最终到来的时候，处理器把正在做的工作临时搁到一边，执行被称作中断服务例程（ISR）的一小段程序。ISR 执行完后，处理器就继续做刚才的工作。

当然，这不是完全自动的。程序员必须自己书写 ISR 然后“安装”并启动它，然后它才会在相关的中断到来的时候被执行。刚开始做这些的时候，可能对你是一个显著的挑战。不过，尽管这样，使用中断通常会从整体上减少代码的复杂性并导致一个更好的结构。与在一段不相干的程序里嵌人设备轮询不一样，这两部分代码保持了适当的独立性。

总的来说，中断与轮询相比是一种更有效的利用处理器的方式。处理器可以用更多的空余时间去做有用的工作。不过，使用中断也会带来一些开销。相对于中断执行一个指令的时间来说需要很多时间来把处理器当前的工作保存到一

旁并把控制权传给中断服务程序，需要把处理器的许多寄存器保存到存储器里，低优先级的中断也要被禁止。所以在实践中这两种方法使用都很频繁。中断用在效率非常重要或者需要同时监控多个设备的情况，轮询用在处理器需要比使用中断更迅速响应某些事件的情况下。

中断映射

大多数嵌入式系统只有很少的几个中断。每个中断都有一个中断引脚（在处理器芯片外部）和一个 ISR。为了使处理器执行正确的 ISR，必须在中断引脚和 ISR 之间存在一个映射。这个映射通常是以中断向量表实现的。向量表通常是位于某些已知内存地址处的指向函数入口的指针，处理器用中断类型（是和每一个引脚相关的一个唯一的数值）来作为这个数组的索引。存储在向量表那个地方的值通常就是要执行的 ISR 的地址（注 2）。

正确地初始化中断向量表非常重要。（如果初始化不正确的话，ISR 也许会响应错误的中断，或者根本不会执行。）这个过程的第一步是创建一个组织了相关信息的中断映射。一个中断映射就是一个包含了中断类型和它们所引用的设备的列表。这些信息会包含在电路板带的文档里。表 5-1 显示了 Arcom 电路板的中断映射。

表 5-1 Arcom 电路板的中断映射

中断类型	引用设备
8	**Time/Counter #0**
17	**Zilog 85230 SCC**
18	**Time/Counter #1**
19	**Time/Counter #2**
20	串行口接收
21	串行口发送

我们的目标依然是把这张表里的信息翻译成对程序员有用的形式。在创建了

注 2：一些处理器在这些地方实际上只存放了 ISR 的头几条指令，而不是指向例程的指针。

像上面那样的一个中断映射以后，你可以在面向电路板的头文件里加人第三个部分。中断映射里的每一行在头文件里是一个**#define** 语句，如下所示：

```
/*************************************
*
*   Interrupt Map
*
*************************************/


/*
*   Zilog 85230 SCC
*/
#define SCC_INT 17


/*
*   On-Chip Timer/Counters
*/
#define TIMER0_INT  8
#define TIMER1_INT  18
#define TIMER2_INT  19


/*
*   On-Chip Serial Ports
*/
#define RX_INT  20
#define TX_INT  21
```

接触处理器

如果你以前没用过你的电路板上的处理器的话，现在就应该花一些时间来熟悉一下。如果你一直用 **C** 或 **C++**编程的话，这花不了很长时间。对高级语言的使用者来说，大多数处理器看起来和用起来都差不多。不过，要是你做汇编语言编程的话，你就需要熟悉一下处理器的结构和指令集。

关于处理器你想了解的每一样东西都可以在制造商提供的数据手册里找到。如果你还没有用于你的处理器的数据手册或程序员指南的话，那就马上弄一本来。如果你想成为一个成功的嵌入式系统程序员的话，你必须能读数据手册并从中得到些什么。处理器数据手册通常写得很好（就像数据手册应该的那样），所以它们会是一个理想的起点。一开始先翻翻数据手册，把和手边的工作最有关系的章节记录下来，然后回头阅读处理器总述这一节。

处理器概述

许多最常见的处理器都是彼此相关的芯片家族的成员。在某些情况下，这样一个处理器的成员代表了发展途径上的几个点。最明显的例子是 **Intel** 的 **80x86** 家族，它从最初的 **8086** 一直横跨到奔腾 **III**，还有更新的。实际上，**80x86** 家族是如此成功，以至于光仿造它都成了一个工业。

本书中，术语处理器用来指微处理器、微控制器和数字信号处理器三种类型的芯片。微处理器这个名字通常保留来代表包含了一个功能强大的 **CPU** 同时不是为任何已有的特定计算目的而设计的芯片。这些芯片往往是个人计算机和高端工作站的基础。最常见的微处理器是 **Motorola** 的 **68K** 家族（在老式的 **Macintosh** 计算机里可以找到）和到处都有的 **80X86** 家族。

除了是专门设计采用在嵌入式系统里面这一点，微控制器很像微处理器。微控制器的特色是把 **CPU**、存储器（少量的 **RAM**、**ROM**、或者两者都有）和其他外设包含在同一片集成电路里。如果你购买包含这一切的一片芯片的话，就有可能充分地减少嵌入式系统的成本。最流行的微控制器包括 **8051** 和它的许多仿造产品，还有 **Motorola** 的 **68HCxx** 系列。也能经常发现流行的微处理器的微控制器版本。比如，**Intel** 的 **386EX** 就是很成功的 **80386** 微处理器的微控制器版本。

最后一种处理器是数字信号处理器，或者叫 **DSP**。**DSP** 里的 **CPU** 是专门设计来极快地进行离散时间信号处理计算的——比方那些需要进行音频和视频通信的场合。因为 **DSP** 可以比其地处理器更快地进行这类运算，它就为调制解调器和其他通信和多媒体设备的设计提供了一个功能强大、价格低廉的微处理器的替代品。两个最常见的 **DSP** 家族分别是来自 **TI** 和 **Motorola** 的 **TMS320Cxx** 和 **5600x** 系列。

Intel 的 80188EB 处理器

Arcom 板上使用的处理器是 **Intel 80188EB**，一个 **80186** 的微控制器版本。除了 **CPU** 以外，**80188EB** 还包含一个中断控制单元、两个可编程 **I/O** 口、三个定时器/计数器、两个串行口、一个 **DRAM** 控制器和一个片选单元。这些额外的硬件设备位于同一个芯片里并被当作片内外设来使用。**CPU** 通过内部总线可以和片内外设通信并直接控制它们。

尽管这些片内外设是截然不同的硬件设备，它们都用作 **80186 CPU** 的很小的扩展。软件可以通过读写被叫做外设控制块（**PCB**）的一个 **256** 字节的寄存器区来控制它们。你也许还记得我们在第一次讨论存储器和 **I/O** 映射的时候碰到过这个块。**PCB** 缺省地位于 **I/O** 空间里，从地址 **FF00h** 开始。不过要是愿意的话，**PCB** 也可以被重定位到 **I/O** 或存储器空间里的任何方便的地址处。

每一个片内外设的控制和状态寄存器都位于相对于 **PCB** 基地址的固定偏移处。在 **80188EB** 微处理器用户手册里可以查到每一个寄存器的确切偏移地址。为了把你的应用软件里和这些细节隔离开，把你会用到的寄存器的偏移地址包合到你的板子的头文件里会是一个很好的做法。我已经为 **Arcom** 电路板做了这个工作，不过了面只显示了会在后面章节里讨论到的寄存器：

```
/*****************************************
*
*   On-Chip Peripherals
*
*****************************************/


/*
*   Interrupt Control Unit
*/
#define EQI     (PCB_BASE + 0x02)
#define POLL    (PCB_BASE + 0x04)
#define POLLSTS (PCB_BASE + 0x06)
#define IMASK   (PCB_BASE + 0x08)
#define PRIMSK  (PCB_BASE + 0x0A)
```

```
#define INSERV  (PCB_BASE + 0x0C)
#define REQST   (PCB_BASE + 0x0E)
#define INSTS   (PCB_BASE + 0x10)

/*
*   Timer/Counters
*/
#define TCUCON  (PCB_BASE + 0x12)

#define T0CNT   (PCB_BASE + 0x30)
#define T0CMPA  (PCB_BASE + 0x32)
#define T0CMPB  (PCB_BASE + 0x34)
#define T0CON   (PCB_BASE + 0x36)

#define T1CNT   (PCB_BASE + 0x38)
#define T1CMPA  (PCB_BASE + 0x3A)
#define T1CMPB  (PCB_BASE + 0x3C)
#define T1CON   (PCB_BASE + 0x3E)

#define T2CNT   (PCB_BASE + 0x40)
#define T2CMPA  (PCB_BASE + 0x42)
#define T2CON   (PCB_BASE + 0x46)

/*
*   Programmable I/O Ports
*/
#define P1DIR   (PCB_BASE + 0x50)
#define P1PIN   (PCB_BASE + 0x52)
#define P1CON   (PCB_BASE + 0x54)
#define P1LTCH  (PCB_BASE + 0x56)

#define P2DIR   (PCB_BASE + 0x58)
#define P2PIN   (PCB_BASE + 0x5A)
#define P2CON   (PCB_BASE + 0x5C)
#define P2LTCH  (PCB_BASE + 0x5E)
```

其他你会想从处理器手册里了解的事情还包括：

- 中断向量表应该放在哪里？它是否必须位于内存中一个特定的地址？如果不是，处理器如何知道在哪里找到它？
- 中断向量表的格式是什么？它只是一个指向 ISR 函数的指针的表吗？
- 处理器自己是否产生一些特殊的叫做陷阱的中断？也要为这些中断写 ISR 吗？
- 怎样开和禁止中断（全部或个别）？
- 怎样得知知或清除中断？

研究扩展的外围设备

现在，你已经研究了除了扩展的外围设备之外的每个部件。这些扩展的外设是位于处理器外部并通过中断和 I/O 或存储映射寄存器来和处理器通信的硬件设备。首先来生成一张扩展设备的列表。根据你的应用的不同，这张表可能会包含 LCD 或键盘控制器、A/D 变换器、网络接口芯片或者一些定制的 ASIC（专用集成电路）。对于 Arcom 电路板，这张表只包含三个部件：Zilog 85230 串行控制器、并行口和调试口。

你应该拿到列表上每个设备的用户于用或数据手册。在项目的前期，你阅读这些文档的目的是要了解这些设备的基本功能。这些设备做些什么？哪些寄存器被用来发命令和取得结果？这些寄存器里下同的位和字段的意义是什么？如果这个设备会产生中断的话，什么时候产生？如何得知或清除一个设备的中断？

当你设计嵌入式软件的时候，你应该是着按设备来分割程序。通常为扩展外设结合一个叫做设备驱动程序的软件模块是个不错的主意。这个工作只是构建一个控制外设执行的软件例程的集会从而把应用软件和具体的硬件设备隔离开来。我会在第七章“外设”里详细介绍设备驱动程序。

初始化硬件

接触你的新硬件的最后一步是写一些初始化程序。这是你和硬件发展一种紧

密的工作关系的最好机会，特别是如果你希望用高级语言编写剩下的软件的活。在硬件初始化的过程中不可避免地要用到汇编语言。不过，完成了这一步以后，你就可以用 C 或 C++编写小程序了（注 3）。

注 3: 为了使第二章“你的第一个嵌入式程序”里的例子更交易懂一些，我在那里不说明任何初始化代码。不过，在你写像闪烁 LED 那样简单的程序之前也要使硬件初始化代码工作起来。

注意：如果你是最早使用一个新的硬件（特别是一个原型产品）的软件工程师之一的话，这个硬件也许不会像所宣称的那样工作。所有基于处理器的电路板都需要进行一些软件测试来确认硬件设计和各种外设的功能的正确性。当一些功能发生错误的时候会把你置于尴尬的境地。你怎么知道指责备硬件还是软件？如果你碰巧对硬件比较熟悉或者可以使用模拟器的话，你也许能设计一些实验来回答这个问题。否则，你可能得请一位硬件工程师来和你一起进行一个调试过程。

硬件初始化应该在第三章“编译、链接和定址”里所讲的启动代码之前执行，那里描述的代码假定硬件已被初始化从而只用来为高级语言程序创建一个合适的运行时环境。图 **5-4** 提供了关于整个初始化过程的一般的描述，从处理器复位到硬件初始化和 **C/C++**启动代码一直到 ***main***。

初始化过程的第一步是复位代码。这是处理器上电或复位时立刻执行的一小段汇编语言（通常只有两到三十指令）。这段代码的唯一目的是把控制传给硬件初始比例程。复位代码的第一个指令必须放在处理器数据手册里指定的在内存里的特定位置，通常叫做复位地址。**80188EB** 的复位地址是 **FFFF0h**。

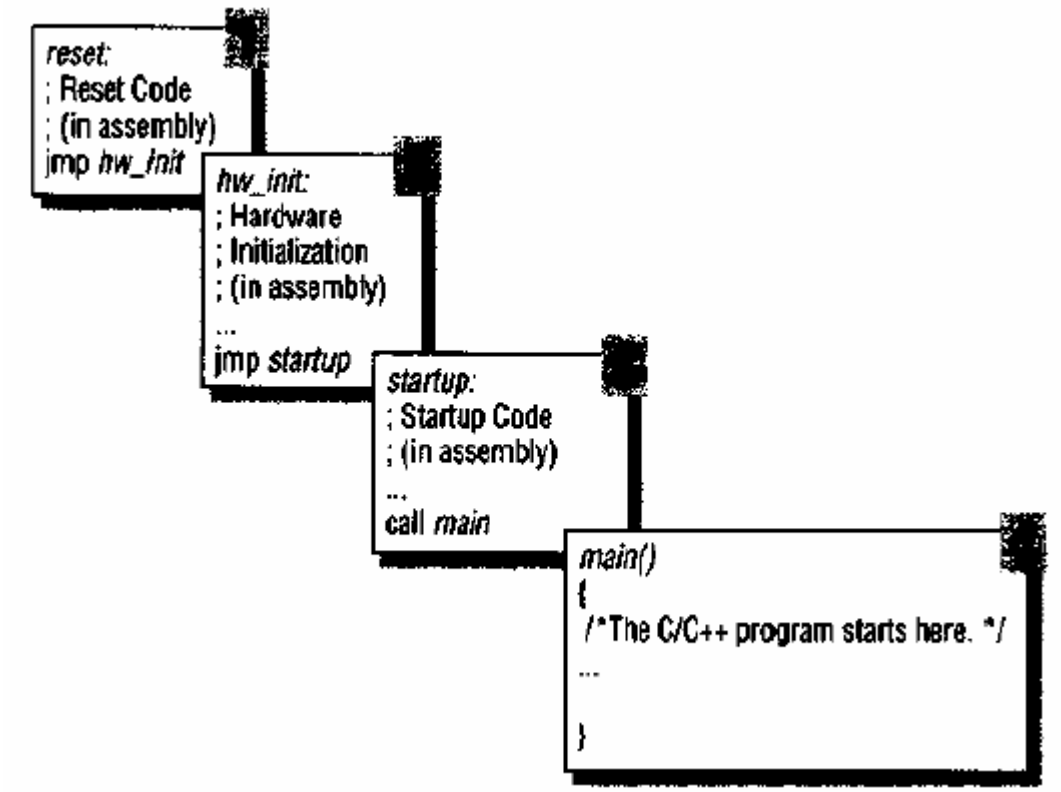

图 5-4 硬件和软件初始化过程

大多数实际的硬件初始出发生在第二个阶段。在这个地方，我们需要告诉处理器它自己所处的环境。这也是初始化中断控制器和其他重要外设的好地方。不太重要的外设可以在启动相应设备驱动程序的时候再初始化，而这些工作经常是在 ***main*** 里面完成的。

在用 **Inlel 80188FB** 处理器做任何工作之前，有几个内部寄存器必须被编程。这些寄存器作为处理器内部的片选单元的一部分负责设置存储器和 **I/O** 映射。通过对片选寄存器编程，你实质上唤醒了连在处理器上的每一个存储和 **I/O** 设备。每一个片选寄存器都和一条连结处理器和其他芯片的“芯片开关”相联系。特定的片选寄存器和硬件设备之间的这种联系是由硬件工程师建立的。你所要做的只是从他那里要一份片选寄存器的设置的清单并把这些设置调到寄存器里。

在刚启动的时候，**80188EB** 假设了一个最坏环境。它假定只有 **1024** 字节的 **ROM**（位于地址范围 **FFC00h** 到 **FFFFFh**）同时没有其他存储或 **I/O** 设备。这是处理器的“胎儿阶段”，它假定了例程 ***hw_init*** 必须位于地址 **FFC00h**（或更高处），而且这个例程必须不要求使用 **RAM**。硬件初始化例程一开始就设置片选寄存器来告诉处理器有关安装在电路板上的存储和 **I/O** 设备的信息。这个任务完

成以后，整个 **ROM** 和 **RAM** 地址范围就会都有效，然后你的剩下的程序就可以位于 **ROM** 或 **RAM** 任何方便的地方。

第三个初始性阶段包含了启动代码，就是我们在第三章里看过的汇编语言代码。提醒一下，它的任务是为用高级语言书写的代码做准备工作。重要的是只有启动代码会调用 ***main***。从这以后，你所有的软件都可以用 **C** 或 **C++**来写了。

你已开始理解嵌人式软件是如何从处理器复位过渡到你的主程序了。必须承认，第一次把所有这些组件（复位代码、硬件初始化、**C/C++**启动代码和应用程序）综合在一起放到一个新板子上可能有些问题，所以要准备花一些时间来分别进行调试。坦率地说，这是这个项目里最难的部分。你很快就会看到，如果你已经有了一个可以工作的闪烁 **LED** 程序作基础的话，工作将越来越容易，或者至少是更像普通的计算机编程。

一直到现在我们都在为嵌入式编程建造基础结构，但是我们接下来在后续章节里要讨论的主题是关于高级结构的：存储器测试、设备驱动程序、操作系统和真正有用的程序。这可能会是你在其他计算机系统项目里见过的软件，不过在嵌入式编程环境下有一些新的做法。

本章内容:

- 存储器的类型
- 存储器的测试
- 验证存储器内容
- 使用快闪存储器

第六章

存储器

Tyrell：如果我们给他们一个过去，我们就为他们的情感创造了一个缓冲，因此我们可以更好地控制他们。

Deckard：回忆，你是在说会议。

——电影《*Blade Runner*》

在这一章里，你将学会嵌入式系统中关于存储器所需要知道的知识。特别是，你会学习几种可能碰到的存储器，怎样测试存储设备以知道它们是否正常工作。以及怎样使用快闪存储器。

存储器的类型

很多类型的存储设备在现代计算机系统中都是可得的。作为一个嵌入式软件工程师，你必须明白它们之间的差别以及理解怎么有效地使用每一种类型的存储器。我的讨论中，我们要从一个软件的角度来接触这些设备。你在读书的时候，要记住这些设备的发展经历了几十年，因而在底层的硬件有着明显的差别。存储器类型的名字常常反映了历史发展的自然过程，经常是令人混淆的而没有更深的含义。

大部分的软件开发者把存储器想成是随机存取的（**RAM**）或者是只读的（**ROM**）。但是，实际上，每一种都有亚型，甚至有混合型的第三类存储器。在一个 **RAM** 设备中，存储在存储器中每一个位置的数据都可以在需要的时候读或者写。在一个 **ROM** 设备中，存储在存储器中每一个位置的数据可以随意的读取，但是不能够写入。有些时候，在一个类 **ROM** 设备中改写其中的数据是可能的。这种设备叫作混合存储器，因为它们同时表现了 **RAM** 和 **ROM** 的一些特征。图 **6-1** 为常用于嵌入式系统中的存储设备提供了一个分类。

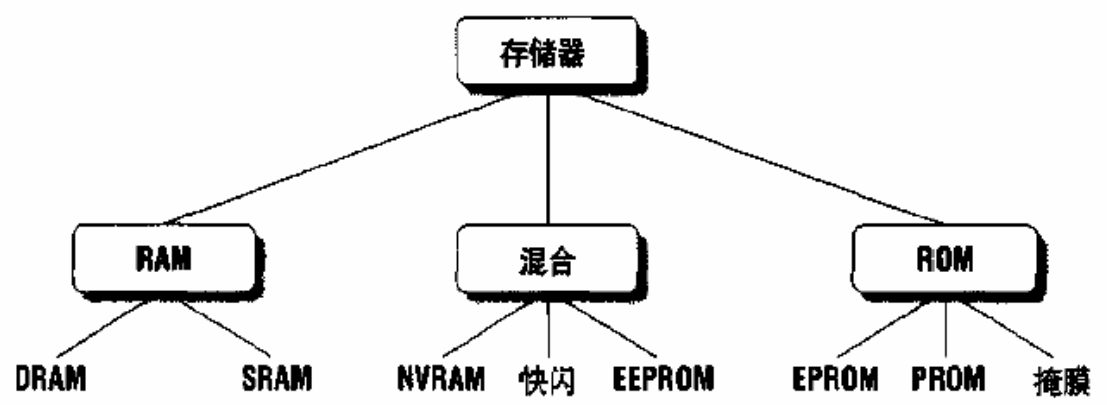

图 6-1 嵌入式系统中常用的存储类型

RAM 的类型

在 RAM 家族中有两种重要的存储设备：SRAM 和 DRAM。它们之间的主要差别是存储于其中的数据的寿命。SRAM（静态 RAM）只要是芯片有电就会保留其中的内容。然而，如果电源切断了或者是暂时断电了，其中的内容就会永远的丢失。另一方面，DRAM（动态 RAM）只有极短的数据寿命——通常不超过 0.25 秒。即使是在连续供电的情况下也是如此。

简言之，当你听到 RAM 这个词的时候，SRAM 具有你想像中的存储器的所有属性。与此相比，DRAM 听起来没有用。一个只保留所存数据零点几秒的存储设备有什么好处呢？就其本身来说这样一个易失数据的存储器确实是无用的。然而，一个叫作 DRAM 控制器的简单硬件可以使 DRAM 的行为更像 SRAM。（请参看后面的选读部分“DRAM 控制器”。）DRAM 控制器的任务是周期性地刷新 DRAM 中存储的数据。通过一秒钟之内几次刷新数据，DRAM 控制器就可以在需要的时间内保持 DRAM 中数据有效。因此 DRAM 归根结底和 SRAM 是同样有用的。

在决定选用哪一种类型的存储器的时候，系统设什者要考虑存取时间和成本。SRAM 设备提供了极快的存取时间（大约比 DRAM 快四倍），但是制造起来十分的昂贵。通常 SRAM 只是用于那此存取速度极端重要的场合。在大量的 RAM 需要的时候，每字节的更低价格使得 DRAM 很吸引人。很多嵌入式系统两种类型都包括：关键数据通道上的一小块 SRAM（几百个千字节）和其他所有地方的一大块 DRAM（以兆计）。

DRAM 控制器

如果你的嵌入式系统包括了 DRAM，板子上（或者是芯片内）可能也会有一个 DRAM 控制器，DRAM 控制器是位于处理器和存储器芯片之间额外的一个硬件。它的主要用途是执行刷新操作使得 DRAM 中数据有效。然而，没有你的帮助它不能够准确地做到这一点。

你的软件首先要做的一件事就是初始化 DRAM 控制器。如果你的系统中没有其他任何的 RAM，你必须在创建堆或者栈之前这样做。结果，这个初始化代码通常用汇编语言编写，放置在硬件初始化模块里。

几乎所有的 DRAM 控制器都需要一个短的初始化序列，其中包括一条或者多条初始化命令。初始化命令告诉控制器硬件对于 DRAM 的接口以及其中的数据必须以怎样的频率刷新。要为作具体的系统确定初始化序列，就要咨询板子的设计者或者是读描述 DRAM 和 DRAM 控制器的数据手册。如果你的系统中 DRAM 的工作好像不正常，那么可能是 DRAM 控制器没有初始化或者没有准确地初始化。

ROM 的类型

ROM 家族中的存储器是按照向其中写入新数据的方法（通常叫作编程）及其可以重写的次数来区分的。这个划分反映了 **ROM** 设备从硬连线，到一次性可编程，到可擦写可编程的演化过程。这些设备的一个共同的特性就是它们都能够永久地保存数格和程序，甚至是断电之后。

真正第一个 **ROM** 是硬连线设备，它包含一组预先编排的数据或者指令。**ROM** 中的内容下得不在芯片生产出来之前指定。因此实际的数据被用来安排芯片内部的晶体管。硬连线内存仍旧在使用，但是它们现在叫作“掩膜 **ROM**”以和其他类型的 **ROM** 区分。掩膜 **ROM** 主要的优点是低的产品成本。不幸的是，只有在需要成百上千相同 **ROM** 的拷贝时，成本才是低廉的。

比掩膜 **ROM** 更进一步的是 **PROM**（可编程 **ROM**）。它买来的时候处于未被编程的状态。如果你要看一个未经编程的 **PROM** 地内容，你会看到数据的每一位完全由 **1** 组成。把你的数据写人 **PROM** 的过程涉及到一个特殊的设备、叫作设备编程器。设备编程器通过向芯片的管脚加电，每一次向设备中写入一个

字节。一旦一个 PROM 通过这种方法被编程了，其中的内容就再也不能改变了。如果存储在 PROM 中的代码或者数据必须改变，目前这个设备就必须废弃。PROM 也称一次性可编程设备。

EPROM（可擦写可编程 ROM）编程的方式和 PROM 完全一样。然而，EPROM 是可以被擦除并且反复被编程的。为了擦除一个 EPROM，你只要把设备暴露在强紫外线光源下。（在设备的顶端有一个让紫外线照射到硅的窗口。）这样做，你基本上可以把整个芯片重置到其初始状态——未编程状态。尽管比 PROM 要贵，但是它们可以被再编程的能力使得 EPROM 成为软件开发及测试过程必需的一部分。

混合类型

由于存储器技术在最近几年已经成熟，在 RAM 和 ROM 设备之间的界线已经变得模糊。现在有几种类型的存储器结合了两者的优点。这些存储器不属于任何一类，总体上可以看作是混合存储设备。混合存储器随意地读写，像 RAM 一样，但是保持其内容而不需要供电，就像 ROM 一样。有两种混合设备，EEPROM 和快闪存储器，是 ROM 设备的子代；第三种，NVRAM，是 SRAM 的改版。

EEPROM 是电可擦除可编程的。在内部，它们和 EPROM 类似，但是擦除操作是完全依靠电力的，而不是通过在紫外线暴晒。EEPROM 中的任何一个字节都可以擦除和重写。一旦写人，新的数据就永远的保留在设备中了——至少直到它被擦除。对于这个改进了的功能的权衡主要是更高的价格。其写入周期也明显比写人一个 RAM 的要长，因此你不要指望利用 EEPROM 作为你的主要系统内存。

快门存储器是存储器技术最新的发展。它结合了目前为止所有存储设备的优点。快闪存储设备具有高密度、低价格、非易失性、快速（读取，而不是写入）以及电气可重编程等特点。这些优点是庄倒一切的，作为一个直接的结果，快闪存储器在嵌入式系统中的使用迅速增长。从软件的观点来说快速存储和 EEPROM 技术十分的类似。主要的差别是快速存储设备一次只能擦除一个扇区，而不是一个字节一个字节的擦除。典型的质区的大小是在 256 字节到 16 千字节的范围。尽管如此，快速存储设备比 EEPROM 要流行的多，并且还迅速地

取代了很多 ROM 设备。

混合存储器的第三个成员是 NVRAM（nonvolatile RAM，非易失 RAM）。非易失性是 ROM 及混合存储器前面讨论过的一个特征。然而，NVRAM 物理上与那些设备非常不同。NVRAM 通常只是一个带有后备电池的 SRAM。当电源接通的时候，NVRAM 就像任何一个其他的 SRAM 一样。但是当电源切断的时候，NVRAM 从电池中获取足够的电力以保持其中现存的内容。NVRAM 在嵌入式系统中是十分普遍的。然而，它是十分的昂贵——甚至比 SRAM 还要昂贵——因此，它的应用被限制于存储仅仅几百字节的系统关键信息，这些信息不可能有更好的存储办法了。

表 6-1 概括了不同存储器类型的特征。

表 6-1 存储器设备特征

存储器种类	易失性	可写	擦除大小	擦除周期	相对价格	相对速度
SRAM	是	是	字节	无限期	昂贵	快
DRAM	是	是	字节	无限期	适中	适中
掩膜 ROM	否	否	无	无	不贵	快
PROM	否	用编程器可写一次	无	无	适中	快
EPROM	否	是，利用编程器	整个芯片	有限制（见说明书）	适中	快
EEPROM	否	是	字节	有限制（见说明书）	昂贵	快，读取快写入慢
快闪存储器	否	是	扇区	有限制（见说明书）	适中	读取快写入慢
NVRAM	否	是	字节	无	昂贵	快

直接存储器存取

直接存储器存取（DMA）是一种直接在两个硬件设备之间传输数据决的技术。如果没有 DMA，处理器必须从一个设备中读取数据并且向另一个设备写入，一次一个字节或者一个字。如果要传输的数据量很大，传输的频率很高，软件的其他部分可能再也没有机会运行了。然而，如果有一个 DMA 控制器，就有可能让它执行整个传输，而几乎不借助于处理器。

DMA 是这样工作的。当一块数据要被传输，处理器向 DMA 控制器提供源和目标地址以及字节的总数。DMA 就自动地把数据从原传输到目标。在每一个字节被拷贝之后，每一个地址加 1 并且剩余字节的数目减 1。当剩余字节数目为零的时候，快传递结束，DMA 控制器向处理器发送一个中断。

在一个典型的 DMA 场景里，数据块被直接传输到内存或者从内存传出。比如，一个网络控制器要把一个网络包在它到达的时候传入内存，只要在整个包接收后通知处理器即可。利用 DMA，处理器可以把更多的时间花费在数据到达后对其的处理，而花费更少的处理数据在设备之间的传输。处理器和 DMA 控制器此时必须共享地址和数据总线，但是这是由硬件自动处理的，另一方面，处理器也不涉及实际的传输过程。

存储器的测试

一个严格的嵌入式软件作首先可能要写的是存储器的测试。一旦原型硬件就绪，设计者要做一些确认，确认她已经正确地连接了地址和数据线，确认内存芯片正常工作。开始，这可能看上去是一个很简单的任务，但是当你更加仔细地查看你的问题的时候。你会意识到利用一个简单的测试来发现细微的内存问题是很困难的。事实上，作为程序员天真的结果。很多嵌入式系统包含的内存测试只能发现最具灾难性的内存故障。其中一些甚至检测不出内存从板子上移走了。

存储器测试的目的是确认在存储设备中的每一个存储位置都在工作。换一句话说，如果你把数 50 存储在一个具体的地址，你希望可以找到存储在那里的那个数，直到另一个数写入。任何存储器测试的基本思想是写一些数值到每一个内存设备的地址，校验读回的数据。如果所有读回的数据和那些写入的数据是一样的，那么就可以说存储设备通过了测试。正如你将要看到的，只有通过认

真选择的一组数据你才可以确信通过的结果是有意义的。

当然，像刚才描述的有储器的测试不可避免地具有破坏性。在内存测试过程中，你必须覆盖它原先的内容。因为重写非易失性存储器内容通常来说是不可行的，这一部分描述的测试通常只适用于 **RAM** 的测试。然而，如果混合存储器的内容不重要——当它们处于产品的开发阶段——这些同样的算法也可以用于这些设备的测试。验证一个非易失性存储器内容的问题在本章的后面部分介绍。

普通的存储器问题

在学习具体的测试算法之前，你应该了解可能遇到的各种存储器问题。在软件工程师中一个普遍的误解是，大部分的存储器问题发生在芯片的内部。尽管这类问题一度是一个主要的问题，但是它们在日益减少。存储设备的制造商们对于每一个批量的芯片都进行了各种产品后期测试。因此，即使某一个批量有问题，其中某个坏芯片进人到你的系统的可能性是微乎极微的。

你可能遇到的一种类型的存储芯片问题是灾难性的失效。这通常是在加工好之后芯片受到物理或者是电子损伤造成的。灾难性大效是少见的，通常影响芯片中的大部分。因为一大片区域受到影响，所以灾难性的失效当然可以被合适的侧试算法检测到。

在沃的经历中，存储器出问题比较普遍的原因是电路板故障。典型的电路板故障有：

- 在处理器与存储设备之间的连线问题
- 无存储器芯片
- 存储器芯片的不正确插人

这些问题是一个好的存储器测试算法可以检测到的。这种测试不用特别地查找灾难性存储器故障，也应该能够检测到这类问题。所以我们更详细地讨论电路板的故障。

电于线路问题

电子线路问题可能是由电路板设计或者制造中的错误造成的，也可能是在加工好以后损坏的。连接存储器和处理器的每一根线都是三种中的一种：地址线、数据线、控制线。地址和数据线分别用来选择内存地址以及传输数据。控制线告诉存储设备处理器是要读还是写，以及数据将被传输的精确时间。不幸的是，这些线路中的一条或者多条有可能被不正确的布置或者以短路（也就是说，和板子上的其他线路连接）或者开路（不和任何电路连接）的方式受到损坏。这些问题经常是由一点焊接飞溅或者是由断路造成的。两种情况都在图 6-2 中做了说明。

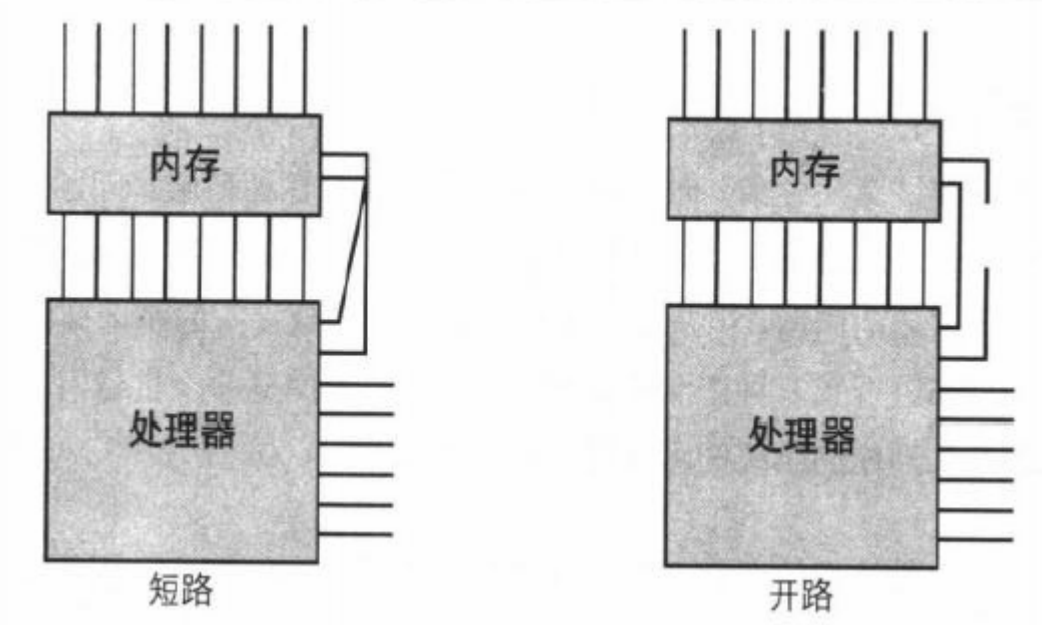

图 6-2 可能出现的错误

连接到处理器的电子线路问题会引起存储设备不正确的行为。数据可能存储的不正确，或存储在错误的地址，或者根本就没有保存。这样的情况可以分别地解释为数据线、地址线以及控制线上的线路问题。

如果问题出在数据线路上，几个数据位可能看上去像是被“粘”在了一起（也就是说，无论传输的数据如何，两个或者两个以上的位总是包含相同的值。类似地，一个数据位可能或者“粘高（总是 1），或者“粘低（总是 0）。这些问题可以通过写入一个设计好的数据序列来检测。每一个数据管脚可以被设置成 0 和 1 而坏受其他管脚的影响。

如果地址线出了问题，那么两个存储器位置中的内容看上去可能像是重叠的。换言之，写到某一个地址的数据会覆盖其他地址的内容。这是因为被短路或者开路的地址线会使得存储设备看到的地址不同于处理器选择的地址。

另一个可能性就是控制线短路或者开路。尽管理论上可能对控制线路问题进行专门的测试，但是不可能描述一个对这类问题的普遍的测试。很多控制信号的操作都是处理器或者存储器结构专用的。庆幸的是，如果控制线路有问题，存储设备根本就不工作，而这是可以通过其他的有储器的测试检测到的。如果你怀疑控制线路有问题，那么在构造一个专门的测试之前最好征求一下板子设计者的意见。

无存储器芯片

无存储器芯片无疑是一个应该被检测出的问题。糟糕的是，由于无连接电子线路的电容特性，一些存储器测试不能检测到这个问题。比如，假设你决定使用下面的测试算法：把值 l 写入到存储器的第一个位置，然后把它读回，验证它的值，把值 2 写人到存储器的第二个位置，读回它并验证，把值 3 写入第三个位置，验证它，依次类推。因为每一次读的操作都是紧跟在相应的写操作之后。所以有可能读回来的数据不代表任何东西，只不过是上次写操作保留在数据总线上的电压罢了。如果数据被过快的读回来，那么表面上看来数据好像已经被正确地保存入了存储器——即使在总线的另一端没有存储器芯片存在。

为了检测出无存储器芯片，测试必须改动。不是在相应的写操作之后立即读人验证，而是执行几个连续的写操作后再进行同样数量的读操作。比如，把值 1 写入到第一个位置，值 2 写入到第二个操作，值 3 写入到第三个位置，然后验证第一个位置的数据，验证第二个位置的数据等等。如果数据的值是唯一的（像刚描述的那个测试那样），那么无存储器芯片就可以被测试出了：读回来的第一个值会对应于最后写人的值（3），而不是第一个写入的值（1）。

芯片的不正确插入

如果有存储器芯片，但是插入到插槽时不正确，系统通常会表现出好像是一个连线问题或者是找不到存储器芯片。也就是说，存储器芯片的一些管脚根本没有和插槽相连，或者插入了错误的地方。这些管脚会是数据线、地址线或者

是控制线的一部分。因此，只要你能检测连线问题及无芯片，任何芯片的不正确插入也就可以自动检测出来了。

在继续讨论之前，让我们快速回顾一下我们必须检测的存储器问题的类型。存储器芯片很少有内部错误，但是，如果它们存在这样的错误，那么它们本质上可能是致命的，会被任何的测试检测到。问题的较普通的原因是电路板故障，其中会遇到连线问题，或者是无存储器芯片，或者是芯片的不正确插入。其他的错误也会遇到，但是，这里描述的问题是最普通的，也是用通用方法测试起来最简单的。

制定测试策略

通过仔细地选择你的测试数据以及被测试地址的顺序，是有可能检测出所有前面描述的存储器错误的。通常最好把你的存储器测试分成小的，思路简单的块。这有助于提高总体测试的效率以及代码的可读性。较专业的测试还会在检测到问题时提供更为详细的关于问题来源的信息。

我发现最好有三个独立的测试：数据总线的测试、地址总线的测试以及设备的测试。前面两个测试针对电子连线的问题以及芯片的不正确插入；第三个倾向于检测芯片的有无以及灾难性失效。作为一个意外的结果，设备的测试也可以发现控制总线的问题，尽管它不能提供关于问题来源的有用信息。

执行这三个测试的顺序是重要的。正确的顺序是：首先进行数据总线测试，接着是地址总线测试，最后是设备测试。那是因为地址总线测试假设数据总线在正常工作，除非数据总线和地址总线已知是正常的，否则设备测试便毫无意义。如果任何测试失败，你都应该和一个硬件工程师一起确定问题的来源。通过查看测试失败处的数据值或者地址，她应该能够迅速地找出电路板上的问题。

数据总线测试

我们首先要测试的就是数据总线。我们需要确定任何由处理器放置在数据总线上的值都被另一端的存储设备正确接收。最明显的测试方法就是写人所有可能的数据值并且验证存储设备成功地存储了每一个。然而，那并不是最有效率的测试方法。一个更快的测试方法是一次测试总线上的一位。如果每一个数据

上可被设置成为 0 和 1，而不受其他数据位的影响，那么数据总线就通过了测试。

独立测试每一个数据位的好办法是执行所谓的“走 l 测试”。表 6-2 说明了这个测试的 8 位版本中使用的数据模式。这个名字“走 1 测试’，来自这样一个事实：一个数据位被设置成 1，并且走过整个数据字。用于测试的数据值的数目和数据总线的宽度一样。这就使数据测试的次数从 2^n 减少到 n，其中 n 是数据总线的宽度。

表 6-2 “走 1 测试法”中的连续数据值

00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000

因为我们只是测试这一点的数据总线，所以所有的数据值都可以写人相同的地址。在存储设备中的任何地址都可以。然而，如果数据总线为了连接不见一个存储器芯片而分裂开来。那么你需要在多个地址进行数据总线的测试，每一个芯片对应一个地址。

为了执行“走 1 操作”，只要把表中的第一个数据写入，通过读操作来验证它，写入第二个数据，验证它等待。当你到达表的结尾的时候，测试就结束了。可以在写操作之后立即进行读操作，因为我们不是在测试芯片的有无。事实上，即使存储器芯片没有安装，这个测试也可以提供有意义的结果。

函数 *memTestDataBus()*说明了如何用 C 来实现“走 1 测试”的。它假设调用者会选择测试地址，并且在这个地址测试整个一组数据。如果数据总线正确工作，函数会返回 0。否则它会返回测试失败的那个数值。如果有故障的话，那个返回的数值对应于第一个有故障的数据行。

```
typedef unsigned char datum;    /* Set the data bus width to 8 bits. */
/************************************************************
```

```
* Function : memTestDataBus()
*
* Description : Test the data bus wiring in a memory region by
*           performing a walking 1's test at a fixed address
*           within that region. The address (and hence the
*           memory region) is selected by the caller.
* Notes:
*
* Returns:   0 if the test succeeds.
*           A nonzero result is the first pattern that failed.
*
****************************************************************/
datum
memTestDataBus(volatile datum * address)
{
    datum pattern;
    /*
    * Perform a walking 1's test at the given address.
    */
    for(pattern = 1; pattern !=0; pattern <<=1)
    {
        /*
        * Write the test pattern.
        */
        *address = pattern;
        /*
        * Read it back (immediately is okay for this test).
        */
        if( *address != pattern)
        {
            return(pattern);
        }
    }
    return(0);
}   /* memTestDataBus() */
```

地址总线测试行

在确认数据总线工作正常之后，你应该接着测试地址总线。记住地址总线的问题将导致存储器位置的重叠。有很多可能重叠的地址。然而，不必要测试每一个可能的组合。你应该按照前面测试数据总线的例子，努力在测试过程中分离每一个地址位。你只需要确认每一个地址线的管脚都可以被设置成 **0** 和 **1**，而不影响其他的管脚。

最小的一组可以覆盖所有组合的地址是“二的整数幂”地址。这些地址类似于“走 **1** 测试法”中的那组数值。对应的存储位置是 **0001h**、**0002h**、**0004h**、**0008h**、**0010h**、**0020h** 等等。此外，地址 **0000h** 也必须被测试。重合位置的可能性使得地址总线测试更难实现。在写入其中一个地址后，你必须检验任何其他的地址没有被覆盖。

不是所有的地址线路都可以用这种方法测试，注意到这一点很重要。部分地址——最左边的位——选择存储器芯片本身。如果数据总线的宽度大于 **8** 位，另外一个部分——最右边的位——可能不重要。比如，如果处理器有 **20** 个地址位，就像 **80188EB** 这样，那么可以寻址 **1** 兆的存储器。如果你想测试一个 **128K** 字节的存储块，那么三个最重要的地址位会保持不变（注 **1**）。那种情况下只有最右边的 **17** 位地址总线能够被真正地测试。

为了确定没有两个存储位置重叠，你应该首先在设备中每一个二的整数幂偏移地址上写一些初始数据。然后写一个新的数据——初始值的反值是一个好的选择——到第一个测试偏移地址，并且验证在其他二的整数幂偏移地址是否仍旧保存着初始数值。如果你发现一个位置（不是刚刚写入的那个），包含了新的数值。那么你就发现了当前地址位的一个问题。如果没有发现重叠，那么在每一个剩下的偏移地址重复这个过程。

函数 ***memTestAddressBus()***说明了这是如何做到的。函数接受两个参数，第一个参数是要测试的存储块的基本地址，第二个是它的大小，以字节为单位。大小是用来决定哪一个地址位应该被测试。对于最好的情况，基本地址应该在它们的每一位里包含一个 **0**。如果地址总线测试失败了，第一个错误被发现的地址会被返回。否则函数会返回 **NULL** 以表明测试成功。

注 1：128K 是 1 兆地址空间的八分之一。

```
/**********************************************************************
*     Function    : memTestAddressBus()
*
* Description : Test the address bus wiring in a memory region by performing
*                     a walking 1's test on the relevant bits of the address and
*                     checking for aliasing. The test will find single-bit address
*                      failures such as stuck-high, stuck-low, and shorted pins.
*                      The base address and size of the region are selected by the caller.
*
* Notes :      For best results, the selected base address should have enough LSB
*                0's to guarantee single address bit changes. For example, to test a
*                64 KB region, select a base address on a 64 KB boundary.
*                Also, select the region size as a power-of-two--if at all possible.
*
* Returns :    NULL if the test succeeds.   A nonzero result is the first address
*                 at which an aliasing problem was uncovered. By examining the
*                 contents of memory, it may be possible to gather additional
*                 information about the problem.
*
**********************************************************************/
datum *
memTestAddressBus(volatile datum * baseAddress, unsigned long nBytes)
{
    unsigned long addressMask = (nBytes - 1);
    unsigned long offset;
    unsigned long testOffset;

    datum pattern = (datum)0xAAAAAAAA;
    datum antipattern = (datum)0x55555555;

    /*
    * Write the default pattern at each of the power-of-two offsets.
    */
    for(offset = sizeof(datum); (offset & addressMask) != 0; offset <<=1)
    {
```

```
        baseAddress[offset] = pattern;
    }
    /*
    * Check for address bits stuck high.
    */
    testOffset = 0;
    baseAddress[testOffset] = antipattern;

    for(offset = sizeof(datum); (offset & addressMask) != 0; offset <<=1)
    {
        if(baseAddress[offset] != pattern)
        {
            return((datum *) &baseAddress[offset]);
        }
    }
    baseAddress[testOffset] = pattern;
    /*
    * Check for address bits stuck low or shorted.
    */
    for(testOffset = sizeof(datum); (testOffset & addressMask) != 0; testOffset
    <<= 1)
    {
        baseAddress[testOffset] = antipattern;
        for(offset = sizeof(datum); (offset & addressMask) != 0; offset <<= 1)
        {
            if((baseAddress[offset] != pattern) && (offset != testOffset))
            {
                return ((datum *) &baseAddress[testOffset]);
            }
        }
        baseAddress[testOffset] = pattern;
    }
    return (NULL);
} /* memTestAddressBus() */
```

设备测试

一旦你知道地址和数据总线是正确的，那么就有必要测试存储设备本身的完整性。要确认的是设备中的每一位都能够保持住 **0** 和 **1**。这个测试实现起来十分简单，但是它花费的时间比执行前面两项测试花费的总时间还要长。

对于一个完整的设备测试，你必须访问（读和写）每一个存储位置两次。你可以自由地选择任何数据作为第一步测试的数据，只要你在进行第二步测试的时候把这个值求反即可。因为存在没有存储器芯片的可能性，所以最好选择一组随着地址变化（但是不等于地址）的数。一个简单的例子是累加测试。

用于累加测试的数值如表 **6-3** 中前两列所示。第三列表示的是在第二步测试时使用的求反后的数据。第二步测试代表了一个递减测试。还有很多其他可能的数据选择。但是累加数据模式已经很充分了并且也容易计算。

表 **6-3** 累加测试的数值

存储器偏移	二进制值	求反值
000h	**00000001**	**11111110**
001h	**00000010**	**11111101**
002h	**00000100**	**11111011**
003h	**00001000**	**11110111**
…	**…**	**…**
0FEh	**11111111**	**00000000**
0FFh	**00000000**	**11111111**

函数 ***memTestDevice*** 就实现了这样一个两个步聚的累加/递减测试。它从调用者那里接受两个参数。第一个参数是起始地址，第二个是要测试的字节数。这些参数给用户设定一个最大的限制，超过这一限制的区域将被覆盖。如果测试成功，函数会返回 **NULL**。否则，包含不正确数值的第一个地址会被返回。

```
/*******************************************
*
* Function : memTestDevice()
*
```

```
* Description : Test the integrity of a physical memory device by performing
*               an increment/decrement test over the entire region.
*               In the process every storage bit in the device is tested as
*               a zero and a one. The base address and the size of the region
*               are selected by the caller.
*
* Notes :
*
* Returns :     NULL if the test succeeds. Also, in that case, the entire memory
*               region will be filled with zeros.

*               A nonzero result is the first address at which an incorrect value
*               was read back. By examining the contents of memory, it may be
*               possible to gather additional information about the problem.
*
**********************************************/

datum *
memTestDevice(volatile datum * baseAddress, unsigned long nBytes)
{
    unsigned long offset;
    unsigned long nWords = nBytes / sizeof(datum);

    datum pattern;
    datum antipattern;

    /*
    * Fill memory with a known pattern.
    */
    for(pattern = 1, offset = 0; offset < nWords; pattern++, offset++)
    {
        baseAddress[offset] = pattern;
    }

    /*
```

```
    * Check each location and invert it for the second pass.
    */
    for(pattern = 1,offset = 0; offset < nWords; pattern++, offset++)
    {
        if(baseAddress[offset] != pattern)
        {
            return((datum *) &baseAddress[offset]);
        }
        antipattern = -pattern;
        baseAddress[offset] = antipattern;
    }

    /*
    * Check each location for the inverted pattern and zero it.
    */
    for(pattern = 1,offset = 0; offset < nWords; pattern++, offset++)
    {
        antipattern = -pattern;
        if(baseAddress[offset] != antipattern)
        {
            return((datum *) &baseAddress[offset]);
        }
        baseAddress[offset] = 0;
    }
    return (NULL);
} /* memTestDevice() */
```

综合

为了使我们的讨论更加具体，让我们考虑一个实际的例子。假定我们想要测试 **Atcom** 板子上 **SRAM** 第二个 **64K** 字节的组块。要做到这一点，我们要依次调用这三个测试例程。每一种情况下，第一个参数都是存储块的基地址。看看我们的存储器映射，我们知道物理地址是 **10000h**，可以表示成存储段：偏移地址 **0x1000:0000**。数据总线的宽度位 **8** 位（**80188EB** 处理器的一个特性），并且总共有 **64K** 字节要测试（对应于地址总线最右边的 **6** 位）。

如果任何存储测试例程返回一个非零（或者非 NULL）的值，我们就立刻打开红色的指示灯以直观地指示错误。否则，在所有这三个测试成功完成以后，我们就打开绿色指示灯。万一发生错误，失败的测试程序会返回一些关于遇到的问题的信息。和一个硬件工程师交谈故障的特性时这个信息是有用的。然而，只有我们在调试器或者是模拟器下运行这个测试程序的时候这些信息才是看见的。

进行测试的最好办法，是把测试程序装人外部设备，让它运行至结束。然后，只要红色指示灯亮了，你就必须使用调试器单步调试程序。研究其返回代码以及内在中的内容以查明哪一个测试失败了以及为什么失败。

```
#include "led.h"

#define BASE_ADDRESS    (volatile datum *) 0x10000000
#define NUM_BYTES    0x10000

/*********************************************************
*
* Function: main()
*
* Description:    Test the second 64 KB band of SRAM.
*
* Notes:
*
* Returns:   0 on success.
*          otherwise -1 indicates failure.
*
*********************************************************/
main(void)
{
    if((memTestDataBus(BASE_ADDRESS) != 0) ||
        (memTestAddressBus(BASE_ADDRESS, NUM_BYTES) != NULL) ||
        (memTestDevice(BASE_ADDRESS, NUM_BYTES) != NULL))
    {
        toggleLed(LED_RED);
```

```
        return (-1);
    }
    else
    {
        toggleLed(LED_GREEN);
        return(0);
    }
} /* main() */
```

不幸的是，不是所有情况都可以用高级语言编写存储器测试程序。比如 C 和 C++都需要使用堆栈。但是堆栈本身就需要工作内存。在一个具有一个以上的存储设备的系统中，这也许是可行的。比如，你可以在一个已知能工作的 RAM 中创建一个堆栈，而测试另一个存储设备。这种一般的情况下，一个小的 SRAM 可以利用汇编来测试，这以后就可以在这里创建一个堆检了。然后，利用一个更好的测试算法，就像前面说明的那一个，可以用来测试一大块 DRAM。如果你没有足够的 RAM 以满足测试程序的对堆栈和数据的需要，那么你就需要完全用汇编重写整个侧试程序。

另一个选择是从模拟器中运行测试程序。这种情况下，你可以把堆栈放在模拟器自己的内部存储器中。把模拟器的内部存储器在目标存储器的映射中移来移去，这样你就可以系统地测试目标存储设备。

在产品的开发阶段，硬件的可靠性以及它的设计都是来经检验的，这时对于存储器测试的需要可能最明显。然而，存储器是任问嵌入式系统中最关键的资源之一，因此，在最终发布你的软件的时候，可能也希望包括一个存储器的测试。那种情况下应该在每一次系统启动或者重启的时候运行存储器测试和其他硬件确认测试。这个初始测试组形成一系列硬件诊断。如果一个或者更多的诊断失败，就要叫一个技师来诊断这个问题，并且替换有缺陷的硬件。

验证存储器内容

通常在处理 ROM 和混合存储设备时执行前面描述的存储器测试是行不通的。ROM 设备根本不可能被写，而混合式设备通常包含不可覆盖的数据和程序。然而，显然这些设备也会发生同样的存储设备问题。一个芯片可能会被遗漏，

不正确地插入、物理或者电子损伤，也可能是电子线路问题。与其假设这些非易失性存储设备能正常工作。不如找一些办法确认设备在正常工作以及其中的数据是有效的，这样你会感觉更好。即采用校验和以及循环冗余码进行数据校验。

校验和

我们怎么知道存储在非易失性存储设备中的数据或者程序是否仍旧有效呢？在数据已知正常的时候（比如，在对 **ROM** 编程之前），一个最容易的方法是计算它们的校验和。任何每次你要确认数据有效性的时候，你只要重新计算校验和并且把结果和上一次计算的结果比较。如果两个校验和相等，那么数据就被认为是有效的。通过小心选择校验和算法，我们可以增加检测出具体类型错误的可能性。

最简单的校验和算法是对所有的数据字节（或者是一个字，如果你喜欢一个 **16** 位的校验和的话）求和，舍去进位。这种算法的一个值得注意的缺陷是如果所有的数据（包括存储的校验和）意外地被重写为零，那么这个数据错误将不会被检测出来。一大块零的和也是零。克服这个缺陷的最简单的办法，是向校验和算法中加人最后的一步：把结果取反。那样，如果由于某种原因数据和校验和被重写为零，那么这个测试就会失败，因为正确的校验和应该是 **FFh**。

糟糕的是，像这样对简单的数据求和的校验和，不能检测到很多极普通的数据错误。很明显如果一位数据出现了错误（从 **1** 变到 **0** 或者相反），错误就会被检测出来。但是如果恰好同一列的两个数据位交替的出现了错误（第一个从 **1** 变到 **0** 第二个从 **0** 变到 **1**）怎么办？如果不改变正确的校验和，错误是不会被检测出来的。如果位错误发生，你可能想用一个更好的校验和算法。我们要在下一个部分学习它们中的一个。

在计算完预计的校验和之后，我们要一个地方来存储它。一个选择是提早计算核验和，然后在验证数据的例程中把它定义成一个常数。这使得容易插入校验和——只是在对存储设见编程之前，计算校验和并把它插人到存储映射中。当你重新计算核验和的时候，你只要跳过包含预期结果的位置，然后把新结果与存储在那里的数值比较。另一个存放校验和的好位置是在另一个非易失性存储设备中。这些解决办法在实际应用中都很有效。

循环冗余码

循环冗余码（CRC）是一个特定的校验和算法，它被设计开来检测最普遍的数据错误。CRC 后面的理论数学味很浓，超出了这本书的范围。然而，循环冗余码在需要存储或者传输大块数据的嵌入式应用中常常是有用的。接下来是一个关于 CRC 技术以及用 C 实现它的一些源代码的简单解释。谢大谢地，要利用它们能检测错误的优点，你不用理解为什么 CRC 可以检测到数据错误——甚至它们是如何实现的。

这里有一个简单的数学描述。当计算一个 CRC 的时候，你把一组数据考虑成一个由 1 和 0 组成的长字符率（也可叫做消息）。这个二进制字符串除以一个叫作“多项式生成器”的更小的固定的二进制字符串（以一种十分特别的方式）。这个二进制长除法的“余数”就是 CRC 检验数。仔细地选择多项式生成帮以满足某个合乎需要的数学属性，这样你就可以利用校验和来检测大部分（但不是全部）消息中的错误。这些多项式生成器中功能最强的能够检测出所有单、双位的错误以及所有奇位长度字符率的连续错误位。此外，所有突发误码（其定义是每一端都有错误的位的系列）的 99.99％可以被检测出来。总的说，任何存储或者发送二进制消息时可能发生的错误中，这些类型的错误占了一大部分。

那些具有非常好的错误检测能力的多项式生成器常常被采纳为国际标准。三个这样的标准参数化表示如图 6-4 所示。和每一个标准相关的是它的宽度（它的位数）。多项式生成器、一个叫作除数的多项式的二进制表示、余数的初始值以及与最后的余数（注 2）做或非运算的值。

注 2: 除数只是多项式生成器的系数的二进制表示——它们中的每一个都是 0 或者 1。使这更加令人困惑的是，多项式生成器的高阶系数（总是 1）是不考虑在二进制表示之内的。比如，第一个标准中的多项式生成器，CCITT，有四个非零的系数，但是相应的二进制表示中只有三个 1（第 12，5，0 位）。

注 3: 还有另外一种叫作”反射（reflection）”的潜在的色环（twist），我的代码不支持这种色环，你可能无论如何也不会需要它的。

表 6-4 国际标准多项式生成器

	CCITT	CRC16	CRC32
校验和大小（宽度）	16 位	16 位	32 位
多项式生成器	$x^{16}+x^{12}+x^5+1$	$x^{16}+x^{15}+x^2+1$	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+1$
除数（多项式）	0x1021	0x8005	0x04C11DB7
初始余数	0xFFFF	0x0000	0xFFFFFFFF
最后的或非值	0x0000	0x0000	0xFFFFFFFF

下面的代码可以计算所有的具有类似参数组（注 3）的 CRC 公式。为了尽可能地使这个计算容易，我把所有的 CRC 参数定义为常量。要改变到 CRC16 标准，只要改变三个常量的值即可。对于 CRC32，改变三个常量，重新定义 width 为 unsigned long 类型。

```
/*
* The CRC parameters. Currently configured for CCITT.
* Simply modify these to switch to another CRC Standard.
*/
#define POLYNOMIAL          0x1021
#define INITIAL_REMAINDER   0xFFFF
#define FINAL_XOR_VALUE     0x0000

/*
* The width of the CRC calculation and result.
* Modify the typedef for an 8 or 32-bit CRC standard.
*/

typedef unsigned short width.

#define WIDTH    (8 * sizeof(width))
#define TOPBIT   (1 << (WIDTH - 1))
```

函数 crcInit()应该首先被调用。它实现 CRC 算法要求的特殊二进制除法. 对于一个字节的消息数据的 256 种可能值它都会预计算其余数。这些中间结果被存储在一个全局的查询表中，这个查询表可以被函数 crcCompute()使用。这样做，一个大消息的 CRC 可以逐字节地计算而不用逐位地计算。这明显地减少了 CRC 计算的时间。

```
/*
* An array containing the pre-computed intermediate result for each
* possible byte of input. This is used to speed up the computation.
*/
width    crcTable[256];

/*****************************************************************************
**
*
* Function :crcInit()
*
* Description : Initialize the CRC lookup table. This table is used
*          by crcCompute() to make CRC computation faster.
*
* Notes :      The mod-2 binary long division is implemented here.
*
* Returns :  None defined.
*
*****************************************************************************
*/
void
crcInit(void)
{
    width remainder;
    width diviend;
    int bit;
    /*
    * Perform binary long division, a bit at a time.
    */
```

```
    for(dividend = 0; dividend < 256; dividend++)
    {
        /*
        * Initialize the remainder.
        */
        remainder = dividend << (WIDTH - 8);

        /*
        * Shift and XOR with the polynomial.
        */
        for(bit = 0; bit < 8; bit++)
        {
            /*
            * Try to divide the current data bit.
            */
            if(remainder & TOPBIT)
            {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            }
            else
            {
                remainder = remainder << 1;
            }
        }
        /*
        * Save the result in the table.
        */
        crcTable[dividend] = remainder;
    }
} /* crcInit() */
```

最后，我们看一下实际的主要例程，*crcCompute()*。这是一个你可以从应用程序中反复调用以计算和验证 CRC 校验和的例程。*crcInit()*和 *crcCompute()*的计算是分开的，这样做的一个额外的好处是函数 *crcInit()*不必要在嵌入式系统中执行。这个函数可以事先运行（在其他的任何计算机上），以产生查询表的内容。

表中的数值可以保存在 **ROM** 中（只需要 **256** 字节）并且可以被 ***crcCompute()*** 反复引用。

```
/*****************************************************************
*
* Function : crcCompute()
*
* Description :   Compute the CRC checksum of a binary message block.
*
* Notes :      This function expects that crcInit() has been called
*          first to initialize the CRC lookup table.
*
* Returns :  The CRC of the data.
*
*****************************************************************/
width
crcCompute(unsigned char * message, unsigned int nBytes)
{
    unsigned int      offset;
    unsigned char    byte;
    width          remainder = INITIAL_REMAINDER;

    /*
    * Divide the message by the polynomial, a byte at a time.
    */
    for( offset = 0; offset < nBytes; offset++)
    {
        byte = (remainder >> (WIDTH - 8)) ^ message[offset];
        remainder = crcTable[byte] ^ (remainder << 8);
    }
    /*
    * The final remainder is the CRC result.
    */
    return (remainder ^ FINAL_XOR_VALUE);
} /* crcCompute() */
```

使用快闪存储器

从程序员的角度看来，快闪存储器是已发明的最复杂的存储设备。自从 **1988** 引入了原始设备之后，硬件的接口有了一定程度上的改善，但是仍旧还有很长的一段路要走。从快闪存储器读操作快速并且容易。实际上，从快闪存储器中读入数据与其他的存储设备基本相同（注 **4**）。处理器只要提供地址，存储设备就返回在那里保存的数据。大部分的快速存储设备在系统重启的时候自动进入这种“读”的状态；启动“读”的状态不需要特别的初始化序列。

向快闪存储器中写入数据很困难。三个因素使得写操作困难。首先，每一个存储位置必须在重写操作之前被擦除。如果旧的数据没有被擦除，写操作的结果会是新旧数值的某个逻辑组合，存储的值通常不是你想要的。

第二件使得写入快闪存储器困难的事情是一次只能有一个扇区或者块被擦除；不可能只是擦除一个单个的字节。具体一个扇区的大小是随设备的不同而变化的，但是它通常是几千字节的量级。比如，在 **Arcom** 板子上的快闪存储设备——**AMD 29F010**——有八个扇区，每一个包含 **16K** 字节。

最后，擦除旧数据的过程和写入新数据的过程是随着制造商的不同而变化的，通常更加复杂。这些设备的编程接口是如此的拙劣以至于通常最好是加入一个软件层以使得快闪存储器更容易使用。如果这个软件层被实现，那么它通常被叫作快闪存储器的驱动程序。

快闪存储器的驱动

因为向快闪存储器中写入数据很困难，因此创建一个快闪存储器的驱动程序是有意义的。快闪存储器驱动的目的是对其余的软件隐藏一个具体芯片的细节。这个驱动应该有一个由擦除和写操作组成的简单应用编程接口。需要修改快闪存储器中的数据的部分应用软件只要调用驱动来控制细节即可。这允许应用程

注 4: 只有一点小区别值得一提。擦除和写入循环比读入循环的时间要长。因此，如果企图在这些操作的中间读入的话，结果将会延迟，或不正确，依设备而定。

序员做一个高级的请求，比如“擦除地址 D0000h 处的块”或者“写人一块数据，其开始于地址 D4000h”。它也使得设备专用代码分离，因此，如果日后使用其他制造商的快闪存储器，它可以客易地被修改。

Arcom 板子上快闪存储器 AMD 29F010 的驱动如下所示。这个驱动包含两个函数：*flashErase()*和 *flashWrite()*。这些留数分别擦除一整块扇区和写入一组字节。你应该能够从代码清单中看出和快速存储设备打交道不是一件容易的事。这段代码只是适用于 AMD 29F010。然而，同样的 API 可以用于任何快速存储设备。

```
#include "tgt188eb.h"

/*
* Features of the AMD 29F010 Flash memory device.
*/
#define FLASH_SIZE              0x20000
#define FLASH_BLOCK_SIZE  0x04000

#define UNLOCK1_OFFSET          0x5555
#define UNLOCK2_OFFSET          0x2AAA
#define COMMAND_OFFSET          0x5555

#define FLASH_CMD_UNLOCK1               0xAA
#define FLASH_CMD_UNLOCK2               0x55
#define FLASH_CMD_READ_RESET            0xF0
#define FLASH_CMD_AUTOSELECT            0x90
#define FLASH_CMD_BYTE_PROGRAM  0XA0
#define FLASH_CMD_ERASE_SETUP           0x80
#define FLASH_CMD_CHIP_ERASE            0x10
#define FLASH_CMD_SECTOR_ERASE    0x30

#define DQ7     0x80
#define DQ5     0x20

/*****************************************************************
```

```
*
* Function : flashWrite()
*
* Description : Write data to consecutive locations in the Flash.
*
* Notes :      This function is specific to the AMD 29F010 Flash
*              memory. In that device, a byte that has been
*              previously written must be erased before it can be
*              rewritten successfully.
*
* Returns :  The number of bytes successfully written.
*
*****************************************************************/
int
flashWrite(unsigned char * baseAddress, const unsigned char data[], unsigned int nBytes)
{
    unsigned char * flashBase = FLASH_BASE;
    unsigned int offset;

    for(offset = 0; offset < nBytes; offset++)
    {
        /*
        * Issue the command sequence for byte program.
        */
        flashBase[UNLOCK1_OFFSET] = FLASH_CMD_UNLOCK1;
        flashBase[UNLOCK2_OFFSET] = FLASH_CMD_UNLOCK2;
        flashBase[COMMAND_OFFSET]= FLASH_CMD_BYTE_PROGRAM;

        /*
        * Perform the actual write operation.
        */
        baseAddress[offset] = data[offset];
        /*
        * Wait for the operation to complete or time-out.
```

```
        */
        while((baseAddress[offset] & DQ7) != (data[offset] & DQ7))
&& !(baseAddress[offset] & DQ5));

        if((baseAddress[offset] & DQ7) != (data[offset] & DQ7))
        {
            break;
        }
    }
    return(offset);
} /* flashWrite() */

/*****************************************************************
*
* Function : flashErase()
*
* Description : Erase a block of the Flash memory device.
*
* Notes :    This function is specific to the AMD 29F010 Flash
*            memory. In that device, individual sectors may be
*            hardware protected. If this algorithm encounters
*            a protected sector, the erase operation will fail without notice.
*
* Returns :  0 on success.
*            otherwise -1 indicates failure.
*
*****************************************************************/
int
flashErase(unsigned char * sectorAddress)
{
    unsigned char * flashBase = FLASH_BASE;

    /*
    * Issue the command sequence for sector erase.
    */
```

```
    flashBase[UNLOCK1_OFFSET] = FLASH_CMD_UNLOCK1;
    flashBase[UNLOCK2_OFFSET] = FLASH_CMD_UNLOCK2;
    flashBase[COMMAND_OFFSET] = FLASH_CMD_ERASE_SETUP;
    flashBase[UNLOCK1_OFFSET] = FLASH_CMD_UNLOCK1;
    flashBase[UNLOCK2_OFFSET] = FLASH_CMD_UNLOCK2;
    *sectorAddress = FLASH_CMD_SECTOR_ERASE;

    /*
    * Wait for the operation to complete or time-out.
    */
    while(!(*sectorAddress & DQ7) && !(*sectorAddress & DQ5));
    if(!(*sectorAddress & DQ7))
    {
        return(-1);
    }
    return(0);
} /* flashErase() */
```

当然，这只是一种和快闪存储器接口的可能的方法，并且不是一个特别高级的。特别是，这个实现不处理任何芯片可能的错误。如果擦除操作永远也完成不了怎么办？函数 ***flashErase()***只会不停地运转，等待操作完成。一个更加强壮的实现应该用一个软件超时作为后备选择。比如，如果快速存储设备在两次最大的期待时间（由数据说明手册规定）内没有响应，例程可以停止轮询并且向调用者以某种方式指出错误。

人们有时用快闪存储器做的另一件事情是实现一个小的文件系统。因为快闪存储器提供了可被重写的非易失性存储，因此它可以被看作类似于任何其他的二级存储系统，比如硬盘。在作为文件系统的情况下，由驱动提供的函数要更加地面向文件。对于驱动程序的编程接口来说，像 **open()**、**close()**、**read()**、**write()** 等标准文件系统函数是一个好的起点。底层文件系统的结构根据你的系统的需要可简可繁。然而，像 **DOS** 系统使用的文件分区表结构这样易于理解的格式对于大部分的嵌入式工程来说是足够的。

本章内容:

- 控制和状态寄存器
- 设备驱动原理
- 一个简单的时钟驱动
- 修改后的闪烁程序

第七章

外围设备

每一个比萨饼滑到一个槽道中，像是一个电路板滑到一个计算机里一样，而这个位置又好像是汽车里随车携带的系统与只能盒之间的接口。客户的地址传达给汽车，汽车在一个平视的显示器上计算盒计划最优的行车路线。

——*Neal Stephenson*

《Snow Crash》

除了处理器和存储器之外，大部分嵌入式系统还包含一些其他的硬件设备。一些设备是专门用于应用领域的，而其他设备如时钟和串行端口。在各种系统中都是有用的。这些设备常常和处理器包含在同一个芯片里，它们叫作内部外围设备或者芯片级外围设备。因此，驻留在处理器芯片以外的硬件设备叫作外部外围设备。这一章里，我们要讨论在与这两种外围设备（简称外设）接口的时候产生的软件问题。

控制和状态寄存器

在嵌入式芯片和一个外围设备之间最基本的接口就是一组控制和状态寄存器。这些寄存器是外围设备硬件的一部分，它们的位置、大小以及具体的意义表明了外围设备的特性。比如，串行端口的寄存器不同于一个时钟/计数器。本节，我们要讨论怎样直接通过 C/C++程序操纵这些控制和状态寄存器中的内容。

根据处理器和板子的设计，外围设备位于处理器的存储空间或者输入输出空间里。实际上，嵌入式系统含有这两种类型的外设是很普遍的。它们分别被叫作存储映像外设和输入输出映像外设。这两种类型中，存储映像外设通常更加容易使用，逐渐地流行起来。

在储映像控制和状态寄存器可以看作是普通变量。要做到这一点，你仅仅需要声明一个指向寄存器或者寄存器块的指针，并且显式地设置指针的值。比如，如果第二章“你的第一个嵌入式程序”中的寄存器 **P2LTCH** 是存储映象的，并且位于物理地址 **7205Eh**，那么我们就可以完全用 **C** 来实现 ***toggleLed()***，如下所示。一个指向 **unsigned short** 的指针——一个 **16** 位的寄存器——被声明并且被显式地初始化为地址 **Ox7200:005E**。从那个点开始，指向寄存器的指针看上去就好像是任何一个指向其他整数变量的指针。

```
unsigned short * pP2LTCH = (unsigned short *) 0x7200005E;
void
toggleLed(void)
{
    *pP2LTCH ^= LED_GREEN; /* Read, xor, and modify. */
} /* toggleLed() */
```

然而，请注意，在设备寄存器和普通变量之间有着显著的差别。设备寄存器的内容会在你的程序不知道或者不介入的情况下发生改变。那是因为寄存器的内容还会被外设硬件修改。相反，变量中的内容不会改变。除非你的程序显式地修改了它们。因此，我们说设备寄存器的内容是易失的，或者会在不注意的时候被改变。

当声明指向设备寄存器的指针的时候，应该使用 **C/C++**的关键字 **volatile**。它会警告编译器不要对存储在这个地址里的数据做任何假设。比如，如果编译器看见两个紧跟着的向同一个易失位置的写操作。它不会假设第一个操作是在不必要地浪费处理器时间。换言之，关键字 **volatile** 提醒编译器，在优化阶段处理那个变量时，应该把它的行为看成在编译时无法预知。

这里有一个例子，使用 **volatile** 向编译器发出警告，注意前面的代码列表里的寄存器 **P2LTCH**。

```
volatile unsigned short * pP2LTCH = (unsigned short *) Ox720O005E;
```

如果你把这个语句理解为“这个指针本身是易失的”，将是错误的。实际上，在 **pP2LTH** 中的变量在程序的运行期间会保持 **0x7200005E**（当然，除非它在别的地里想的太多就容易把自己搞糊涂。只要记住寄存器的位置是固定的，但是

其内容却不一定。如果你使用了 **volatile** 关键字，编译器也会这样认为。

另一种设备寄存器，**I/O** 映像寄存器的主要的缺点是 **C/C++**没有标准的方法来访问它们。这种寄存器只有通过特殊机器指令才是可访问的。这些处理器专用的指令是不被标准的 **C/C++**语言所支持的。因此有必要利用一个特殊的库例程或者是内联的汇编（就像我们在第二章做的那样）来读写一个 **I/O** 映像设备的寄存器。

设备驱动原理

当开始设计设备驱动的时候，你应该总是把注意力集中在一个简单的规定目标上：完全隐藏硬件。当完成的时候，你的设备驱动模块应该是整个系统中唯一的一个直接读写某一个具体设备的控制和状态寄存器的软件。此外，如果设备产生任何中断，响应它们的中断服务例程应该是设备驱动完整的一部分。在本节中，我要解释为什么我要推荐这个原理以及它是如何实现的。

当然，企图完全隐藏硬件是困难的。你选择的任何编程接口都会反映设备广泛的特性。那是所期望的。目标应该是这样一个编程接口，这个接口在底层的外设被另一个它的通用型所替换时不需要改变。比如，所有的快闪存储设备共享一个扇区的概念（尽管扇区的大小可能在芯片间有所不同）。擦除操作只能在整个扇区上执行，并且一旦擦除，每一个字节或者字可以被重写。因此，上一章快速存储器驱动提供的编程接口应该适用于任何快速存储设备。**AMD 29F010** 的特性在这一层如愿以偿地被隐藏了。

嵌入式系统的设备驱动与工作站的差别很大。一个现代的计算机工作站，设备驱动最经常涉及的是满足操作系统的需要。比如，工作站操作系统通常对于它们本身与网卡之间的软件接口施加一个严格的要求，而不管底层硬件的特性和能力。想要用网卡的应用程序只能使用操作系统提供的网络 **API**，而不能直接访问网卡。这种情况下，完全隐藏硬件的目标容易满足。

相反，在嵌入式系统中的应用软件能够容易地访问你的硬件。实际上，因为所有的软件都连接在一起成为一个二进制映像，甚至在应用软件，操作系统以及设备方被修改了）。而它指向的那个数据会在无意间被改动。这是十分微妙的一点，而驱动之间几乎没有什么区别。这些界线的划分以及强制限制对硬件的

访问纯粹是软件开发者的责任。这两条是开发者必须有意识地做出的决定。换言之，在软件设计上，嵌入式程序员们可能比他们非嵌入式的同行更容易作弊。

好的设备驱动设计程序的好处有三点。第一，因为模块化，软件的总体结构更容易理解。第二，因为只有一个模块曾经直接和外设的寄存器相互作用，硬件的状态能更加容易地被跟踪。最后一点，但也是相当重要的一点，由硬件变动导致的软件变化集中在设备驱动程序上。这些好处都有助于减少嵌入式系统中程序错误的总数。但是你设计时不得不花一点精力来实现它们。

如果你认可把所有硬件特性以及相互作用隐藏在设备驱动里的原理。设备驱动通常由下面列出的五个部分组成。为了使得驱动的实现尽可能的简单和循序渐进，这些元素应该按照它们出现的顺序来开发。

1. 覆盖设备的存储映像控制及状态寄存器的数据结构

驱动程序开发过程的第一步是创建一个 **C** 风格的 **struct**，它看上去就好像你的设备的存储映像寄存器。这通常要研究外设的数据手册并且创建一个控制及状态寄存器和它们偏移地址的表。然后从最低偏移处的寄存器开始填充 **struct**。（如果一个或者多个位置是未用的，或者是保留的，一定要把哑元变量放在那里以填充这个附加的位置。）

一个这样数据结构的例子如下所示。这个结构描述了 **80188EB** 处理器芯片内其中的一个时钟/计数器单元。设备有三个寄存器，安排如下面的 **TimerCounter** 数据结构所示。每一个寄存器是 **16** 位宽，因此应该被看作一个无符号的整数，尽管它们中的一个 **control** 寄存器，实际上是一个个单独的符号位的集合。

```
struct TimerCounter
{
    unsigned short count;          //Current Count, offset 0x00
    unsigned short maxCountA; //Maximum Count, offset 0x02
    unsigned short _reserved;     //Unused Space, offset 0x04
    unsigned short control;        //Control Bits, offset 0x06
};
```

为了使得在 **control** 寄存器中的位易于单独地读写，我们可能也要定义下面的位屏蔽；

```
#define TIMER_ENABLE    0xC000 //Enable the timer.
#define TIMER_DISABLE   0x4000  //Disable the timer.
#define TIMER_INTERRUPT    0x2000  //Enable timer interrupts.
#define TIMER_MAXCOUNT   0x0020  //Timer complete?
#define TIMER_PERIODIC0x0001  //Periodic timer?
```

2. 跟踪目前硬件和设备驱动状态的一组变量

驱动程序开发过程的第二步是确定需要那个变量来跟踪硬件和设备驱动的状态。比如，前面所讲的时钟/计数器单元的例子里，我们可能要知道硬件是否已经被初始化。而且，如果已经初始化，可能还要知道正在运行倒计数的计数器的值。

一些设备的驱动创建了不只一个软件设备。这纯粹是一个逻辑设备，它实现于基本外围硬件之上。比如，容易设想从单独的一个时钟/计数器单元可以创建多个的软件时钟。时钟/计数器单元应该被设置以产生一个周期性的时钟节拍（**tick**），设备驱动则通过保持每一个软件时钟的状态信息来管理一组不同长度的软件时钟。

3. 一个把硬件初始比到已知状态的例程

一旦你知道怎样跟踪物理和逻辑设备的状态，那么就该开始写实际与设备交互及控制设备的函数了。最好是从一个硬件的初始比例程开始，那是一个熟悉设备交互的好方法。

4. 合起来为设备驱动的用户提供 API 的一组例程

你成功地初始化设备之后，你就可以开始向驱动程序中加人其他的功能。如果顺利的话，你已经决定了各个不同例程的命名和目的，以及它们各自的参数和返回值。现在所要做的就是实现并测试它们。我们在下一部分将看到这种例程的例子。

5. 中断服务例程

最好在启用中断之前，就已经设计，实现并且测试了大部分的设备驱动例程。确定那些与中断相关问题的来源是十分具有挑战性的工作。如果你把其他驱动模块的错误与之混合起来，确定错误的来源甚至是不可能的。最好使用轮询来添补驱动工作之间的间隙。那样当你开始查找中断问题来源的时候，你就知道设备是怎样工作的（以及它确实在工作）。而且，可以肯定会有一些这样的错误。

一个简单的时钟驱动

我们要讨论的设备驱动的例子是用来控制 80199EB 处理器的一个时钟/计数器单元。我选择用 C++来实现这个程序——并且本书所有剩下的例子部将用 C++来实现。尽管 C++在访问硬件寄存器方面没有提供比 C 更多的附加帮助，但是在对于这种类型的抽象方面有很多好的理由使用它。最明显的是，C++类与任何 C 特性或者编程技巧相比，允许我们更完全地隐藏实际的便件接口。比如，可以加入一个构造函数，在每一次新的时钟对象被声明的时候自动地设置硬件。这省去了应用软件对初始比例程的显式调用的需要。此外，有可能把对应于设备寄存器的数据结构隐藏在相关类的私有部分。这有助于防止应用程序员从程序的其他部分意外的读写设备案存器。

Timer 类的定义如下：

```
enum TimerState(Idel, Active, Done);
enum TimerType(OneShot, Periodic);

class Timer
{
    public:
        Timer();
        ~Timer();

        int start(unsigned int nMilliseconds, TimerType = OneShot);
        int waitfor();
        void cancel();
```

```
        TimerState state;
        TimerType type;
        unsigned int length;
        unsigned int count;
        Timer * pNext;

    private:
        static void interrupt Interrupt();
};
```

在讨论这个类的实现之前，让我们研究一下前面的声明并且考虑一下设备驱动的总体结构。我们看到的第一个东西是两个枚举类型。**TimerState** 和 **TimerType**。这些类型的主要目的是使得其他的代码更具可读性。从中，我们可以知道每一个软件时钟有一个当前状态——**Idle**、**Active** 或者 **Done**——以及一个类型——**OneShot** 或者 **Periodic**。软件时钟的类型告诉驱动程序当软件时钟到期的时候应该如何处理它们。一个 **Periodic** 类型的软件时钟会被重新启动。

Timer 类的构造函数也是设备驱动的初始化例程。它保证时钟/计数器硬件正在活动地产生每毫秒一次的时钟节拍。类的其他公用函数——**start()**、**waitfor()** 以及 **cancel()**——为软件时钟提供了一个 **API**。这些函数分别允许应用程序员启动一次性和周期性时钟，等待它们到期以及取消运行的时钟。这个接口比 **80188EB** 芯片内的时钟/计数器硬件提供的接口简单和通用。一则，时钟硬件不知道像毫秒这样的人类的时间单位。但是另一方面，因为时钟驱动隐藏了具体硬件的细节，所以应用程序员水远不需要了解它们。

类的数据成员也会有助于你了解设备驱动程序的实现。前三项是回答下面关于软件时钟问题的变量：

- 时钟的当前状态是什么？（空闲、活动或者完成）
- 时钟的类型是什么？（一次性的、周期性的）
- 时钟的总长度是多少？（以时钟节拍为单位）

接下来是另两个数据成员，它们都包含专用于实现这个时钟驱动的信息。变量 **count** 和 **pNext** 只是在活动软件时钟格表的环境中才有意义。这个链表是按照

每一个时钟节拍的次数来排序的。因此，在这个软件时钟被置为溢出之前、count 包含了剩余节拍数的信息（注 1），pNext 是指向下一个最先溢出的软件时钟的指针。

最后，有一个叫作 Interrupt()的私有的方法——我们的中断服务例程。Interrupt()方法被声明为 static，是因为它不被允许操纵具体软件时钟的数据成员。因此，比如，中断服务例程不被允许修改如问时钟的 state。通过使用关键字 static，这个限制会自动地由 C++编译器为我们加上。

从类的声明中要学到的最重要的事情就是，尽管所有的软件时钟都由同一个硬件时钟/计数器单元驱动，但是它们都有自己的私有数据存储区。这允许应用程序员同时创建多个软件时钟而由设备驱动程序在幕后管理它们。一旦你掌握了这个思想，你就可以学习驱动程序中的初始比例程、API 以及中断服务例程。

Timer 类的构造函数负责初始化软件时钟和底层的硬件。对于后者，它负责设置时钟/计数器单元，把中断服务例程的地址插人到中断向量表中，以及启用时钟中断。然而，因为这个方法是一个构造函数，它可能被多次调用（每一个 Timer 对象声明的时候被调用一次），所以我们要足够小心使得只在对它进行第一次调用的时候才执行这些硬件的初始化。否则，时钟/计数器单元可能在不合适的时刻被重启或者和设备驱动程序失步。

这就是下面代码中静态变量 bInitialized 出现的原因。这个变量在声明的时候初值为零，当硬件初始比序列执行之后被设置成 1。当以后对于 Timer 构造函数的调用时，看到 bInltialized 不再是零，就跳过这部分初始化序列。

```
#include "i8010xEB.h"
#include "timer.h"
#define CYCLES_PER_TICK (25000/4) //Number of clock cycles per tick.

/*****************************************************
*
* Method : Timer()
```

注 1：明确地说，count 表示了链表中，该软件时钟之前的所有定时器到期后所剩下的时钟节拍数。

```
*
* Description : Constructor for the Timer class.
*
* Notes:
*
* Returns : None defined.
*
*****************************************************/
Timer::Timer(void)
{
    static int bInitialized = 0;
    //
    // Initialize the new software timer.
    //
    state = Idle;
    type = OneShot;
    length = 0;
    count = 0;
    pNext = NULL;
    //
    //Initialize the timer hardware, if not previously done.
    //
    if(!binitialized)
    {
        //
        // Install the interrupt handler and enable timer interrupts.
        //
        gProcessor.installHandler(TIMER2_INT, Timer::Interrupt);
        gProcessor.pPCB->intControl.timerControl &= ~(TIMER_MASK |
TIMER_PRIORITY);
        //
        // Initialize the hardware device (use Timer #2).
        //
        gProcessor.pPCB->timer[2].count = 0;
        gProcessor.pPCB->timer[2].maxCountA = CYCLES_PER_TICK;
```

```
        gProcessor.pPCB->timer[2].control = TIMER_ENABLE |
TIMER_INTERRUPT | TIMER_PERIODIC;
        //
        // Mark the timer hardware initialized.
        //
        bInitialized = 1;
    }
} /* Timer() */
```

全局对象 **gProcessor** 在头文件对 **i8018EB.h** 中声明。它表示 **Intel 80188EB** 处理器。**i8018xEB** 类是我写过的东西，它包含了简化与处理器以及芯片级外设的交互的方法。其中一个方法就是 **installHandler**，它的工作是把中断服务例程插人到中断向量表中，这个类还包括一个叫作 **PCB** 的全局数据结构，它能够覆盖外设控制块的存储映像寄存器（注 **2**）。与时钟/计数器单元 **2** 相关的三个寄存器正好是这个 **256** 字节的数据结构的一小部分。（为了美观，我把 **PCB** 数据结构实现成为一组嵌套式结构。因此，时钟/计数器单元 **2** 的控制寄自器可以像 **pPCB->timer[2].control** 这样来访问。）

时钟/计数器单元的初始化由这几个部分组成：**count** 寄存器重置为零，把倒计时长度调入 **maxCountA** 寄存器以及在 **control** 寄存器里设置几个位。上面我们做的是启动一个 **1ms** 的周期性时钟，它在每一个周期结束的时候产生一个中断。（这个周期性时钟将作为一个时钟节拍，我们需要用它来创建任意长度的软件时钟。）调人 **maxCountA** 的值可以通过数学的办法来决定，因为它代表了 **1ms** 内输人到时钟/计数器中的时钟周期数。按照 **80188EB** 的数据手册，这会是 **1ms** 内处理器周期数的四分之一。因此，对于一个 **25MHz** 的处理器，就像我们现在正在使用的那个（也就是说，一秒钟 **25000000** 个周期，或者你喜欢的话，一毫秒 **25000** 个周期），**maxCountA** 该被设置成 **25000/4**——正如前面的常数 **CYCLES_PER_TICK** 一样。

注 2：聪明的读者可能会想起在第五章“开始认识硬件”中，我说过 PCB 位于 80188EB 处理器的 I/O 空间。然而，因为在设备驱动的情况下存储映像寄存器更为可能，所以我就把整个的 PCB 定位到物理地址 72000h，即存储空间里。书中剩余的部分将假设这个新的位置。要知道这个重定位是如何可实现的，看看 i8018xEB 类的构造函数就可以了。

一旦硬件被初始化，时钟被建立，就可能启动一个任何长度的软件时钟了，只要那个长度可以表示成为时钟节拍的整数倍。因为我们的时钟节拍是 **1ms** 长，因此应用程序员能够创建在 **1** 到 **65535ms** 之间（**65.536** 秒）的任意长度的时钟。他可以通过调用 **start** 方法实现这一点。

```
/*********************************************************
*
* Method : start()
*
* Description : Start a software timer, based on the tick from the underlying hardware timer.
*
* Notes:
*
* Returns : 0 on success, -1 if the timer is already in use.
*
*********************************************************/
int
Timer::start(unsigned int nMilliseconds, TimerType timerType)
{
    if(state != Idle)
    {
        return (-1);
    }
    //
    // Initialize the software timer.
    //
    state = Active;
    type = timerType;
    length = nMilliseconds / MS_PER_TICK;

    //
    // Add this timer to the active timer list.
    //
    tiemrList.insert(this);
```

```
    return(0);
} /* start() */
```

当一个软件时钟被启动时，数据成员 **state**、**type** 和 **length** 被初始化，时钟被插入到一个叫作时钟列表的活动时钟键表。在时钟列表中的时钟是经过排序的以使第一个到期的时钟在表的顶端。此外，每一个时钟有一个与其相关的 **count** 变量。这个值代表了所有列表前面的时钟到期时该软件时钟剩余的时钟节拍数。总之，这些设计的选择有利于快速更新时钟列表，而付出的代价是减慢了插入和删除的速度。刷新的速度是重要的，因为时钟列表在每次硬件产生时钟节拍中断的时候——每毫秒一次——都要刷新。

图 **7-1** 说明了在运转的时钟列表。记住每一个软件时钟都有它自己的唯一的长度和开始时间，但是一旦它们被插入列表，就只有 **count** 字段和排序有关。在说明的例子中，第一个和第二个时钟同时启动（第一个可能实际上是被重启的，因为它是周期性的）。由于第二个比第一个 **5ms**，因此它晚 **5** 个时钟周期到期。列表中第二个和第三个碰巧是同时结束，虽然第三十时钟可能要多运行 **10** 倍的时间。

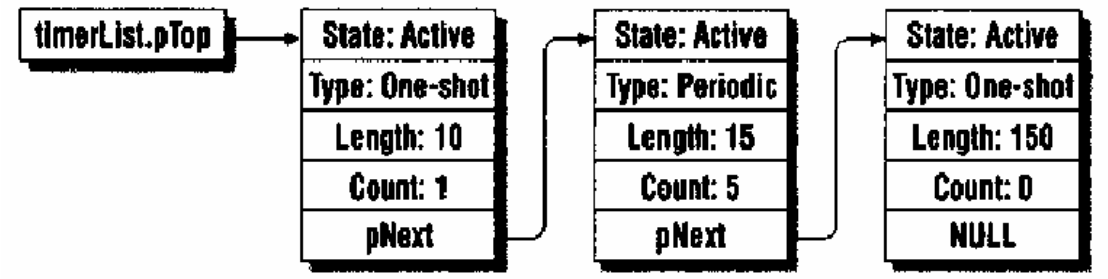

图 **7-1**　运转的时钟列表

中断服务例程的代码如下所示。这个例程被声明为 **void interrupt** 类型。关键字 **interrupt** 是只有针对 **80x86** 处理器的编译器才能理解的 **C/C++**语言的一个延伸。通过这样的声明，我们要求编译器在进出的时候保存和恢复所有处理器的寄存器，而不仅仅是普通函数调用过程中保存的那些寄存器。

```
/*****************************************************
*
* Method : Interrupt()
```

```
*
* Description : An interrupt handler for the timer hardware.
*
* Notes: This method is declared static, so that we cannot
*    inadvertently modify any of the softwre timers.
*
* Returns : None defined.
*
*****************************************************/
void interrupt
Timer::Interrupt()
{
    //
    // Decrement the active timer's count.
    //
    timerList.tick();

    //
    // Acknowledge the timer interrupt.
    //
    gProcessor.pPCB->intControl.eoi = EOI_NONSPECIFIC;

    //
    // Clear the Maximum Count bit ( to start the next cycle ).
    //
    gProcessor.pPCB->timer[2].control &= ~TIMER_MAXCOUNT;
} /* Interrupt */
```

当然，TimerList 类中的 tick 方法做了大部分的工作。这个方法是和链表操作最相关的，但是看上去不是十分的令人兴奋。简单地说，tick 方法开始就把列表最顶端时钟的节拍计数减 1。如果那个时钟的 count 已经为零了，它就把软件时钟的状态改变到然后从时钟列表中删除它。对于任何在那个时刻要到期的时钟，它会执行相同的操作。这些就是位于列表新的头部并且 count 值为零的时钟。创建和启动一个软件时钟之后，应用程序员可以做一些其他的工作，然后检查一下是否软件时钟已经到期。waitfor 方法就是为了这个目的而提供的。这个例

程会堵塞，直到 TimeList.tick 把软件时钟的状态改变成 Done。这个方法的实现如下：

```
/*****************************************************
*
* Method : waitfor()
*
* Description : Wait for the software timer to finish.
*
* Notes:
*
* Returns : 0 on success, -1 if the timer is not running.
*
*****************************************************/
int
Timer:waitfor()
{
    if(state != Active)
    {
        return(-1);
    }

    //
    // Wait for the timer to expire.
    //
    while(state != Done);

    //
    // Restart or idle the timer, depending on its type.
    //
    if(type == Periodic)
    {
        state = Active;
        timerList.insert(this);
    }
```

```
    else
    {
        state = Idle;
    }
    return(0);
} /* waitfor() */
```

关于这段代码要注意到的最重要的事情是 **while(state != Done)**不是无限循环。那是因为，正如我们几个段落之前学过的那样，时钟的状态是被 **timerList.tick()**修改的，而它是被中断服务例程所调用的。实际上，如果我们是细心的嵌入式程序员，我们会把 **state** 声明成 **volatile** 型的变量。这样做会防止编译器不正确地认为时神状态是完成或者是未完成，从而把 **while** 循环优化掉（注 **3**）。

Timer 类的最后一个方法用来取消一个运行的时钟。这容易实现，因为只要把时钟从时钟列表中删除即可，并且把它的状态改变为 **Idle** 状态即可。实际的代码如下：

```
/*********************************************************
*
* Method : cancel()
*
* Description : Stop a running timer.
*
* Notes:
*
* Returns : None defined.
*
*********************************************************/
```

注 3: 使用 waitfor 要小心，这个实现在等待软件时钟改变到完成状态的时候不停地运转。这就是叫作忙等待，并且它是一种既不优雅又没有效率的使用处理器的方法。在第八章“操作系统”中，我们会看到引入操作系统可以提高这个实现。

```
void
Timer::cancel(void)
{
    //
    // Remove the timer from the timer list.
    //
    if(state == Active)
    {
        timerList.remove(this);
    }

    //
    // Reset the timer's state.
    //
    state = Idle;
} /* cancel() */
```

当然，Timer 类还有一个析构函数，我没有在这里列出其代码。不过说明下面这一点就已经足够了。它只是查看软件时钟是否活动，如果是，就从时钟到表中删除它。这样可以防止超出范围的周期性时钟无限地留在时钟到表里，可以防止任问指向“死”时钟的指针留在系统里。

为了保持完整性，最好加入一些公用的方法，也许叫作 poll，它使得使用 Timer 类的用户可以测试软件时钟的状态而不堵塞。限于篇幅，我就不把它加入到我的实现中来，但是加入这样一个例程很简单，它只要返回表达式 state == Done 的当前值即可。然而，要这么做，需要设计某种技术来重启周期时钟而不再调用 waitfor 函数。

Timer 类的另一个潜在的特性是异步回叫（callback）。也就是说，为什么不让软件时钟的创建者向它绑定一个函数。这个函数可以被自动地调用——通过 timerList.tick——每一次时钟到期的时候。当你阅读下面一部分的时候，一定要思考如果使用异步同叫，那么 LED 闪烁程序看上去会是什么样子。这是异步回叫特别适合的一种应用。

修改后的闪烁程序

现在 **Timer** 类已经在我们的掌握之中，因此有可能重写这本书的第一个例子以使得它的定时更加准确。记得在我们的原始实现中，我们基于这样一个事实，对于一个给定的处理器和速度“减 **1** 和比较”操作的长度是固定的。我们只是猜想那可能会多长，然后基于经验的测试修改我们的估计。通过使用 **Timer** 类，我们能够在增加程序可靠性的同时去除这个猜测的工作。

在下面修改后 **LED** 闪烁程序中，你会看到我们现在能够简单地启动一个周期为 **500ms** 的软件时钟，切换指示灯，然后在再次切换指示灯之前等待时钟到期。同时，我们可以执行应用程序其他处理任务。

```
#include "timer.h"
#include "led.h"
/*******************************************************
*
* Function : main()
*
* Description : Blink the green LED once a second.
*
* Notes: This outer loop is hardware-independent. However, it calls
*      the hardware-dependent function toggleLed().
*
* Returns : This routine contains an infinite loop.
*
*******************************************************/
void
main(void)
{
    Timer timer;

    timer.start(500, Periodic);    //Start a periodic 500 ms timer.

    while(1)
    {
```

```
            toggleLed(LED_GREEN);    // Toggle the green LED.

            // Do other useful work here.

            timer.waitfor();  // Wait for the timer to expire.
        }
} /* main() */
```

监视定时器（watchdog）

你可能听说过另一种经常被提到的关于嵌入式系统的时钟是监视定时器。这是一种特殊的硬件，它保护系统免受软件挂起之苦。监视定时器总是从一个大数倒计收到零。这个典型的过程要花费几秒钟来完成。在此期间，嵌入式软件可能“踢出”监视定器，重设它的计数器为原先的大数。如果计数器曾经到达零，那么监视定时器会认为软件被挂起。它重新启动嵌入式芯片，从而重启软件。

在系统运行时软件也许会被意外地挂起，监视定时器是一个恢复系统的普遍方法。比如，假设你的公司的新产品将进入太空。无论你在使用之前做了多少测试，未发现的错误仍然有可能潜伏在软件里，并且其中的一个或者多个可能会使整个系统被挂起。如果软件被挂起，你根本就不能够与它通信，你就不能远程地发出重启的命令。因此你必须把一个自动恢复机制植入系统。这就是监视定时器时钟出现的原因。

监视定时器“踢出”的实现看上去很像这一章里的 **LED** 闪烁程序。不同的是其中不是切换指示灯而是监视定时器被重启。

第八章

操作系统

本章内容:

- 历史和目的
- ADEOS
- 实时特征
- 选择过程

操作系统恐惧症（**osophobia**）**[名]** 嵌入式系统
开发人员普遍存在的恐惧症

嵌入式编程的大多数问题都可以因为操作系统的引人而获益，这里的操作系统可以是你自己写的微内核的操作系统，或是功能完整的商用操作系统。对于这些操作系统，你需要了解其最关键的性能，以及这些性能将来对应用程序的影响。至少，你需要知道嵌入式操作系统从外面看起来应该是什么样的。也许除了深入到一个小的操作系统里，没有其他办法可以让你对各种接口更加理解。这就是我们在本章要做的事情。

历史和目的

在早期的计算机中，没有操作系统一说，应用程序开发人员都要对处理器（**CPU**）和硬件进行彻头彻尾的控制。实际上，第一个操作系统的诞生，就是为了提供一个虚拟的硬件平台，以方便程序员开发。为了实现这个目标，操作系统只需要提供一些较为松散的函数、例程——就好像现在的软件库一样——以便于对硬件设备进行重置、读取状态、写入指令之类的操作。

现代的操作系统则在单处理器上加入了多任务机制，每个任务都是一个软件模块，可以是相互独立的。嵌入式的软件经常是可以划分成小的互相独立的模块。例如，第五章“接触硬件”讲到的打印共享设备就包含三个不同的软件任务：

- 任务 **1**：从计算机的串行口 **A** 接收数据
- 任务 **2**：从计算机的串行口 **B** 接收数据
- 任务 **3**：格式化数据并输送到计算机的并行口（打印机就连接在并行口）

这些任务的划分提供了一个很关键的软件抽象概念，这使得嵌入式操作系统的设计和实现更加容易，源程序也更易于理解和维护。通过把大的程序进行模块化划分，程序员可以集中精力克服系统开发过程中的关键问题。

坦言之，一个操作系统并不是嵌入式或其它计算机系统的必需的组件，它所能做的，也是像时用程序要实现的功能一样。本书中的所有例子都说明了这一点。应用程序执行起来，都是从 **main** 开始，然后进入系统调用、运行、结束。这与系统中只有一个任务是一样的。对于应用程序来说，仅仅是实现使 **LED** 进行闪烁，这就是操作系统的主要功用（屏蔽了很多复杂的操作）。

如果你以前没作过对操作系统的研究，那么，在这里得提醒一下，操作系统是非常复杂的。操作系统的厂商肯定是想使你相信，他们是唯一能生产出功能强大又易用的操作系统的科学家。但是，我也要告诉你：这并不是根困难的。实际上嵌入式操作系统要比桌面操作系统更容易编写，所需的模块和功能更为小巧、更易于实现。一旦明确了要实现了功能，并有一定的实现技能，你将会发现，开发一个操作系统并不比开发嵌入式软件艰难多少。

嵌入式操作系统很小，因为它可以缺少很多桌面操作系统的功能。例如，嵌入式操什系统很少有硬盘或图形界面，因此，嵌入式操作系统可以下需要文件系统和图形用户接口。而且，一般来说，是单用户系统，所以多用户操作系统的安全特性也可以省去了。上面所说的各种性能，都可以作为嵌入式操作系统的一部分，但不是必须的。

AOEOS

下面要讲的是我自己开发的嵌入式操作系统，我称之为 **ADEOS**，意即“得体的嵌入式操作系统（**A Decent Embeded Operation System**）”。**ADEOS** 是一个操作系统，不算太好也不算太差。实际上，一共不超过 **1000** 行程序、**3/4** 的代码是用 **C++**编写的，是与硬件平台无关的。其余的代码是与硬件平台有关的，是用汇编语言开发的。在后面的讨论中，我将详细讲解这些 **C++**代码，并贯穿一些原理性的概念。那些汇编代码，就不作详细讲解了，有兴趣的读者可以下载这些代码并自己进行深入研究。

如果你想用 **ADEOS**（或是对它进行修改）作为嵌入式操作系统，那请便。实际上我很高兴能有人使用它。我花了很大的力气改善 **ADEOS** 的性能。但是，

我不能保证本章的代码对任问想了解嵌入式操作系统的人都很有用。如果你决定自己使用它，那就应该自己努力进一步提高 ADEOS 的性能。

任务

我们已经谈论过多任务的概念和操作系统如何在“同一时间执行多个程序”的想法。那么，操作系统究竟是如何同时执行多个任务的呢？实际上，任务并不是同时执行的，相反，是准并行的，它们只是轮流使用 CPU。就像很多人一起来读一本书，而每个人只有拿到书的时候才能读，但是他们可以传递着读。

操作系统的职责就是决定某一时刻执行哪一个任务，同时，它还要维护每个任务的状态信息，这个信息被称作任务的场景（context），就像书签的作用一样。对于每个读者来说，都有必要拥有自己的书签。书签的用户必须有办法识别书签（也就是说，书签必须有用户的名字），它必须知道上回他读到哪里了。这就是读者的上下文环境信息。

在另外一个任务控制处理器之前，任务场景必须记录处理器的状态，通常包括指向下一条要执行的指令的指针、当前堆栈指针的位置、处理器的标志寄存器和通用寄存器的值。在 16 位的 80x86 处理器上，这些寄存器是 CS、IP、SS 和 SP、Flags、DS、ES、SI、DI、AX、BX、CX 和 DX 等。

为了维护任务及其场景，操作系统必须对每个任务进行单独的维护。用 C 语言编写的操作系统一段把这些信息保存在数据结构中，称作“任务控制块”。ADEOS 是用 C++编写的，能以对象的形式更好地维护这些数据信息，相关的 C++对象为 Task，它包含了操作系统所需的信息，如下所示：

```
class Task
{
    public:
        Task(void(*function)(), Priority p, int stackSize);

        TaskId  id;
        Context context;
        TaskState    state;
```

```
        Priority priority;
        int *        pStack;
        Task *       pNext;

        void (*entryPoint)();

    private:
        static TaskId nextId;
};
```

这个类的许多数据成员变量的含义，要到我们详细讨论过操作系统之后才会显现。**id** 变量是一个单字节的整数（从 **0** 到 **255**），用它来作为任务的标识。换言之，**id** 是书签的标识。**context** 是与处理器相关的数据结构，包含了上一个控制处理器的任务的处理器状态。

任务状态

还记得我刚才说过，在一个时刻只能有一个任务使用处理器（**CPU**）吗？那个正在使用处理器的任务被称作“运行（**running**）”的任务，没有别的什么任务能在同一时刻处于这一状态。准备好运行的任务（目前还没有占用处理器）处于“就绪（**ready**）状态，而正在等待外部通知信号一来就运行的任务处于“等待（**waiting**）”状态。图 **8-1** 显示了这三种状态之间的关系。

“就绪”和“运行”状态的转换是当操作系统选择了一个新任务来运行的时候发生的。在运行之前，任务要处于“就绪”状态，然后处于就绪任务缓冲队列中的新任务被选择并且运行。一旦任务运行起来，就会脱离“就绪”状态，除非操作系充强制它继续保持“就绪”或“等待”状态，或者，它还需要别的什么外部信息才能继续运行。对于后一种情况，我们称作任务被阻塞了。这时，该任务转入“等待”状态，操作系统会再挑选一个处于“就绪”状态的任务运行。所以说，无论当前有多少任务处于“等待”或“就绪”状态，有且只有一个任务是处于“运行”状态的。

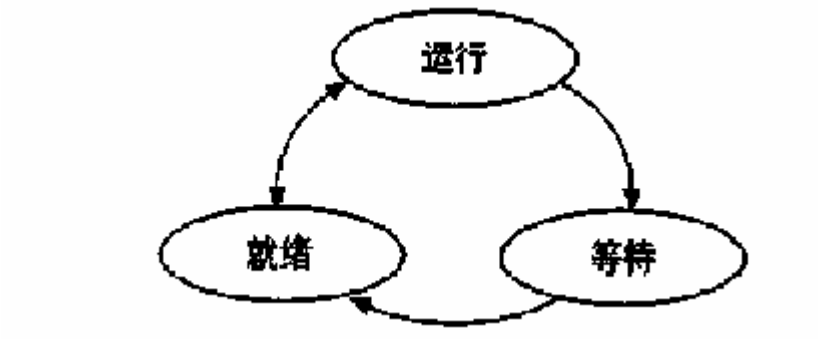

图 8-1 一个任务可能的状态

下面就是 ADEOS 中如何定定义任务状态：

enum Taskstate (Ready, Running, Waiting)

值得注意的是，只有操作系统的“任务调度器”可以决定任务的状态。最新创建的任务以及因等待外部信息而停止的任务。被“任务调度器”放入“就绪”队列。

任务调度机制

作为 ADEOS 的开发者（或是其他操作系统的开发者），你需要知道如何创建和使用任务。就像别的抽象数据结构，Task 类有自己的成员函数。ADEOS 的的任务接口比别的大多数操作系统要简单一些，因为它只是创建一个新的 Task 对象。一旦创建，ADEOS 任务继续在系统中存在，直到相关的函数返回。当然，这也许永远不会发生（意即 ADEOS 任务也许永远不会结束），但是，如果一旦发生了，那么该任务就会被操作系统删除掉。

Task 的构造函数如下所示。调用者通过构造函数的参数分配一个函数，一个权限值，和一个可选择的新任务的堆栈大小。第一个参数，fUnCtion，是一个指向 C/C++语言或汇编语言的函数指针，该函数是要在新任务的上下文环境中运行的。该函数不需要任何输人参数，也不返回任何结果。第二个参数 P，是一个单字节的整数（从 1 到 255），代表了任务的权限级别，这个权限级别是与别的任务相对而言的，在任务调度器选择新的任务运行的时候会用到（p 的值越大，表示权限越高）。

```
TaskId  Task::nextId = 0

/*************************************************************
*
* Method :  Task()
*
* Description :   Create a new task and initialize its state.
*
* Notes :
*
* Returns :
*
*************************************************************/
Task:Task(void (*function)(), Priority p, int stackSize)
{
    stackSize /= sizeof(int);   //Convert bytes to words.
    enterCS();              //Critical Section Begin

    //
    // Initialize the task-specific data.
    //
    if          = Task::nextId++;
    state           = Ready;
    priority = p;
    entryPoint   = function;
    pStack        = new int[stackSize];
    pNext         = NULL;
    //
    // Initialize the processor context.
    //
    contextInit(&context, run, this, pStack + stackSize);
    //
    // Insert the task into the ready list.
    //
    os.readyList.insert(this);
```

```
    os.schedule();    // Scheduling Point
    exitCS();     // Critical Section End
} /* Task() */
```

注意这个例程的功能块被两个函数 **enterCS()**和 **exitCS()**的调用包围。在这些调用之间的代码块叫作临界区（**critical section**）。临界区是一个程序必须完整执行的一部分。也就是说，组成这一个部分的指令必须没有中断地按照顺序执行。因为中断可能随时发生，保证不受到中断的唯一办法就是在执行关键区期间禁止中断。因此在关键区的开始调用 **enterCS** 以保存中断的允许状态以及禁止进一步的中断。在关键区尾部调用 **exitCS** 以恢复前面保存的中断调用。我们会看到在下面每一个例程中都应用了同样的技巧。

在前面代码中，有几个在构造函数里调用的其他例程，但是在这里我没有空间列出。它们是 **contextInit()**和 **os.readyList.insert()**例程。例程 **contextInit()**为任务建立了初始的设备场景。这个例程必定是处理器专用的，因此是用汇编语言写的。

contextInit()有四个参数。第一个是一个指向待初始比的设备场景数据结构指针。第二个是一个指向启动函数的指针。这是一个特殊的 **ADEOS** 函数，叫作 **run()**，它被用来启动一个任务，并且如果以后相关的函数退出了，它被用来做其后的清理工作。第三个参数是一个指向新任务对象的指针。这个参数被传递给 **run()**，因此相关的任务就能够被启动。第四个和最后一个参数是指向新任务栈的指针。

另一个函数调用是 **os.readyList.insert()**。这个函数把新任务加入到操作系统内部的就绪任务列表中。**readyList** 是一个 **TaskList** 类型的对象。这个类是那些具有 **insert()**和 **remove()**两个方法的任务（按照优先级排序）的链表。感兴趣的读者如果想知道这些函数是如何实现的就应该下载和研究其 **ADEOS** 的源代码。你将在下面的讨论中了解到更多有关就绪列表的问题。

调度程序

任何操作系统的核心和灵魂是它的调度程序（**scheduler**）。操作系统的这部分决定了在给定的时间用哪一个就绪任务有权使用处理器。如果你曾经写过用于主流操作系统的软件，那么你可能会熟悉一些普通的调度算法：先进先出、

短任务优先以及循环法（**round robin**）。这些是用于非嵌套式系统的简单调度算法。

应用程序接口（API）

关于嵌入式系统最烦人的事情之一就是它们缺乏一个公共的 API。对于那些希望在基于不同操作系统的产品之间共享应用程序代码的公司来说，这是一个特别问题。我曾经工作过的一个公司甚至在操作系统之上创建它们自己的层，只是为了把它们的应用程序从这些操作系统之间的差别里分用出来。但是可以肯定，这不过是把创建另一个 API 的问题加入到总的问题中来。

每一个嵌入式的操作系统的基本功能大致一样。每一个函数或者方法代表了操作系统可以向应用程序提供的一种服务。但是没有那么多不同的可能的服务。经常是这种情况：两个实现之间真正的不同只是在于函数和方法的名称。

这个问题已经持续了几十年了，并且在可见的日子里也不会解决。然而，与此同时的 Win32 和 POSIX API 已经分别占据了 PC 机和 Unix 工作站。因此，为什么没有出现一个嵌入式操作系统的类似标准呢？不是因为缺乏尝试。实际上，原始 POSIX 标准（IEEE 1003.1）的作者们也为实时操作系统创建了一个标准（IEEE 1003.4h）。一些很像 Unix 的嵌入式操作系统（让人想起了 VxWorks 和 LynxOS）是符合这个标准 API 的。然而，对于绝大部分的应用程序员来说，必须为每一个使用的操作系统学习一个新的 API。

幸运的是，有了一线曙光，Java 编程语言已经支持嵌入式的多任务和任务同步。那意味着不管 Java 程序运行在什么样的操作系统中，创建和处理任务以及使它们同步的机理是一样的。由于这个以及其他的一些原因，对于嵌入式程序员来说 Java 会是一个很好的语言。我希望有一天大家需要一本关于嵌入式系统用 Java 语言编程的书，并且因此不再需要这样介绍了。

先进先出（**FIFO**）调度描述了一个像 **DOS** 一样的操作系统，它不是一个多任务的操作系统。相反地，每一个任务一直运行到它结束为止，并且直到那时下一个任务才被启动。然而，在 **DOS** 中一个任务可以把自己挂起，因此为下一个任务释放出处理器。那恰恰也是旧版本的 **Windows** 操作系统如何允许用户从

一个任务切换到其他任务的工作机理。**Windows NT** 之前任何微软的操作系统都不包含真正的多任务。

短任务优先描述了一个近似的调度算法。唯一的不同是，每一次运行的任务在完成或者挂起自己的时候，下一个被选择的任务是那个需要最小处理器完成时间的任务。短任务优先在早期的主流系统中是普遍的，因为它能使最大多数用户满意。（只有那些有最长任务的用户才会警告和抱怨。）

循环法是这三个调度算法中唯一的一个调度算法，它可以使运行中的任务被占先，也就是说，使它在运行的过程中遭到中断。在这种情况下，每一个任务都运行一个预先决定的时间。那个时间间隔过后，运行的程序被操作系统占先，然后下一个任务有机会运行。被占先的任务直到其他所有的任务都有机会运行一轮之后才再次开始运行。

不幸的是，嵌入式操作系统不能利用任何这样简单的调度算法。嵌入式操作系统（特别是实时操作系统）几乎总是需要一种方式来共享处理器，这种方式使得最重要的任务只要在它们需要的时候就可以获得处理器的控制。因此，大部分的嵌入式操作系统利用一个支持占先的基于优先级的调度算法。这是一个流行的说法，也就是说，在任何一个给定的时刻，正在使用处理器的任务保证是就绪任务中优先级最高的任务。低优先级的任务必须等待直到高优先级任务使用它处理器之后才可以继续它们的工作。“占先”这个词的意思就是如果一个高优先级的任务就绪之后任何任务都能被操作系统中断。调度程序在有限的一组时间点检测这种情况，这些时间点叫作调度点。

当使用一个基于优先级的调度算法时，有一个备份的策略也是必要的。这是在几个就绪任务具有相同优先级时使用的调度算法。最普通的备份调度算法是循环调度法。然而，为了简单起见，对于我的备份策略我只用了一个 **FIFO** 调度实现。因此，**ADEOS** 的使用者实尽可能的注意给每一个任务分配一个唯一的优先级。这不应该成为一个问题，因为 **ADEOS** 支持的优先级与任务数（一个 **ADEOS** 最多支持 **255** 个）一样多。

在 **ADEOS** 中地调度程序是在一个叫作 **Sched** 地类中实现的。

```
class Sched
{
```

```
    public:
        Sched();

        void start();
        void schedule();

        void enterIsr();
        void exitIsr();

        static Task * pRunningTask;
        static TaskList readyList;

        enum SchedState(Uninitialized, Initialized, Started);

    private:
        static SchedState state;
        static Task idleTask;
        static int interruptLevel;
        static int bSchedule;
};
```

定义了这个类之后，在操作系统的一个模块中这种类型的对象就被初始化了。那样，**ADEOS** 的使用者只要连接 **sched.obj** 以包含一个调度的实例。这个例子叫作 **OS**，它声明如下：

```
    extern Sched os;
```

对于这个全局变量可以在这个应用程序的任何一个部分引用。但是，你很快就会明白在每个应用中只有一个这样的引用是必要的。

调度点

简单地说，调度点是一组导致调用调度程序的操作系统事件。我们已经遇到了两个这样的事件：任务创建和任务的删除。每一个事件期间，方法 **os.schedule**

被调用以选择下一个要运行的任务。如果目前运行的任务仍旧在就绪的任务中具有最高的优先级，那么它将被允许继续使用处理器。否则，接下来，最高优先级的任务将被执行。当然，在任务删除的情况下，选择的总是一个新任务：根据它不再存在的事实，所以目前运行的任务不再处于就绪的状态。

一个第三方的调度点叫作时钟节拍。时钟节拍是一个由时钟中断处触发的周期性事件。时钟节拍提供了一个机会以唤醒那些等待软件时钟到期的任务。这和我们在前而章节中看到的时钟节拍几乎是一样的。实际上，对于软件时钟的支持是嵌入式系统的一个普遍特性。在一个时钟节拍期间，操作系统使每一个活动的软件时钟减值并且检测它们。当一个时钟到期了，所有等待它结束的任务从等待状态变为就绪状态。然后，调度程序被调用以确定新唤醒的任务和此时钟中断之前运行的任务相比是否具有更高的优先级。

在 **ADEOS** 中时钟节拍例程几乎与第七章“外围设备”中的那个例程是一样的。实际上，我们仍旧使用相同的 **Timer** 类。只有这个类的实现被轻微地改变了。这些变化是为了考虑到多个任务可能等待同一个软件时钟的事实。此外，所有对于 ***disable()***和 ***enable()***的调用都被 ***enterCS()***和 ***exitCS()***替换了，并且时钟节拍的长度从 **1 ms** 增加到 **10 ms**。

就绪列表

调度程序使用一个叫作就绪列表的数据结构来追踪处于就绪状态的任务。在 **ADFOS** 中，就绪列表被实现成为一个普通的按照优先级排序的链表。因此，这个列表的头总是具有最高优先级的就绪任务。跟踪一个调度程序的调用，这和追踪当前运行的任务一样。实际上，唯一不是这种情况的时候是在重调度过程中。图 **8-2** 为操作系统运行时的就绪列表。

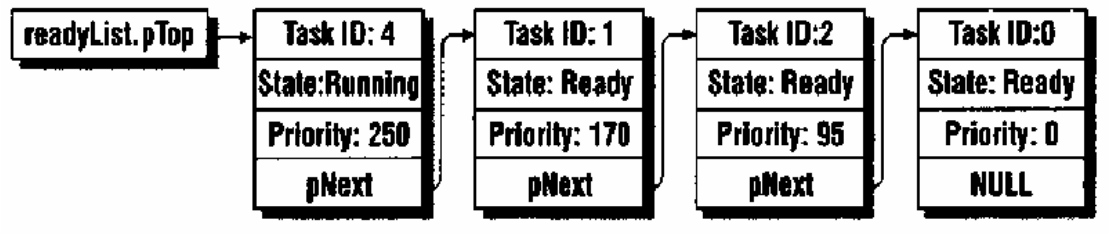

图 **8-2**　就绪列表

像这样排过序的链表的主要好处在于调度程序容易通过它选择下一个要运行的任务。（它总是在最顶端。）不幸的是，在查询时间和插入时间之间需要权衡。由于数据成员 **readyList** 总是直接指向就绪的任务，查询时间被最小化了。然而，每次一个新的任务改变到就绪状态时，在 **insert()**方法中的代码必须一直搜寻就绪列表，直到它发现一个任务，这个任务比正在插入的任务的优先级更低。新就绪的任务被插在那个任务的前面。结果，插入时间正比于就绪列表中任务的平均数量。

空闲任务

如果没有任务处于就绪状态，空闲任务将被执行。空闲任务看起来与其他操作系统中的一样。它只是一个不作任何事情的循环。在 **ADEOS** 中，空闲任务对于应用开发者来说是完全隐藏的。然而，它确实有一个任务的 **ID** 和优先级（两者都是零）。空闲任务总是被认为处于就绪的状态（当它不运行的时候），并且由于它的优先级最低，所以他总是出现在就绪列表的末尾。那样，调度程序在没有任何其他任务处于就绪状态时会自动的找到它．其他的任务则被认为是用户任务，以区分空闲任务。

调度程序

因为我利用一个排序的链表来保存就绪列表，所以调度程序容易实现。它只是检测正在运行的任务与最高优先级的就绪任务是否是一个任务和同样的优先级。如果是，调度任务完成。否则，它会启动一个从前者到后者的设备场景的切换。这里是用 **C++**实现的代码。

```
/*************************************************************
*
* Method :  schedule()
*
* Description :   Select a new task to be run.
*
* Notes : If this routine is called from within an ISR, the
*         schedule will be postponed until the nesting level
*         returns to zero.
```

```
*
*       The caller is responsible for disabling interrupts.
*
* Returns : None defined.
*
*********************************************************************/
void
Sched::schedule(void)
{
    Task * pOldTask;
    Task * pNewTask;

    if(state != Started) return;

    //
    // Postpone rescheduling until all interrupts are completed.
    //
    if(interruptLevel != 0)
    {
        bSchedule = 1;
        return;
    }
    //
    // If there is a higher-priority ready task, switch to it.
    //
    if(pRunningTask != readyList.pTop)
    {
        pOldTask = pRunningTask;
        pNewTask = readyList.pTop;

        pNewTask->state = Running;
        pRunningTask = pNewTask;

        if(pOldTask == NULL)
        {
```

```
            contextSwitch(NULL, &pNewTask->context);
        }
        else
        {
            pOldTask->state = Ready;
            contextSwitch(&pOldTask->context, &pNewTask->context);
        }
    }
} /* schedule() */
```

正如你从代码中看到的一样，有两种情况调度程序不启动设备场景的切换。第一种情况是如果多任务没有被启用的时候。这是必要的，因为有时应用程序员想要在调度程序真正启动之前创建其任务的一些或者全部。在那种情况下，应用程序的 **main** 例程会类似于下面这段程序。每次一个 **Task** 对象被创建的时候，调度程序就被调用（注 **1**）。然而，因为调度程序为了检查多任务是否已被启动而检测变量 **state** 的值，所以直到 **start()**被调用以后才会发生设备场景的转换。

```
#include "adeos.h"

void taskAfunction(void);
void taskBfunction(void);

/*
 * Create two tasks, each with its own unique function and priority.
 */
Task taskA(taskAfunction, 150, 256);
Task taskB(taskBfunction, 200, 256);

/*****************************************************************
*
* Method :  main()
```

注 1：记住，任务的创建是我们调度点其中的一个。如果调度程序已经启动了，新任务仍然可能是最高优先级的就绪任务。

```
* Description : This is what an application program might look like
*       if ADEOS were used as the operating system. This
*       function is responsible for starting the operating system only.
*
* Notes :     Any code placed after the call to os.start() will never
*          be executed. This is because main() is not a task,
*          so it does not get a chance to run once the scheduler is started.
*
* Returns : This function will never return!
*
*********************************************************************/
void
main(void)
{
    os.start();
    // This point will never be reached.
} /* main() */
```

因为这是一段重要的代码，所以让我重新讲解你正在看的东西。这是一个你可能作为 **ADEOS** 用户写出的应用代码的例子。你在开始加入头文件 **adeos.h** 和声明你的任务。在你声明了你的任务和调用 **os.start** 之后，任务函数 **taskAfunction** 和 **taskBfunction** 开始执行（以伪并行的方式）。当然，**taskB** 在这两个任务中具有最高的优先级（**200**），因此它将先运行。然而，只要它由于任何原因放弃对处理器的控制，其他的任务就有机会运行。

另一种 **ADEOS** 调度程序不进行设备场景切换的情况是在中断进行期间。操作系统跟踪目前运行的中断服务例程的嵌套级别，并且只有在嵌套级是零的时候才允许设备场景切换。如果调度程序是从一个 **ISR** 调用的（就像它在时钟节拍期间一样），**bSchedule** 标志被置位，表明只要最外层的中断处理程序退出，调度程序就应该再次被调用。这个被延迟的调度加快了整个系统中断响应的时间。

设备场景切换

从一个任务转变到另一个任务的实际过程叫作设备场景切换。因为设备场景

是处理器专用的，实现设备场景切换的实现也是这样。那意味着它总是要用汇编来写。与其向你展示我在 ADEOS 中使用的 80x86 专用的汇编代码，不如我用一种类 C 的伪代码来展示设备场景切换例程。

```
void
contextSwitch(PContext pOldContext, PContext pNewContext)
{
    if(saveContext(pOldContext))
    {
        //
        // Restore new context only on a nonzero exit from saveContext().
        //
        restoreContext(pNewContext);
        // This line is never executed!
    }
    // Instead, the restored task continues to execute at this point.
}
```

例程 contextSwitch()实际上是被调度程序凋用，而调度程序又在那此终止中断的系统调用中被调用，因此它不一定在这里终止中断。此外，由于调用调度程序的操作系统调用是用高级语言写的，所以大部分运行任务的寄存器已经被保存到它自己当地的栈中了。这减少了例程 saveContext()和 restoreContext()需要做的工作。它们只需要关心指令指针，栈指针以及标志位的保存。

例程 contextSwitch()的实际行为是很难仅仅通过看前面的代码来理解的。大部分的软件开发者以连续的方式思考问题，认为每一行代码会紧接着上一条代码破执行。然而，这个代码实际为并行地执行了两次。当一个任务（新任务）转变到运行状态，另一个（旧任务）必须同时返回到就绪状态。想一下新任务当它在 restoreContext()代码中被恢复的时候就会明白。无论新任务以前做什么，它在 saveContext 代码里总是醒着的——因为这就是它的指令存放的地方。

新任务如何知道它是否是第一次（也就是，在准备休眠的过程）或者是第二次（醒来的过程）从 saveContext()中出来的呢？它确实需要知道这个差别，因此我不得不用一种有点隐蔽的方法来实现 saveContext()。例程 saveContext()不是保存了准确的目前的指令指针。实际上是保存了一些指令前面的地址。那样，

当保存的设备场景恢复的时候，程序从 **saveContext** 中另一个下同的点继续。这也使得 **saveContext** 可能返回不同的值：当任务要休眠的时候为非零，当任务唤起的时候为零。例程 **contextSwitch()**利用这个返回的值来决定是否调用 **restoreContext()**。如果不进行这个检测，那么与这个新任务相关的代码永远不会执行。

我知道这可能是一个复杂的事件序列，因此我在图 **8-3** 中说明了整个的过程。

任务的同步

尽管我们常常把多任务操作系统中的任务看作是完全独立的实体来谈论，但是这个刻画不完全准确。所有的任务是在一起工作以解决一个大问题，必须偶尔和另外一个任务通信使得它们的活动同步。比如，在一个打印机共享的设备中。打印机任务直到其中一个计算机任务向它提供了新的数据之前是没有任何工作可见的。因此，打印机和计算机任务必须互相通信以协调它们获得共享数据。做到这一点的一种办法是使用一种叫作互斥体（**mutex**）的数据结构。

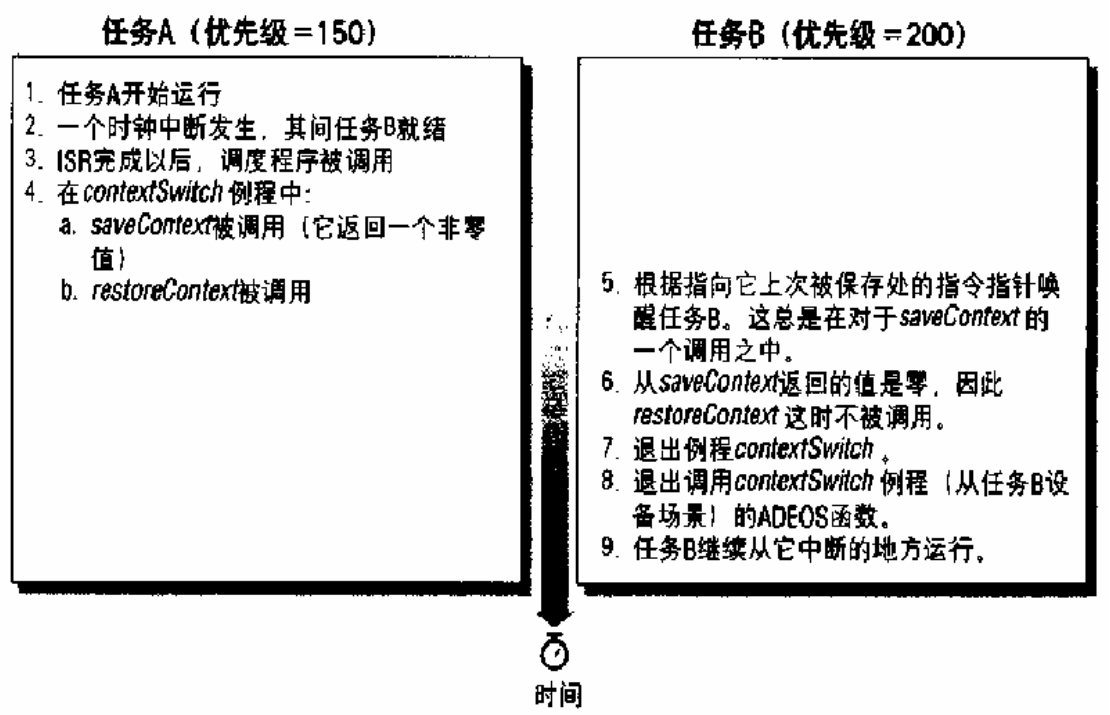

图 8-3　设备场景切换

很多的操作系统提供互斥体来协助实现任务的同步。然而，它并不是唯一可以实现任务间同步的机制。其他的机制有：信号灯、消息队列以及监视器。可

是如果你有一种这样的数据结构，就可以实现其他的任何一种。实际上，互斥体本身就是一种特殊的信号灯，称为二元信号灯，或者是互斥信号灯。

你可以把一个互斥体想成是一个能感知多任务的二元标志。因此，与一个具体互斥体相关的含义是由软件设计者来选择，被每一个使用它的任务来理解的。比如，由打印机和计算机共享的数据缓存也许需要一个和它们相关的互斥体。当这个二元标志位被设置的时候，共享数据缓存被假定是其中一个任务在使用。所以在读写缓存中的数据之前，其他的任务必须等待，直到标志位被清除（然后又被它们自己设置）。

我们说互斥体是感知多任务的，这是因为设置和清除二元标志位都是最基本的操作。也就是说，这些操作不可能被中断。一个任务可以安全地改变互斥体的状态而不会在改变之中冒设备场景切换的危险。如果设备场景切换发生，二元标志位会处于一种不可预知的状态并且会导致任务之间的死锁。互斥体设置和清除操作的不可分割性得到了操作系统的加强。在读取和修改二元标志使之前操作系统禁止中断。

ADEOS 包含了一个 **Mutex** 类。利用这个类，应用软件可以创建和销毁互斥体，等待一个互斥体被清除，然后再设置它，或者清除一个前面设置的互斥体。后面的两个操作被分别称为获取和释放一个互斤体：

这里是 **Mutex** 类的定义：

```
class Mutex
{
    public:
        Mutex();

        void take(void);
        void release(void);

    private:

        TaskList waitingList;
```

```
        enum{Available, Held} state;
    }
```

创建一个新的 **Mutex** 类的过程简单。每一次一个新的互斥体对象被启动的时候下面的构造函数会自动执行。

```
/*****************************************************************
*
* Method :  Mutex()
*
* Description : Create a new mutex.
*
* Notes :
*
* Returns :
*
*****************************************************************/
Mutex::Mutex()
{
    enterCS();          ////// Critical Section Begin

    state = Available;
    waitingList.pTop = NULL;

    exitCS();           ////// Critical Section End
} /* Mutex() */
```

所有的互斥体是在 **Available** 状态下创建的，并且和一个开始为空的等待任务链表相关。当然，一旦你创建了一个互斥体，就有必要通过某种方法改变它的状态，因此我们要讨论的下一个方法是 **take()**。一般来说，这个例程会被一个任务在它读写一个共享资源之前调用。当对 **take()**的调用返回时，调用它的任务时于那个资源独占式的存取将受到操作系统的保证。这个例程的代码如下：

```
/*****************************************************************
*
```

```
* Method :  take()
*
* Description : Wait for a mutex to become available, then take it.
*
* Notes :
*
* Returns :        None defined.
*
*****************************************************************/
void
Mutex::take(void)
{
    Task * pCallingTask;

    enterCS();        ////// Critical Section Begin

    if(state == Available)
    {
        //
        // The mutex is available, Simply take it and return.
        //
        state = Held;
        waitingList.pTop = NULL;
    }
    else
    {
        //
        // The mutex is taken. Add the calling task to the waiting list.
        //
        pCallingTask = os.pRunningTask;
        pCallingTask->state = Waiting;
        os.readyList.remove(pCallingTask);
        waitingList.insert(pCallingTask);

        os.schedule();          // Scheduling Point
```

```
        // When the mutex is released, the caller begins executing here.
    }

    exitCS();           ////// Critical Section End
}   /* take() */
```

关于 take 方法最巧妙的是：如果互斥体被另一个任务占据（也就是二元标志位已经被设置），那么直到互斥体被那个任务释放之前，调用 take()的任务会被挂起。这就好像是告诉你的配偶你要打个盹，在晚餐准备好了之后让他或者她叫醒你。甚至多个任务等待一个互斥体也是可能的。实际上，与每一个互斥体相关的等待列表是按照优先级排序的。因此最高优先级的等待任务总是第一个被唤醒。

接下来的方法是用来释放一个排斥体。尽管这个方法可能被任问其他的任务调用，但是人们希望只有先前调用过 take 方法的任务可以请求它。不像 take 方法，这个例程不会阻塞。然而释放互斥体的一个可能结果是唤起一个优先级更高的任务。那种情况下，释放排斥体的任务会立刻被迫（被调度程序）放弃对处理器的控制，支持高优先级的任务。

```
/*************************************************************
*
* Method :  release()
*
* Description : Release a mutex that is held by the calling task.
*
* Notes :
*
* Returns :       None defined.
*
*************************************************************/
void
Mutex::release(void)
{
    Task * pWaitingTask;
```

```
    enterCS();          ////// Critical Section Begins

    if(state == Held)
    {
        pWaitingTask == waitingList.pTop;

        if(pWaitingTask != NULL)
        {
            //
            // Wake the first task on the waiting list.
            //
            waitingList.pTop = pWaitingTask->pNext;
            pWaitingTask->state = Ready;
            os.readyList.insert(pWaitingTask);

            os.schedule();          // Scheduling Point
        }
        else
        {
            state = Available;
        }
    }
    exitCS();           ////// Critical Section End
} /* release() */
```

临界区

互斥体的使用主要是为了保护共享资源。共享资源包括全局变量，存储缓存或者是被多任务存取的设备寄存器。对于那些在某一时刻只有一个任务可以访问的资源，互斥体可以用来限制它们的存取。它就好像控制十字路口交通的指示灯。记住在多任务环境中，你通常不知道运行时任务会按照什么顺序执行。当一个任务突然被一个高优先级的任务中断的时候，它可能在向一个存储缓存中写入一些数据。如果高优先级的任务也要修改同一块存储区域那么糟糕的事情就会发生了。至少，一些低优先级任务的数据会被覆盖。

死锁和优先级倒置

互斥体是一个实现各任务访问共享资源时同步的有力工具。但是，它们也不是没有问题。需要当心的两个最重要的问题是死锁和优先级倒置。

一旦任务与资源存在循环依赖的时候，死锁就会发生。最简单的例子是有两个任务，它们都需要两个互斥体：A 和 B。如果一个任务获得了 A 在等待 B，同时另一个任务获得了 B 并且在等待 A，那么两个任务都在等待一个永远不可能发生的事件。

当一个高优先级的任务被阻塞，等待一个被低优先级任务保留的互斥体的时候，优先级倒置就发生了。这可能不像一个大问题——毕竟，互斥体正在仲裁对于共享资源的访问——因为高优先级任务知道有时低优先级任务会利用它共享的资源。然而，考虑一下，如果出现一个优先级介于两者之间的第三方任务会发生什么事情。

这种情况在图 8-4 中做了说明。这里有三个任务：高级、中级以及低级。低级的任务首先就绪（由凸沿表示），那以后很快就获得互斥体。现在当高级任务就绪，它必须被阻塞（用阴影区表示）直到低优先级的任务处理定它们共享的资源。问题在于中级任务，由于它不需要对那个资源访问，所以抢先于低优先级的任务执行，于是它就会耽误高优先级任务对于处理器的使用。对于这个问题人们已经提出了很多解决方案。其中最普遍的方案叫作“优先级的继承”。这个解决方案在高优先级任务等待互斥体的时候，就把低优先级的任务的优先级升到那个高优先级任务的优先级。一些操作系统在它们的互斥体实现中加入了这个修正，但是大部分都没有。

访问共享资源的代码段包含临界区。我们已经在操作系统里面看到了类似的东西。在那里，我们只是在临界区内禁止中断。但是任务不能够（这是明智的）禁止中断。如果它们被允许这样做，其他的任务——甚至是那些不共享相同资源的高优先级任务——不能够在此期间执行。因此我们希望并且也需要一个在任务内部不禁止中断又能够保护临界区的机制。互斥体提供了这种机制。

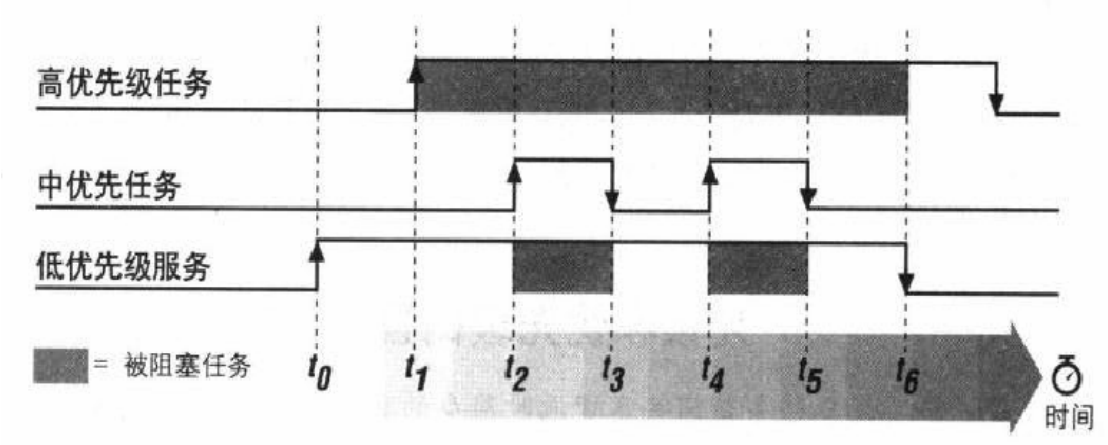

图 8-4　优先级倒置的例子

你现在已经学会了一个简单嵌入式操作系统要学习的每一样东西。它的基本元素有：调度程序和调度点、设备场景切换例程、任务的定义以及任务间通信的机制。每一个有用的嵌入式操作系统都会具有这些相同的元素。然而，你不必总是搞清楚它们是如何实现的。你通常只要把操作系统看待成一个应用程序员可以依赖的黑盒子。你只是为每一个任务编写代码，在必要的时候对操作系统做调用。操作系统会保证这些任务相对于其他任务来说在合适的时间运行。

实时特征

工程师经常使用“实时”这个术语来描述这样一个计算问题：延迟的响应和错误的响应是同样糟糕的。这些问题被说成是具有最终期限，而嵌入式系统常常在这种限制之下操作。比如，如果控制作解锁闸的嵌入式软件漏掉了一个最终期限，那么你可能会发现自己出了事故（你甚至会被杀死!）。因此对于实时嵌入式系统的设计者来说，了解每一件他们所能了解的有关硬件和软件的行为及性能的事情，是极其重要的。这一部分里我们要讨论实时使作系统特征，这些特征是实时系统的共同成分。

实时系统的设计者花费大量的时间关心最糟糕情况下的系统性能。他们必须不断地回答自己如下的问题：最坏情况下，在人压迫闸板的操作和一个中断信号到达处理器之间要多少时间？最坏情况的中断潜伏期即：在中断到达与启动相关中断服务例程之间的时间，是多少？在最坏的情况下软件对于触发刹车机构的响应时间是多少？平均或者预期情况的简单分析在这种系统中是不够的。

大部分商用嵌入式操作系统被设计为可以运行在实时系统之中。在理想的情况下，这意味着它们在最坏情况下的性能都被很好地了解并且记录在案。要得到与众不同的“实时操作系统”（**RTOS**）的头衔，一个操作系统应该是确定性的，并且保证最坏情况的中断等待时间以及设备场景切换的时间。如果给出这些特征以及你的系统中任务和中断的相对优先级，就可能分析软件在最坏情况下的表现了。

如果每一个系统调用在最坏情况下的执行时间都是可以计算的，那么这个操作系统就被说成是确定性的（**deterministic**）。一个对待其 **RTOS** 实时行为采取慎重态度的操作系统供应商通常会发布一个数据表单，该表单提供了每一个系统调用要求的最小、平均及最大数目的时钟周期。这些数字对于不同的处理器可能不同。但是如果算法在一个处理器上是确定性的，那么在其他任何的处理器都是如此（实际的时间可能会有差别），这样期望是合理的。

中断等待时间是从一个中断信号到达处理器开始到启动相关的中断服务例程为止所经历的时间长度。当一个中断发生的时候，处理器在执行 **ISR** 时必须采取几个步骤。首先，处理器必须完成现在正在执行的指令。这大概花费不到一个时钟周期，但是一些复杂的指令需要花费比这更多的时间；下一步，必须识别中断类型。这是由处理器硬件来做的，不会减慢或者挂起正在运行的任务；最后，只要中断是允许的，那么和中断相关的 **ISR** 就会被启动。

当然，如果中断在操作系统中被禁用。那么最差情况的中断等待时间会增加一个它们被关闭的时间。但是正如我们刚才看到的那样，有很多地方中断是被禁止的。这些就是我们前面讨论的关键区，并且没有其它的方法来保护它们。每一个操作系统都会在一段不同的时间内禁止中断，因此时于你来说了解你的系统的要求是很重要的。某一个实时过程可能要求保证中断响应时间为 **1**μs，而另一个可能只需要 **100**μs。

操作系统的第三个实时特征是执行一个设备场景切换所需的时间。这是重要的，因为它代表了整个系统的开销。比如，想一下任间任务在它阻塞之前的平均执行时间为 **100us**，但是设备场景切换的时间也是 **100us**。这种情况下，处理器的一半的时间完全花费在设备场景切换例程上了。此外，没有不可思议的数字，并且实际的时间通常是机处理器而定的，因为它们依赖于必须保留的寄存器的数目以及它们的位置。所以一定要从你想要使用的操作系统的供应商那里得到这些数据。那样，就不会有任何最后关头的惊奇了。

选择过程

尽管我前面说过写自己的操作系统是如何的容易，但是如果你可以负担得起的话，我还是强烈地建议你买一个商业的操作系统。我再说一遍：我强烈建议购买一个商业的操作系统。而不是写一个自己的操作系统。我知道几个不错的操作系统，只要几千美金就可以得到。考虑到写一个操作系统所花的时间，几乎从任何角度来说那都是一个不错的买卖。实际上，很多种操作系统都很经济并且可以满足大部分工程的需要。在这一部分，我们要讨论选择一个最适合你工程需要的商业操作系统的过程。

商业操作系统形成了一个功能、性能以及价格的统一体。那些在低端范围内的操作系统只是提供一个基本的调度程序以及一些其他的系统调用。这些操作系统通常并不昂贵，提供你可以改动的源代码，并且不需要支付版税。**Accelerated Technology** 公司的 **Nucleus** 和柯达公司的 **AMX** 都属于这一类型(注 **2**)，和任何的 **DOS** 的嵌入式版本一样。

范围另一端的操作系统一般会包括很多有用的功能而不仅仅是调度程序。对于实时性能它们可能会有更强（好的）的保证。然而，这些操作系统可能会很贵，启动成本在**$10000** 和**$50000** 之间，装载人 **ROM** 内的每一个拷贝都要支付版税。然而，这个价格经常包括免费的技术支持和培训以及一系列配套的开发工具。风河（**Wind River**）系统公司的 **VxWorks**、**Integrated** 系统公司的 **pSOS** 以及 **Microtec** 公司的 **VRTX** 都属于这一类。这些是市场上最流行的三个实时操作系统。

在这两个极端之间的那些操作系统除了基本的调度程序之外还有一些功能，对于实时性能它们提供了一些合理的保证。前期成本和版税也是合理的，这些操作系统通常不包括源代码，并且技术支持可能需要另外收费。在前面没有提到的大部分商业操作系统就是属于这一类。

有如此众多的操作系统和特性要选择，很难决定哪一个最适合作的工程。试着把你的处理器，实时性能以及预算要求放在首位。这些都是你不可改变的标准，因此你可以用它们把范围缩小到十几个或者更少的产品。然后和所有这些

注 2；请不要抱怨。我不是贬低这些操作系统。事实上，从我的经验，、我高度推荐它们作为优质、低成本的商业解决方案。

操作系统的厂商联系以获得更加详细的技术信息。

在这一点上，很多人基于对现有的交叉编译器、调试工具以及其他开发工具的兼容性作出他们的选择。但是决定哪些附加的特性对于你的工程最重要实际上是你的责任。无论你决定买哪一个，其基本的内核与本章描述的这个操作系统大致是一样的。差别可能是在于支持的处理器、最大及最小的存储要求、外接式附加软件模炔（网络协议栈、设备驱动以及 **Flash** 文件系统是常见的例子）以及对于第三方开发工具的兼容性。

记住选择一个商业操作系统最好的理由是，使用在分测试过的东西是有益的，因此商业操作系统比你内部开发的（或者从一本书上免费得到的）内核要更可靠。因此你要从操作系统销售商那里找的最重要的就是经验。如果你的系统要求实时性能，你就一定要采用一个曾经在很多实时系统中成功应用的操作系统。比如，查出 **NASA** 为它最近的任务使用了哪个操作系统。我很乐意打赌它是一个不错的选择。

第九章

合成一个整体

本章内容:

- 应用程序的概述
- 闪烁指示灯
- 打印“Hello, World!”
- 利用串行端口
- Zilog 85230 串行端口控制器

一个高水平的技术无异于一个奇迹。
——*Arthur C. Clarke*

在这一章里，我试图把到目前为止所有我们讨论过的单元合在一起，使之成为一个完整的嵌入式的应用程序。在这里我没有把很多新的素材加入到讨论中，因此本章主要是描述其中给出的代码。我的目的是描述这个应用程序的结构和它的源代码，通过这种方式使你对它不再感到神奇。完成这一章以后，你应该对于示例程序有一个完整的理解，并且有能力开发自己的嵌入式应用程序。

应用程序的概述

我们将要讨论的这个应用程序不比其他大部分编程书籍中找到的“**Hello，World**”例子更复杂。它是对于嵌入式软件开发的一个实证，因此这个例子是出现在书的结尾而不是开始。我们不得不逐渐地建立我们的道路通向大部分书籍甚至是高级语言编译器认为是理所当然的计算平台。

一旦你能写“**Hello, World**”程序，你的嵌入式平台就开始着上去很像任何其他编程环境。但是，嵌入式软件开发过程中最困难的部分——使自己熟悉硬件，为它建立一个软件开发的过程，连接到具体的硬件设备——还在后面呢。最后，你能够把你的力量集中于算法和用户界面，这是由你要开发的产品来确定的。很多情况下，这些程序的高级方面可以在其他的计算机平台上开发，和我们一直在讨论的低级的嵌入式软件开发同时进行，并且只要把高级部分导入

嵌入式系统一次，两者就都完成了。

图 9-1 包含了一个“Hello, World!”应用程序的高级的示意图。这个应用程序包括三个设备驱动程序，ADEOS 操作系统和两个 ADEOS 任务。第一个任务以每秒 10Hz 的速度切换 Arcom 板上的红色指示灯。第二个每隔 10 秒钟向主机或是连接到位子串口上的哑终端发送字符串“Hello，WOrld!”。

图 9-1 “Hello World!”应用程序

这两个任务之外，图中还有三个设备的驱动程序。这些驱动程序分别控制着 Arcom 板子的指示灯、时钟以及串行端口。虽然通常把设备驱动画在操作系统的下面，但是我把它们三个和操作系统放在同一个级别，是为了着重说明它们事实上依赖于 ADEOS 比 ADEOS 依赖于它们更多。实际上，ADEOS 嵌入式操作系统甚至不知道（或者说下关心）这些设备驱动是否存在于系统之中。这是嵌入式操作系统中设备驱动程序和其他硬件专用软件的共性。

程序 main()的实现如下所示。这段代码简单地创造厂两个任务，启动了操作系统的日程表。在这样一个高的级别上，代码的含义是不言而喻的。事实上、我们已经在上一章中讨论了类似的代码。

```
#include "adeos.h"

void flashRed(void);
void helloWorld(void);

/*
 * Create the two tasks.
 */
```

```
Task taskA(flashRed, 150, 512);
Task taskB(helloWorld, 200, 512);

/*************************************************
 *
 * Function :   main()
 *
 * Description :    This function is responsible for starting the ADEOS scheduler
only.
 *
 * Notes :
 *
 * Returns :    This function will never return!
 *
 *************************************************/
void
main(void)
{
    os.start();
    // This point will never be reached.
} /* main() */
```

闪烁指示灯

正如我早先说的。这个应用程序所做的两件事之一是使红色指示灯闪烁。这可以通过下面的代码来做到。这里函数 **flashRed()**作为一个任务来执行。然而，如果忽略这一点以及新的函数名，那么这段代码和第七章“外围设备”中我们研究过的 **LED** 闪烁函数几乎是一样的。在这一级上的唯一差别就是新的频率（**10HZ**）和指示灯的颜色（红色）。

```
#include "led.h"
#include "timer.h"

/*****************************************************************
```

```
 *
 * Function :   flashRed()
 *
 * Description : Blink the red LED ten times a second.
 *
 * Notes : This outer loop is hardware-independent, However, it
 *       calls the hardware-dependent function toggleLed().
 *
 * Returns :    This routine contains an infinite loop.
 *
 ****************************************************************/
void
flashRed(void)
{
    Timer timer;

    timer.start(50, Periodic); // Start a periodic 50 ms timer.
    while(1)
    {
        toggleLed(LED_RED);// Toggle the red LED.
        timer.waitfor();   // Wait for the timer to expire.
    }
} /* flashRed() */
```

LED 闪烁程序主要的改变在这段代码中是看不见的。这些主要的变化是对 toggleLed()函数和 Timer 类进行了修改以兼容多任务的环境。toggleLed()函数就是我现在称之为指示灯驱动的东西。一旦你开始这样考虑这个问题，也许会考虑把驱动重写成一个 C++类，并已加入新的方法以明确地设置和清除指示灯，这个方法也是讲得通的。但是，采用和第七章一样的实现方法，并且仅使用一个互斥体来保护 P2LTCH 寄存器不被一个以上的任务同时访问，就已经足够了（注 1）。这里是修改后的代码：

```
#include "i8018xEB.h"
#include "adeos.h"

static Mutex gLedMutex;
```

```
/*****************************************************************
 *
 * Function :   toggleLed()
 *
 * Description : Toggle the state of one or both LEDs.
 *
 * Notes : This version is ready for multitasking.
 *
 * Returns :    None defined.
 *
 *****************************************************************/
void
toggleLed(unsigned char ledMask)
{
    gLedMutex.take();

    // Read P2LTCH, modify its value, and write the result.
    //
    gProcessor.pPCB->ioPort[1].latch ^= ledMask;
    gLedMutex.release();

} /* toggleLed() */
```

注 1：在早先的那个 toggleLed()函数中存在竞争的情况。为了理解这一点，返回去查看这段代码，并且假想两个任务在共享指示灯，第一个任务恰好调用了那个切换红色指示灯的函数，突然之间，第一个任务被第二个任务占先，此时，在 toggleLed()函数中，指示灯的状态都被读入并存入一个处理器的寄存器。现在第二个任务导致两个指示灯的状态都被再次读入并且存入另一个处理器的寄存器，然后两个指示灯的状态都被修改以改变绿色指示灯的状态，并且导致结果写出到 P2LTCH 寄存器。当中断的任务重新启动时，它已经有一个指示灯状态的拷贝。但是这个拷贝不再准确了。当发生这个变化后，指示灯变成红灯，并且比新的指示灯状态写到 P2LTCH 寄存器。这样，第二个任务的改变将会被撤销。加入一个互斥体可以消除这个潜在的危险。

第七章中的时钟驱动程序在应用于一个多任务的环境之前，必须对它做类似的改动。然而这时不存在竞争的情况（注 2）。我们需要利用一个互斥体来消除 **waitfor** 方法中的轮询。通过把一个互斥体和每一个软件时钟关联，我们可以使任何一个在等待时钟的任务休眠，因此，释放了处理器以执行就绪的低优先级的任务。当等待的时钟到期了，休眠的任务会被操作系统唤起。

为此，一个指向互斥体的指针，叫做 **pMutex**，被加入到类的定义中：

```
class Timer
{
    public:
        Timer();
        ~Timer();

        int start(unsigned int nMilliseconds, TimerType = OneShot);
        int waitfor();
        void cancel();

        TimerState state;
        TimerType type;
        unsigned int length;

        Mutex * pMutex;

        unsigned int count;
        Timer * pNext;
    private:
        static void interrupt Interrupt();
};
```

每次构造函数创建软件时钟的时候，这个指针被初始化。此后，无论何时一个时钟对象被启动，它的互斥体可以像下面这样获得：

注 2：记得时钟硬件只初始化一次（在第一次构造函数请求的时候），并且此后时钟专用寄存器只能由一个函数（即中断服务例程）来读写。

```
/*****************************************************************
```

```
 *
 * Function :   start()
 *
 * Description : Start a software timer, based on the tick from the
 *       underlying hardware timer.
 *
 * Notes : This version is ready for multitasking.
 *
 * Returns :    0 on success, -1 if the timer is already in use.
 *
 ***************************************************************/
int
Timer::start(unsigned int nMilliseconds, TimerType timerType)
{
    if(state != Idle)
    {
        return(-1);
    }
    //
    // Take the mutex. It will be released when the timer expires.
    //
    pMutex->take();

    //
    // Initialize the software timer.
    //
    state = Active;
    type = timerType;
    length = nMilliseconds / MS_PER_TICK;

    //
    // Add this timer to the active timer list.
    //
    timerList.insert(this);

    return(0);
```

```
} /* start() */
```

通过在时钟启动的时候获得互斥体，我们可以保证在同一个互斥体释放之前没有其他任务（甚至是启动时钟的任务）能够再获得它。并且这在时钟自然到期（中断服务程序）或是人为取消（通过取消的办法）之前是不会发生的。所以在 **waitfor()**中的轮询可以由 **pMutex->take()**来替换如下：

```
/*****************************************************************
 *
 * Function :   waitfor()
 *
 * Description : Wait for the software timer to finish.
 *
 * Notes : This version is ready for multitasking.
 *
 * Returns :    0 on success, -1 if the timer is not running.
 *
 *****************************************************************/
int
Timer::waitfor()
{
    if(state != Active)
    {
        return(-1);
    }
    //
    // Wait for the timer to expire.
    //
    pMutex->take();

    //
    // Restart or idle the timer, depending on its type.
    //
    if(type == Periodic)
    {
        state == Active;
```

```
        timerList.insert(this);
    }
    else
    {
        pMutex->release();
        state = Idle;
    }

    return(0);
} /* waitfor() */
```

当时钟最后到期时，中断服务程序将会释放互斥体，调用的任务会在 **waitfor()** 中被唤起。唤起的过程中，互斥体已经为下一个运行的时钟获得。互斥体只有当时钟是 **OneShot** 类型时才会被释放，因此，是不会自动重启的。

打印“Hello，World!”

应用程序的其他部分是一个定时向串行端口打印文本字符串“**Hello，World**”的任务。此外，用时钟的驱动程序来创建这个周期。然而，这个任务还依赖于串行端口的驱动程序，这是我们以前没有见过的。串行端口驱动内容将在本章的最后两部分进行描述．但是在这里先说明这个用到它的任务。为了理解这个任务，你只需要知道串行端口是一个 **C++**类，并且利用 **puts()**方法可以从那个端口打印字符串。

```
#include "timer.h"
#include "serial.h"

/*****************************************************************
 *
 * Function:    helloWorld()
 *
 * Description: Send a text message to the serial port periodically.
 *
 * Notes: This outer loop is hardware-independent.
```

```
 *
 * Returns: This routine contains an infinite loop.
 *
 ************************************************************/
void
helloWorld(void)
{
    Timer timer;
    SerialPort serial(PORTA, 19200L);

    timer.start(10000, Periodic);  //Start a periodic 10 s timer.

    while(1)
    {
        serial.puts("Hello, World!");  //Output a simple text message.
        timer.waitfor();       //Wait for the timer to expire.
    }
} /* helloWorld() */
```

尽管周期具有不同的长度，但是这个任务的总体结构和 **flashRed()**函数的结构是一样的。因此，剩下来我们要讨论的唯一的事情就是串行端口的组成。我们从描述通用串行端口的接口开始，到 **Arcom** 板子上可以找到的专用串行控制器结束。

利用串行端口

在应用程序级，一个串行端口只是一个双向的数据通道。这个通道的每一端通常端接一个叫做串行通信控制器（**SCC**）的硬件设备。在 **SCC** 中的每一个串行端口——每一个控制器至少有两个串行端口——一边连接到的嵌入式处理器，另一边连接到电缆（或者是某一设备的连接器）。电缆的另一端通常是一个具有自己内部的串行通信控制器的主机（或者是一些其他的嵌入式系统）。

当然，串口的实际作用是依赖于应用的。但是总的概念是这样的：在两个智能系统之间或者是这样一种设备和人类操作之间交换数据流。一般来说，通过

串行端口可以传送和接收的最小的数据单元是 8 位的字符。因此，在传输之前，二进制数据流需要被识别成字节。这个限制类似于 C 的 stdio 库的限制，因此，从其接口借用一些编程的约定是有意义的。

为了支持串口通信和仿效一个 stdio 风格的接口，我定义了 SerialPort 类，如下所示。这个类把应用程序对串口的使用抽象成双向数据通道，使接口尽可能的和我们曾见过的类似。除了构造函数和析构函数之外，这个类还包括四个方法——putchar()（注 3）、puts()、getchar()和 gets()——以发送和接收字符及字符串。这些程序定义得就与在任何 ANSI C 兼容版本的头文件 stdio.h 里的一样。

```
#include "circbuf.h"

#define PORTA 0
#define PORTB 1

class SerialPort
{
    public:

        SerialPort(int port,
              unsigned long baudRate = 19200L,
              unsigned int txQueueSize = 64,   //Transmit Buffer Size
              unsigned int rxQueueSize = 64);  // Receive Buffer Seize
        ~SerialPort();

        int putchar(int c);
        int puts(const char *s);

        int getchar();
        char * gets(char *s);
```

注 3: 你可能会奇怪为什么这个方法接受一个整数参数而不是一个字符。毕竟我们正通过串口发送 8 位字符，对不？嗯，别问见。我只是试图遵守 ANSI C 库标准，也在问自己这个问题呢。

```
    private:

        int channel;

        CircBuf * pTxQueue; //Transmit Buffer
        CircBuf * pRxQueue; //Receive Buffer
};
```

注意私有数据成员 **channel**、**pTxqueue** 和 **pRxqueue**。这些是在构造函数中初始化的，用作串行端口驱动的硬件专用部分的接口，这会在下一节介绍。关于这个接口我一会儿会详细的说，但是现在只要知道 **SerialPort** 类没有包含任何对应于具体串口控制器的特殊代码。这一切都隐藏在它引用的 **SCC** 类里面。

让我们看一下 **SerialPort** 的构造函数。这个程序负责初始化三个私有的数据成员和设置 **SCC** 中请求的数据通道。

```
#include "scc.h"

static SCC scc;

/*****************************************************************
 *
 * Function:    SerialPort()
 *
 * Description: Default constructor for the serial port class.
 *
 * Notes:
 *
 * Returns: None defined.
 *
 *****************************************************************/
SerialPort::SerialPort(int port,
            unsigned long baudRate;
            unsigned int txQueueSize,
            unsigned int rxQueueSize)
{
```

```
    //
    // Initialize the logical device.
    //
    switch(port)
    {
        case PORTA:
            channel = 0;
            break;
        case PORTB:
            channel = 1;
            break;
        default:
            channel = -1;
            break;
    }
    //
    // Create input and output FIFOs.
    //
    pTxQueue = new CircBuf(txQueueSize);
    pRxQueue = new CircBuf(rxQueueSize);


    //
    // Initialize the hardware device.
    //
    scc.reset(channel);
    scc.init(channel, baudRate, pTxQueue, pRxQueue);
} /* SerialPort() */
```

一旦 **SerialPort** 对象被创建了，就可以使用前述的方法发送和接收数据。比如，在早先的 **helloWorld** 函数中，**puts("Hello, World!")**是把文本字符串送到端口 **A**（也叫作：**SCC** 通道 **0**）。数据按照 **SeriaIPort** 构造函数中由 **baudRate**，参数选择的波特率——**19200bps**，被送到串行端口。

发送和接收方法依赖于分别由 **pTxQueue** 和 **pRxQueue** 指向的循环缓存器。**PTxQueue** 是一个传输缓存器，它在应用程序发送字符的速率大于通道的波特率

时提供了溢出存贮。这通常发生在一个很短的时间里，因此，可以认为传输缓存通常自始至终都没有充满过。类似地，接收缓行，**PRxQueue**，为那些已经被串行端口接收但是还没有被应用程序读取的字节提供溢出存储。默认的情况下，上面的构造函数为它们分别创建 **64** 字节的缓存。然而，缓存的大小可以设置成更小或是更大的值，这要视你应用程序的需要而定。设置时只要简单的替换构造函数中的默认参数就可以了。

发送方法 **putchar()**和 **puts()**的实现说明如下。在 **putchar()**中我们首先检验是否传输缓存已经满了。如果已经满了，我们就向调用者返回一个出错信息，因此它会知道字符没有被发送。否则，我们把新的字符加入到传输缓存，接着确保 **SCC** 传输引擎在运行，然后成功返回。**puts()**方法做了一系列 **putchar** 的调用，对字符串中的每一个字符做一次调用，然后在末尾加入一个换行字符。

```
/*************************************************************
 *
 * Function:    putchar()
 *
 * Description: Write one character to the serial port.
 *
 * Notes:
 *
 * Returns: The transmitted character is returned on success.
 *      -1 is returned in the case of an error.
 *
 *************************************************************/
int
SerialPort::putchar(int c)
{
    if(pTxQueue->isFull())
    {
        return(-1);
    }
    //
    // Add the character to the transmit FIFO.
    //
    pTxQueue->add((char) c);
```

```
    //
    // Start the transmit engine(if it's stalled).
    //
    scc.txStart(channel);

    return(c);
} /* putchar() */

/*****************************************************************
 *
 * Function:    puts()
 *
 * Description: Copies the null-terminated string s to the serial
 *      port and appends a newline character.
 *
 * Notes:   In rare cases, this function may return success though
 *      the newline was not actually sent.
 *
 * Returns:     The number of characters transmitted successfully.
 *      Otherwise, -1 is returned to indicate error.
 *
 *****************************************************************/
int
SerialPort::puts(const char * s)
{
    const char * p;

    //
    // Send each character of the string.
    //
    for( p=s; *p != '\0'; p++)
    {
        if(putchar(*p) < 0) break;
    }
```

```
    //
    // Add a newline character.
    //
    putchar('\n');

    return((p - s) + 1);
} /* puts() */
```

接收方法 **getchar()**和 **putchar()**类似。它首先检查接收缓存是否为空。如果是空的，返回一个出错信息。否则，从接收缓存中移走一个字节的数据给调用者。**gets** 方法反复的调用 **getchar** 直到发现一个换行符或者在单行端口中已经没有数据了。它然后把到那一点为止找到的任何字符串返回。这两个方法的代码如下：

```
/*************************************************************
 *
 * Method:  getchar()
 *
 * Description: Read one character from the serial port.
 *
 * Notes:
 *
 * Returns:     The next character found on this input stream.
 *      -1 is returned in the case of an error.
 *
 *************************************************************/
int
SerialPort::getchar(void)
{
    int c;

    if(pRxQueue->isEmpty())
    {
        return(-1); // There is no input data available.
    }
```

```
    int rxStalled = pRxQueue->isFull();

    //
    // Read the next byte out of the receive FIFO.
    //
    c = pRxQueue->remove();

    //
    // If the receive engine is stalled, restart it.
    //
    if(rxStalled)
    {
        scc.rxStart(channel);
    }

    return(c);
} /* getchar() */

/*************************************************************
 *
 * Method:  gets()
 *
 * Description: Collects a string of characters terminated by a newline
 *      character from the serial port and places it in s.
 *      The newline character is replaced by a null character.
 *
 * Notes:   The caller is responsible for allocating adequate space for the string.
 *
 * Warnings:    This function does not block waiting for a newline.
 *      If a complete string is not found, it will return
 *      whatever is available in the receive queue.
 *
 * Returns:     A pointer to the string.
 *      Otherwise, NULL is returned to indicate an error.
 *
 *************************************************************/
```

```
char
SerialPort::gets(void)
{
    char * p;
    int c;

    //
    // Read characters until a newline is found or no more data.
    //
    for(p = s; (c = getchar()) != '\n' && c >= 0; p++)
    {
        *p = c;
    }

    //
    // Terminate the string.
    //
    *p = '\0';

    return(s);

} /* gets() */
```

Zilog 89230 串行端口控制器

在 **Arcom** 板上的两个串行端口是同一个 **Zilog 85230** 串行通信控制器的一部分。不幸的是，这个特殊的芯片设置和使用起来比较复杂。因此，我决定把串行端口驱动程序划分成两个部分，而不是用早先的特殊设备代码来填充 **SerialPort** 类。上层是我们刚讨论过的类。这个上层的类要通过任何两通道的 **SCC** 来工作，这个 **SCC** 提供面向字节的传输和接收接口以及可以设置的波特速率。这一切所必需的是实现一个设备专用的 **SCC** 类（下面要描述的低屋），这个类具有和 **SerialPort** 类相同的 **reset()**、**init()**、**txStart()**和 **rxStart()**接口。

实际上，**Zilog 85230 SCC** 设备如此难以设置和使用的一个原因是它有很多

的选项，超过了这个简单程序真正的需要。这个芯片不仅能够发送字节而且可以发送不超过八位的字符。除了可以选择波特速率外，它还可以设置一个或者两个通道的很多其他特性，可以支持很多其他的通信的协议。

```
#include "circbuf.h"

class SCC
{
    public:

        SCC();

        void reset(int channel);
        void init(int channel, unsigned long baudRate,
                CircBuf * pTxQueue, CircBuf * pRxQueue);
        void txStart(int channel);
        void rxStart(int channel);

    private:

        static void interrupt Interrupt(void);
};
```

注意这个类也依赖于 **CircBuf** 类。**init()**方法中 **pTxQueue** 和 **pRxQueue** 参数用来为通道建立输入和输出缓存。这使得有可能在 **SCC** 设备内部把其中一个物理通道和逻辑串行端口连接起来。**init()**方法是独立于构造函数的，这样定义的原因是大部分的 **SCC** 芯片控制着两个或者更多的串行端口。构造函数在第一次被调用的时候对它们都进行了重置。然后，调用 **init** 以设置具体某个通道的波特速率及其他的参数。

关于 **SCC** 类，还要说的就是专门针对 **Zilog 85230** 设备的一个内部功能部分。由于这个原因，我决定不在这本书里列出或是解释这段长而复杂的模块。只要说明代码由读写设备寄存器的宏处理接收和发送中断的中断服务程序以及一些排错方法组成，如果接收和发送过程在等待更多数据时被停止，那么这些排错方法可以重新启动它们。有兴趣的读者可以在文件 **scc.cpp** 中找到实际的代码。

第十章

优化你的代码

本章内容:

- 提高代码的效率
- 减小代码的大小
- 降低内存的使用
- 限制 C++的影响

事情应该尽可能简化，而不只是简单一点点，
一爱因斯坦

虽然使软件正确的工作好像应该是一个工程合乎逻辑的最后一个步骤，但是在嵌入式的系统的开发中，情况并不总是这样的。出于对低价系列产品的需要，硬件的设计者需要提供刚好足够的存储器和完成工作的处理能力。当然，在工程的软件开发阶段，使程序正确的工作是很重要的。为此，通常需要一个或者更多的开发电路板，有的有附加的存贮器，有的有更快的处理器，有的两者都有。这些电路板就是用来使软件正确工作的。而工程的最后阶段则变成了对代码进行优化。最后一步的目标是使得工作程序在一个廉价的硬件平台上运行。

提高代码的效率

所有现代的 C 和 C++编译器都提供了一定程度上的代码优化。然而，大部分由编译器执行的优化技术仅涉及执行速度和代码大小的一个平衡。你的程序能够变得更快或者更小，但是不可能又变快又变小。事实上，在其中一个方面的提高就会对另一方面产生负面的影响。哪一方面的提高对于程序更加的重要是由程序员来决定。知道这一点后，无论什么时候遇到速度与大小的矛盾，编译器的优化阶段就会作出合适的选择。

因为你不可能让编译器为你同时做两种类型的优化，我建议你让它尽其所能的减少程序的大小。执行的速度通常只对于某些有时间限制或者是频繁执行的代码段是重要的。而且你可以通过手工的办法做很多事以提高这些代码段的效率。然而，手工改变代码大小是一件很难的事情，而且编译器处于一个更有利的位置，使得它可以在你所有的软件模块之间进行这种改变。

直到你的程序工作起来，你可能已经知道或者是非常的清楚，哪一个子程序或者模块对于整体代码效率是最关键的。中断服务例程、高优先级的任务、有实时限制的计算、计算密集型或者频繁调用的函数都是候选对象。有一个叫作 **profiler** 的工具，它包括在一些软件开发工具组中，这个工具可以用来把你的视线集中到那些程序花费大部分时间(或者很多时间)的例程上去。

一旦你确定了需要更高代码效率的例程，可以运用下面的一种或者多种技术来减少它们的执行时间。

inline 函数

在 **c++**中，关键字 **inline** 可以被加入到任何函数的声明。这个关键字请求编译器用函数内部的代码替换所有对于指出的函数的调用。这样做删去了和实际函数调用相关的时间开销，这种做法在 **inline** 函数频繁调用并且只包含几行代码的时候是最有效的。

inline 函数提供了一个很好的例子，它说明了有时执行的速度和代码的太小是如何反向关联的。重复的加入内联代码会增加你的程序的大小，增加的大小和函数调用的次数成正比。而且，很明显，如果函数越大，程序大小增加得越明显。优化后的程序运行的更快了，但是现在需要更多的 **ROM**。

查询表

switch 语句是一个普通的编程技术，使用时需要注意。每一个由机器语言实现的测试和跳转仅仅是为了决定下一步要做什么工作，就把宝贵的处理器时间耗尽了。为了提高速度，设法把具体的情况按照它们发生的相对频率排序。换句话说，把最可能发生的情况放在第一，最不可能的情况放在最后。这样会减少平均的执行时间，但是在最差情况下根本没有改善。

如果每一个情况下都有许多的工作要做，那么也许把整个 **switch** 语句用一个指向函数指针的表替换含更加有效。比如，下面的程序段是一个待改善的候选对象：

```
enum NodeType {NodeA, NodeB, NodeC}
switch(getNodeType())
```

```
{
    case NodeA:
        ...
    case NodeB:
        ...
    case NodeC:
        ...
}
```

为了提高速度，我们要用下面的代码替换这个 **switch** 语句。这段代码的第一部分是准备工作：一个函数指针数组的创建。第二部分是用更有效的一行语句替换 **switch** 语句。

```
int processNodeA(void);
int processNodeB(void);
int processNodeC(void);

/*
 * Establishment of a table of pointers to functions.
 */
int (* nodeFunctions[])() = { processNodeA, processNodeB, processNodeC };
...
/*
 * The entire switch statement is replaced by the next line.
 */
status = nodeFunctions[getNodeType()]();
```

手工编写汇编

一些软件模块最好是用汇编语言来写。这使得程序员有机会把程序尽可能变得有效率。尽管大部分的 **C/C++**编译器产生的机器代码比一个一般水平的程序员编写的机器代码要好的多，但是对于一个给定的函数，一个好的程序员仍然可能做得比一般水平的编译器要好。比如，在我职业生涯的早期，我用 **C** 实现了一个数字滤波器，把它作为 **TI TMS320C30** 数字信号处理器的输出目标。当时我们有的编译器也许是不知道，也许是不能利用一个特殊的指令，该指令准确地执行了我需要的那个数学操作。我用功能相同的内联汇编指令手工地替

换了一段 C 语言的循环，这样我就能够把整个计算时间降低了十分之一以上。

寄存器变量

在声明局部变量的时候可以使用 **register** 关键字。这就使得编译器把变量放入一个多用选的寄存器，而不是堆栈里。合适地使用这种方珐，它会为编译器提供关于最经常访问变量的提示，会稍微提高函数的执行速度。函数调用得越是频繁，这样的改变就越是可能提高代码的速度。

全局变量

使用全局变量比向函数传递参数更加有效率。这样做去除了函数调用前参数入栈和函数完成后参数出栈的需要。实际上，任何子程序最有效率的实现是根本没有参数。然而，决定使用全局变量对程序也可能有一些负作用。软件工程人士通常不鼓励使用全局变量，努力促进模块化和重入目标，这些也是重要的考虑。

轮询

中断服务例程经常用来提高程序的效率。然而，也有少数例子由于过度和中断关联而造成实际上效率低下。在这些情况中，中断间的平均时间和中断的等待时间具有相同量级。这种情况下，利用轮询与硬件设备通信可能会更好。当然，这也会使软件的模块更少。

定点运算

除非你的目标平台包含一个浮点运算的协处理器，否则你会费很大的劲去操纵你程序中的浮点数据。编译器提供的浮点库包含了一组模仿浮点运算协处理器指令组的子程序。很多这种函数要花费比它们的整数运算函数更长的执行时间，并且也可能是不可重入的。

如果你只是利用浮点数进行少量的运算，那么可能只利用定点运算来实现它更好。虽然只是明白如何做到这一点就够困难的了，但是理论上用定点运算实现任何浮点计算都是可能的。(那就是所谓的浮点软件库。)你最大的有利条件是，你可能不必只是为了实现一个或者两个计算而实现整个 **IEEE 754** 标准。如果真

的需要那种类型的完整功能，别离开编译器的浮点库，去寻找其他加速你程序的方法吧。

减小代码的大小

正如我早先说的那样，当问题归结于减小代码的大小的时候，你最好让编译器为你做这件事。然而，如果处理后的程序代码对于你可得的只读存贮器仍然太大了，还有几种技术你可以用来进一步减少体程序的大小。在本节中，自动的和人工的代码优化我们都要讨论。

当然，墨菲法则指出，第一次你启用编译器的优化特性后，你先前的工作程序会突然失效，也许自动优化最臭名昭著的是“死码删除”。这种优化会删除那些编译器相信是多余的或者是不相关的代码，比如，把零和一个变量相加不需要任何的计算时间。但是你可能还是希望如果程序代码执行了编译器不了解的函数，编译器能够产生那些“不相关”的指示。

比如，下面这段给出的代码，大部分优化编译器会去除第一条语句，因为 ***pControl** 在重写(第三行)之前没有使用过：

```
*pControl = DISABLE;
*pData = 'a';
*pCotrol = ENABLE;
```

但是如果 **pControl** 和 **pData** 实际上是指向内存映像设备寄存器的指针怎么办?这种情况下，外设在这个字节的数据写入之前将接收不到 **DISABLE** 的命令。这可能会潜在地毁坏处理器和这个外设之间的所有未来的交互作用。为了使你避免这种问题，你必须用关键字“**volatile**”声明所有指向内存映像设备寄存器的指针和线程之间(或者是一个线程和一个中断服务程序之间)共享的全局变量。你只要漏掉了它们中的一个，墨菲法则就会在你的工程的最后几天里回来，搅得你心神不宁。我保证。

警告：千万不要误以为程序优化后的行为会和未优化时的一样。你必须在每一次新的优化后完全重新测试你的软件，以确保它的行为没有发生改变。

更糟糕的是，或者退一步说，调试一个优化过的程序是富有挑战性的。启用了编译器的优化后，在源代码中的一行和实现这行代码的那组处理器指令之间的关联关系变得更加微弱了。那些特定的指令可能被移动或者拆分开来，或者两个类似的代码可能现在共用一个共同的实现。实际上，高级语言程序的有些行可能完全从程序中去除了(正如在前面例子里那样)。结果，你可能无法在程序特定的一行上设置一个断点或者无法研究一个感兴趣变量的值。

一旦你使用了自动优化，这里有一些关于用手工的办法进一步减少代码大小的技巧。

避免使用标准库例程

为了减少你的程序的大小，你所能做的最好的一件事情就是避免使用大的标准库例程。很多最大的库例程代价昂贵，只是因为它们设法处理所有可能的情况。你自己有可能用更少的代码实现一个子功能。比如，标准 **C** 的库中的 **spintf** 例程是出了名的大。这个庞大代码中有相当一部分是位于它所依赖的浮点数处理例程。但是如果你不需要格式化显示浮点数值(**%f** 或者**%d**)，那么你可以写你自己的 **sprintf** 的整数专用版本，并且可以节省几千字节的代码空间。实际上，一些标准 **C** 的库(这让我想起 **Cygnus** 的 **newlib**)里恰好包含了这样一个函数，叫作 **sprintf**。

本地字长

每一个处理器都有一个本地字长，并且 **ANSI C** 和 **C++**标准规定数据类型 **int** 必须总是对应到那个字长。处理更小或者更大的数据类型有时需要使用附加的机器语言指令。在你的程序中通过尽可能的一致使用 **int** 类型，你也许能够从你的程序中削减宝贵的几百字节。

goto 语句

就像对待全局变量一样，好的软件工程实践规定反对使用这项技术。但是危急的时候，**goto** 语句可以用来去除复杂的控制结构或者共享一块经常重复的代码。

除了这些技术以外，在前一部分介绍的几种方法可能也会有帮助，特别是查

询表、手工编写汇编、寄存器变最以及全局变量。在这些技术之中，利用手工编写汇编通常可以得到代码最大幅度的减少量。

降低内存的使用

在有些情况下，限制你的应用程序的因素是 **RAM** 而不是 **ROM**。在这些情况下，你想要降低对于全局变量、堆和栈的依赖。这些优化由程序员来做比用编译器来做会更好。

由于 **ROM** 通常比 **RAM** 更加便宜(以每字节为基准)，所以一个可接受的降低全局数据量的策略是把常数移到 **ROM** 中去。如果你用关键字 **const** 声明所有的常数，那么这可以由编译器自动完成。大部分的 **C/C++**编译器把所有它们遇到的常全局数据放入一个特殊的数据段里，这个数据段可以被定位器识别为可分配 **ROM** 的数据段。如果有很多的字符串和导向表数据在运行的时候不发生变化，那么这项技术是最有价值的。

如果有些数据一旦程序运行起来就固定了，但不一定是不变的，那么常数数据段可以改放在一个混合存储设备中。然后，这个存贮设备可以通过网络来更新，或者由一个指派的技术员来完成这个改变。在你的产品要部署的每一个地区的税率就是这种数据的一个例子。如果税率发生了改变，那么存储设备可以更新，但是同时也节省了附加的 **RAM**。

减小栈的大小也可以降低你的程序对于 **RAM** 的需要。有一种方法可以准确地计算出你需要多大的栈。做法是用一个特殊的数据类型填满整个为栈保留的存储区域。然后，在软件运行一段时间之后——最好在正常和紧张两种情况下都运行一下——用调试工具研究被修改过的栈。有一部分仍然包含有你的特殊类型数据的栈存储区，因此可以安全地从栈的大小中减去那部分存储区的大小**(注 1)**。

如果你在使用一个实时的操作系统，就要特别当心栈的大小。大部分操作系

注 1：当然，你可能想在栈中留一点额外的空间——万一你的测试没有持续足够长的时间，或者没有准确地反映所有可能的运行场景。千万不要忘记栈的溢出对于你的软件来说是一个潜在的致命事件，要不惜一切代价避免。

统为每一个任务创建一个分离的栈。这些栈用于函数的调用以及在一个任务的设备场景中遇到的中断服务倒程。你可以通过前面介绍的方式为每一个任务的栈决定其数量。你可以设法减少任务的数量或者切换到一个操作系统，这个操作系统具有分离的为执行所有中断服务例程而建立的“中断栈”。后一种方法可以显著地降低每个任务对栈大小的要求。

堆的大小受限于 **RAM** 在所有的全局数据和栈空间都分配以后剩余的数量，如果堆太小，你的程序就不能够在需要的时候分配内存，因此在废弃它之前一定要把 **malloc** 和 **new** 的结果和 **NULL** 比较。如果你试过了所有这些建议，而且你的程序仍然需要太多的存储空间，那么你除了完全删除所有的堆之外没有别的选择。

限制 C++的影响

在决定写这本书的时候我面临的一个最大的问题是：是否把 **C++**加入到讨论中去。尽管我熟悉 **C++**，但是我不得不用 **C** 和汇编来写几乎所有我的嵌入式软件。而且在嵌入式软件界对于 **C++**是否值得所产生的性能损失的问题存有很大的争议。一般认为 **C++**程序会产生更大的代码，这些代码执行起来比完全用 **C** 写的程序要慢。然而，**C++**给于程序员很多好处，并且我想在这本书中讨论一些这样的好处。因此，我最终决定把 **C++**加入到讨论中来，但是在我的例子中只是使用那些性能损失最小的特性。

我相信很多的读者在他们自己的嵌入式系统编程的时候会面对相同的问题。在结束这本书之前。我想简单地评判一下每一种我使用过的 **C++**特性。并且提醒你一些我没有使用过的比较昂贵的特性。

当然，并不是每一件 **C++**引入的事情都是昂贵的。很多老的 **C++**编译器并入了一个叫作 **C.front** 的技术，这项技术把 **C++**的程序变成 **C**，并且把结果供给标准的 **C** 编译器。这个事实暗示这两种语言之间的句法差别很小，或与运行代价无关(注 2)。只有最新的 **C++**特性，如模板，不能够用这种方式处理。

比如，类的定义是完全有益的。公有和私有成员数据及函数的列表与一个

注 2： 而且，要澄清的是，用 C++编译器编译一个普通的 C 程序不会有损失。

struct 及函数原型的列表没有大的差别。然而，**C++**编译器能够用 **public** 和 **private** 关键字决定，哪一个方法调用和数据访问是允许的或者是不允许的。因为这个决定在编译的时候完成，所以运行时不会付出代价。单纯的加入类既不会影响代码的大小，又不会影响你的程序的效率。

嵌入式的 C++标准

你可能想知道为什么 C++语言的创造者加入了如此多的昂贵的——就执行时间和代码大小来说——特性。你并不是少数，全世界的人都在对同样的一件事情困惑——特别是用 C++做嵌入式编程的用户们。很多这些昂贵的特性是最近添加的，它们既不是绝对的必要也不是原来 C++规范的一部分。这些特性一个接着一个的被添加到正在进行着的“标准化”进程中来。

在 1996 年，一群日本的芯片厂商联台起来定义了一个 C++语言和库的子集，它更加适合嵌入式软件开发。他们把他们新的工业标准叫作嵌入式 C++。令人惊奇的是，在它的初期，它就在 C++用户群中产生了很大的影响。

作为一个 C++标准草案的合适子集，嵌入式 C++省略了很多不限制下层语言可表达性的任何可以省略的东西。这些被省略的特性不仅包括像多重继承性、虚拟基类、运行时类型识别和异常处理等昂贵的特性，而且还包括了一些最新的添加特性，比如：模板、命名空问、新的类型转换等。所剩下的是一个 C++的简单版本，它仍然是面向对象的并且是 C 的一个超集，但是它具有明显更少的运行开销和更小的运行库。

很多商业的 C++编译器已经专门地支持嵌入式 C++标准。个别其他的编译器允许手工的禁用具体的语言特性，这样就使你能够模仿嵌入式 C++或者创建你的很个性化的 C++语言。

默认参数值也是没有损失的。编译器只是加入代码使得在每次函数被无参数调用的时候传递一个默认的值。类似地，函数名的重载也是编译时的修改。具有相同名字但是不同参数的函数在编译过程中分别分配了一个唯一的名字。每次函数名出现在程序中的时候编译器就替换它，然后连接器正确的把它们匹配起来。我没有在我的例子中使用 **C++**的这一特性，但是我这幺做过而没有影响性能。

操作符的重载是另一个我使用过但是没有包括在例子中的特性。无论何时编译器见到这样一个操作符，它只是用合适的函数调用来替换它。因此，在下面列出的代码，最后两行是等价的，性能的损失很容易明白：

```
Complex a, b, c;
c = operator+(a, b)
// The traditional way: Function Call
// The C++ way: Operator Overloading
```

构造函数和析构函数也有一点与它们相关的损失。这些特殊的方法去分别保证每次这种类型的对象在创建或者超出了范围时被调用。然而，这个小量的开销是为减少错误而支付的一个合理代价。构造函数完整地删除了一个 **C** 语言编程中与未初始化数据结构编程错误有关的类。这个特性也被证明是有用的，因为她隐藏了那些与像 **Timer** 和 **Task** 这样复杂的类相关的笨拙初始化顺序。

虚拟函数也具有一个合理的代价收益比。不要深究太多的关于什么是虚拟函数的细节，让我们只是说一下没有它们多态性就是不可能的。而没有多态性，**C++**就不可能是一个真正的面向对象的语言。虚拟函数唯一一个明显的代价是在调用虚拟函数之前附加了一个存储查询。普通的函数和方法调用是不受影响的。

就我的体验来说太昂贵的 **C++**特性有模板、异常事件及运行类型识别。这三个特性都对代码的大小有负面的影响，而且异常事件和运行时类型识别还会增加执行时间。在决定是否使用这些特性之前，你可能要做一些实验来看看它们会怎么样影响你自己的应用程序的大小及速度。

附录

Arcom 的 Target188EB

本书所有的例子都是为 **Target188EB** 这个平台编写、测试的。这块板子是低成本、高速度的嵌入式控制器，由 **Arcom** 控制系统公司设计、生产。本章下面的部分就是关于订购该产品时应该了解的相关信息。

Target188EB 硬件包括如下的组件：

- 处理器：**Intel 80188EB(25MHz)**
- **RAM：128K SRAM(**可增至 **256K)**，电源备份可选
- **ROM 128K EPROM** 和 **128K** 快闪存储器**(**可增至 **512K)**
- 两个 **RS232** 兼容的串口**(**外置 **DB9** 连接器**)**
- **24** 通道的并口
- 三个计时器
- **4** 个中断输入
- **8** 位 **PC/104** 扩展总线接口
- 一个可选的 **8** 位 **STEBus** 扩展接口
- 一个远程调试适配器，包含两个 **RS232** 兼容的串口

这块板子的开发就像 **PC** 机编程一样，很容易。使用板子自带的免费开发工具**(Borland C++**和 **Turbo Assembler** 编译器**)**，你可以开发 **C/C++**或汇编语言的应用程序。同时，在板子的闪存中预先装好了调试监视器，以便于用 **Borland Turbo Debugger** 很容易地发现、修正应用程序的 **bug**。还有一个硬件接口函数库，使得开发人员对板子上的硬件进行编程就像用 **C** 语言的 **stdio** 库一样容易。

本书中的所有例子都是用 **Borland C++ 3.1** 编译、连接、调试的。但是，用 **Borland** 的任何兼容 **80186** 的产品都可以作同样的事情。这包括 **Borland C++ 3.1**、**4.5**、**4.52**。如果你有这几个版本中的某一个，就可以直接应用了。如果没

有，那你得询问一下 Arcom 公司，Borland 公司的最新产品是否可以直接用。

如果订货量少的话，Targetl88EB 板子(具体的编号是 Targetl88EB-SBC)的零售价是 195 美元(注 1)。一般说来是不包括开发工具和完善的技术支持的。但是，Arcom 会免费提供自己的开发工具(原价 l00 美元)给本书(注 2)的读者。如果你需要订货的话。请按如下的联系方式：

Arcom Control System
13510 South Oak Street
Kansas City，MO 64145
Phone：888-941-2224
Fax：816-941-7807
Email：sales@arcomcontrols.com
Web：http://www.arcomcontrols.com/

注 1:　有关这个板的供货与价格的最新信息请与 Arcom 公司联系。

注 2.　在我或 O′Reilly 与 Arcom 公司之间没有任何经济合约关系。我在此推荐这块板子，只是为了感谢 Arcom 公司生产了这么好的产品，并为我写作此书给予了大力支持。

参考书目

开始嵌人式系统的一大困难是参考书籍太少。大多数的此类书籍都是不令人满意的。下面就是本书的参考书目、杂志和其他有用的资源。我并不是想罗列所有的相关资料，其实，很多的书籍部被我省略掉了，因为它们给我留下的印象并不深刻。下面的书籍是值得收藏的、杂志是值得订阅的、网站是值得放入收藏里的。

书籍

Ball, Stuart R. *Embedded Microprocessor Systems: Real World Design* Newton，Mass.：Butterworth-Heinemann, 1996

这本小书收集了很多关于嵌入式系统开发的硬件知识，这是每个嵌入式系统开发工程师都应该好好阅读一下的。

Brown, John Forrest *Embedded Systems Programming in C and Assembly*. New York：Van Nostrand Reinhold，1994

我几年前还不知道有这本书，但是这是好事，否则我就不可能自己钻研了。当我试图竭力从汇编语言中解脱出来的时候，这本书帮了我很大的忙。

Ganssle，Jack G. The Art of Programming Embedded Systems. San Diego: Academic Press，1992

这本书中有很多很有实际应用价值的好的建议。这本书的作者还是《Embedded Systems Programming》(在后面介绍)杂志的撰稿人。Ganssle先生在这本书中收集了很多很有帮助的建议。本书的内容尤其易于查找、检索。

Kernighan，Brian W., and Dennis M. Ritchie. The C Programming Language Englewood Cliffs. N.J.: Prentice-Hall, 1988

来自创造者的对 C 语言语法和语义的简洁阐释。编程人员必备藏书。

Labrosse，Jean J.uC/OS: The Real-Time Kernel. Lawrence，Kans.: R&D Publications，1992

一个实时操作系统，有源码。有注释——本书的价格是物有所值的。对于那些想开发自己的操作系统，或者正在寻找免费源程序的人来说，这本书是很值得购买的。uC/OS(发音为 micro-COS)已经被移植到了很多处理器上，有相当多的用户群体(译注 1)。

Rosenberg, Jonathan B. How Debuggers Work：Algorithms, Data Structures, and Architecture. New York: John P. Wiley&Sons, 1996

如果你对调试器的内部机理感到好奇，本书正适合你。它将帮助你更好的理解调试器和调试监视器的区别，以及调试器与你的程序发生冲突的潜在威胁。

Satir, Gregory, and Doug Brown. C++: The Core Language. Cambridge, Mass: O'Reilly & Associates, 1995

这本书是对 C 语言开发者进行 C++进阶的。如果你还没有收藏一本 C++的书，那这本书很适合你(译注 2)。

Van der Linden，Peter．Expert C Programming：Deep C Secrets. Englewood Cliffs，N.J: Prentice-Hall．1994

本书由 Sun 公司编译器开发组成员撰写，可以使你从一名普通的 C 程序员成长为一位高手。这些高级主题虽然并非全部都是必要的，但只有理解了它们，你才能成为更好的嵌入式系统程序员。本书是一本绝佳的手册，而且写得非常有趣。

译注 1:　此书新版本为《Micro C/OS-II: The Real-Time Kernel》。中文版即将由中国电力出版社出版。

译注 2: 中文版《C++语言核心》已由中国电力出版社出版。

Van Sickle, Ted. Programming Microcontrollers in C Solana Beach, Calif HighText Publications．1994

正如我见过的大多数讲嵌入式系统的书一样，这本书也是针对某个特定处理器系列的。但是，由于这本书写得很不错，而且 **Motorola** 微控制器被广泛应用，有些读者还是感到这本书很有用的。

杂志和会议

Embedded System Programming

一个月刊，其内容主要是关于嵌入式系统软件开发人员在工作中遇到的问题。每篇文章都有评语和建议，而且使用特殊字体，这样更易于阅读。我强烈建议每个人都读一下这个杂志，甚至是放下我这本书几分钟，马上到 **http://www.embedded.com/mag.shtml** 去订购一份。通常需要几个月的时间才能拿得到杂志，但是这是很值得一等的。

另外，你可能想购买一份放在 **CD-ROM** 中的杂志。这张 **CD-ROM** 中存有几百篇文章。详情可以参考 **http://www.embedded.com/cd.html**。

Embedded System Conference

这项技术会议是由《**Embedded System Programming**》杂志承办的，每年有几次，已经有 **10** 年的历史了，每年的参加者都在增加。在会上学习知识的速度要远远快于在书海中苦读。我会想尽办法去参加这个会的。

World Wide Web

Chip Directory (http://www.hitex.com/chipdir/)

一个令人难以置信的关于处理器和外设的信息汇集地。这个站点不是唯一的，但却是同类站点中最棒的，而且它还有许多有价值的链接。

CPU Info Center (http://infopad.eecs.berkeley.edu/ClC/)

这个网站上有大量关于处理器的信息，有一个专栏是关于嵌入式处理器的。

CRC Pitstop (http://www.ross.net/crc)
这个网站是专门讨论 CRC 实现的，包括 Ross William 的“Painless Guide to CRC Error Detection Algorithms。”后者是我所见过的最好的关于讲述 CRC 计算的书。

Electronic Engineers' Toolbox (http://www.eetoolbox.com/ebox.htm)
这个网站专门讨论嵌入式系统、实时软件开发问题和其他 Internet 催生的技术，旨在使你的工作更轻松。网站作者鉴别、索引和总结了成千上万个相关的 Inlernet 资源，你可以在此将它们一网打尽。

Embedded Intel Architecture (http://www.intel.com/design/intarch/)
Intel 主页里关于嵌入式处理器的内容，包括 80188EB 的信息。这个网站上不但有硬件信息，还有很多免费的软件开发、调试工具和很多源码。

news:comp.arch.embedded
一个新闻组，讨论的内容在本书中都有所涉及，主要包括软件开发工具和开发过程、商用实时操作系统的比较，对于关键代码的处理等。

news:comp.realtime
另外一个讨论嵌入式系统的新闻组，更多的讨论内容集中在实时操作系统的任务调度上。在这个新闻组上可以找到一系列 FAQ：http://www.faqs.org/faqs/by-newsgroup/comp/comp.realtime.html。

词汇表

ASIC　专用集成电路

与应用相关的集成电路。集成在一个芯片中的用户设计的硬件。

address bus 地址总线

与处理器及外设相连的电路线。地址总线被处理器用来选择内存地址或指定外设的寄存器。如果地址总线包括 **n** 条电路线，处理器就可以寻址 **2n** 个地址。

application software应用软件

与某个特定嵌入式项目相关的软件模块，这种软件模块一般是不可重复利用的，因为每个嵌入式的系统都不大一样。

assembler　汇编程序

一种软件开发工具，可以把人能读懂的汇编语言转换成处理器可以识别的机器码。

assembler language 汇编语言

一种人能读得懂的处理器指争集。大多数与处理器相关的代码都出须用汇编语言编写。

binary semaphore　二元信号灯

一种信号灯，只有开和关两种状态。也叫互斥体**(mutex)**。

board support package 板级支持软件包

与处理器或硬件平台相关的软件包。一般来说是一些示例源程序。这些源程序必须与别的一些软件包一起编译、链接。

breakpoint 断点

程序中的某个位置，程序执行到这里要被中断，然后控制权要由处理器交到调试器那里。生成、删除断点的方法一般是由调试工具提供的。

CISC(Complex Instruction Set Computer)　复杂指令集计算机

处理器家族的一员。**CISC** 处理器可以产生长度可变的指争和多地址格式，而且只包含很少的寄存器。**Intel 80x86** 处理器都是 **CISC** 的。与 **CISC** 相对的是 **RISC**。

CPU(Central Processing Unit)　中央处理器

处理器中自责执行指争的部件。

compiler 编译器

一种软件开发工具，能把高级语言转换成相应处理器能识别、执行的机器码。

context 场景

当前与处理器的寄存器和标志相关的状态。

context switch 场景切换

在多任务操作系统中，从一个任务切换到另一个任务的过程。一个场景切换的过程包括保存当前正在运行的任务的场景，并把以前保存起来的某个任务的场景加载到处理器中。这一过程的代码一定是与相应处理器有关的。

counting semaphore 计数型信号灯

一种信号灯，用来跟踪多个同类的资源。对于这种信号灯的操作只有在所有要跟踪的资源都在被使用时情况下才不能进行。与这种信号灯相对的是 **Binary semaphorec**。

critical section 临界区

一段不允许被中断的代码，如果被中断则代码运行无法得到正确的结果。参见 **Racecondition**。

cross-compiler 交叉编译

在一种处理器的机器上运行，为另一种处理器的机器产生目标代码。

DMA(Direct Memory Access)　直接存储器存取

一种在外设之间（通常是内存和 **I/O** 设备）传递数据的技术，基本上不需要处理器参与。**DMA** 传输是由 **DMA** 控制器管理的。

DRAM(Dynamic Random-Access Memory)　动态随机存取存储器

一种随机存取存储器，可以暂时的保留所存内容，直至设备中存储的数据按正常间隔刷新。刷新周期通常是由一种称为 **DRAM** 控制器的外围设备负责的。

data bus 数据总线

与处理器及同其通信的所有外围设备相连的一组电气线路。当处理器要读／写某一特殊外围设备的存储器单元或寄存器中的内容时，它将适当的设置地址总线引脚，并接收／发送数据总线的内客。

deadline 时间限制

系统必须完成某个特定运算的时间。参见 **real-time system**。

deadlock 死锁

一种不希望出现的软件状态，其中整个任务集合都被阻塞。等待着只有同一集合中的某一任务才能引发的事件。如果死锁发生，唯一的解决办法是重启系统。但是，如果遵循一定的软件设计实践，避免死锁通常是可能的。

debug monitor 调试监视器

专门设计来用作调试工具的一种嵌入式软件。通常驻留在 **ROM** 中，通过串行端口或网络连接与调试器通信。调试监视器提供一套原语命令：察看和修改存储器单元和寄存器，创建和删除断点，并执行程序。调试器结合这三组原语以实现程序下载和单步调试等高级请求。

debugger 调试器

用于测试和调试嵌入式软件的一种软件开发工具。调试器运行在主机上，通过串行端口或网络连接与目标机相连。使用调试器，你可以将软件下载到目标机上立即执行，还可以设置断点，并检查某个存储器单元和寄存器的内容。

device driver 设备驱动

一种隐藏外围设备细节并提供高级编程接口的软件模块。

device programmer 设备编程器

非易失存储芯片和其他电气可编程设备的编程工具。通常情况下，将可编程设备插入设备编程器的插座中，然后将存储缓冲的内容传入。

digital signal processor(DSP)　数字信号处理器

与微处理器类似的一种设备，不同之处在于，其内部 **CPU** 是为离散时间信号处理的应用专门优化的。除了标准微处理器指令外，数字信号处理器通常还支持一套用来快速执行通用信号处理运算的复杂指令。常见的数字信号处理器产品是 **TI** 公司的 **320Cxx** 和 **Motorola** 公司的 **5600x** 系列。

EEPROM 电可擦除可编程只读存储器

英文读音为“**double-EPROM**”。一种可以通过电气方式擦除的可编程只读存储器**(PROM)**。

EPROM 可擦除可编程只读存储器

一种可以通过紫外线暴晒来擦除的可编程只读存储器**(PROM)**。一旦擦除，它可以借助设备编程器重新编程。

embedded system 嵌入式系统

一种软件、硬件的组合，目的是为了完成某项特定的功能。与之相对的是“**general purpose computer(**通用计算机**)**”。

emulator 仿真器

在线仿真器**(In-Circuit Emulator, ICE)**的简称。一种代替**(**模仿**)**目标板子上处理器的调试工具。仿真器经常被并入目标处理器的特殊打包版本中，使用户可收在程序执行时观察和记录处理器的内部状态。

executable 可执行(文件)

包含了目标代码的文件，可以被读取并执行。

firmware 固件

存放在 **ROM** 中的嵌入式的软件代码。在 **DSP** 编程中，这个术语是很常见的。

flash memory 快闪存储器

RAM-ROM 的一种混合。可以在软件控制下擦除和重写。这种设备分为可分别擦除的多个块**(**称为扇区**)**。快速存储器在需要廉价的非易失数据存储的系统中非常普遍。在某些场合，甚至有用太快闪存储器代替磁盘驱动器的。

general-purpose computer 通用计算机

用作通用计算平台的计算机软硬件的组合。例如，一台个人计算机。与之相对的是 **embedded system(嵌入式系统)**。

heap 堆

用于动态存储分配的存储区域。调用 **C** 的 **malloc** 和 **free** 函数，使用 **C++**的 **new** 和 **delete** 运算，可以在运行时对堆进行操作。

high-level language(HIL)　高级语言

独立于处理器的语言，如 **C** 或 **C++**。使用高级语言编程，可以不必考虑特定处理器的细节，将精力集中在算法和程序上。

host 主机

通过串行端口或网络连接与目标机通信的通用计算机。此术语通常用来区分调试器运　行的平台计算机和用来开发嵌入式系统的计算机。

ICE(In-Circuit Emulator)　在线仿真器

参见 **Emulator(仿真器)**。

I/O(Input/Output)　输入，输出

处理器和外设的接口。最简单的例子是“开关”**(输入)**和 **LED(输出)**。

I/0 map I/0 映射

包含 **I/O** 空间中每个处理器可访问的外围设备的名字和地址范围的表或图。**I/O** 映射对于了解硬件很有帮助。

I/O space I/0　(地址)空间

某些处理器提供的专用存储空间，通常是为 **I/O** 设备的连接保留的。**I/0** 空间中的存储　单元和寄存器只能通过特珠指令存取。例如，**80x86** 系列有称为 **in** 和 **out** 的特殊 **I/O** 空间指令。与之相对的是 **memory space(存储器空间)**。

instruction pointer 指令指针

处理器中的寄存器，含有下一条要执行的指令。也称作 **Program counter(程序计数器)**。

interrupt 中断

一种从外围设备到处理器的异步电信号。当外围设备发出此信号时，我们称发生了一个中断。一旦中断发生，处理器保存当前状态，并执行一个中断服务例程。当中断服务例程退出，处理器的控制将返回到中断前正在运行的软件位置。

interrupt latency 中断等待时间

从中断发生开始，到中断服务程序开始运行之间的时间间隔。

interrup service routine(ISR)　中断服务例程

与特定中断相关的软件代码。

interrupt type 中断类型

与每个中断相关的唯一数字。

interrupt vector 中断向量

中断服务例程的地址。

interrupt vector table 中断向量表

一个包含中断向量的表，以中断类型为索引。这张表包含了处理器关于中断到中断向量的映射，必须由程序员初始化。

intertask communication 任务间通信

任务和中断服务例程用以共享信息，使对共享资源的存取同步的一种机制。最常见的任务间通信的构件是信号灯和互斥体。

linker 链接器

一个工具软件，以一个或几个 **OBJ** 文件为输入参数，输出是可重定位的程序。链接器走在所有的源程序都被编译之后才运行的。

locator 定址器

为链接器生成的可重定位程序分配物理地址的一种软件开发工具。这是嵌入式系统执行之前的最后一个软件准备步骤。所生成的文件称为可执行文件。某些情况下，定址器的功能隐藏在链接器中。

logic analyzer 逻辑分析仪

用于实时捕捉几十乃至成百上千个电气信号的逻辑电平(O 或 1)的硬件调试工具。逻辑分析仪在调试硬件问题和复杂的处理器－外围设备相互作用时非常有用。

memory map 存储器映射

一张包含了外设的名称和地址空间的表，可以被处理器寻址。对于鉴定硬件的类型来说，存储器映射是很好的工具。

memory-mapped I/O 存储器映射 I/O(方法)

一种常见的硬件设计方法，把 I/O 地址放到内存中，而不走放到 I/O 地址空间里。从处理器的角度上看，存储器映射 I/O 设备与内存设备是一样的。

memory space 存储器空间

一个处理器的标堆地址空间。与之相对的是 I/O space(I/O 空间)。

microcontroller 微控制器

微控制器与微处理器是很相似的。主要的不同是，微控制器更适用于嵌入式系统。微控制器包括 CPU、内存(少量的 RAM、ROM)和同一芯片上的外设。例如，805l、Intel 80196 和 Mototola 68HCxx 系列。

microprocessor 微处理器

含有通用的 CPU 的芯片。最常见的例于是 Intel 80x86 和 Motorola 680xO 系列。

monitor 监视器

本书中就是指调试监视器。但是，还有另外一种与任务间通信相关的意思。在那里监视器是一种语言级的同步化部件。

multiprocessing 多处理器(技术)

在一个计算机系统中使用一个以上的处理器(的技术、方法)。所谓“多处理器系统”通常有多个处理器可以通信和共享数据的公共存储器空间。而且，有些多处理器系统还支特并行处理。

multitasking 多任务

多个软件任务游辈(12)兄葱杏的一种情形。每个任务都是相对独立的线程。操作系统通过分割处理器时间片来实现这种幼疾病(12)杏。

mutex 互斥体

一种相互排斥的数据结构，也称二无信号灯。互斥体本质上是一种多任务二元标志可用于保护关键区免于中断。

mutual exclusion 互斥

对共享资源的独占性存取的一种保证措施。在嵌入式系统中，共享资源通常是存储器的一块区域，一个全局变量，或一组寄存器。至斥可以通过使用信号灯或互斥体来实现。

NVRAM(Nonvolatile Random-Access Memory)　非易失随机存取存储器

一种在系统失电的情况下仍然能保留数据的随机存取存储器(RAM)。非易失随机存取存储器经常由一个静态 RAM 和一个长寿电池组成。

OTP

参见 One-time programmable(一次可编程)。

object code 目标代码

一组处理器能读得懂的代码和数据，编译器、汇编程序、链接器和定址器的输出文件都包括目标代码。

object file 目标代码文件

含有目标代码的文件就是目标代码文件，是编译器或汇编程序的输出结果

one-time programmable 一次可编程

任何终端用户只能编程一次的可编程设备，如可编程 ROM。但此术语几乎只用于指代片上可编程 ROM 的微控制器。

opcode 操作码

被处理器作为其指令集中指令的一组二进制代码序列

operating system 操作系统

一组使多任务成为可能的软件。操作系统一般是由一系列函数或软件中断构成的。操作系统负责决定在某小时刻应该运行某个任务，并控制共享资源的存取。

oscilloscope 示波器

可用来观察一个或多个电气线路的电压的一种硬件调试设奋。例如，你可以使用示波器确定目前是否出现了某个中断请求。

PROM(Programmable Read-Only Memory)　可编程只读存储器

一种可以用设备编程器进行编程的只读存储器(ROM)。PROM 只能被写入一次，所以有时也称作“一次性写入存储器”。

parallel processing 并行处理

同时使用两个或多个处理器进行计算的能力。

peripheral 外围设备

除处理器以外的硬件，通常是存储器或 I/O 设备。外围设备可能与处理器在同一个芯片上，这时称为内部外围设备。

physical address 物理地址

在对存储器或寄存器进行寻址时，放在地址总线上的真实地址。

preamptive 占先

当有更高优先级的任务就绪时，如果允许正在运行的任务暂停，就称此调度程序是占先的。非占先调度程序更易实现，但不适用于嵌入式系统。

priority 优先级别

任务重要程度的标志。

priority inversion 优先级倒置

一种不希望出现的软件状态，其中高优先级的任务被延迟，等待存取无法使用的共享资源。实践中，延迟期间此任务的优先级将被降低。

process 进程

进程的概念很容易跟线程、任务搞混。它们之间最重要的区别是：任务是共享

内存空间的。而进程，却有各自独立的内存空间。进程在多用户操作系统中很常见，但是在嵌入式操作系统中却很少见。

processor 处理器

微处理器、微控制器和数字信号处理器的通称。本书中使用此术语的原因是，处理器的实际类型对所描述的嵌入式系统开发影响很小。

processor family 处理器系列

一组相关的处理器，通常是同一厂商的连续几代产品。例如，**Intel** 的 **80x86** 系列始于 **8086**，目前有 **80186**，**286**，**386**，**486**，**Pentium** 等。一个系列中，后出的通常会与前面的产品保持向后兼容。

processor-independent 处理器无关

用来描述与运行的处理器平台无关的软件的一个术语。用高级语言编写的大多数程序是干处理器无关的。与之相对的是 **processor-specific**。

processor-specific 处理器有关

用来描述高度依赖所运竹的处理器平台的软件的一个术语。这些软件的代码通常是用汇编语言编写的。与之相对的是 **processor-independent**。

profiler

一种收集和报告程序执行统计数据的软件开发工具。这些数据包括每个例程调用的次数和花费的总时间，可用来获知那个例程最为关键，从而需要最好的代码效率。

program counter 程序计数器

参见 **instruction pointer**。

RAM(Random-Access Memory)　随机访问内存

一种被广泛使用的内存。其中的内存位置可以按照需要进行读写访问。

RISC(Reduced Instruction Set Computer)　精减指令集计算机

一种处理器的系列。**RISC** 处理器通常只能产生固定长度的指争，并需要大量的寄存器。**MIPS** 处理器就是优异的 **RISC** 处理器。与之相对的是 **CISC**。

ROM(Read-Only Memory)　只读内存

一种被广泛使用的内存，其中的内存位置可以接照需要进行只读访问。

ROM emulator ROM 仿真器

一种代替(模仿)目标板子上处理器的调试工具。**ROM** 仿真器很像调试监视器，但它含有自己的与主机的串行或网络连接。

ROM monitor ROM 监视器

参见 **debug monitor**。

RTOS(Real-Time Operating System)　实时操作系统

一种专门用于实时任务环境的操作系统。

RTOS(Real-Time Operating System)　实时操作系统

一种专门用于实时任务环境的操作系统。

race condition 竞争条件

程序的结果会受到指令执行顺序影响的一种条件。竞争条件仅当中断和／或占先可能且存在关键区的情况下，才会产生。

real-time system 实时系统

有时间限制的任何计算机系统、嵌入式系统或其他系统。以下问题可用于辨别一个系统是否是实时的：响应延迟是否和错误响应一样糟，甚至更糟？或者说，如果运算没有按时完成，会发生什么事？如果没有什么不好的结果，该系统就不是实时的。如果因此会使任务失败或造成严重事故，我们通常称之为“硬”实时系统，意思是系统的时间限制非常严格。介于此两种情况之间的，我们称之为“软”实时系统。

recursion 递归

指软件的自我调用。递归在嵌入式系统中通常是应该避免使用的，因为它经常需要大堆栈。

reentrant 可重入软件

指软件可以同时执行多次。可重入函数可安全的递归调用，或被多个任务所调用。使代码可重入的关键在于，确保任何时候存取全局变量或共享寄存器都是

互斥的。

register 寄存器

处理器或外设的一种内存地址。换句话说，它不是普通的内存。一般来说，寄存器的每一位都对控制更大的外设有作用。

relocatable 可重定位(文件)

包含目标代码的文件。该目标代码已为在目标机上执行基本准备就绪。剩下的步骤是使用定址器修改代码中剩下的可重定位地址。处理生成的结果是可执行文件。

reset address 复位地址

处理器刚加电或重新启动的时候运行的第一条指令的地址。

roset code 复位代码

放在启动地址处的一小段代码。通常是用汇编语言写的，可能就是简单的相当于说一句“跳到启对程序”。

reset vector 启动向量

见 **Reset address**。

SRAM(Static Random-Access Memory)静态随机访问内存

RAM 的一种，**SRAM** 中的数据直到系统关闭电源才丢失。

scheduler 调度程序

操作系统中一个部分，由它决定下一步被运行哪个任务。这种决策的做出，是以每个任务是否就绪，它们的相对优先级，以及具体的调度算法为基础的。

semaphore 信号灯

用于任务间通信的一种数据结构。信号灯通常是由操作系统提供的。

simulator 模拟器

一种运行在主机上，模拟目标处理器的调试工具。模拟器可以在还没有嵌入式硬件的叶候，用来测试软件片断。不幸的是，试图模拟复杂外围设备的交互作用，往往得不偿失。

software interrupt 软件中断

一种由软件指令产生的中断。软件中断通常用于实现断点和操作系统入口点。与之相对的是 **Trap(陷阱)**。

stack 堆栈

包含后进先出队列的存储器区域，用于存储参数，自动变量、返回地址和其他在函数调用中必须保存的信息。在多任务环境中，每个任务都生成自己的堆栈。

stack frame 栈帧

与某个函数调用相关的堆栈区域。

startup code 启动代码

为同高级语言编写的软件做准备的一段汇编语言代码。大多数 **C/C++**交叉编译器都带有启动代码，你可以修改、编译它，并将它与嵌入程序链接起来。

target 目标机

嵌入式系统的另一种称呼，通常在软件开发的时候使用，或者为了区分与嵌入式系统通信的设备而这样称呼。

task 任务

操作系统的中心抽象**(central abstraction)**。每个任务都必须保存自己的指令指针和通用寄存器值。与进程不同，任务共享公用的内存空间，并小心的避免重写其他任务的代码和数据。

thread 线程

任务的另一种称呼，在支特进程的操作系统中经常使用这种称呼。

tracepoint 跟踪点

与断点很相似，只不过它不是使程序停止，而是增加计数。并不是所有的调试工具都支持跟踪点。

trap 陷阱

由处理器自己内部的硬件产生的中断。与之相对应的是 **software Interrupt(**软件中断**)**。

volatile 易失的

没有软件干预即可改变的值，我们称之为易失的。例如，某些 **I/O** 设备寄存器的值会因为外部事件发生改变。**C** 的关键字 **volatile** 可用于提醒编译器注意指向这种寄存器的指针，以确保每次使用数据所读取的都是实际值。

watchdog timer 监视定时器

由软件来进行重置的硬件时钟，如果软件崩溃了，那么该时钟也就报废了，整个系统就会自动重新启动。