单片机与嵌入式系统丛书

http://www.phei.com.cn

嵌入式系统固件揭秘

Embedded Systems Firmware Demystified

• [美] Ed Sutter 著　• 张晓林　等译

为21个平台重建 GNU X-Tools

- 从本质上理解硬件知识
- 全面了解嵌入式系统
- 建立可扩展的开发平台

CMPBooks　電子工業出版社 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

含 光 盘

嵌入式系统固件揭秘

Embedded Systems Firmware Demystified

作者简介：

Ed Sutter于1981年在AT&T贝尔实验室开始他的职业生涯。现在是朗讯的工程师。他于1983年开始从事嵌入式系统开发工作。从8085开始，Ed Sutter已经使用过大多数常见的CPU。他致力于硬件、固件及软件开发工作。对C语言和嵌入式系统汇编语言都很熟悉，同时还开发过Win32和UNIX下的程序。他的固件工具已成为Embedded Systems Programming和Circuit Cellar Online期刊近期的主题。

单片机与嵌入式系统丛书

- 嵌入式系统设计
- 嵌入式系统固件揭密
- PIC16F87X单片机原理与专题应用
- VHDL数字控制系统设计范例

ISBN 7-5053-8668-9

9 787505 386686 >

CMPBooks

责任编辑：富　军
赵丽松

封面设计：张　昱

ISBN 7-5053-8668-9/TP·5034　定价：39.00元（含光盘）

单片机与嵌入式系统丛书

嵌入式系统固件揭秘

Embedded Systems Firmware Demystified

[美] Ed Sutter 著

张晓林　等译

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

本书通过一个完整的嵌入式系统设计的全过程，向读者展示了嵌入式系统的基本框架及设计、编程、调试等技术细节。书中详细讲述了嵌入式系统中的存储器、微处理器与微控制器、数据总线与地址总线等基本概念，以及在设计中要考虑的要素。另外还给出了关键程序的源代码，使读者通过本书可学会如何看懂嵌入式系统的原理图，了解系统是如何工作的，掌握嵌入式系统开发平台的主要固件。

本书适于从事嵌入式系统研发的技术人员及高校相关专业的师生阅读。

图书在版编目(CIP)数据

嵌入式系统固件揭秘/(美)休特(Sutter,E.)著；张晓林等译. —北京：电子工业出版社，2003.6
(单片机与嵌入式系统丛书)
书名原文：Embedded Systems Firmware Demystified
ISBN 7-5053-8668-9

Ⅰ.嵌… Ⅱ.①休…②张… Ⅲ.微型计算机—系统设计 Ⅳ.TP36

中国版本图书馆 CIP 数据核字(2003)第 029515 号

责任编辑：富 军 赵丽松
印 刷：北京天竺颖华印刷厂
出版发行：电子工业出版社 http://www.phei.com.cn
北京市海淀区万寿路 173 信箱 邮编 100036
经 销：各地新华书店
开 本：787×980 1/16 印张：21.75 字数：447.2 千字 附光盘 1 张
版 次：2003 年 6 月第 1 版 2003 年 6 月第 1 次印刷
印 数：5 000 册 定价：39.00 元(含光盘)

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：(010)68279077

译 者 序

本书是一本述说嵌入式系统固件的畅销书，通过介绍一整套嵌入式系统的设计过程，使读者了解嵌入式系统的设计和开发方法。本书共分 13 章，第 1、2、3 章主要介绍了嵌入式系统硬件和软件的基础知识和微控制器开发平台；第 4、5 章讲述了所需的外围硬件和软件接口；第 6、7 章介绍了闪存文件系统；第 8 章介绍了命令的运行；第 9～13 章介绍了网络的连接和文件传输及实际应用中系统平台的调试过程。

本书以介绍嵌入式系统固件开发为主，列举并讲解了大量的程序实例，对设计中的有关技术问题做了相应说明。本书简明扼要，通俗易懂，可供高等学校有关专业学生及研究生、教师和从事嵌入式系统设计开发工作的技术人员阅读参考。

在本书的翻译、整理和校对过程中，路程、李蕾、谭征、徐彬、魏春风、陈飞虎、林琳、刘佳、张志轩等人做了大量的工作。由于译者水平有限，加之工作繁忙，译文中难免有不妥之处，恳切希望广大读者批评指正。

张晓林　等译

前　言

嵌入式系统是嵌入到其他产品内部的计算机。奇怪的是，虽然人们知道许多关于编程和计算机的知识，但却在神秘的嵌入式系统世界中感到迷惑。在嵌入式系统编程世界中，大量的细节（包括硬件和软件）使开发过程像是在探险，甚至感到“难于”生存下来。软件、硬件、固件到底有何区别？怎样将 10 万条程序嵌入到比指甲还小的器件中？什么是闪存？为什么需要高速缓存（cache）？任务和进程之间有什么区别？要不要考虑可重入性？在阅读这本《嵌入式系统固件揭秘》的过程中，读者将逐步了解这些问题，不再像起初那样感到迷惑。

嵌入式系统编程覆盖了从最具体的底层编程到最抽象的高层 UNIX 编程的各个方面。它引起了近 20 年来工业界爆发的革命性变化。在 20 世纪 70 年代后期，汇编程序已被认为是够丰富多采了。一般的嵌入式系统可以使用小于 64Kb（位，不是字节）的系统内存。没有什么硬件要留给固件开发者去处理。一般由同一个人完成画图、焊接样品、写固件，把所有的事拉拢到一起。当 Intel 公司引入 8085 芯片时，很明显那些复杂的微处理器还停留在这个水平上。在 20 世纪 80 年代，Motorola 与 Intel 展开了 CPU 大战，C 语言成为了少数敢于用高级语言编程并烧制 EPROM 的人普遍采用的编程语言。今天，微处理器随处可见，范围从在工业界占主导地位的 4 位和 8 位微处理器家族到 1GHz 的几乎需要冷却设备（当然也是由微处理器控制的）冷却的 64 位微处理器。

多年来，这些系统的复杂度像滚雪球式的增大。工业界已经从在前台用二进制代码对 DEC PDP 机器编程发展到给烤面包机的微控制器提供面向对象的设计。系统的发展速度非常迅猛，已有的微处理器、微控制器、RAM、DRAM、SDRAM、管道、超标量体系结构、EPROM、闪存、RISC 和 CISC、RAS、CAS 及高速缓存等，也仅仅是一个开始。

现在，从牙刷（不是开玩笑，是事实）到喷气式战斗机都由某种微处理器控制着。这种趋势自然产生许多工具和技术。这些可供选择的硬件（固件必须使用的集成电路）和软件（用于建立固件应用程序的工具）使人眼花缭乱。

本书的目标是通过一个完整的嵌入式系统设计过程将你带入真正的嵌入式系统工程。而且，这个工程的源代码包括一些固件（一个嵌入式启动平台），可以简化你今后的工程。本书使用一个包括 CPU、内存及一些外部设备的小硬件系统设计实例，给出了基本原理结构图并讲解如何将指令从内存中取出，同时也将讲到设备

的概念。本书将讲述闪存与EPROM、SRAM与DRAM、微处理器与微控制器、数据总线与地址总线，还将讲述如何将C语言和汇编语言源代码转换成二进制映像，并装入设备存储器中以便在CPU引导（引导闪存）时使用。

本书用几章的内容讲述启动嵌入式系统并执行一个应用程序（包括在汇编程序中的基本引导程序）的基本概念，先不涉及句柄、闪存驱动程序、闪存文件系统、串行和以太网连接等。这样可以使读者理解嵌入式系统工程如何启动、如何建立起支持嵌入式系统的平台。

这些听起来让人兴奋吗？美妙吗？恐慌吗？事实并非如此。本书的目的并不是要讨论最新出现的超标量体系结构和在印刷电路板上的铜线所产生的天线辐射效果，也不想涉及高级的抽象设计过程（高级的体系结构和传输线的效果固然重要，但这些不是本书的主题）。本书是面向那些不想被一大堆工业行话和特殊技术细节所烦恼的、又想尽快了解系统的读者撰写的。读完本书，读者将学会如何看原理图、了解引导闪存器件是如何工作的、掌握完整的嵌入式系统开发平台的主要组件。

本书的读者对象

本书的读者应有一些C语言编程经历和基本的汇编语言概念。本书并不要求读者有电子技术或硬件知识背景。因此，只要读者有一些编程知识背景将会发现本书大有益处。

计算机科学或电子工程专业方面的学生不需要有固件开发的背景，只要有兴趣，就可以从本书中获益。

初级固件开发者将发现本书中的例子很有帮助，因为本书中的例子包括文档和代码注释，可以将其扩展到固件开发平台。书中将详细讲解引导新硬件和CPU与外部设备的交互方式。涉及到从小文件传输协议（Trivial File Transfer Protocol, TFTP）到以太网的底层引导过程。读者可以将本书中的代码或一部分代码引入到自己的固件平台中。

硬件开发者将发现本书讲述的平台有助于在复杂的CPU上分析并调试硬件，也会出于好奇，将注意力从硬件上转到学习固件过程上。本书提供了不脱离硬件而进入固件学习的起点（硬件设计者会自然地过渡到固件/软件的编程世界中）。

项目主管也会发现本书非常有用，因为这里呈现的固件包是一个成熟的平台。这个平台可广泛地应用到实时操作系统（RTOS）和目标体系结构中，而且很容易放到新系统中。这个平台是面向目标的，且独立于RTOS，这使得它很容易转到各种目标或RTOS系统中。

本书的内容

第 1 章　艰难的开始　本章讲述 CPU 的基本结构及其支持的外设，本书讨论的固件将在这些外设上实现。本章将涉及核心处理器、RAM、闪存、串行 I/O 及 CPU 监控等与设计相关的内容，对每项设计要素都将详细讨论（如 CPU 监控、动态与非动态存储器等）。本章还将讲述微处理器如何从存储器中取得指令及高速缓存如何提高工作效率。本章包含了当今许多微处理器常用的外设，没有特指某个专用外设。

第 2 章　开始动手　本章介绍建立和编写嵌入式设备程序的方法，讲述本机编译与交叉编译环境之间的差异，将讨论文件格式和制作启动闪存的具体步骤，还将解释连接编译文件的重要性，讲述它如何将应用程序的代码部分恰当地装入目标存储器。本章还讨论一些调试方法，为今后进行固件编程打下基础。

第 3 章　微型监控器　本章介绍嵌入式系统中的启动平台和启动监控器。这里的监控器叫微型监控器，它有许多特性，是对学习固件开发过程极有帮助的工具。

第 4 章　所需的汇编语言　本章描述微型监控器的复位操作和启动代码，然后建立一些串行驱动程序并讨论如何建立异常处理程序。

第 5 章　命令行接口　本章描述如何建立在第 4 章中介绍过的核心功能的命令解释程序。本章从命令行接口（CLI）开始，解释如何分析输入的命令行，然后调用相应的函数处理命令行。命令行接口界面包括命令解释变量和符号、命令行编辑和历史记录、用户分级及密码保护。

第 6 章　闪存的接口　本书中共有 3 章讨论有关闪存的内容。本章是其中的第一章，将描述一个带有多个闪存的平台，讨论涉及到闪存编程的底层细节，并提供一个对带有不同闪存配置的目标系统非常有益的方法。

第 7 章　闪存文件系统　本章讨论建立驻留在闪存中的小文件系统（Tiny File System，TFS）的方法。这个简单的文件系统可以说明一些针对多闪存文件系统设计的更复杂的方法。本章还将讨论一些 TFS 设计方法的优缺点。

第 8 章　执行脚本　本章讨论用于命令解释的 CLI 和用于文件存储的 TFS 的特点，使文件像脚本或小程序一样执行。本章还解释将 CLI 和 TFS 相结合以创建一个简单的解释器的令人惊叹的方法细节。本章用 CLI 命令为 CLI/TFS 用户提供了一个简单有效的编程环境。条件分支和子程序只是本系统的一点小功能。

第 9 章　网络连通性　本章讨论接入网络的必备元素。与闪存正逐渐成为标准的引导存储器一样，以太网也正逐步成为许多嵌入式系统的标准接口。本章讨论几

个用于嵌入式系统的协议，包括 Ethernet、ARP、ICMP、IP、UDP 及 BOOTP/DHCP。最后，本章还利用几小段只知道本身网络包有效负载的程序代码来检查网络各层的封装。

第 10 章　文件/数据传输　本章讨论两个常用的文件传输协议：用于串行接口的 Xmodem 和用于网络的 TFTP，将在启动监视器中以如何使用这些协议为例展开讨论。本书中实现的 Xmodem 和 TFTP 可以在源和目标系统之间完成文件和存储器原始数据的传输。

第 11 章　添加应用程序　本章展示平台如何为应用程序提供基础。本章中的例子是一个非 RTOS 应用程序，但本章中的这个平台却支持各种环境。作为一个开发者，要判断在应用程序取代系统后，平台上还有多少功能可供应用程序使用。本章讨论应用程序如何驻留在 TFS 中，以及装入平台的程序如何自动将对文件从闪存空间传输到 RAM 空间来执行。

第 12 章　基于监视器的调试　本章讨论一些监视器的联机程序调整能力和说明实现这些能力的理由。调试程序对任何一个开发人员来说，都是必不可少的。本章将讲述如何建立基本的符号调试器，以便将内存像一个结构（表）一样显示出来（而不是一堆原始数据）、设置断点、单步执行、预定义程序的执行步骤、清空堆栈等。本章讨论的调试技术不需要片内支持和外部调试器。

第13章　将微型监控器接入 ColdFire MCF5272　本章介绍将微型监控器固件包接入 Motorola MCF5272（ColdFire）评估板的过程，详细讲述了本书所附 CD 上的源代码的目录结构，以及在接入的过程中配置监控器的细节。本章完成了以前章节中开始介绍的接入过程。

附录 A　建立基于主机的工具箱　本附录提供一些对与主机文件、串口等连接非常有帮助的代码段。尽管本书是关于固件方面的事实，但也离不开某些最终的本地编程。即使读者可能已经做过许多本地编程，但本附录中提供的代码会加快你的工作速度。

附录 B　RTOS 概述　本附录介绍一些实时操作系统（RTOS）的基本概念，使读者可以从单线程的编程开发环境扩展到多线程环境。在这方面有专门的图书，所以本附录不深入讲解，而是提供一些基础知识，使读者理解多任务能做什么，了解优先控制权为什么是既优秀又带有危险性的特点。

源代码

本书包含大量源代码，在叙述过程中，当谈到各种程序时，使用了其中部分或

全部功能。本书中的全部源代码都已转换成 C 源代码，并放在本书所附的 CD 中。这些源代码中有许多部分不是本书讨论的重点，所以在书中没有全部列出来，以使本书叙述更清晰。CD 中提供的源代码是最终工作采用的版本。如果书中所列的程序不完整，请参考 CD 中的代码。

致谢

首先感谢我在朗讯工作的好友和领导 Roger Levy、 Paul Wilford，感谢他们的支持与鼓励。同时也感谢我的好友 Patricia Dunkin 和小 Agesino Primatic。感谢他们帮我审定稿件并提出各种好建议。

感谢本书的出版公司 CMP，感谢 Joe Casad、Catherine Janzen 和 Robert Ward 所做的编辑和技术支持（特别感谢 Robert 在本书的全部过程解答我众多的问题，并及时纠正我的偏差）。感谢 Michelle O'Neal、Justin Fulmer、James Hoyt 和 Madeleine Reardon Dimond 为本书排版、图表及索引所做的工作。

最后，感谢我的爱妻 Lucy，是她始终不渝的鼓励才使我写成了本书。还要感谢我的爱子 Tommy，他始终将我的写作放在首位。最要感谢的还是我们的救世主——耶稣基督，是他无私的爱给了我永恒的力量。没有任何力量可以与其媲美！

目　录

第 1 章　艰难的开始

尽管本书关注的重点主要是固件开发，但是一个好的固件开发者必须对其所依赖的硬件有很好的了解。人们可以将固件编程者与加油站换油的工作人员相比：如果他所知道的只是换油，那他就无法发现各种事故隐患，而一个训练有素的技术工人会马上发现这些迹象。受训不久的技术工人也许可以胜任换油的工作，但却不会注意诸如盖垫密封片泄漏之类的问题。如果这些问题一直不被发现，则某些部件迟早会出问题，并且可能会由于技术工人的知识有限而产生额外的损失。

本章阐述在一个典型系统里，固件工程师需要了解关于硬件方面的知识。这样做的目的并不是把你变成硬件设计者，而是为了使你能做比“换油”更多的事情。为了达到这个目的，本章讨论一些常见的 CPU 支持的外围设备并讨论处理器如何完成工作。从讨论本书涉及的系统开始，你将了解现在基于微处理器系统的一些最常见的特点，同时还将讨论 CPU 获得指令的步骤及使用缓存所引入的好处和缺陷。

至本章的最后（尽管你还不是一名合格的硬件设计师），你会对固件的硬件平台有更好的理解。

1.1　系统要求

嵌入式系统所需要的硬件因其用途不同而有显著差异。有些微型系统只有不到 1KB 数据空间和 16KB 地址空间，而高性能系统可能运行于 1GHz 64 位处理引擎下，拥有 32MB 快速闪存和 128MB DRAM。本书将关注在这两者之间的系统性能，所讨论的固件的存储空间（32~256KB 的闪存，8~128KB 的 RAM）视所要实现的功能而定，所以这种系统不适合于小的微处理器（8051，68HC05/11/12 等等）。另一方面，这些代码不需要大型机器就能成功运行。

图 1.1 显示了一个完整的系统模型框图。利用这个系统模型，你可以对自己的器件进行编程、与 PC 会话，甚至给网络浏览器提供 HTML 页面服务。如果想节约，则可以去掉复位/看门狗（Watchdog）及有备用电源的 RAM（BBRAM）/每日时间控制器（TOD）。在模型中包括的这些功能是因为它们极其有用，同时它们在大多数的微处理器设计中也成为标准的配备。

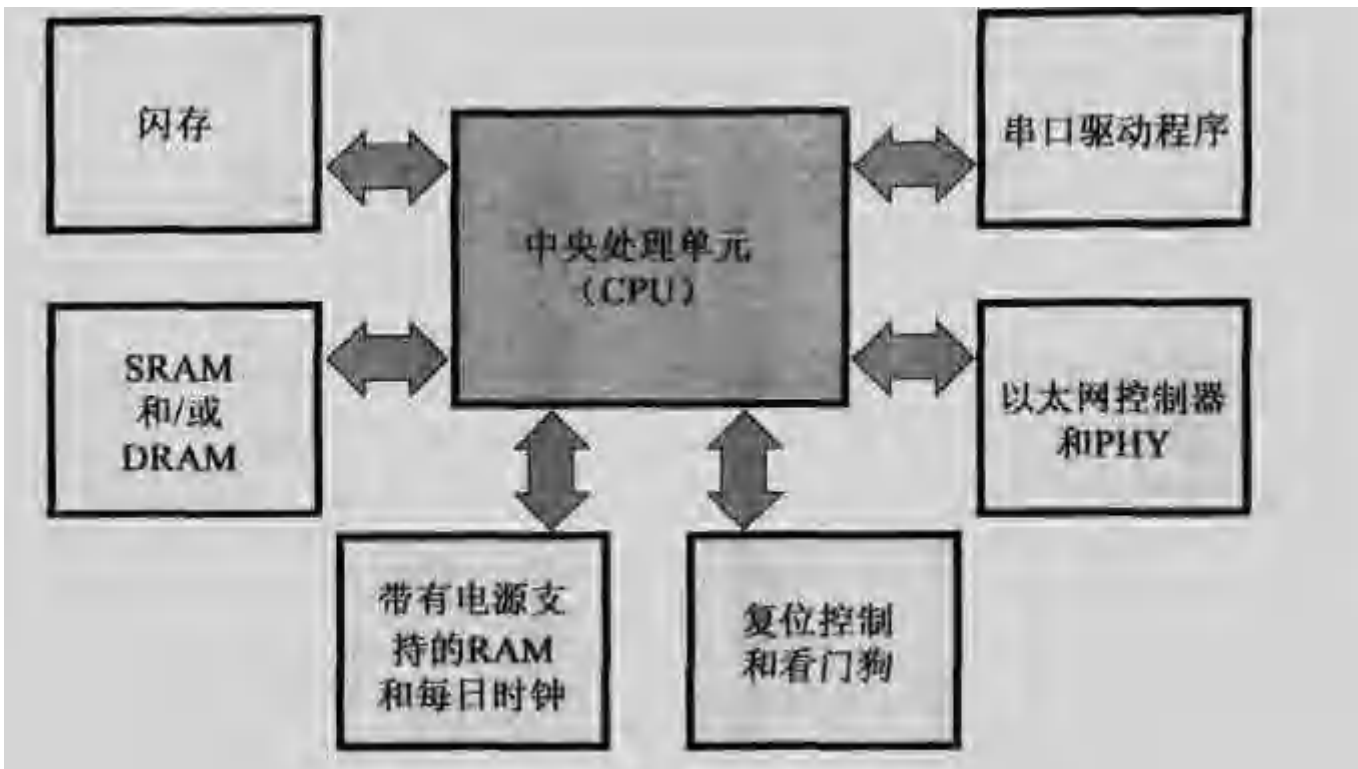


图 1.1 典型的嵌入式系统框图

这些是嵌入式系统中最常见的组件。所有的嵌入式系统都使用某种非易失性的存储器（闪存，EPROM，ROM）和 RAM。大多数系统都能与其开发环境通信（串口，以太网端口，JTAG 端口）。

1.2 中央处理器

嵌入式系统是围绕微处理器或者微控制器来设计的。在我们讨论的范围内，这两者的区别并不大，部分原因是现代技术的发展使两者间界限变得模糊了。几年前，微处理器被定义为执行单元，而其他则都视为外围设备。微控制器则将其所需要的所有东西均集成在一个芯片上，以此同微处理器区别开来。今天，这种区别已经不是很明显了。我们曾经称之为微处理器的芯片越来越多地与附加的外设集成在一起。这些芯片包括串口、可编程芯片选择、中断控制、DRAM 控制器、内部定时器以及可编程 I/O 管脚，有些芯片甚至内建有以太网设备和 PCI 控制器。有些芯片被生产厂商称之为微控制器，而其他人则称之为微处理器（就像某些人说 potaeto，而另些人说 potahto，其实都是指法式油炸食物！），不要让名字把你搞糊涂了。本书将采用下列定义。

- 微控制器——通过设定，能百分之百运行的器件。它不需要任何其他的外围设备（除了电源和晶振）。微控制器包含少量存储程序的非易失性存储器和易失性的用于编程的存储器，并通常带有 8 位或 16 位的 CPU。属于这个范围的器件有 8051 和 68HCXX 系列。

- 台式电脑微处理器　　除了处理引擎其他几乎什么都没有，需要复杂的辅助芯片作为外设，低端采用如 32 位芯片，但是 64 位和 128 位芯片正逐渐占据主流。这个范畴内的器件有 K6、奔腾及 G4 处理器。
- 嵌入式微处理器——包括一个处理单元和一些常见外设的处理器，通常属于 16 位或 32 位的范围，包括诸如 68332、PPC 860 或 SH 2 的处理器。
- CPU（中央处理器）——可以指以上的任何一条或者仅仅指微控制器或者嵌入式微处理器的处理单元。

请记住当然这些并不是工业的标准规定，同时也请注意不同的术语存在某些重叠。微控制器和台式电脑微处理器都可被应用于嵌入式系统。实际上，在 20 世纪 80 年代微控制器甚至也可以应用于最早的桌上型电脑。

正如上文所定义的那样，微控制器以几个数量级的优势占据了嵌入式系统的市场。然而，芯片（处理器和存储器）的价格在下降，同时 16 位和 32 位处理器在嵌入式系统中也得到了应用。本书将着眼基于嵌入式微处理器的系统。

嵌入式微处理器有不同的形状和尺寸，甚至可被作为逻辑内核集成到大型可编程器件里（像 FPGA）。曾经生产可编程逻辑器件的厂家如今已经在生产内建处理器的逻辑器件。

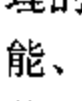
下面对目前一些嵌入式微处理器所包含的典型外设进行简要描述，每一项都用办公室环境中类似的过程来比喻。

1.2.1　可编程芯片的选择

就像一个委员会的主席，可编程芯片的选择需通过协调通信来减少混淆。设想你正参加一个会议，且你试图对小组的某人说话。在这种情况下，你需要以某种方式使那人知道你想对他说话。更理想化一点，这种方式同时也应该让其他人知道你不是和他们说话。芯片的选择可使微处理器执行这项任务。处理器的外设占据处理器一定的地址范围，当外围器件的使能输入端没被激活，则其数据总线处于高阻或者断开状态。这个思想很重要，因为在任何时候只能有一个外设对数据总线写数据（总线冲突意指在某给定的时间内有多于一个的外设试图对数据总线执行写操作）。

固件必须知道每个外围设备的地址范围，这大多在硬件设计阶段就决定了。每个外设都有一些设置特性，而处理器必须处理这些特性才能在两者之间进行交互。有些外设占用少量的地址空间，其他的则占用几兆空间。有的工作速度足够快且能够跟上处理器的速度，有的则迫使处理器等待。在某些情况下，外设完成其工作后异步地通知处理器，而在其他情况下则只有经过预定的时钟周期才能再次获得外设

的控制权。所有这些细节均需要在微处理器和外设之间增加大量的额外连接电路（Glue logic）。为了减少这种连接电路，大多数面向嵌入式系统的处理器提供可编程芯片选择的管脚。片选是 CPU 使用的一个管脚，用来获得处理器总线上特定地址范围。通常在微处理器上都有 3~6 条这样的管脚。与这些管脚相连的硬件可以提供简单的“一器件每芯片（one-device-per-chip）”的选择。另外，处理器与外设之间也可通过附加的逻辑使多个器件连接到单一的片选上。



注意

连接电路（Glue Logic）意指连接两个器件所需的额外电路（逻辑）。

1.2.2 中断控制器

中断控制器用来帮助对输入消息进行优先级排列。这一点任何一个办公室人员都应该能够理解。设想如下的办公小场景：A 正在办公室桌子后面接电话，B 穿过大厅走进 A 的办公室问问题。A 可以做如下一些事情：

- 完全忽略 B 的问题，继续接电话。
- 迅速对 B 说“好的，我等会儿和你谈这个问题。”这样把 B 的请求和其他的一堆事情放进一个队列。
- 告诉电话那边的人一会儿再谈，然后对 B 做出反应，这就像 A 认为 B 比电话里的那人更重要一样。
- 结束对话。在 B 是上司的时候，这种情况自然会出现，不管电话里的人是谁。

你可以说 A 处理了来自于 B 的中断。许多因素决定了 B 的问题是否被处理，包括电话里的人的重要程度等等。然而，A 必须根据正在进行的事情区分中断的优先级次序。

现在，让我们假想 A 接电话就如同 CPU 执行指令，而 B 就如同需要 CPU 处理的外围设备。一个集成的中断控制器使得 CPU 对来自内部和外部的中断进行使能、屏蔽及优先级排列等控制。通常所有的中断都能够被屏蔽掉，除了 NMI（非屏蔽中断）和复位信号（就像上文场景中的上司）。

1.2.3 定时-计数单元

设想这样一个场景：几个打字员在同一个办公室工作，仅仅是因为好玩，决定

看谁在 1min 内打字多。他们集中在电脑前，每人进行 1min 的试验。这时，他们需要某种方式来记录从开始到结束的 60s 的时间间隔。另一种可供选择的方式是他们只记录每人打一定文本所需的时间。在这种情况下，他们并不记录一个固定的时间，而是记录每个人完成固定量文本所花的时间。

嵌入式系统中的定时-计数单元能帮助我们完成这项工作，给 CPU 提供了计算完成某事所花时间的能力，也使得固件能够产生周期事件，包括根据输入脉冲进行计数的事件。

注意

微处理器中的定时-计数单元本身通常不处理每天的真实时间，所以不要假定“定时器”等同于“挂钟”。在大多数情况下，“定时器”相当于“秒表”。如果在某个开始时间秒表和挂钟同步，则秒表可等同于挂钟。处理器的定时器也就是这样获得真实时间的。

1.2.4　DMA 控制器

设想办公室有一个文件柜，柜里的文件可供好几个办公室的员工使用。其中有些员工只是偶尔使用一次，而其他一些员工则用得频繁得多。管理人员于是制定这样的规定：有些员工用自己的钥匙来打开文件柜，其他员工则必须从管理人员那里取得公用的钥匙。换句话说就是，有些员工可以直接访问文件柜，而其他员工则需通过间接的方式访问。

在许多嵌入式系统设计中，CPU 是惟一与存储器相连的器件，这意味着每一个操作都必须通过 CPU 来获得保存在存储器中的数据（如果这个操作涉及存储器的话），就像有些员工通过管理人员得到钥匙一样。直接存储访问（DMA）是一种允许外设在没有 CPU 干预的情况下直接访问内存的方式，这些外设相当于那些有自己钥匙的员工。

例如，如果没有 DMA，则串口的一个字符输入操作将对 CPU 产生中断，此时固件将转到这个中断的处理过程，并外设获得字符，放进一块存储区域。但如果通过 DMA，则串口可把输入的字符直接放进内存。当达到某个可预设的阈值时，DMA 控制器（而不是串口）给 CPU 中断信号，并迫使其处理存储器中的数据。DMA 过程更有效，许多集成的微处理器有多条 DMA 通道，使之能够进行 I/O 到存储器、存储器到 I/O 及存储器到存储器的数据传送。

1.2.5 串口

设想人们可以自由出入事务所的前门，并很方便地接触到事务所提供的设施。这并不意味着没有其他的途径来接触这些设施，而是因为前门是方便而标准的实现方法。

串口的任务就是提供这样的访问机会。串口给控制台或其他利用同样协议的器件提供基本的通信支持。串口或者通用异步接收发射机（UART）给 CPU 提供 RS-232 位数据流，这也是 PC 的 COM 端口采用的技术。在这个标准上，不同处理器有不同的变化，但是几乎在所有的情况下，数据均是基本的异步串口比特流（利用 RS-232 标准）。它由一位标志开始，且由一些数据位（通常在 5~9 位之间）中的 1~2 位来表征结束。

1.2.6 DRAM 控制单元

想像你是个经理，需管理很多有天赋的员工，同时他们不得不花时间照顾他们的孩子。则你可以告诉他们，你没法在办公室里应付这些孩子，或者你可以灵活点，让办公室的其他一些人专门照顾孩子。

这就是动态 RAM（DRAM）。RAM 在所有基于微处理器的项目中都是很有用的。DRAM 比 SRAM 便宜得多。但是，不像 SRAM 和闪存，DRAM 需要“临时保姆”或者说额外的逻辑电路来使比特稳定，DRAM 的类型和大小决定“临时保姆”的复杂性。DRAM 控制单元完成“临时保姆”的工作，这样 CPU 就能够直接同动态 RAM 交互。

1.2.7 内存管理单元（MMU）

内存管理单元的主要职责是指定并实现一种界限，可分隔不同的任务和过程。为了理解对任务的分隔，设想一下两种不同的办公环境。第一种环境（正式的环境）是指每个雇员都有各自的办公室。他有自己的钥匙并被认为是这个房间的所有者。第二种环境（不那么正式的环境）是指大房间的隔间设置，在雇员之间没有墙，每个人在同一层同一空间里。这两种配置都有其优点和缺点。第一种不必担心一个雇员打扰另一个雇员，因为他们之间有墙。不管某个雇员如何喧闹，隔壁的员工不会被扰乱。对于不那么正式的设置情况就不同了。如果某个雇员在公共场所大声打电话，这个不顾忌他人的雇员会使得其他的雇员分神，但其优点是如果雇员们都很体

谅别人，两个员工可以不必离开其座位即可快速地交流。同样，如果某员工需要借别人的订书机，这仅仅是一步的事情。而在正式的环境中，如果需要交流时，就需打电话或去其他办公室，员工们必须走过一系列的楼梯。

每个雇员可以被比作一段代码块（或者任务），并可在嵌入式系统程序中实现特定的功能。办公室空间相当于目标系统的存储空间，行为无礼的员工所产生的噪音相当于程序的 bug，它破坏了其他人的存储空间。如果硬件中的 MMU 配置得当，则相当于正式办公室环境中的墙。每个任务的存储空间被 MMU 所加的限制来约束。这意味着，错误的指针不会破坏其他任务的空间，并且过程与过程间的会话将需要更多的管理花费。

在本书的稍后部分你将可看到有 MMU 保护的代码块被称为过程（Process），而没有被保护的则被称为任务（Task）。几年前，在一个嵌入式系统里对 MMU 的最大限度利用是很少见的。即使在今天，大多数嵌入式系统也没有完全利用上 MMU 的管理能力。但随着嵌入式系统变得越来越复杂，CPU 的速度也持续上升，因此利用 MMU 在过程间产生“墙”也正变得越来越常见。

1.2.8　缓存

设想你需要经常查询办公桌旁文件柜里的文件夹，但其中的有些文件是你经常查询的，而另一些则很少查询。你可以在每次查完文件后随手放回橱柜，但也可以使用一些小的架子来放那些经常查询的文件夹，这样你就不必在每次查询时都起身了，且架子上的文件夹会根据当天你决定查询多少文件而不同。如果组织得当，则身边的架子会节省你往返于办公桌和文件柜的时间。

微处理器和存储器的情形与此类似。微处理器的速度很快而系统存储器速度相对较慢。为了弥补这种差异，小块快速（也更昂贵）的存储器被设在微处理器和系统存储器之间，这样频繁读取内存的操作可以通过这种更快的缓冲存储器来完成，以代替比较慢的标准存储器。

1.2.9　可编程 I/O 管脚

作为办公室的经理，你知道理想的雇佣方案是雇佣拥有多种技能的员工。尽管每个雇员在任何时候都只能做一件事情，雇佣拥有多种技能的员工使你可以选择用不同的方法来使用同一组员工。

大多数现代的微处理器也给予你这种选择，所有前面提到的外设都需要利用微处理器上特定的管脚。在很多情况下，大部分管脚可用来实现他们特定的功能（串

口接收器、定时器输出、DMA 控制信号等等），或者可以编程设定为简单的输入、输出管脚 （PIO）。这种灵活性允许芯片根据设计的需要对其进行不同的配置。比如，如果你不需要两个串口（微处理器有两个），那么第二个管脚能作为简单的 PIO 管脚，用来驱动 LED 或者作为读取切换开关。

注意

可编程管脚有时意指双重功能管脚。注意这种双重功能并不是假定的。每个管脚如何配置以及配置管脚的功能（使之在不同模式下运行）均依赖于微处理器的实现。通常，管脚名字的选取要反映这种双重功能。比如，如果 RX2 能设成串口 2 的接收端或者 PIO 管脚，那么它很可能被标记为 RX2/PION（或者类似的形式），这里的 *N* 是 1~*M* 之间的数字，而 *M* 则是微处理器的管脚数目。你应该意识到某些微处理器做广告时说有某些特点，但实际上是通过双重功能的管脚来提供的。因此，宣传的全套特性（串口 2 和 32PIO）也许不能同时获得（因为第二个串口的管脚同时也是 PIO 管脚）。所以请仔细阅读数据表。

1.2.10 把所有部件集成起来

所有这些办公室类比中最重要的是不同的办公室可以根据不同的业务需要进行各种配置。这种情况也适用于微处理器。

到目前为止，我们的讨论是假定这些组件与 CPU 都存在于芯片上，但是也可能作为物理上分开的器件出现。下一个部分的讨论通常是在芯片之外的组件里。

1.3 系统存储器

除了 CPU 本身，在任何基于微处理器的系统中存储器是最基本的组件。CPU 从内存读取指令，这些指令告诉 CPU 做什么。如果存储器编程不正确或者错误的连接到 CPU，那么即使最复杂的处理器也将变得混乱。

有几种不同的存储器可供使用，不同的设计有不同的需要，可使不同结构的存储器受欢迎。比如，有些系统需要大量的但并不需要很快速度的存储器；有些需要少量的快速存储器；有些需要在断电时不丢失数据的存储器等。下面将对目前最常用的存储器及其特性进行讨论。

1.3.1　ROM，PROM，EPROM 和 EEPROM

ROM 是 Read-Only-Memory 的字组的缩写。这种器件的编程是作为制造过程的一部分，他不能被 CPU 或者程序员改写。可编程只读存储器（PROM）可以被程序员编程，但是不能被擦写。PROM 给予程序员在对器件编程时一次写的机会。如果程序中有错误或者程序需要更新程序，那么旧的 PROM 必须扔掉，并用一块新的正确编程的器件来代替。可擦写只读存储器（EPROM）可以被编程和擦写。限制就是必须使用外部器件完成编程，且必须通过紫外光源来擦写。电可擦写可编程只读存储器（EEPROM）不需要外部器件就可完成编程和擦除的操作。粗看上去，EEPROM 将是理想的存储器件，但其速度比较慢，价格又贵。随着新技术（比如 FLASH）的逐渐流行及价格的下降，ROM、PROM 甚至 EPROM 将会渐渐消失。

1.3.2　RAM

RAM 是 Random Access Memory 的字组的缩写。不像 EPROM 和术语 RAM，不能很好地反映它的特性（至少从我们的观点来看）。它的名字表明了任何字节均可以在任何时候获得。在它第一次出现在电子学中时，其相对于以前的存储器（Sequential Access Memory）已经前进了一大步了。由于我们讨论的原因，我们假定所有的存储器都是随机存储的，但并不是所有的存储器都能够被 CPU 写。因此对我们来说，区别点（与 EPROM 相比）是处理器可以对 RAM 进行写操作。RAM 是可读可写的，也是易失性的，这意味着当断电时，数据不会被保留。

有两种基本的 RAM，即静态（SRAM）和动态（DRAM）。SRAM 是比较容易处理的，因为它是静态的，不需要处理器的“照顾”就能完成工作，直接接到处理器上就可以用了。DRAM 需要外部电路周期的刷新，这样其内部电容才能保存电量。DRAM 技术便宜得多，但它也慢得多并需要额外的硬件来使其运行（早先提到的 DRAM 控制器），因此 DRAM 通常用于需要大容量存储空间的系统（>1MB），这样就可与控制器相匹配了。SRAM 比较简单，但是每个字节存储空间的成本也比较高，通常用于仅需较少存储空间或快速可写存储空间的系统（像缓冲存储器）。例如，在典型的 PC 中，有一些 DRAM 用做一般用途，而少量的 SRAM 则作为 CPU 的缓存。

1.3.3　闪存（Flash Memory）

就像 EPROM 一样，闪存也是非易失性的存储器，相对于 EPROM 来说，其最

大的优点是系统内可编程，就是说不需要另外的器件来修改内容。早先的器件需要高电压（通常是 12V），但是现在需要的电压和主板上的其他器件一样。由于闪存在系统（IN-SYSTEM）可写的，所以不需要高电压擦写器。

闪存的结构随着发展而有些变动，同时尽管现代的快速闪存是系统内可编程的，但仍然没有 RAM 使用起来方便。典型的一次擦除过程只能处理存储器的一个簇（SECTOR）。簇相对来说通常是很大的，一次擦除将使那个簇中所有的位设为 1（所有字节＝0xff）。写一个字节只能改变某些位使其为 0。每项操作（擦、写等等）除了读，都必须通过特定的程序算法来实现。这种算法是独特的，以保证不会干扰 CPU 和存储器之间的正常交互。

快速闪存迅速成为现代设计中标准非易失性存储器的首选。除了读写存储器所需的算法外，快速闪存仅有的另一个不足是一个簇可被擦写的次数有个上限。虽上限数值通常是很高的（100 000 或者 1 000 000 次），但在设计时是必须要被考虑到的。

1.3.4 其他

还有几种其他的存储器，他们中的大多数都是以上一种或者几种的衍生物。虽然他们没这么流行，但是他们通常需要满足一定的市场小环境。例如，PSRAM（假静态 RAM）是一种内建刷新控制器的 DRAM。其满足这样一种系统的需求：需要比简单 SRAM 器件更多的 SRAM 支持，但又不需要 DRAM 的大容量。非易失性的 SRAM（NVRAM）是带有电源支持的 RAM，其中有些器件实际上是内建有备份电源，而另一些具有的非易失性仅仅是因为硬件设计带有电源保护器件，还有一些在切断电源时能够自动地将 RAM 中的内容备份到闪存。串行 EEPROM 是一种用 2~4 个 I/O 管脚与 CPU 通信的 EEPROM，访问速度较慢，但物理尺寸极小，没有地址或数据总线。

1.4 CPU 监控

这部分将讨论在 CPU 经受灾难性情形时帮助 CPU 维持自身正常工作的功能。这里所涉及到的功能包括（以重要性排序）复位脉冲发生器、看门狗定时器、SRAM 非易失性的电源监控以及每日时钟。后两者实际上不被认为是 CPU 监控的一部分，但是在所谓的“CPU 监控器”的集成电路中，功能的各种组合是很常见的。

1.4.1　复位

在 CPU 能做任何事之前，它需要接通电源，这仅仅是需要接上器件上的电源和地即可。一旦接上电源，器件将从一个合理的状态开始，这一点特别重要。复位信号迫使 CPU 各主要组件返回到一个预知的初始状态，对处理器复位输入端的典型要求是能够在一段时间内（比如说 100ms）保持在某种常量的状态（通常是低电位）。在一些设计中，当电源给系统供电时，一个简单的阻/容电路在低电位状态被用来保持“复位”段 100ms（这被称为 power-on reset）。

为了不至于陷入太多的细节，我们假设如图 1.2 所示的阻容（RC）电路提供时延充电。当其他的管脚立即获得供电电压（信号 A）时，与“复位”管脚相连的 RC 电路在低电位的较长一段时延（信号 B）保持信号。这样当开启系统后，它提供了至少 100ms 的低电位状态给“复位”，但 RC 电路却并不擅长侦测何时给复位管脚提供低电位信号。这就是说它在远程系统中不能很好地工作，当电源损耗过多时需要自动重启。在这些情形中，电源损耗可能会引起电位的下降，迫使 CPU 产生混乱，而此时 RC 电路并不能将复位端电压拉下来使 CPU 脱离混乱状态。从某些方面来说，RC 电路是数字问题的模拟解答。安全开机重启的解决办法应该是当电源电压低于 CPU 电压阈值时，复位 CPU。幸运的是有现成的器件可完成这个工作。有几种不同的器件，他们可监视电源供给并在电源电压降低到某个值时自动向 CPU 产生复位脉冲。

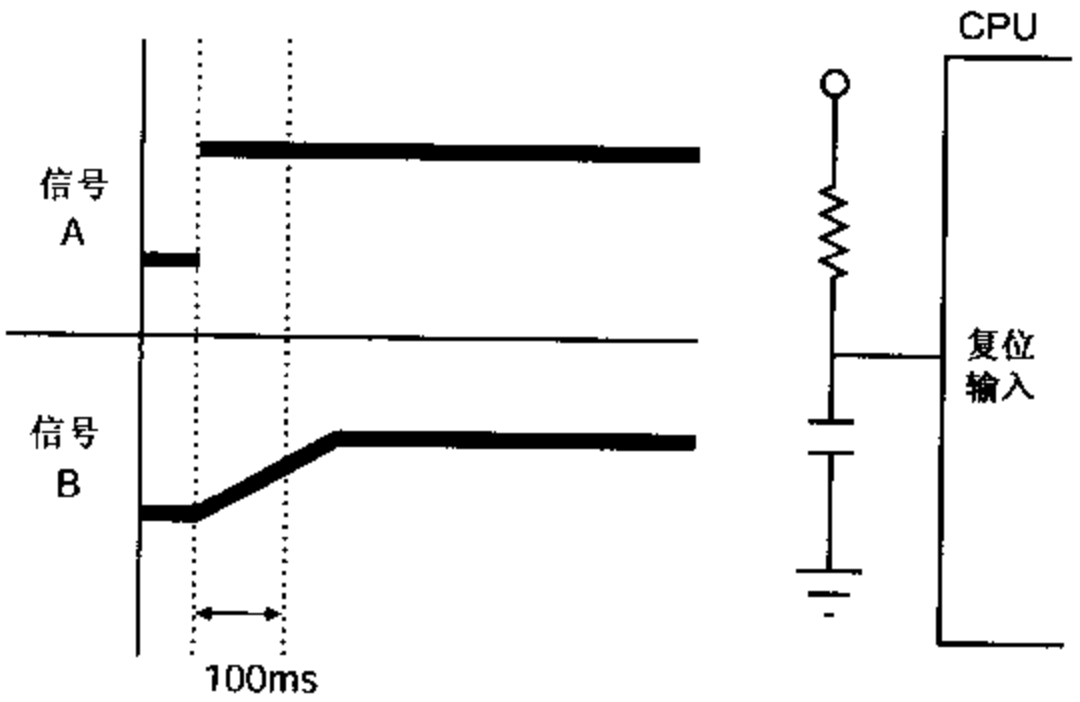


图 1.2　有条件的复位输入

当电源对系统供电时（信号 A），与复位输入（信号 B）相连的 RC 电路的充电时间延缓了复位的激活。虽然这种复位机制在电源被周期性复位时有效，但在电源瞬间断掉时会产生问题。

1.4.2 看门狗定时器（watchdog timer）

看门狗定时器（WDT）就像系统的安全网络。如果软件停止响应或者停止进入当前任务，则看门狗定时器会自动重启软件。系统可能会因为任何难以侦测的硬件或固件上的缺陷而停止响应。例如，如果某个不寻常条件引起缓存的溢出而破坏了堆栈结构，则某个函数的返回地址就可能被改写。当某个函数完成时，它将返回到一个错误的地址而使系统陷入混乱。错误的指针（固件）或者数据总线的小故障也会引起相似的冲突，不同的外部因素会引起“小故障”。例如，即使靠近器件的很少量的静电放电也会产生足够的干扰，使得地址或者数据总线的某一位发生瞬时改变，这种缺陷可能会经常出现，从而导致在项目测试阶段很容易丢失数据。

看门狗定时器是很好的保护者。它惟一的目的是用“你抓伤我的背，我就抓伤你的背”的原则监视 CPU。典型的看门狗定时器（如图 1.3 所示）具有必须周期性触发的管脚（例如每秒一次）。如果这段时间内看门狗没有被触发，则会在一个输出管脚产生脉冲。通常，这个输出管脚与 CPU 的复位端或某个非屏蔽中断相连，而输入管脚则与 CPU 的 I/O 端相连。因此，如果处理器不能使看门狗定时器的输入端以专门的频率触发，则看门狗定时器将假定处理器停止工作，并使 CPU 复位。

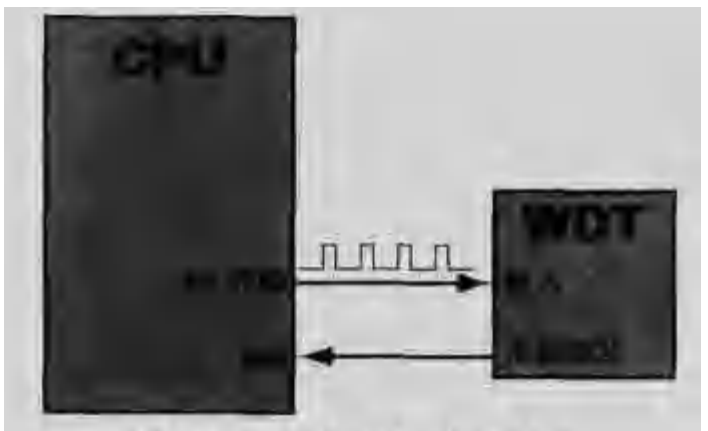


图 1.3 外部看门狗定时器

看门狗定时器是简单的可重触发定时器。当应用程序正常工作时，可通过触发它的输入来周期性地复位 WDT。如果某些信息引起程序挂起或看崩溃，则 WDT 迫使 CPU 复位。

1.4.3 带备份电源的 SRAM

当电源关掉时，不是所有的系统都需要保持 SRAM 的内容，但是因为提供这种功能的器件正变得便宜而易用，所以这种要求也就越来越普遍。现在，一些公司提供的非易失性 SRAM 模块或带有电池或内建电源监控电路，可保证能保留数据 10 年（当然对实际的时间以及内部供给 SRAM 的电源有限制）。有的带有内建电池

的模块与标准 SRAM 芯片的管脚兼容，在你需要向现有设计增加非易失性读写存储时，这种管脚兼容的模块将是你的首选。

1.4.4　每日时钟

对于大多数嵌入式系统来说，CPU 均提供对时间的支持。但是，如果没有使用备份电源，那你将无法得到真实的时间。CPU 对时间的概念仅在 CPU 有电源并运行时才存在。当系统被复位时，CPU 的时钟也被复位，使得 CPU 无法维持自己的时间。如果在你的系统中需要真实的时间，那你将需准备好电池和支持 time-of-day 的芯片。有种情况例外：嵌入式系统知道它在复位后，可以从某外设获得当前时间。

1.5　串口驱动器

许多的嵌入式系统采用串口作为对外的接口，不管串口历史多么悠久（它甚至比尘土还要老），但在现代多数单板的计算机中仍然留有串口。嵌入式系统中的串行器件由两个部分组成，即协议和物理接口。

协议部分规定起始终止位和每字符的比特数，根据波特率确定每个比特的带宽，将串行比特流转化为易于被 CPU 识别的并行字节流。物理接口将 CPU 的电压转化为接口所需的电压。在很多情况下，物理接口也提供物理连接与 CPU 之间的电子隔离。嵌入式系统可对串口采用两种不同的基本传输机制，即直接电缆传输和差分驱动传输。

1.5.1　直接电缆数据传输

直接电缆数据传输是在数据传输的两个方向上各有一根共地的导线连接。尽管这种方式比差分驱动传输要简单，但是它受传输速度及发送端到接收端电缆长度的限制。最常见的标准是 RS-232。RS-232 是 PC 的 COM 及 UNIX 系统 TTY 的串行接口，是目前为止工业上最常用的串行通信标准。

RS-232 的一些特性

RS-232 是作为数据终端设备（Data Terminal Equipment, DTE）或者数据传输设备（Data Communications Equipment, DCE）连接而产生的。如果你想把两个 RS-232 端口对连并在端口间交换数据，那么其中一个必须被设为 DTE，而

另一个必须被设为 DCE。这个术语源自于此接口的最早用途：它把一个哑（dumb）的终端接口与 Modem 相连来提供对远程计算机的访问。在这里，终端就是 DTE，而 Modem 就是 DCE。

如果接口没有 DTE 和 DCE，而你却试图将两设备对连（通过电缆），那么发送端将与发送端相连，接收端将与接收端相连。这样做并不好!把一端设为 DTE，而另一端设为 DCE，这样允许通过电缆将两终端直接连起来。例如，对 DTE 来说，管脚 2 可能是发送端，而对 DCE 来说，管脚 2 就应该是接收端。同样，DTE 的端口 3 可能是接收端，而 DCE 的管脚 3 则是发送端。

如果你有两台设备，而它们都有 RS-232 接口，且是相同的类型，那你怎么办？这就是 Null-modem 的用途所在。Null-modem 通过特殊的电缆或者接在电缆和某设备接口间的小适配器来完成所有的数据交换。

1.5.2 差分驱动传输

差分驱动传输不像直接电缆传输那样应用广泛。它需要更多的导线，使得传输距离大大加长，同时支持超过 1MB 的传输速度。差分驱动传输接口因其固有的抗噪声能力，所以能将更快的信号传到更远的距离。差分传输原理是将极性相反的一对信号（一个信号和它的反相信号）送到两根绞在一起且路径相同的导线上。两根导线相互包裹可减小干扰。接收端根据两根导线电压的差值来决定信号的状态。如果干扰信号影响了电缆，那它的噪声信号将会加入到两根导线上，而这并不会影响检测到的差值（如图 1.4 所示）。

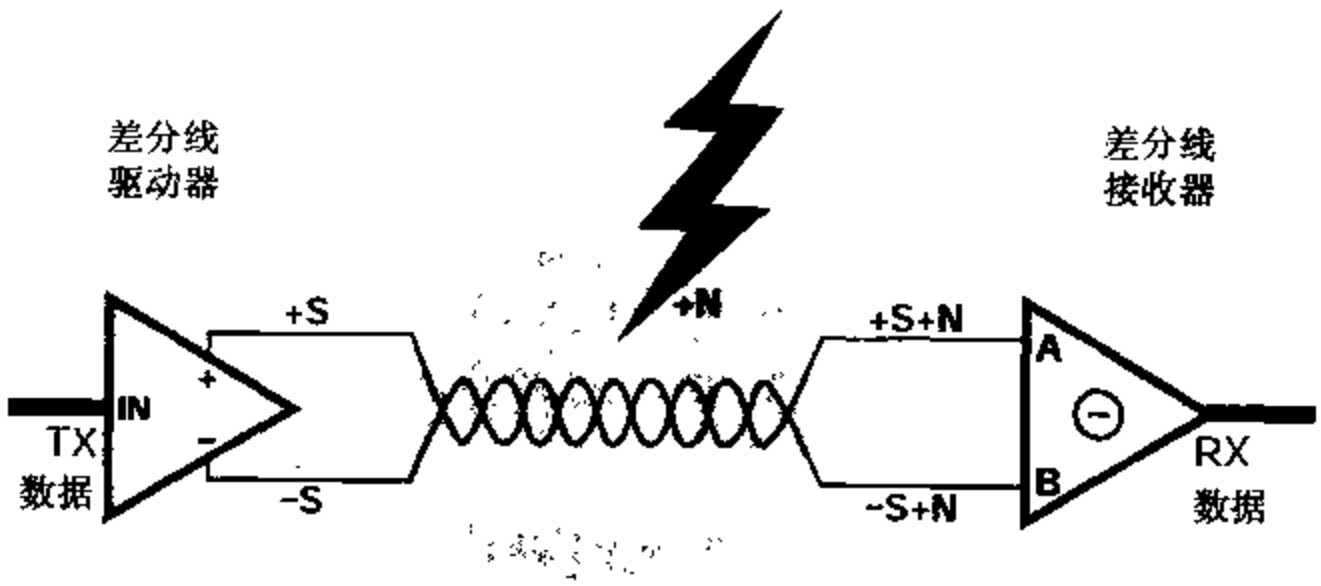


图 1.4 差分传输和抗噪声

（+s+n）－（−s+n）=2s，线路接收器利用类似的技术来计算 A−B。这样线路传输引入的噪声在接收端都被抵消。

常见的利用此技术的嵌入式接口有 RS-422 和 RS-485。RS-422 是 RS-232 的差分传输替代品。在固件没有其他改变时，RS-232 接口可以被 RS-422 代替，同时导线的最大传输量和速度将增加。RS-485 在连接时可有超过一个的发送或接收端，因而增强了功能，且可以支持多设备通信间的主（Single-master）/多从（Multi-slave）或者多主（Multi-master）模式。RS-485 因其良好的抗噪声性能和对多设备连接的支持而广泛用于工厂场地的网络。

1.6　以太网接口

尽管有些处理器芯片包含了部分的以太网接口，但更常见的是接口可作为单独的设备存在，就像串口一样，以太网接口也可分为两层（协议层和物理层）。协议层是作为一个叫媒体访问层（Media Access Layer，MAC）的单一块实现的。物理层由两部分组成，即 PHY 和传输器。更常见的是把 PHY 和以太控制器集成到一个设备里，但其传输器仍然是分开的。因此，以太网接口可能由两个或三个独立的设备组成。

以太控制器是接口中完成信息打包的部分。对于输入的数据包，控制器验证输入的帧是否具有有效的循环冗余校验（CRC），忽略与特定的 MAC 地址不相匹配的包，并将帧组成一个可被 CPU 接收的包（通常通过 FIFO 或 DMA 传输），最后根据驱动器设置的不同参数来产生中断。对于发出的数据包，以太控制器计算 CRC，把数据从内存传到 PHY，给小的包填充完数据，然后通过中断通知 CPU 数据包已被发出。

PHY 是主管接口协议的最低层，负责特定环境的不同参数（如比特率）。

传输器给通过电缆的信号提供隔离和电子的转化。

1.7　闪存设备的选择

所有闪存设备都由大量的扇区组成。在某些设备中，扇区大小相同，其他大小则多样化；有些允许固件锁定扇区，这样设备中的内容就不会被擦除掉；有些有复位输入端，其他的则没有。单个闪存设备的容量从 64KB~8MB 不等。

1.7.1　闪存锁定工具

擦除或写快速闪存包括特殊而重要的算法。假定这个算法不会被意外的执行，

则相对是较安全的，但是也可以选择保护特定扇区不受错误代码影响的算法，有很多闪存都允许保护特定的一个或者一组扇区不能被写入。有些设备通过将其置于外部编程器并对某管脚写入高电压来实现这样的保护。而另一些则有更灵活的设置，它们通过执行特定的命令序列来实现扇区的写保护或者锁定。一旦被锁定，扇区只有在被解锁后才能被修改。这个过程使得扇区被意外破坏的可能性减小了很多。锁定用来保证 CPU 总是可以得到一些基本的引导启动的代码，而不管其他扇区的程序怎样。

如果这种方法还不够，那还有一种可供选择的技术，即重新启动，扇区不能被改写。典型的实现方法是使写保护的管脚有效，然后把锁定命令序列初始化在特定的扇区中。因为写操作的保护管脚有效，并且除非重启它才不能改变状态，这时是没有解锁的命令序列的，所以除了那些一开始引导就发生的写入，才使扇区不会被错误的写入。如果这种方法提供的保护还不够，那就用 EPROM 吧！

1.7.2 底部引导(Bottom-Boot)和顶部引导(Top-Boot)闪存设备

有些设备是这样组织的：一些小的扇区在底层地址空间，接着一些大的扇区填满剩下的空间。有的设备在顶部地址空间是小的扇区，而大的扇区在底层地址空间。因为引导代码通常放在小的扇区，闪存有时被描述为顶部或底部引导，这决定于小扇区的位置。

基本上，有一个扇区包含 CPU 在重启时需要访问的存储空间。这个扇区通常被称为“引导区”，因为某些 CPU 的复位向量表在存储空间顶部还有一些在存储空间底部，所以闪存设备有底部引导的和顶部引导的两种。引导在内存顶部的处理器可能会利用顶部引导设备，而引导在内存底部的处理器则更适合底部引导的设备。

当闪存设备的引导扇区存在于 CPU 的启动地址空间时，这些扇区可以通过使之不可被修改而得到保护。因为通常只有很少量的代码需要基本的引导启动，所以如果你把一些扇区设为不可被修改时并不会浪费多少空间。

1.8 CPU/存储器接口

任何计算机系统中最关键的接口是 CPU 和存储器之间的接口。如果这个接口不能正常工作，那么由于 CPU 不能获得指令从而也不能正常工作。如果处理器不能可靠地获得指令，那么系统中其他的组件即使能工作，那也无济于事——你根本用不到它们。

理解 CPU/存储器的接口比数据和指令流更重要。在大多数系统中，外围设备与内存共享数据和地址总线。因此，理解这些总线的协议对于更好地理解硬件十分重要。

下面将概括地解释CPU如何利用地址和数据总线与系统的其他部分进行通信。为了使我们的讨论更加具体，将按照如图 1.5 和图 1.6 所示假定的机器来描述操作过程。不用担心，你不需要拥有一个电子工程学位就能看得懂这些图，因为图中已把除相关连接之外的东西都忽略掉了，只比典型的系统框图稍微具体点，但也代表了大多数真实示意图的一部分。你必须理解这些示意图才能让嵌入式微处理器很好地工作。如果你能知道控制、数据及地址总线，那你就应该知道怎么读懂示意图了。

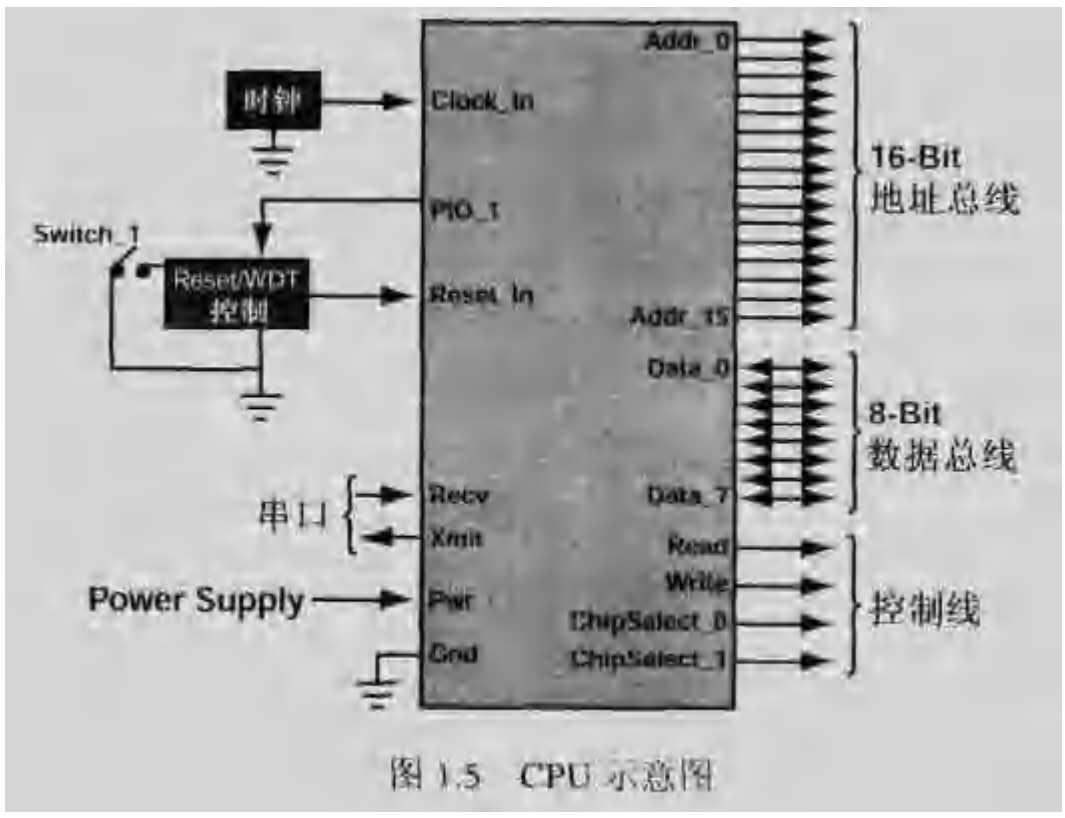


图 1.5　CPU 示意图

在这个示意图中，我们把信号分组，以表明它们是如何与不同的系统总线相关的。注意，几乎所有的 CPU 管脚都被用来实现这些总线了。

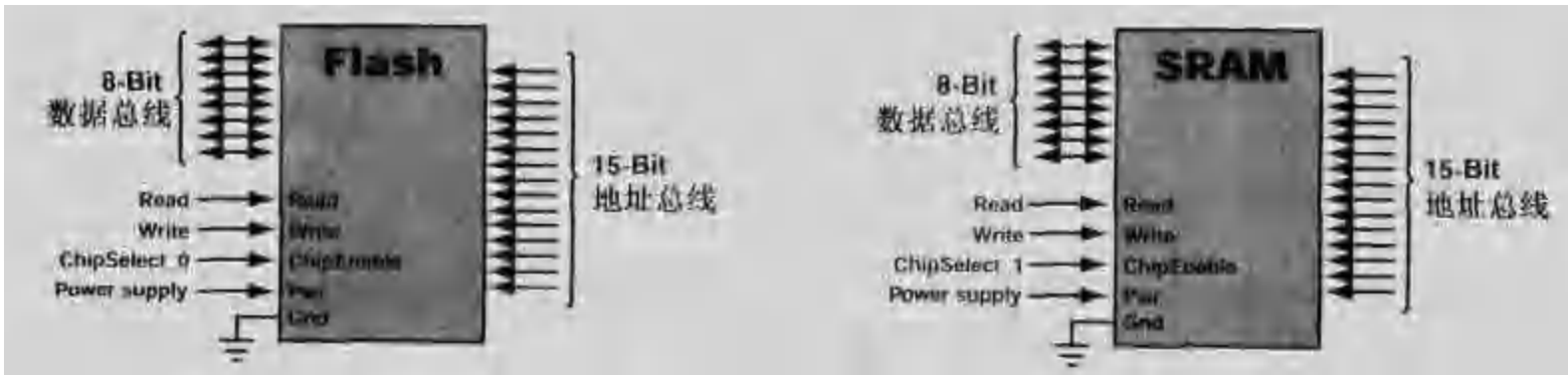


图 1.6　Flash 和 RAM 示意图

存储器直接与系统总线相连。因为每片只有 32KB，所以只占用 15 条地址线。注意，每片是如何被片选电路激活的。

1.8.1 CPU

CPU 的示意图（如图 1.5 所示）只有一个“大”的部分，即 CPU，也包括另两个功能块，即时钟和复位电路。时钟可以使得 CPU 一步一步地进行处理过程。处理过程变化范围从每条指令一个周期（RISC）到每条指令多个周期（CISC）不等。时钟可以是晶振，也可以是完整的时钟电路，可根据 CPU 的需要决定。reset/WDT 电路在复位管脚提供给 CPU 一个逻辑电位，使得 CPU 返回其初始状态。这个特别的电路确保开机或者开关被按下时复位管脚保持一段时间的低电位。请注意，有一个 CPU 之外的 PIO 管脚连进这个电路，它可给 WDT 提供一个周期性的脉冲。

这个设计之中的 CPU 使用 16 位的地址，但是一次只传输 8 位数据。因此它有 16 位的地址总线，却只有 8 位的数据总线。利用这 16 位，处理器可以寻址 64KB 的存储空间，其中的一半是 32KB 的闪存，另外 32KB 是 SRAM（如图 1.6 所示）。

每条 CPU 管脚属于如下四组中的一组，即地址、数据、控制及输入/输出。

在这个简单的示意图中，CPU 的大部分管脚都用做了地址和数据总线。因为每片存储器只能提供 32KB 的地址空间，因此它只有 15 条地址线。图 1.5 中低位 15 地址位直接与存储器上的 15 条线相连，另有两个 CPU 控制信号——ChipSelect_0 和 ChipSelect_1 用来激活适当的存储芯片。管脚 Addr_15 没有利用（如果 CPU 不提供快捷的片选管脚，则可以利用这条管脚和额外一些逻辑电路，即被称为地址解码电路来选择合适的存储器）。

CPU 向存储器里写一个字节，需要把字节的地址输到地址总线上。如果地址是 0x0000，则 CPU 把所有地址线置低电平，也就是逻辑 0 的状态。如果地址是 0xFFFE，则 CPU 将把除了最低位的其他地址线置成高电平，即逻辑 1 的状态。

当某设备没有被选择时，它处于高阻状态（电路上断开）。CPU 还有两根控制线（读和写）控制被选设备怎么与数据总线连接。如果“读”管脚处于激活状态，那么被选存储器的输出电路有效，把数值输到数据总线上去。如果“写”管脚处于激活状态，那么 CPU 的输出电路有效，被选定的存储器只把其输入电路与数据总线相连。

十六进制和总线信号

地址和数据的数值用 ASCII 码来表示是很常见的。例如，“Put 0xBE at 0x26A4”，“put 0xBE”表示把 0xBE 输到数据总线上去，而“0x26A4”表示把 0x26A4 输到地址总线上去。为了了解地址或者数据总线的状态，需要把十六进制转化为二进制。每位十六进制代表 4 位。如下所示：

十六进制	二进制	十六进制	二进制
0x0	0000	0x1	0001
0x2	0010	0x3	0011
0x4	0100	0x5	0101
0x6	0110	0x7	0111
0x8	1000	0x9	1001
0xA	1010	0xB	1011
0xC	1100	0xD	1101
0xE	1110	0xF	1111

因此，0xBE 代表 1011 1110。

请注意，十六进制数字的位数暗示了总线的带宽。0xBE 只有两个十六进制位，表明数据总线只有 8 位带宽。而有 4 位十六进制数的 0x26A4 则意味着地址总线的带宽是 16 位。由于十六进制数和总线大小暗含的这种关系，因此用 0 来填满地址和数据的值以使总线上所有的位都有值，这已经成为一条规则。比如，当我们谈到机器上的 32 位地址的 1 时，应该写 0x00000001 而不是 0x1。

图 1.7 概括了 CPU 和闪存存储设备间的连接。如果对比这个和前面的示意图，则会发现只是多了闪存器件的电源和地的连接。

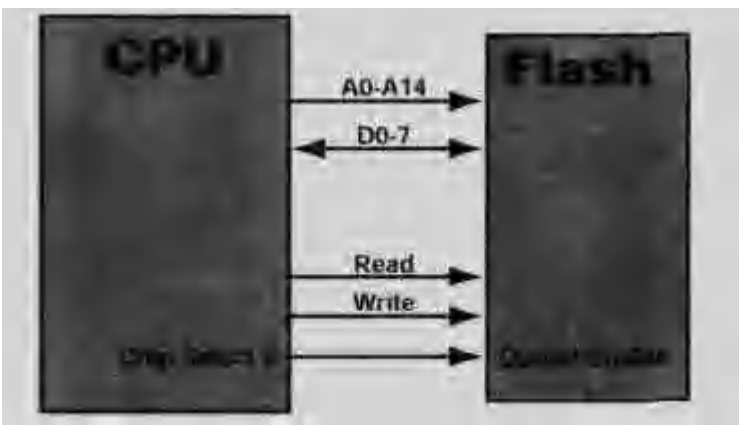


图 1.7　CPU 和引导启动的闪存间的连接

CPU 利用读和写信号来控制各种存储器和外围设备的输出驱动器，以此来控制数据总线上数据流的方向。

CPU 与闪存间的交互可以总结为如下的步骤：

（1）CPU 把所需的地址输到地址总线管脚上；

（2）CPU 使“读”管脚有效；

（3）CPU 激活适当的片选管脚；

（4）快闪存储器把对应地址的数据输出到数据总线上去；

（5）CPU 读取数据总线上的数据；

（6）CPU 释放片选，处理数据。

这一系列步骤允许 CPU 从内存获得最终成为指令的字节。这通常被称为“读取指令”，如果你理解这个接口，则很容易连接到其他的设备上，因为它们的基本原理是一样的。SRAM 的接口除了不同的片选管脚被激活外，其他也是相似的，配置不同的片选线路，使每条线将激活 32KB 的地址空间（ChipSelect_0 选择 0~32KB 范围，ChipSelect_1 选择 32~64KB 的范围）。而写操作访问本质上相同，除了“写”管脚处于有效，则数据流从 CPU 到存储器而不是存储器到 CPU。

这就是地址、数据和控制的本质。你拥有一定数量的地址空间（依赖于存储设备的实际大小）、8 位的数据（不同的设备可能是 16 位或者 32 位），及一些控制线（读、写和片选），示意图中剩下的就只有串口了。由于现在大多处理器内建 UART，这样只有一个驱动器需要连上，而这涉及与 CPU 的交互。换句话说，现在你已经了解基于微处理器简单硬件设计的基本知识。

1.8.2 缓存的功能及缺陷

对于标准的编程环境来说，缓存是一个福音。如果代码利用得当，它可以大大提高运行速度（如图 1.8 所示）。缓存利用所谓的本地引用“Locality of Reference[1]”现象。本地引用表明，在程序执行的任一给定点，CPU 将从一块存储区域反复地读取。如果能够把内存中的内容放在较快的存储区块中，那么将是一种有效地提高访问低速率存储空间的方法。

缓存是一块存储空间，位于 CPU 中，用于改善 CPU 和外设之间通信的时间，实现起来有不同的类型。由于对不同缓存实现的讨论超出了本书的范围，因此我们将在下面部分特别关注一下现在嵌入式系统所用的两种主要的缓存，即指令缓存（I-Cache）和数据缓存（D-Cache）。

正如其名字所表明的那样，两种不同的缓存用在 CPU 对存储器的两种访问方式上，即访问指令空间和访问存储空间。缓存之所以要分成两种访问方式，是因为 CPU 访问两种内存区域的方式不一样。CPU 对数据空间进行读和写的操作，但是对指令空间读比写要多得多。

---

1. M.Morris Mano，Computer System Architecture，第二版.（Englewood Cliffs，NJ：Prentice Hall，1982），第 501 页。

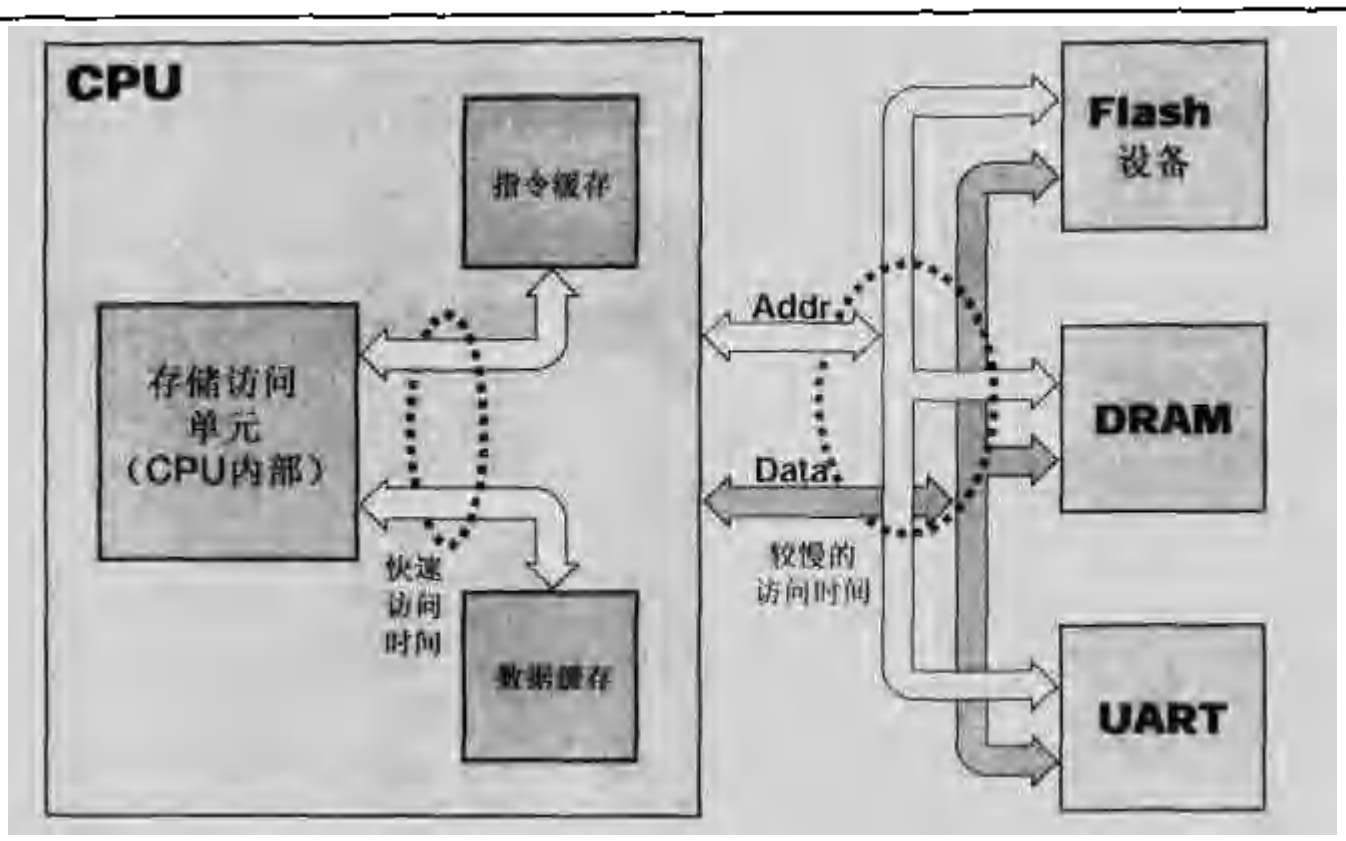


图 1.8　CPU 与外部设备间的缓存

图中显示有不同级别的缓存，最快的是位于 CPU 内部的缓存（称为一级缓存）。

缓存仅有的限制是对于高级的系统编程者来说的，修改指令空间是十分危险的，有时将其称为代码自身修改（Self-modifying Code）。

数据缓存在数据转移过程中会涉及到，因此目的地是物理存储器的指令可能停留在数据缓存中一段时间，直到执行访问的代码完成操作。这里有两种产生错误的可能，即写进指令空间的数据可能不在物理存储器中（因为它还在数据缓存里），或者指令地址的内容可能已经在指令缓存里了。这意味着取指令的操作实际上并没有访问到包含新指令的物理存储器。

图 1.9 表明缓存与数据写入和指令读出的关系：步骤 A 显示 CPU 通过数据缓存向存储器写入；步骤 B 显示数据缓存内容到物理存储器的传输；步骤 C 代表物理存储器到指令空间的传输；步骤 D 表明 CPU 中内存可访问单元从指令缓存中取得指令。如果这一系列事件能够保证按照 A—B—C—D 的步骤进行，那么一切都将运行得很好。但由于这个顺序并不能保证，因此将会消除使用缓存所带来的好处。缓存的要点是只要有可能就跳过 B 或者 C 或者两者同时被跳过，最终结果可能是获取指令被打断。

对于嵌入式系统来说，这个问题变得更糟。明白了上述问题使得第二个问题变得很清楚了。注意到如图 1.8 所示中有一个闪存设备 DRAM 和一个 UART，另外两点复杂性就变得很明显了。

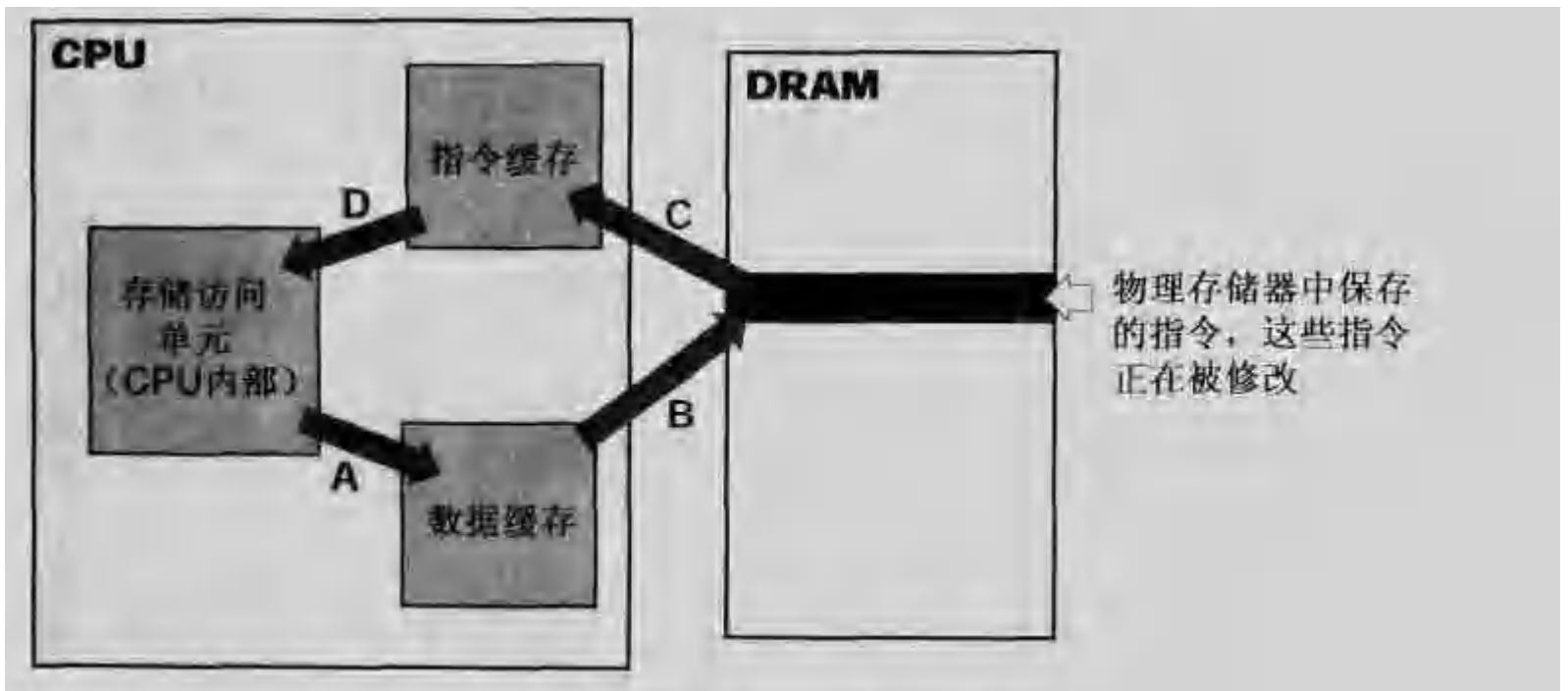


图 1.9 数据缓存和指令缓存的矛盾

缓存通过允许 CPU 从快速的内部缓存，而不是较慢的从外部存储空间获取常用的数值以提高性能。但因缓存控制机制对数据和指令如何被控制做不同的假设，所以代码自我修改可能会产生问题。缓存如果掩盖了外部寄存器状态的改变，则也会产生问题。

（1）UART 与存储器在同一条数据或地址总线上。这意味着访问 UART 或者其他任何 CPU 之外的设备必须考虑到缓存可能插进来。设计硬件只有记住这一点（或者固件必须设置硬件）时，某个 CPU 外部设备才能在不涉及数据缓存时很容易地被访问到。

（2）UART 可以配置成利用 DMA，将输入的数据放进存储空间。在很多系统中，DMA 和缓存是相互独立的。由于 DMA 传输，因此数据缓存有可能不知道存储器的改变。这意味着数据缓存存在于 CPU 和存储空间之间，更多的矛盾将待解决。

硬件的复杂性和速度的需求使得这个问题很棘手，但并不是不可解决。前面大多数问题都通过好的硬件和固件设计解决了。最初的指令缓存和数据缓存的矛盾可通过引入清空数据缓存操作和使指令缓存无效来解决。一次清空操作迫使缓存中的数据写到物理存储器中；一次“失效”操作清空指令缓存使以后的 CPU 请求将获得相应内存的最新拷贝。

同样，缓存也可以帮助解决这些难题。比如，一次彻底的写数据缓存操作保证写进内存的数据同时也被写进物理存储器。这就保证写数据操作将加载到缓存，同时通过缓存写到物理存储器里，因此它仅仅是提高了读数据的速度。类似地，一些 CPU 所谓的“总线侦测”也可以帮助解决与 DMA 和缓存相关的内存矛盾。当 DMA

被用来加进缓存的存储空间时，总线侦测进行检测并自动使相应的缓存区域无效，但不是所有的系统都有实现侦测的硬件。另外，为了避免缓存干涉 CPU 和 UART，设备可以被映射到一块完全不包含缓存的存储区域。CPU 把缓存限制在一定的可缓冲存储区域，而不是指定整个存储空间都可缓冲，这对 CPU 来说是很常见的。

1.9　小结

尽管嵌入式系统正经历一系列可喜的变化，但最底层的硬件层通常都有很多相似性，存储系统通过地址和数据总线与 CPU 交互；系统通过串口和开发环境交互；看门狗定时器即使在软硬件间歇出现 bug 时也能提供更稳定而强健的操作。理解这些设备的通用结构可使你在学习特定新系统时建立很好的框架概念。

本章所涉及到的硬件并不会使你成为一个硬件专家，但可为你更好地理解自己特定硬件的文档做好准备，更重要的是（本书的目的所在），可为你理解本书后面章节中的硬件问题做好准备。

本章主要描述了硬件的一些特点，下一章将着眼于软件问题，特别是怎么给嵌入的目标编译和加载程序。嵌入式系统需要更多的硬件知识，需要能在交叉平台中运行工具，这两点使得嵌入式系统和应用程序的开发区分开来。

第2章 开始动手

第1章介绍了一个典型的嵌入式系统的特点。本章将集中介绍对指定的嵌入式器件是如何构造和加载程序的。

因为我们将工作在一个不可测试的交互式硬件开发环境下，第一个程序是一个很小且很有限的测试程序。尽管这个程序很小，但将其导入目标却并不容易。在希望测试固件之前，你必须做到：

- 了解硬件环境；
- 确认有必需的编程和调试工具；
- 确认编程和测试工具适合硬件环境；
- 执行简单的测试以确认硬件可以工作。

在本章的后面将会学到更多的知识。首先，我们先看一看跨平台编译过程及嵌入式环境与普通PC环境的不同。

在做下一步之前，必须认识到在本书中讨论的程序不包括图形用户界面（GUI），也不需要昂贵的工具。我们用到的开发环境是GNU X-Tools™，在本书的CD中包含了一套完整的X-Tools。本章的例子用到的工具有Gcc、Objcopy、Objdump和Make，且可以使用自己的编辑器。本书用的是Elvis，使用它能很方便地进行二进制文件编辑（以后你会发现它很容易上手）。如果你习惯工作在现在流行的桌面开发环境，那么这些工具看起来是很原始的。虽然这些工具很小，但功能很强。用GUI开发环境进行固件开发就像在雨中打着雨伞工作一样，你会把更多的时间用于把持雨伞而不是如何工作，这比被淋湿好不了多少。刚开始工作在这种简单的命令行工具下，可能会感到有些冷，但是为了理解固件的开发过程，必须放下雨伞。

2.1 在PC上的实现

为了在目标机上构建一个方便的可执行程序，开发者必须自己实现许多操作系统提供的关键服务。这些服务在台式PC上是被用户和程序员当做理所当然的。PC经常用DOS和BIOS执行许多任务，一旦把一些功能引入到嵌入式系统时，就必须知道PC是如何工作的。

考虑程序清单 2.1 中的程序：

程序清单 2.1　一个简单的例子

```
int
PrintAMessage(void)
{
    return(printf("This is a message\n"));
}

int
main(int argc,char *argv[])
{
    int msize;

    sleep(1);
    msize = PrintAMessage();
    return(msize);
}
```

一般来说，这个程序是作为一个可执行文件而存在的，后缀名为.exe。当用户敲入文件名时，路径命令行解释器会搜索文件目录系统且寻找相匹配的文件。文件的头信息用来检查文件是否有效，最后的可执行文件被调入 PC 的 DRAM 并执行。

这个过程需要四步：

- 命令行解释器从键盘上获得文件名，并确认它是不是内部命令；
- 解释器搜索文件系统直到它被找到；
- 加载程序、确认程序并把它调入内存；
- 加载程序在内存中控制程序并执行它。

这四步从最高层次上描述了运行一个程序的过程。如果真的对它感到好奇，那将有更多的细节需要考虑。例如，解释器从哪开始工作？在 PC 内部一个字符是怎样传到 CPU 的？CPU 怎样从磁盘驱动器找到文件？因这些问题中，有些与嵌入式系统相光顾，故需要予以考虑。尽管一个台式 PC 和典型的嵌入式系统有一些不同，但它们基本上是相似的。如果你知道 PC 如何工作，那你会发现理解固件是很容易的。

PC 的 4 个概念对理解嵌入式系统环境是有很大帮助的。

- 命令行接口——在 DOS 中，command.com 是 PC 打开后从磁盘读取的第一

个文件。command.com 程序是一个解释器，可为用户和 PC 硬件之间提供一个接口。在嵌入式系统中，硬件和程序之间没有解释器，程序必须直接和硬件打交道。换句话说，你必须为你的嵌入式系统创建和 command.com 相同功能的解释程序。你不能直接写一句 printf（"hello world\n"）。

- 文件系统——文件系统允许将程序放在大容量的存储器上，并只在需要时才调入处理机的存储器中。在嵌入式系统中，程序一般一直储存在闪存中，因为很少有其他的存储器能检索到程序，但这并不意味着闪存设备和文件系统看起来或是操作起来不一样。
- 加载程序——由于 PC 在同一时间可以运行多个应用程序（通过中断和矢量表），所以程序运行时没有办法预先知道它会在内存中的哪一段。加载程序会通过可执行文件的重定位信息管理存储器的定位细节。在嵌入式系统中，你可以假设一个程序运行在存储器的特定位置，所以你可以在目标存储器的特定位置构造程序。如果一个程序能在运行时动态决定内存地址，则将其称为可重定位（Relocatable）的；如果程序运行在特定的存储区，则称它是绝对（Absolute）的。
- 服务——在 PC 中，应用程序可以假设某些服务是由已经载入 CPU 向量表的中断服务程序提供的。这些服务程序是由 BIOS 建立的服务层的重要组成部分。但由于嵌入式系统没有 BIOS，因此应用程序不能直接得到服务。这意味着原来编译器提供的库不能被直接应用，也就是说你不能假设平台上存在任何便捷的底层方法。例如，一个嵌入式应用程序不能直接用 fopen（）来打开一个文件，甚至不能用 putchar（）函数，因为系统调用的串行端口需要 BIOS 的支持。如果想在嵌入式系统应用这些功能，那你必须自己提供。

总结一下会发现许多 PC 上所应具备的功能均需自己实现。本书的后面将介绍如何去实现。

程序清单 2.1 中程序的执行需在多项前提之下，例如：

- 在进入 main（）之前，堆栈指针必须指向存储器临时存放数据的位置；
- 必须能够向 main（）函数传递参数（这个函数是程序的开头）；
- 一些时间概念必须建立起来，以便让 sleep（）等函数正常工作；
- 当 main（）函数返回时，一些管理者如操作系统（OS）能接管控制。

2.1.1 交互编译过程

在本文的讨论中，有两种不同的编译方法，即本地法和交互法。两种方法产生

的文件都包含二进制机器码，但这是惟一的共同点。

你可能熟悉本地编译。在本地编译中，程序员在 PC 上写一段程序，并在 PC 上编译和运行，程序编译的环境和程序运行环境相同。而嵌入式系统的开发者通过交互编译产生机器码，在 PC 上可以写程序和编译程序，但需在其他平台上运行（编译程序的宿主平台不一定是 PC，也可能是 UNIX 机或其他操作系统平台）。不但它们的宿主机可能不同，目标机和主 CPU 也可能不同。

本地编译的目的是产生一个操作系统加载程序能够识别的文件（加载程序是主机的系统工具，它能提取程序并在系统执行该程序时做必要的工作）。依据主机的操作系统，可执行文件是可重定位的或绝对的。如果是可重定位的，则加载程序会把它放在主机存储器的任何地方，否则会被放在存储器的固定地址（通常是 0）。很明显，存储器中只有一个真正的 0 地址，但加载程序知道主机的 MMU 会采取必要的步骤以便让程序认为是从 0 地址开始运行的。所以在本地环境下，程序员无法知道程序被载入存储器的什么位置。

交互式编译的结果也生成可执行文件，在最后的处理过程之前，这个文件更像 UNIX 或 DOS 机上的标准可执行文件。当我们建立一个启动目标系统的程序时，该文件是不能重定位的[1]。这是因为启动程序总是驻留在存储器的固定位置（不一定是 0 位置），并且启动程序代码一旦被执行，MMU（如果有的话）即被禁用。在启动时，没有更底层的加载程序，更底层的就是电路板了。各种工具可以把可执行程序编译成几种常用的文件格式。在本书中，我们不讨论所有的格式，只讲现在几种有代表性的格式。可执行文件被分为两个主要部分，即一系列头文件和接下来的一系列部件。第一个头文件描述整个文件，接下来的头文件描述各个部件。各部件头文件包含的信息有这部分的物理位置、大小、指令及数据。如果是数据的话，则是否可写。与未被初始化的数据所需的 RAM 空间相应的头文件通常被称为 BSS[2]，因为它没有数据联系，所以并不实际拥有相应的部件。这一空间在开始后被清空，其他包含二进制数据的部分用于引导设备，包含符号信息的部分被调试器用于调试程序。由于我们把精力集中在为引导设备提供信息上，所以没有讨论包含调试信息的设备。

---

1. 一般来说，嵌入式系统的程序目标都是不可重定位的。但是如果程序的目标系统已经拥有底层平台，那么只要该平台支持重定位就是可选的。
2. BSS 是 block started by symbol 的缩写。这个词最初来源于老的 IBM 框架世界，表示一块没有初始化的内存。Gintaras R.Gircys，Understanding and Using COFF，(Sebastopol，CA:O'Reilly &Associates. 1998)，第 9 页。

头文件信息本身并没有给 CPU 太大的帮助，而 CPU 也并不知道头文件，只是想从目标内存的启动位置中取指令并执行。交互式编译过程需要增加一步，那就是把如图 2.1 所示中的格式转化成目前 CPU 所知的可执行文件格式。

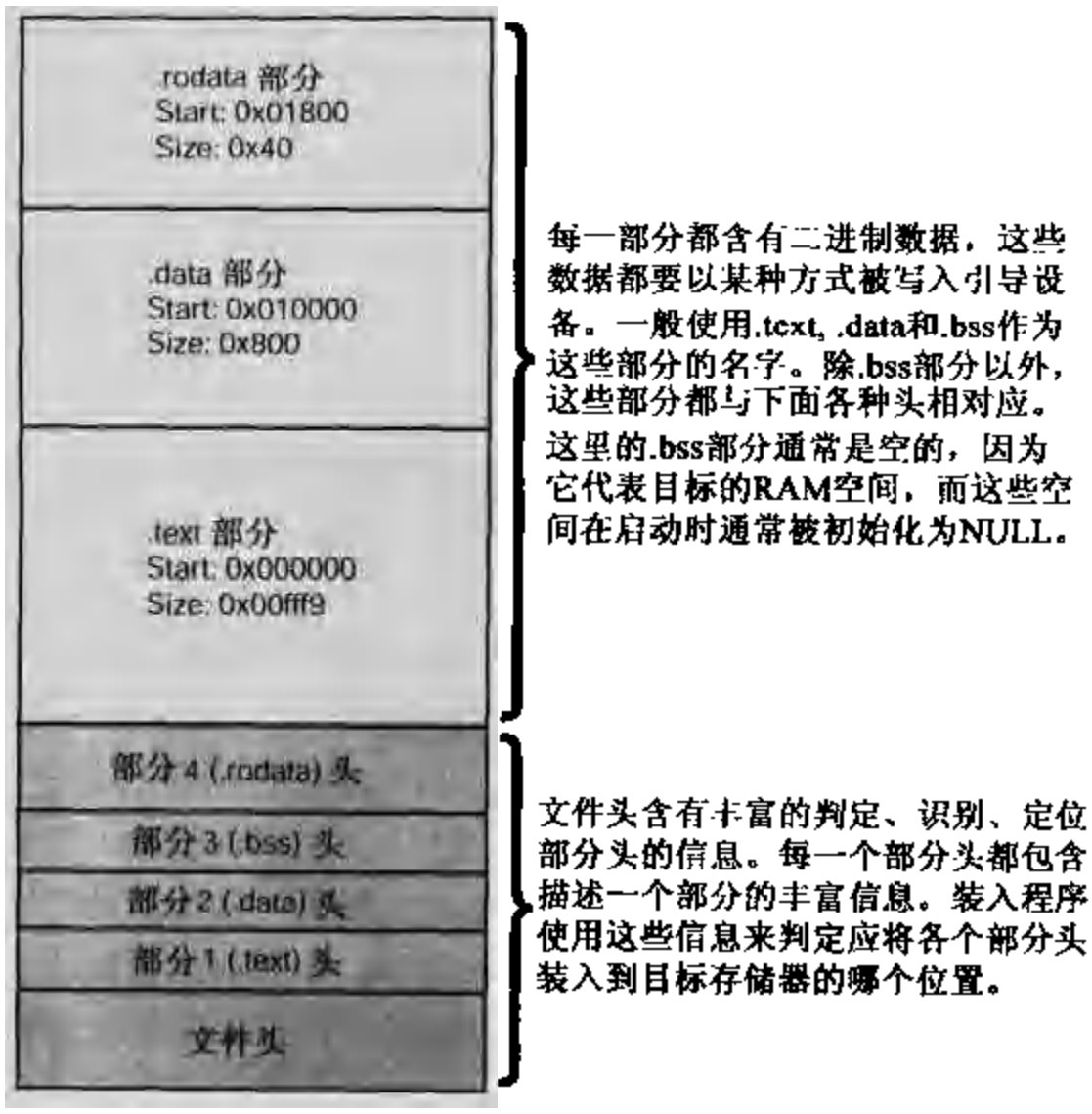


图 2.1 一个可执行文件的格式

可执行文件一般被分为几个部分，每个部分使用一个连续的存储块。为讲解方便，下面假设可执行文件都采用上面的这种结构。

由于可执行文件代表一个绝对的内存映射，因此代码被编译、连接并在内存中的特定位置执行。连接器从输入的内存映射文件中解出必需的位置信息，并告诉连接器各输出文件（文本、数据和 BSS）放在什么地方。

最后一步是把可执行文件转化为二进制信息，这会使处理器把它认为是原始指令或数据。这通常是很简单的一步，但它的复杂度取决于内存映射图。出于讨论方便，可以假设系统的引导闪存设备的开始位置是 0x000000，RAM 开始位置是 0x800000（假设 CPU 是 24 位总线），再假设.text 部分存放在 0，其他非 BSS 部分连接在此位置之上，此时会使二进制转换变得很简单。.test 部分变成映像文件的开始，原始数据.data 和 .rodata 部分紧随其后。

注意，即使指示连接器让每一个部分在内存中紧密连接，每两个部分之间也还是可能有空隙，并通过观察开始地址和每一部分的大小来确定空隙的大小。如果 .text 部分是 0xfff9 位，开始地址为 0x0000，并且 .data 文件紧随其后，那 .data 文件的开始地址将是 0xfff9。如果地址比 0xfff9 高，那说明由于各种原因而使连接器变换了这部分的开始地址。这种转换在映像文件填充时必须考虑到。例如，如果 .data 部分从 0x10000 开始，则必须在连接.data 部分到新的映像文件之前插入 0x10000–0xfff9 位，剩下的各个部分也要考虑转换。这导致了二进制文件看起来类似如图 2.2 所示一样。

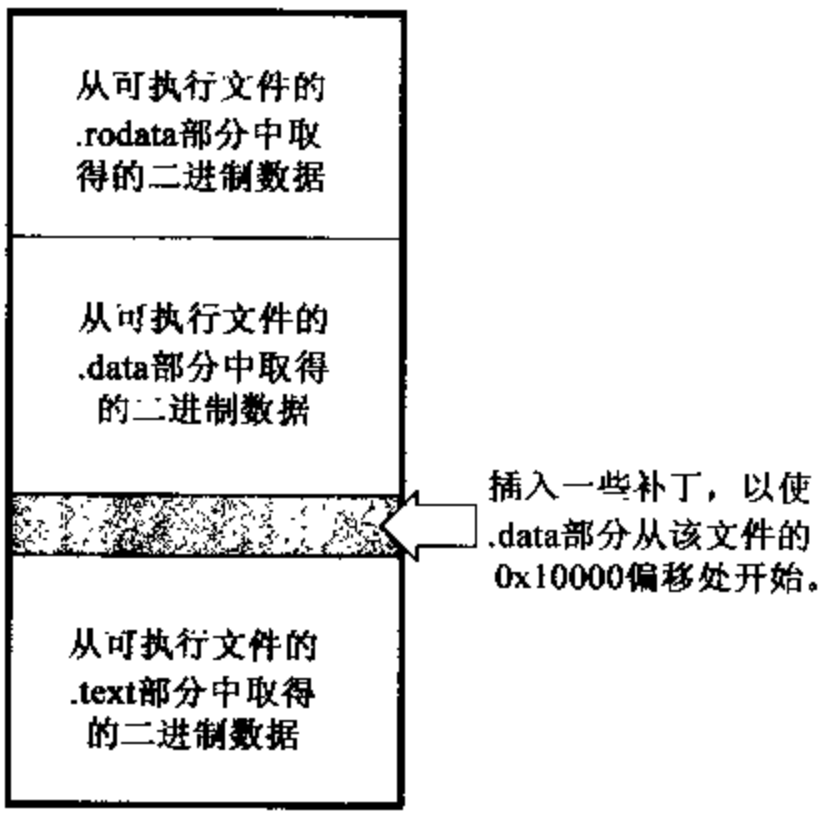


图 2.2　包含启动闪存中原始二进制数据的文件

本图说明图 2.1 中所示的部分是如何映射到最终的二进制映像中，并装入闪存中。注意，这里已没有那些部分头。

图 2.2 的文件包含指令和数据部分，将被 CPU 从内存中取出来并转化为相应的逻辑操作。通过编程器，开发者能把二进制数据写入闪存中的非易失性空间中。在闪存设备装入目标系统后，一旦 CPU 复位，则 CPU 将从中取出二进制数据并运行程序。

现在该知道一些嵌入式系统是如何在内部进行“执行”工作的。开始，文件看起来和 UNIX 或 DOS 机上相似，但相似很快消失没有命令行解释器唤醒系统加载程序，也没有加载可执行文件的文件系统，取而代之的是嵌入式系统拥有一个空的插槽，可以将经过编程的存储器件插在上面。

一个典型的嵌入式系统的加载程序（如果你敢于这么叫的话）工作流程如下：

（1）把原始的二进制文件转移到软盘；

（2）把软盘放入闪存编程器中；

（3）装载软盘并把二进制数据复制到编程器的缓存中；

（4）将闪存设备插入编程器插槽中；

（5）擦除闪存并把编程器缓存中的内容写入闪存；

（6）等待编译，然后移去软盘和闪存器件，将闪存器插入目标主板；

（7）打开电源。

嵌入式系统的加载程序和前面介绍的 PC 加载程序有很大的不同。

现在，交互式开发环境提供了更加方便、快捷的方法。一些程序设计器有网络连接功能，文件能在网络间传输，这省去了软盘那一步。如果你幸运的话，目标系统有 JTAG 和 BDM 接口，那你就可以直接加载程序映像。这可使你不再需要使用外部的编程器。

2.1.2 建立内存映射

内存映射描述了设计者将内存及外围设备放置在什么位置。内存映射可能是简单的闪存或 RAM。换句话说，处理机可能从内存的顶端以外启动，多个器件没有必要在连续的内存空间。

通常我们要设计硬件以便让固件用到所有相连的内存区。所有的闪存放在相临的块中，如 RAM 和 EPROM 等。处理器同时需要一块稳定的块存放硬件的复位信息，这个内存块一般被称为启动区，它是必须有的。CPU 在此复位并引导内存，因此它必须稳定。

内存映射的详细信息决定于硬件设计（就是固件开发的输入）。固件开发者通过被称为连接编译文件和内存映射文件的配置文件来进行连接器和其他工具的对话。通过固件，这些文件将被编译。

2.1.3 连接编译文件

下一个例子很基础但又很完整，显示了连接编译文件如何通知内存的物理地址及文件各部分将被放在何处。内存的每一行描述一个特殊名字、开始地址及每个内存块的长度。如程序清单 2.2 所示，有一个 256KB（0x40000）的闪存，开始位置是 0，还有一个 512KB 的动态内存（0x80000），开始地址是 0x80000。下面的例子

把两个内存块称为 bill 和 mary，可以用它描述你是如何进行配置的。对每一个代码段，其他的开发者均可修改它。

程序清单 2.2　连接编译文件

```
/* Memory Map File for widget.   This hardware has .25Meg of FLASH
 * from 0-0x3ffff and .5Meg of DRAM from at 0x80000-0xfffff.
 * This program is built such that initialized data is left in
 * ROM space.   This keeps things simple, but does require
 * that no initialized data be written at runtime.
 */

MEMORY
{
     flash : org = 0,          len = 0x40000
     dram : org = 0x80000,   len = 0x80000
}

/* Note the use of boot_base, bss_start and bss_end to tag beginning
 * of flash, and boundaries of .bss space respectively.   These tags
 * can be used by the program to reference the hard-coded memory locations
 * they represent.
 */
SECTIONS
{
     .text     :
     {
          boot_base = .;
          *(.text)
     } >flash

     .data     :
     {
          *(.data)
     } >flash
```

```
        .bss      :
        {
            bss_start = .;
            *(.bss)
            bss_end = .;
        } >dram

    }
```

文件建立的每一部分都放在一块内存区中（见程序清单 2.2）。*（.text），*（.data）和*（.bss）行告诉连接器把所有的 text、data 及 BSS 部分放入所指的内存区中。如果需要，则用户可以把所有的目标文件列表以便把它们放入内存中也可在命令行连接器中进行排序。

Boot_base=.，bss_start=.及 bss_end=.行可用来创建标签，并能被 C 代码识别。这些标签被放在内存映射的(.)位置。在程序清单 2.2 中，boot_base 代表地址 0x0000（在闪存块中），bss_start 标签是 BSS 空间的开始，而 bss_end 代表分配给 BSS 的 DRAM 的最后地址。

注意

有些工具包提供这些标签，有些不提供。所以，为了连贯性，最好提供你自己的标签套并使用它们，而不管工具包提供与否。

2.1.4 文本、数据和 BSS

到目前为止，我们集中讨论了内存管理、文本、数据和 BSS 文件。这些都是基本部分，文本部分包括代码（从存储器取出被 CPU 执行）；BSS 部分是不包含初始化的内存部分，通常在开始后清零；数据部分包含初始化信息，还包括变量重定义所需的空间，同时也包含整个代码段的信息。

如何进一步编译数据部分呢？让初始化数据成为可写的。不可写的初始化数据是闪存的一部分，可写的数据也一定是闪存的一部分。但由于数据可写，因此运行时变量一定要放在 RAM 中。当程序开始运行后，所有的初始数据均从闪存复制到

RAM 中，而程序只操作 RAM 中的变量。

程序清单 2.3 的代码列举了不同的数据，实际的功能函数为 func，放在文本部分；Printf（）后面的字符串代表只读数据；变量 justStarted 被初始化，但代码要修改它，所以变量只能放在数据部分以便开始后能从闪存中复制到 RAM 中；变量 sysClock 未被初始化，但可写，所以被放在 BSS 部分中，表示 RAM 空间在开始后清零。

程序清单 2.3　一个用到文本、数据和 BSS 的程序

```
int justStarted = 1;
int sysClock;

int
func()
{
    if (justStarted == 1) {
        printf("Hey, we just started!\n");
        justStarted = 0;
    }
    else {
        printf("Hello sysclock = %d\n",sysClock);
    }
}
```

在汇编语言中，你可以把代码放入数据区，也可以把数据放入代码区。汇编语言的引导程序允许这样做。但是，标准的 C 编译器把不同的二进制文件放在不同的区域，所以你只能按它的方法做。

不同的复位矢量对应不同的内存映射。

值得注意的是不同的 CPU 需用不同的指针指向内存以便复位。一些跳到内存底部，一些在顶部，而有些在中间。不管复位向量指向哪里，都会到闪存的引导区。无论怎样，复位指针都能编译内存映射。

显然，CPU 的复位体系在建立系统的内存映射上起了很大的作用。对于顶端 CPU，引导区在内存的顶端物理地址，而其他都在下面；对中端 CPU，其他内存可以在两边，而底端 CPU 的所有其余内存都在上面。图 2.3 显示了三种情况。

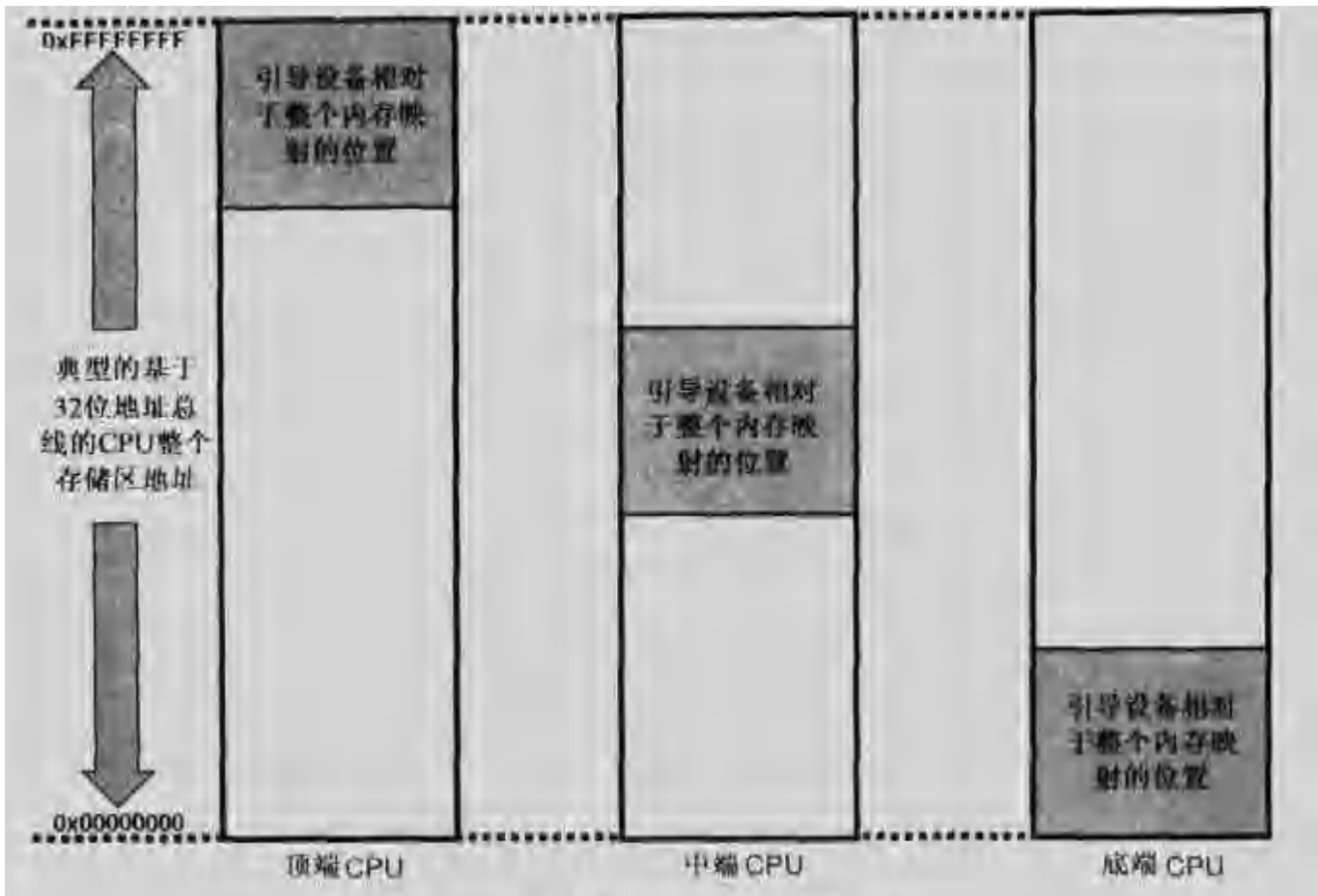


图 2.3 复位向量与闪存引导区设置

处理器的复位操作决定了引导设备的位置（在地址空间），也就是在闪存中的位置。

2.1.5 Make 文件

下面我们将讨论一个简单的 make 文件，主要介绍的是交互编译环境下如何建立进程，而不讨论如何制作 make 文件。

一个简单的 make 文件（见程序清单 2.4~程序清单 2.7），假定程序用 GNU 交互编译工具，且工作在应用 ColdFire 5272 微处理器的 bash 环境，产生的输出文件格式是普通目标文件格式（COFF）。COFF 是许多输出文件格式的一个特例。GNU 工具可产生 COFF、ELF 或 AOUT 文件格式。Bash 提供的环境非常像典型的 UNIX，因此你会看到如 rm 代表 delete 和 cp 代表 copy 的命令。

本书把 make 文件分成四部分，即初始化、连接、修改和组合。

Make 文件在交互式编译环境和本地编译环境下的基本格式是相同的。下面的例子会让你知道交互式编译环境下，make 文件会稍微复杂一点。

程序清单 2.4 ColdFire 5206makefile 初始化部分

```
##############################################################################
#
```

```
# Makefile for building M5272C3 based system.
#

PROG        = myprog
OBJCOPY     = m68k-coff-objcopy
OBJDUMP     = m68k-coff-objdump
NM          = m68k-coff-nm
CC          = m68k-coff-gcc
ASM         = m68k-coff-as
ASMCPP      = cpp -D ASSEMBLY_ONLY
LD          = m68k-coff-ld
CCOPTS      = -Wall -g -c -m5200 -o $@
ASMOPTS     = -m5200 -o $@

OBJS=obj/reset.o obj/start.o obj/cpuio.o obj/main.o
```

Make 文件初始化部分显示命令 GNU/Cold-Fire-specific 代替了标准的 cc 和 ld 命令。代码也定义了另外的命令对不同的输出文件产生的变量。首先 PROG 定义了一个服务作为标准输出文件，你会发现许多不同的输出文件的产生都基于 ld 产生的最终可执行文件，因此 PROG 的定义都基于不同的文件，大多数的工具都是简单的一对一关系。例如，对于 ColdFire，cc 对 m68k-coff-gcc，ld 对 m68k-coff-ld，等等。大多数的选择都是标准的：-g 包含最终输出文件象征信息。-c 告诉编译器某一块不完善，需要最后连接。-o 告诉工具输出放在哪里？注意 CCOPTS 包含-m5200，它告诉编译器 ColdFire 用何种型号的微处理器运行代码。

程序清单 2.5　顶层的所有目标

```
#############################################################################
#
# Top level target to create executable image:
#
all: $(OBJS) makefile
        $(LD) -Map=$(PROG).map -TROM.lnk -nostartfiles -ecoldstart \
            -o $(PROG) $(OBJS)
        coff -B $(PROG).bin $(PROG)
        $(NM) --numeric-sort $(PROG) > $(PROG).sym
```

顶层的所有目标（见程序清单 2.5）取决于在执行连接（LD）前被编译或组合的单独目标块（OBJS）。LD 取出所有的块（reset, start, cpuio, main），并把它们顺序连接在一起，通过-TROM.lnk 建立内存映射（ROM.lnk 是内存映射文件）；-MAP 告诉连接器产生一个文件（myprog.map），以便描述文件产生的内存映射。加载程序在运行 main（）函数前会自动包含一段代码用来初始化。由于嵌入式系统运行在裸机上，因此必须写入代码初始化硬件（reset.s 和 start.c）。-nostartfile 选择器告诉连接器删除默认的开始块(crt0.o)；-ecoldstart 告诉连接器用 coldstart 作为指针(代替默认指针 crt0.o)。

Coff 把 ld 产生的 COFF 输出文件转化成 CPU 要求的二进制文件（可执行 coff 工具由 CD 提供，不是标准 GNU 工具库的一部分），使 NM 命令能方便地让开发者查询 myprog.sym 文件中加载映像中的变量信息。

程序清单 2.6　规则部分

```
#############################################################################
#
# Individual rules:
#
#

obj/cpuio.o: cpuio.c cpuio.h
	$(CC) $(CCOPTS) cpuio.c

obj/start.o: start.c config.h
	$(CC) $(CCOPTS) start.c

obj/main.o: main.c config.h
	$(CC) $(CCOPTS) main.c

obj/reset.o:	reset.s	config.h
	$(ASMCPP) reset.s >tmp.s
	$(ASM) $(ASMOPTS) tmp.s
```

规则部分除了在 CPP 和 ASM 上用做 reset.s 文件外，其余都是按标准使用的。调用工具能让开发者使用 C 语言和汇编语言的所有头文件，尽管一些汇编语言支持 CPP。

Make 文件的最后部分（见程序清单 2.7）显示了一些不需要本地编译的选择项，

也证明了如何有组织地确定一个 PROG。根据不同的输出文件，Make 确定的目标能出 ld 产生。S-record 文件格式能够用来代替 coff 工具产生的二进制文件格式。Showmap 存放头信息。由于头部分包含了整个部分的信息，因此 showmap 在试图诊断不正确的内存映射时是很有用的。dis 和 disx 目标段存放的列表包括原始资料的分解与组合。当我们用原始调试器在函数内部设置断点时，列表是很有用的。

程序清单 2.7　其他部分

```
##################################################################
#
# Miscellaneous utilities:
#
clean:
	rm -f obj/*


clobber:	clean
	rm -f $(PROG) $(PROG).map
	rm -f $(PROG).bin $(PROG).srec $(PROG).dis $(PROG).sym

gnusrec:
	$(OBJCOPY) -F srec $(PROG) $(PROG).srec

showmap:
	$(OBJDUMP) --section-headers $(PROG)

dis:
	$(OBJDUMP) --source --disassemble $(PROG) >$(PROG).dis

disx:
	$(OBJDUMP) --source --disassemble --show-raw-insn $(PROG) >$(PROG).dis
```

2.2　建立库

库收集了常用的函数，像 strcpy（），printf（），strcmp（），atoi（）等。这些函数能组成一些基本的程序。例如，如果程序用 strcmp（）函数，则 strcmp（）的

代码将从库中取出且应用到程序中去。不管库中包含多少个函数，只有程序需要用到的函数，其代码才能被调入到程序中，所以在用库时，必须做到两点：

- 库中有一个只有加入 strcmp.o、printf.o 及 atoi.o 等才能连接的工程。程序组不需要提醒它们从库中需得到什么函数，程序也只需包含一些像-lc 之类的命令到连接行，其他的就不用管了。
- 库减少了最终程序块的大小（和包含库原始信息相比），这是因为不需要的函数没有放到可执行文件中。

在一些情况下，你可能需要建立自己的库。例如，当代码要涉及的外围窗口部件被许多工程应用时，把函数包做成一个库是非常方便的。

如果你对进程只了解一点，则可能会通过把所有相关的函数[widget_funcA（），widget_funcB（），widget_funcC（）]放到一个简单文件中的方法来建库，且用-c 来编译，产生一个目标（.o）文件，然后通过一些工具（lib 或 ar）从.o 文件产生一个库（.a）。

但这种方法并不是很有效，因为需将所有的函数编译成一个简单的.o 文件。一个库基本上是目标块的串联（或.o 文件），当连接器进入库寻找相关文件时，会搜索每一个目标块。当发现相匹配时，它会把整个目标块放入程序代码，而不管在这目标块中有多少个函数。如果一些函数一起被编译成一个.o 文件，那么将导致和一个函数相关的所有函数均被组合进程序中。用这样库的用户将不知道多花多少钱。这个问题很容易解决，只需在建库时，把每个函数放入分别建立的特殊文件中即可。

浪费空间并不是避免集成建库的惟一原因。假如我想用前面提到的库作界面部件的接口，那么可以用特殊的标记写自己的 widget_funcB（），并使用库里的 widget_funcA（）。如果用集成库，那么在程序的最后有两个 widget_funcB（），因为当连接器进入库取 widget_funcA（）时，另外的函数也被调出来了，也即库中的 widget_funcB（）和我写的 widget_funcB（）之间有冲突。

当试图用自己的 printf（）时[当然，在嵌入式系统编程你迟早会用自己写的 printf（）]，如想用 printf（）调用 vsprintf（）（假设在库中），但因 printf（）和 vsprintf（）位于库中相同的模块上，连接器显示有两个 printf（）函数（自己的和用 vsprintf（）结果调出来的），所以不能调用 vsprintf（）。

重要的是要记住，无论何时你在建自己的库时，一定要把一个函数放在一个文件里，并建立许多独立的块，这样才不会发生冲突并可节省内存空间。

2.3 准备活动

在介绍写闪存器件前，确信你已准备好了。下面将讨论一些初步的措施。许多

措施看起来明显，但他们很重要，值得一提。下面要讨论的内容包括：

- 参与硬件设计；
- 认识硬件并善待设计师；
- 拥有所有数据的本地备份；
- 确信硬件可以工作；
- 慢慢开始；
- 查看你生成的文件。

首先需考虑在开发过程中所应用到的工具，并确定你的设计能够用这些工具进行编译。一些基于硬件的开发调试工具是能在嵌入式系统下工作的，包括仿真器、逻辑分析器、JTAG 和 BDM 接口及逻辑探测器。所有这些工具能直接被目标器件利用。

仿真器

通常，电路内仿真器（ICE）是插入硬件 CPU 插槽的一个器件。一个仿真器为一个特殊的 CPU 设计，可比其他工具更接近 CPU 仿真。仿真器可提供许多特殊功能，像指令跟踪缓存器（收集先前可执行指令）、复杂断点及 CPU 指令高速缓存区。如果需要复杂的调试，则需要一个缓存器。既然仿真器可为固件调试提供灵活的方法，那么我们为什么用其他的呢？原因是它并非免费，且都很贵。

如果一个仿真器能提供最深入而又灵活的硬件调试，为什么还用其他的呢？但这种复杂的功能并不是免费的。仿真器通常很贵。

因为仿真器通常插入和 CPU 相同的插槽中，所以主板必须为仿真器准备好条件。例如，主板设计者必须在 CPU 周围留出足够的空间，以便插入仿真器或大体积的连接器。

如果你计划用仿真器，则你必须确定硬件能够处理连接仿真器和主板所带来的物理需求。这些物理通道有时和外围设备连接，连接器通过多管脚的带状线挂在 12 英寸范围之内。因此，如果目标在一个插件上或右面紧邻一个插件，那你就有麻烦了。

因为要代替 CPU，所以仿真器很复杂。内存运行逻辑仿真器加长了内存存取循环的时间，仿真器一般不在 CPU 全速运行时运行且当你把它插入主板时，仿真器的加载程序可能和系统上 CPU 的加载程序不同。在某些情况下，这种不同会导致噪音，让你的系统在噪声环境下运行，直到仿真器被拿开。有时，这种不同还会扰乱硬件，从而引起其他问题。

在目标板上插进插座的另一个问题：仿真器提供给目标系统的装载可能与CPU已经提供的完全不同。这种不同可能掩盖或导致错误。在某些情况，这种不同可能减少引入的噪音，允许系统在有噪音的环境下运行，直到仿真器被去掉。有时这种不同干扰目标的硬件会导致一些本不应该出现的问题。

尽管有这些警告，仿真器依然很容易上手。这些问题的出现很大程度上取决于环境。

2.3.1 开始硬件设计

首先，需确定引导器是方便的、可编程的。这一点很明显，但我们经常发现一些焊在主板上的引导器是不可编程的，除非我们取下器件。这种设计是痛苦的，特别是对那些写固件引导程序的人。因此引导器要可编程的，而不需要用焊锡固定。

设计的同时也需要包含一些机构以便让固件的引导器能方便的和其设计者交互。这种交互要通过一系列端口和一些LED进行。如果应用程序极端消耗，则额外部分的硬件请求不执行，这可能包括一些在最终产品上的连接器。在开发期间，连接器能为调试提供额外接口。这个决定以花销限制和其他因素为转移，但却可使开发过程的一些连接节省很多时间。

接下来将谈到JTAG。如果CPU有调试的接口，那么要确保连接的管脚是可以接触得到的。这些接口变得非常有用，尤其是当没有其他的通信设备连接到处理器时。当硬件已经排好后，即可找到类似的JTAG，对你的CPU有效的工具，找到你打算用于工具的管脚输出端并保证硬件与之有一个连接。

逻辑分析仪

逻辑分析仪用于一般的逻辑分析和数字硬件设计。由于对固件调试工具需要的增长，因此公司生产的逻辑分析仪加入了异常分支和扩充设备，以便让其看起来更像固件开发工具。例如，在CPU工作时，逻辑分析仪可把处理器的地址和数据总线连接起来，进行采集和存取。当开发者观测缓存区时，它不仅仅能显示地址总线和数据总线的变量，而且也能以反汇编显示。

所有仿真器应用时的缺点逻辑分析仪都能克服。逻辑分析仪通常很贵，且需要很多连接（如果你想用它们跟踪指令和数据），可提供系统加载程序。当用逻辑分析仪进行固件调试时，可用于监控运算器的物理访问地址和数据总线。这些外部的交换并不是数据和指令在缓存区的检索。如果高速缓存启用，则跟

踪缓存不会告诉你整个结果；如果高速缓存禁用，则用户必须意识到处理器是在取指令块而不是每次取一条指令，同时一些块中的指令不会被运行。这种取法有时被称为存储器指令预读。以本人的观点，逻辑分析仪尽管很贵，但它却能进行硬件和固件的开发，也能用于不同的硬件平台，包括 CPU。

2.3.2　认识硬件并善待设计师

固件和硬件之间的良好连接在整个工程中能节省很多的麻烦和时间。让我们通过经验来看一看，一些很小的、细微的问题会令你怀疑硬件，但却并不是硬件的问题。这个建议看起来很普通，但能令你在开发过程中有所提高。

了解硬件的意思并不是说你必须趴在设计师的肩膀上看他们写硬件描述语言（VHDL），但是如果你能至少熟悉 CPU 的结构框图将是十分有用的。花点时间和硬件设计师聊聊，提一些问题，与设计师建立良好的关系，取得设计图纸的备份并做好标记，对你的设计都是很重要的。除此之外，你还必须花一些时间了解你打算使用的处理机。

JTAG 和 BDM 接口

JTAG 和 BDM 接口在硬件开发中变得很流行。JTAG 或联合测试存取组是 BIST 硬件的标准接口。由于它被扩充，因此能用于嵌入式系统。背景调试模式（BDM）与其相似，专门用于与 CPU 相关的调试。JTAG 和 BDM 接口工作必须有 CPU 支持，所以不能认为它可以用于所有工程，而要取决于处理器的选择。JTAG 和 BDM 接口的花费一般少于 1 000 美元，能被许多处理器族（不是一个）应用，且只需要小的低脚带状线和处理器连接。JTAG 和 BDM 的缺点是它们需要依靠 CPU 的编译接口，且不能提供任何形式的缓存；优点是能够帮助固件调试，也能用于编写闪存。你一般可以用这个少于 1 000 美元的工具节省开发所需的外出工作。

2.3.3　拥有所有数据的本地备份

你必须了解比设计图内容更多的知识。设计图上的每一个器件都可能有一个 200 页的手册，因目前的器件越来越复杂且越来越密集，如硅器件。这就要求固件开发者均需要掌握器件的基本性能。在这个电气时代，相关的手册大多数都能得到，通过手册可让你知道设计时的错误或对器件性能的了解。

借助手册可检查器件并看是否有勘误表，能够发现器件有问题并不普通，特别是当你用的器件是企业新生产的器件时。

2.3.4 确信硬件可以工作

如果硬件设计商标是新的，且主板刚从工厂取出，那么在你工作之前需确定它们是好的。我们的第一步是确定主板和闪存连接正确。如果你第一次用主板，则应确定如何正确加电源，这一点乍听起来好像愚蠢，但你会确定硬件设计的第一步有没有走对，会不会因为电源接错而烧了主板。

2.3.5 慢慢开始

像婴儿学步一样，在你测试小的引导代码之前，不要测试大的程序。你事先要考虑做的事：

- 你的程序是否和内存空间匹配？
- 你是否真能理解 CPU 如何复位？
- 你的可执行文件是否能正确转化为二进制文件？
- 编程器设置是否正确？
- 如果你的引导内存比 8 位宽，并包含多个器件，你能确定放入正确器件吗？如果是奇字节，那么是 MSB 还是 LSB？
- 硬件工作吗？

谦虚一点会节省你许多额外的工作。你可以查找 CPU 制造商手册，从而获得引导程序代码。在很多情况下，你会发现问题。如果有可能，则可以寻找硬件助手。如果你不知道如何使用示波器或逻辑分析仪，则可以寻找会用的人，这些工具在这个阶段是非常有用的。

注意

逻辑探测器的使用有时会有限制，但其使用的方便和价格值得一提。逻辑探测器能在探测器连接处读逻辑层（高、低），且可支持检测时钟，就像笔一样让你接近源文件，用“笔尖”接触管脚读它的逻辑，读出（像 LED）指示管脚逻辑是高、低、平稳或变化，尤其在你确定硬件稳定时更加有用。

2.3.6　查看你的生成文件

建立工具能让你存放内存映射，可使你写闪存前查看 S-记录或二进制文件，甚至文件的大小也能给你一些线索。如果程序在汇编语言中包含小循环，则最终二进制文件会很小。

找一些能以 ASCII 码形式展示二进制文件的工具（我用 ELVIS），并用这些工具建立进程。例如，证明闪存寄存器代码放置正确，修正一些基于闪存的源文件（见程序清单 2.8）。源文件转化成二进制文件后，用存储工具检查文件，检查偏移格式是否符合闪存。

程序清单 2.8　“让”代码确定位置

```
coldstart:
    .byte    0x31, 0x32, 0x33, 0x34
    assembler code here
```

程序清单 2.9 是一个从 elvis vi 复制偏移量到文件的例子。数据是十六进制 ASCII 和正常 ASCII 形式，因此闪存偏移量是 0x0000。

程序清单 2.9　转存的例子

```
OFFSET    ASCII_CODED_HEX_DATA                        ASCII_DATA
000000:   31 32 33 34 ff fd 78 14    38 60 00 30 4b fc 00 0e   1234..x.8`.0K"..
000010:   38 80 00 00 38 a0 00 00    38 c0 00 00 3c e0 00 04   8C..8a..*l..<x..
```

在程序清单 2.9 中，你需要看的是文件的第一个字节如你所料（0x31，0x32，0x33 和 0x34）。

提示

在你不得不把数据分开时，二进制存储器很有用，因此可以对硬件中多个平行器件编程。在分离前如程序清单 2.9 所示，在分离后（假定分为两个文件）一个在“Split A”中，另一个在“Split B”中，需区分清楚两个文件。

```
Split A
OFFSET    ASCII_CODED_HEX_DATA                        ASCII_DATA
000000:   31 33 ff 78 38 00 4b 00    38 00 38 00 38 00 3c 00   13.x8.K.8.8.*.<.
```

Split B

OFFSET	ASCII_CODED_HEX_DATA	ASCII_DATA
000000:	32 34 fd 14 60 30 fc 0e 80 00 a0 00 c0 00 e0 04	24..`0".C.a.l.x.

相关器件与 CPU 地址

一些工具可产生二进制文件，另一些工具可产生 S-记录文件。如所讨论，由于二进制文件包含闪存器中的精确数据，因此二进制文件能直接转化为程序写入闪存器。另一方面，S-记录是文本文件，在写入闪存器前必须转化为原始二进制文件。这种转化通常需通过编程器进行，因此你必须把 S-记录文件放到编程器，工具组必须包含能描述 S-记录格式的工具。这种格式看起来如下：

<S><T><LL><AAAA…><DDDDDDDDDDDD…><CC>

当 S 指示 S-记录，T 通常是 1，2 或 3 指示 AAAA 区的大小。LL 区包含记录的长度；AAAA 区是十六进制地址，通常是 4B，6B 或 8B 长度（16，24，32 位地址）。DDDD…是数据存放地址，CC 是检测行，进行 S-记录的快速统计。警告：你用的工具在你的程序建立 S-记录时，其结果无法调节。这是因为产生的 AAAA 区和 CPU 相关而不是闪存器。CPU 引导区的物理地址不在 CPU 地址空间的 0x00000000。

根据如图 2.4 所示的配置 1，引导区在和 CPU 相关的内存区的 0 位置，S-记录的 CPU 相关地址和闪存器件的地址有关，它也会很好地工作，配置 2 不会工作。这里 CPU 的引导区不在 0 位置，所以闪存零偏移量不依靠 0 物理地址。假定 CPU 引导区在 0x8F000000，所以 S-记录文件在 AAAA…区开始地址为 0x8F000000，因为 CPU 在这里寻找指令。然而，当我们完成硬件设计开始往闪存写程序时，我们必须把 S-记录的地址从 0x8F000000 改为 0x00000000，因为 CPU 中的 0x8F000000 地址和闪存中 0x00000000 地址是等价的。S-记录空间地址可事先修改，也可在编程器中修改，如果支持修改基址的话。本人的意见是用原始的二进制文件代替 S-记录文件，以避免麻烦。

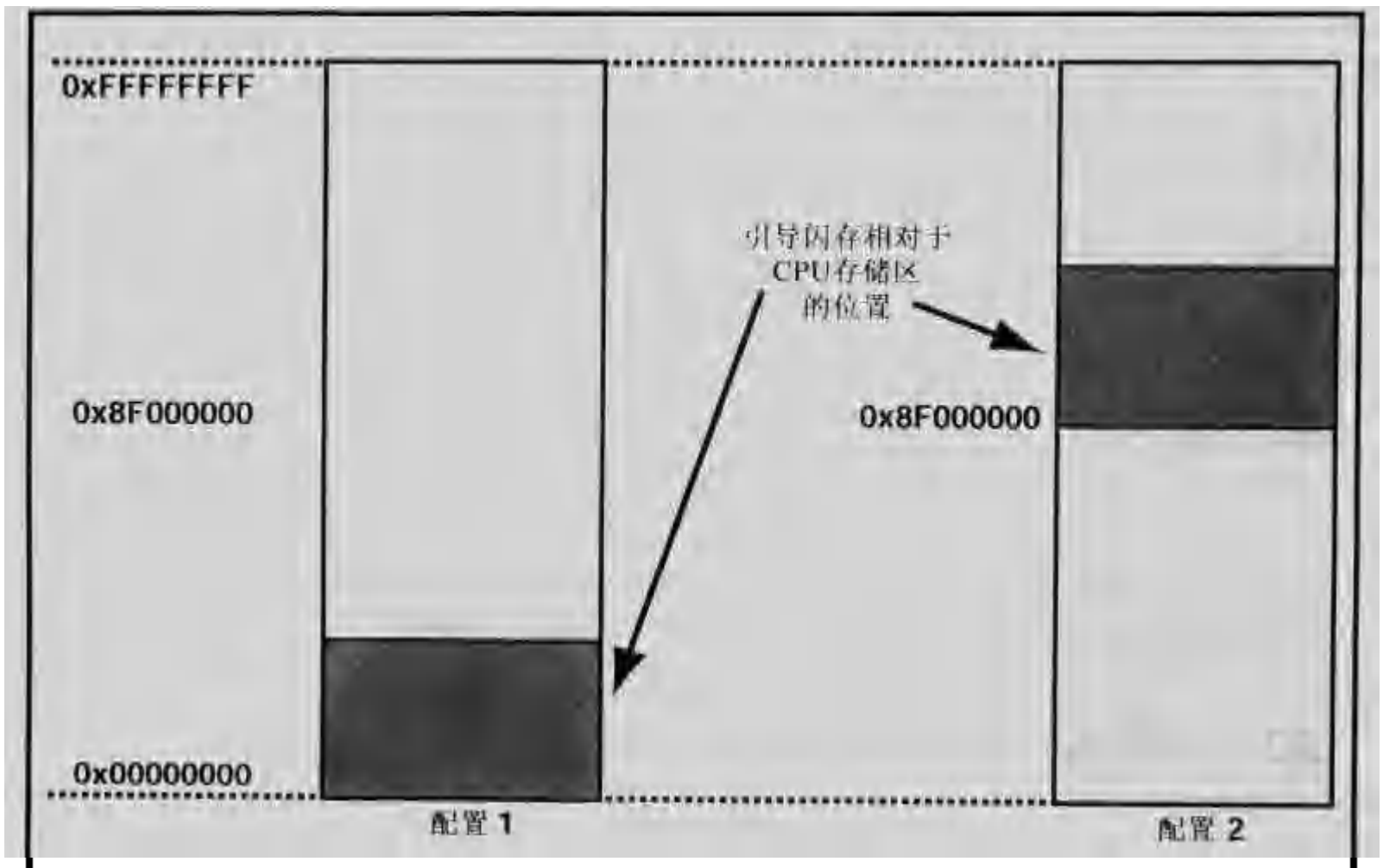


图 2.4　闪存与 CPU 地址空间

由于存储器通常占用处理器地址空间的一部分，因此目标文件（如 S-记录）的绝对地址需要程序编译器正确解释和修订。这是因为程序编译器只知道内存地址而不是处理器的地址。

2.4　运行时间

假如你预先知道细节，则可以进行你的硬件作业设计了。下一部分将讨论当工作在嵌入式系统下时，首先需要做的步骤。

2.5　为固件开发进行全面的硬件测试

如果你的工作做完了，但对硬件产生怀疑时，那你将如何做呢？实际上，如果是复杂的硬件设计，你不会做太多，但这里有一些高水平的测试可帮你确定常见的问题。假定你读了设计图，手里有万用表和示波器或逻辑分析仪——不要期望更多，这足够了。尽管你发现了问题，但你也需要等待硬件人员来确定你发现的问题。这

会让你对硬件的认识更进一步，且对你总是有帮助的。

2.5.1 确定电源电压

第一步是要确定电源是否插上。不是开玩笑！用万用表去测电源提供的电压。之后，用示波器确定电压稳定性以及噪声和振荡。单独一个仪表是不够的。

2.5.2 验证时钟的有效性

使用示波器来验证输入到处理器的时钟是稳定在你预期的频率附近的。一般来说，只要时钟是稳定的系统就会工作，但是如果时钟的频率或是占空比不对的话还是有可能导致问题，因此不要假设只要时钟稳定就没问题了。

2.5.3 检查启动芯片的片选和读信号

将示波器探头连在启动芯片的片选端，按下复位键，看看输出电平是不是跳变到激活状态（至少有一瞬间）。重复上述步骤来检查读信号。

2.5.4 取出放大镜

如果你见过近期出的元器件，就会明白为什么你会需要一个放大镜了。看看CPU 和启动芯片，那些管脚实在是太靠近了！检查一下有没有抬起的管脚、被焊锡误连的管脚或是虚焊连接点（看起来比其他好的连接点要暗得多）。如果你确实发现了上述问题，则需把情况报告给实验室负责焊接工作的人员。除非你对焊接这些细小管脚的工具很熟悉了，否则最好不要去碰它们。因为一旦使用不当，极有可能损坏元件甚至整个印刷电路板。

2.5.5 小心静电

不管你用什么方式处理硬件，你都要预防静电放电（ESD）的问题，可在手腕上或脚踝上戴上皮带，并且和设计者研究一些其他的预防措施。ESD 还不足以损坏硬件，它只会使硬件工作异常。对于嵌入式系统来说，没有什么比异常更糟糕的事情了。

2.5.6　复位时简单的循环

复位时来一个简单的循环以验证启动闪存设备的片选信号总是跳变的。这一步可以是一条简单的汇编语言语句（见程序清单 2.10）。

程序清单 2.10　确认启动地址

```
coldstart:
    jmp coldstart
```

如果不用外部测试设备来观察片选信号的跳变，那么程序清单 2.10 的测试可能还不足以证明系统正常工作了。由于 CPU 处理未编程的引导设备的情况各不相同，因此 CPU 在执行空语句或是闪存设备中乱七八糟的其他语句时可能表现得仍然很正常，所以看到闪存设备片选信号跳变并不等于 CPU 正在执行测试程序。从另外一方面来说，如果片选信号不跳变，则我们可以肯定有什么地方出问题了。如果你接一个逻辑分析仪到设备上，也许可以从逻辑分析仪中得出哪儿出问题了。

2.5.7　一个 LED 的重要作用

前面已经提到过示波器和逻辑分析仪的用处。如果你既没有示波器也没有逻辑分析仪，则希望你能在设计中最好包含 LED。通过 LED 的闪烁频率或是亮灭的状态可以监控系统中的一些信号。LED 并不是一个好用的工具，因为你不知道 LED 是一直亮着还是因为闪烁得太快看起来是一直亮着的。如果 LED 是亮着的，那是因为它通电点亮了还是仅仅因为闪存设备没有插到插座里亮着呢？一旦你确定了在某一点上只要 LED 亮就表示正常工作了，那你就可以使用 LED 来很方便地代替示波器。在测试结束的时候让 LED 点亮并进入一个死循环（让 LED 一直亮着）。通过这种方式，LED 可以用来标识任何一个你想要测试的地方。

在此基础上，建立 3 个小的汇编语言程序，即 LED_on，LED_bslow 及 LED_bfast。LED_bslow 和 LED_bfast 中 LED 的闪烁频率应该相差很大，这样便于区分。这样，这 3 个程序就可以用来表示正在处理、成功及失败 3 种状态了。另外，如果你有不止 1 个 LED（比如 4 个），那么你就可以表示 16 种状态了。

2.5.8　RAM 和“不需要堆栈的”串行输出

下一步的工作取决于你的硬件上有什么。这些测试用来确定为了支持编译过的

C 语言程序系统所需的最少功能，主要的工作是检查运行时堆栈是否可靠。如果处理器有 RAM，就应该尽量使用它们，尽管内部 RAM 不包含外部地址和数据总线。测试外部 SRAM 可以通过一个循环，将所有 RAM 空间置成 0x55，然后读回这些值，再置成 0xAA，再读回。如此不断循环下去。在两个不同的测试中，示波器可以用来分别观察读写信号，确保数据线上出现的数据总是 0x55 或 0xaa。

如果系统中没有任何 SRAM，则下一步的工作就是处理串行输出口。开始的时候先简单地往串口循环写入一个字符，暂时忽略串口状态，即使把缓冲区写溢出了，字符也应该能在 RS-232 串口上偶尔出现几次。如果没有的话，则把一个示波器连到串口的 TX 端上看看是否有什么反应。如果没有任何反应，那问题就比较棘手了，有可能是因为写入设备已经工作了，但 UART（通用异步收发机）由于一些参数（比如波特率）设置得不对而出了问题。另外，如果将设备连到一个终端，则可能需要一个 nullmodem 连接器。

如果硬件上只有 DRAM，则在一个还未证实已经正常工作的设备上初始化 DRAM 是很困难的。这有可能需要你和硬件设计师进行讨论。初始化 DRAM 的细节已经超出了这本书的范围。LED 和串口对于确定 DRAM 的初始状态是很有用的。

2.5.9 开始 C 语言层次

到现在为止，你已经让引导设备正常工作了，且测试了一些存储器，点亮了一个 LED，并且发送了几个字符到串口。信不信由你，在新设备上的这些进展是喜人的。下一步就该转向 C 语言层次了。因为你有了工作 RAM，所以转向 C 并不是件难事，建立一个不使用全局变量而只使用堆栈的简单 C 语言函数。该函数利用一个 for 循环在 LED 状态变化的间隔中插入延时。假设 LED 的地址是在 0xff8000，将该位设成 0 以点亮 LED，那么对应的 C 语言函数应该类似于程序清单 2.11。

程序清单 2.11　点亮一个 LED

```
#define LED_ADDRESS 0xff8000
#define LED_ON          0x0001
#define LED_OFF         0x0000

void
First_C_Function(void)
{
```

```
    volatile int loopcnt;

    while(1)
    {
        *(unsigned short *)LED_ADDRESS = LED_ON;
        for(loopcnt=0;loopcnt < 500000; loopcnt++);
        *(unsigned short *)LED_ADDRESS = LED_OFF;
        for(loopcnt=0;loopcnt < 500000; loopcnt++);
    }
}
```

现在使用汇编语言将堆栈指针初始化到一个能被 16 整除的地址，留出几百字节的堆栈空间，确保知道堆栈的增长方向并跳到 C 函数的地址开始执行。如果一切正常，则 LED 应该会闪烁。

重复上述测试，但这次将 loopcnt 变量设为静态的。如果程序仍能正常工作，则可以说明已经在连接器映射存储空间时建立了合适的 BSS 空间（通过将 loopcnt 变量设为静态的，你就告诉了编译器不要将该变量放在本地堆栈中，因此该变量就被映射到由.bss 部分分配的空间中去了）。

提示　不要用 C 语言改变初始化数据

当你使串口开始工作的时候，测试程序可能已经包含 100 行汇编代码和一堆 C 语言函数了。情况开始变化得越来越快了，但为了简单起见，你必须做出一点小小的牺牲，暂时不要写任何改变初始化数据的代码。初始化变量对于在工作台上的编程人员来说是可以改变的。在嵌入式系统中，数据的初始值在系统加电时必须存在于内存中的某个地方，表明初始化数据必须存储在非易失性的存储空间中。如果这些数据将被 C 语言改变，那么一旦开始执行 C 语言程序，这些数据就不可能存在于非易失性存储空间中了，因为该空间是不可写的。解决这一矛盾的方法是初始化数据存储在非易失性空间中，在启动的时候（执行 C 语言之前）将之复制到易失性空间，这样问题就解决了。这一步并不是很难，但从一套工具转到另一套工具的时候过程可能稍稍有些不同。目前避免这个问题的最简单办法就是不要假设你的初始化数据是可写的。

2.6 小结

完成这一步，后面还有很多更复杂更繁琐的步骤等着你呢。几个闪烁的 LED 和终端上出现的几个字符并不能给人留下很深的印象，但他们表明了很大的进步。

你已经可以正常使用基本的功能了（内存和串口），而且已经建立了一个 C 语言函数，下一步就是用 C 语言写一个简单的 putchar（）和 getchar（）函数。这些就能允许你使用 printf 函数了。简而言之，你已经拥有了一个可以工作的计算机了！

虽然在前面已经提过但仍然很重要的就是，在这个简单的启动过程中要一小步一小步地做，不要凭想像做任何事情，确保缓存是关闭的并且不对 C 函数做任何优化，稳定而持续的进展才是有效的进展。

第 3 章　微型监控器

以后几章的目的是建立一个大型的程序以实现一个可以扩展的框架平台。在此平台上，你可以添加自己的程序。本章假设你已经成功地做完了第 2 章的所有步骤。如果你正在启动一个新的系统，那么就不要从这里开始，要从前面更小更精确的步骤开始。本章假设你已经有了扎实的基础知识，并且有了一个可以开始建立真正启动代码的硬件设备。

3.1　一个嵌入式系统启动平台

启动监控器是指运行于计算机系统的一段代码。这段代码是常驻于硬件平台框架最低层的。你可以在各种各样的计算机系统中找到很多类型的启动监控器，当然不限于嵌入式系统。一个典型的启动监控器可给系统提供一些基本的启动功能，还能为用户提供一定的保障，因为就算其他所有的功能都丧失了（指上层的操作系统），系统仍然可以退回到启动监控器。一般而言，启动监控器的建立是为了给系统诊断和启动提供一套基本的功能，一旦系统已经启动，启动监控器的使命也就完成了。从这一点来说，本章将要介绍的嵌入式系统启动平台是一般启动监控器的一个超集。

嵌入式系统启动平台和启动监控器一样都是一个常驻系统的环境，可提供一些功能用于增强系统的开发过程。简单来说，利用这个平台可以加速一些早期阶段的嵌入式系统的开发。更具体地说，这个平台可以提供一个简单的闪存文件系统、转移应用程序用的普通文件传输协议（TFTP）/Xmodem 接口以及其他一些增强开发环境的命令和功能。作为一个开发体系的一部分，嵌入式系统启动平台提供了建立一个项目的基础，是应用程序本身的一个内部部分，给系统提供了一些核心的特征。不管在何种操作系统下，这些特征都可以被应用程序使用。有了这些定义，我们可以说微型监控器就是一个嵌入式系统的启动平台。

微型监控器是一个框架，在 CPU 复位或上电后就会立即执行，常驻于目标系统的非易失性的闪存之中。启动系统之后，我们的责任是能够让用户通过某种接口（通常是 RS-232 或是以太网）进入系统。该接口提供的功能取决于平台在建立的时候所配置的功能。当微型监控器完成一些初始化的工作以后，就将自己也作为一

个命令行接口提供给用户，通常是通过 RS-232 串口。如果硬件包含以太网接口，则开发者也可以通过用户数据包协议（UDP）与微型监控器通信。

命令集包括一些基本的功能，比如内存读写、并行 I/O 控制和命令行编辑/历史记录。微型监控器还配置了一部分闪存作为文件系统（TFS 将在第 7 章讨论）。文件系统的存在意味着代码可以通过名字而不是地址对闪存进行读写。增加文件系统给嵌入式系统增添了一套新的功能，这就是将微型监控器从其他启动监控器中分离出来的原因。文件系统的一个功能是允许系统和它的软件动态重新配置而不需要重新编译。比如说，如果文件系统支持“可自动启动”属性，那么监控器在启动的时候就会寻找所有有此标记的文件并自动运行它们。这些文件可能是编译过的可执行二进制文件，或是配置文件用于配置系统运行动态主机配置协议（DHTP）或是引导协议（BOOTP）。

你可以配置多个文件自动启动，微型监控器会按照字母的顺序自动执行这些文件。另外，TFS 和微型监控器的命令行接口（CLI）共享用户层次的概念。当系统第一次启动的时候，它在最高的用户层次运行（类似 UNIX 的超级用户或是 Windows 管理员）。当控制权转移到各自的 TFS 文件时，就可以调整用户层次，使系统（文件和命令）的一部分只能开放给一定的用户层。用户层次由存储在 TFS 文件中的密码控制，从系统的 MAC 地址可以得到一个优先密码。

在这个设计中，除了监控器本身以外都是一个文件——既使是监控器调用的主程序也是 TFS 中的一个文件。当程序运行的时候（监控器将程序从 TFS 闪存中读入 DRAM），其他文件可被运行的程序读写。这样你就可以建立一个二进制的文件来配置它自身，该文件基于几个在 TFS 闪存中的本地可编辑的 ASCII 文件。另外，因为程序在 DRAM 以外也可以执行（只要是在 TFS 中的闪存以外），所以当 DRAM 中程序还在执行的时候，一个新的程序即可被读入闪存中执行。

正如你所见的那样，微型监控器与 TFS 有着很紧密的联系。因此，许多 CLI 命令集命令都假设文件系统是存在的。Xmodem 和 TFTP 都是面向文件系统的，因此上传或下载都是基于名字而不是地址的。这个设计带来了很多方便，管理常驻平台顶层的程序也非常容易了。就基本环境而言，微型监控器使嵌入式系统看起来像是 PC。PC 提供了一套应用程序可以利用其核心功能，即文件系统、BIOS 和标准 I/O。微型监控器允许系统作为一个平台提供一些类似的功能。

3.1.1 常驻系统命令集

下面列出了 CLI 中的所有微型监控器命令。详细的说明请参考 CD-ROM。

argv	建立 argv 列表
arp	地址解析协议（ARP）
call	调用嵌入功能
cast	在内存中定义一个跨数据结构
cm	内存复制
dhcp	发现 DHCP/BOOTP
dis	释放内存
dm	显示内存内容
echo	向控制台输出字符串
edit	编辑文件或缓冲区
etest	异常处理操作验证测试
ether	以太网接口操作
exit	终止一个脚本
flash	闪存操作
fm	填充内存
gosub	在脚本中调用一个子程序
goto	跳转
heap	堆操作
help	显示命令帮助
history	显示历史命令
icmp	网间控制报文协议（ICMP）接口操作
idev	设备接口操作
if	条件转移
item	从列表中提取项目
let	让变量等于表达式的值
mstat	监控状态
mt	内存测试
pm	调入内存
read	交互式变量入口
reset	重启
return	子程序返回
set	变量操作
sleep	秒（或毫秒）延时
sm	内存查找

strace	堆栈跟踪
tftp	TFTP 客户端/服务器操作
tfs	TFS 操作
ulvl	显示或改变当前的用户等级
unzip	解压
xmodem	Xmodem 文件传输
version	版本信息

3.1.2 给应用程序提供的 API

以下是微型监控器给应用程序提供的 API 函数。详细说明请参考 CD-ROM。

mon_addcommand（）	给监控器增加一条命令
mon_appexit（）	应用程序终止时调用
mon_com（）	监控器和应用程序之间基本连接
mon_cprintf（）	小 printf（）
mon_crc32（）	crc32 函数
mon_decompress（）	解压数据
mon_docommand（）	调用监控器中的一条命令
mon_free（）	释放堆
mon_getargv（）	从监控器中返回当前参数列表
mon_getbytes（）	从控制端返回字符
mon_getchar（）	从控制端返回一个字符
mon_getenv（）	从指定变量中返回值
mon_getline（）	从控制端返回一行字符
mon_gotachar（）	从控制台返回字符指针
mon_intsoff（）	关闭中断
mon_intsrestore（）	恢复中断状态
mon_malloc（）	从堆中分配内存
mon_pioclr（）	清除指定 PIO 脚状态
mon_pioget（）	返回指定 PIO 脚状态
mon_pioset（）	设置指定 PIO 脚状态
mon_printf（）	小 printf（无浮点数）
mon_putchar（）	将字符输出到控制台
mon_restart（）	重新开始

mon_setenv（）	建立新的环境变量
mon_setUserLevel（）	建立新的用户等级
mon_sprintf（）	小 printf（无浮点数）
mon_tfsadd（）	向 TFS 增加一个文件
mon_tfseof（）	从打开的文件中返回 EOF
mon_tfsinit（）	初始化闪存
mon_tfsclose（）	关闭 TFS 文件
mon_tfsctrl（）	在 TFS 文件中执行不同控制函数
mon_tfsfstat（）	返回文件头的一个本地备份
mon_tfsgetline（）	返回一个打开的 ASCII 文件的下一行
mon_tfsipmod（）	在适当的位置修改 TFS 文件
mon_tfsnext（）	下一个 TFS 文件头列表
mon_tfsopen（）	打开 TFS 文件
mon_tfsread（）	读 TFS 文件
mon_tfsrun（）	运行 TFS 中的一些可执行文件
mon_tfsseek（）	在 TFS 文件中搜索
mon_tfstruncate（）	当前打开的可写文件大小截尾
mon_tfsunlink（）	从 TFS 中删除文件
mon_tfswrite（）	写 TFS 文件
mon_xcrc16（）	crc16 文件

3.1.3　基于主机的命令集

以下是 PC 的命令列表，也是微型监控器包的一部分。详细说明请参考 CD-ROM。

aout	AOUT 格式文件接口工具
bin2srec	二进制到 S-记录转换
coff	COFF 格式文件接口工具
com	基本的串行通信工具
defdate	产生日期/时间字符串
dhcpsrvr	单线程 dhcp/bootp 服务器
elf	ELF 格式文件接口工具
fcmp	文件比较工具
f2mem	文件内存转换
maccrypt	建立后门密码的工具

moncmd	UDP 通信接口
monsym	符号表格转换
newmon	更新启动闪存的工具
title	在 Win32 控制台窗口的标题栏输出文字
ttftp	单线程 TFTP 客户/服务器
whence	显示执行路径

如果一台 PC 只是运行 DOS，那是没有什么用处的，除非你在 DOS 的平台上运行其他的应用程序，而这些应用程序又总是假设了一些基本功能的存在，且基本功能均由 BIOS 提供。对于微型监控器来说，这些基本功能通过钩子允许应用程序与监控器直接接口。微型监控器通过 API 函数给应用程序提供了一些基本的便利条件，但是它并没有阻止（事实上它鼓励）应用程序与硬件的直接联系（参见图 3.1）。监控器不只是一个独立地运行于系统中的程序，还是其他应用程序能够被下载到系统上并执行的基础。试想，为了第一次建立监控器，你不得不在其他设备上将启动程序写到闪存上。一旦监控器稳定工作了，你就可以利用它修改自身或是下载其他应用程序，再也不需要器件编程器了。

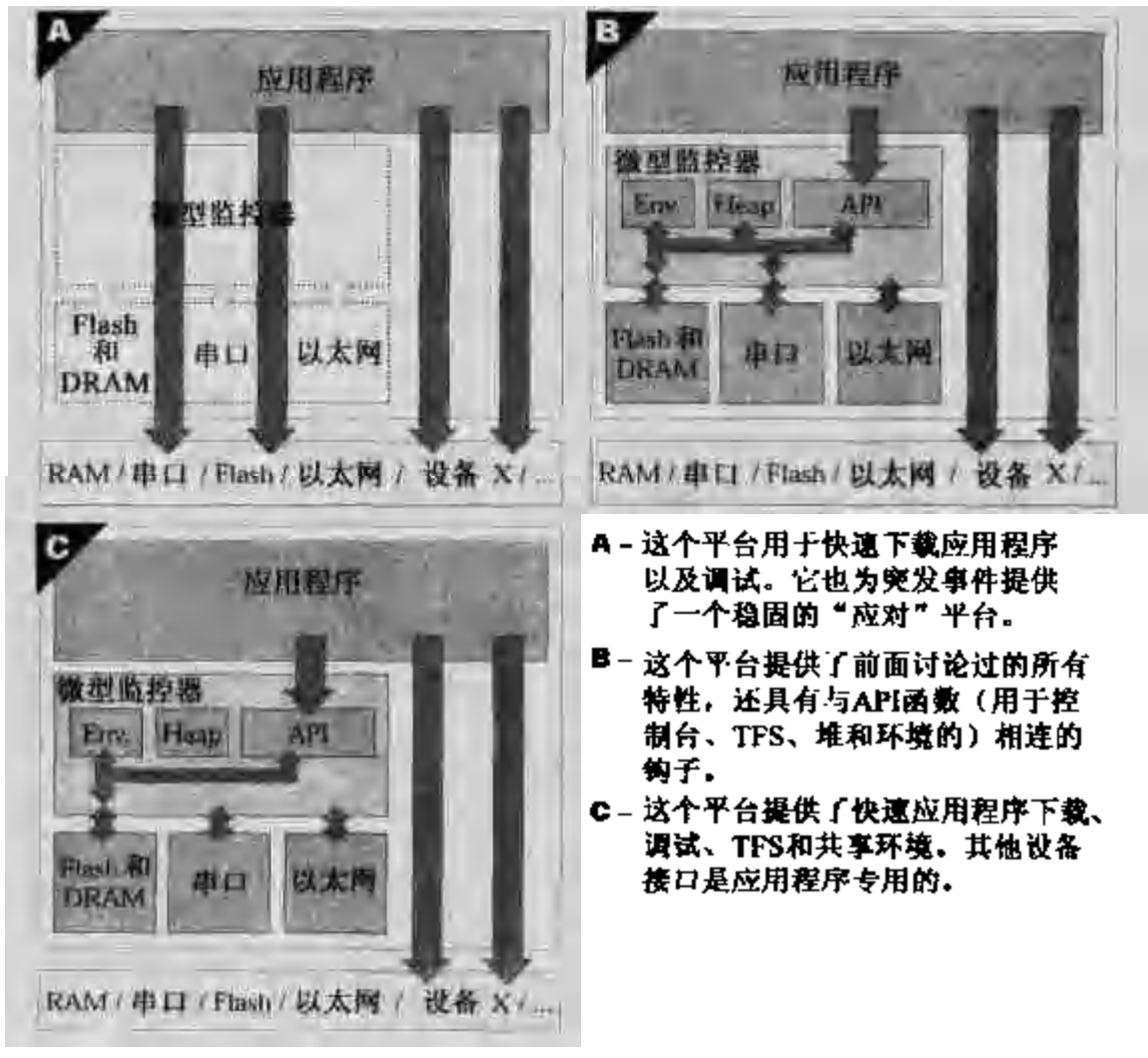


图 3.1　应用程序和微型监控器

运行在微型监控器上的应用程序可以选择绕过微型监控器（A），而仅仅依靠微型监控器的核心服务（B），或仅使用微型监控器服务（C）的一个子集。

3.2　小结

微型监控器提供了可以开发各种不同类型应用程序的环境，实现的简易性可以使其简单地从一个系统转移到另一个系统，不阻止对于基本的 RTOS 和闪存文件系统（FFS）的使用，为不同项目的开发提供了一个通用的基础——不论硬件以下或是 RTOS 以上有什么不同。该通用的基础立即提供开发者一个闪存文件系统、快速应用程序下载、便利的升级路径和其他更多的条件。像其他监控器一样，微型监控器也是常驻在启动闪存中，并且提供一些系统特定功能，允许你能够对硬件进行访问，同时微型监控器还提供给实时应用程序一个有价值的环境。

第 4 章　所需的汇编语言

本章将讲述微型监控器是怎样处理基本的系统启动事务的。在这里，你将了解到系统如何与程序执行相关的部分，比如堆栈、异常处理和重要的状态信息初始化。

完成启动的代码是非常重要的。因为一旦有什么地方不对，则所有的地方就都不会正常工作了，而且最关键的是这些代码的硬件相互独立，因此要想让监控器正常工作，你必须做到以下几点：

- 采用系统的存储器映射模式；
- 识别出冷启动和热启动；
- 掌握系统异常处理；
- 学会保存处理器的全部状态；
- 正确地禁止任何缓存的使用；
- 初始化并正确操作通信端口。

这些启动代码包含了很多与特定硬件相对应的细节——除非你对他们的处理是完全正确的，否则你是不会用工作的设备来进行任何基于代码的调试的。虽然如此，还是有很多结构可以用相对硬件无关的 C 语言来编写。本章将详细地说明这部分结构的高层次部分，并且解释一些顶层硬件支持代码应该考虑的问题，顺便也会指出一些更为重要的设计上的优劣，以便当你编写自己项目的代码时考虑。

4.1　复位之后

复位之后立刻执行的代码必须是用汇编语言编写的。当按下电源或是固件触发重启的时候，这些代码会得以执行（这两种重启方式分别叫做冷启动和热启动，两者完成的功能很相似）。不管采用哪种方式重启，系统都认为是经历了一个重新上电复位的过程。热启动时，执行的代码应该完成所有在上电复位时 CPU 固有的工作（见程序清单 4.1）。但是稍后执行的代码应该能通过某种方式知道到底是哪种启动方式，以便决定是否需要初始化监控器的全局变量空间，是否运行一些高层次的系统启动程序等等。下面是重启时基本的初始化列表：

- CPU 禁止中断（不是外设而是 CPU）；

- 禁用刷新缓存，使之无效；
- 允许按所需的速度和地址读取启动闪存；
- 允许按所需的速度和地址读取系统 RAM/DRAM；
- 初始化堆栈指针。

程序清单 4.1 给出了 reset.s 的伪代码。

程序清单 4.1　reset.s 的伪代码

```
        .file       "reset.s"


        .extern start, moncom
        .global warmstart, moncomptr


    coldstart:
        Initialize "something" to store away a state variable.
        StateOfTarget = INITIALIZE
        JumpTo continuestart

    moncomptr:
        .long moncom


    warmstart:
        /* Load into StateOfTarget the parameter passed to warmstart
         * as if it was the C function:
         * warmstart(unsigned long state).
         */


    continuestart:
        Disable interrupts
        Flush/invalidate/disable cache
        Adjust boot device access
        Adjust system ram access
        Establish the stack pointer
        JumpTo start()
```

注意 如果你不习惯看汇编语言

- 以冒号结束的字符串（如 coldstart:，moncomptr:，warmstart:，continustart:）是标识符或变量名。
- 以点开头的字符串（如.file，.extern，.global）是指令。它告诉汇编编译器做一些事情而不是作为助记符来产生代码。.extern 指令告诉编译器指定的变量在另一个文件中；.global 告诉编译器指定的全局变量可以读取（被其他文件中的其他函数）。

coldstart 标识符应该置于重启后的 CPU 引导地址。现在先不管 moncomptr。Warmstart 在 C 语言代码中被看成是一个函数，把它放在这里就给 C 语言代码一个指向重启固件的指针。

coldstart 处的代码将初始值存储在不会被重启代码中断的位置，因此它是可以在第一个 C 函数 start（）开始的时候恢复的。可恢复初始值允许固件决定在 start（）函数之后作为完全或是部分过程执行。一个完全的启动过程将会初始化所有的目标全局变量和固件子系统。一个部分启动过程（固件触发的热启动）不会重新初始化全局变量，因此状态将会保持而基本的外设会被重新初始化。

注意程序清单 4.1 中 StateOfTarget 变量的使用。根据状态的存储位置，各个系统的实现稍微有些不同。目标的状态可以被存储在内存，外设的可写寄存器或是在不会被冷启动中断的 CPU 寄存器中。

目标执行的第一个 C 语言函数就是 start（）（见程序清单 4.2）。根据系统环境的不同，你可能需要将该函数的第一部分（BSS 初始化）放在之前，用于调用该函数的汇编代码的尾部，将 BSS 初始化代码放在汇编代码的后面，从而可以避免 BSS 的初始化循环进入到它不该进入的地方。BSS 初始化循环是进行基本的内存测试的好地方，用于验证监控器所使用的内存是否正常。一个简单的地址到地址的测试（见程序清单 4.3）可以被插入到 ram 初始化循环的上面。

程序清单 4.2 start()

```
void
start(void)
{
    int argc;
    register unsigned long state;
```

```
volatile register ulong *ramstart, *ramend;

/* Copy StateOfTarget to a register prior to the bss
 *   init loop.
 */
state = StateOfTarget;

/* Initialize monitor-owned ram... Since this loop initializes
 *   all monitor ram, the code within the loop must NOT use any
 *   ram-based variables (ramstart   & ramend are registers).   Also,
 *   since the stack gets cleared during this operation, this
 *   function must never return, and cannot expect non-register
 *   local variables to retain their values until after this loop.
 */
if (state == INITIALIZE) {
     ramstart = (ulong *)&bss_start;
     ramend    = (ulong *)&bss_end;
     while(ramstart < ramend) {
          *ramstart++ = 0;
     }
}

/* Now that bss has been initialized, we can store
 *   StateOfTarget once again in bss space.
 */
StateOfTarget = state;

/* Load bottom of stack with 0xdeaddead for use by stkchk(). */
TargetStack[0] = 0xdeaddead;

/* Initialize vector/exception table. */
vinit();

/* Initialize system devices. */
```

```
        devInit(19200);

        /* Build argument list and go to main(). */
        argc = 0;
        while(argv[argc] != 0) {
            argc++;
        }
        main(argc, argv);

        /* The function main() should not return, but just in case... */
        while(1);
    }
```

程序清单 4.3　测试 RAM

```
    #if INCLUDE_RAMTEST_AT_STARTUP
        ramstart = (ulong *)&bss_start;
        ramend   = (ulong *)&bss_end;
        while(ramstart < ramend) {
            *ramstart = ramstart;
            ramstart++;
        }

        ramstart = (ulong *)&bss_start;
        ramend   = (ulong *)&bss_end;
        while(ramstart < ramend) {
            if (*ramstart != ramstart)
                error!
            ramstart++;
        }
    #endif
```

注意

地址到地址的测试是几个很简单的循环。第一个循环将 0 写入地址 0、4 写入

地址 4 等等。第二个循环重新遍历所有的地址，看看读到的值和地址值是否相等。这个简单但有效的测试可以发现大部分影响地址或是数据总线的常见问题。

另外，这个测试还是一个发现粘连地址线的好办法，比如说，我认为我有 64KB 的 RAM，但是地址线的最高位与低电位粘连了，当循环开始写第二个 32KB 地址时，它会覆盖第一个 32KB 地址。当测试试图验证第一个 32KB 内存的时候就会失败——由此我们可以发现问题的根源。

注意程序清单 4.2 和程序清单 4.3 中 bss_start 和 bss_end 的用法。bss_start 和 bss_end 标签必须在连接—编辑内存映射文件中定义。大部分的工具提供一些固有的标签，这些标签标识各个不同部分的开始和结束。有些工具定义的标签并不常用，可增加一些自己的标签到连接—编辑文件中，以便确切知道这些标签的意义。

注意，start（）函数用于调用 main（）函数的结束。很显然，系统现在可以支持 C 语言代码了，到了 C 层次并不意味着你就可以随心所欲了，还得注意以下几点：

- 堆栈是有限的或至少可能是有限的。要点是你现在处在新标记的空间中，所以不要开始写需要很多堆栈的函数。堆栈的深度和可靠性现在仍然是未知的（你能保证你正确地初始化了堆栈指针吗？）。事实上，你应该让 start（）函数是惟一使用堆栈的函数，其他一切内联函数都使用全局变量。
- 初始化的数据是不可写的。记住，微型监控器设计的一个目标就是尽量使其为编译器/CPU/目标独立，而从 ROM 复制初始化数据到 RAM 的过程对于不同的工具总是各不相同。
- 你仍然没有一个 C 层次的串口。

为了不需要复制任何初始化数据到 RAM 而建立的第一阶段的固件是很容易实现的，消除了处理需用不同工具处理初始化方法的争论。但是，这种方法确实需要在写代码的时候遵守一些规则，因为如果你试图改变初始化数据是不会有任何编译警告的（除非你特别小心地使用常量）。

另外，高层次的函数不使用任何编译器支持的库也是很方便的（除非非得使用）。虽然基本规则可能有点保守，但是对于一个启动平台，最重要的是应把握代码以便使微型监控器使用它自己的 printf（）和 malloc（）等函数。注意，我并不是太极端——我没有建议你写自己的数学原型！

涉及外围芯片时使用的元件编号

如果你写一些与外围设备接口的代码，则应该使用元件的编号而不是一些通用

的名称或代号。如果你看别人写的是你不很熟悉的代码，那么看到这样的代码“触发蝎子芯片的 4 管脚”以接口工作一定很不舒服。什么是“蝎子”芯片呢？“蝎子”芯片就是原理图根据编号显示元件而无需根据代号显示元件，因此“蝎子”芯片只能使用编号。如果你使用通用名称来代替元件编号也会发生类似的问题，例如，代码可能提到“串口”。如果你的系统有不止一个串口设备该怎么办呢？对于原始设计者来说可能是很清楚的，一个端口用于控制，另一个端口用于与外部设备通信。可是对于一个缺少这些信息的使用者来说怎么可能清楚，所以应该在注释中写清楚这些细节。

4.2 I/O 初始化

Start（）函数结束以后，基本的初始化工作就结束了，且系统已经处于 C 层次了。监控器已经初始化了内存和堆栈，下一步就是和外设通信了。首先确保所有的外设都已就位，为此复位并禁止所有 I/O[1]，然后初始化 UART。

UART 应该总是第一个被配置，以便系统可以使用 printf（）函数来向用户报告系统状态。根据“小步幅”准则，一次只做一件事情。UART 是一个查询接口[2]，允许避免复杂的中断处理。记住，我对基础仍然不是很确定，所以需继续很小心地迈每一步。我用 4 个函数来与 UART 接口，即 init（）、putchar（）、getchar（）和 gotachar（）。前两个函数是最重要的。当你能够启动系统，并确实看到从串口输出的字符时，会很有成就感。

UART 函数，即 putchar（）、getchar（）和 gotachar（）是用查询模式写的。程序清单 4.4 是 putchar（）的一种实现。

程序清单 4.4　查询模式的 putchar()

```
int
putchar(int c)
{
    int timeout;

    for(timeout=0;timeout<MAX_WAIT;timeout++) {
```

1. 如果你想在硬件设计的过程中做任何事情，那么都要保证微型监控器具有在不需要重启就能复位各个外围器件的能力。
2. 实际上，监控器几乎不使用中断，这样各个目标的接口就变得容易多了。

```
            if (XMIT_HOLD_EMPTY())
                break;
        }
        if (timeout == MAX_WAIT) {
            ERROR();
        }
        STORE_XMIT_HOLD_REG((char)c);
        return((int)char);
    }
```

注意，等待转换保持缓冲区变空的循环不是无限的。考虑到当第一次建立这个函数时能退出这个循环是很重要的，因为如果这个接口出了什么问题，则错误处理程序就可以让你退出这个循环，而不是一直等在循环中。

4.3　建立异常处理

下一步是初始化 CPU 向量表。CPU 的向量表是一些存储块（在固定位置或是由系统寄存器指针决定）。这些存储块中包含代码或是指向代码的指针。当异常发生的时候，作为一些系统事件的结果，运行代码转入向量的 1 个入口。这个系统事件可能是一个中断、无效指令的执行、一个断点或是一个地址调整错误。这些异常和向量表的实现一样，对于各个 CPU 都不相同。

通常，异常表是以下两种形式之一，即一个函数指针表或是小块代码表。每块代码的目的是执行一些非常基本的任务，然后跳转到一些通用的异常处理函数。从监控器的观点来看，有两件重要的事情要做：首先，你必须创建表；然后，代码必须能处理异常。微监控器将记录所发生的异常类型，并将异常发生时寄存器的状态保存起来。

总而言之，我试图避免将讨论的主题与任何一种特定的处理器相关联，但不得不做一些假设。现以 PowerPC™ 作为例子。PPC 结构定义的异常表是存储块，每个块是典型的（不总是）256 个字节。对于许多实现来说，异常表可以被放置在内存中不同的地方，寄存器的内容告诉 CPU 在内存中的什么地方可找到它们。假设异常表的物理地址是 0x0，第一个入口（块）没有定义，以后的入口地址每次增加 0x100（十进制 256）字节。

4.3.1 ROM 中的异常处理

通常缺省的异常表在启动闪存中，因为启动闪存覆盖的存储空间必须和复位时异常表映射的存储空间相同。复位本身也是异常，因此在启动闪存中必须有复位的异常处理，在上电后的一瞬间至少有一部分异常表已经就位了。异常表中代码与启动闪存中其他代码的惟一区别，就是异常表的入口在固定的位置，而不是由连接器动态决定的。程序清单 4.5 是程序清单 4.1 中伪代码的扩展，现在的代码中包含了一部分异常表、一些 PPC 汇编助记符和一些新的指令。

程序清单 4.5 初始化异常向量

```
        .file           "reset.s"

        .extern start, moncom
        .global warmstart, moncomptr

        .text

        .balign 0x100

coldstart:
        Initialize "something" to store away a state variable.
        StateOfTarget = INITIIALIZE
        JumpTo continuestart

moncomptr:
        .long moncom

        .balign 0x100

vector_type1:
        li   regX, V_TYPE1
        ba   saveregs

        .balign 0x100
```

```
vector_type2:
    li   regX, V_TYPE2
    ba   saveregs

    .balign 0x100

vector_type3:
    li   regX, V_TYPE3
    ba   saveregs

    /* More exception handlers would be here */

warmstart:
    /* Load into StateOfTarget the parameter passed to warmstart
     * as if it was the C function:
     * warmstart(unsigned long state).
     */

continuestart:
        Disable interrupts
        Flush/invalidate/disable cache
        Adjust boot device access
        Adjust system ram access
        Establish the stack pointer
        JumpTo start()
```

程序清单 4.5 中算法非常简单，每 256 个字节的偏移地址（由.balign 直接创建）就包含一个惟一的代码用于异常处理。这种惟一性就是简单地让 regX[1] 寄存器包含一个标签以告诉后面的函数发生了什么异常。每个异常处理函数均需记录异常（在 regX 中）并转向 saveregs（）函数，以保存异常发生时的上下文（或寄存器组）。在异常处理完后，监控器或是重新启动应用程序或是简单地返回到监控器的命令行接口（CLI）。为了让寄存器能跨平台保存，实现函数构造了一个初始化的寄存器表，

1. 尽管我使用 PPC 作为基本的例子，但我并不使用 PPC 寄存器的名称。

如程序清单 4.6 所示。

程序清单 4.6 给寄存器命名

```
char   *regnames[ ] = {
       "RA", "RB", "RC", "RD", "RE", "RF", "RG", "RH",
       "RI", "RJ", "RK", "RL", "RM", "RN", "RO", "RP",
};

#define REGTOT    (sizeof regnames/sizeof(char *))

ulong regtbl[REGTOT];
```

表的名称对于不同 CPU 都不相同，就算是同一个 CPU 族，寄存器组也不相同（现在使用的是一个虚拟的通用寄存器组），除了这个字符串数组，还分配了一个无符号长整型值表，大小和上面的 regnames [ ] 数组中的指针数相等。该数组建立了一个字符串表和一个潜在值表，所有这些都是为了让异常处理能把每个寄存器的值正确地放入适当的位置。例如，寄存器 RA 将被存储在偏移 0 处，RB 将被存储在偏移 1 处等等（注意，这些代码假设一个寄存器的大小和无符号长整型变量相等）。程序清单 4.7 是 PowerPC 利用这些表来存储寄存器的汇编代码。

程序清单 4.7 保存寄存器

```
/* saveregs:
 *   Save register set into regtbl[] array.
 */
saveregs:
        lis     regY,(regtbl)@ha
        addi    regY,regY,(regtbl)@l
        stw     rA,0(regY)
        stw     rB,4(regY)
        stw     rC,8(regY)
        stw     rD,12(regY)
        stw     rE,16(regY)
        stw     rF,20(regY)
        stw     rG,24(regY)
        stw     rH,28(regY)
```

```
stw     rI,32(regY)
stw     rJ,36(regY)
stw     rK,40(regY)
stw     rL,44(regY)
stw     rM,48(regY)
stw     rN,52(regY)
stw     rO,56(regY)
stw     rP,60(regY)

mr      rC,regX         /* parameter to exception (type) */
lis     rP,(exception)@ha
addi    rP,rP,(exception)@l
mtctr   rP
bctr
```

以上代码使用寄存器 regY 作为指向 regtbl 数组的指针。由于每一个寄存器都被保存了，因此寄存器 regY 的偏移是以 4 为单位的（32 位值的字节大小）。在寄存器都被保存后，regX 的值被转移到另一个寄存器。该寄存器被编译器作为第一个函数的参数寄存器（例如，假设第一个函数的参数寄存器是 rC.），下一个被调用的函数接收异常类型作为它的参数，并且基于类型信息就可以执行相应的异常处理事务（见程序清单 4.8）。

程序清单 4.8　复位前的相应异常处理

```
void
exception(int type)
{
    switch (type) {
        case V_TYPE1
            break;
        case V_TYPE2
            break;
        case V_TYPE3
            break;
        default:
            break;
```

```
        }

        ExceptionType = type;
        ExceptionAddr = getreg("SRR");
        warmstart(EXCEPTION);
    }
```

异常处理的最后一步就是以 EXCEPTION 为参数调用 warmstart（）。Warmstart 和复位（或冷启动）的意思几乎一样。Warmstart 的调用可以让系统重复启动过程，但不会造成 BSS 空间的重新初始化。在 StateOfMonitor 被置成 EXCEPTION 的同时，start（）函数被调用（注意，因为现在的状态不是 INITIALIZE，start（）函数中代码不会重新初始化 BSS 空间），然后 main（）函数被调用，监控器可以根据异常类别决定下一步该干什么。

在程序清单 4.9 中，StateOfMonitor 变量包含 EXCEPTION，所以 main（）函数可以基于此执行。通常，如果系统状态不是 INITIALIZE，那么系统只是部分重启，且一些状态被认为已经初始化好了而不会被重新初始化。例如，你在后面将看到监控器的一部分是一个闪存文件系统，被称之为小文件系统（TFS）。当 StateOfMonitor 的值为 INITIALIZE 时，启动的步骤之一就是扫描 TFS 中的所有文件并执行那些可以自动启动的文件。而当值为 EXCEPTION 时，这些文件是不会被执行的。在这种情况下会进行一些重新初始化，打印异常类型和寄存器组，然后自动重新启动应用程序。exceptionAutoRestart（）函数查看一些其他的环境变量以决定是否应该有选择地运行一个脚本或是重新启动应用程序（或是返回到监控器 CLI）。

程序清单 4.9　MicroMonitor 的 main()

```
void
main(int argc,char *argv[])
{
    if (StateOfMonitor == INITIALIZE) {
        /* Do some higher level initialization here */
    }

    switch(StateOfMonitor) {
        case INITIALIZE:
            break;
```

```
        case APP_EXIT:
            reinit();
            printf("\nApplication Exit Status: %d (0x%x)\n",
            AppExitStatus,AppExitStatus);
            break;
        case EXCEPTION:
            reinit();
            printf("\nEXCEPTION (offset 0x%x) occurred near 0x%lx\n\n",
            ExceptionType,ExceptionAddr);
            showregs();
            exceptionAutoRestart(INITIALIZE);
            break;
        default:
            printf("Unexpected monitor state: 0x%x\n",StateOfMonitor);
            break;
    }
```

```
    /* Enter the endless loop of command processing: */
    CommandLoop();
}
```

最后一个注意点是关于异常处理机制。在寄存器的值被保存在 regtbl [ ] 数组中后，这个数组的内容与 regnames [ ] 数组中的内容相等。这样就可以很容易地构造一个函数，使其可根据寄存器的名称返回任何一个寄存器的值。Getreg（）函数（见程序清单 4.10）演示了这种机制。

程序清单 4.10　getreg()

```
int
getreg(char *name, ulong *value)
{
    int     i;
    char    *p;

    p = name;
    while(*p) {
```

```
            *p = toupper(*p);
            p++;
        }
        if (!strcmp(name,"SP")) {
            name="R1";
        }

        for(i=0;i<REGTOT;i++) {
            if (!strcmp(name,regnames[i])) {
                *value = regtbl[i];
                return(0);
            }
        }
        printf("getreg(%s) failed\n",name);
        return(-1);
    }
```

getreg（）函数以寄存器的名称和一个无符号长整型指针为输入，简单地遍历 regnames［］表直到找到所需的名称。匹配的序号作为 regtbl［］的一个偏移量。这个值由长整型指针参数返回。

4.3.2 RAM 中的异常处理

这个实现函数将所有的异常处理置于 ROM 中，或是非易失性的空间中。这种方式对于复位异常处理非常方便，因为复位时你需要异常处理就位，但是其他异常处理只有在对应的异常发生时才需要被调用，把这些异常向量表置于可写的存储空间中会有很多优势。当监控器已经启动并且一个应用程序已经被执行时，这个应用程序可能会有其他处理异常的方式，因此如果异常表可以被更改则会更好一些。另外，在一些 CPU 中，异常表的空间可能超过 8KB，这对于闪存有限的系统来说可能会溢出。

如果向量表最终将常驻 RAM，则必须是由 100%的可浮动代码写的。因为向量表将从闪存中被复制到 RAM 中，只要对先前的异常处理做很小的修改就可以很容易地将向量表转换成可浮动代码（见程序清单 4.11）。

程序清单 4.11　安装向量列表

```
/* This function is copied into the vector table DRAM by vinit().
 * Multiple copies of general_vector are made, with the value
 * loaded into regX being modified for each vector.
 */
general_vector:
    li      regX,0x1234     /* '0x1234' is modified by copyGeneralVector(). */
    lis     regZ,(saveregs)@ha
    addi    regZ,regZ,(saveregs)@l
    mtctr   regZ
    bctr
```

注意，在这里使用的是间接分支而不是与 PC 相关的分支指令。这种变化使得分支操作是可浮动的，所以现在要做的就是将分支代码复制到最终将存储异常表的 RAM 空间（见程序清单 4.12）。

程序清单 4.12　复制到 RAM

```
void
copyGeneralVector(ushort *to, ushort vid)
{
    extern  ulong general_vector;

    memcpy((char *)to,(char *)&general_vector,20);
    (to+1) = vid;               /* Modify the vector ID value */
}

void
vinit()
{
    char    *base;
    int offset;

    base = RAM_VTABLE;
    for(offset = 0x00; offset < 0x1000; offset += 0x100) {
        copyGeneralVector((ushort *)(base + offset),offset);
```

```
        }

        asm("    sync");              /* Wait for writes to complete */
        putevpr(RAM_VTABLE);          /* Set vector table pointer to RAM */
    }
```

汇编语言复制函数的大小是 20 个字节（5 个指令），复制分支代码（就像它在数据区）到新的基于 RAM 的向量列表中，但该方法必须基于知道在这点上的缓冲区配置的基础上，否则指令和数据缓冲区都应该禁止使用。这个方法只使用闪存的 20 个字节来创造一个超过 8KB 的向量列表。

4.3.3 当心寄存器

最后强调的是应该了解寄存器在使用时不受控制。寄存器在每次执行时都会有些不同，但总的来说，至少会有一个寄存器可用于异常程序处理。你可以假定寄存器没有被应用程序的其他部分使用，那么对于监控器来说，异常处理只是简单地记录异常类型并储存所有的寄存器。代码假定只有少量的寄存器可用于异常处理过程，所以必须确认这种假定，不同的 CPU/RTOS/编译器组合其细节也不同。一般来说，替代可把寄存器缓存到全局列表中，实时异常处理器（也就是与设备驱动相关）把受影响的寄存器装入中断任务堆栈。

4.4 小结

你最先的设计可以是相当简单的——也应该是。我在以前强调过，在早期你应该采取小步骤前进。代码的模块化有利于代码的增长。启动代码的第一次实现应该尽可能地简单，禁止异常处理并滤掉异常处理器，禁止缓冲器和内存管理单元，禁止除了通信设备以外的外围设备，只使用最简单的通信信道——推荐串口。一旦实现了简单的启动，你就能得到更复杂的版本。再一次强调，微型监控器的模块化结构有助于程序的修订。

如果你小心地走每一步，而每一步只实现最基本的功能，那你就会发现构造微型监控器将是一条宽广的大道。

第 5 章　命令行接口

第 4 章的内容抛开了控件和串行接口函数，通过热启动为微监控器组建了一个单一的系统。而本章将通过控制端口实现用户接口。最近几年，我已经把命令行接口重编了 6 次，而现在这个版本比较可靠，我在短时间内不会再改写它。

设计命令行接口的目的是在不增加复杂度的情况下，提供更多的灵活度。该命令行有以下几种功能，即核心变量和符号、命令行编辑和记录、命令输出转移、用户分级及密码保护。如果你不想要全部的功能，则可以通过在主监控器的配置文件中设置一些宏，选择你需要的功能。为了实现这种配置而又不强迫改变#ifdefs，我调整了一些功能并增加了少量限制，最终实现了一个健全而又不是非常复杂的命令行管理器。

5.1　命令行接口的特点

在本章中，命令行接口可提供给微监控器的功能有以下几个特点：

- 函数命令执行——除了记录和编辑，所有命令行的处理都是由一个单独的函数 docommand（）来实现的。Docommand（）函数是命令行与实际程序之间的通道。微监控器的其他组件（不仅仅是控制台连接）都可以在规则内利用 docommand（）来调用其他命令。
- 命令行标记——在系统内，每条命令都有一个标准原型 cmdfunc(int argc, char *argv[ ])。命令行可以输入字符串参数，这样可以使之完成特殊命令功能。
- 核心变量和符号——命令行处理器能为核心变量和符号提供替代。在命令行上，可以用“$”作为前缀来查找核心变量。命令行接口查找美元标志后面的字符串，看在微处理器的环境下是否有与之匹配的核心变量。如果字符串匹配核心变量，则替代就发生了；如果不匹配，则字符串将不改变。而以符号“%”作为前缀将能更加精确地找到字符串。
- 命令行编辑和记录——当要修改已输入的一行字符时，命令行接口提供一种比用退格键退回原处重新输入更有效率的方式。命令记录允许使用者重新得到前面输入的命令，可以再次执行或修改一部分从而得到一个新的命令。
- 命令输出转移——命令行接口通过使用 printf（）和 putchar（）向使用者传送专用命令信息，而输出转移则使显示的数据转传至 RAM，最终再从 RAM 传到闪存文件系统内的文件里。

- 用户分级和密码保护——当命令行接口接收到每一个命令时，都要判断用户的等级和执行该命令所需要的等级。如果用户的等级比所需要的等级高，则命令执行；否则，命令将被命令行使用界面拒绝执行。
- 脚本执行——如果一个命令在命令表内没有找到，则作为最后手段，命令行接口将把命令传给 TFS 作为可执行脚本运行（如果存在该名字的脚本）。
- 强制执行——微监控器的应用程序既可以在其自身环境下运行，也可以在 DOS 环境下运行。每一个应用程序基本上都有自己的命令集，命令在监控器内比在应用程序内有用得多（因为监控器的应用程序接口允许应用程序的命令行接口访问监控器的命令行接口内的命令）。为了使应用程序和微监控器能同时识别相同的命令串（“help”是最常见的例子），微监控器的命令行接口需发出一个强制执行命令。因此，字符串_help 能被传到应用程序中，而这个应用程序不能在监控器命令行接口运行，而是在 help 下运行。

5.2 命令行接口的数据结构和命令列表

在微监控器内，每个基本命令都有如程序清单 5.1 所示的数据结构。命令数据结构存储在命令列表里。

程序清单 5.1 命令行接口的数据结构

```
struct  monCommand {
    char      *name;        /* Name of command seen by user. */
    int       (*func)();    /* Function called when command is invoked. */
    char      **helptxt;    /* Help text. */
};
```

第一项 name 指向一个字符串，该字符串必须与命令行上第一个向量相同。函数指针 func 指向命令名匹配的结果。helptxt 指针指向该命令的帮助文档。

在命令表里，预处理器#include 使你能增加一些有特殊目的的命令。这些命令独立于基本命令[1]。命令行接口处理器查看数据结构表，找出命令行中与 name 项相匹配的象征量。程序清单 5.2 是一个简单的命令列表。

程序清单 5.2 命令列表

```
extern  int       Add(), Echo(), Help(), Version();
```

1. 我们在讨论用一个启动监控器来支持用于不同目标特点要求的扩展，这包括允许不实际接触监控器的实际命令列表而添加扩展。

```
extern   char     *AddHelp[ ], *EchoHelp[ ], *HelpHelp[ ];
extern   char     *VersionHelp[ ];

struct monCommand cmdlist[ ] = {
     { "add",          Add,          AddHelp, },
     { "echo",         Echo,         EchoHelp, },
     { "help",         Help,         HelpHelp, },
     { "?",            Help,         HelpHelp, },
     { "version",      Version,      VersionHelp, },
#include "xcmdtbl.h"
     { 0,0,0, },
};
```

5.3　命令行接口处理

docommand（）函数，即命令行处理器，可提供命令行串和冗长等级。微监控器通过选择冗长等级调试源程序。冗长等级分为三级：

- 零冗长；
- 当该行命令传给 docommand（）函数时，在下一行输出该命令；
- 在内部变量和符号转换后输出该行命令。

在 docommand（）函数内，全部命令行处理的优先级均高于一般命令函数。命令行处理包括：

- 为将要运行的命令做一个备份；
- 扩展内部变量和符号；
- 将字符串分解为各自独立的受限令牌；
- 为命令行重新定向做准备；
- 确认在命令表内，第一个令牌是有效的命令；
- 提供用户等级确认。

5.4　命令名下的函数

docommand（）函数管理可传递令牌、核心变量和符号，分析当前命令行，并

把它转换为一组令牌，然后传给命令函数进行基本指令处理。与通过 int argc 和 char *argv[ ] 给 main（）函数传递参数的过程一样，每个命令函数可接到令牌表和自变量。而该命令行函数的返回值用来表示是否成功处理命令行。命令行接口界面定义三种不同的返回值：

CMD_SUCCESS　通知 docommand（）函数命令功能成功完成。

CMD_FAILURE　通知 docommand（）函数命令功能没有完成，但并不需要 docommand（）函数输出任何报错信息，因为基本命令函数已经输出任何有关的信息。

CMD_PARAM_ERROR　通知 docommand（）函数基本命令函数没有收到有效的自变量集，docommand（）函数输出语法错误信息。

当命令函数返回值为 CMD_SUCCESS 或 CMD_FAILURE 时，除了记录命令执行失败的信息外，命令函数不再做任何事。当命令函数返回值为 CMD_PARAM_ERROR 时，docommand（）函数输出语法错误信息，从而认定命令帮助文档的第二个字符串为使用信息并输出该字符串（见程序清单 5.3 Add_help[ ]数组），命令使用信息可提供给用户将要使用函数的正确语法格式。

每个命令帮助文档都要遵循一定的语法格式：第一个字符串是命令的主要功能描述；第二个字符串是使用信息，它有公共的格式：大括号{ }表示需要的自变量，中括号[ ]表示可选择的自变量；剩下的一行是特定的命令；NULL 指针结束该列表。

这些文档有不同的帮助功能——一些由 docommand（）提供，另一些由 help 命令提供。通过遍历命令列表，help 命令以表格的形式列出每个命令。换句话说，每条命令可按照命令帮助文档第一行的描述输出。得到命令名后，help 就可以为该命令输出完整的帮助文档。所有命令文档以同一种方式产生，而不是单独由各自的函数产生。在帮助文档列表内的头两个字符串有特殊的含义，该列表的结束标志为 NULL。

程序清单 5.3 是 add 命令的帮助文档函数代码。该命令接收到两个参数并打印结果。为了便于说明，我设置两个参数必须大于零。当加上–v 参数时，结果将放在内部变量而不输出。

程序清单 5.3　add 命令的帮助文档和函数代码

```
char *Add_Help[] =
{
    "Add two numbers together",
    "–[v:] {num1} {num2}",
    "Options:",
```

```
        " -v {varname}      put result in shell var 'varname'",
        (char *)0,
};

int
Add(int argc, char *argv[])
{
        char *varname;
        int opt, val1, val2, result;

        varname = (char *)0;
        while((opt = getopt(argc,argv,"v:")) != -1)
        {
                switch(opt)
                {
                case 'v':
                        varname = optarg;
                        break;
                default:
                        return(CMD_FAILURE);
                }
        }

        if (argc != optind+2)
        {
                return(CMD_PARAM_ERROR);
        }

        val1 = atoi(argv[optind]);
        val2 = atoi(argv[optind+1]);

        if ((val1 <= 0) || (val2 <= 0))
        {
                printf("Argument out of range\n");
```

```
            return(CMD_FAILURE);
        }

        result = val1 + val2;
        if (varname != (char *)0)
            shell_sprintf(varname,"%d",result);
        else
            printf("%d + %d = %d\n",val1,val2,result);

        return(CMD_SUCCESS);
    }
```

注意，add 函数的返回值可能出现上述三种情况。当从 getopt（）[1] 函数的返回值为零时（说明调用异常处理），则 add 函数返回值为 CMD_FAILURE。由于 getopt（）函数输出产生错误的信息，因此不需要 add 函数输出额外的信息。当在执行的过程中没有输入两个参量时，则 add 函数的返回值为 CMD_PARAM_ERROR，同时 docommand（）函数把语法错误信息加在 Add_Help [] 数组中的使用文档一项。当两个参量同时大于零时，add 函数的返回值也有可能是 CMD_FAILURE。因为 add 函数告诉用户某个地方有错误的信息，而 docommand（）函数不仅能输出错误信息，且能继续监视错误。当命令成功执行，add 函数的返回值为 CMD_SUCCESS 时，docommand（）函数执行完毕。命令函数本身并不输出任何出错信息，而是由 docommand（）函数来处理。

当 getopt（）运行结束，optind 的值将被记录在变量表内。微监控器的命令能使 getopt（）不仅处理其他命令选项，而且能重新定义函数的参数表。由于 getopt（）使用静态变量记录目前运行命令的行数，所以 docommand（）必须在控制命令函数前，先初始化静态变量。

程序清单 5.4 显示 docommand( )如何分解命令列表及如何使用帮助数组格式。

程序清单 5.4　分解命令列表

```
int
docommand(char *cmdline, int verbose)
{
```

1. getope( )功能通常在 UNIX 程序中用于提供一个普通方式来处理命令选择（可选择的单字节变量被假设跟在一个短线之后并可能需要他们自己的变量）。

```
int ret, argc;
char *argv[ARGCNT];
struct monCommand *cmdptr;

    ...

/* Step through the command table looking for a match between
 * the first token of the command line and a command name.
 */
cmdptr = cmdlist;
while(cmdptr->name) {
    if (strcmp(argv[0],cmdptr->name) == 0)
        break;
    cmdptr++;
}
if (cmdptr->name) {
    ret = cmdptr->func(argc,argv);

    /* If command returns parameter error, then print the second
     * string in that commands help text as the usage text.  If
     * the second string is an empty string, then print a generic
     * "no arguments" string as the usage message.
     */
    if (ret == CMD_PARAM_ERROR) {
        char *usage;

        usage = cmdptr->helptxt[1];
        printf("Command line error...\n");
            printf("Usage: %s %s\n",
                    cmdptr->name,*usage ? usage : "(no args/opts)");
    }
    return(ret);
}
/* If command is not in command table, then see if it is in
 * the file system.
```

```
        */
        if (tfsstat(argv[0])) {
            int err;
            err = tfsrun(argv,verbose);
            if (err != TFS_OKAY) {
                printf("%s: %s\n",argv[0], (char *)tfsctrl(TFS_ERRMSG,err,0));
            }
            return(CMD_SUCCESS);
        }
        else
            printf("Command not found: %s\n",argv[0]);

        return(CMD_NOT_FOUND);
    }
```

为了方便讨论，程序清单 5.4 只列出 docommand（）函数的一部分而不是全部。代码从 char *argv［］得到执行命令。Argv［0］是待执行命令的名字，程序从命令列表中寻找出与之匹配的命令。如果没有找到，则 name 指针将为 NULL，表明输入的命令无效。如果 name 不为 NULL，则程序将认为命令参数与列表中的命令匹配。接着，函数将执行该命令，按照函数的返回值决定 docommand（）是否需要输出附加的错误信息。再一次强调，基本命令执行时不处理任何一般类错误信息，代码只是简单地把 CMD_PARAM_ERROR 传给 docommand（）和输出基本信息。

注意

当程序发现 argv［0］和列表中的命令不匹配时，另外一个命令将执行，目的是查看 argv［0］是否为闪存文件系统的脚本。闪存文件系统的脚本将在第 6 章、第 7 章和第 8 章中再谈。

5.5 内部变量和符号处理

微处理器的命令行接口提供两种相似的代号替换，即内部变量和符号。内部变量（前面加“$”）从 RAM 得到所要替换的数据。符号（前面加“%”）从文件中得

到所要替换的数据。从整体上讲，内部变量和符号以同一种方式操作，只是在微监控器为符号查找文件和为内部变量查找 RAM 时才稍有不同。内部变量更方便，不需要文件系统。但是，由于 RAM 的空间有限和不能断电保存，因此大量内部变量的存储几乎是不可能的。换句话说，如果 TFS 在监控器的结构内，则储存符号的文件能被下载到 TFS 内，系统才可以处理大量寄存在闪存中的符号。由于利用微监控器的符号列表特性，储存实际应用的符号-地址转换表，因此当不熟悉符号调试器时，可把符号-地址转换表传入符号列表，将会使调试更加方便。

程序清单 5.5 是 docommand（）函数初始化替代过程的代码。

expandshell vars（）函数实现替代过程（见程序清单 5.6）。expandshell vars（）函数把字符串转换为内部变量，可提供$VARNAME 和${VARNAME}两种形式，支持内部变量嵌入到另一个内部变量，如可将${VAR${NAME}}扩展为两个过程，即首先是${NAME}，然后是${VARXXX}（XXX 替代${NAME}变量）。

程序清单 5.5　扩展内部变量和符号

```
/* If there are any instances of a dollar or percent sign within the
 * command line, then expand any shell variables (or symbols) that may
 * be present.
 */
if (strpbrk(cmdcpy,"$%")) {
    if (expandshellvars(cmdcpy) < 0) {
        return(CMD_LINE_ERROR);
    }
}
```

程序清单 5.6　expandshellvars()

```
int
expandshellvars(char *newstring)
{
    char    *cp;
    int     result, cno, ndp;

    /* Verify that there is a balanced set of braces in the incoming
     * string...
     */
    ndp = 0;
```

```
if (braceimbalance(newstring,&cno,&ndp)) {
        printf("Brace imbalance @ %d%s.\n",
                cno,ndp ? " ({ missing $ or %)" : "");
        return(-1);
}

/* Process the variable names within braces... */
while((result = processbraces(newstring)) == 1);
if (result == -1)
        return(-1);

/* Process dollar signs (left-most first)...        */
while((result = processprefixes(newstring)) == 1);
if (result == -1)
        return(-1);

        /* Cleanup any remaining "\{", "\}" or "\$" strings... */
        cp = newstring+1;
        while(*cp) {
                if (*cp == '{' || *cp == '}' || *cp == '$' || *cp == '%') {
                        if (*(cp-1) == '\\') {
                                strcpy(cp-1,cp);
                                cp -= 2;
                        }
                }
                cp++;
        }
        return(0);
}
```

程序清单 5.6 中的 braceimbalance（）函数首先被执行，检查命令行的大括号。如果命令行有偶数个大括号，则 braceimbalance（）函数的返回值为零，处理反斜线符号\}或/{可以把装入内部变量的大括号替换为文字括号。如果 braceimbalance（）

返回值不为零，则将输出错误信息并返回“-1”。如果 braceimbalance（）成功执行，则密码的其他部分可以至少在命令行中承担基本主架。如果主架平衡，则 expandshellvars（）函数将调用 processbraces（）函数。该函数为最基本的主架浏览引入字符串，并把这些主架和包含的变量名替代为储存在通信内部的变量值。如果处理完一组主架，则函数的返回值为 1；如果处理完全部主架，则返回值为 0；如果处理过程发生错误，则返回值为-1。

Processprefixes（）函数的执行类似于环路过程，可为内部变量处理“$”，为符号处理“%”并从输入字符串中查找到最后一个“$”（或“%”），从而做内部变量（或符号）代换。如果没有找到前缀（$或%），则 processprefixes（）返回值为 0。

移去程序清单 5.6 末端的 while 循环中所有的反斜线符号，该符号使在其之前的代码不执行，每段函数的完整代码在随书附赠的 CD 中。由于分析每行时均包含基本详情，因此在这里只列出顶层的详情。

5.6　命令行重新定向

微监控器同样支持命令行重新定向。命令行重新定向可以非常方便地把命令输出记录在文件中，增加命令行重新定向是为了把堆栈输出转存到文件中。

来自前线

我比较早地讨论了例外的操作。在接下来的章节里，我将描述一个闪存的文件系统，先是如何转存应用程序堆栈。考虑到你的系统每月都会出现一次异常，那么怎样既不在用户端而又能找出发生异常的原因呢？当异常操作能有选择地执行特定的脚本（从文件系统中）时，这个脚本就可在其他事件中发出堆栈轨迹命令并把输出重新定向为文件。如果这个脚本被激活，则当用户查看任务记录时，可以为轨迹输出文件排列文件系统，且看到引起异常的原因。

所有默认 printf 的输出均通过函数 putchar（），它把每个字符串都送到控制端口。由于 putchar 具有更多的灵活性，因此能把输入的字符传到已知大小的缓冲区内，输出字符就可以记录在 RAM 里。为了使内部记录转换到输出重定向，仅需写一个进程，告诉微监控器把缓冲区的数据写入文件里。传统上讲，命令行重定向的语法为

```
echo hey you >filename
```

因为我要把输出首先传到 RAM，所以该语法对微监控器命令行重定向无效。于是，改为以下两种形式：

- echo hey you > buffer, buffer_size[, filename]

表达式由一个箭形符号、一个或两个逗号及字符串组成。字符串包含缓冲区地址和大小及选定的文件名。当文件名确定后，命令输出拷贝到缓冲区（如果有必要，可取大小为 buffer_size），然后按照文件名传送到 TFS，运行缓冲区指针复位到 buffer 的基地址。如果命令名省略，则命令行输出拷贝到缓冲区，指针指向输入数据的左边位置（假设缓冲区没有满）。

- echo hey you again >>[filename]

表达式常用于把命令输出附加到缓冲区，该缓冲区由“>”符号创建。如果包含文件名，缓冲区的内容将按照指定文件名传送到 TFS，缓冲区指针复位到 buffer 的基地址。如果文件名没有指定，则内容将不会传送到文件，在数据复制之后，指针往后移（再一次强调，假设缓冲区没有满）。

为了实现上述特征，putchar（）函数需要在顶端附加调用（见程序清单 5.7）。

程序清单 5.7　在 putchar( )中加入重定向

```
int
putchar(uchar c)
{
    RedirectCharacter(c);
    if (c == '\n')
        rputchar('\r');
    rputchar(c);
    return((int)c);
}
```

RedirectCharacter（）函数（见程序清单 5.8）查看状态以决定是否执行程序。在不做任何工作的情况下，也可以在检查到缓冲区指针没有到达缓冲区的末端之后，把字符复制到缓冲区并使指针后移。

程序清单 5.8　RedirectCharacter( )

```
void
RedirectCharacter(char c)
{
    if (RedirectState == REDIRECT_ACTIVE) {
```

```
            if (RedirectPtr < RedirectEnd)
                *RedirectPtr++ = c;
        }
    }
```

docommand（）函数修改后，可识别重定向语法（设置 RedirectState 为 REDIRECT_ACTIVE）。当合适时，告诉重定向程序命令执行完毕（设置 RedirectState 为 REDIRECT_IDLE），所以必须在 docommand（）中增加下列代码：

```
if (RedirectionCheck(cmdcpy)== -1)
    return(CMD_LINE_ERROR);
```

在内部变量正在处理时，把上述代码插入 docommand（），可允许用户执行类似于下述的代码：

```
echo hi >$APPRAMBASE,100
```

当调用 RedirectionCheck（）时可以知道，前面描述的命令行接口内部变量处理器把$APPRAMBASE 的内容转换为物理地址。在 docommand（）的末端，需增加另一个调用程序显示命令完成。调用是 RedirectionDone（）是在 docommad（）的返回之前。

RedirectionCheck（）函数（见程序清单 5.9）在查找重定向符号时开始执行。如果在行的末尾，则 RedirectionCheck（）没有找到右箭头符号（>），只是简单地返回。如果找到第二个箭头符号，则 RedirectState 应该被设置为 REDIRECT_ACTIVE，任何其他状态都表明出现错误。如果 RedirectState 是正确的，则 RedirectionCheck（）将查找文件名，并储存以备将来使用。当 RedirectionCheck（）没有找到第二个箭头时，它将处理在箭头后由逗号限制的字符串，并设置相应的指针和计数器。

程序清单 5.9　分析重定向命令

```
#define REDIRECT_UNINITIALIZED   0
#define REDIRECT_ACTIVE          0x12345678
#define REDIRECT_IDLE            0x87654321

static int    RedirectSize, RedirectState;
static char *RedirectBase, *RedirectPtr, *RedirectEnd;
static char RedirectFile[TFSNAMESIZE];
```

```
int
RedirectionCheck(char *cmdcpy)
{
    int inquote;
    char *arrow, *base, *comma, *space;

    base = cmdcpy;
    arrow = (char *)0;

    /* Parse the incoming command line looking for a right arrow.
     * This parsing assumes that there will be no negated arrows
     * (preceding backslash) after a non-negated arrow is detected.
     * Note that a redirection arrow within a double-quote set is
     * ignored.  This allows a shell variable that contains a right arrow
     * to be printed properly if put in double quotes.
     *  For example...
     *  set PROMPT "maint> "
     *  echo $PROMPT      # This will generate a redirection syntax error
     *       echo "$PROMPT"   # This won't.
     */
inquote = 0;

while(*cmdcpy) {
    if ((*cmdcpy == '"') && ((cmdcpy == base) || (*(cmdcpy-1) != '\\'))) {
        inquote = inquote == 1 ? 0 : 1;
        cmdcpy++;
        continue;
    }
    if (inquote == 1) {
        cmdcpy++;
        continue;
    }
    if (*cmdcpy == '>') {
        arrow = cmdcpy;
```

```
            if (*(arrow-1) == '\') {
                strcpy(arrow-1,arrow);
                cmdcpy = arrow+1;
                arrow = (char *)0;
                continue;
            }
            break;
        }
        cmdcpy++;
    }

    if (arrow == (char *)0)
        return(0);

    /* Remove the remaining text from the command line because it is to
     * be used only by the redirection mechanism.
     */
    *arrow = 0;

    /* Now parse the text after the first non-negated arrow. */
    if (*(arrow+1) == '>') {
        if (RedirectState == REDIRECT_UNINITIALIZED) {
            printf("Redirection not initialized\n");
            return(-1);
        }
        arrow += 2;
        while(isspace(*arrow))
            arrow++;
        if (*arrow != 0)
            strncpy(RedirectFile,arrow,TFSNAMESIZE);
    }
    else {
        RedirectPtr = RedirectBase = (char *)strtoul(arrow+1,&comma,0);
        if (*comma == ',') {
```

```
            RedirectSize = (int)strtol(comma+1,&comma,0);
            if (RedirectSize <= 0) {
                printf("Redirection size error: %d\n",RedirectSize);
                return(-1);
            }
            RedirectEnd = RedirectBase + RedirectSize;
            if (*comma == ',') {
                space = strpbrk(comma," \t\r\n");
                if (space)
                    *space = 0;
                strncpy(RedirectFile,comma+1,TFSNAMESIZE);
            }
            else
                RedirectFile[0] = 0;
        }
        else {
            printf("Redirection syntax error\n");
            return(-1);
        }
    }
    RedirectState = REDIRECT_ACTIVE;
    return(0);
}
```

docommand（）函数调用 RedirectionCmdDone（），并告诉重定向程序命令执行完毕（见程序清单 5.10）。

程序清单 5.10　RedirectionCmdDone( )

```
void
RedirectionCmdDone(void)
{
    if (RedirectState != REDIRECT_UNINITIALIZED) {
        RedirectState = REDIRECT_IDLE;
        if (RedirectFile[0]) {
            tfsadd(RedirectFile,0,0,(uchar *)RedirectBase,
```

```
                    (int)(RedirectPtr-RedirectBase));
                RedirectFile[0] = 0;
                RedirectPtr = RedirectBase;
            }
        }
    }
```

当状态已被初始化［RedirectionCheck（）］时，该函数需检查重定向文件名是否存在。如果文件名作为命令的一部分被指定，则重定向文件名将存在。假定重定向文件名已被设置，将利用闪存文件系统调用 tfsadd（）把缓冲区的数据传送给文件。其中，tfsadd（）只是简单地把一块内存的数据传送给闪存文件系统内的文件。

虽然重定向执行时可能比较复杂，但是整个工具可以很简单地与命令行接口结合。注意，使 docommand（）复杂的是调用 RedirectionCheck（）和 RedirectionCmdDone（）函数。命令行重定向的实现是模块化功能的一个很好的例子，整体的复杂度由功能的复杂度决定，复杂度并不会分散在其他组件上。

5.7　命令行编辑和记录

命令行接口记录可以让你重新得到先前输入的命令并再次使用。命令行接口编辑可以通过“ESC”键或“Ctrl”键修改命令行。把命令行接口记录和编辑结合起来，你就有了非常方便的工具并可进行命令行循环使用。当你经常做很多重复性的命令，而这些命令只有细微差别时，命令行循环的使用将会变得非常有用。

来自前线

命令行循环使用是极好的工具，它可使输入命令更快一些。事实上，我们常常花费大量的时间使用“Ctrl”键或“ESC”键，在命令行编辑器上撤消我们输入的错误，从而代替重新输入整行语句。当你输入的命令是 DM 0x14280040 0x40 时，下一个命令就是 DM 0x14290040 0x40，命令行的重新利用将会给你很大帮助。

像基于终端的文件编辑器，命令行编辑器有两种形式，即模态和非模态。模态是输入换码符进入“mode”状态，且重新定义键的使用；然后，输入其他字符中止“mode”状态，并退回到一般字符输入状态。非模态是把特定控制序列结合在编辑

器内，而不用进入到“mode”状态就可以使用这些控制序列。

可选择执行模态命令行接口编辑器实现，因它是有其他内核提供的类似于 vi 编辑器的子集。选择编辑器时，如比较熟悉 UNIX，则可选择 vi 形式的界面。随着知识的积累，则可选择不同工具的结合。目前很多产品并不支持非模态类似 emacs 的版本，而模态更利于与字符重新利用和处理相隔离，因此选择模态近似。

当执行时，函数只需要找到换码符就可重新得到一行命令。在接到字符之后，将转到行编辑器。因此，行编辑器代码与系统的其他部分相隔离，这使得无论加入还是去掉行编辑都非常简单。下面的代码把命令行传入 docommand（）。

```
#define ESC 0x1b

...

#if INCLUDE_LINEEDIT
    if (inchar == ESC) {
        (void)line_edit(inbuf);
        break;
    }
    else
#endif
```

无论何时，只要 inchar 包含换行符，就执行函数 line_edit（），它把命令行输入传到编辑模式。line_edit（）函数使指针指向当前缓冲区。其余的命令行内容将与用户交换信息，编辑模式的参数由 line_edit（）给出。由于当编辑命令行时，所有的字符均重新执行，因此使得 line_edit（）函数本身杂乱，但其复杂度与一般的命令行检索无关。

在访问 line_edit（）函数之后，监控器将处在编辑模式下。虽然在函数内有几种子模式，但这些子模式在编辑行时却必须被执行。用户可以在行内插入字符、删除字符、覆盖字符、移动字符或者可以把目前的命令表移到编辑其中的记录缓冲区。同时，函数必须知道命令表在命令行的位置才可以控制指针左移还是右移。不幸的是，你除非限制函数只能对这些终端起作用，否则不能利用这些方便的终端模式，即使箭头符号的使用可以在行的位置上、下、左、右移动。使用箭头符号可能是不实际的，那是因为一些终端（或终端仿真器）不能映像箭头符号为串口字符串。

程序清单 5.11 是 linedit.c 代码的一部分（请参考在 CD 上完整的文件 linedit.c。完整的命令行编辑器代码的讨论在文档里。我将介绍它的实现过程，但余下的请读

者参考 CD 上的源代码）。当接到要求去编辑命令行时，line_edit（）调用函数 lineeditor（）[1]。此时，编辑器刚刚启动，且处在命令（CMD）模式下，函数把当前位置指针指向换码符刚刚出现的行的位置。一些附加的指针和状态变量也已设置，代码循环处理输入的字符，第一个处理的字符是换码符。换码符是结束行编辑，还是结束其中的子模式并使编辑器返回到命令模式，由当前的模式决定。任何不是换码符的字符都在基本模式下被处理。

程序清单 5.11　linedit.c

```
static int      stridx;         /* history storage index */
static int      shwidx;         /* history display index */
static int      srchidx;
static int      lastsize;       /* size of last command */
static char     curChar;        /* latest input character */
static char     *curPos;        /* current position on command line */
static char     *startOfLine;   /* start of command line */
static int      lineLen;        /* length of line */
static int      lMode;          /* present mode of entry */
```

```
static char *
lineeditor(char *line_to_edit,int type)
{
    lMode = CMD;
    startOfLine = line_to_edit;
    curPos = line_to_edit;
    while(*curPos != ESC)
        curPos++;
    *curPos = 0;                        /* Remove the escape char from the line */
    lineLen = (ulong)curPos – (ulong)startOfLine;
    if (lineLen > 0) {
        curPos--;
        pwrite(1," \b\b",3);
    }
    else
```

1. 行编辑器也用于 flash 文件编辑器，同时在这种情况下也可应用一些不同的规则。

```
        pwrite(1," \b",2);
    lastsize = 0;
    shwidx = stridx;
    srchidx = stridx;

    while(1) {
        curChar = (char)getchar();
        switch(curChar) {
            case ESC:
                if (lMode != CMD) {
                    lMode = CMD;
                    continue;
                }
                else {
                    putchar('\n');
                    return((char *)0);
                }
        case '\r':
        case '\n':
                putchar('\n');
                if (lineLen == 0)
                    return((char *)0);
                *(char *)(startOfLine + lineLen) = '\0';
                return(startOfLine);
    }
    switch(lMode) {
        case CMD:
                lcmd(type);
                if (lMode == NEITHER)
                    return((char *)0);
                break;
        case INSERT:
                linsert();
                break;
```

```
                case EDIT1:
                case EDIT:
                    ledit();
                    break;
            }
            if (lineLen >= MAXLINESIZE) {
                printf("line overflow\n");
                    return((char *)0);
            }
        }
    }
```

该命令行编辑器支持三种子模式，即 INSERT、EDIT 和 CMD。INSERT 模式是指用户在当前命令行的某处插入字符。字符通过 linsert（）函数插入到命令行。EDIT 模式是指用户改变命令行中的字符（ledit（）函数）。CMD 模式是指在命令行中的输入字符能完成一定的功能，并把编辑器转入到另一个状态。

如果系统处在 CMD 模式，则 l cmd（）函数（见程序清单 5.12）遵照类似 vi 的命令行编辑规则处理不同的字符串；零值告诉编辑器把光标移到行首；A 告诉编辑器进入到 INSERT 模式，光标移到当前行的最后一个字符位置；i、r 和 R 命令仅仅是执行 EDIT 或 INSERT 模式的转换；l 和 h 命令是把光标左移或右移；j 和 k 命令则告诉编辑器调出命令行记录表中的命令。

程序清单 5.12　lcmd()

```
static void
lcmd(int type)
{
    switch(curChar) {
        case '0':
            gotobegin();
            return;
        case 'A':
            gotoend();
            putchar(*curPos);
            curPos++;
            lMode = INSERT;
```

```
        return;
    case 'i':
        lMode = INSERT;
        return;
    case 'x':
        ldelete();
        return;
    case ' ':
    case 'l':
        if (curPos < startOfLine+lineLen-1) {
            putchar(*curPos);
            curPos++;
        }
        return;
    case '\b':
    case 'h':
        if (curPos > startOfLine) {
            putchar('\b');
            curPos--;
        }
        return;
    case 'r':
        lMode = EDIT1;
        return;
    case 'R':
        lMode = EDIT;
        return;
    case '/':
        putchar('/');
        historysearch();
        return;
    case '+':
    case 'j':
        shownext();
```

```
                return;
            case '-':
            case 'k':
                showprev();
                return;
        }

        /* Beep to indicate an error. */
        putchar('\007');
    }
```

如要查询有关命令行编辑的更详细的资料，请参考 CD 上的 lineedit.c 的源代码。

5.8　用户分级

在前面的讨论中，微监控器的命令行接口可以访问应用程序的用户接口，这样做既有好处也有坏处。好处是当应用程序运行时，可访问微监控器。因此，开发者能够在应用程序上做一些“原始金属”的原料，而不需要在应用程序上设置更多的参量。但是，允许应用程序访问微监控器命令行接口，也会使最终用户也能访问到它。这就意味着最终用户可以修改 RAM、可擦写闪存、编辑/移动 TFS 内的文件和采取对系统影响更大、更长的操作。如果你担心用户控制系统，则最简单的解决办法是取消联系，使微监控器命令不能通过应用程序访问到。但是，这样做会中止从应用程序中使用监控器命令行接口。

为了缓和这些问题，微监控器使用用户分级，使每条命令可以单独设置，这样可使其只能被比它级别高的用户运行。如果应用程序运行比它级别低的用户等级的微监控器命令时，用户将不能通过应用程序访问命令。用户等级功能化可提供较大的灵活性，并且可被简单地加到命令行接口。

实现用户等级安全需要为每条命令存储用户使用等级和密码，把每条命令的用户等级存储在一个数组里，该数组与命令列表相对应。注意，这个数组与命令列表不一样，是因为命令列表数组在 ROM 里，而等级和密码数组在 RAM 和微监控器里，假设没有可记录的初始化数据，声明如下所示：

```
char cmdUlvl[(sizeof(cmdlist)/sizeof(struct monCommand))];
```

由于这个列表的入口数量与命令行列表的入口数量相等，所以这两个列表的内

容十分相似。如果在 cmdlist [ ] 列表的第三条命令是 add，则执行 add 命令所需的命令等级储存在 cmdUlvl 列表的第三项，通过向 docommand（）函数加入代码，能使命令行处理器也具有用户分级功能（见程序清单 5.13），getUsrLvl（）函数返回系统的当前用户等级。指针算法用于确定 cmdUlvl [ ] 数组所需值及索引值与 getUsrLvl（）的返回值的比较结果。

程序清单 5.13　向 docommand()增加用户分级

```
...

cmdptr = cmdlist;
while(cmdptr->name) {
    if (strcmp(argv[0],cmdptr->name) == 0)
        break;
    cmdptr++;
}
if (cmdptr->name) {
    if (cmdUlvl[cmdptr-cmdlist] > getUsrLvl()) {
        printf("User-level access denied.\n");
        return(CMD_ULVL_DENIED);
    }
    ret = cmdptr->func(argc,argv);
}

...
```

每条命令的默认用户等级设置为零，微监控器自身运行在第三等级，更高的用户等级意味着更高的特权，所以默认的行为对于所有的命令都是可访问的。微监控器命令 ulvl 允许用户重新配置监控器运行用户等级和访问每条命令所需的命令等级，进入比自身低的用户等级是被允许的，进入比自身高的用户等级则需要密码。由于监控器有四个命令等级，则系统需要三个密码（0 级不需要密码）。ulvl 命令处理密码文件的管理。密码文件经常储存在最高级用户等级，因此对于所有比第三等级低的用户，密码文件是不可访问的。在闪存文件系统的文件存储时，可以根据用户等级设置密码（参考第 7 章“用户分级”）。

程序清单 5.14 是设置命令的用户等级的代码。输入的字符串包含命令名，它由所需的用户等级决定。

程序清单 5.14　设置命令的用户等级

```
int
setCmdUlvl(char *cmdandlevel, int verbose)
{
    struct monCommand *cptr;
    int newlevel, idx;
    char *comma;

    /* First verify that the comma is in the string... */
    comma = strchr(cmdandlevel,',');
    if (!comma)
        goto showerr;

    /* Retrieve and verify the new level to be assigned... */
    newlevel = atoi(comma+1);
    if ((newlevel < MINUSRLEVEL) || (newlevel > MAXUSRLEVEL))
        goto showerr;

*comma = 0;

/* Don't allow adjustment of the ulvl command itself.
 *    It must be able to run as user level 0 all the time...
 */
if (!strcmp(cmdandlevel, ULVLCMD))
{
    printf("Can't adjust '%s' user level.\n", ULVLCMD);
    return(-1);
}

/* Find the command in the table that is to be adjusted...
 */
for(idx=0,cptr=cmdlist;cptr->name;cptr++,idx++)
{
    if (!strcmp(cmdandlevel, cptr->name))
```

```
            {
                /* If the command's user level is to be lowered, then
                 * the current monitor user level must be at least as
                 * high as the command's current user level...
                 */
                if ((newlevel < cmdUlvl[idx]) &&
                    (getUsrLvl() < cmdUlvl[idx]))
                {
                    if (verbose)
                        printf("Ulvl failed: %s\n", cmdandlevel);
                    return(1);
                }
                cmdUlvl[idx] = newlevel;
                return(0);
            }
        }
    showerr:
        if (verbose)
            printf("Input error: %s\n", cmdandlevel);
        return(-1);
    }
```

注意，ulvl 命令不能改变自身的用户等级，经常不需要密码就可以访问。另外，你可以通过把 ulvl 命令设置为第三等级的命令，把用户等级设为第二等级。在等级改变之后，将没有任何方法再设回第三等级，因为你必须已经在第三等级才能使用 ulvl 命令，强制 ulvl 命令在零等级结束这种令人左右为难的规定。

5.9 密码保护

不同用户的密码存储在一个文件里。设置该文件需要第三用户等级，并且当微监控器在第三用户等级时，文件只是可读的。为了保护密码，它们的存储格式也加密。当输入一个密码时，相同加密的算法将被使用。用户输入的加密版本将与文件中的版本相比较。程序清单 5.15 是加密函数一个非常典型的例子。

程序清单 5.15　加密函数

```
static unsigned char *datatbl =
"This_should_be_an_initialized_table_of_256_bytes_of_printable_ascii_characters.";

char *
scrambler(char *string, char *setting, char *result)
{
    int offset, csum;
    char *rp, *sp;

    csum = 0;
    sp = string;
    while(*sp)
        csum += *sp++;
    rp = result;

    /* Set up an offset into the data table that is based on the
     * checksum of the incoming string plus the sum of the two
     * setting characters...
     */
    offset = (csum + setting[0] + setting[1]) % 255;

    /* For each character in the incoming string, replace it with a
     * character in the data table.  If the incoming character has the
     * 00001000 bit set, then use that character twice.
     */
    while(*string)
    {
        offset += (int)*string;
        offset = offset % 255;
        *rp = datatbl[offset];
        if (*string & 8)
        {
            rp++;
```

```
                offset += (int)*string;
                offset = offset % 255;
                *rp = datatbl[offset];
            }
            string++;
            rp++;
        }
        *rp = 0;
        return(result);
    }
```

为了维护密码文件和更新命令用户等级，ulvl 命令使用两个函数，即 setCmdUlvl（）和 scrambler（）。

注意

当系统第一次启动时，monrc 文件包含的内容包括调整目前微监控器用户等级和配置用户等级可访问的命令，这些都属于 ulvl 命令。当启动时，Monrc 文件由监控器自动执行（参考第 7 章的内容）。

5.10 小结

相对来说，已开发的命令行接口是强大的、可扩展的及高度模块化的。命令行接口是值得注意的，因为它可构成监控器的某个组件。命令行接口是一个“巨大的钩子”，可把监控器上的所有部分连在一起；它还是调试命令的接口，可执行应用程序；它又是一个解释器，可以处理配置脚本。

即使命令行接口具有决定性和中心性，也还是具有高度可配置性的。你可以按照需要的特征配置一个特殊的程序执行方式。

第 6 章　闪存的接口

在嵌入式系统中，闪存正在成为标准的存储器。闪存的多功能性、每字节价格和密度的增加，使得它是目前嵌入式系统项目最好的选择。闪存是一种不易失的、系统可写的内存。换句话说，当关掉电源后，闪存可使器件中的内容完整无缺的保存下来，但擦除和写入闪存器件都需要一个复杂的过程。

闪存接口就是为解决上述问题而产生的，从闪存读出数据的方式和从内存读出数据的方式一样，但也就是这点相同。闪存的容量为 64KB~8MB，密度也随之增加。闪存也可被分为几个族，每个族是一个可擦写单元。当一个单元数据被删除后，这个单元的数据全都被置为“1（字节=0xff）”。当往闪存中写数据时（在删除之后），通常是把一位从 1 改为 0。当该位设置为 0 之后，把它变为 1 的惟一方法是擦除整个单元内的所有位。在执行擦写操作时，处理器与内存之间有很复杂的联系。

当（擦或写）操作执行时，CPU 不能访问闪存中的任何数据，因此当你在执行操作时，不能使用闪存之外的器件。一般来说，一个单元可以被至少擦写 100 000~1 000 000 次——对于嵌入式微处理器系统来说，这并不是很多次。

本章将讲述如何对多个不同字长的闪存提供支持，将列出微处理器中可执行闪存操作的命令，讨论有关微处理器闪存特点的函数。这个讨论将延伸到第 7 章，在第 7 章将描述闪存文件系统。

6.1　接口函数

这里将讲述 5 个主要闪存接口函数。这些函数具有控制结构的初始化、ID 器件的重复利用和器件擦写功能。

- FlashInit（）——由闪存器件初始化数据结构，使其与板上器件一起工作。
- Flashid（）——返回系统电路板上的闪存指示值，能详细检查闪存是否可写（在大多数时候，你必须操作闪存读出它的 ID 值），允许防火墙支持转载系统上的多种不同的器件。
- Flasherase（）——完全擦写闪存单元。
- FlashWrite（）——包含变量有一个源指针、目的指针及大小参数，并把一个源数据写入目的地址，并且目的地址在闪存中是可写的。

- FlashEwrite（）——把 Flasherase（）和 FlashWrite（）合并，执行与 FlashWrite（）函数相同，但是在写之前，先擦除所要用的单元，用于重新写一个新的监控器。

6.1.1 闪存库

闪存驱动支持多个器件库（bank），而这些器件不必相邻，库中的一个或两个器件可以被作为单元访问。举例来说，如果一个 8 位器件可被 CPU 访问，那么这个器件是一个单个库。

如图 6.1 所示，如果一个双字节器件与两个内存连接——但每个器件可作为 8 位存储器单独访问（注意，用两个单独的片选），系统之所以包含两个驱动库，是因为在任何给定的时候只有一个器件可被访问。

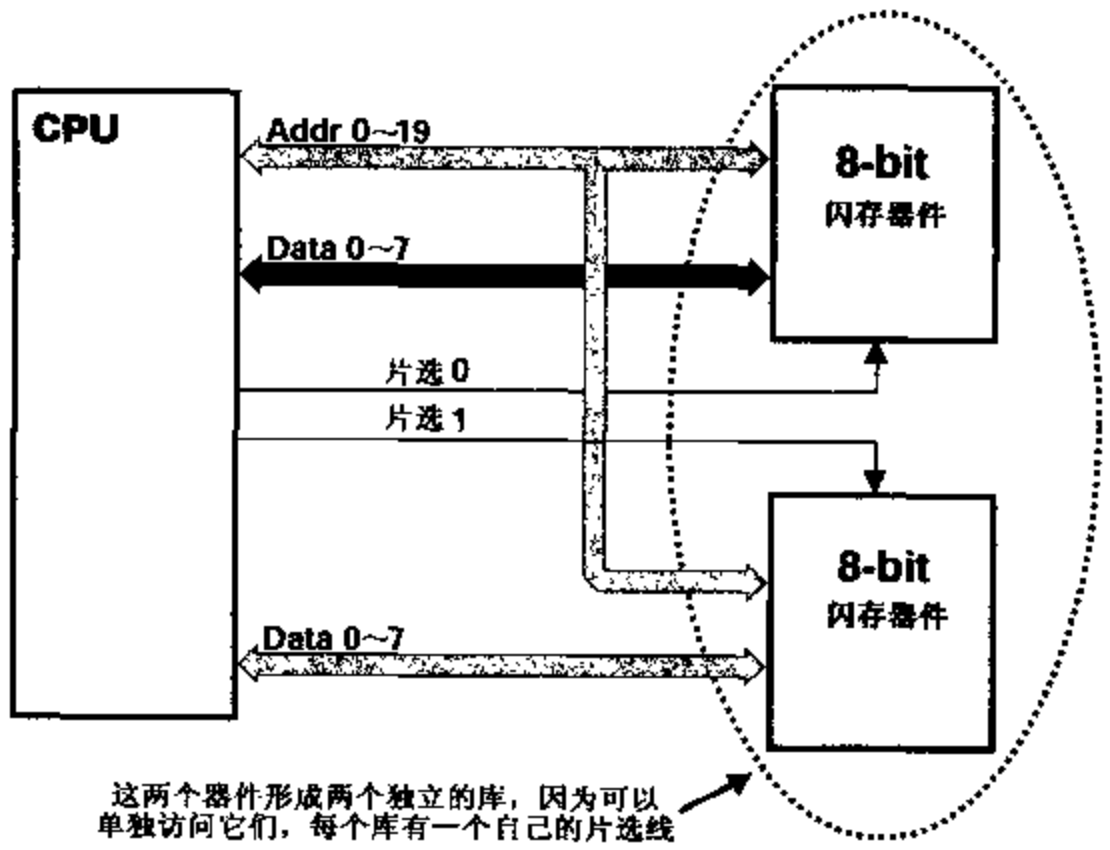


图 6.1 两个闪存器件，两个单独的库

在一个小型系统里，每个内存芯片都有单独的库。在大型系统里，单独的库包括多个并行的芯片（如图 6.2 所示）。本例中，处理器芯片的片选线用于激活相对应的库。

如果连接双字节器件，则 CPU 可以把它们当成一个 16 位的器件访问这两个器件（如图 6.2 所示——只有一个芯片被选择），结果它被处理成一个单独的双器件库。CPU 也可以把由 4 个 8 位器件或 2 个 16 位器件当成一个闪存。

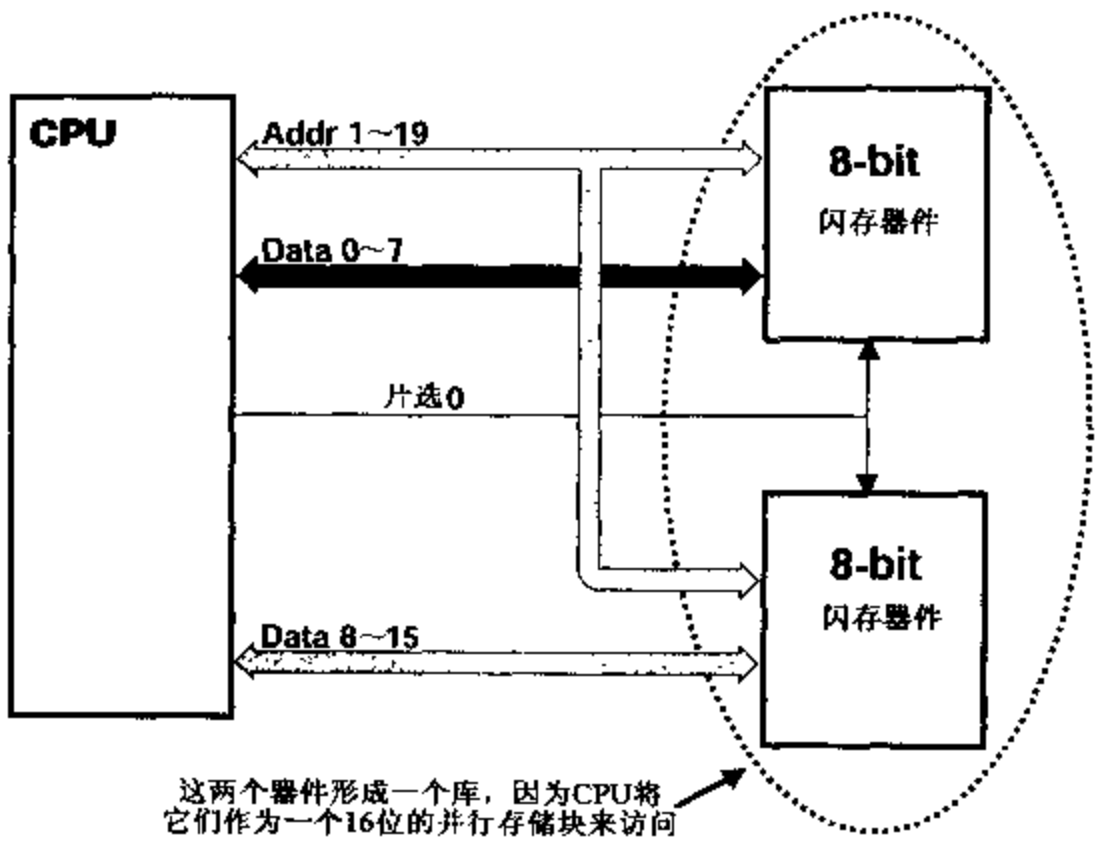


图 6.2　两个闪存器件，一个单独库

本例中，两个 8 位内存芯片并行连接成一个 16 位内存。注意，使用该芯片即激活整个库。

提示

操作闪存经常比较费时，控制要拉器件一下，然后等待一些内部操作完成，依靠设计的目的和要求把更多的器件连在一起，许多闪存器件可以配置成 8 位或 16 位模式。如果你的设计使用大量的闪存（如 4 个），则把 4 个 8 位的闪存组成一个 32 位的闪存，比两个 16 位的闪存更有效率。如将 4 个闪存并行连接，则在执行操作时可以同时访问它们，减少了每字节的编程时间。

6.1.2　在 RAM 中重新部署闪存操作函数

写入闪存接口函数的一个难题是闪存接口函数必须在内存外执行，而不能在闪存器件中处理。有一些比较新的器件可在执行器件的一部分时操作另一部分，但一般不会用到这些新的器件。

闪存接口函数的功能是在闪存中创造一个函数，然后把它复制到 RAM 中，并提取复制空间的地址为函数指针。函数指针用于访问在 RAM 中的函数备份。函数

中的代码必须完全是自相关的（也就是可浮动的）。

来自前线

一般来说，在不调用其他函数时，函数代码经常都是自相关的。但是，一些编译器在使用固有函数进行基本数学运算时，如乘法或除法，虽看不见调用过程，但它确实存在。大多数工具支持浮动地址代码的功能。浮动地址代码的意义是使通往其他函数的分支（从同一函数）不能通过 PC 相关的分支执行，整个函数地址被装载到寄存器中，从这个寄存器中取出分支。这个方法仍不能改变闪存接口函数的问题，原因是如果你调用的函数在闪存中，那么即使你努力避免，还是有可能执行时超过预先设想的空间。这个消息的指针指向重定位在 RAM 中的闪存操作函数。这个函数应该保持一致，并且遵守规则，在代码中不放入任何函数调用。

系统应该复制闪存代码到 RAM，它优先于任何 CPU 的缓存工作（因为 RAM 复制使用数据访问指令存储空间，你不必担心指令可能在数据缓存中）。程序清单 6.1 是一个非常简单的例子，它可实现如何复制一个函数到 RAM 中。注意，缓冲器为 32 位，用 longs 阵列代替 chars，使用 longs 保证缓冲器以 4 模地址开始工作[1]。它假定连接器把函数放在相邻内存空间和函数装载到特定缓冲器中。

程序清单 6.1　复制闪存函数到 RAM

```
int (*RamFlashWrite)();
unsigned long RamFlashWriteBuf[1000];


int
FlashWrite()
{
    /* code to write to flash goes here */
}


int
EndFlashWrite()
{
```

1. 一些处理器要求每一个指令为 32 位，但这并不影响那些没有满足这些条件的 CPU。

```
        return(1);
    }

    ...

    /* Copy from flash space to RAM space: */
    memcpy((char *)RAMFlashWriteBuf,(char *)FlashWrite,
    (int)((ulong)EndFlashWrite-(ulong)FlashWrite));

    /* Establish a function pointer into the RAM buffer: */
    RamFlashWrite = (int(*)())RamFlashWriteBuf;
```

在这段代码完成后，函数指针 RamFlashWrite 用于访问在 RAM 中的 FlashWrite（）函数。RAM 中的函数使用与闪存中相同的 API（应用程序接口）；惟一的不同是从 RAM 中取数，而不是从闪存中取散。

6.1.3　闪存控制结构初始化

为了使相同的 API 能运行在不同的器件中，且使器件类型对于代码可忽略，一般需用较少的结构保存专用器件的信息，主要结构为 flashinfo（见程序清单 6.2）。flashinfo 结构包括一组函数指针，它们被初始化为指向包含闪存操作函数的 RAM 空间。为了使一个驱动支持同一族中的不同器件，器件的 ID 需重新检索，且基于 ID 的关于器件的附加特殊部分信息能被装载，存储在 sectorinfo 结构中。每一扇区有一个 sectorinfo 结构。在 flashinfo 结构中，指向 sectorinfo 结构的指针指向 sectorinfo 结构列表，大小与器件的扇区数量相等，且由 sectors 指针指向的列表在 FlashInit（）函数中建造（见程序清单 6.4）。

程序清单 6.2　闪存器件描述器

```
struct   sectorinfo {
    long      size;              /* size of sector */
    int snum;                    /* number of sector (amongst possibly */
                                 /* several devices) */
    int protected;               /* if set, sector is protected by window */
    unsigned char *begin;        /* base address of sector */
    unsigned char *end;          /* end address of sector */
```

```
};

struct   flashinfo {
    unsigned long    id;          /* manufacturer & device id */
    unsigned char    *base;       /* base address of bank */
    unsigned char    *end;        /* end address of bank */
    int        sectorcnt;         /* number of sectors */
    int        width;             /* 1, 2, or 4 */
    int        (*fltype)();
    int        (*flerase)();
    int        (*flwrite)();
    int        (*flewrite)();
    struct sectorinfo *sectors;
};
```

6.1.4 29F040 系列的闪存操作系统

29F040 系列器件现在比较流行，有多种封装形式，并且有多个生产制造商。下面举一个器件比较的例子。

首先，创造几个 RAM 数组（见程序清单 6.3），用于存放闪存操作函数——每个操作均包括 TYPE、ERASE、WRITE 和 ERASEandWRITE，再定义一个 flashinfo 结构列表。列表（FLASHBANKS）的大小等于系统中扇区的数量。在本例中，列表的大小为 1，数据的最后部分为 sectorinfo 结构列表。由于 29F040 和 29F010 都有 8 个扇区，所以相同的列表能被任何一个器件使用。

程序清单 6.3　RAM 数组

```
ulong     FlashTypeFbuf[FLASHFUNCSIZE];
ulong     FlashEraseFbuf[FLASHFUNCSIZE];
ulong     FlashWriteFbuf[FLASHFUNCSIZE];
ulong     FlashEwriteFbuf[FLASHFUNCSIZE];

struct    flashinfo FlashBank[FLASHBANKS];

struct    sectorinfo sinfo040[8];
```

FlashInit（）函数（见程序清单 6.4）在系统启动时被调用。调用 flashopload（）

函数（没有列出）是把闪存中的闪存操作函数复制到 RAM。如果复制失败，则 flashopload（）返回值为-1，FlashInit（）放弃执行。因为在存储器复制时，可能在数据缓存器中留有一些指令，所以有可能造成混乱。为避免这种混乱，在这种情况下要禁止系统缓存。在此时，将闪存操作复制到 RAM 是很重要的[1]。Flashopload（）函数是特殊的 memcpy（），检验缓冲区的闪存操作函数和检验写操作是否成功（通过读取复制中的数据）。接下来，在 FlashBank［］列表中的 flashinfo 结构被初始化，器件的基数和宽度被记录下来，每个函数指针指向 RAM 数组中相应的位置，而数组中包含闪存操作的代码。

程序清单 6.4　初始化与闪存有关的结构

```
int
FlashInit(void)
{
    int         snum;
    struct   flashinfo *fbnk;

    snum = 0;

    if (flashopload((ulong *)FlashType040,(ulong *)EndFlashType040,
        FlashTypeFbuf,sizeof(FlashTypeFbuf)) < 0)
        return(-1);
    if (flashopload((ulong *)FlashErase040,(ulong *)EndFlashErase040,
        FlashEraseFbuf,sizeof(FlashEraseFbuf)) < 0)
        return(-1);
    if (flashopload((ulong *)FlashEwrite040,(ulong *)EndFlashEwrite040,
        FlashEwriteFbuf,sizeof(FlashEwriteFbuf)) < 0)
        return(-1);
    if (flashopload((ulong *)FlashWrite040,(ulong *)EndFlashWrite040,
        FlashWriteFbuf,sizeof(FlashWriteFbuf)) < 0)
        return(-1);

    fbnk = &FlashBank[0];
```

1. 我们可以在该函数末尾使缓存器失效，但因为这项操作在系统初始化时就已完成，所以也可以在使能缓存器前执行。

```
        fbnk ->base = (unsigned char *)FLASH_BANK0_BASE_ADDR;
        fbnk->width = FLASH_BANK0_WIDTH;
        fbnk->fltype = (int(*)())FlashTypeFbuf;
        fbnk->flerase = (int(*)())FlashEraseFbuf;
        fbnk->flwrite = (int(*)())FlashWriteFbuf;
        fbnk->flewrite = (int(*)())FlashEwriteFbuf;
        fbnk->sectors = sinfo040;
        snum += FlashBankInit(fbnk,snum);
        sectorProtect(FLASH_PROTECT_RANGE,1);
        return(0);
    }
```

为了决定器件的类型，闪存初始化程序调用 FlashBankInit（）（见程序清单 6.5）。FlashBankInit（）函数调用 flashtype（），它是 fbnk->fltype 函数指针的包装［指向 RAM 数组的指针包括闪存 TYPE 操作——由于 flashopload（）］。器件的 ID 是可回收的，并依赖于器件是 AMD20F040 还是 AMD29F010 来配置扇区信息结构（AMD20F040 的扇区大小是 64KB，AMD29F010 的扇区大小是 16KB）。在 FlashBankInit（）底端的循环根据器件类型特征初始化每个 sectorinfo 的入口。

程序清单 6.5　初始化闪存描述符

```
int
FlashBankInit(struct flashinfo *fbnk, int snum)
{
    int i. ssize;



    flashtype(fbnk);
    switch(fbnk->id) {
        case AMD29LV040:
        case SGS29LV040:
        case SGS29F040:
        case AMD29F040:
            fbnk->sectorcnt = 8;
            ssize = 0x10000 * fbnk->width;
            fbnk->end = fbnk->base + (0x80000 * fbnk->width) -1;
```

```
            break;
        case AMD29F010:
            fbnk->sectorcnt = 8;
            ssize = 0x4000 * fbnk->width;
            fbnk->end = fbnk->base + (0x20000 * fbnk->width) -1;
            break;
        default:
            printf("Flash device id 0x%lx unknown\n", fbnk->id);
            return(-1);
    }
    for(i=0;i<fbnk->sectorcnt;i++) {
        fbnk->sectors[i].snum = snum+i;
        fbnk->sectors[i].size = ssize;
        fbnk->sectors[i].begin = fbnk->base + (i*ssize);
        fbnk->sectors[i].end = fbnk->sectors[i].begin + ssize -1;
        fbnk->sectors[i].protected = 0;
    }
    return(0);
}
```

在程序 FlashInit（）结束调用 SectorProtect（）函数（见程序清单 6.4）时，SectorProtect（）函数设定了某些特定的扇区为受保护状态。这里，术语“Protected”是指受“软件”保护。当对器件进行擦除或写入操作的时候，闪存操作首先测试一下受操作影响的扇区是否为受保护状态。如果某个扇区被保护，则闪存操作将被取消。注意，这种方法只是提供一种软件级的保护。在这种情况下，这种保护已经完全可以满足需要，因为两个不同的函数必须顺序地被调用才可以进行下一步的操作——首先解除对某个扇区的保护，然后对其进行操作。因为“解除保护”的操作是通过 MicroMonitor CLI 来完成的，这两个必需的步骤并不在同一个函数中，这就避免了对受保护扇区的某些写或擦的误操作发生。

现在已经解释了闪存的函数是怎样复制到 RAM 中去的，下面将说明闪存的内部接口是怎样工作的。首先我们来看一下闪存写操作的某些细节问题。

29F040 含有 8 个 64KB 的扇区，每个部分都可以被擦除 100 000 次以上。我们可以对 RAM 和 DRAM 进行任意的写操作，但闪存与它们不同。对闪存进行实际写操作的时候，操作者必须清楚地了解是对哪个器件进行写操作。一次典型的

写操作包含了将某些数据块写入内存内的大块空间后，实际的数据单位才被写入。运算法则等待闪存的内核显示写操作已完成之后，才最终通知器件返回标准的读取模式。

程序清单 6.6 列出了对 29F040 进行写操作的代码。函数的指针指向了结构体 flashinfo、原始文件、目标文件及文件的大小。注意，3 个 WRITE 宏在 FWRITE 宏的前面，3 个 WRITE 序列使器件准备好进行实际的写操作，而最终的写操作由 FWRITE 宏来完成。在 FWRITE 传输完信息之后，一个等待的循环将检测器件的操作是否结束。当操作结束之后，另外一个命令序列写入器件使器件返回标准的读取模式，从而使器件为读操作做好准备。

程序清单 6.6　闪存写操作

```
/* Macros used for flash operations: */

#define ftype                   volatile unsigned char
#define WRITE_AA_TO_5555()      (*(ftype *)(fdev->base + (0x5555<<0)) = 0xaa)
#define WRITE_55_TO_2AAA()      (*(ftype *)(fdev->base + (0x2aaa<<0)) = 0x55)
#define WRITE_80_TO_5555()      (*(ftype *)(fdev->base + (0x5555<<0)) = 0x80)
#define WRITE_A0_TO_5555()      (*(ftype *)(fdev->base + (0x5555<<0)) = 0xa0)
#define WRITE_F0_TO_5555()      (*(ftype *)(fdev->base + (0x5555<<0)) = 0xf0)
#define WRITE_90_TO_5555()      (*(ftype *)(fdev->base + (0x5555<<0)) = 0x90)
#define WRITE_30_TO_(add)       (*(ftype *)add = 0x30)
#define READ_0000()             (*(ftype *)(fdev->base + 0x0000<<0)))
#define READ_0001()             (*(ftype *)(fdev->base + (0x0001<<0)))
#define READ_5555()             (*(ftype *)(fdev->base + (0x5555<<0)))
#define IS_FF(add)              (*(ftype *)add == 0xff)
#define IS_NOT_FF(add)          (*(ftype *)add != 0xff)
#define D5_TIMEOUT(add)         ((*(ftype *)add & 0xdf) == 0x20)
#define FWRITE(to,frm)          (*(ftype *)to = *(ftype *)frm)
#define IS_EQUAL(p1,p2)         (*(ftype *)p1 == *(ftype *)p2)
#define IS_NOT_EQUAL(p1,p2)     (*(ftype *)p1 != *(ftype *)p2)

int
FlashWrite040(struct flashinfo *fdev, ftype *dest,ftype *src,long bytecnt)
{
```

```
int i, ret;
ftype     val;

/* Each pass through this loop writes 'fdev->width' bytes... */
ret = 0;
for (i=0;i<bytecnt;i++) {

    /* Flash write command */
    WRITE_AA_TO_5555();
    WRITE_55_TO_2AAA();
    WRITE_A0_TO_5555();

    /* Write the value */
    FWRITE(dest,src);

    /* Wait for write to complete or timeout. */
        while(1) {
            if (IS_EQUAL(dest,src)) {
                if (IS_EQUAL(dest,src))
                    break;
            }
            /* Check D5 for timeout... */
            if (D5_TIMEOUT(dest)) {
                if (IS_NOT_EQUAL(dest,src))
                    ret = -1;
                goto done;
            }
        }
        dest++; src++;
    }
done:
    /* Read/reset command: */
    WRITE_AA_TO_5555();
    WRITE_55_TO_2AAA();
```

```
        WRITE_F0_TO_5555();
        val = READ_5555();
        return(ret);
}

/* EndFlashwrite():
 *  Function place holder to determine the "end" of the
 *  Flashwrite() function.
 */
void
EndFlashWrite040()
{}
```

Flasherase（）函数（见程序清单 6.7）非常类似于 FlashWrite（）函数。一系列的 WRITE 通知器件将要进行擦除操作后，对某个特定扇区进行擦除操作。一个等待的循环将检测完成信号，然后，一系列的 WRITE 将通知器件返回标准的读取模式。

程序清单 6.7　Flasherase()

```
/* Flasherase():
 * Based on the 'snum' value, erase the appropriate sector(s).
 * Return 0 if success, else −1.
 */
int
FlashErase040(struct flashinfo *fdev,int snum)
{
        ftype                   val;
        unsigned long           add;
        int                     ret, sector;

        ret = 0;
        add = (unsigned long)(fdev->base);

        /* Erase the request sector(s): */
        for (sector=0;sector<fdev->sectorcnt;sector++) {
```

```
if ((!FlashProtectWindow) &&
        (fdev->sectors[sector].protected)) {
        add += fdev->sectors[sector].size;
        continue;
}
if ((snum == ALL_SECTORS) || (snum == sector)) {
        /* Issue the sector erase command sequence: */
        WRITE_AA_TO_5555();
        WRITE_55_TO_2AAA();
        WRITE_80_TO_5555();
        WRITE_AA_TO_5555();
        WRITE_55_TO_2AAA();

        WRITE_30_TO_(add);

        /* Wait for sector erase to complete or timeout..
         * DQ7 polling: wait for D7 to be 1.
         * DQ6 toggling: wait for D6 to not toggle.
         * DQ5 timeout: if DQ7 = 0, and DQ5 = 1, timeout.
         */
        while(1) {
            if (IS_FF(add)) {
                if (IS_FF(add))
                    break;
            }
            if (D5_TIMEOUT(add)) {
                if (IS_NOT_FF(add))
                    ret = -1;
                break;
            }
        }
    }
    add += fdev->sectors[sector].size;
}
```

```
        WRITE_AA_TO_5555();
        WRITE_55_TO_2AAA();
        WRITE_F0_TO_5555();
        val = READ_5555();
        return(ret);
}

/* EndFlasherase():
 * Function place holder to determine the "end" of the
 * sectorerase() function.
 */
void
EndFlashErase040()
{}
```

来自前线

在 Flasherase（）函数的开始部分，一个全局变量 FlashProtectWindow 与结构体 sectorinfo 的 protected 成员进行比较。FlashProtectWindow 变量只能通过 CLI 命令 flash opw（open flash protection window）来设定，并且其结果被赋值为 2。在每一个 CLI 命令的结束处，这个变量被检验是否为零，如果为正，则将被减 1。在 flash opw 命令结束时，这个变量将被减至 1。因此，变量不为零的时间窗口就是 CLI 命令，这便是我们前面提到的软件保护。

6.1.5 对 16 位与 32 位的扩展（banks）操作

讨论闪存驱动的开始曾经提到过这些器件可以存在很多扩展，每一个扩展后的存储器可以是 8 位、16 位或者是 32 位。前面的例子详细阐述了一个 8 位的器件接口，这对说明如何对闪存器件进行操作是很重要的。下面我们将讨论一个稍微复杂的问题。如果用两个器件并行地组成一个 32 位的扩展存储器，那么对 FlashWrite（）函数的调用将面临这样的可能性：写入操作的开始地址可能不是 32 位的。扩展

存储器的模型要求组成的存储器具有相同的宽度位数，如果在某些奇数地址的存储位置上开始进行写入操作，则需要返回到前面 32 位排列的地址，并且作为第一次写入操作单元的一部分包括那些早已存在于闪存内的信息。请参见图 6.3。

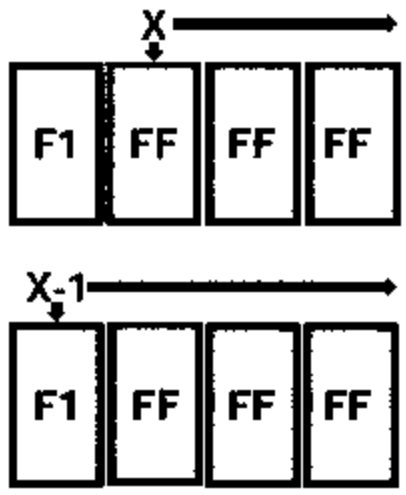


图 6.3　调整首地址为 32 位排列

为了从 X 地址位置开始写入 0x112233，我们将 3 个字节开始部分的写操作转变为 4 个字节后半部分的写操作。这个过程首先是指针返回至前一个目的地址，然后将返回的那一个字节作为 4 个字节的一部分写入存储器的存储空间中，结果是我们在 X-1 地址处写入了 0xF1112233。

同样的原理也适用于不是以 32 位排列的写入操作。我们必须读取之前的一些字节，并且在最后的 32 位的写入操作中包括这些读取的字节。

6.2　闪存驱动的前端（Front End）

在这里，闪存器件驱动程序中的与闪存有直接关系的部分被复制到 RAM 中，这样，系统在脱离了 29F040 后仍然可以执行闪存操作。然而，比较低级的函数，如 FlashWrite040（）和 FlashErase040（）是依赖于器件的。因此，当我们要开始在系统的剩余部分使用这些功能时，需要增加另外一层。我们要一个前端来允许我们通过使用一个类似于 memcpy（）的接口函数来对任意的闪存器件进行写入操作（例如，源与目标和大小）。对于擦除操作，我们需要能够调用某个函数并确定我们要擦除的扇区。这个函数使高级语言程序不用知道具体的闪存器件。

在这个命令行中，闪存通过 flash 命令进行操作。程序清单 6.8 可解释这个命令是怎样在不知道某个具体器件的情况下对闪存进行操作的。

程序清单 6.8 flash 命令

```
int
FlashCmd(int argc,char *argv[])
{
    int ret;
    ulong    dest, src, oints;
    long     bytecnt, rslt;
    struct   flashinfo *fbnk;

    fbnk = &FlashBank[FlashCurrentBank];
    ret = CMD_SUCCESS;

    if (strcmp(argv[1],"bank") == 0)  {
        int tmpbank;
        if (argc == 3) {
            tmpbank = atoi(argv[2]);
            if (tmpbank < FLASHBANKS)
                FlashCurrentBank = tmpbank;
                printf("Subsequent flash ops apply to bank %d\n",
                    FlashCurrentBank);
        }
        else
            printf("Current flash bank: %d\n",FlashCurrentBank);
}
else if (!strcmp(argv[1],"write")) {
    if (argc == 5) {
        dest = strtoul(argv[2],(char **)0,0);
        src = strtoul(argv[3],(char **)0,0);
        bytecnt = (long)strtoul(argv[4],(char **)0,0);
        rslt = AppFlashWrite((ulong *)dest,(ulong *)src,bytecnt);
        if (rslt == -1)
            printf("Write failed\n");
    }
    else
```

```
                ret = CMD_PARAM_ERROR;
        }
        else if (!strcmp(argv[1],"opw")) {
            FlashProtectWindow = 2;
        }
        else {
            ret = CMD_PARAM_ERROR;
        }

        return(ret);
    }
```

flash 命令的格式为：

```
flash {command}[args based on command]
```

例如：

```
flash write 0x1000 0xfff80000 128
```

它是将 128 个字节的内容从 0xfff80000 地址复制到 0x1000 的闪存目标位置处。由于这个命令没有采用任何选择（见程序清单 6.8），因此 getopt（）函数没有被使用，FlashCurrentBank 的内容被指向了适当的层。基于前面的讨论，我们知道 29F040 的对象只有一个层，但是可以通过增加 FlashBank［］数组的大小使系统内层的数目成倍增加。程序清单 6.8 同时也说明了如何设定 FlashProtectWindow 变量时，其他可能被软件保护的程序才可以运行。这里要注意在 FlashCMD（）函数中调用了 AppFlashWrite（）函数。

AppFlashWrite（）函数（见程序清单 6.9）可用来提取闪存的特征。注意，没有任何的特殊器件信息被传送到这个函数中，因此这个调用与闪存的底层特征无关。

程序清单 6.9　提取闪存的特征

```
int
AppFlashWrite(uchar *dest,uchar *src, long bytecnt)
{
    struct   flashinfo *fbnk;
    int      ret;
```

```
    long        tmpcnt;

    ret = 0;
    while(bytecnt > 0) {
        fbnk = addrtobank(dest);
        if (!fbnk)
            return(-1);

        if (((int)dest + bytecnt) <= (int)(fbnk->end))
            tmpcnt = bytecnt;
        else
            tmpcnt = ((int)(fbnk->end) - (int)dest) + 1;

        ret = fdev->flwrite(fbnk,dest,src,tmpcnt);
        if (ret < 0) {
            printf("AppFlashWrite(0x%lx,0x%lx,%ld) failed\n",
                (ulong)dest,(ulong)src,bytecnt);
            break;
        }
        dest += tmpcnt;
        src += tmpcnt;
        bytecnt -= tmpcnt;
    }
    return(ret);
}

struct flashinfo *
addrtobank(uchar *addr)
{
    struct   flashinfo *fbnk;
    int      dev;

    for(dev=0;dev<FLASHBANKS;dev++) {
        fbnk = &FlashBank[dev];
```

```
            if ((addr >= fbnk->base) && (addr <= fbnk->end))
                return(fbnk);
        }
        printf("addrtobank(0x%lx) failed\n",(ulong)addr);
        return(0);
    }
```

AppFlashWrite（）函数存在一些技巧。我们注意到，addrtobank（）将目的地址转换为相应的 flashinfo 结构体的指针后，目的地址将与层的末地址进行比较。如果目标地址与 bytecount 之和超过了层的末地址，则 bytecount 减小至在层的末地址结束时的值。在下一次循环中，addrtobank（）被再次调用，返回到下一次写周期的新的 flashinfo 指针。注意，flashinfo 结构体又一次被用来调用闪存写入函数（通过指向与目标地址重合的闪存函数指针 fdev->flwrite 来完成）。这只适用于具有多个闪存层的系统。

6.3　小结

本章讲述了闪存的内部接口函数，讨论的是总线结构的闪存器件中如何使用前端来工作。API 和其他的命令提供了使用者与闪存器件之间的接口，要求使用者懂得通过 API 来写入或擦除闪存器件的地址空间。不管底层的器件如何，只要 API 被执行，写入和擦除操作就会成功。

本章讨论的闪存内部接口程序在应用层次上是很方便的，但在驱动层次上，还需要写一些与器件的接口有关的协议（类似于我们刚刚讨论过的 29F040 器件）。记住，29F040 是结构和接口都非常类似的元器件系列中的一员。这个系列应用了器件的特征 ID，无论实际焊接在电路板上的器件如何，均允许固件（firmware）进行自我配置。因为器件是可以进行自我配置的，我们可以不用对固件做任何的更改，只需利用一个系列中不同的器件来作为目标硬件。如果有一两种不同类型的器件支持这种解决方案的话，那么对那些焊接在电路板上的器件就足够简单了。

第 7 章　闪存文件系统

监控器的设计包含了关于硬件完整基础框架的特征。前面已经介绍了启动、串行接口、闪存驱动程序及固件的命令行接口，在接下来的章节里，我们将更进一步地深入学习。本章讲述有关闪存文件系统 TFS（Tiny File System）方面的知识。闪存文件系统在嵌入式系统中是比较罕见的，但随着高密度闪存部分和可移动闪存介质的使用，我们将会遇到更多的闪存文件系统。过去闪存文件系统被认为是琐碎浪费的，因此程序设计者必须经常利用某些自己编写的源程序作为应用程序与闪存器件接口。闪存文件系统的确可以使嵌入式系统更易于管理，但从草稿开始写会比你预期的更复杂。

本章将要讲述可以使程序设计者以命名来确定闪存内的存储空间而不是用地址空间的机制。TFS 解决了与闪存接口的某些主要问题。例如，TFS 提供了一个独立于底层闪存器件的 API，而且如果需要的话并不抑制程序直接存取未被使用的存储空间。TFS 提供了可供选择的有效的实际应用方法，只需要一个闪存文件系统，而并不要求一个复杂的损耗平衡功能（wear-leveling）、目录层次、DOS 的文件格式的兼容性执行。TFS 提供了可以与微监控器及微监控器加载的应用程序的 API 接口。TFS 的 API 提供了可以进行读、写的打开、关闭、查找、统计集合信息等操作的接口。微监控器的 TFS 命令可以在闪存器件上进行列表、删除、创建、显示、复制、解压缩、加载、执行和清除文件的操作，可以自动地导入一个或多个应用文件，并且允许导入平台的其他部分来显示已经存在的闪存文件系统。

7.1　TFS 在平台上的作用

我们所建立的平台在很大程度上依赖于现有的 TFS 文件系统，即使是微监控器基本的启动也要从文件中获取某些特定的配置参数，控制监控器运行文件 monrc 的执行，允许 TFS 中的文件来设定平台中的其他各种各样的参数。在开始时，自动运行的源文件中像 IP 与 MAC 地址的参数均被设定好。在第 8 章中，我们将会了解到 TFS 是如何允许文件被修改（scripts）（类似于 DOS 批处理文件）。通信接口（Xmodem 与 TFTP）都可以用来与 TFS 接口来进行文件传输，且在应用程序被转

换为TFS之后，可以在系统启动时通过TFS的加载而自动地从闪存下载至RAM中。于是，当程序运行的时候可以通过 TFS 的 API 来读取闪存中的文件。TFS 不仅仅有一个接口，而且也是这个平台的主要组成部分。

7.2　TFS 的设计标准

TFS 的基本目标是使固件提供直接通过命名空间而不是地址空间来处理系统的闪存。这种方法免除了每一个新的应用程序对以某种不熟悉或者笨拙的方式来处理闪存的需要。同时，我们并不想失去在简单存储的文件里读写信息的能力，因此需要一个 API 来支持命名空间的模型，想要一个未使用闪存的钩子（hook）来支持基本的地址/信息模型。如果要同时支持这两种读写方法，那么 TFS 必须保证每一个文件的信息均保存在闪存中相邻的地址空间中。

另外一个设计的目标是使TFS独立于器件及RTOS——为了避免可能存在的实时冲突。TFS 被设计成可以不要求系统的中断状态，且可在一个很大的闪存器件范围内工作。对底层闪存器件惟一的限制是它的一个扇区的大小一定要大于 TFS 的头的大小(目前是 92B)，使 TFS 可以支持要求文件模型的高级应用(通过打开、关闭、读、写的 API 来对文件的信息进行操作)，同时也可以支持需要快速读取存储器的实时应用（通过文件名读取文件，在文件里通过地址处理信息）。

TFS 是一个线性的文件系统，可以提供给嵌入式系统工程可能需要的所有的类似于文件系统的特点。TFS 不支持任何的复杂的损耗平衡功能运算法则，也不需要目录层次，不兼容于其他的文件系统。我们需要对那些需要经常读写闪存而需要 wear leveling 的项目做一些工作。如果介质是不可移动的话，那么与 DOS 的兼容性是不必要的。TFS 是独立于 RTOS 的（它不需要 RTOS），并且很容易地跟踪（hook）在应用程序中。

7.3　文件属性

TFS 支持文件属性。每一个属性（或标志）都简单地描述了 TFS 的文件。在文件头里，属性由一个简单的比特设置。在命令行中，每一个属性均通过一个字母设定。表 7.1 是所有文件属性的列表，包括每个属性的简单描述。

表 7.1 文件属性

属 性	缩 写	描 述
可执行的 script	e	可以作为 script 执行的 ASCII 文件
可执行的二进制文件	E	基于此格式下载和运行的二进制文件
自动加载	b	文件在开始启动时自动执行
提示是否自动执行	B	如果用户询问通过，则文件在开始启动时自动执行
压缩	c	文件被压缩
存档	I	文件已存档
不可读	u	文件在低级用户模式下不可读
用户等级	0~3	文件在它的用户等级及用户等级之外只能被写

目前，可执行的二进制文件格式为 AOUT（old Unix file format）、COFF（common object file format）和 ELF（executable and linking format）。TFS 可以很容易地引入新的文件格式。

7.3.1 可以自动加载的文件

在启动过程中的某个时刻，微监控器在 TFS 中查找可以自动加载的文件，且可支持三种可以自动加载的文件，即两种由文件的属性赋予的文件类型和一种特殊的情况（monrc 文件）。如果要求 monrc 文件自动运行，那这个文件必须存在，并被设定为 e 属性（executable script）。在 TFS 中，monrc 文件运行的优先级高于其他可自动加载的文件，并且高于微监控器完成自身初始化的优先级。微监控器因此可以利用 monrc 文件执行时产生的参数来完成自身的配置，剩余的可以自动加载的文件在微监控器完成自身初始化之后才开始运行。由于这些文件按照字母的顺序运行，因此它们在 TFS 中存放的先后顺序是不重要的，通过 tfs ls 命令列出的顺序便是它们被执行的顺序。

程序清单 7.1 tfs ls 命令的输出

```
Name                Size      Location     Flags
bkgd.jpg            5976      0x80047a6c
cardtilt.gif        6099      0x80040c1c
boot_diag           109358    0x8004921c   BE
construction.gif    20222     0x8004243c
```

form.html	466	0x8004004c	
my_app	792080	0x80155a7c	BE
index.html	442	0x8004026c	
info1.html	1053	0x8004047c	
info2.html	734	0x800408ec	
lucentlogo.gif	1680	0x8004738c	
monrc	823	0x80154d2c	e

程序清单 7.1 给出文件自动加载的顺序为：

（1）monrc；

（2）boot_diag；

（3）my_app。

所有其他列出的文件均是被应用程序调用的资源文件。这两个支持可自动加载的属性是 B 和 b，两种文件都是在启动的时候运行的。B 类型在控制台端口询问用户，提供了选择该文件退出自动加载的可能性。如果在用户回答之前询问时间（大约 2s）已经结束，那么这个文件便被执行了，因此可以利用自动加载属性来设定 scripts 和二进制的可执行性。

7.3.2　用户级别

微监控器支持用户级别的概念。在任何时刻，一个用户的级别被认为是起作用的。TFS 提供了设定文件为某用户级别的功能，可以根据用户的级别来限制其对某文件的读写。个别的文件可以被设定为只读或者不可读。因为每一个用户级别只能通过密码来确定，一个系统可以建立在第三级，然后降为第二级、第一级或者第 0 级，从而对未经授权的读写提供不同程度的文件保护。

7.4　高级的详细内容

下面将针对一个基本的系统来进行讨论，包括 CPU、I/O、RAM 和一个闪存器件。这个系统的闪存包括 3 个部分，即被微监控器自身代码所使用的闪存空间、用于存储文件的闪存空间和用于可中断的碎片整理程序的空间（“备用的扇区”）。TFS 存储文件的闪存开始于一个扇区的边界，备用的扇区紧接在文件存储的最后一个扇区之后，不能小于这个器件内其他的扇区，而且在备用的扇区之前的其他扇区被认为是相同大小的（如图 7.1 所示）。

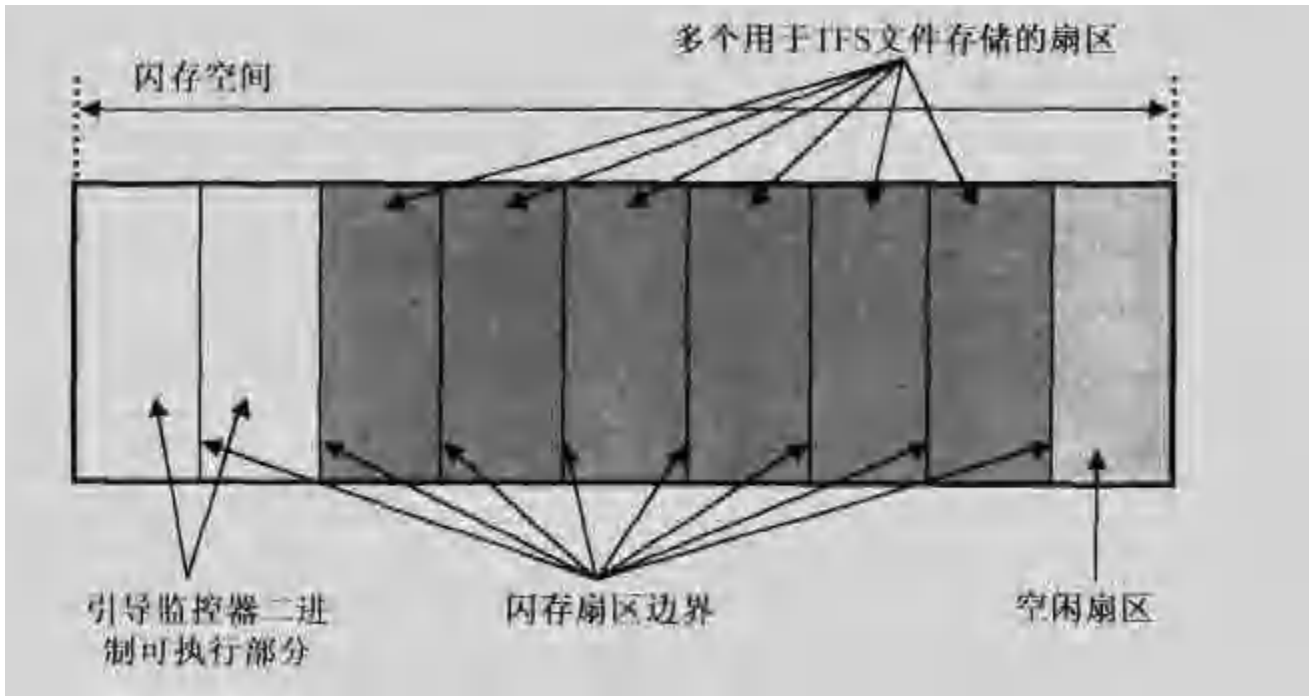


图 7.1 TFS 在闪存上的分配

TFS 通过一个连续的单向的链表来组织闪存中的文件。文件的初始部分是文件头（如图 7.2 所示），包括了本文件的信息、指向下一个文件的指针及一个关于文件头和文件信息部分的 32 位循环冗余检验（CRC）。保持惟一的文件头和内容的 CRC 检验允许 TFS 检测错误。文件的大小仅受由 TFS 所决定的闪存数量所限制。这里对扇区的边界没有任何的限制。文件头结构体的内容见程序清单 7.2。

程序清单 7.2 TFS 的文件头结构体

```
struct tfshdr {
    ushort   hdrsize;                  /* Size of this header.                 */
    ushort   hdrvrsn;                  /* Header version #.                    */
    long     filsize;                  /* Size of the file.                    */
    long     flags;                    /* Flags describing the file.           */
    ulong    filcrc;                   /* 32 bit CRC of file.                  */
    ulong    hdrcrc;                   /* 32 bit CRC of the header.            */
    ulong    modtime;                  /* Time when file was last modified.    */
    struct   tfshdr  *next;            /* Pointer to next file in list.        */
    char     name[TFSNAMESIZE+1];      /* Name of file.                        */
    char     info[TFSINFOSIZE+1];      /* Miscellaneous info field.            */
#if TFS_RESERVED
    ulong    rsvd[TFS_RESERVED];
```

```
#endif
};
```

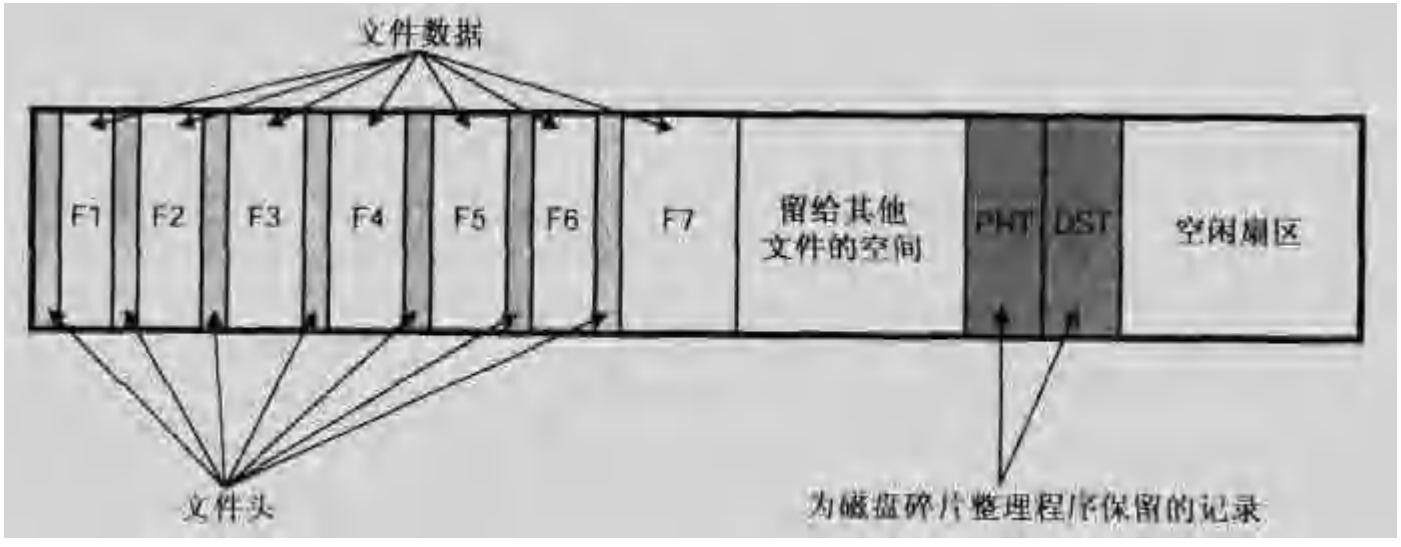


图 7.2　闪存空间内的文件（信息及文件头）

TFS 一定要在系统第一次被构建的时候进行初始化，因此分配给 TFS 的闪存空间必须被擦除。当文件被创建的时候被附加在文件链表的末尾。当一个文件被从链表里删除的时候，这个文件仅仅被标识为已被删除。当许多文件被删除之后，就需要通过运行碎片整理程序来清理 TFS 的闪存空间。碎片整理程序要求备用扇区以及附加的一些空间从最后的 TFS 扇区的头部开始向下增加，而它们的大小是由正在运行的文件数量决定的。

注意，备用扇区不能位于已被 TFS 使用的空间内部，而必须位于闪存空间的尾部，这是因为 TFS 假定所有的文件都是在闪存空间里连续的。信息的连续性对时间要求特别严格的应用来说是非常好的特性。我们可以在闪存里存储信息文件，并可以通过文件名读写文件来重新得到信息的开始点。这样，我们就可以通过简单而更有效的闪存读写从存储空间内获取信息。

7.5　TFS 所要求的闪存空间

TFS 不能自由地覆盖（overlaying）在闪存器件上，需要一定数量的管理空间。某些特殊的空间是固定的，而某些是依据存储的文件而定，如图 7.2 所示，有 5 种管理空间必须被考虑。

- TFS 的文件头（header）——每个文件包含 92B 的管理空间。92 的值是假定 name 和 info 的大小均为 23（对 NULL 终止的＋1）字符，并且保留的条目数为 4。
- 碎片整理后的报头表格——这个文件头的表格（加上碎片整理程序的信息）

位于 TFS 空间的结束处。在每个碎片整理循环的时候，TFS 为每一个有用的（未被删除的）文件创建了一个表格中的条目。

- 碎片整理状态表格——每个扇区包括两个 32 位的 CRC，其中一个 CRC 计算碎片整理前的扇区，而另一个 CRC 在扇区被碎片整理的运算规则处理后计算。
- 备用扇区——碎片整理运算规则用来计算 scratch pad 空间的扇区。备用空间不能小于 TFS 空间内的其他扇区。
- 双缓冲区——当刷新一个文件时使用。也就是说，如果文件 A 存在于 TFS 的闪存空间中并且在某个时刻文件 A 被刷新，则原有的文件 A 在新文件 A 被成功创建之前并没有被删除。结果是系统必须为在某时刻附加的文件处理做准备，包括原有的文件以及即将存在的新文件。

如果忽略头中的双缓冲器，那么我们可以使用以下的方程来计算由 TFS 引入的全部管理空间：

```
overhead= (FTOT*(HDRSIZE+DEFRAGHDRSIZE+16)) +SPARESIZE+
(SECTORCOUNT*8))
```

这里：

- FTOT 是被储存的文件数目；
- HDRSIZE 为 TFS 的文件头的大小（目前为 92B）；
- DEFRAGHDRSIZE 为 TFS 磁盘整理程序的头的大小（目前为 64B）；
- SPARESIZE 为备用扇区的大小；
- SECTORCOUNT 为 TFS 分配的扇区数目（不包括备用扇区）。

注意，一个被标识为已删除的文件比一个有用文件的管理空间更小，这是因为被标识为已删除的文件不要求磁盘整理的头，即使这个文件并没有真的从闪存中擦除，当删除它时仍然可以释放某些存储空间。

7.6 碎片整理

底层的闪存技术并不允许擦除任意大小的器件内的存储空间。因此，当一个文件被删除的时候，它只是在报头内简单地标识为已删除，而实际仍遗留在闪存里。为了使已被删除文件占据的空间得到重新利用，可将“空闲”的范围重新定位为被擦除的位置。图 7.3 显示了这个碎片整理的过程，被加深的阴影表示的区域代表了文件头，被稍轻的阴影标识的是已删除的文件。

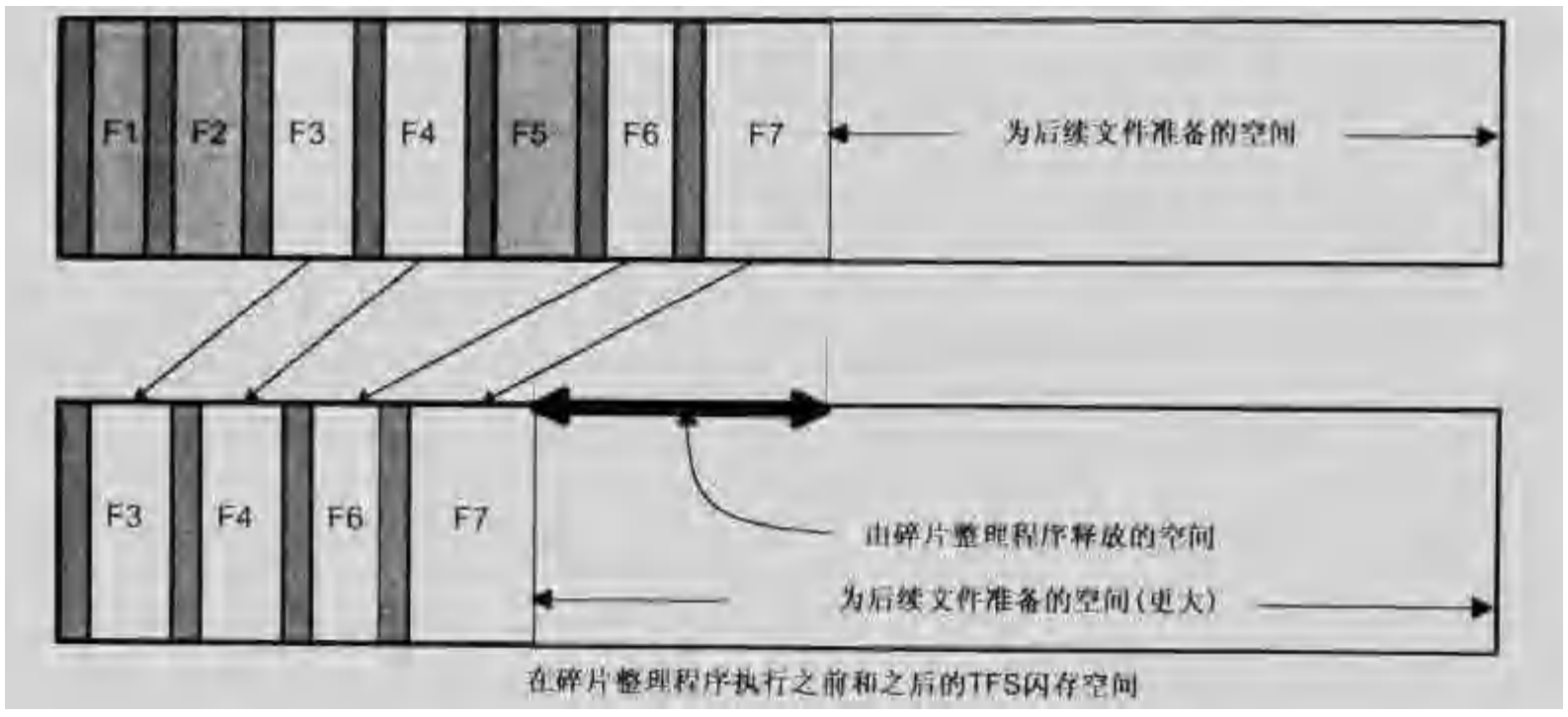


图 7.3　TFS 碎片整理的过程

清除或者碎片整理存在几种不同的方法。类似于大多数其他的嵌入式系统，方法的选择在很大程度上依赖于应用环境的特点、提供的硬件及在复杂性和容量之间的折中考虑。我们当然可以找到微监控器之外的更复杂方法来进行碎片整理，但是通常这些增加的复杂性会使整个文件系统的实现变得更为复杂。我们的目标是要保持文件存储的简单化和线性化，以满足应用中对任一文件在闪存中连续性存在的假定。下面将讨论两种不同的实现方法：第一种非常简单但存在一定的缺陷（与系统有关，但可以被接受）；第二种方法相对功能强大，并且可通过一些附加的选择项变得与闪存相一致。

7.6.1　简单但是存在潜在危险的方法

碎片整理最简单的处理方法是将每一个没有被删除的文件集中起来放入 RAM 空间中，擦除闪存上为 TFS 文件存储分配的所有空间，将全部的连续数据在复制后闪存。此处理方法的优点是非常简单，而且不需要将闪存的某个扇区预先设定为备用扇区。此方法的缺点是假定 RAM 中与闪存中分配给 TFS 相同大小的一部分始终静态工作。如果系统在进行磁盘碎片整理时重新启动或者遇到了电源的抖动，那么文件系统将可能被破坏。

7.6.2　比较复杂但功能更为强大的方法

另一种更为可行方法的功能比较强大，它允许系统在碎片整理过程的任意时刻均可以重新启动，并不需要使用比较大的 RAM 空间，但要求在进行碎片整理前便预先

分配好闪存中一部分固定的空间（至少与 TFS 占用的最大扇区相同大小）。在 TFS 中，这部分空间是备用空间，在闪存中紧接在 TFS 用来进行文件存储的扇区之后。因为备用扇区太小以至于不可能将所有的有用（未删除的）文件全部保留复制，所以碎片整理的过程变得有些复杂。复制必须一部分 一部分地进行，而不是全部一次性地完成。

尽管这种方法更为复杂，但避免了电源波动与系统重新启动所带来的破坏。这种碎片整理可以在任 一时刻重新开始。其缺点之一是备用扇区需进行频繁的循环使用。备用扇区是最有可能达到技术的限制（擦除循环的数量）而开始出错的地方。这种限制达到的时刻依赖于器件、分配给 TFS 的扇区数量及文件被删除和重新产生的速度等因素。方案的改进（提高闪存器件整个的使用周期）方法之一是用带有后备电源的 RAM 代替备用扇区，但费用比较高，不适合那些预算比较紧张的应用。因此，使用后备电源并不十分可行。

这部分讲了微监控器中最困难的一部分代码。碎片整理操作使用了三种不同的闪存空间：

- 已经被预先分配好的备用扇区；
- 碎片整理进程的状态列表，包括标识 TFS 存储文件的扇区的两个 32 位的 CRC。这些信息可以使被中止的碎片整理操作在重新开始时即可以探测到中止时所处理的扇区。
- 碎片整理后的报头列表，是所有经过碎片整理的文件（所有未被删除的文件）报头的列表。碎片整理的报头与我们前面讨论过的头结构体有些类似，在重新开始一个被中止的碎片整理操作时被碎片整理状态报表所使用。

7.7 TFS 的应用

所有 TFS 代码的讨论将涉及到大量的细节，超出了本书所应讨论的范围。因此，下面只是大体地浏览一下 TFS 的主要代码（主要的应用程序在 CD 上），在讨论 TFS 设计与应用中遇到的关键问题之上，着重讨论闪存碎片整理的操作、文件的添加与删除、TFS 文件的下载、文件压缩及除了 TFS 加载模型之外的执行能力。

7.7.1 带有安全电源的碎片整理操作

碎片整理操作只是在概念上将所有 TFS 存储空间中有用的文件移动到闪存地址空间的前一部分，并将已被删除的文件移动到闪存地址空间的后面部分。在重新整理文件之后，碎片整理操作将擦除最后一个有用文件之后的所有空间。该空间被

无效文件占据重新用于存储新的文件。碎片整理操作过程听起来很简单，但实现起来却比较复杂。在闪存碎片整理的任一时刻，重启或断开电源经常发生，启动时碎片整理运算规则需要恢复所有的文件，并且不能丢失在被中断的碎片整理操作之前在 TFS 中存在的任何文件。

下面将举一个 TFS 空间的例子。TFS 空间包括 7 个储存的扇区和 1 个备用扇区，假设所有的扇区是等大的（当然，这在实际中并不一定）。图 7.4 存在 4 个文件（F1，F2，F3，F4）、两个死区（已删除的文件）及少量可用空间和碎片整理管理程序所要求空间（备用扇区与状态存储）的 TFS 空间。

注意，相对的大小是不准确的（报头典型值大约为 92B，根据器件的不同，扇区的大小可以在 2~256KB 的范围内变化）。

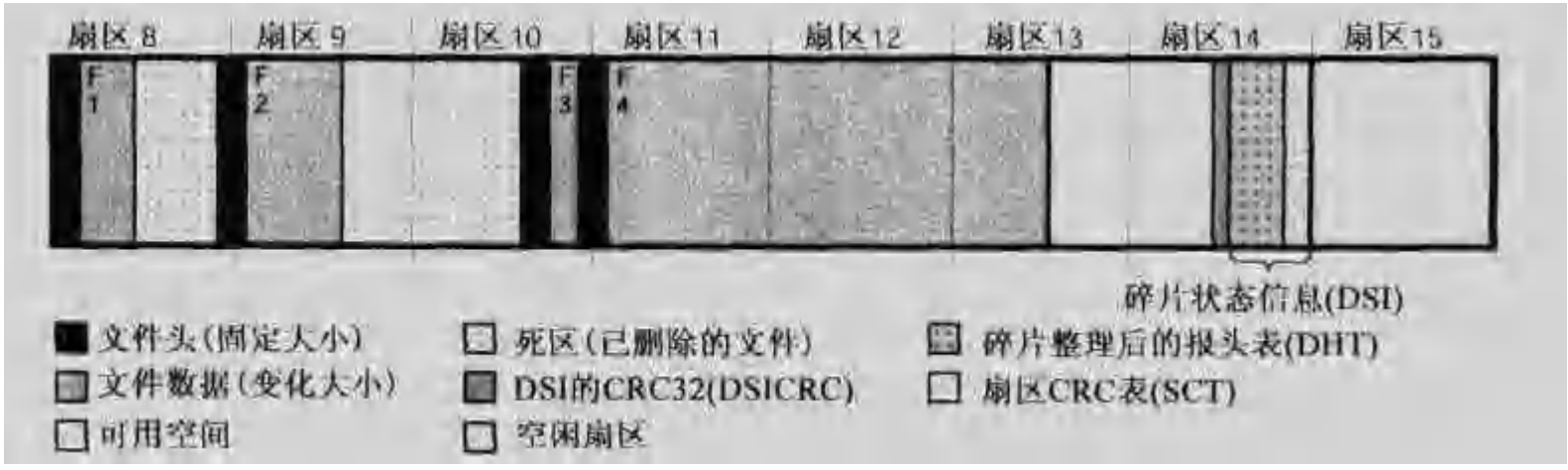


图 7.4　文件存储的结构举例

图 7.4 说明了在系统使用一段时间之后，TFS 空间的可能分布情况。标识为 F1、F2、F3 和 F4 的部分为有用文件。注意，图中存在一些与已删除文件有关的死区。

当加入的文件比可提供的空间大但小于包括死区在内的全部可以提供的空间时，需要进行碎片整理。其目标是将所有的文件推向左边（较低的地址空间），而将所有的死区推向右边（较高的地址空间）。这个过程可将死区转化为可用空间，从而允许 TFS 将新文件附加在线性链表之后。碎片整理的运算规则有以下 3 步：

（1）DSI 的创建——在扇区与有用文件当前状态的基础上创建碎片整理的状态信息（DSI：the Defragmentation State Information）。DSI 包括 3 个部分，即一个扇区的 CRC 表（SCT：a sector CRC table）、碎片整理后的报头表（DHT：a post-defragmentation header table）及两个表的 CRC（DSICRC：a CRC of the two tables）。

（2）文件的重新分布——从第一个 TFS 的扇区开始对每个扇区执行以下步骤：

- 检测本扇区内是否存在任何改变。如果没有，则处理下一扇区；
- 把将要被调整的扇区（有用扇区）的内容复制到备用扇区内；

- 擦除有用扇区；
- 利用 DHT 中的信息扫描每一个未被删除的文件，看它的一部分（文件头或内容）是否被重新定位到有用扇区。这个操作必须考虑从最后一个扇区或者从备用扇区（依据文件被定位前的位置）转换的文件。

（3）清除——清除所有被最后重新定位的文件之后的所有剩余空间（这些空间就变成了新的可以使用的空间的一部分），擦除备用空间。

在每个扇区中，SCT 包括两个 32 位的 CRC，大小由表里 TFS 空间所包括的所有扇区的数量决定。CRC 用来帮助被重新开始的碎片整理操作，并确定已被碎片整理的扇区，允许碎片整理从被中止的位置重新开始。DHT 包括每个未删除文件的报头，但并不是文件头，是碎片整理操作重新分配给每个文件在闪存内新位置所需要的信息。在闪存里 SCT 与 DHT 被建立之后，32 位的 CRC 作为检验 DSI 空间是否有效的检验机制被存放在 DSI 空间的底部。如果并不存在被擦除的区域而 CRC 的检验失败，那么 DSI 就会被认为正在生成中。

碎片整理的初始化程序是一段嵌套循环代码。外部循环将检验分配给 TFS 存储文件的每一个扇区，且在任一时刻，只有一个扇区起作用。外部循环的第一步用于判断扇区里的某部分是否将作为碎片整理的结果而被改变。如果没有，则外部循环将处理下一个扇区。在对有用的扇区做任何操作（闪存的写入操作）之前，它将被复制到备用扇区而后被擦除。这时，内部循环开始，对每一个 DHT 中的文件进行检验，判断在重新分配空间之后文件是否会覆盖有用的扇区。如果这种情况发生，那这部分文件将被复制到有用的扇区内。注意，这个被复制的部分可以是整个文件，也可以是文件的一部分。例如，报头与部分信息、部分的文件头或者是部分的信息有多少被复制，决定于文件重新分配后与有用扇区的重叠情况。

注意，复制的源可能是一个较靠后的文件（TFS 的某些在当前被激活的扇区之后的扇区）或者是备用扇区。如果重新分配的文件来自当前被激活的扇区位置，则源信息将被存放在备用扇区。图 7.5~图 7.10 为 TFS 结构重新分配操作的图示。

在如图 7.5 所示中，第一个扇区（sector 8）被复制到备用扇区且被擦除后，预定在碎片整理之后且位于第 8 扇区的的信息被复制回第 8 扇区。注意，这些信息的一部分来自备用扇区，而另一部分来自稍靠后的扇区。

图 7.6 对第 9 扇区重复了这一过程。在当前激活扇区里的信息被复制到备用扇区，当前激活扇区被擦除后，新内容被复制到当前激活扇区里。其一部分内容来自备用扇区，而另一部分来自稍靠后的扇区。

图 7.7 与前面大致相同。注意，第 10 扇区是复制到备用扇区的当前激活扇区，但是没有任何内容从备用扇区复制回当前激活扇区。因为第 10 扇区原始内容都没有被保留，实在是没有复制有用扇区的必要。图 7.8 和图 7.9 是通过相同的过程，

在被激活的第 11 扇区和第 12 扇区结束了碎片整理操作。

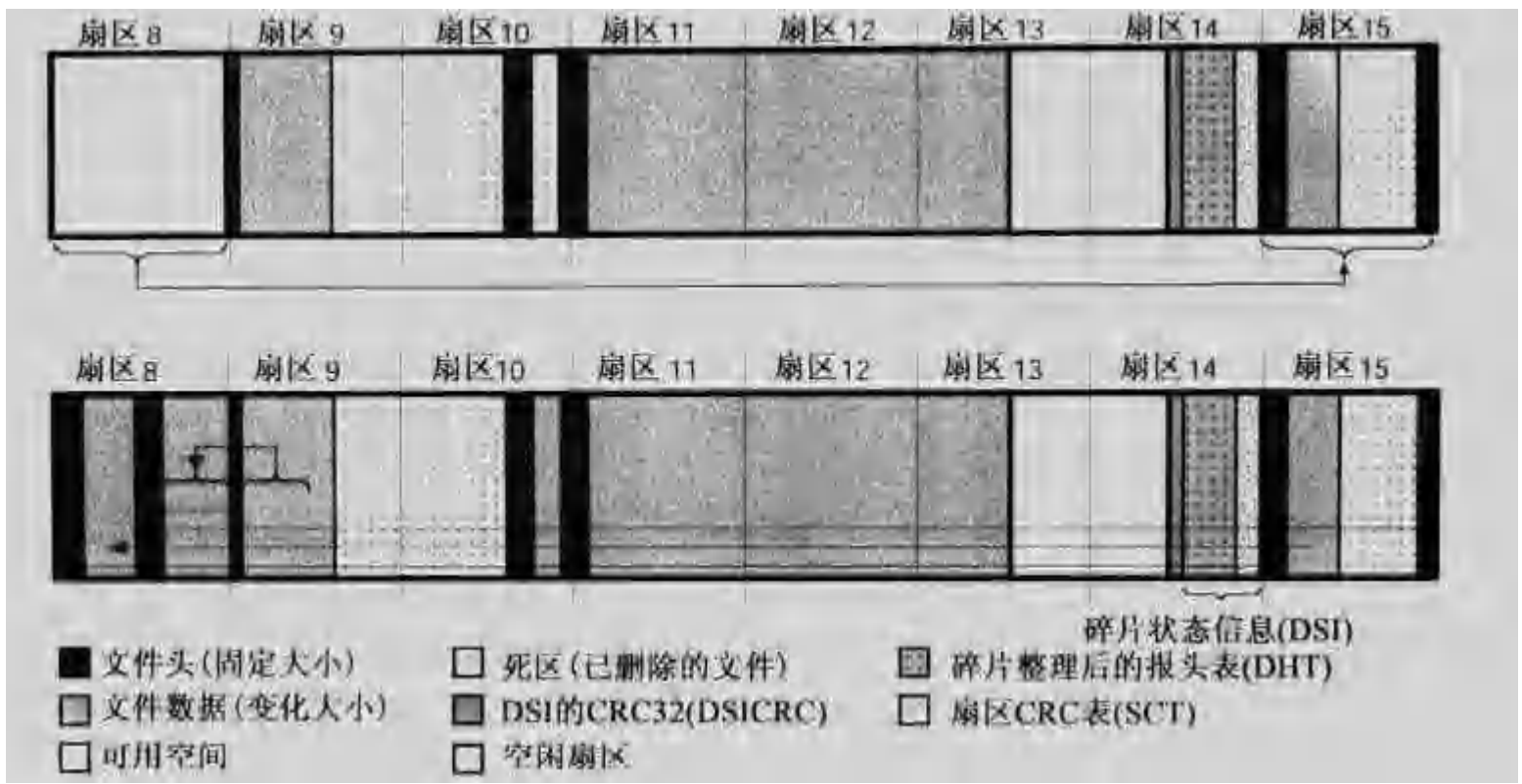


图 7.5　碎片整理操作当前激活扇区＝8

在碎片整理循环的第一个循环中，第一个扇区（sector 8）被复制到备用扇区，然后被一个紧密的图形重新填充。注意，紧凑的操作可以从文件系统的任意位置取得内容的材料。

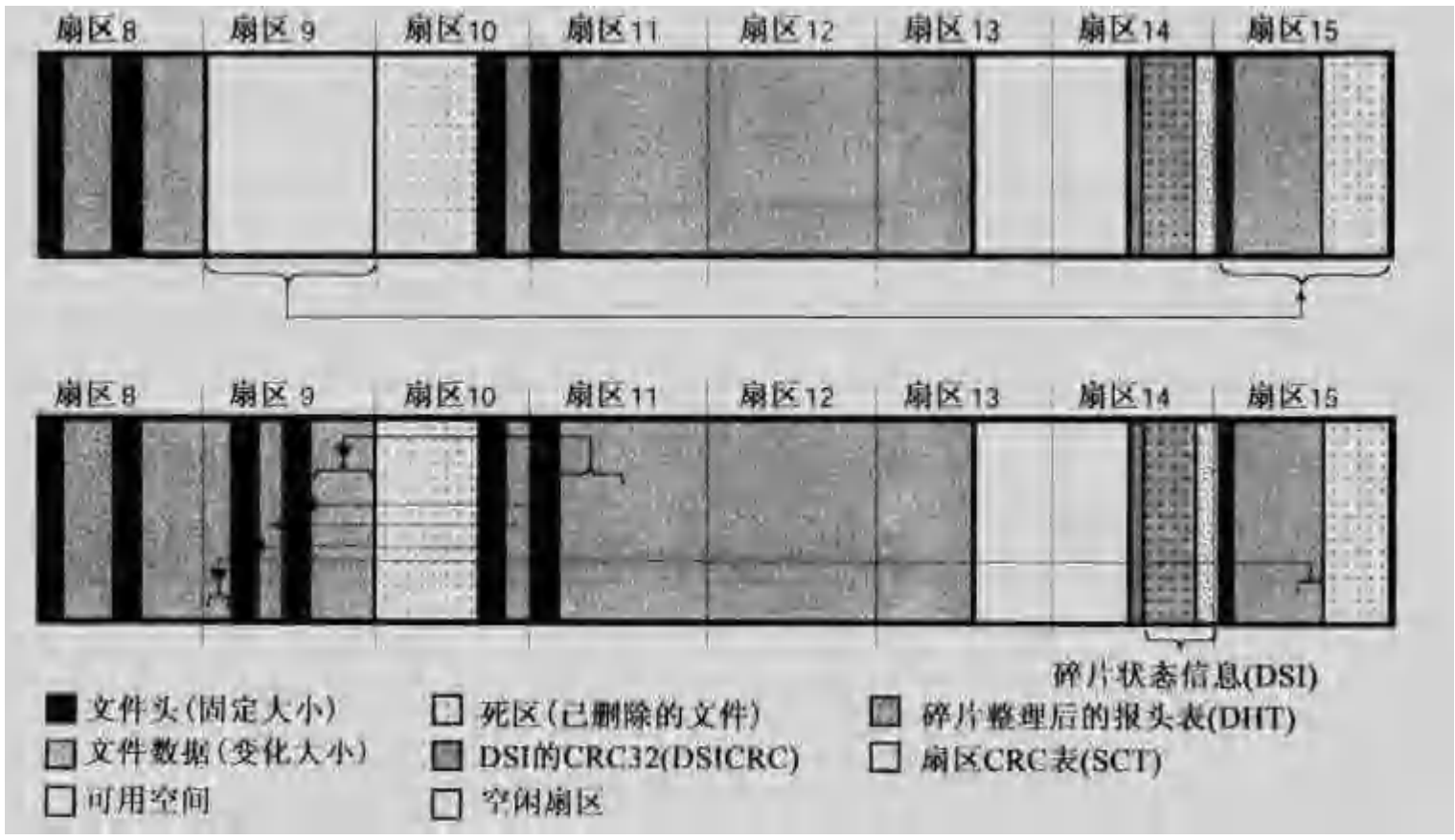


图 7.6　碎片整理操作当前激活扇区＝9

本图详细显示了对第 9 扇区的碎片整理的循环

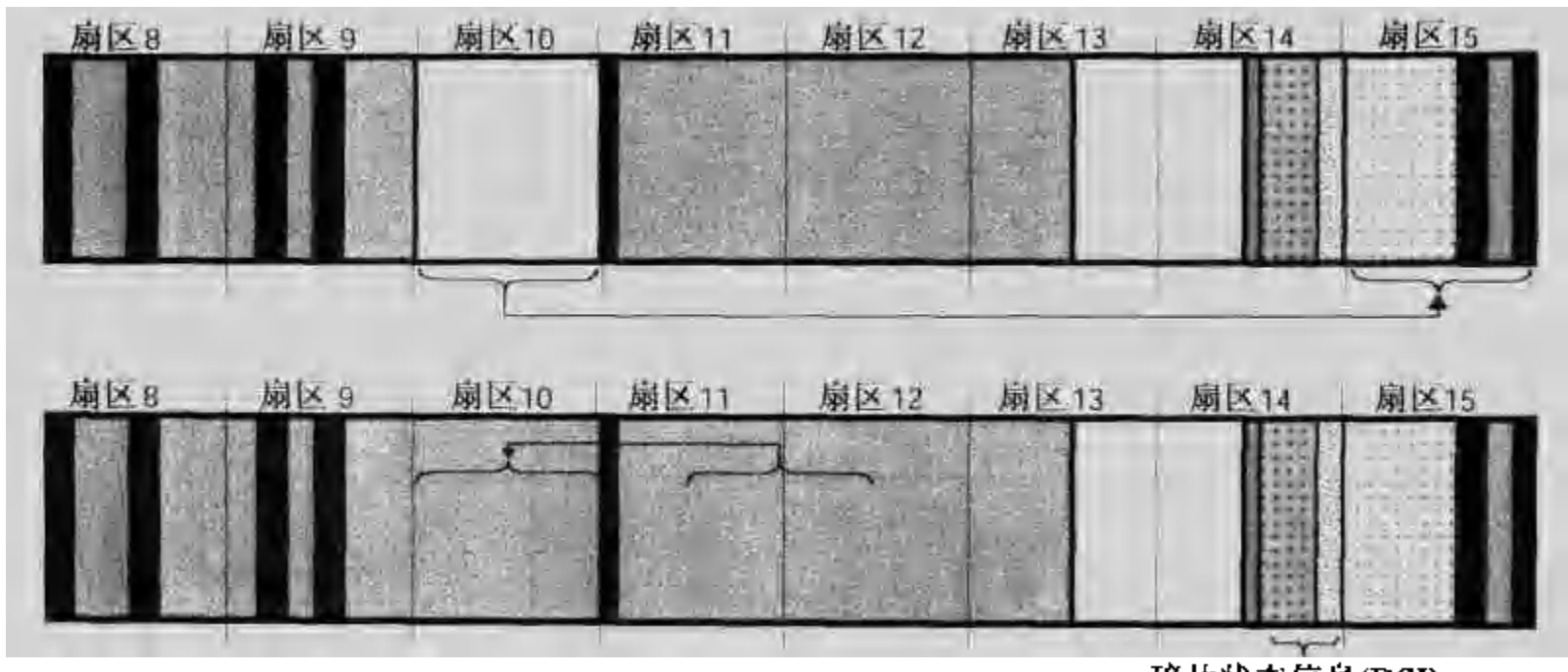


图 7.7 碎片整理操作当前激活扇区＝10

本图详细显示了对第 10 扇区的碎片整理的循环

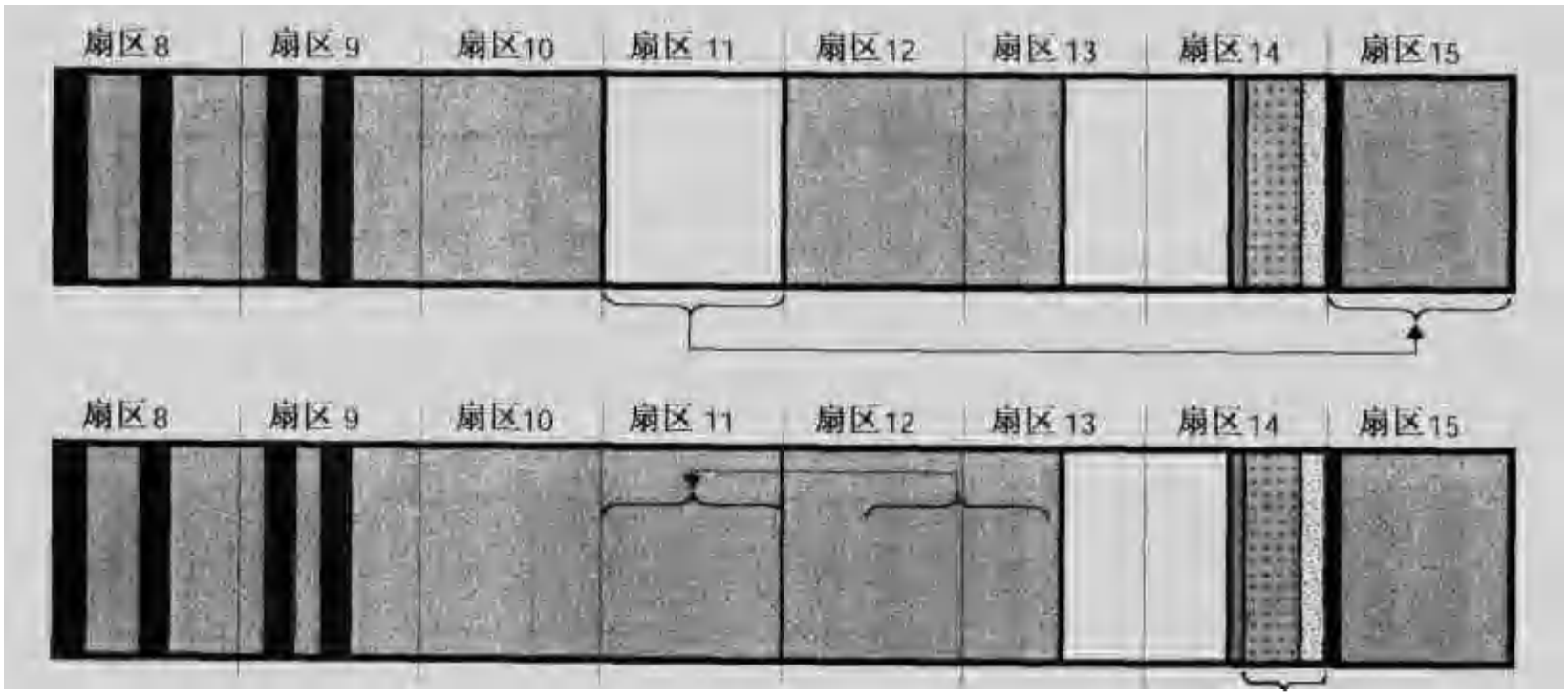


图 7.8 碎片整理操作当前激活扇区＝11

本图详细显示了对第 11 扇区的碎片整理的循环

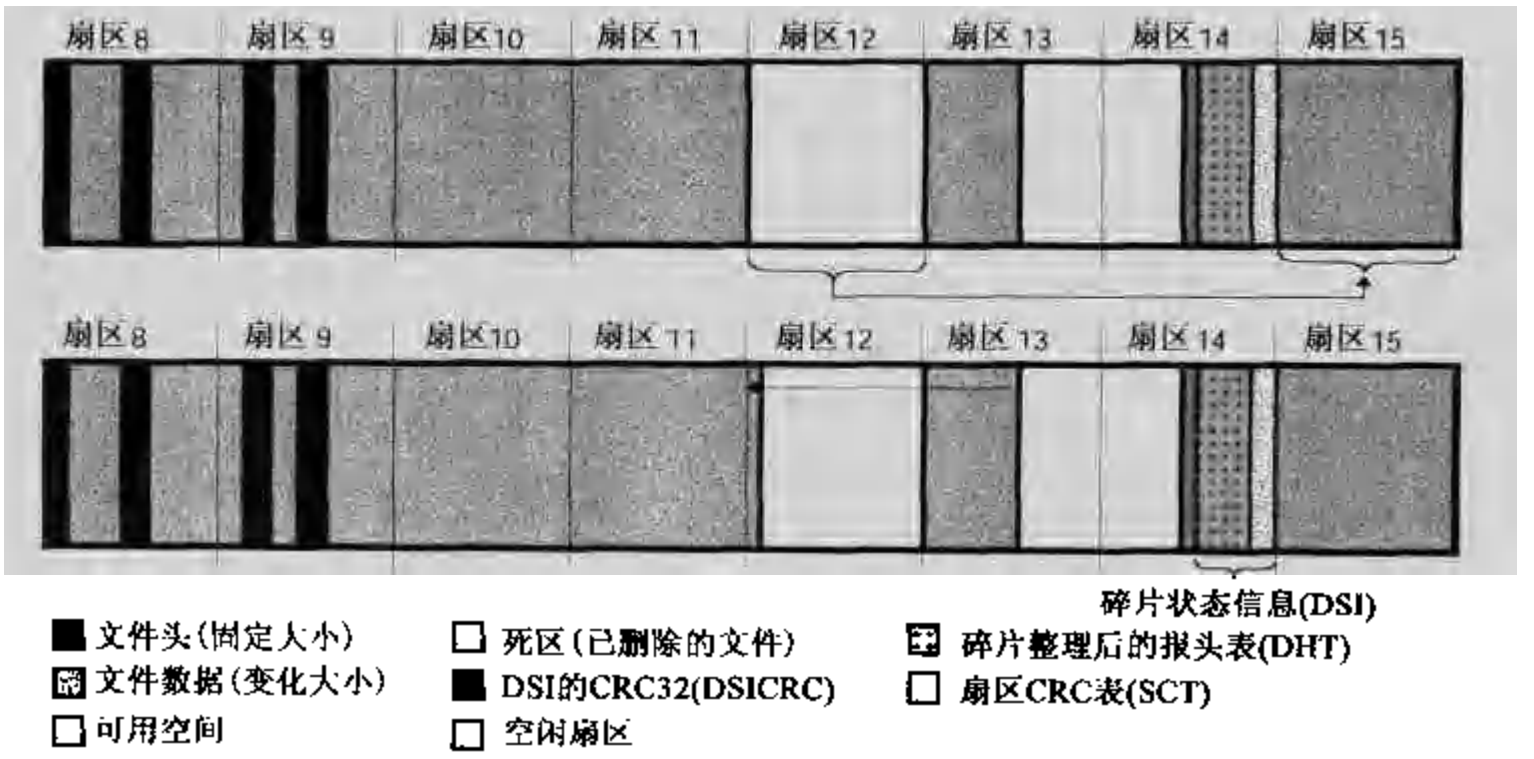


图 7.9　碎片整理操作当前激活扇区＝12

本图详细显示了对第 12 扇区的碎片整理的循环

在第 12 扇区被处理之后，所有保留的内容均非常清楚，第 13、14 和 15 扇区被擦除。图 7.10 显示了碎片整理操作的结果，其中的阴影部分表示了标识死区在概念上被转换的位置。注意，可用的空间大小明显增大了。

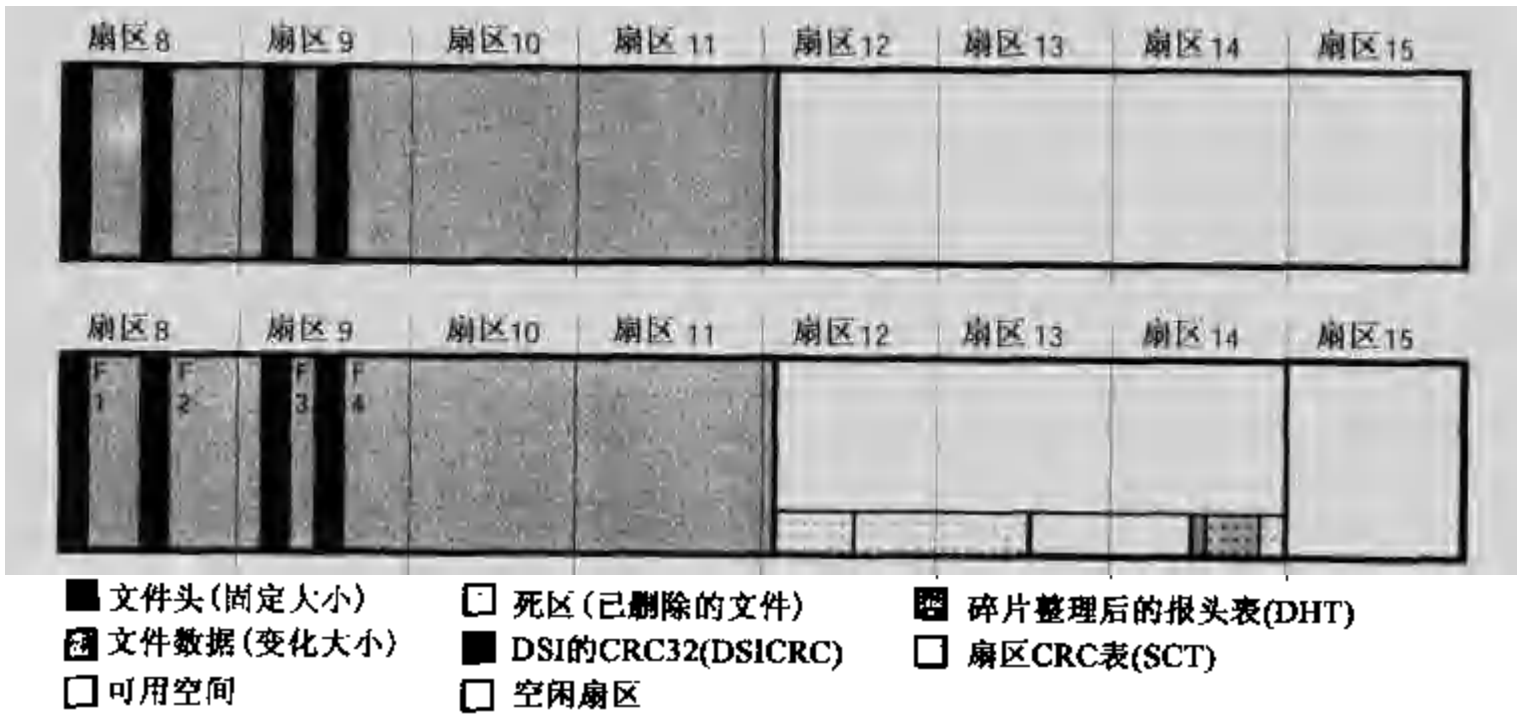


图 7.10　清除操作及碎片整理操作的完成

7.7.2　没有安全电源的 tfsclean（）函数

现在回到这个复杂问题的另一个方面，即前面提到的没有安全电源的碎片整理

操作的运算规则此时可以很容易地实现。完成碎片整理功能的函数是tfsclean()（见程序清单7.3），使用了一个变量和一个TDEV的结构体的指针。TDEV结构体告诉tfsclean()，哪一个闪存器件将要进行碎片整理（在TFS中可能存在多个闪存器件）。

程序清单7.3 tfsclean()

```
int
_tfsclean(TDEV *tdp)
{
    TFILE    *tfp;
    uchar    *tbuf;
    ulong    appramstart;
    int      dtot, nfadd, len, err;

    if (TfsCleanEnable < 0)
        return(TFSERR_CLEANOFF);

    appramstart = getAppRamStart();

/* Determine how many "dead" files exist. */
dtot = 0;
tfp = (TFILE *)tdp->start;
while(validtfshdr(tfp)) {
    if (!TFS_FILEEXISTS(tfp))
        dtot++;
    tfp = nextfp(tfp,tdp);
}

if (dtot == 0)
    return(TFS_OKAY);

printf("Reconstructing device %s with %d dead file%s removed...\n",
    tdp->prefix, dtot,dtot>1 ? "s":"");

tbuf = (char *)appramstart;
```

```
tfp = (TFILE *)(tdp->start);
nfadd = tdp->start;
while(validtfshdr(tfp)) {
    if (TFS_FILEEXISTS(tfp)) {
        len = TFS_SIZE(tfp) + sizeof(struct tfshdr);
        if (len % TFS_FSIZEMOD)
            len += TFS_FSIZEMOD - (len % TFS_FSIZEMOD);
        nfadd += len;
        err = tfsmemcpy(tbuf,(uchar *)tfp,len,0,0);
        if (err != TFS_OKAY)
            return(err);
        ((struct tfshdr *)tbuf) ->next = (struct tfshdr *)nfadd;
        tbuf += len;
    }
    tfp = nextfp(tfp,tdp);
}
```

```
    /* Erase the flash device: */
    err = _tfsinit(tdp);
    if (err != TFS_OKAY)
        return(err);

    /* Copy data placed in RAM back to flash: */
    err = AppFlashWrite((ulong *)(tdp->start),(ulong *)appramstart,
        (tbuf- (uchar*)appramstart));
    if (err < 0)
        return(TFSERR_FLASHFAILURE);

    return(TFS_OKAY);
}
```

这个函数首先检验碎片整理操作是否因某种原因而被中止（见程序清单 7.3）。如果被中止，则返回值为 TFSERROR_CLEANOFF。复制必须拥有可以连接所有文件的 RAM 空间。这里，我们假设在使用微监控器 .bss 的 RAM 空间之后，函

数 getAppRamStart（）返回了指向这部分 RAM 的起始地址的指针。假定函数 tfsclean（）有足够的 RAM 空间，可以满足整个 TFS 闪存器件被复制（应用闪存图时，必须将这些可能性考虑在内）。程序扫描确定的器件文件列表，检测是否存在已被删除的文件。如果没有，便没有必要进行任何的清除操作，而只需直接、快速、成功地返回函数值。

第二个 while（）循环进行了实际的复制和链接操作。在 appramstart 开始之后，每一个现行的文件紧密相接地被复制回 RAM 空间。注意，在被复制的报头里，每一个 TFS 报头的 next 指针被更新，新的报头可以准确地标识不同闪存空间内的文件情况。在链接操作完成之后，通过调用 tfsinit（）函数，使用相同的传递给 tfsinit（）函数的 TDEV 结构体指针擦除分配给 TFS 文件存储的闪存空间。最后，闪存已经被擦除，并将所有连接的文件从 RAM 中复制回闪存器件的底部。

7.8 增加和删除文件

开始只是简单地将另外的文件头和文件信息块附加到当前闪存内文件链表的末尾。在理想的情况下是这样的，但实际上却复杂得多。本节将讨论从 TFS 中增加［利用 tfsadd（）函数］和删除［利用 tfsunlink（）函数］的操作。

7.8.1 tfsadd（）函数

TFS 将一个文件增加到文件系统中要执行以下几步：

（1）判断是否已有重名的文件存在 TFS 中：

- 如果有，并且已经存在的文件内容与新文件相同，则返回；
- 如果新文件内容与原文件不同，则原文件被标识为“stale”，继续进行增加新文件的操作；
- 如果没有，则将继续下一步。

（2）对新文件进行 32 位的 CRC 操作。

（3）判断是否有足够的空间存储文件：

- 如果有，则继续下一步；
- 如果没有，则对 TFS 的闪存空间进行碎片整理。如果在碎片整理之后有足够的空间，则进行下一步；否则，返回已经没有足够空间的提示。

（4）对新文件第二次运行 32 位的 CRC。如果第一次与第二次运行结果不同，便返回一个错误提示（两次 CRC 检测均要通过的原因将在后面详细论述）。

（5）将文件转换到闪存中，建立新的 TFS 文件头，并将其复制到闪存中。

（6）如果一个文件在前面第一步被标识为 “stale”，则将其标识为已删除，最终完成所有步骤。

接下来的几个列表（见程序清单 7.4~程序清单 7.13）是 tfsadd（）函数的部分代码。这个函数处理了所有增加一个新文件到 TFS 的复杂情况。参数 name、info 和 flags 均为字符串类型（见程序清单 7.4）。变量 name 为表示文件名的字符串；字符串 info 为可选项，用来描述这个文件；参数 flags 是 ASCII 码的字符串，描述了将要适用于这个文件的性质。注意，tfsadd（）缺省时，认为 info 与 flags 为空，剩下未讨论的是 src 和 size。（argument）src 是一个指向实际数据的指针，这些数据将组成 TFS 储存空间数据部分。size 是储存的字节数。

程序清单 7.4　tfsadd()

```
int
tfsadd(char *name, char *info, char *flags, uchar *src, int size)
{
    TDEV        *tdp;
    TFILE       *fp, tf;
    ulong       endoftfsflash, nextfileaddr, state_table_overhead;
    ulong       crc_pass1, crc_pass2, bflags;
    int         ftot, cleanupcount, err, stale, ssize;

    if (!info) info = "";
    if (!flags) flags = "";
```

注意

用户可以跟踪一下 TFS 的某些函数，设置一下 tfsTrace 变量以便能够跟踪（见程序清单 7.5）。可跟踪的结果是 tfsadd（）函数被调用在用户可以观察的控制台上。

程序清单 7.5　跟踪特性

```
if (tfsTrace > 0)
    printf("tfsadd(%s,%s,%s,0x%lx,%d)\n",
        name,info,flags,(ulong)src,size);
```

注意

TFS 包括了…个“perror-like”函数。该函数允许用户将一个返回的错误代码转变为某些错误信息。错误信息可以返回某些特殊的代码，并传送回另外的TFS API函数，以便将其扩展为更为详细地描述。程序清单 7.6 的错误检查说明了这一特点。

程序清单 7.6 接近末尾处调用 tfsFileIsOpened（）的执行，是为了使正在被执行的脚本不会被某些不知道的任务突然删除。

程序清单 7.6　检查操作的有效性

```
/* Check for valid size and name: */
if ((size <= 0) || (!name))
    return(TFSERR_BADARG);

/* If name or info field length is too long, abort now... */
if ((strlen(name) > TFSNAMESIZE) ||
    ((info) && (strlen(info) > TFSINFOSIZE)))
    return(TFSERR_NAMETOOBIG);

/* If the file is currently opened, then don't allow the add... */
if (tfsFileIsOpened(name))
    return(TFSERR_FILEINUSE);
```

程序清单 7.7　处理特性标志

```
/* If incoming flags are illegal, abort now... */
if (*flags == 0) {
    bflags = 0;
}
else {
    err = tfsflagsatob(flags,&bflags);
    if (err != TFS_OKAY)
        return(err);
}
```

字符串 flags 作为被传送进来的参数必须被转化为位的区段（实际实现为无符号的长整型），确定的位代表在字符串 flags 特殊的字符（我们承认以字符形式输入 flags 使用户更容易在 CLI 中指定一系列的标志）。函数 tfsflagsatob（）的调用建立了这个位区段，并且如果转换失败，则返回一个错误值。

程序清单 7.8　计算初始化的校验和

```
/* Take snapshot of source crc. */
crc_pass1 = crc32(src, size);
```

在开始复制到闪存之前，程序将计算输入数据的 32 位的 CRC。实际上，这个 CRC 被计算了两次，因此 tfsadd（）可以检测到源内容的改变。这个步骤的优点为：

- 如果输入的文件有所改变，则有可能是源地址错误。
- 输入的指针不应该指向 TFS 空间的原始信息。如果输入的指针指向了 TFS 空间的原始信息，并且碎片整理程序运行了，则 TFS 中的信息已经转换。

在复制开始之前，输入的信息检测 TFS 的多个闪存器件列表（见程序清单 7.9）。如果输入名字的前缀与某一个器件相符合，TFS 便认为这个文件将被加入相关的闪存器件内。

程序清单 7.9　识别闪存器件

```
/* Establish the device that is to be used for the incoming file
 * addition request... The device used depends on the prefix of
 * the incoming file name. If the incoming prefix doesn't match
 * any of the devices in the table, then place the file in the
 * first device in the table (assumed to be the default).
 */
for(tdp=tfsDeviceTbl;tdp->start != TFSEOT;tdp++) {
    if (!strncmp(name,tdp->prefix,strlen(tdp->prefix)))
        break;
}
if (tdp->start == TFSEOT)
    tdp = tfsDeviceTbl;
```

程序清单 7.10　搜索最后的报头

```
tryagain:
    fp = (TFILE *)tdp->start;
```

```
/* Find end of current storage: */
ftot = 0;
while (fp) {
    if (fp->hdrsize == ERASED16)
        break;
    if (TFS_FILEEXISTS(fp)) {
        ftot++;
        if (fp->flags & TFS_NSTALE) {
            if (!strcmp(TFS_NAME(fp),name)) {
            /* If file of the same name exists AND it is identical to
             * the new file to be added, then return TFS_OKAY and be
             * done; otherwise, remove the old one and continue.
             * Don't do the comparison if src file is in-place-modify
             * because the source data is undefined.
             */
                if ((!(bflags & TFS_IPMOD)) &&
                    (!tfscompare(fp,name,info,flags,src,size)))
                    return(TFS_OKAY);

            /* If a file of the same name exists but is different than
             * the new file, make the current file stale, then after
             * the new file is added we will delete the stale one.
             */
                stale = 1;
                err = tfsmakeStale(fp);
                if (err == TFS_OKAY)
                    goto tryagain;
                else
                    return(err);
            }
        }
    }
    fp = nextfp(fp,tdp);
}
```

```
if (!fp)      /* If fp is 0, then nextfp() (above) detected corruption. */
    return(TFSERR_CORRUPT);
```

TFS 通过搜索相关的闪存器件来寻找存储在这个器件上的最后一个文件（这样它才可以将新文件附加至当前链表）。搜索操作包括浏览链表。在这个过程中，TFS 同时检验每一个文件头是否完全。函数 nextfp（）的调用完成了某些基本的检验功能。如果 TFS 发现了最后的报头或者返回 TFS_CORRUPT，则标志的报头是错误的。如果被添加的文件早已存在于 TFS 中（名称相同），就需要进行附加的步骤了。

- 如果文件存在且内容、标志与添加的文件信息域相同，则程序立刻返回。因为两个文件图示相同，忽略将其复制至闪存没有任何影响。实际上，这样做可避免闪存的擦除和程序的循环，从而延长了器件的寿命。
- 如果文件不相同，那么现存的文件被标识为 STALE。这个文件将最终被新文件所代替，但需要在程序中止或直到新的文件在实际删除现存的文件后才能成功地被复制。
- 如果文件并不存在，则新文件直接被添加，就不需要前面所列举的复杂操作了。

当 TFS 找到当前文件链表的末尾时，它必须判断是否有足够的空间来加入新的文件（见程序清单 7.11）。如果存在足够的空间，那这个新文件便被附加到链表上。如果不存在足够的空间，则附加的程序便开始了碎片整理的循环。

计算可以使用的文件空间需要考虑碎片整理管理费用。这个管理费用包括对备用空间、碎片整理报头和扇区 CRC 表的空间（这两者在 TFS 空间的结束处开始构造）的管理。当碎片整理完成之后，TFS 第二次扫描链表，发现结束点并判断是否存在足够的空间。如果存在，则继续；反之，则程序返回 TFSERR_FLASHFULL。

程序清单 7.11　检测空闲的空间

```
/* Calculate location of next file (on mod16 address). This will be
 * initially used to see if we have enough space left in flash to store
 * the current request; then, if yes, it will become part of the new
 * file's header.
 */
nextfileaddr = ((ulong)(fp+1)) + size;
if (nextfileaddr & 0xf)
    nextfileaddr = (nextfileaddr | 0xf) + 1;

/* Make sure that the space is available for writing to flash...
```

```
 * Remember that the end of useable flash space must take into
 * account the fact that some space must be left over for the
 * defragmentation state tables. Also, the total space needed for
 * state tables cannot exceed the size of the sector that will contain
 * those tables.
 */
state_table_overhead = ((ftot+1) * DEFRAGHDRSIZ) +
      (tdp->sectorcount * sizeof(long));

if (addrtosector((uchar *)(tdp ->end),0,&ssize,0) < 0)
      return(TFSERR_MEMFAIL);

if (state_table_overhead >= (ulong)ssize)
      return(TFSERR_FLASHFULL);

endoftfsflash = (tdp ->end + 1) - state_table_overhead;

if ((nextfileaddr >= endoftfsflash) ||
      (!tfsSpaceErased((uchar *)fp,size+TFSHDRSIZ))) {
      if (!cleanupcount) {
            err = tfsautoclean(0,0,0,0,tdp,0,0);
            if (err != TFS_OKAY) {
                  printf("tfsadd autoclean failed: %s\n",
                        (char *)tfsctrl(TFS_ERRMSG,err,0));
                  return(err);
            }
            cleanupcount++;
            goto tryagain;
      }
      else
            return(TFSERR_FLASHFULL);
}
```

程序清单 7.12　测试稳定的的信息

```
/* Do another crc on the source data. If crc_pass1 != crc_pass2 then
 * somehow the source is changing. This is typically caused by the fact
 * that the source address is within TFS space that was automatically
 * defragmented above. Since we are aborting the creation of the file,
 * we must undo any stale file that may have been created above.
 * No need to check source data if the source is in-place-modifiable.
 */
if (!(bflags & TFS_IPMOD)) {
    crc_pass2 = crc32(src,size);
    if (crc_pass1 != crc_pass2) {
        if (stale)
            tfsstalecheck(0);
        return(TFSERR_FLAKEYSOURCE);
    }
}
else
    crc_pass2 = ERASED32;
```

现在开始进行闪存的写入操作。下一步是对输入的文件进行第二次 CRC（见程序清单 7.12）。如果 CRC 不能满足，则程序将返回标识输入信息不稳定的错误。

在实际写入操作的准备过程中，函数 tfsadd（）在局部结构体 tf 中（见程序清单 7.13）创建了新的文件头。在新的文件头被创建之后，输入文件的数据部分被复制到闪存中，最后文件头也被复制到闪存中。文件数据在文件头之前被复制，因此文件头的操作是闪存最后一次的写入操作，它可以更容易地检测到被中断的写入操作。在新文件被成功复制之后，tfsadd（）函数进行了又一次的 CRC，检测闪存复制是否成功。这时，TFS 便可以删除所有剩余的旧文件。最后一步是调用 tfslog（），如果成功的话，则将记录下实际对闪存进行调整所有的文件操作。

程序清单 7.13　创建报头

```
memset((char *)&tf,0,TFSHDRSIZ);


/* Copy name and info data to header. */
strcpy(tf.name, name);
strcpy(tf.info, info);
```

```
tf.hdrsize = TFSHDRSIZ;
tf.hdrvrsn = TFSHDRVERSION;
tf.filsize = size;
tf.flags = bflags;
tf.flags |= (TFS_ACTIVE | TFS_NSTALE);
tf.filcrc = crc_pass2;
tf.modtime = tfsGetLtime();
tf.next = 0;
tf.hdrcrc = 0;
tf.hdrcrc = crc32((uchar *)&tf,TFSHDRSIZ);
tf.next = (TFILE *)nextfileaddr;

/* Now copy the file and header to flash.
 * Note1: the header is copied AFTER the file has been
 * successfully copied. If the header was written successfully,
 * then the data write failed, the header would be incorrectly
 * pointing to an invalid file. To avoid this, simply write the
 * data first.
 * Note2: if the file is in-place-modifiable, then there is no
 * file data to be written to the flash. It will be left as all FFs
 * so that the flash can be modified by tfsipmod() later.
 */

/* Write the file to flash if not TFS_IPMOD: */
if (!(tf.flags & TFS_IPMOD)) {
    if (tfsflashwrite((ulong *)(fp+1),(ulong *)src,size) == -1)
        return(TFSERR_FLASHFAILURE);
}

/* Write the file header to flash: */
if (tfsflashwrite((ulong *)fp,(ulong *)(&tf),TFSHDRSIZ) == -1)
    return(TFSERR_FLASHFAILURE);

/* Double check the CRC now that it is in flash. */
```

```
    if (!(tf.flags & TFS_IPMOD)) {
        if (crc32((uchar *)(fp+1), size) != tf.filcrc)
            return(TFSERR_BADCRC);
    }

    /* If the add was a file that previously existed, then the stale flag
     * will be set and the old file needs to be deleted...
     */
    if (stale) {
        err = _tfsunlink(name);
        if (err != TFS_OKAY)
            printf("%s: %s\n",name,tfserrmsg(err));
    }

    tfslog(TFSLOG_ADD,name);
    return(TFS_OKAY);
}
```

7.8.2　tfsunlink（）函数

由于惟一输入的参数是将要被删除的文件名称，因此文件删除函数 tfsunlink（）（见程序清单 7.14）并不像 tfsadd（）函数那样要求错误检查。类似于 tfsadd（）函数，tfsunlink（）应用了跟踪机制，并检测文件当前是否处于打开状态。删除的代码将通过调用 tfsstat（）函数重新得到指向文件的指针。如果 tfsstat（）返回值为 NULL，tfsunlink（）则认为 TFS 中不存在与输入文件相同名字的文件且返回相应的错误代码。如果 tfsstat（）返回了一个指针，那么文件头的 TFS_ACTIVE 位被清除，标识文件已被删除。最后一步，调用 tfslog（）函数来记录闪存的处理。

程序清单 7.14　tfsunlink()

```
int
tfsunlink(char *name)
{
    TFILE *fp;
```

```
    ulong flags_marked_deleted;

    if (tfsTrace > 0)
        printf("_tfsunlink(%s)\n",name);

    /* If the file is currently opened, then don't allow the deletion... */
    if (tfsFileIsOpened(name))
        return(TFSERR_FILEINUSE);

    fp = tfsstat(name);
    if (!fp)
        return(TFSERR_NOFILE);

    if (TFS_USRLVL(fp) > getUsrLvl())
        return(TFSERR_USERDENIED);

    flags_marked_deleted = fp->flags & ~TFS_ACTIVE;
    if (tfsflashwrite((ulong *)&fp->flags,
        &flags_marked_deleted,sizeof(long)) < 0) {
        return(TFSERR_FLASHFAILURE);
    }

    tfslog(TFSLOG_DEL,name);
    return (TFS_OKAY);
}
```

7.9 加载的应用

TFS 支持两种可执行的文件，即二进制和脚本文件。脚本文件是包括单线程 CLI 命令的简单 ASCII 码文件。作为这种文件的写入操作，可执行的二进制文件可以是 ELF、COFF 和 AOUT 格式。这三种格式非常类似，都包含一个文件头、扇区头和一系列扇区。与可执行的脚本文件不同，在将格式化后的扇区从 TFS 的存储空间复制到 RAM 的空间之后，二进制的文件被执行。接着，程序在 RAM 外运行。

这个过程通常叫做文件的加载。在工作站和服务器型的操作系统中，加载是一种很普遍的操作，因为在这种操作环境下，CPU 从磁盘上读取数据是物理上不可实现的。嵌入式系统的应用不存在磁盘，但闪存的应用与它们不同。对 CPU 来说，闪存为本地存储器，意味着嵌入式系统的处理器可以直接从闪存中应用程序（事实上，微监控器确实是在闪存外直接执行）。这样，嵌入式系统监控器无论是加载一个应用程序还是直接从文件系统中执行，仅取决于设计者的选择。在此我们将从两方面来权衡考虑。

加载操作的优点包括：

- 不需要重新建立文件格式；
- 通过不重复的绝对地址空间创建多个应用程序，这样可以在相同的目标区域运行它们；
- 目标区域的 RAM 空间可以稍大，从而更有效地获取指令；
- 利用可写的存储空间的指令可以很容易地在指令序列里加入中断（或断点）；
- 将可执行的程序压缩，从而节约闪存空间。

另一方面，加载也存在明显的缺陷：将文档和初始化后的数据段从闪存转移到 RAM 中需要一定的时间和储存空间。

实际的选择取决于应用的特点和所拥有的储存空间。因为我们建立了一个在应用程序和闪存之间存在缓冲区的操作平台（允许应用程序将闪存空间作为命名空间而不是地址空间），且早已接受为了提供一个更容易维护的平台而加入管理程序的事实。因此，我们设计了用 TFS 来支持给加载操作。

通常我们认为，为了维护能力而牺牲效率是值得的。在 TFS 的设计过程中，在二者的取舍之间我们选择了维护能力。

程序清单 7.15 表述了 ELF 文件格式加载的代码片断。程序清单 7.15 的参数如下所示：

- fp 是指向于 ELF 文件相联系的 TFS 文件头结构体的指针；
- verbose 决定是否添加额外的报告；
- entrypoint 是一个指向 long 型的指针，如果非零的话，则将被 ELF 文件的插入点的地址所装载；
- verifyonly 是一个标志位，它告诉这个函数应该只比较早已加载和即将加载的文件。

注意，程序清单 7.15 要求几种不同等级的报头。TFS 报头与 ELF 本身并没有关系，只是用来管理闪存中文件储存的报头。TFS 中的每个文件包括两部分，即文件头与数据。数据是一个 ELF 文件，而 ELF 文件包括另一系列的报头。函数必须

处理这两种不同的报头，有时可能会带来一定理解上的困难。

程序清单 7.15 ELF 文件格式装载器

```
/* tfsloadelf():
 *  The file pointed to by fp has been determined to be an ELF file.
 *  This function loads the sections of that file into the designated
 *  locations.
 *  Caches are flushed after loading each loadable section.
 */
int
tfsloadebin(TFILE *fp,int verbose,long *entrypoint,int verifyonly)
{
    Elf32_Word   size, notproctot;
    int          i, err;
    char         *shname_strings;
    ELFFHDR *ehdr;
    ELFSHDR *shdr;

/* Establish file header pointers... */
ehdr = (ELFFHDR *)(TFS_BASE(fp));
shdr = (ELFSHDR *)((int)ehdr + ehdr->e_shoff);
err = 0;

/* Verify basic file sanity... */
if ((ehdr->e_ident[0] != 0x7f) || (ehdr->e_ident[1] != 'E') ||
    (ehdr->e_ident[2] != 'L') || (ehdr->e_ident[3] != 'F'))
    return(TFSERR_BADHDR);

/* Store the section name string table base: */
shname_strings = (char *)ehdr + shdr[ehdr->e_shstrndx].sh_offset;

notproctot = 0;

/* For each section header, relocate or clear if necessary... */
```

```
for (i=0;!err && i<ehdr->e_shnum;i++,shdr++) {
    if ((size = shdr->sh_size) == 0)
        continue;

    if ((verbose) && (ehdr->e_shstrndx != SHN_UNDEF))
        printf("%-10s: ", shname_strings + shdr->sh_name);

    if (!(shdr->sh_flags & SHF_ALLOC)) {
        notproctot += size;
        if (verbose)
            printf("        %7ld bytes not processed (tot=%ld)\n",
                size,notproctot);
        continue;
    }
    if (shdr->sh_type == SHT_NOBITS) {
        if (tfsmemset((char *)(shdr->sh_addr),0,size,
                verbose,verifyonly) != 0)
            err++;
    }
    else {
        if (TFS_ISCPRS(fp)) {
            int         outsize;

            outsize = decompress((char *)(ehdr)+shdr->sh_offset,
                        size,(char *)shdr->sh_addr);
            if (outsize == -1) {
                err++;
                continue;
            }
            if (verbose)
                printf("dcmp %7d bytes from 0x%08lx to 0x%08lx\n",
                    outsize,(ulong)(ehdr)+shdr->sh_offset,
                    shdr->sh_addr);
        }
```

```
                else {
                    if (tfsmemcpy((char *)(shdr->sh_addr),
                        (char *)((int)ehdr+shdr->sh_offset),
                        size,verbose,verifyonly) != 0)
                        err++;
                }
                /* Flush caches for each loadable section... */
                flushDcache((char *)shdr->sh_addr,size);
                invalidateIcache((char *)shdr->sh_addr,size);
            }
        }

        if (err)
            return(TFSERR_MEMFAIL);

        if (verbose & !verifyonly)
            printf("entrypoint: 0x%lx\n",ehdr->e_entry);

        /* Store entry point: */
        if (entrypoint)
            *entrypoint = (long)(ehdr->e_entry);

        return(TFS_OKAY);
    }
```

程序清单 7.15 开始时为 ELF 文件头结构（ehdr）建立的结构格式。块头指针（shdr）是建立在从文件头位置开始的偏移量（e_shoff）的基础上。

指针初始化之后的测试需通过确认头部是否包含 ELF 标志的方法来实现一些基本格式的确认（而不是陷入 ELF 本身的很多细节，shname_strings 是另一个文件更深处的偏移，指向一个字符串表。指针帮助使用者详细列出了不同的块名，使用者可以通过这些块指针来进行分析）。

下面的循环驱动了对 ELF 文件块的处理。块名称被立即标志出来，SHF_ALLOC 标志被立即检测。这个标志指出块是否有与执行的图像（程序）直接联系的部分。例如，一个.text 块之所以会有这个标志，是因为.text 块包含对 CPU 的指令。而一

个符号表的块之所以不会有此标志，是因为它与实际处理图像没有任何关系。在这些附加检查之后，块不仅被拷贝到 RAM，而且如果 sh-type 被设成 SHF_NOBITS，则存储器空间就被清空。

注意 TFS_ISCPRS(fp)宏。TFS_ISCPRS(fp)是对 TFS 惟一的 ELF 文件格式的扩展。ELF 文件中的每一块都可以被压缩，是微监控器允许的。因此，当宏返回时，装入程序把从闪存中解压出的数据装入目标存储器空间（下一部分更详细地叙述了这一特征）。

每当一次拷贝操作完成时，装载程序就会强行冲掉 D 缓存并使 I 缓存无效。这些缓存操作是必需的，因为当装载程序把.text 块从存储器中的一点拷贝到另一点时，实质上是把指令从一点拷贝到另一点。拷贝过程中，指令就像给 CPU 的数据，所以如果 CPU 有数据和指令缓存，指令就有可能放在数据缓存中。如果缓存没有在 CPU 要执行刚拷贝的代码之前被冲掉或置无效，可能就会产生严重错误，因为指令是在数据缓存中而不是物理存储器中。

一旦拷贝循环完成装载程序储存进入的位置，就可返回。

7.10　文件解压缩

如前所述，一个典型的可执行操作可能是 COFF、ELF 或 AOUT。这些文件格式都含有多个文字和数据块，它们必须在操作执行的地方从闪存空间拷贝到 RAM/DRAM 空间。TFS 通过允许文件中个别块被压缩的方式，扩展了这些文件的一般格式。默认情况下，可执行的是未压缩的，数据从闪存传到 RAM。

TFS 支持一块一块地压缩，这样就允许了块头部保持未压缩，即使文件剩下的部分都被压缩。如果压缩时选了整体文件（全有或全无的）选项，则整个文件需要被解压到 RAM 中来提取块信息——只有当可以从块中提取信息时，才可以在装入的位置重新拷贝一次。

可以选择只压缩文件中个别的块，这种一次一块地压缩留下了所有可运行的有效重定位信息，也证明了这种压缩的优点——避免了整个文件被压缩所需要的双存储器拷贝。

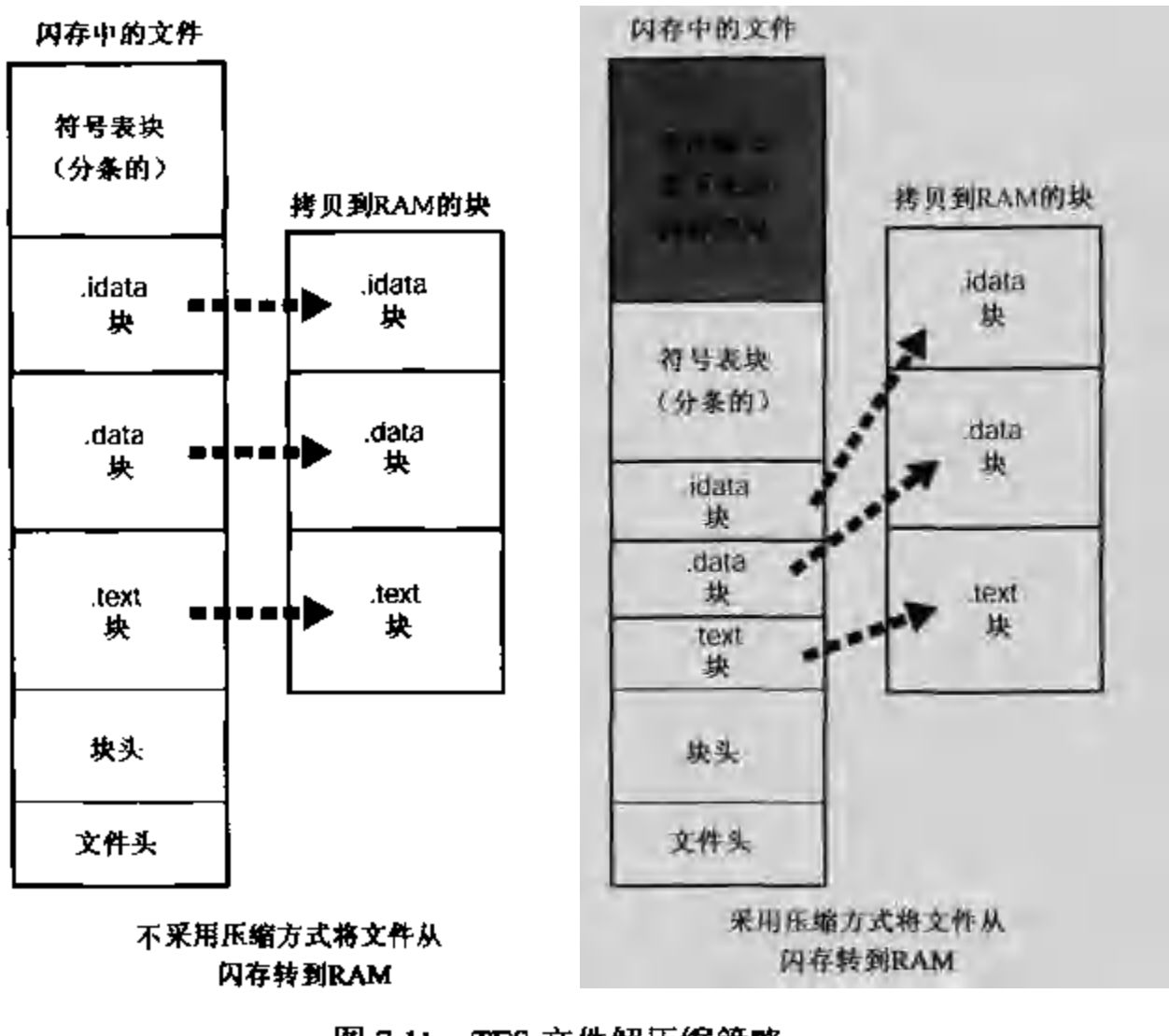


图 7.11 TFS 文件解压缩策略

在文件的格式下，文件中所有的块被压缩。这使 ELF 格式保持完整，但是仍然支持压缩图像。

7.11 现场执行

前面两部分讨论了 TFS 怎样从驻留的闪存空间到从运行配置的 DRAM 空间，传输固定格式文件（也就是 ELF 格式文件）的块在分配给 TFS 的闪存空间之外没有代码运行。一次存储器拷贝（可能是一次解压缩）通常是复杂的。前面提到的这种方法的缺点是在不需要的时候（由于.text 空间被认为是不可写的），把操作使用的.text 空间拷贝到 RAM 上。这种装载过程的一种方式是现场执行（XIP），意味着程序的.text 块驻留在闪存，CPU 可在闪存之外直接运行，不需要任何存储器的拷贝。

名字空间（不是地址空间）的优点是为什么 TFS 不支持 XIP 的关键原因。XIP（没有地址译码）需要程序驻留在存储器中固定的部分，而 TFS 不能保证文件系统中的任何文件会在一个固定的位置，允许程序留在闪存并且直接在闪存外执行。实际上，TFS 不知道这个程序的存在，但因为 TFS 可以驻留在闪存器件中，所以微监

控器仍可以在 XIP 模式下执行程序。

设计监控器/器件闪存映像的标准方法如“无 XIP 支持”这张图（如图 7.12 所示）所示。监控器在所有剩余闪存空间为 TFS 专用的情况下工作。如果程序在 XIP 模式下执行，则 TFS 的空间会被移动（如图 7.12 所示中“XIP 支持”一侧），一些固定数目的扇区会被 XIP 程序专用。注意，微监控器和程序仍然访问 TFS 数据存储器，但是程序本身不在 TFS 中。

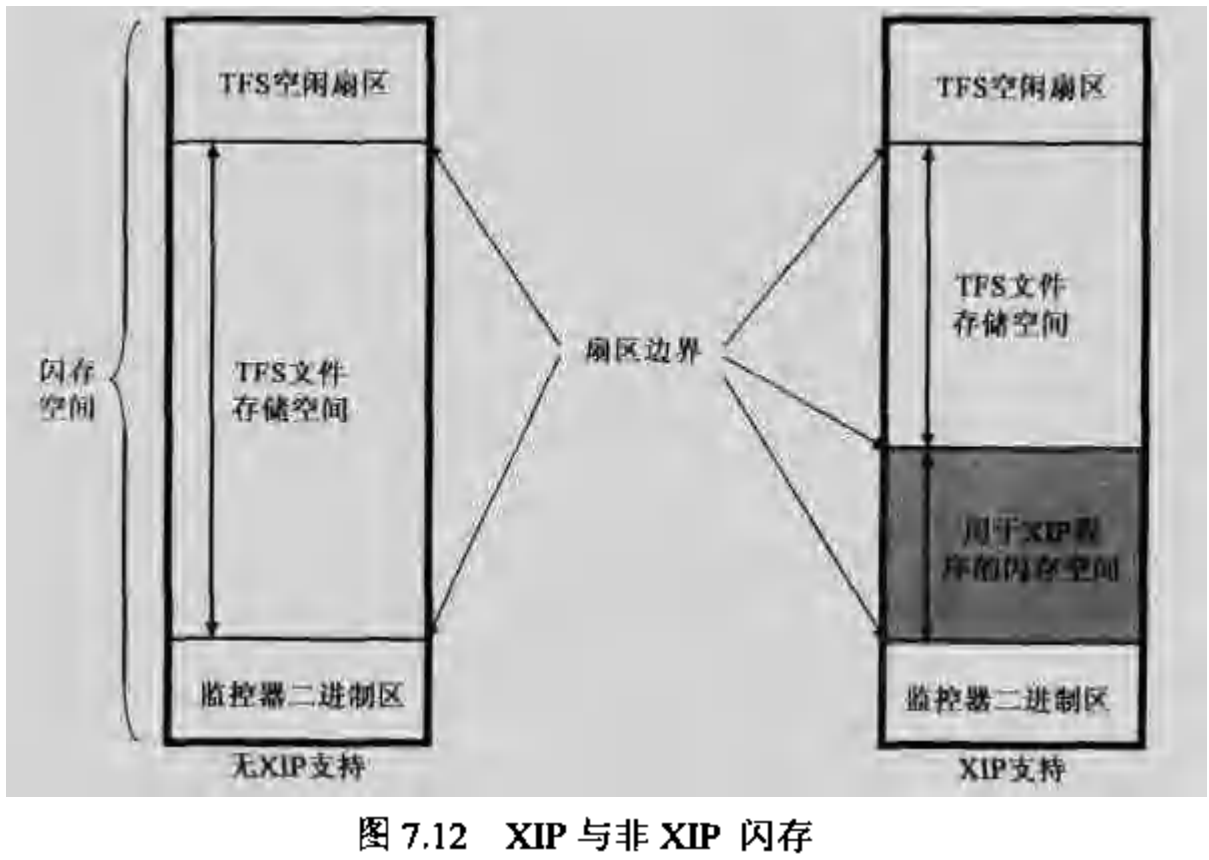


图 7.12　XIP 与非 XIP 闪存

监控器支持 XIP 仅因为 TFS 开始的点可以被调整，一定数目的闪存扇区会被 XIP 程序的存储器专用。

7.12　小结

TFS 需要的代码比本章所讲的多得多。TFS 包括一整套 API 函数来支持读、写、搜寻、打开、关闭和其他文件操作。完整的包还包括把这些函数看做 CLI 一部分的微监控器命令，但追求 TFS 中每个函数的细节会占用太多的篇幅，因此本章所讲内容只能让你了解闪存文件系统的基本特征。完整的程序代码请从本书所附 CD 上得到。

第8章 执行脚本

监控系统一般包括一个 flash 文件存储系统和一个命令行接口。系统的扩展可使文件像一条脚本一样地被执行，脚本因文件可轻松地具有类似的指令特征——这一特征给微监控系统带来了很大的灵活性。在微监控器的命令集中有一些脚本运行专用命令，在这些命令的帮助下，一个单独的脚本运行函数可以为简单的脚本执行提供通用的环境。

8.1 脚本运行器

脚本运行器函数［tfsscript（）］（见程序清单 8.1）把一个程序文件当做一串命令来对待，读取文件的每一行，且读一行即执行一行。但有些命令直接影响指令的运行，例如 goto、gosub、return 和 exit，其细节将在以后仔细讨论。

程序清单 8.1 tfsscript()

```
int
tfsscript(TFILE *fp, int verbose)
{
    char    lcpy[CMDLINESIZE];
    int     tfd, lnsize;

    /* TFS does not support calling a script from within a subroutine. */
    if (ReturnToDepth != 0)
        return(TFSERR_SCRIPTINSUB);

    tfd = tfsopen(fp->name,TFS_RDONLY,0);
    if (tfd < 0)
        return(tfd);

    ReturnToTfd = tfd;
```

```
while(1) {
    lnsize = tfsgetline(tfd,lcpy,CMDLINESIZE);
    if (lnsize == 0)       /* end of file? */
        break;
    if (lnsize < 0) {
        printf("tfsscript ( ) : %s\n",tfserrmsg(lnsize));
        break;
    }
    if ((lcpy[0] == '\r') || (lcpy[0] == '\n')) /* empty line? */
        continue;

    lcpy[lnsize-1] = 0;              /* Remove the newline */

    /* Just in case the goto tag was set outside a script, */
    /* clear it now. */
    if (ScriptGotoTag) {
        free(ScriptGotoTag);
        ScriptGotoTag = (char *)0;
    }

    ScriptExitFlag = 0;

    /* Execute the command line: */
    tfsDocommand(lcpy,verbose);

    /* Check for exit flag.  If set, then in addition to terminating the
     * script, clear the return depth here so that the "missing return"
     * warning  is not printed.  This is done because there is likely
     * to be a subroutine with an exit in it and this should not
     * cause a warning.
     */
    if (ScriptExitFlag) {
        ReturnToDepth = 0;
        break;
```

```
}

/* If ScriptGotoTag is set, then attempt to reposition the line
 * pointer to the line that contains the tag.
 */
if (ScriptGotoTag) {
        int         tlen;

        tlen = strlen(ScriptGotoTag);
        tfsseek(tfd,0,TFS_BEGIN);
        while(1) {
                lnsize = tfsgetline(tfd,lcpy,CMDLINESIZE);
                if (lnsize == 0) {
                        printf("Tag '%s' not found\n", ScriptGotoTag+2);
                        free(ScriptGotoTag);
                        ReturnToDepth = 0;
                        ScriptGotoTag = (char *)0;
                        tfsclose(tfd,0);
                        return(TFS_OKAY);
                }
                if (!strncmp(lcpy,ScriptGotoTag,tlen)) {
                        free(ScriptGotoTag);
                        ScriptGotoTag = (char *)0;
                        break;
                }
        }
    }
}
tfsclose(tfd,0);
if (ScriptExitFlag & REMOVE_SCRIPT)
        tfsunlink(fp ->name);
if (ReturnToDepth != 0) {
        printf("Warning: '%s' missing return.\n",fp ->name);
        ReturnToDepth = 0;
```

```
        }
        return(TFS_OKAY);
    }
```

此程序的运行最初要忽略程序清单 8.1 中一些全局变量的使用，只从基本程序开始。tfsscript（）函数开始时调用 tfsopen（）[1]函数来打开一个 TFS 文件。如果文件存在，则脚本运行器就会进入一个执行文件每一行的循环。循环从最顶开始，调用 tfsgetline（）函数去取回文件中的下一行。如果取得的行是空的，那么循环马上会跳过这一行；否则，这一行就会被传到 tfsDocommand 所指向的函数中。变量 tfsDocommand 是一个函数指针，默认的是指向 docommand（）函数（在后面的章节中，我们会解释为什么这是一个函数指针，而不是一个普通的函数调用）。我们暂时忽略 ScriptGotoTag 分支，从而使指令运行并不是特别复杂。这个函数读指令文件的每一行，读了一行后随即把它们送到命令处理器中去。当函数读到文件的结尾时，函数就关闭，然后返回。

8.1.1 Exit

tfsscript（）函数中的全局变量给脚本运行器提供了一个功能，使它可以跳到某个程序标识符（goto 命令），还可以转入（gosub 命令）子程序或从子程序返回（return 命令）。在脚本的任何一处，exit 命令都可以使整个运行中止。这些命令（goto、gosub、return 和 exit）的执行均需通过 docommand（）命令。程序清单 8.2 是 exit 命令的实例。

清序清单 8.2　exit 命令

```
int ScriptExitFlag;


char *ExitHelp[] = {
    "Exit a script",
    "-[r]",
    "Options:",
    "-r     remove script after exit",
    0,
};
```

1. 我们在讨论监控器代码。监控器不仅向应用程序提供使用 TFS 的功能，而且监控器本身也使用 TFS API。

```
int
Exit(int argc, char *argv[])
{
    ScriptExitFlag = EXIT_SCRIPT;
    if ((argc == 2) && (!strcmp(argv[1],"-r")))
        ScriptExitFlag |= REMOVE_SCRIPT;
    return(CMD_SUCCESS);
}
```

如果一行脚本中包含 exit 命令行，则以上的代码就在 tfsscript（）以外执行，全局变量 ScriptExitFlag 被置成 EXIT_SCRIPT。如果 -r 选项存在，则在全局变量 ScriptExitFlag 中的 REMOVE_SCRIPT 位也会被置位。参考程序清单 8.1 中的 tfsscript（）函数，在 tfsDocommand（）函数返回之后，代码会检查 ScriptExitFlag 变量是否已被更改。如果 ScriptExitFlag 是非零的，则循环就被中止，文件被关闭。如果 ScriptExitFlag 中的 REMOVE_SCRIPT 位被置位，则这条指令就会被自动删除。

8.1.2 Goto

tfsscript（）函数也提供一个全局变量 ScriptGotoTag。goto 命令的格式是 goto {tagname}，并建立了 ScriptGotoTag（一个字符指针——见程序清单 8.3）来指向作为 goto 参数的命令行标识符。参考 tfsscript（）函数（见程序清单 8.1），如果 ScriptGotoTag 变量在从 tfsDocommand（）返回后被置位，则当前打开的脚本会被搜寻是否开头含有 pound 标志、空格或标识。例如，goto TOPOFLOOP 会使 tfsscript（）搜索当前运行的整个指令来寻找#TOPOFLOOP 行，然后文件指针会被设置成在脚本的新位置继续执行。

程序清单 8.3 goto 命令

```
char      *ScriptGotoTag;


/* gototag():
 *  Used with tfsscript to allow a command to adjust the pointer into the
 *  script that is currently being executed.  It simply populates the
 *  "ScriptGotoTag" pointer with the tag that should be branched to next.
 */
```

```
void
gototag(char *tag)
{
    if (ScriptGotoTag)
        free(ScriptGotoTag);
    ScriptGotoTag = malloc(strlen(tag)+8);
    sprintf(ScriptGotoTag,"# %s",tag);
}

char *GotoHelp[] = {
    "Branch to file tag",
    "{tagname}",
    0,
};

int
Goto(int argc, char *argv[])
{
    if (argc != 2)
        return(-1);
    gototag(argv[1]);
    return(CMD_SUCCESS);
}
```

8.1.3　gosub 和 return

为了完成 tfsscript（）的细节描述，我们来讲述进入和从子程序返回的脚本运行器功能，即 gosub 和 return。

程序清单 8.4　gosub and return 命令

```
char *GosubHelp[] = {
    "Call a subroutine",
    "{tagname}",
    0,
```

```
};

int
Gosub(int argc, char *argv[])
{
    if (argc != 2)
        return(-1);
    gosubtag(argv[1]);
    return(CMD_SUCCESS);
}

char *ReturnHelp[] = {
    "Return from subroutine",
    " ",
    0,
};

int
Return(int argc, char *argv[])
{
    if (argc != 1)
        return(-1);
    gosubret(0);
    return(CMD_SUCCESS);
}
```

以上程序结束的命令（见程序清单 8.4）在本质上是相同的，只不过是一个调用 gosubtag（），而另一个调用 gosubret（）（见程序清单 8.5）。

程序清单 8.5　gosub and return Under the Hood

```
#define MAXGOSUBDEPTH    15

static long ReturnToTbl[MAXGOSUBDEPTH+1];
static int   ReturnToDepth, ReturnToTfd;

void
```

```
gosubtag(char *tag)
{
    if (ReturnToDepth >= MAXGOSUBDEPTH) {
        printf("Max return-to depth reached\n");
        return;
    }
    ReturnToTbl[ReturnToDepth] = tfstell(ReturnToTfd);
    ReturnToDepth++;
    gototag(tag);
}

void
gosubret(char *ignored)
{
    if (ReturnToDepth <= 0)
        printf("Nothing to return to\n");
    else {
        ReturnToDepth--;
        tfsseek(ReturnToTfd, ReturnToTbl[ReturnToDepth], TFS_BEGIN);
    }
}
```

gosub 命令调用 gosubtag（）函数，此函数记载了子程序返回点在文件中的位置。ReturnToTbl[]数组中的下一个元素是由 tfstell（）返回的值提供的，其本质是一个存放返回地址的堆栈（用文件偏移的形式），像我们以前讨论的一样，函数 gosubtag（）调用 gototag（）。return 命令通过调用 gosubret（）结束调用联系。函数 gosubret（）从 ReturnToTbl[]表中取出返回地址，然后按返回文件中正确的位置［用 tfsseek（）］返回。返回地址堆栈的深度减少了 1，同时搜寻文件中的返回点。函数调用的最大嵌套层数受 ReturnToTbl[]表大小的限制，用 gosubtag（）函数检测调用是否超过了这个限制。

8.2　条件转向

还有一些命令提供了额外的功能来增强脚本的运行环境，尽管它们不像 exit、

goto、gosub 和 return 一样直接作用于 tfsscript（）函数，但其中一个值得详细介绍的命令就是 if 命令。if 命令提供了微控制器编程环境中所需的条件转向功能，执行使用的是以前介绍的 gosub、goto、exit 和 return 命令。

程序清单 8.6　Conditional Branch Support:The if Command

```
char *IfHelp[] = {
    "Conditional branching",
    "–[t:] [{arg1} {compar} {arg2}] {action} [else action]",
    " Numeric/logic compare:",
    "   gt lt le ge eq ne and or",
    " String compare:",
    "   seq sne",
    " Other tests (–t args):",
    "   gc, ngc, iscmp {filename}",
    " Action:",
    " goto tag | gosub tag | exit | return",
    0,
};

int
If(int argc, char *argv[])
{
    int opt, arg, true, if_else, offset;
    void    (*iffunc)(), (*elsefunc)();
    long    var1, var2;
    char    *testtype, *arg1, *arg2, *iftag, *elsetag;

    testtype = 0;
    while((opt=getopt(argc,argv,"t:")) != –1) {
        switch(opt) {
        case 't':
            testtype = optarg;
            break;
        default:
```

```
            return(CMD_PARAM_ERROR);
        }
    }

    elsetag = 0;
    elsefunc = 0;
    offset = true = if_else = 0;
```

if 命令的目的是做一些测试和比较，并根据结果做出某种动作。通常，比较是把变量和某些常数相比，或变量和变量相比。这些变量可以是字符串或数字，选项 -t 可使用户在控制器中检测某个字符或确定文件是否已被压缩。

if 命令的基本格式是

```
if {comparison between A & B is true} {perform actionX}
[else {perform actionY}]
```

或

```
if {-t testtype} {perform actionX} [else {perform actionY}]
```

这里，testtype 的测试结果是 true 或 false。

if 命令有两种可能的结果，所以必须用两个不同的函数指针来调入所需的函数（iffunc 和 elsefunc）来执行特定的行为。见程序清单 8.6，命令行首先检测-t 选项。如果存在，就设定特定的标志。

程序清单 8.7　Parsing Else Clauses

```
/* First see if there is an 'else' present... */
    for (arg=optind;arg<argc;arg++) {
        if (!strcmp(argv[arg],"else")) {
            if_else = 1;
            break;
        }
    }

    if (if_else) {
        elsetag = argv[argc-1];
        if (!strcmp(argv[argc-1],"exit")) {
```

```
        offset = 2;
        elsefunc = exitscript;
    }
    else if (!strcmp(argv[argc-1],"return")) {
        offset = 2;
        elsefunc = gosubret;
    }
    else if (!strcmp(argv[argc-2],"goto")) {
        offset = 3;
        elsefunc = gototag;
    }
    else if (!strcmp(argv[argc-2],"gosub")) {
        offset = 3;
        elsefunc = gosubtag;
    }
    else
        return(CMD_PARAM_ERROR);
    }

    iftag = argv[argc-offset-1];
    if (!strcmp(argv[argc-offset-1],"exit"))
        iffunc = exitscript;
    else if (!strcmp(argv[argc-offset-1],"return"))
        iffunc = gosubret;
    else if (!strcmp(argv[argc-offset-2],"goto"))
        iffunc = gototag;
    else if (!strcmp(argv[argc-offset-2],"gosub"))
        iffunc = gosubtag;
    else
        return(CMD_PARAM_ERROR);
```

下面进一步分析（见程序清单 8.7）确定 else 关键字在命令行中是否存在。如果有，就装载 elsefunc 函数指针来执行 else 的程序。这个行为可能是 goto、gosub、exit 或 return，然后跟着的四路转向语句装入 iffunc 函数指针来执行相应的行为。

程序清单 8.8　Performing the Test or Comparison

```
if (testtype) {
        if (!strcmp(testtype,"gc")) {
            if (gotachar())
                true=1;
        }
        else if (!strcmp(testtype,"ngc")) {
            if (!gotachar())
                true=1;
        }
        else if (!strcmp(testtype,"iscmp")) {
            TFILE *tfp;

            tfp = tfsstat(argv[optind]);
            if (tfp) {
                if (TFS_ISCPRS(tfp))
                    true=1;
                )
                else
                    printf("'%s' not found\n",argv[optind]);
            }
            else
                return(CMD_PARAM_ERROR);
        }
        else {
            arg1 = argv[optind];
            testtype = argv[optind+1];
            arg2 = argv[optind+2];

            var1 = strtoul(arg1,(char **)0,0);
            var2 = strtoul(arg2,(char **)0,0);

            if (!strcmp(testtype,"gt")) {
                if (var1 > var2)
                    true = 1;
```

```
}
else if (!strcmp(testtype,"lt")) {
        if (var1 < var2)
                true = 1;
}
else if (!strcmp(testtype,"le")) {
        if (var1 <= var2)
                true = 1;
}
else if (!strcmp(testtype,"ge")) {
        if (var1 >= var2)
                true = 1;
}
else if (!strcmp(testtype,"eq")) {
        if (var1 == var2)
                true = 1;
}
else if (!strcmp(testtype,"ne")) {
        if (var1 != var2)
                true = 1;
}
else if (!strcmp(testtype,"and")) {
        if (var1 & var2)
                true = 1;
}
else if (!strcmp(testtype,"or")) {
        if (var1 | var2)
                true = 1;
}
else if (!strcmp(testtype,"seq")) {
        if (!strcmp(arg1,arg2))
                true = 1;
}
else if (!strcmp(testtype,"sne")) {
```

```
            if (strcmp(arg1,arg2))
                true = 1;
        }
        else
            return(CMD_PARAM_ERROR);
    }
```

参考程序清单 8.8，执行测试和比较，true 标志在结果基础上被置为相应的值。

程序清单 8.9　Execute the Appropriate Action

```
/* If the true flag is set, call the 'if' function.
 * If the true flag is clear, and "else" was found on the command
 * line, then call the 'else' function...
 */
if (true)
    iffunc(iftag);
else if (if_else)
    elsefunc(elsetag);

    return(CMD_SUCCESS);
}
```

最后，根据 true 标志的值及 iffunc 和 elsefunc 指针的内容，相应的动作被执行（见程序清单 8.9）。总之，用户只用一些很小的、简单的代码就可得到大量的指令功能。关于 read、item、sleep、echo 和其他相关命令信息可以参考所附 CD。

8.3　一些例子

前面已经介绍了一些脚本的运行。现在我们要举一些例子来说明这些脚本是怎样使用的。这些例子中，一些命令的细节在文中并没有描述，详细细节请参考 CD。

8.3.1　例子#1：ping

微控制器命令系统的基本原理是使低级的功能可以通过书写组合形成高级的功能，而不是在命令列表中提供许多命令。以下是这个基本原理的例子。它组合 icmp

命令的功能来建立一个 ping 命令。icmp 命令在微控制器命令行中支持一些 ICMP 特征。echo 说明 icmp 命令是一个基本的 ping。但是，ping 命令不在微控制器命令系统中，因为它是由一串脚本组成的。

程序清单 8.10 Building ping

```
#
# PING:
# Script using "icmp echo" and "argv" commands:
# Syntax: ping IP_ADDRESS [optional ping count]
#
argv -v
if $ARGC eq 2 goto PING_1
if $ARGC eq 3 goto PING_N
echo $ARG0: requires IP address
exit


# PING_1:
icmp echo $ARG1
exit


# PING_N:
icmp -c $ARG2 echo $ARG1
exit
```

argv -v 命令修改了传递给 ping 的命令行，并且开辟适当的变量。对于此例，如果指令如此调用

```
ping 135.3.94.1 6
```

则$ARG0 会保持 ping，$ARG1 会保持 135.3.94.1，$ARG2 会保持 6，$ARGC 会保持 3。注意，注释的应用、条件转向语句和 goto 命令。PING_1 和 PING_N 在命令行说明的基础上标记了两个不同的逻辑路径，剩下的就很容易理解了。

8.3.2 例子#2：外壳数组

程序清单 8.11 在微处理器命令解释器中使用嵌套的外壳变量来执行简单的程

序，所有的格式都是与程序清单 8.10 中相似的，此例显示了微控制器中嵌套外部变量的功能。

程序清单 8.11　Implemnting Simple Arrays

```
#
# Build a list (or array) of vegetables:
#
set VEG_1 Lettuce
set VEG_2 Broccoli
set VEG_3 Carrot
set VEG_4 Corn

set idx 1
set max 4

#
# Now print the list of vegetables using the 'idx' shell variable
# as an index:
#
# TOP:
if $idx gt $max exit          # Exit loop when idx is greater than max.
echo ${VEG_${idx}}            # Use nested shell var to print $VEG_N.
set -i idx                    # Increment content of idx.
goto TOP                      # Do it again.
```

程序清单 8.11 的运行输出是

```
Lettuce
Brocoli
Carrot
Corn
```

set 命令给外部变量分配了一个值，当 CLI 处理器分析到

```
echo ${VEG_${idx}}
```

时，分两步通过外部变量的处理阶段：第一步将${idx}变为它的值；第二步将

${VEG_N}（这里 N 是$idx 的值）变为它的值。注意，idx 外部变量使用时像一个名字数组的索引。这里，这个数组叫 VEG_。

8.3.3 例子#3：子程序、条件转向、TFS 及其他

程序清单 8.12 进一步说明了条件转向和子程序的用法，且使用一些 tfs 命令的特性作为示范的一部分。指令从 TFS 闪存中装入一个可执行文件到 DRAM，提示用户文件是否已被压缩。

程序清单 8.12 Using a Script to Load an Executable

```
#
# Load file:
# Script syntax: fload {filename}
#
argv -v
if $ARGC ne 2 goto USAGE

# See if file exists...
tfs size $ARG1 FSIZE
if $FSIZE seq \$FSIZE goto NOFILE

# See if file is compressed...
if -t iscmp $ARG1 gosub COMPRESSED else gosub NCOMPRESSED

# Verbosely load the file to dram...
tfs -v ld $ARG1
exit

#
# COMPRESSED:
#
echo File $ARG1 is compressed.
echo
return
```

```
#
# NCOMPRESSED:
#
echo File $ARG1 is not compressed.
echo
return

#
# NOFILE:
#
echo File $ARG1 not found.
exit

#
# USAGE:
#
echo Usage: $ARG0 {filename}
exit
```

经过基本的说明性检测，脚本可使用 tfs size 命令来开辟外部变量 FSIZE 记载文件的大小。如果文件存在，则 FSIZE 变量会按之赋值；否则，FSIZE 会被清零。如果 FSIZE 变量被清零，则 CLI 处理器会保持$FSIZE 不动，所以$FSIZE 和\$FSIZE 会显示同样的字符串，因为第二个字符串中的反斜杠会在 CLI 分析时被清除。第二个 if 语句用来进行比较。第三个 if 语句显示 if 使用-t 选项时的附加测试功能。在此测试中，if 正在检测文件以确定文件是否被压缩。COMPRESSED 语句指向一个子程序，NOFILE 和 USAGE 是转向目标，它们都可打出一些用户信息。最后，这个文件被装入，传递未压缩文件（在此例中是文件 abc）名字时的显示，见程序清单 8.13。

程序清单 8.13　The Load Statistics

```
File abc is not compressed.

.text       : copy    852808 bytes from 0xf0142fdc to 0x00080000
.rodata     : copy     68964 bytes from 0xf0213324 to 0x00150348
```

```
.data       : copy    61984 bytes from 0xf022408c to 0x001610b0
.ctors      : copy       12 bytes from 0xf02332ac to 0x001702d0
.dtors      : copy       12 bytes from 0xf02332b8 to 0x001702dc
.got        : copy       16 bytes from 0xf02332c4 to 0x001702e8
.sbss       : set       916 bytes   at   0x001702f8 to 0x00
.bss        : set     91252 bytes   at   0x00170690 to 0x00
.comment    :           17160 bytes not processed (tot=17160)
.shstrtab :             101 bytes not processed (tot=17261)
entrypoint: 0x80000
```

以上例子展示了脚本运行器的书写组合功能，也显示了一些微控制器的 CLI 列表中一些命令的多功能性。若想得到命令系统的全部详细说明，请参阅所附 CD。

8.4 小结

如果“嵌入式系统”这个词联想到的是失败的工程师正在拼命地从他们的代码中挤出另外 15 个字节来适应 1KB 的片上存储器，那么指令系统的功能可能看上去好像是不可实现的奢望。但实际上不是这样。首先，现代的硅集成技术使大多数器件能够轻易地提供足够的代码空间——尤其是在已经进步了的时候。如果需要，则监控器还可以被缩小体积。

其次，如以前举例子所述，指令可以是一个用非常强大的工具建立起来的系统级的功能和开发工具。

最后，脚本功能可促使使用者努力实现代码的模块化。脚本运行器允许你写入大部分监控器和应用程序时使用的独立的、简单的函数，并把它们用不同的脚本组合起来且形成有意义的功能。虽然用编程代码即可实现同样的功能，但是脚本却更容易书写、修改和组合。因为脚本是依靠监控器来执行它的内容，在很多情况下，脚本要明显小于同等的程序。

第 9 章　网络连通性

现在，以太网非常普遍，嵌入式系统也不能忽视它。以太网的通信接口比简单的 RS-232 要复杂一些，但安装和运行却是廉价的。

TCP/IP 协议结构被分成了若干层，每一层都被看做是把一块数据（或载荷）放在上面。以太网的载荷是 IP 和 ARP，IP 的载荷是 ICMP、UDP 或 TCP（及其他），UDP 和 TCP 的载荷为 TFTP、HTTP 和 FTP。FTP 在 TCP 上，TCP 在 IP 上，IP 在以太网上。你可以把 TCP/IP 协议看成一串卡车。每一辆卡车都正好小到能被装进前一辆卡车里面。第一辆卡车并不在乎它里面装的是什么货物。实际上，也可将货物比作是一辆小一点的卡车。每一辆卡车都只是把载荷从起点送到目的地。载荷到了目的地之后就和卡车没有了任何联系。

注意

如果你确实想了解网络，请买一本关于网络的书。本章将不能讲述整个网络方面的知识，主要目的是讨论一些基本协议及把嵌入式系统和 IP 网络相连的步骤。

为了将嵌入式系统与 IP 网络相连，你必须与一些以太网络设备接口相连，且还要写一个包处理程序。

9.1　以太网

所有的事情都是从最大的载重卡车开始的，也即以太网的结构。以太网的数据包格式为：

目标主机地址	**6B**
源主机地址	**6B**
结构类型	**2B**
载荷	1 500B

目的地址就像即将要看到的，可以分为明确的 MAC 地址或广播地址。MAC

地址格式通常是用 6 个冒号分开的字节。每个字节占 8 位，范围 0~255，值表现为十六进制的 ASCII 码。例如，00:60:1D:02:08:B5 是一个有效的 MAC 地址。注意，每个收到的数据包都包含源数据地址。

9.2 ARP

以太/IP 网络上的每一个器件都有两个地址——以太网层上的 MAC 地址和网络层上的 IP 地址。要想通过以太网连接某个器件，你需要知道这个器件的 MAC 地址。可是，由于你运行 IP 是在以太网层的顶端，因此地址分析协议（ARP）允许使用 IP 地址来定位其他器件，而不是用它们的 MAC 地址。除了 IP，ARP 是我们本章惟一要讨论的以太网络负载（还有其他的，但是超出了本章讨论的篇幅）。ARP 数据包结构是这样的：

硬件	**2B**
协议	2B
HIen	**1B**
Plen	1B
操作	**2B**
发送硬件地址	6B
发送 IP 地址	**6B**
目标硬件地址	6B
目标 IP 地址	**4B**

注意，ARP 并没有负载，也没有更多的包装，惟一的目的是解析 MAC/IP 地址组合。网络上的每一个器件必须有能力接收到来的 ARP 请求和适当回应请求。

ARP 解决了本来会出现的 catch-22 问题。如果器件想与目标联系，则必须先询问目标的 MAC 地址，但要做到这一点，就必须已经知道目标的 MAC 地址！这就需要使用特殊的以太网广播地址（FF:FF:FF:FF:FF:FF）向所有的器件发出 ARP 请求。

网络上的所有器件必须有能力接收以太网广播。广播中的消息是 ARP 格式的，到来的 ARP 请求包含了希望得到 MAC 地址（目标的 IP 地址）的器件的 IP 地址。每个器件上的 ARP 包处理器必须将收到的广播请求中的 IP 地址与自己的 IP 地址相比较。如果匹配，则这个器件就用收到的包中的硬件（MAC）地址和 IP 地址给请求者发出 ARP 回应。

一旦请求者收到了这个信息，它就在请求者的本地存储器中存储一段时间（这

个本地存储器叫做 ARP 缓存）。存储的 MAC 地址使器件可以修正给定的 IP 地址的 MAC 地址，而不用发出另一条 ARP 请求。网络层次协议的低层驱动不断地接收到来的数据包，然后刷新 ARP 缓存中的 MAC/IP 地址组合。如果缓存的入口在一段时间（定为 2~20min）内没有使用，则这个入口便会被从缓存中移除（或冲掉）。

9.3　IP

因特网协议（IP）是所有 TCP/IP 网络的工场。它的包结构是这样的：

样式和头长度	1B
服务种类	1B
包长度	2B
验证	2B
碎片弥补区	2B
活动时间	1B
协议	1B
校验	2B
源 IP 地址	4B
目标 IP 地址	4B
载荷	不超过 64KB

不像 ARP，IP 总是有载荷的，目的是从一地向另一地传递数据（像一辆很好的载重卡车）。载荷是与记载设备的规模和编号的信息在一起的，这样可使包自己找到目的地。不同种类的网络设备使用包的不同部分来做不同用途，因此最重要的信息是协议、包长度、校验、源地址和目标地址部分。协议部分告诉我们载荷是什么。在此，载荷是 ICMP、UDP 及 TCP 等等。校验是对包的基本完整性的检测。源和目标地址是发送数据包的设备和假定接收数据包的设备的 IP 地址。包处理器必须查看所有这些部分的信息后才能接收和弹出数据包，进而确定数据包要去的目的地。

9.4　ICMP

协议中必须有一部分的信息用来确定载重卡车是否运行正常或是否按正确的路线送往目的地。这个工作也是网络控制信息协议（ICMP）责任的一部分。微控

制器只履行 ICMP 的一小部分。对我们来说，ICMP 的最重要部分是具有的回应功能。通常，ICMP 回应借助 ping。ping 的目的是使一个网络器件可找出是否有另一个工作正常的设备在网络上。一个发送 ICMP 请求的设备试图确定是否有另一个正常的网络设备。收到 ICMP 回应请求的设备要确定这个请求是否被激活。如果你想让其他的设备可以确定目标系统是否在网络上，则 ICMP 的支持是很重要的。ICMP 的另一个重要用途（在这种情况下）是可以告知发送方一个包能否传输。例如，微监控器不支持网络文件系统（NFS），所以如果系统收到 NFS 请求，它会用 ICMP 不可用的协议消息来告知发送方请求不可满足。

注意

在嵌入式系统中，ICMP 回应功能还有另一个方便的用途。本章前面的部分提到了 ARP 缓存。ARP 缓存对网络的稳定是非常有用的。当一个设备连上网络时，它的 MAC 和 IP 地址是固定的（假设现在没有 DHCP 的许可）。在开发环境下，网络连接设备只要建立和调试，则建立和调试处的网络构造的稳定性都略差一些，此时 ARP 缓存装置即可发挥作用，特别是你需要循环一个特殊的 IP 地址时。

例如，你正在进行一些使用网络的工程，网络管理员分给你一个独立的 IP 地址用来硬件检测。乍看上去，这种安排似乎是合理的，但在几个不同的嵌入式系统同时工作时，在同一时间你只能有一个 IP 地址，所以你建立的所有器件均在同一个 IP 地址上。当你使用 IP/UDP 和第一个设备相连时，工作一切正常。你的主机识别这个设备的 IP 地址为 135.3.94.138，MAC 地址为 00:60:1d:02:0b:fe，此时主机默默地在它的 ARP 缓存中加上了这个连接的内容。

现在关闭设备 1 连接设备 2。设备 2 也被配置成 IP 地址为 135.3.94.138，而 MAC 地址为 00:60:1d:02:0b:f0。当你试图和这个设备对话时，你的主机就会说它不能找到这个设备。

如果这时你输入命令 arp –a（或其他类似的命令）来导出主机 ARP 缓存中的内容，你就会看到主机仍旧认为 135.3.94.138 的 MAC 地址是 00:60:1d:02:0b:fe（因为主机在和设备 1 交互时，把 IP/MAC 地址组合放入 ARP 缓存）。如果你不能主动地冲掉主机上的 ARP 缓存，则有两个选择：你可以等到机器自动地冲掉这个缓存，但可能需要数分钟的时间；另一种是可以从你的目标发出一个 ICMP 回应到子网上的任何其他设备。主机的网络接口会收到 ICMP 且立即更新它的 ARP 缓存。现在你的主机和设备 2 就可以正常通信了。

9.5　UDP 和 TCP

IP 载荷的内容是会变化的，大多数的 IP 包包含用户数据包协议（UDP）或传输控制协议（TCP）。在 IP 层，你不能知道除了包的大小和发送者之外的很多关于包的信息。如果到来的 IP 载荷是 UDP，则你能在 UDP 开头找到关于是谁和谁在通信的信息。例如，假设你有一个文件服务器可以给多路机器提供服务文件，在机器 1 上的一个用户需要文件 X，另一个用户需要文件 Z，同时在机器 2 上一个用户需要文件 Y。当这些请求同时到来时，低层协议应怎么知道到来的文件请求是适用于这台机器上的文件服务功能呢？这很像运行在任何一台机器（如 TFTP、FTP、HTTP 或 TELNET）上的复合服务器，当文件服务设备收到了同样的请求时，它是怎么知道该回应哪个访问者呢？当文件服务器收到来自同一个机器上的两个不同用户的两个请求时，它怎样才能正确回应用户呢？这比现在的单纯 IP 地址要复杂些，工作也变得更加复杂了，因为机器只有 IP 地址，但是却有多个用户或设备正在试图通过网络通信。

UDP 的数据包头如下。它由一个头和一个伪头组成。伪头由从 IP 层取得的信息组成，在表中用斜体字书写。

源点	***2B***
目标点	*2B*
包长度	***2B***
校验	*2B*
源地址	***4B***
目的地址	*4B*
零	***1B***
协议	*1B*
包长度	***2B***

伪头不是 IP 包说明中的一部分（因此叫“伪”头），由它下面的 IP 头组成并且连接到 UDP 包上。源点、目标点、源 IP 地址和目标 IP 地址的组合使这些包的发送者和接收者不仅能够跟踪设备（或机器），而且也能跟踪设备（或机器）中的器件，这也是包的责任之一。

TCP 使 UDP 看上去像是一件很容易的事情。TCP 是非常复杂的和强大的协议，仅用几段文字说明是肯定不够的。UDP 和 TCP 最重要的区别是 UDP 不能保证包会被收到。UDP 只是向目的地发送一个包，如果这个包没有被收到，则器件代码必

须恢复；相反，TCP 却可以保证（这是有原因的）包被收到。为了这个保证，就要大量增加代码的复杂性和实时性。TCP 提供的地址与 UDP 提供的相似，实现了发送方和接收方以状态为基础的非常精确地握手，保证包能够到达目的地。微控制器可回应所有的带有连接复位的 TCP 连接请求，用这种方法可避开这一复杂性。

9.6 DHCP/BOOTP

自举协议（BOOTP）和动态主机配置协议（DHCP）是在网络启动时的 IP 协议。DHCP 是 BOOTP 的现代化。这些协议允许一个目标在它本身未觉察时启动。在目标自举后，它会和 BOOTP 或 DHCP 服务器对话以修正其中的关于自己的配置信息。下面是 BOOTP 头的格式。DHCP 与此相同。

操作	1B
高字节	1B
Hlen	1B
Hops	1B
处理 ID	4B
秒	2B
未使用	2B
访问者 IP	4B
你的 IP	4B
服务器 IP	4B
路由器 IP	4B
访问者 MAC 地址	16B
服务器主机名称	64B
引导文件名称	128B
供应商特定区域	64B

为了改正配置数据，目标设备必须发出 BOOTP 广播请求。像以太网一样，IP 也有一个用于广播的地址：255.255.255.255。如果网络上有一个 BOOTP 服务器，则 BOOTP 服务器可能会回应。回应包括的信息有服务器 IP 地址、目标将使用的 IP 地址、这个子网网关的 IP 地址、子网掩码、引导文件名称和这个引导文件来自的 IP 地址。DHCP 扩展了功能，并使目标和服务器能够建立动态的 IP 地址。当目标引导时，它和 DHCP 服务器用对话来修正信息并使之与 BOOTP 信息相同，但是这一操作分配给设备的 IP 地址可以是暂时的，这种方法叫 DHCP IP 地址租用。特

别是当服务器和访问者建立的联系超时时，访问者必须请求延长 IP 地址租用。这样的申请是否被批准，要看 DHCP 服务器是怎样配置的。

9.7　嵌入式系统的应用

你可能会奇怪，为什么所有的关于网络协议的信息都与嵌入式系统的硬件有关。其答案是因为要建立完整的接口，因此必须在以太网层处理到来的包。以太网设备本身保证只有当数据包的目标是设备自己的 MAC 地址时，才会接受。设备硬件要负责丢弃除了广播之外的所有的数据包。广播并不是设定发向本地 IP 地址的，一般硬件所做的这些处理是可靠的。

为了写驱动代码，你需要懂得怎样检测数据包的到来和怎样从网络控制器寻回数据包。两种操作对于设备本身都是特定的，所以需要参考设备说明文档。通常，以太网设备看上去像一个 FIFO 或者缓冲区描述器，每个描述器描述一个输入或输出缓冲区（或数据包）。

缓冲区描述器模式非常普遍，当你懂得它的用法时则是非常容易使用的。每个设备对缓冲区描述器均有自己的定义。缓冲区描述器主要包括一个命令/状态区、一个长度区、一个包含目前缓冲区指针的区和一个指向下一个缓冲区描述器的区。下图描述了缓冲区的联系结构。

图 9.1 显示了典型的缓冲区描述器和缓冲区结构。描述器和缓冲区的数目完全依赖器件和目标系统有效的存储器空间。以太网设备通常给出指向第一个缓冲区描述器的指针，当数据传输开始时，设备就假设缓冲区描述器已被正确初始化。

使用 C 结构访问外部设备

通过建立一个像设备的结构，并将它覆盖在设备占用的地址空间来实现 C 和外部设备的接口是非常普遍的。这样做时，你必须清楚一些代码发生器也可能被允许这样做。在一般程序中使用结构时，编译器/CPU 的组合会精确确定结构是怎样分配到存储器中去的。当你写了一个函数来访问结构中的一部分，并在结构基础上正确偏移的情况下，低层的代码能访问到正确的部分，并不需要明确地知道结构的每一部分在存储器上是怎样分配的。

当使用 C 结构作为通向外部设备的通道时，因外围设备上都有固定的格式，因此你建立的结构必须与外围设备的硬件相匹配，必须明确地知道结构的格式是怎样的；否则，它就不能在外围器件上准确定位。

出于各种各样未说明的原因，编译器会在结构的部分之间加入一些“填充”。这些“看不见的”填充会干扰到设备的访问，因为虽然硬件和软件结构看上去是匹配的，但看不见的填充会导致一些不匹配。在大多数情况下，这个问题的解决方法是简单的。你只需要告知编译器你不想让它在结构中加入任何填充即可，因禁止编译器做什么事应该是很正常的。如果你研究交互编译器的说明，则可能会发现一些特殊的语法（通常叫程序）会指导编译器不要在有关的结构中加入任何不可见的填充。

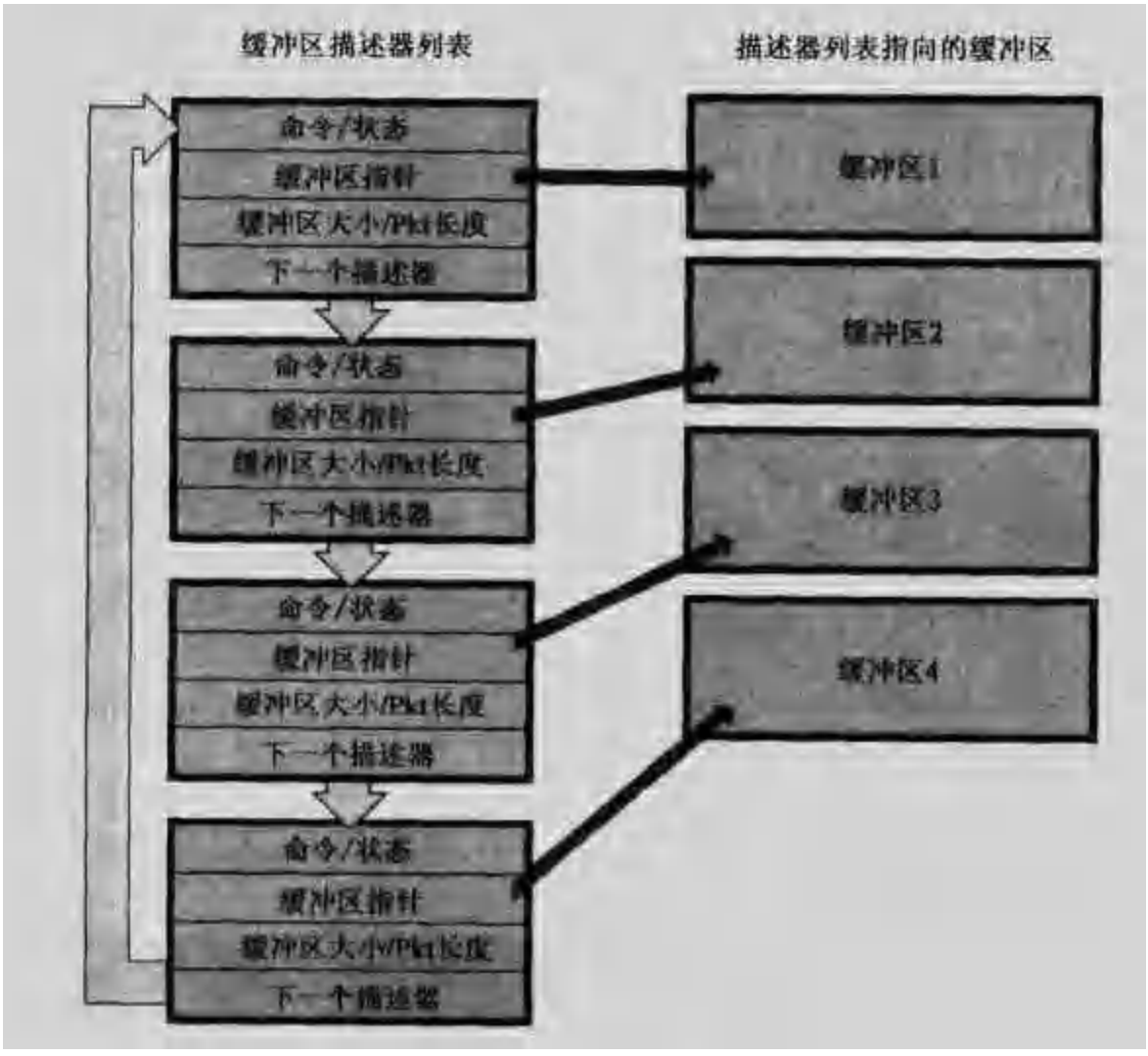


图 9.1　以太网驱动程序缓冲区描述器

以太网控制器使用与缓冲区描述器的连接来实现与器件的数据通信是很常见的。注意描述器的环形连接。

程序清单 9.1　一块缓冲区描述器的初始化

```
struct rdesc {
    ulong    cmdstat;          /* Command/Status                    */
```

```
    uchar     *bp;              /* Pointer to buffer              */
    ushort    bsize;            /* Size of buffer                 */
    ushort    psize;            /* Size of received packet        */
    struct    rdesc    *ndp;    /* Pointer to next descriptor */
};

unsigned char RbufBase[BUF_SIZE * RX_DESCRIPTOR_COUNT];
struct    rdesc RdescBase[RX_DESCRIPTOR_COUNT];

for(i=0;i<RX_DESCRIPTOR_CNT;i++) {
    RdescBase[i].cmdstat = DEVICE_IS_IDLE;
    RdescBase[i].bsize = BUF_SIZE;
    RdescBase[i].psize = 0;
    RdescBase[i].bp = (uchar *)&RbufBase[i * BUF_SIZE];
    if (i == RX_DESCRIPTOR_CNT-1) {
        RdescBase[i].ndp = RdescBase;
    }
    else {
        RdescBase[i].ndp = &RdescBase[i+1];
    }
}
```

程序清单 9.1 中的代码示例显示了应怎样初始化一块缓冲区描述器。rdsec 结构被当做缓冲区描述器使用，并且被声明为匹配描述器的格式，就像已被设备定义了一样。一块存储器（RbufBase）被用做缓冲区分配，一块存储器（RdescBase）被用做缓冲区描述器分配。for 循环初始化了连接表。注意，这个表是循环的，表中的最后一项指向第一项。

9.7.1　processPACKET（）函数

在包接收的情况下，硬件必须修正以太网的载荷，并且提取包的类型来确定包是否为 ARP 或 IP 包。如果包既不是 ARP 也不是 IP，就扔掉它。几乎所有的以太网代码的包均被送到 IP 或者 ARP 包处理器函数。如果包是 ARP，则正确的反应是发送，并且完成对包的处理。如果包是 IP，则软件必须进一步分析

包头来确定应该用哪个包处理程序。接口可以处理 UDP、DHCP/BOOTP、ICMP 和 TCP 的位。这样，在到来包内容的基础上，即可调用几个不同的高层包处理器中的一个。

程序清单 9.2　processPACKET()

```
/* processPACKET():
 *   This is the top level of the message processing after a complete
 *   packet has been received over ethernet.   It's all just a lot of
 *   parsing to determine whether the message is for this board's IP
 *   address (broadcast reception may be enabled), and the type of
 *   incoming protocol.   Once that is determined, the packet is either
 *   processed (TFTP, DHCP, ARP, ICMP-ECHO, etc...) or discarded.
 */
void
processPACKET(struct ether_header *ehdr, ushort size)
{
    int i;
    ushort   *datap, udpport;
    ulong    csum;
    struct ip *ihdr;
    struct Udphdr *uhdr;

    if (ehdr->ether_type == htons(ETHERTYPE_ARP)) {
        processARP(ehdr,size);
        return;
    }
    else if (ehdr->ether_type != htons(ETHERTYPE_IP)) {
        printPkt(ehdr,size,ETHER_INCOMING);
        return;
    }

    /* If we are NOT in the middle of a DHCP or BOOTP transaction, then
     * if destination MAC address is broadcast, return now.
     */
```

```
if ((DHCPState == DHCPSTATE_NOTUSED) &&
    (!memcmp((char *)&(ehdr->ether_dhost),BroadcastAddr,6))) {
    return;
}

/* If source MAC address is this board, then assume we received our
 * own outgoing broadcast message...
 */
if (!memcmp((char *)&(ehdr->ether_shost),BinEnetAddr,6)) {
    return;
}
```

程序清单 9.2 是 processPACKET（）函数的开头，从以太网驱动来的未处理的包和包的大小是参数。以太网的头立即被测试，用以确定是 ARP 还是 IP 包。如果包是 ARP，则执行会转到 ARP 的处理代码。如果包是某些未知的协议，则在删除它之前，会被送到控制台（类似一种诊断工具）。函数中剩下的代码假定包是 IP 的，任何到来的与 DHCP/BOOTP 无关的广播都会丢掉，而且因为到来的包也许就是这里发送的（依赖于以太网接口设备的配置），所以如果到来包的以太网源地址与目标地址相同，那这个包就被丢掉。

程序清单 9.3　Verifying the Packet Integrity

```
ihdr = (struct ip *) (ehdr + 1);

/* If not version # 4, return now... */
if (getIP_V(ihdr->ip_vhl) != 4)
    return;

/* IP address filtering:
 * At this point, the only packets accepted are those destined for this
 * board's IP address, plus, DHCP, if active.
 */
if (memcmp((char *)&(ihdr->ip_dst),BinIpAddr,4)) {
    if (DHCPState == DHCPSTATE_NOTUSED)
        return;
    if (ihdr->ip_p != IP_UDP)
```

```
            return;
        uhdr = (struct Udphdr *)(ihdr+1);
        if (uhdr->uh_dport != htons(DhcpClientPort)) {
            return;
        }
}

/* Verify incoming IP header checksum...
 */
csum = 0;
datap = (ushort *) ihdr;
for (i=0;i<(sizeof(struct ip)/sizeof(ushort));i++,datap++)
        csum += *datap;
csum = (csum & 0xffff) + (csum >> 16);
if (csum != 0xffff) {
        EtherIPERRCnt++;
        return;
}

printPkt(ehdr,size,ETHER_INCOMING);
```

下面介绍的是覆盖这个 IP 结构（见程序清单 9.3）到收到的包上，然后进行一些确认。首先，校验包与 IP 一致；然后，收到的包被定位为本地的 IP 地址；最后，校验匹配这个包。如果包通过了所有的确认，则调用 **printPkt**（）来显示这个包［只在允许冗长时，printPkt（）才可以打印］。

程序清单 9.4　Dispatching for Protocol-Specific Processing

```
if (ihdr->ip_p == IP_ICMP) {
        processICMP(ehdr,size);
        return;
}
else if (ihdr->ip_p == IP_TCP) {
        processTCP(ehdr,size);
        return;
}
```

```
else if (ihdr->ip_p != IP_UDP) {
        int j;

        SendICMPUnreachable(ehdr,ICMP_UNREACHABLE_PROTOCOL);
        if (!(EtherVerbose & SHOW_INCOMING))
                return;
        for(j=0;protocols[j].pname;j++) {
                if (ihdr->ip_p == protocols[j].pnum) {
                        printf("%s not supported\n",
                                protocols[j].pname);
                        return;
                }
        }
        printf("<%02x> protocol unrecognized\n", ihdr->ip_p);
        return;
}
```

程序清单 9.4 中的代码通过检查 IP 包的类型来确定怎样（是否全部）处理到来的 IP 包。这些检查与程序清单 9.2 中的以太网结构检查相同。微控制器支持 UDP、ICMP 和最小的 TCP，所以逻辑会为 ICMP 调用 processICMP（），为 TCP 调用 processTCP（）。如果到来的包不是 UDP，则微监控器会发送 ICMP 未取到错误回应。

程序清单 9.5　Processing UDP Packets

```
uhdr = (struct Udphdr *)(ihdr+1);

/* If non-zero, verify incoming UDP packet checksum...
 */
if (uhdr->uh_sum) {
        int         len;
        struct   UdpPseudohdr       pseudohdr;

        memcpy((char *)&pseudohdr.ip_src.s_addr,
                (char *)&ihdr->ip_src.s_addr,4);
```

```
memcpy((char *)&pseudohdr.ip_dst.s_addr,
        (char *)&ihdr->ip_dst.s_addr,4);
pseudohdr.zero = 0;
pseudohdr.proto = ihdr->ip_p;
pseudohdr.ulen = uhdr->uh_ulen;

csum = 0;
datap = (ushort *) &pseudohdr;
for (i=0;i<(sizeof(struct UdpPseudohdr)/sizeof(ushort));i++)
        csum += *datap++;

/* If length is odd, pad and add one. */
len = ntohs(uhdr->uh_ulen);
if (len & 1) {
        uchar     *ucp;
        ucp = (uchar *)uhdr;
        ucp[len] = 0;
        len++;
    }
    len >>= 1;

    datap = (ushort *) uhdr;
    for (i=0;i<len;i++)
        csum += *datap++;
    csum = (csum & 0xffff) + (csum >> 16);
    if (csum != 0xffff) {
        EtherUDPERRCnt++;
        return;
    }
}
udpport = ntohs(uhdr->uh_dport);

if (udpport == MoncmdPort)
    processMONCMD(ehdr,size);
```

```
    else if (udpport == DhcpClientPort)
        processDHCP(ehdr,size);
    else if ((udpport == TftpPort) || (udpport == TftpSrcPort))
        processTFTP(ehdr,size);
    else {
        if (EtherVerbose & SHOW_INCOMING) {
            uchar *cp;
            cp = (uchar *)&(ihdr->ip_src);
            printf("   Unexpected IP pkt from %d.%d.%d.%d ",
                cp[0],cp[1],cp[2],cp[3]);
            printf("(sport=0x%x,dport=0x%x)\n",
                ntohs(uhdr->uh_sport),ntohs(uhdr->uh_dport));
        }
        SendICMPUnreachable(ehdr,ICMP_UNREACHABLE_PORT);
    }
}
```

程序清单 9.5 中的过程与程序清单 9.3 中的相同，是协议的更高一层。这些代码覆盖 UDP 结构在 IP 载荷上，并且运行 UDP 头和伪头的校验。如果到来的包已将校验值设成零，则这样的校验可能就不是必须的。处理器处理到来的 UDP 端口数，从而转向处于微控制器软件（TFTP、DHCP 或 MONCMD）更深层的恰当代码。如果这个端口微控制器并不支持，处理器便会返回 ICMP 错误。

注意

MONCMD 是控制器的 777 端口。这个端口支持越过 UDP 执行 CLI 命令。

9.7.2　总结

驱动器代码是在 C 中执行的，结构被覆盖在收到的包上，并使包的分析简单化。例如，程序清单 9.6 中的结构可以在 IP 包处理中使用。

程序清单 9.6 包含了一些用户应该了解的细节。首先，依赖于包在目标存储器空间的结束位置，考虑不同的 CPU 分配请求。例如，除非整数在偶地址上，否则

一些 CPU 可能会看不到 2B 或 4B 的整数（分别为短整型和长整型）。如果数据在一个奇地址上而 C 代码想访问它，则处理器会产生“寻址意外”。知道了这一点，就可以在必要时做出调整。如果你不预测存储缓冲中的排列，就只能在访问之前拷贝 2B 或 4B 的数据到一个排好的位置。

程序清单 9.6　A Structure Declaration for the IP Header

```
struct ip_header {
    uchar   ip_vhl;         /* version and header length      */
    uchar   ip_tos;         /* type of service                */
    ushort ip_len;          /* length of packet               */
    ushort ip_id;           /* identification                 */
    ushort ip_offset;       /* fragment offset field          */
    uchar   ip_ttl;         /* time to live                   */
    uchar   ip_proto;       /* protocol                       */
    ushort ip_csum;         /* checksum                       */
    ulong   ip_source;      /* source IP address              */
    ulong   ip_dest;        /* destination IP address         */
}
```

目标的 CPU 可能是前或后字节排列顺序，增加了复杂程度。网络上所有的数据均是按网络字节顺序传输的，且是前字节排列顺序。如果你幸运地使用的是前字节排列的 CPU，则前字节排列顺序的数据就没有什么问题；如果 CPU 是后字节排列，则在头中的包长度、校验和其他多字节量必须被排列转换。网络到主机长整型（ntohl）、主机到网络长整型（htonl）、网络到主机短整型（ntohs）和主机到网络短整型（htons）的转换宏完成了这种字节的排列顺序转换。在前字节排列顺序的机器上，这些宏没有什么用；在后字节排列顺序的机器上，它们就可完成这些转换。

9.8　小结

本章的目的不是建立整个网络结构，而是履行足够的网络协议。这足以使嵌入式系统能够和网络的其他部分对话。以太网驱动（与我们的串行接口驱动相同）是按轮询模式写的，同一时间内只能有一个包被处理。因为这个系统甚至没有中断，无需担心排列复合包。当轮询确定收到一个包时，软件即处理这个包，如果正确，

就发送回应，然后等下一个包。这种途径使处理简单化，且在基本原理的“初步”，也是驱动器的第一步工作。这一阶段学的东西可以在下一阶段（如果需要）的中断驱动中使用，即使是在中断的最终也会用到。这一原理在前述的任何驱动中都用得上。

第 10 章　文件/数据传输

现在的监控器包括 flash 系统、串口通信和以太网的连接，可提供一个允许文件与目标机来回传输的机构。本章给出的文件传输协议虽不是最新的，但由于其简明性仍被广泛使用。监控器使用 Xmodem 和 TFTP 分开 RS-232 和以太网传输协议。两种协议数据均可作为原始数据或文件来传输。MicroMonitor 程序包包括基于 PC 的 TFTP 用户/服务器和基于 PC 的 Xmodem 传输。本章将讨论的仅限于目标方面，完整执行的细节超出了讨论的范围（在本书附赠的 CD 中可找到完整的源程序）。

Xmodem 和 TFTP 都是同步锁定传输协议，意味着当一个终端发送数据时，直到收到另一个终端数据已收到的确认信息后，才能传输新数据。尽管这种设计会降低速度，特别是 TFTP，但同步锁定近似法提供了简明性的优点。

10.1　Xmodem

Xmodem 出现后，即被广泛使用。由于它用在很多方面，因此目前还在一直被使用。Xmodem 涉及到许多变量，从而可以支持 CRC、校验和、128B 与 1024B 的包及多路文件的传输。在此不讨论这些变量，因为本章的重点是 MicroMonitor's Xmodem——而不是更大的协议。注意，MicroMonitor 假设目标机与主机通过无错连接通信及 Xmodem 强制执行。讨论的重点在于让你感觉 Xmodem 协议好像用于 MicroMonitor 中，并让你体验自然环境下的基本 Xmodem。

Xmodem 基于两个重要的函数，即 Xup（）和 Xdown（）。这些函数分别提供下载和上传的功能，使用相同的数据结构来描述和跟踪传输。程序清单 10.1 给出了这一结构的说明。

程序清单 10.1　Xmodem 的控制结构

```
struct xinfo {
    uchar   sno;        /* Sequence number.                    */
    uchar   pad;        /* Unused, padding.                    */
    int     xfertot;    /* Running total of transfer.          */
    int     pktlen;     /* Length of packet (128 or 1024).     */
```

```
    int     pktcnt;     /* Running tally of number of packets processed.    */
    int     filcnt;     /* Number of files transferred by ymodem.           */
    long    size;       /* Size of upload.                                  */
    ulong   flags;      /* Storage for various runtime flags.               */
    ulong   base;       /* Starting address for data transfer.              */
    ulong   dataddr;    /* Running address for data transfer.               */
    int     errcnt;     /* Keep track of errors (used in verify mode).      */
    char    *firsterrat; /* Pointer to location of error detected when      */
                        /* transfer is in verify mode.                      */
    char    fname[TFSNAMESIZE];
};
```

Xmodem 的传输不论是上传还是下载，均以 MicroMonitor CLI 命令开始。一旦命令被设定，Xmodem 程序将等待下一个结束（通常是 Windows 中的 HyperTerminal）来连接并开始实质传输。

程序清单 10. 2 给出了用户需要把文件从目标机传到主机时使用的 Xup（）的功能。

程序清单 10.2　Xup()

```
static int
Xup(struct xinfo *xip)
{
    uchar    c, buf[PKTLEN_128];
    int      done, pktlen;
    long     actualsize;

    Mtrace("Xup starting");

    actualsize = xip->size;

    if (xip->size & 0x7f) {
        xip->size += 128;
        xip->size &= 0xffffff80L;
    }
```

```
printf("Upload %ld bytes from 0x%lx\n",xip->size,(ulong)xip->base);

/* Startup synchronization... */
/* Wait to receive a NAK or 'C' from receiver. */
done = 0;
while(!done) {
    c = (uchar)getchar();
    switch(c) {
    case NAK:
        done = 1;
        Mtrace("CSM");
        break;
    case 'C':
        xip->flags |= USECRC;
        done = 1;
        Mtrace("CRC");
        break;
    default:
        break;
    }
}

done = 0;
xip->sno = 1;
xip->pktcnt = 0;
while(!done) {
    c = (uchar)putPacket((uchar *)(xip->dataddr),xip);
    switch(c) {
    case ACK:
        xip->sno++;
        xip->pktcnt++;
        xip->size -= xip->pktlen;
        xip->dataddr += xip->pktlen;
        Mtrace("A");
```

```
            break;
        case NAK:
            Mtrace("N");
            break;
        case CAN:
            done = -1;
            Mtrace("C");
            break;
        case EOT:
            done = -1;
            Mtrace("E");
            break;
        default:
            done = -1;
            Mtrace("<%2x>",c);
            break;
        }
        if (xip->size <= 0) {
            rputchar(EOT);
            getchar();   /* Flush the ACK */
            break;
        }
        Mtrace("!");
    }
    Mtrace("Xup_done.");
    return(0);
}
```

注意

Mtrace（）函数（见程序清单 10.2）需要一个快速注释。Xmodem 程序使用 Mtrace（）来帮助调试，使用与 printf（）相同的串口，所以不能加 printf（）语句来调试程序，因为加 printf（）语句违反了协议。Mtrace（）（或存储流程）函数与 printf（）相似，除了输出到控制台，还能保留到内部 RAM（随机存储器）的缓冲

器。协议完成时，通过选项可以转存该缓冲器的内容，从而可以看到发生的流程记录。Mtrace（）对于编写不容许串口打印的程序而言是很有用的工具。现在让我们回到 Xup（）。

立即批处理函数必须处理 Xmodem“yukism”传输大小必须转换成 mod-128 的大小。Xmodem 的最小数据包为 128B，若实际传输大小为 129B，则 Xmodem 发送 256B。延长了数据长度后，可通过收到另一个结束信号来进行连接，从而与接受端同步。在串口连接的另一端，程序用发送 NAK（负数确认）来启动使用求和校验的传输或发送字符 C 启动使用 CCITT CRC16 的传输。当同步完成时，数据包开始流动。Xmodem 工具可能是所能得到的最简单的工具了。函数 Xup（）发送每个数据包后，使用 putPacket（）函数等待响应。putPacket（）函数（未给出）发送 SOH（标题开头）字符，增量序列数值为 1，并发送序列数值和它的反码及数据包数据和 CHECKSUM 或 CRC，然后等待响应，并返回 Xup（）函数。如果响应是正确确认（ACK），则 Xup（）继续下一个数据包；否则将停止，没有重发。如果传输失败，则用户要重试。

10.1.1 Xdown（）

Xdown（）是当用户需要文件从主机传到目标机时，由 Xmodem 命令调用的函数。

程序清单 10.3 Xdown()

```
/* Xdown():
 * Called when a transfer from host to target is being made (considered
 * a download).
 */

static int
Xdown(struct xinfo *xip)
{
    long    timeout;
    char    c, tmppkt[PKTLEN_1K];
    int     done;
```

```
xip->sno = 0x01;
xip->pktcnt = 0;
xip->errcnt = 0;
xip->xfertot = 0;
xip->firsterrat = 0;

/* Startup synchronization... */
/* Continuously send NAK or 'C' until sender responds. */
Mtrace("Xdown");
while(1) {
    if (xip->flags & USECRC)
        rputchar('C');
    else
        rputchar(NAK);
    timeout = LoopsPerSecond;
    while(!gotachar() && timeout)
        timeout--;
    if (timeout)
        break;
}

done = 0;
Mtrace("Got response");
while(done == 0) {
    c = (char)getchar();
    switch(c) {
    case SOH:                    /* 128-byte incoming packet */
        Mtrace("O");
        xip->pktlen = PKTLEN_128;
        done = getPacket(tmppkt,xip);
        break;
    case STX:                    /* 1024-byte incoming packet */
        Mtrace("T");
```

```
            xip->pktlen = PKTLEN_1K;
            done = getPacket(tmppkt,xip);
            break;
        case CAN:
            Mtrace("C");
            done = -1;
            break;
        case EOT:
            Mtrace("E");
            rputchar(ACK);
            done = xip->xfertot;
            printf("\nRcvd %d pkt%c (%d bytes)\n",xip->pktcnt,
                xip->pktcnt > 1 ? 's' : ' ',xip->xfertot);
            break;
        case ESC:           /* User-invoked abort */
            Mtrace("X");
            done = -1;
            break;
        default:
            Mtrace("<%02x>",c);
            done = -1;
            break;
        }
        Mtrace("!");
    }
    if (xip->flags & VERIFY) {
        if (xip->errcnt)
            printf("%d errors, first at 0x%lx\n",
                xip->errcnt,(ulong)(xip->firsterrat));
        else
            printf("verification passed\n");
    }
    return(done);
}
```

因为 Xdown（）是协议的另一种结果，所以 Xdown（）看上去像 Xup（）。程序清单 10.3 的开头是同步步骤，代码传送 C 或 NAK（取决于传输的配置），然后等待回应。一旦收到回应信号，数据包又一次开始流动，不过这次数据包却是向另一个方向流动（流进主机存储器），在传输过程中，数据包被 SOH 或 STX 标识。SOH 指出一个即将开始的 128B 数据包，STX 表示 1024B 返回的数据包。getPacket（）函数（未给出）所做的工作是得到数据包并发出回应信号［getPacket（）作用与 Xup（）中的 putPacket（）一样］。当发信人完成传送时，发信人发出一个终止传输（EOT）标志来表示传输结束。收到 EOT 标志使得循环终止，完成传输，但也有其他的一些例外情况也可中断传输。

注意，紧跟循环后的 VERIFY 标志检验是 MicroMonitor's Xmodem 执行的一个重要选项。这个选项允许主机下载并检验（不重写，仅比较）所下载数据与目标机中已存在的数据。

10.1.2　MicroMonitor 中的 Xmodem

Xmodem 使得 MicroMonitor 能够在目标之间传送文件。文件传送可以是目标中来/回的文件传送或来/回的未加工的存储内容传送，前边提到的检验选项对简单的存储器测试或检验代码没有覆盖数据块都很有用。经实际检验，MicroMonitor 中的 Xmodem 可支持大部分选项，例如 128 或 1024B 数据包、校验和 CRC 及 Ymodem 扩展等等。详情请见 CD。

MicroMonitor 中的 Xmodem 还有一个特别要注意的特点，其引导监控器驻留在引导闪存中，有时引导闪存还需要刷新，可以用一个程序来驱动新装置并安装它。MicroMonitor's Xmodem 也可钩住闪存驱动代码且容许用户下载二进制映像并自动重写板上的监控器代码。这一特性非常方便，特别是当你把监控器移到新目标时。串口和闪存驱动到位后，可以在对监控器端口做其他工作时，利用 Xmodem 快速启动闪存刷新。

10.2　TFTP

小文件传输协议（TFTP）可看成是以太网中的 Xmodem，是一个基础锁定步骤协议，支持使用 UDP 传输的文件。由于 MicroMonitor 支持 TFTP 用户和服务器，因此可以使用服务器传输文件到各个方向，用户只需从远程服务器取回文件。像 Xmodem 一样，TFTP 也是一个大话题，本书不能详述，但会提及一些支持 TFTP

的 MicroMonitor 函数。

ProcessTFTP（）函数（见程序清单 10.4）被 processPACKET（）（见程序清单 9.5）调用。为了 TFTP 传输， ProcessTFTP（）必须处理 5 种标准要求（或操作码）：

- TFTP_RRQ——读要求。远程系统要求文件或数据从目标机传输到远程系统。
- TFTP_WRQ——写要求。远程系统要求文件或数据从远程系统传输到目标机。
- TFTP_DAT——将开始的数据包作为已收到的预先 WRQ 或 DAT 的结果。
- TFTP_ACK——远程系统已收到数据包，意味着下一个数据包可以传送。
- TFTP_ERR——一些类型的错误发生。

程序清单 10.4 后半部分的 Switch 语句用于分配处理每一个操作码。

程序清单 10.4 processTFTP()

```
int
processTFTP(struct ether_header *ehdr,ushort size)
{
    static  uchar    *oaddr;
    struct  ip *ihdr;
    struct  Udphdr *uhdr;
    uchar    *data;
    int      count, tmpcount;
    ushort   opcode, block, errcode;
    char     *comma, *tftpp, *filename, *mode, *errstring, msg[64];

    if (TftpTurnedOff) {
        SendICMPUnreachable(ehdr,ICMP_UNREACHABLE_PORT);
        return(0);
    }

    ihdr = (struct ip *)(ehdr + 1);
    uhdr = (struct Udphdr *)((char *)ihdr + IP_HLEN(ihdr));
    tftpp = (char *)(uhdr + 1);
    opcode = *(ushort *)tftpp;
```

```
switch (opcode) {
```

注意，processTFTP（）（见程序清单 10.4）与 processPacket（）有相同的自变量，其功能是在引入的数据包上做许多结构覆盖图。如果 TFTP 工具由于某种原因不能用，则 TftpTurnedoff 标志将被设定。当标志被设定后，目标机就用一个 ICMP 不能达到的端口信息来回应引入的数据包。

程序清单 10.5　启动下载

```
case htons(TFTP_WRQ):
    filename = tftpp+2;
    if ((EtherVerbose & SHOW_TFTP) || (!MFLAGS_NOTFTPPRN()))
        printf("TFTP rcvd WRQ: file %s\n", filename);

    if (!tftpStartSrvrFilter(ehdr,uhdr))
        return(0);

    mode = filename;
    while(*mode)
        mode++;
    mode++;

    /* Destination of WRQ can be an address (0x...), environment
     * variable ($...) or a TFS filename...
     */
    if ((filename[0] == '$') && (getenv(&filename[1]))) {
        TftpAddr = (uchar *)strtol(getenv(&filename[1]),(char **)0,0);
    }
    else if ((filename[0] == '0') && (filename[1] == 'x')) {
        TftpAddr = (uchar *)strtol(filename,(char **)0,0);
    }
    else {
        if (MFLAGS_NOTFTPOVW() && tfsstat(filename)) {
            SendTFTPErr(ehdr,6,"File already exists.",1);
            return(0);
```

```
        }
        TftpAddr = (uchar *)getAppRamStart();
        strncpy(TftpTfsFname,filename,sizeof(TftpTfsFname) -1);
        TftpTfsFname[sizeof(TftpTfsFname) -1] = 0;
    }
    TftpCount = -1; /* not used with WRQ, so clear it */

    /* Convert mode to lower case... */
    strtolower(mode);
    if (!strcmp(mode,"netascii"))
        TftpWrqMode = MODE_NETASCII;
    else if (!strcmp(mode,"octet"))
        TftpWrqMode = MODE_OCTET;
    else {
        SendTFTPErr(ehdr,0,"Mode not supported.",1);
        TftpWrqMode = MODE_NULL;
        TftpCount = -1;
        return(0);
    }
    block = 0;
    tftpLastblock = block;
    oaddr = TftpAddr;
    TftpChopCount = 0;
    break;
```

第一个操作码是 TFTP_WRQ（见程序清单 10.5）。注意，switch 检测 htons（TFTP_WRQ），但不仅仅是 TFTP_WRQ，参见宏定义，引入操作码数值与目标上的操作码数值相等。第一步检测本地 TFTP 服务器状态是否正常，并可以处理 WRQ 要求（调用 TftpStartSrvrFilter（），未给出）。若状态不对，则回应 TFTP_ERR 信号（在 TftpStartSrvrFilter（）中），返回值为 0。

引入数据包包括文件名和文件传输模式。这一执行允许 3 种类型的文件名和两种不同模式。

文件选项：

- 一个标准名。在这种状态下，数据首先被传送到 RAM（静态存储器），然

后通过 tfsadd（）传至 TFS。

- 一个十六进制地址。假设从 0x 开始，在这种状态下，数据被直接传送到指定位置。
- 一个存在外部变量中的地址。假设以$开始，在这种状态下，数据被传送到外部变量中的十六进制地址中。

模式选项：

- netascii 传输 ASCII 文件，引入\r\n 代替\n。
- octet 传输二进制文件，引入数据是未改变的。

这些选项使用户可以通过硬编码地址或外部变量内容直接传送到 RAM（静态存储器）（这并不是说不能传送文件名以 0x 或$开头的文件，但这种限制是有道理的）。注意，宏 MFLAGS_NOTFTPOVW（）用来测试所要求的文件是否传送到一个已经存在的文件。若系统标志被确定为不容许通过 TFTP 重写一个已存在的文件，则监控器返回 TFTP_ERR。这就是 TFTP_WRQ，服务器状态提前到“接收”状态并且已确认请求（通过 ACK）。

程序清单 10.6　启动上传

```
case htons(TFTP_RRQ):
    filename = tftpp+2;
    if ((EtherVerbose & SHOW_TFTP) || (!MFLAGS_NOTFTPPRN()))
        printf("TFTP rcvd RRQ: file %s\n",filename);
    if (!tftpStartSrvrFilter(ehdr,uhdr))
        return(0);
    mode = filename;
    while(*mode) mode++;
    mode++;
    comma = strchr(filename,',');
    if (!comma) {
        TFILE    *tfp;
        tfp = tfsstat(filename);
        if (!tfp) {
            SendTFTPErr(ehdr,0,"File not found, try 'address,count' ",1);
            TftpCount = -1;
            return(0);
        }
```

```
        TftpAddr = (uchar *)TFS_BASE(tfp);
        TftpCount = TFS_SIZE(tfp);
    }
    else {
        comma++;
        TftpAddr = (uchar *)strtol(filename,(char **)0,0);
        TftpCount = strtol(comma,(char **)0,0);
    }
    if (strcmp(mode,"octet")) {
        SendTFTPErr(ehdr,0,"Must use binary mode",1);
        TftpCount = -1;
        return(0);
    }
    block = tftpLastblock = 1;
    tftpGotoState(TFTPACTIVE);
    SendTFTPData(ehdr,block,TftpAddr,TftpCount);
    return(0);
```

程序清单 10.6 给出了进程 TFTP_RRQ 所要求的代码。代码又一次以一些状态校验开始［调用 tftpStartSrvrFilter（）］。引入数据包的格式与 TFTP_WRQ 所确认的相似。这一实现允许两种不同的文件名语法：

- 一个标准文件名。在这种情况下，数据从 TFS 的一个文件中读出。
- 一个十六进制的地址和大小，其格式为 0xADDR，SIZE。这种语法支持使用 TFTP 的用户从主机上检索一大块未加工的存储单元（再来一次，语法将排除某些文件名）。

在决定了地址和所要传送的数据长度后，代码开始检查模式，且只支持 octet，随后，存入状态，发送第一个数据块，监控器进入等待状态，挂起即将开始的 TFTP_ACK 操作码。

程序清单 10.7　接收数据包

```
    case htons(TFTP_DAT):
            block = ntohs(*(ushort *)(tftpp+2));
            count = ntohs(uhdr->uh_ulen) - (sizeof(struct Udphdr)+4);

            if (EtherVerbose & SHOW_TFTP)
```

```
        printf("    Rcvd TFTP_DAT (%d,blk=%d)\n",count,block);

    /* This TFTP_DAT may be from a local "get" request... */
    if (TftpState == TFTPSENTRRQ) {
        tftpLastblock = 0;
        if (block == 1) {
            TftpRmtPort = ntohs(uhdr->uh_sport);
            tftpGotoState(TFTPACTIVE);
        }
        else {
            SendTFTPErr(ehdr,0,"invalid block",1);
            return(0);
        }
    }
    /* Since we don't ACK the final TFTP_DAT from the server until after
     * the file has been written, it is possible that we will receive
     * a re-transmitted TFTP_DAT from the server.    This is ignored by
     * Sending another ACK...
     */
    else if ((TftpState == TFTPIDLE) && (block == tftpLastblock)) {
        SendTFTPAck(ehdr,block);
        if (EtherVerbose & SHOW_TFTP)
            printf("    (packet ignored)\n");
        return(0);
    }
    else if (TftpState != TFTPACTIVE) {
        SendTFTPErr(ehdr,0,"invalid state",1);
        return(0);
    }

    if (ntohs(uhdr->uh_sport) != TftpRmtPort) {
        SendTFTPErr(ehdr,0,"invalid source port",0);
        return(0);
    }
```

```
if (block == tftpLastblock) {          /* If block didn't increment, assume */
    SendTFTPAck(ehdr,block);           /* retry.   Ack it and return here.  */
    return(0);                         /* If block != tftpLastblock+1,      */
}                                      /* return an error, and quit now.    */
else if (block != tftpLastblock+1) {
    SendTFTPErr(ehdr,0,"Unexpected block number",1);
    TftpCount = -1;
    return(0);
}
TftpCount += count;
oaddr = TftpAddr;
tftpLastblock = block;
data = (uchar *)(tftpp+4);

/* If count is less than TFTP_DATAMAX, this must be the last
 * packet of the transfer, so clean up state here.
 */
if (count < TFTP_DATAMAX) {
    tftpGotoState(TFTPIDLE);
}
```

程序清单 10.7 给出了如何处理 TFTP_DAT 操作码。代码检测 TftpState 来决定从哪一方启动传输。如果这个状态变量已被发送到 TFTPSENTRRQ，则当前的数据包是目标机 TFTP 用户发送“tftp get”到远程服务器的结果。否则，数据包将作为来自远程用户 TFTP_WRQ 的回应被传送。代码必须包括远程机器发送并复制的 TFTP_DAT 的情况。通常，由于到 TFS 的写操作太大，以至于在目标结束传输数据到 flash 之前，远程主机已暂停，数据包会被重新传输。注意，接收计算机应该在文件已被成功地写到 TFS 后才发送 ACK 信号，因此大文件可以引起重新发送。最后的状态检测必须确定系统已经进入等待数据的模式。进一步的检测是看发送数据的主机与先前发送 RRQ 的主机是否相同。接收代码也要检测数据块序号。

程序清单 10.8　保存数据

```
/* Copy data from enet buffer to TftpAddr location... */
    tmpcount = count;
```

```
while(tmpcount) {
    if (TftpWrqMode == MODE_NETASCII) {
        if (*data == 0x0d) {
            data++;
            tmpcount—;
            TftpChopCount++;
            continue;
        }
    }

    *TftpAddr = *data;
    if (*TftpAddr != *data) {
        sprintf(msg,"Write error at 0x%lx",(ulong)TftpAddr);
        SendTFTPErr(ehdr,0,msg,1);
        TftpCount = -1;
        return(0);
    }
    TftpAddr++;
    data++;
    tmpcount—;
}

/* If the transfer is complete and TftpTfsFname[0] is non-zero,
 * then write the data to the specified TFS file... Note that a
 * comma in the filename is used to find the start of (if any)
 * the TFS flags string.  A second comma, marks the info field.
 */
if ((count < TFTP_DATAMAX) && (TftpTfsFname[0])) {
    char *fcomma, *icomma, *flags, *info;
    int err;

    info = (char *)0;
    flags = (char *)0;
    fcomma = strchr(TftpTfsFname,',');
```

```
if (fcomma) {
    icomma = strchr(fcomma+1,',');
    if (icomma) {
        *icomma = 0;
        info = icomma+1;
    }
    if (tfsctrl(TFS_FATOB,(long)(fcomma+1),0) != -1) {
        *fcomma = 0;
        flags = fcomma+1;
    }
    else {
        SendTFTPErr(ehdr,0,"Invalid flag spec.",1);
        TftpTfsFname[0] = 0;
        break;
    }
}
if ((EtherVerbose & SHOW_TFTP) || (!MFLAGS_NOTFTPPRN()))
    printf("TFTP adding file: '%s' to TFS.\n",TftpTfsFname);
err = tfsadd(TftpTfsFname,info,flags,
    (char *)getAppRamStart(),TftpCount+1-TftpChopCount);
if (err != TFS_OKAY) {
    sprintf(msg,"TFS err: %s",
        (char *)tfsctrl(TFS_ERRMSG,err,0));
    SendTFTPErr(ehdr,0,msg,1);
}
TftpTfsFname[0] = 0;
}
break;
```

在所有 TFTP_DAT 状态检测完成后，程序清单 10.8 的代码假设从以太网数据缓冲空间到一些其他的目标存储单元的数据传输已经开始了。若 netascii 模式已经被确立，那么 0x0d(\r)标志必须被撤消。传输的数据包长度要小于 TFTP-DATAMAX(512)的最大长度，它标志着传输结束。如果正常，则数据应被传送到 TFS 的文件中。

注意

为了支持 TFS 的标志和信息，文件名可以被逗号分隔，如 filename、flags 及 info，其中，filename 是文件名；flags 是一串与文件有关的所有的标志（或属性）；info 是与 TFS 文件有关的信息（在这个逗号分隔串中，空格是非法的）。这种格式允许标准的 TFTP 用户通过字符串的标志和信息指定目标文件（在 TFS 中），惟一的（合理的）限制是逗号必须被用做分隔符。

程序清单 10.9　处理和 ACK

```
case htons(TFTP_ACK):
    block = ntohs(*(ushort *)(tftpp+2));
    if (TftpState != TFTPACTIVE) {
        SendTFTPErr(ehdr,0,
            "Illegal server state for incoming TFTP_ACK",1);
        return(0);
    }
    if (EtherVerbose & SHOW_TFTP)
        printf("    Rcvd TFTP_ACK (blk#%d)\n",block);

    if (block == tftpLastblock) {
        if (TftpCount > TFTP_DATAMAX) {
            TftpCount -= TFTP_DATAMAX;
            TftpAddr += TFTP_DATAMAX;
            SendTFTPData(ehdr,block+1,TftpAddr,TftpCount);
            tftpLastblock++;
        }
        else if (TftpCount == TFTP_DATAMAX) {
            TftpCount = 0;
            tftpGotoState(TFTPIDLE);
            SendTFTPData(ehdr,block+1,TftpAddr,0);
            tftpLastblock++;
        }
```

```
        else {
            TftpAddr += TftpCount;
            TftpCount = 0;
            tftpGotoState(TFTPIDLE);
        }
    }
    else if (block == tftpLastblock-1) {
        SendTFTPData(ehdr,block+1,TftpAddr,TftpCount);
    }
    else {
        SendTFTPErr(ehdr,0,"Blockno confused",1);
        TftpCount = -1;
        return(0);
    }
    return(0);
```

TFTP_ACK 信息通知收件人先前的数据包已收到，并为下一个数据包准备好另一个的结束。每一个 ACK 均可潜在地触发另一个数据包传送。像其他处理一样，这一代码以检测开始。若一切就绪，则代码决定是否有足够的数据填充数据包。否则，这个数据包将作为最后一个。最后一个 512B 的数据包被作为特殊情况处理。TftpCount 保存了所传输数据的来源地址。若数据块序号正确，则处理器发送一个新的数据包。若数据块序号比预想的小 1，则代码重新发送先前的数据包，且其余的数据块序号都会触发一个错误响应。

程序清单 10.10　操作错误

```
    case htons(TFTP_ERR):
        errcode = ntohs(*(ushort *)(tftpp+2));
        errstring = tftpp+4;
        if (EtherVerbose & SHOW_TFTP)
            printf("    Rcvd TFTP_ERR #%d (%s)\n",errcode,errstring);
        TftpCount = -1;
        tftpGotoState(TFTPERROR);
        strncpy(TftpErrString,errstring,sizeof(TftpErrString) -1);
        TftpErrString[sizeof(TftpErrString) -1] = 0;
        return(0);
```

```
        default:
            if (EtherVerbose & SHOW_TFTP)
                printf("    Rcvd <%04x> unknown TFTP opcode\n", opcode);
            SendTFTPErr(ehdr,0,"Unexpected opcode received.",1);
            TftpCount = -1;
            return(-1);
        }
        SendTFTPAck(ehdr,block);
        return(0);
    }
```

TFTP_ERR 操作码需要非常小的程序设计，就像程序清单 10.10 所示。头部只不过把状态改变为错误，记下引入的错误，打印错误信息到控制台。程序清单 10.10 还给出了对于不希望发生的操作码的默认处理。

10.3　自升级功能

本章我们讨论了一些关于 Xmodem 和 TFTP 的执行细节。两个协议都是很可靠的，而且在 MicroMonitor 包中这两个协议都提供了在目标系统中升级文件的功能。本章所描述的代码可以做 MicroMonitor 顶部的应用程序的一个域升级机制的基础。就像我们把 TFS 和 CLI 结合起来，形成第 8 章的脚本运行器一样，你可以把 TFS 与 Xmodem 或 TFTP 结合（或两个一起）形成域升级机制。应用程序根据自身的需要可以不通过监控器的功能而执行升级，也可以自己成为升级机制的一部分。下面将讨论这两种情况。

10.3.1　应用程序不知道潜在升级路径

应用程序升级的简单方法是使应用程序不知道升级的路径。这种模式与 PC 上的其他程序或其他主机系统使用的模式相似。程序被安装在系统上进行工作。若工作改变或发现故障，则程序停止，一个新的程序被安装。新的程序被重启，但程序代码本身完全不知道刚发生的安装操作，许多平台（例如 Windows，DOS，UNIX）能够提供这个功能。MicroMonitor 能够提供这个功能，而且目标系统一般具有串口或以太网连接。

通过这种方法，应用程序以可执行文件的形式驻留在 TFS 上。文件是自启动

的，所以当 MicroMointor 启动时，它自动启动应用程序。此时，监控器提供了一个允许自启动序列中止的机制。当自启动被中止时，MicroMonitor 以独立模式运行。TFTP 和 Xmodem 工具在监控器上工作，主机可以下载新的应用程序文件替换现在的应用程序。下载完毕，可以重启目标系统，自启动序列开始执行新的应用程序。

10.3.2 应用程序是自升级的一部分

另一种方法是在应用程序中建立 Xmodem 或 TFTP 代码。当应用程序运行时，这一工具可用于下载文件到 TFS 闪存。TFS 的执行被存在闪存中，但在 RAM（随机存储器）外执行，所以 TFS 中的文件在应用程序运行时可以被控制。这一操作的优点是在执行升级的过程中，应用程序可以不变；在不运行的时间里，目标系统可以复位，TFS 中的新应用程序可以自启动。

10.4 小结

当使用外部程序进行开发时，若可以下载新的代码，并将它直接装到目标系统，并使用内部电路编程功能来刷新闪存，则开发过程将非常有效率，因此首先是应得到一些文件传输工具。MicroMonitor 支持两个最通用的文件传输协议 Xmodem 和 TFTP。

一般情况下，当新安装 MicroMonitor 时，我们可通过串口和 Xmodem 进行，既使底板包括一个以太网接口，一般也使用串口（若提供）进行初始开发工作。Xmodem 代码更小、更简单，并且 Mtrace 的特性也使得调试连接更容易，基于主机的开发工具更倾向于与主机的串行连接而不是网络连接，因此当我们有更高级的调试工具时，可以安装 TFTP 支持并转向 TFTP 进行文件下载。

两个协议都能提供很好的互联性。虽然 Xmodem 有意回避那些通常由面向调制解调器设计的系统中经常考虑的错误处理操作，但它还是具有良好的兼容性。只要你直接使用它，无错行，就应该可以和大部分的 Xmodem 数据包一起工作。TFTP 的功能则强大得多，而它完全遵守相关的注释要求（RFC）。甚至在没有 TFS 时或 TFS 还不完善时，这两个协议也在早期的开发阶段起着重要作用。

第 11 章　添加应用程序

我们已经多次提及监控器顶部的应用程序。监控器像 DOS 那样支持应用程序，它提供一个供应用程序运行的平台。由于应用程序可以使用监控器的 API 访问平台资源，因此与底层硬件的具体细节无关。本章将讲述应用程序与监控器是如何很好地共存于目标系统的。

11.1　各种存储映像

由于我们是在监控器的顶层建立了一个应用程序，并且监控器的模式是将应用程序从 TFS 装入 RAM，所以应用程序的存储映像变得非常简单，实际上，所有的应用程序可以在同一块 RAM 或 DRAM 中。监控器（TFS 的一部分）装入程序后可自动把 TFS 闪存中的可执行文件的内容传送到 RAM。如图 11.1 所示。

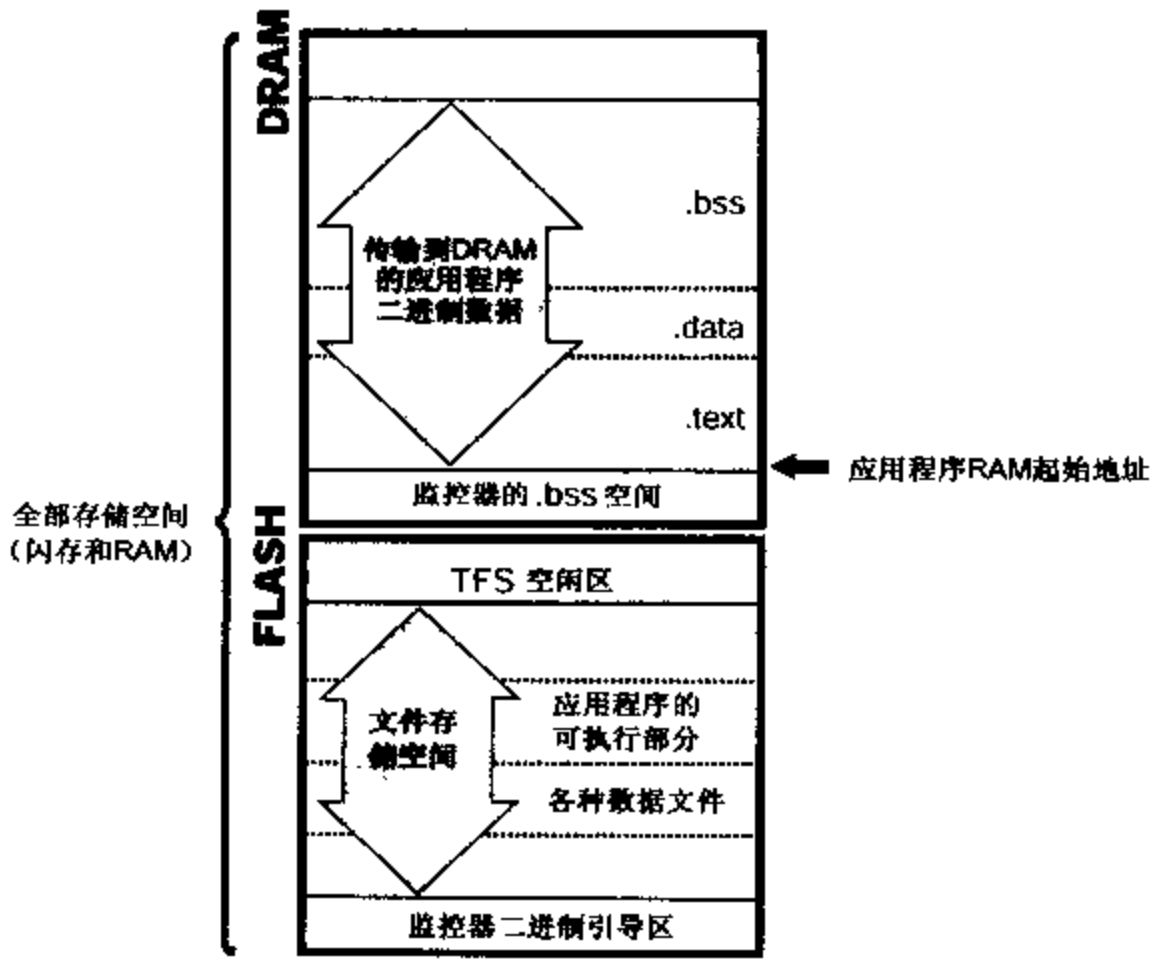


图 11.1　监控器从闪存传输文件到 DRAM

监控器访问 TFS 中的可执行应用文件，将其复制到 DRAM，并通过从 DRAM 取出的指令将控制权转给它。

11.2 弱启动

应用程序可以假设它能用尽目标系统在引导时所装入的各种资源。因此，它可以不去做许多细节工作，因为监控器在引导时已经接管了这些工作。应用程序不必去管复位向量的事。实际上，因为监控器已经初始化所有异常处理程序的句柄，所以应用程序可以忽略它们，只需要关心与自己相关的异常处理句柄就可以了。

对于监控器来讲，所有初始化后的数据都不能改写，因为初始化数据被保留在闪存中，以避免从引导闪存中再复制到 RAM。随着从 TFS 装入应用程序，这个问题也就产生了，因为在默认情况下，所有正文和数据都要从闪存复制到 RAM，所以，不用应用程序自己再去复制，所有初始化数据都变成在运行时可被重写的。

11.3 建立应用程序堆栈

当监控器将控制权传送到应用程序时，应用程序将使用分配给监控器的堆栈空间。若应用程序简单，例如单线程程序，那么这样可能就足够了，因此应用程序不用再生成一个堆栈区域。若 RTOS 在应用程序中运行，则 RTOS 的 API 将十分方便地处理每个程序的堆栈分配。

11.4 连接到监控器的 API

MicroMonitor 提供一个连接机制，通过它，应用程序可访问 MicroMonitor 的 API。连接机制与 DOS PC 中的 BIOS 表相似，目的是提供一个工具，允许应用程序使用一些监控器的函数，而应用程序并不用知道函数在监控器指令空间中的确切位置；还可使连接机制完全独立于 CPU 之外。因此，BIOS 这种方法可近似于使用 CPU 特殊的软中断，但这并不是一个选项。

正如第 4 章的内容所述，复位代码包括一个叫做 moncomptr 的标签，在程序清单 11.1 中可以看出它的情况。

程序清单 11.1 再看一次 moncomptr

```
coldstart:

    Initialize "something" to store away a state variable.
```

```
        StateOfTarget = INITIALIZE
        JumpTo warmstart

moncomptr:
        .long moncom

warmstart:
        /* Load into StateOfTarget the parameter passed
         * to warmstart as if it was the C function:
         * warmstart(unsigned long state).
         */
```

moncomptr 标志被安排在闪存中的一个位置，即使监控器重建也不移动。由于标志位置与复位矢量代码有关，因此位置不移动[1]。这个指示器的位置在内存映像图中是固定的，包含一个到函数 moncom（）的指示器。每次监控器重建，moncom（）的位置便会改变，但这没关系。由于指示器在一个固定的位置，因此函数可存取。

11.4.1　Moncom（）函数

应用程序必须知道在监控器中的固定地址（moncomptr），包含到一个函数的指示器，知道函数的 API，通过 API 便产生连线图。监控器的一些头文件与应用程序共享，源代码提供函数 monConnect（）。应用程序不需要知道监控器内存映像图的其他信息[2]。监控器中的函数叫做 moncom（），用于监控器通信。程序清单 11.2 是函数 moncom（）的片段。

程序清单 11.2　函数到函数指示器连接

```
#include "monlib.h"

int
moncom(int cmd, void *arg1, void *arg2, void *arg3)
```

1. 由于复位矢量在一个固定位置上，而 moncomptr 在一个与复位矢量地址有关的固定位置上，因此 moncomptr 的地址是固定的。
2. 不一定真实但讨论的上下文正确。应用程序需要知道监控器内存映像图，从而不使用已被监控器占用的空间。

```
{
    int retval;

    retval = 0;
    switch(cmd) {
        case GETMONFUNC_PUTCHAR:
            *(ulong *)arg1 = (ulong)rputchar;
            break;
        case GETMONFUNC_GETCHAR:
            *(ulong *)arg1 = (ulong)getchar;
            break;
        case GETMONFUNC_GOTACHAR:
            *(ulong *)arg1 = (ulong)gotachar;
            break;
        etc.....
    }
}
```

11.4.2 MonConnect（）函数

应用程序知道 moncom（）函数在监控器中的位置，因为这个函数的指示器存储在大家都知道的地址中（应用程序必须知道这个地址）。应用程序中的其他函数可以通过这一函数连接应用程序到监控器，结合点在 monConnect（）进入的地方。函数 monConnect（）用它的一个自变量作为这个地址。

程序清单 11.3　函数 monConnect()

```
static int   (*_rputchar)(), (*_getchar)(), (*_gotachar)();


/* monConnect():
 * This must be the first call by the application code to talk to the
 * monitor.   It is expecting three incoming function pointers:
 *
 * mon: Points to the monitor's _moncom function;
 *          This is a "well-known" address because the monitor and
```

```
 *          application code (two separately linked binaries) must
 *          know it.
 * lock:    Points to a function in the application code that will be
 *          used by the monitor as a lock-out function (some kind of
 *          semaphore in the application).
 * unlock:  Points to a function in the application code that will be
 *          used by the monitor as an un-lock-out function (undo
 *          whatever lock-out mechanism was done by lock).
 */
void
monConnect(int (*mon)(), void (*lock)(), void (*unlock)())
{
    /* Assign incoming lock and unlock functions... */
    _monlock = lock;
    _monunlock = unlock;

    /* If the mon pointer is non-zero, then make the
     * mon_ connections...
     */
    if (mon) {

        _moncom = mon;

    /* Make the connections between "mon_" functions that are
     * symbolically accessible by the application and the
     * corresponding functions that exist in the monitor...
     */
        _moncom(GETMONFUNC_PUTCHAR,&_rputchar,0,0);
        _moncom(GETMONFUNC_GETCHAR,&_getchar,0,0);
        _moncom(GETMONFUNC_GOTACHAR,&_gotachar,0,0);
    }
}
```

监控器中的 moncom（）函数和应用程序中的 monConnect（）函数使用相同的 monlib.h 文件编译。这一头文件包括所有的 GETMONFUNC_XXX 定义。应用程序调用 monConnect，连接函数指针_rputchar、_getchar 及_gotachar 到相应监控器中的 rputchar、getchar 及 gotachar 函数。因此，应用程序可以只使用函数指针，但如能使用函数（没有指针）来访问监控器代码，则将会更好，且当监控器处于多任务处理环境时，它（包括所有 API）可成为同步访问控制的共享资源。如果应用程序经常通过函数访问监控器，则这一函数可以用信号量或其他可优先进入监控器代码空间的同步机制管理对监控器的访问。在应用程序空间中，这一代码可被用于调用监控器的 rputchar 服务。程序清单 11.4 中的代码可实现锁定。

程序清单 11.4　增加锁定到 putchar()

```
int
mon_putchar(uchar c)
{
    int ret;

    if (_monlock)
        _monlock();

    ret = _rputchar(c);

    if (_monunlock)
        _monunlock();

    return(ret);
}
```

监控器的同步功能是可选择的，使用 monConnect（）函数（见程序清单 11.3）的第 2，3 自变量来建立。

建立机制可归纳为：

- 启动时，应用程序调用 monConnect（）函数，传给它那个重要的地址和锁定解锁功能的指针（NULL 指针将有效地使这种同步特征无效）。
- monConnect（）函数通过那个地址调用监控器中的 moncom（）。在应用程序中，每次调用 moncom（）均通过监控器中相应函数的地址装入函数指针，在调用 moncom（）时使用 GETMONFUNC_XXX 宏。

- 当 monConnect（）完成时，应用程序可以在应用程序空间中直接调用整套的 mon_xxx 函数（如 mon_putchar（）,见程序清单 11.4）。

注意

monConnect（）建立了许多函数连接，远远多于程序清单 11.3 所给的 3 个函数。其他的连接建立方法与示例中的 3 个连接相同。

11.5　应用程序 start（）函数

一个简单应用程序的入口叫做 start。当通过 MicroMonitor 引入一个应用程序时，start 立即成为一个 C 函数。开始地址是应用程序装入后监控器调用的入口。典型的 start（）函数见程序清单 11 .5。

程序清单 11.5　典型的 start()函数

```
#include "monlib.h"

int
start(void)
{
    char    **argv;
    int     argc;

    /* Connect the application to the monitor.  This must be done
     * prior to the application making any other attempts to use the
     * "mon_" functions provided by the monitor.
     */
    monConnect((int(*)())(*(unsigned long *)0xfffc0000),(void *)0,(void *)0);

    /* Extract argc/argv from structure and call main(): */
    mon_getargv(&argc,&argv);
```

```
        /* Call main, then return to monitor. */
        return(main(argc,argv));
    }
```

注意，程序清单 11.5 中的应用程序不需要初始化自己的.bss。当 MicroMonitor 从闪存装入应用程序到 RAM 时，它可以访问应用程序的内存映像，并可以在将控制权转到入口点之前，自动清除所有应用程序的.bss 部分。

11.6 应用程序 main（）函数

应用程序 main（）函数（见程序清单 11.6）从 start（）中被调用。start（）函数通过 monConnect（）调用建立了与监控器的连接，并立即执行 mon_getargv（）的 API 功能来检索监控器中的参数列表。参数列表可通过监控器的 argv 命令建立，或在监控器命令行调用基于 TFS 的可执行程序时自动装入。因此，运行于监控器的应用程序中的 main（）函数与其他 main（）函数基本相同。

程序清单 11.6 main()

```
#include "monlib.h"

main(int argc,char *argv[])
{
    int     i;
    char    *env;

    for(i=0;i<argc;i++) {
        mon_printf("argv[%d] = %s\n",i,argv[i]);
    }

    mon_printf("Hello embedded world!\n");
    env = mon_getenv("ENV");
    if (env) {
        mon_printf(" The ENV variable is : %s\n",env);
    }
    mon_appexit(0);
```

```
}
```

程序清单 11.6 所调用的 3 个不同的 mon_ xxx 证明了当监控器存在时，应用程序可用到的附加 API 功能，一些关键特性如下：

- 从 TFS 装入的应用程序可以传给不同参数列表，以调用不同功能，这取决于需求；
- 应用程序通过 mon_printf（）立即访问监控器控制台接口；
- 应用程序可通过调用 mon_getenv（），使 API 访问监控器生成的外部变量；
- 应用程序通过回到调用程序 start（）或发出 mon_appexit（），使 API 被调用并回到对监控器的控制。

11.7　为应用程序创建的驱动程序

监控器有一套串口和以太网的驱动程序。第 9 章已讨论了 MicroMonitor 的以太网特性，还有应用程序可利用一些串口连接到控制台的功能（例如通过 mon_printf（）和 mon_putchar（））。因为所有 MicroMonitor 的驱动程序都是在查询状态下工作，所以高性能的应用程序需要控制监控器，并建立新的、中断驱动的驱动程序。建立新的驱动程序是非常可取的方法，因为监控器并没有核心级/用户级的界限。其驱动接口依赖于 RTOS，但其硬件的详细内容已超出了本书所应讨论的范围。应用程序不强制使用 MicroMonitor 提供的便利工具（详情参见图 3.1）。

11.8　基于应用程序的 CLI 使用监控器 CLI

假设有一个应用程序想通过自己的 CLI（命令行接口）连接到控制器端口，并安装了自己的串口驱动而且使用自己的接口连接到用户。除了仍使用一些监控器的命令外，应用程序还有些特殊的应用命令，监控器也有很多方便的工具用于软/硬件的分析及与文件系统的简单连接。应用程序的扩展命令可包含监控器的命令，因为 MicroMonitor 的 CLI 过程可以通过一个函数［docommand（）］被访问，应用程序也可以通过监控器的 API 调用 mon_docommand（）来访问 docommand（）。因此，应用程序的 CLI 可传送所有未知的命令串（未被特殊应用命令语法分析的命令串）到 mon_docommand（）进行监控器 CLI 的进一步过程。

在前面曾讲过，监控器忽略掉命令串中的起始下划线。为了能够更好地理解，下面给出应用程序在监控器顶层运行的上下文的一个例子。对于应用程序和监控

器，很有可能出现一个命令有多个名字的情况（help 命令是一个很好的例子）。如果看针对监控器的命令版本，用户会发现这个命令带有一个下划线。应用程序的命令处理器不处理这个标记，而是把它传给 mon-docommand（），再由这个函数去掉下划线。因此，help 被应用程序的 CLI 处理，-help 被传送到监控器的 CLI。

当应用程序的 CLI 变为监控器的 CLI 时，应用程序应关闭检查，监控器通常使用的串口被改变（因为应用程序已建立新的串行驱动）。若监控器的命令被应用程序的 CLI 调用，则将如何打印到控制台？我们可以通过对 putchar（）函数的小改动来解决这个问题（见程序清单 11.7）。

程序清单 11.7　putchar()

```
int
putchar(int C)
{
    extern int   (*remoteputchar)();
    int timeout;

    if (remoteputchar) {
        return(remoteputchar(C);
    }

    for(timeout=0;timeout<MAX_WAIT;timeout++) {
        if (XMIT_HOLD_EMPTY())
            break;
    }

    if (timeout == MAX_WAIT) {
        ERROR();
    }

    STORE_XMIT_HOLD_REG((char)C);
    return((int)C);
}
```

putchar（）包括指向 remoteputchar（）的函数指针。如果这个指针不是空指针，则不使用特殊监控器代码来连接串口，而是使用一些其他的远程功能。远程功能指

针通过类似于建立 mon_xxx 函数的机制给监控器赋值。惟一的区别是这个指针在监控器空间中而不在应用程序空间中。mon_com（）函数是被 monConnect（）使用的_moncom（）指针的封装函数。

```
#include "monlib.h"
mon_com(CHARFUNC_PUTCHAR,appPutchar,0,0);
```

这一调用告诉监控器使用的是函数 appPutchar（）而不是自己的接口。函数指针 remoteputchar（）（在监控器中）被装在 appPutchar（）的地址中（在应用程序中）。其他几个函数与此类似，逻辑上是相同的。

11.9　通过应用程序 CLI 运行脚本

MicroMonitor 还有一个功能值得注意。假设你现在处于应用程序的 CLI（不是监控器的 CLI），而且有一个要执行的脚本在 TFS 中，你可以给脚本赋予某个不与其他命令名冲突的名字，如 widget_script，在应用程序的 CLI 中键入 widget_script，通过上述机制，命令传到监控器，监控器的 CLI 将其传到 TFS，作为脚本执行。

如果脚本仅包含监控器的命令，则脚本执行的会很好。但你不能通过这种方法得到应用程序自由的脚本，因为缺省时，脚本运行器使用 docommand（）（在监控器中）来处理脚本的每一行，脚本中特殊应用程序的命令不能被识别。

然而，脚本运行器对这些情况早有准备，且经常使用函数指针来访问 CLI 函数。缺省时，这个函数指针随着监控器的 docommand（）装入，但这个指针可以被重新改写为指向应用程序空间的某个函数。

```
#include "tfs.h"
#include "monlib.h"
mon_tfsctrl(TFS_DOCOMMAND,(long)appDocommand,0);
```

上面的代码告诉监控器，脚本运行器使用函数 appDocommand（）而不是缺省的 docommand（）（tfsDocommand（）是被脚本运行器使用的函数指针），经过这一改变，应用程序可以得到自由的脚本。在应用程序的 CLI 中，你发出的命令 widget_script 被传到监控器，脚本运行器通过赋值给 tfsDocommand 函数指针的 appDocommand（）来执行每一个命令。应用程序的 CLI 为应用程序和监控器处理命令。因此，在监控器中，脚本运行器可被配置为从应用程序的 CLI 和监控器的 CLI 来执行命令。

11.10 小结

很明显，从本章讨论的内容中可以看出，应用程序可以做任何想做的事。你可以不使用任何监控器的功能，也可以选择只使用它的子集（如图 3.1 所示）。实际上可以在基础平台上建立不同种类的应用程序，但如果建立在监控器上会带来更多益处。监控器将应用程序与硬件细节隔离开。它通过生成一个方便的可执行上下文来简化应用程序。综上所述，它使得一些高级功能（如脚本）更易实现。

第 12 章　基于监控器的调试

前几章仅讨论了与调试启动代码有关的基本调试步骤，而本章将利用监控器的调试工具来调试位于微监控器顶层的应用程序。与远程的、基于主机的调试不同，基于监控器的调试仅需使用目标资源和目标处理器的内在能力。在很多情况下（并不是所有的情况），它需要可以写入应用程序的指令空间。例如，当用监控器的命令行接口（Command Line Interface，CLI）设置一个断点时，监控器调整了应用程序的代码空间或者设置了处理器的某些寄存器，且将这些控制转化到应用程序中。当在断点处的指令被执行时，控制将返回给监控器的 CLI。

赋予监控器显示内存情况的能力是相对简单的。根据正在调试的应用程序的复杂程度，它可以单步执行和在断点之后继续执行，可使监控器为调试提供必需的工具。然而，由于监控器不能读取源代码的符号信息（几乎所有的信息均与编译一起保存在主机上），因此基于监控器的调试仍存在很多问题。

例如：

- 如何显示内存情况？由于你运行的是目标的 CLI，因此不可能通过名字来寻找结构；相反，却可能需要在由连接器产生的主存分配处理程序中寻找代码的地址。由于微监控器并不知道变量的类型，也不知道符号的类型，因此并不知道要为 shorts、longs 或 char 等字符串分配多少字节。当然你也不知道结构体的显示。
- 如何确定一个断点的位置？同样地，符号信息的缺乏决定了一切。你不能只是简单地说“在函数 foo 处设置一个断点”，而必须首先寻找函数的地址。当知道了加载的地址之后，就可以认为某些命令，例如 b 0x123456 中的 0x123456 是想加入断点的位置。记住，你每次需在 foo（）设置断点的时候，都要执行这种查找操作，因为每次重新建立应用程序时，地址均可能不同，使简单的操作复杂化。
- 如何使源代码与汇编语言的执行相联系？现在的单步执行是汇编级的，不是 C 级的，而对高级程序员来说，汇编级的单步执行没有多大用处。
- 监控器怎样与串口通话？当断点发生时，监控器全部接管，但串口却由应用程序控制着。如果监控器为自己重新初始化串口，则应用程序与串口之间的接口可能丧失，这意味着将控制返回给应用程序将十分困难。
- 监控器如何暂时关闭应用程序？当断点发生时，系统全部由监控器接管。

如果应用程序是 RTOS 主机的任务，存在可行的中断和不同种类的外围设备，则监控器怎样才能全部关闭它们呢？

上面讲述了基于监控器调试的复杂性。但在决定放弃它之前，我们还是要重新讨论这个问题，看看是否存在着可取之处。

12.1 不同类型的调试方法

基于监控器的调试环境不能实现所有的功能。它并不能与 ICE（在线仿真器）相比。但是，如果奢望不是太高，你会发现基于监控器的调试确实提供了许多功能的。

注意

当要调试一个嵌入式系统时，要考虑些什么？你可能会想到基于 JTAG/BDM 的调试端口、仿真器、逻辑分析器、printf（）语句或一些复杂的源代码级调试器。它们各有优缺点。有些仿真器虽功能强大但价格昂贵，有些仅适用于某种 CPU 家族，有些仿真器只能用特殊编译器工具集。基于 JTAG 的调试端口可能是最常用的，虽性价比较好，但仍需 1 000 美元，而且只有当它接入系统时才有用，接入时一般都要花费相当大的气力。

我们将从建立监控器仿真模式开始。不考虑典型断点的限制，只考虑把断点作为一个特性的运行时间分析。不考虑简单的步进，并且不允许应用程序在断点把控制转向监控器的 CLI 之后继续。而是考虑自动返回断点从而运行时间分析。还将提供一些对符号表的访问（变量将不仅仅作为基本内存区被显示出来），还可以支持堆栈跟踪功能。主要的限制是监控器调试器在硬中断发生后不能返回控制到运行的应用程序（见 12.2）。如果可以接受，在基于监控器调试环境的限制下，你可以实现很多功能。

12.2 断点

断点可以告诉特殊地址或事件的应用程序，使控制转向其他权限（如监控器/调试器）。当应用程序放弃控制时，调试器可以使用所有上下文显示本地变量、给出堆栈踪迹，等等。下面将描述两种截然不同的断点。

- 硬断点——由监控器建立，因此当断点发生时，由监控器的 CLI 进行控制。

断点发生后，需重启应用程序，才能返回应用程序代码。

- 软断点——为了运行时间分析由监控器建立（以前是自动返回断点）。当断点指令执行时，控制将转向监控器代码（通过例外处理）。在存储信息后，监控器实时返回控制到应用程序。软中断有不同的类型。每种类型均可改变监控器维护的状态，且这一状态信息可被统计并反映执行行为，或控制监控器自身使用的数值（可能改变软断点处理的作用）。下面将解释实现软断点如何比硬断点困难。

通常，断点是通过触发其他一些异常的指令来代替指定地址中的指令而插入到应用程序中的[1]。这里我们把这种指令叫陷井。这一中断的异常处理程序应保存 CPU 完整的执行上下文（或寄存器设置），然后传输控制到监控器中指定调试器的进入点。若中断由硬断点引起，则监控器将提供它的 CLI。若中断由软断点引起，则监控器将记录一些状态并返回控制到应用程序。将控制返回到应用程序比较困难，这使得软断点比硬断点难。

在返回控制到应用程序之前，监控器必须在断点恢复代码（记住，插入中断来生成断点），且恢复所有在断点处激活的寄存器，并在断点地址恢复执行。

若监控器简单地用原始代码替换中断，则断点将被擦去。为了恢复执行并保存断点，监控器必须执行断点处的指令，并恢复足够长的控制来重新插入中断指令，然后允许应用程序继续执行。因为在断点处改变指令等于在自修复代码，所以监控器需要采取特殊工作来保持高速缓冲存储器的一致性。

断点的完整算法包括以下步骤：

（1）配置与陷井联系的异常处理句柄，使它指向监控器的代码。为了支持软断点，必须配置属于监控器的处理器跟踪（或单步）异常处理句柄。

（2）在指令的地址空间插入中断（要注意高速缓冲存储器）。

（3）传送控制到所调试的应用程序。

（4）在异常处理时，复制所有寄存器到监控器可访问的本地区域。

（5）判断异常的类型，采取相应的处理。若断点是硬断点，则转移到监控器的 CLI；否则，进行基于软断点的操作，继续第（6）步。

（6）把原始指令装回地址空间（要注意高速缓冲存储器）。

（7）恢复寄存器上下文。

（8）使处理器进入跟踪状态并从异常返回包含原始指令的地址。执行原始指令并产生跟踪例外。

---

1. 在 68 000，相应的指令是 trap。对于 x86，使用 INT3。大部分 CPU 有一些引起异常的指令，甚至仅是一个被零除所引起的溢出。

（9）当跟踪异常发生时，重新安装陷阱指令并控制再次返回应用程序（陷井必须重新安装，以便当下次 CPU 从那个地址再取指令时，断点能够被激活）。

可以通过仅执行你所需要的功能来避免复杂性。例如，软断点可限制为每次设置只中断一次。在断点到达后，监控器恢复原始指令（禁止断点）并继续全速执行。这一限制消除了第（8），（9）步的复杂性，同时也消除了再次实时使用断点的功能。为了进一步简化，将整个软断点机制省略，消除了第（5），（6），（7），（8），（9）各步。

由于监控器这部分的代码复杂并且极其依赖于特殊处理器机制，因此这里就不举例了。如果你对其细节感兴趣，则可以参看 CD 上的例子。

12.2.1 使用断点进行代码分析

我怀疑大部分的程序员把断点单纯地看成是跟踪程序执行的工具。下面将介绍的代码分析模式均把断点看做一个可触发一些活动的事件。代码分析的目的是提供一个开发器，且开发器带有一些用于集中运行时间信息的方便工具。信息会指出代码执行的路径，还会反映执行程序的其他信息。例如，多少堆栈空间被使用或是否特殊的数据结构被改变。代码分析要求使用上述的软断点机制。重要的结果不是遇见一些断点，而是信息被作为断点事件的结果被收集。

来自前线

这部分使读者掌握捕捉关键点的办法，并执行前边讨论的整个软断点机制。注意，在 CD 上提供的这部分调试器的代码只有当监控器带有相应的端口时才有意义。

为了进行这种方式的调试，监控器必须支持一些连接用户定义的方法和每个断点的特殊断点代码。在 MicroMonitor 中，at 命令产生这种联系。为了增加方便性，代码可以执行一些行为或基于某些情况选择执行一些行为。

监控器的命令语法如下：

```
at {breakpoint tag}[if condition]{action}
```

命令后的 3 个参数：

- breakpoint tag——特定断点的名字。标志的存在意味着 at 命令必须与一些其他建立断点的特定处理器一致（使用前述的一些方法）。这个标志是与特定处理器相关的，因为断点机制是与处理器相关的。

- if condition——可以作为逻辑的一部分，决定是否要执行行为的这样一个选项。条件可以是某个函数调用的非 0 返回值，也可以是 at 变量到达某些计数值，或 at 标志包括一些预定比特设置。
- action——由命令控制的操作。操作可以调整由命令维护的状态或仅返回控制到监控器。

下面的例子将说明怎样使用这一工具。

例 1

下面的 at 命令将建立一个基于异常的计数断点，异常将作为 DATA_1RD 断点的结果发生。由于断点机制依赖于 CPU，因此可使用 CPU 提供的第 1 个数据断点产生的异常。当异常发生时，at 句柄逻辑是异常处理句柄的一部分，对用户建立的每个 at 语句均有一个通过逻辑。第 1 个通过时将 ATV1 变量加工（在 at 命令上下文中），第 2 个通过时检查 ATV1 是不是 5。若 ATV1 是 5，则逻辑暂停应用程序并转向监控器 CLI 的完全控制。

```
at DATA_1RD ATV1++                  # Increment internal AT variable one
at DATA_1RD if ATV==5 BREAK         # If ATV1 equals 5, then break
```

例 2

根据 at 命令的规定，当断点 ADDR_2 执行后，断点 ADDR_1 执行时将中断（控制转移给监控器 CLI）。

```
at ADDR_1 FSET01                    # Set bit zero of the internal AT flag
at ADDR_1 FALL03 BREAK              # If both bits are set, then break
at ADDR_2 FCLR01                    # Clear bit zero of the internal AT flag
at ADDR_2 FSET02                    # Set bit 1 of the internal AT flag
```

例 3

根据例子的示范代码分析辅助应用程序中函数的用法。如果函数在地址 0x1234 返回值为 1，则中断发生。

```
at ADDR_1 0x1234()==1 BREAK
```

例 4

在最后一个例子中，我们将用 at 命令来帮助检测存储器的遗漏。假定 ADDR_1 是 malloc，ADDR_2 是 free，此时 ADDR_3 被找到，并且没有被存储器所指定，通过观察断点后变量 ATV1 的内容能核对出 malloc/free 之间的细微差别。

```
at ADDR_1 ATV1++          # Increment ATV1 at ADDR_1 (malloc)
```

```
at ADDR_2 ATV1—        # Decrement ATV1 at ADDR_2 (free)
at ADDR_3 BREAK        # Break at ADDR_3
```

上述各例主要是展示 at 命令的功能和灵活性。CPU 指定的代码与将实现的基础断点相似，如加入 at [if-condition] {action}拓展，则可增加基于监控器断点的新功能。当代码的开销很小时，at 行为可以造成实时的损失，因每次断点发生时在运行数据流中均可以加入额外代码。这种开销可否接受由应用程序的要求决定，因为所有的处理都在板上，所以性能损失通常可以接受。

12.2.2 一些 CPU 提供了调试吊钩

以前，基于监控器调试的用户会被指令空间一般不可修改的情况所限制。对于大多数的嵌入系统设计来说，指令空间是不能修改的，尤其当指令存储于闪存（flash）[1]或者 EPROM 中，且 CPU 从这个空间里直接执行代码时，更是如此。

在现代的处理器设计过程中，很多处理器配备了特殊的调试手段来消除这个限制（在 MicroMonitor 中，应用程序传送到 RAM，可执行的应用程序的指令空间是可写的）。

目前，由于 CPU 可以在每一个指令地址设置断点，而并不要求指令空间是可写的，因此调试寄存器增加了基于监控器调试的通用性。另外，由于一些处理器只是数据断点（有时候也被称做 watchpoints），而数据断点可以在指定的数据或数据段被访问的任何时间触发中断，因此可以通过数据断点来确定中断只发生在读数据或写数据时还是读写时都发生。

监控器必须能够处理这些增加的调试手段（手段会因 CPU 不同而不同），且应该能够开发额外的功能，而不能只是为了在指令空间中加入一些陷阱来实现通用的结构。

数据断点给监控器代码增加了更多的复杂性，因此更难被支持。因为监控器不能用异常发生时的地址来判断触发异常的断点，所以，如果利用数据断点，则其异常处理必须在数据断点发生时能够查找 CPU 的状态，并且当设置了多个断点的时候，能够确定其中的哪个断点引发了中断。

12.3 增加符号能力

虽然在此不会谈到太多的编程细节，但是在程序中将直接给出监控器的增加符

1. 虽然闪存可以被写，但并不说明我们可以在断点处理程序中可以那样做。

号能力。CLI 已经有了对程序变量进行必要的识别和求值的渠道，扩展此功能，使其能够很容易地查到特定文件中的其他未定义符号。

例如，如果 flash 存储器中有一个 symtbl 的文件，则文件中有如下的代码：

```
main        0x123456
func1       0x123600
func2       0x123808
varA        0x128000
varB        0x128004
varC        0x12800c
```

在其中执行下面的代码：

```
echo The address of main() is %main
echo The variable 'varA' is located at %varA
```

输出：

main（）地址是 0x123456

变量 varA 被设置在 0x128000

MicroMonitor 也允许改变前面的 at 命令，将

```
at ADDR_1 0x1234（）==1    BREAK
```

改为

```
at ADDR_1 %{func_name}（）==1 BREAK
```

因为可以用函数名来代替 0x1234，所以需要做的就是通过工具设置（编译器/连接器）从应用程序中生成 symtbl 文件。

12.4　显示存储器

几乎所有的带有嵌入系统的引导监控器都提供了存储器的显示命令。即使对于高级软件开发人员，这些命令也是非常有用的。如果这些命令支持十六进制和十进制的地址空间显示，并提供 1B、2B 及 4B 的数据包，则 CLI 处理符号的基本功能允许显示器以基本形式显示变量。例如，假设 short 类型的变量 varB 用十进制来显示，则可以用下面的命令：

```
dm  –2d  %varB 1
```

其中，dm 用于显示存储器的命令；–2d 是可选的字段，代表这个数据以 2B 的十进制格式显示；%varB 是要显示的变量名字；1 代表只有一个单位被显示。其结果就是你可以像用高级的调试器那样来显示变量，你要做的就是确保你的 symtbl 文件与你调试的程序同步。

你可以多试几次，并生成一些简单的程序。例如，下面的程序即可显示不同的整数类型。

文件 int2：

```
dm –2d $APG1 1     #Decimal 16-bit integer
```

文件 uint4：

```
dm –4 $APG1 1     #Hex 32-bit integer (no –d option, means display in hexadecimal)
```

现在，无需键入 dm –2d %varB 1,int2 可以用做

```
int2 %varB
```

-s 的选项也可以加入 dm 中，这样，存储器可以以字符串的形式显示，而不是以十六进制的格式显示。程序清单 12.1 是完成显示存储器命令 dm 代码的开始。

程序清单 12.1　显示存储器命令的代码

```
char *DmHelp[] = {
    "Display Memory",
    " –[24bdefs] {addr} [cnt]",
    " –2     short access",
    " –4     long access",
    " –b     binary",
    " –d     decimal",
    " –e     endian swap",
    " –f     fifo mode",
    " –m     use 'more'",
    " –s     string",
    " –v {var} quietly load 'var' with element at addr",
    0,
};
```

```
#define BD_NULL             0
#define BD_RAWBINARY        1
#define BD_ASCIISTRING      2

int
Dm(int argc,char *argv[])
{
        int         i, count, count_rqst, width, opt, more, size, fifo;
        int         hex_display, bin_display, endian_swap;
        char        *varname, *prfmt, *vprfmt;
        uchar       *cp, cbuf[16];
        ushort      *sp;
        ulong       *lp, add;

        width = 1;
        more = fifo = 0;
        bin_display = BD_NULL;
        hex_display = 1;
        endian_swap = 0;
        varname = (char *)0;
        while((opt=getopt(argc,argv,"24bdefmsv:")) != -1) {
                switch(opt) {
                case '2':
                        width = 2;
                        break;
                case '4':
                        width = 4;
                        break;
                case 'b':
                        bin_display = BD_RAWBINARY;
                        break;
                case 'd':
                        hex_display = 0;
```

```
            break;
        case 'e':
            endian_swap = 1;
            break;
        case 'f ':
            fifo = 1;
            break;
        case 'm':
            more = 1;
            break;
        case 'v':
            varname = optarg;
            break;
        case 's':
            bin_display = BD_ASCIISTRING;
            break;
        default:
            return(CMD_PARAM_ERROR);
        }
    }

    add = strtoul(argv[optind],(char **)0,0);

    if (argc- (optind-1) == 3) {
        count_rqst = strtoul(argv[optind+1],(char **)0,0);
        count_rqst *= width;
    }
    else
        count_rqst = 128;
```

程序清单 12.1 详细地介绍了可选参量的处理。最初的几行代码是默认的初始化。调用 getopt（）是用户可以继承默认选项。选项包括对访问宽度的规定（默认为 1B，可设为 short 或 long），提供了从 FIFO 中以不同格式转存数据的能力，如用原始的二进制、高位和低位转换及 ASCII 或十进制格式表示数值。最后几行是从命

令行中查找参数并建立内存转存的地址和大小。

程序清单 12.2 列出了格式化结果的代码，hex_display 和 width 变量的内容用来确定后面的 printf 的字符串格式，并列出了两种字符串格式：一种是输出到控制台的数据(prfmt)；另一种是用于格式化命令行并解释程序中的变量，这个变量可能由 –v 的选项限定(vprfmt)。

程序清单 12.2　格式化结果

```
if (hex_display) {
    switch(width) {
    case 1:
        prfmt = "%02X ";
        break;
    case 2:
        prfmt = "%04X ";
        break;
    case 4:
        prfmt = "%08X ";
        break;
    }
    vprfmt = "0x%x";
}
else {
    switch(width) {
    case 1:
        prfmt = "%3d ";
        break;
    case 2:
        prfmt = "%5d ";
        break;
    case 4:
        prfmt = "%12d ";
        break;
    }
    vprfmt = "%d";
```

```
}
```

程序清单 12.3 二进制和 ASCII 码格式

```
if (bin_display != BD_NULL) {
    cp = (uchar *)add;
    if (bin_display == BD_ASCIISTRING) {
        puts(cp);
        if (varname) {
            shell_sprintf(varname,vprfmt,cp+strlen(cp)+1);
        }
    }
    else {
        for(i=0;i<count_rqst;i++) {
            putchar(*cp++);
        }
    }
    putchar('\n');
    return(CMD_SUCCESS);
}
```

如果以二进制或者 ASCII 码的格式显示的选项被选中的话，则程序清单 12.3 的代码将被执行，并完成命令。注意，两种显示模式的访问宽度都是 1B。

最后的循环（见程序清单 12.4）是以 1B，2B，4B 转存内存的代码。这里只显示了 width= =2 时的代码，其他的可做类似处理，每 16B 另起一行。如果–v 的选项被选中，将不显示。指定地址的内容将会被放到命令行程序的变量中，命令结束。循环本身被用来支持–m 选项，如果这个选项被选中，将提示用户转存更多的数据。

程序清单 12.4 控制数据宽度

```
do {
    count = count_rqst;

    if (width == 1) {
        ...
    }
```

```
else if (width == 2) {
        sp = (ushort *)add;

        if (varname) {
                shell_sprintf(varname,vprfmt,
                        endian_swap ? swap2(*sp) : *sp);
        }
        else {
                while(count>0) {
                        printf("%08lx: ",(ulong)sp);
                        if (count > 16)
                                size = 16;
                        else
                                size = count;

                        for(i=0;i<size;i+=2) {
                                printf(prfmt,
                                        endian_swap ? swap2(*sp) : *sp);
                                if (!fifo)
                                        sp++;
                        }
                        putchar('\n');
                        count -= size;
                        if (!fifo) {
                                add += size;
                                sp = (ushort *)add;
                        }
                }
        }
        else if (width == 4) {
                ...
        }
} while (more && More());
```

```
    return(CMD_SUCCESS);
}
```

12.5 将 C 结构覆盖到内存

如果可以将内存显示成结构和链接的程序列表，那将是非常好的。但使监控器做到上述要求将很难，因监控器一般不访问编译器和链接器提供的结构的格式化信息。假设我有文件系统，人们可能认为我将工具集生成数据放入一个文件中并允许监控器解析数据。但是解析这个数据可能是复杂的，尤其是当考虑文件中的格式时可能会因为编译器的关系而更加复杂。即使是限制在特定的文件格式，符号表的格式也会因编译器的不同而不同。简单的解决办法是在监控器中生成一条能够在文件系统中查找结构定义的文件命令，来决定如何在目标内存块的顶层覆盖结构化的显示。此办法消除了对外部文件格式的所有依赖，使其在不同的 CPU 类型和设置的情况下都能工作。

结构定义文件是包含与 C 头文件中类似的结构定义的 ASCII 码文件。监控器中的命令 cast 可以以这个文件作为参考来显示内存中的特定部分，利用结构信息和用 symtbl 文件的监控器，可以发布类似 cast　abc　%InBuf 的命令。

命令可在 falsh 文件系统中查找 structfile 的文件，如果能找到，则覆盖掉与符号%InBuf（来自监控器的 symtbl 文件）相关联的地址顶层中定义的 abc 结构。通过加强这个命令，你能够确定下一个指针指向的结构中的成员，可以在几乎不增加代码的情况下将 cast 命令转成链接的列表显示工具。

MicroMonitor 希望在 structfile 文件中进行结构定义。这个文件的格式一般与标准 C 结构定义相类似，但也有一些限制，支持 char、short 及 long 类型，并以 1B、2B 及 4B 十进制整数显示。十六进制和字符显示格式由类似的符号确定：char.x、short.x 及 long.x 显示十六进制，char.c 类型显示字符的值。

程序清单 12.5 中的结构定义是以十进制的格式显示成员 i，以十六进制的格式显示 j，以 1B 十进制整数的格式显示 e。

程序清单 12.5　结构定义

```
struct abc {
    long     i;
    long.x j;
    char.c c;
```

```
    char.x d;
    char    e;
}
```

如果结构包含数组，则用户必须将数组定义为上述列出的基本类型，并给它一个合适的大小。cast 命令不显示结构中的数组，因为输出太复杂，与数组成员关联的大小被当成填充，只显示数组的名字和大小。程序清单 12.6 是结构定义文件，解释了 cast 命令的功能。注意，# 标号代表注释。

程序清单 12.6　结构定义文件在 cast 命令中的应用

```
struct abc {
    long          I;
    char.c        c1;
    pad[3];              # Not displayed, just adds padding
    struct def d;
}


struct def {
    short     s1;
    short.x s2;
    long      ltbl[5];   # Data is not displayed
    short     s3;
}
```

注意，程序清单 12.6 嵌入的结构、.x 前缀的使用及 pad[]描述符。pad[]描述符用于 CPU/编译器特定的填充。cast 命令不清楚编译器特定的填充和 CPU 特定的调整要求。如果结构定义把 long 型数据放入奇地址空间上而 CPU 却不支持，cast 仍然试图访问，这就将产生异常，此时用户必须加入合适的填充以反映实际的调整要求。如果成员是 char.c * 或 char.c[]类型，则 cast 将显示 ASCII 字符串（如果你不想让指针被重定义，就用 char.x*.）。

程序清单 12.7　cast 命令

```
static   ulong memAddr;
static   int castDepth;


#define OPEN_BRACE   '{'
```

```
#define CLOSE_BRACE '}'

#define STRUCT_SEARCH       1
#define STRUCT_DISPLAY      2
#define STRUCT_ALLDONE      3
#define STRUCT_ERROR        4

#define STRUCT_SHOWPAD   (1<<0)
#define STRUCT_SHOWADD   (1<<1)
#define STRUCT_VERBOSE   (1<<2)


#define STRUCTFILE "structfile"

struct mbrinfo {
    char *type;
    char *format;
    int size;
    int mode;
};

struct mbrinfo mbrinfotbl[] = {
    { "char",       "%d",       1 },    /* decimal */
    { "char.x",     "0x%02x",   1 },    /* hex */
    { "char.c",     "%c",       1 },    /* character */
    { "short",      "%d",       2 },    /* decimal */
    { "short.x",    "0x%04x",   2 },    /* hex */
    { "long",       "%ld",      4 },    /* decimal */
    { "long.x",     "0x%08lx",  4 },    /* hex */
    { 0,0,0 }
};

char *CastHelp[] = {
    "Cast a structure definition across data in memory.",
```

```
        "-[apv| {struct type} {address}",
        "Options:",
        " -a      show addresses",
        " -l{linkname}",
        " -n{structname}",
        " -p      show padding",
        " -t{tablename}",
        0,
};

int
Cast(int argc,char *argv[])
{
        long        flags;
        int         opt, tfd, index;
        char        *structtype, *structfile, *tablename, *linkname, *name;

flags = 0;
name = (char *)0;
linkname = (char *)0;
tablename = (char *)0;
while((opt=getopt(argc,argv,"apl:n:t:")) != -1) {
        switch(opt) {
        case 'a':
                flags |= STRUCT_SHOWADD;
                break;
        case 'l':
                linkname = optarg;
                break;
        case 'n':
                name = optarg;
                break;
        case 'p':
                flags |= STRUCT_SHOWPAD;
```

```
            break;
        case 't':
            tablename = optarg;
            break;
        default:
            return(0);
        }
    }
    if (argc != optind + 2)
        return(-1);

    structtype = argv[optind];
    memAddr = strtoul(argv[optind+1],0,0);
```

cast 命令用于程序清单 12.7，支持一些提供链接列表显示的选项，包括显示的数据地址和填充的显示。和其他的 MicroMonitor 命令一样，cast 函数首先进行默认的初始化，调用 getopt（）替换默认，找回必要的命令行参数。

程序清单 12.8　校验结构

```
/* Start by detecting the presence of a structure definition file... */
    structfile = getenv("STRUCTFILE");
    if (!structfile) {
        structfile = STRUCTFILE;
    }

    tfd = tfsopen(structfile,TFS_RDONLY,0);
    if (tfd < 0) {
        printf("Structure definition file '%s' not found\n",structfile);
        return(0);
    }
```

由于命令需要结构定义文件，因此必须验证结构定义文件是否是当前的文件（见程序清单 12.8）。文件的默认名由 STRUCTFILE 决定，如果是当前的文件，则可以由环境变量 STRUCTFILE 替换。这个技术可用于几个地方，例如可用于代码中的值，也可用于由命令解释程序变量的内容替换时。

程序清单 12.9　显示若干结构

```
index = 0;
do {
    castDepth = 0;
    showStruct(tfd,flags,structtype,name,linkname);
    index++;
    if (linkname) {
        printf("Link #%d = 0x%lx\n",index,memAddr);
    }
    if (tablename || linkname) {
        if (askuser("next?"))    {
            if (tablename) {
                printf("%s[%d]:\n",tablename,index);
            }
        }
        else {
            tablename = linkname = (char *)0;
        }
    }
} while(tablename || linkname);

tfsclose(tfd,0);
return(0);
}
```

cast 命令中最后的循环（见程序清单 12.9）允许调用 showStruct（）显示多个结构或表入口。显示每一个结构后，用户可以停止或继续下一个入口。

程序清单 12.10　showStruct()

```
/* castIndent():
 *    Used to insert initial whitespace based on the depth of the
 *    structure nesting.
 */
void
castIndent(void)
```

```
{
    int i;

    for(i=0;i<castDepth;i++) {
        printf("    ");
    }
}

/* strAddr():
 *  Called by showStruct().  It will populate the incoming buffer pointer
 *  with either NULL or the ascii-hex representation of the current address
 *  pointer.
 */
char *
strAddr(long flags, char *buf)
{
    if (flags & STRUCT_SHOWADD) {
        sprintf(buf,"0x%08lx: ",memAddr);
    }
    else {
        buf[0] = 0;
    }
    return(buf);
}

int
showStruct(int tfd,long flags,char *structtype,char *structname,char *linkname)
{
    struct mbrinfo *mptr;
    ulong   curpos, nextlink;
    int     i, state, snl, retval, tblsize;
    char    line[96], addrstr[16], format[64];
    char    *cp, *eol, *type, *eotype, *name, *bracket, *eoname, tmp;
```

```
type = (char *)0;
retval = nextlink = 0;
curpos = tfsctrl(TFS_TELL,tfd,0);
tfsseek(tfd,0,TFS_BEGIN);
castIndent();

if (structname) {
    printf("struct %s %s:\n",structtype,structname);
}
else {
    printf("struct %s @0x%lx:\n",structtype,memAddr);
}
castDepth++;
```

showStruct（）函数（见程序清单 12.10~程序清单 12.15）是 cast 命令的主力，利用递归显示结构中的结构。它解析结构定义文件（tfd 是描述符），并查找 structtype 中定义的结构类型，从 memAddr 开始的内存区域以结构的形式显示。注意，showStruct（）不检验 structfile 中结构定义的语法，由用户决定避免混淆这个函数。

showStruct（）函数记录了 structfile 的当前位置，在递归时找回后，开始寻找文件，并调用 castIndent（）。castIndent（）函数知道 showStruct（）的深度，基于这个深度输出相关空间的数目。当递归深度增加时，空间数增加。这个过程可使嵌入的成员如预期的那样缩进后，showStruct（）输出结构名或地址，并开始查找结构定义文件中的下一个结构类型。

程序清单 12.11　通过结构定义文件查找

```
state = STRUCT_SEARCH;
snl = strlen(structtype);

while(1) {
    if (tfsgetline(tfd,line,sizeof(line) -1) == 0) {
        printf("Structure definition '%s' not found\n",structtype);
        break;
    }
    if ((line[0] == '\r') || (line[0] == '\n')) { /* empty line? */
        continue;
```

```
}

eol = strpbrk(line,";#\r\n");
if (eol) {
    *eol = 0;
}

if (state == STRUCT_SEARCH) {
    if (!strncmp(line,"struct",6)) {
        cp = line+6;
        while(isspace(*cp)) {
            cp++;
        }
        if (!strncmp(cp,structtype,snl)) {
            cp += snl;
            while(isspace(*cp)) {
                cp++;
            }
            if (*cp == OPEN_BRACE) {
                state = STRUCT_DISPLAY;
            }
            else {
                retval = -1;
                break;
            }
        }
    }
}
```

最初，showStruct（）在(STRUCT_SEARCH)的状态下寻找特定的结构，历经文件的每一行，直到发现结构定义（见程序清单 12.11）后，showStruct（）改变到状态 STRUCT_DISPLAY，表明它已指定结构显示指定的内存。

程序清单 12.12　显示结构

```
else if (state == STRUCT_DISPLAY) {
```

```
type = line;
while(isspace(*type)) {
        type++;
}

if (*type == CLOSE_BRACE) {
        state = STRUCT_ALLDONE;
        break;
}

eotype = type;
while(!isspace(*eotype)) {
        eotype++;
}
*eotype = 0;
name = eotype+1;
while(isspace(*name)) {
        name++;
}
bracket = strchr(name,'[');
if (bracket) {
        tblsize = atoi(bracket+1);
}
else {
        tblsize = 1;
}

if (*name == '*') {
        castIndent();
        printf("%s%-8s %s: ",strAddr(flags,addrstr),type,name);
        if (!strcmp(type,"char.c")) {
                printf("\"%s\"\n",*(char **)memAddr);
        }
        else {
```

```
            printf("0x%lx\n",*(ulong *)memAddr);
        }
        memAddr += 4;
        continue;
    }
```

showStruct（）在 STRUCT_DISPLAY（见程序清单 12.12）状态后，每行均被解析，用来滤除空格并在行中使字符指针正确地对齐令牌。注意，这个机制并不健全。为了保持合理的简洁性，代码假设结构定义文件健全。

解析的代码首先寻找需要重被指定的指针的星号（*）。如果没有，则原始内存是在结构的每一个成员的基础上格式化的。Mbrinfotbl[]表（见程序清单 12.13）用来浏览结构定义文件中的行，查找所支持的数据显示格式。程序清单 12.12 监测到[]的存在，支持数组说明。

程序清单 12.13　用 mbrinfotbl[ ]

```
mptr = mbrinfotbl;
while(mptr->type) {
    if (!strcmp(type,mptr->type)) {
        castIndent();
        eoname = name;
        while(!isspace(*eoname)) {
            eoname++;
        }
        tmp = *eoname;
        *eoname = 0;

        if (bracket) {
            if (!strcmp(type,"char.c")) {
                printf("%s%-8s %s: ",
                    strAddr(flags,addrstr),mptr->type,name);
                cp = (char *)memAddr;
                for(i=0;i<tblsize && isprint(*cp);i++) {
                    printf("%c",*cp++);
                }
                printf("\n");
```

```
        }
        else {
            printf("%s%-8s %s\n",
                strAddr(flags,addrstr),mptr->type,name);
        }
        memAddr += mptr->size * tblsize;
    }
    else {
        sprintf(format,"%s%-8s %%s: %s\n",
            strAddr(flags,addrstr),mptr->type,mptr->format);
        switch(mptr->size) {
            case 1:
                printf(format,name,*(uchar *)memAddr);
                break;
            case 2:
                printf(format,name,*(ushort *)memAddr);
                break;
            case 4:
                printf(format,name,*(ulong *)memAddr);
                break;
        }
        memAddr += mptr->size;
    }
    *eoname = tmp;
    break;
}
mptr++;
}
```

程序清单 12.14　嵌入式结构的处理

```
if (!(mptr->type)) {
    int padsize;
    char *subtype, *subname, *eossn;
```

```
if (!strcmp(type,"struct")) {
    subtype = eotype+1;
    while(isspace(*subtype)) {
        subtype++;
    }
    subname = subtype;
    while(!isspace(*subname)) {
        subname++;
    }
    *subname = 0;

    subname++;
    while(isspace(*subname)) {
        subname++;
    }
    eossn = subname;
    while(!isspace(*eossn)) {
        eossn++;
    }
    *eossn = 0;
    if (*subname == '*') {
        castIndent();
        printf("%s %s %s %s: 0x%08lx\n",
            strAddr(flags,addrstr),
            type,subtype,subname,*(ulong *)memAddr);
        if (linkname) {
            if (!strcmp(linkname,subname+1))
                nextlink = *(ulong *)memAddr;
        }
        memAddr += 4;
    }
    else {
        for (i=0;i<tblsize;i++) {
            if (bracket) {
```

```
                    sprintf(bracket+1,"%d]",i);
                }
                if (showStruct(tfd,flags,subtype,subname,0) < 0) {
                    state = STRUCT_ALLDONE;
                    goto done;
                }
            }
        }
    }
    else if (!strncmp(type,"pad[",4)) {
        padsize = atoi(type+4);
        if (flags & STRUCT_SHOWPAD) {
            castIndent();
            printf("%spad[%d]\n",strAddr(flags,addrstr),
                padsize);
        }
        memAddr += padsize;
    }
    else  {
        retval = --1;
        break;
    }
  }
  else {
    state = STRUCT_ERROR;
    break;
  }
}
```

如果 showStruct（）不能找到对应的行中内容和 mbrinfotbl[]数组，则惟一的可能就是有其他的结构（如出现递归情况或程序的深入）或者不可见的填充请求（见程序清单 12.14）。结构定义文件中的 pad 入口可通知 showStruct（）忽略特定数目字节且不显示任何东西。这个选项很有用，因为：

（1）有时你必须插入填充才能正确地调整结构中的成员，如 C 编译器。

（2）有时你只对结构中的小部分感兴趣，从而减少了不必要的视觉干扰。Pad[] 具有减少不必要的视觉干扰的特征。

当循环结束时（见程序清单 12.15），如果一切正常，则 state 的值是 STRUCT_ALLDONE。否则，switch 语句输出一个错误消息。

程序清单 12.15 showStruct()

```
    done:
        switch(state) {
            case STRUCT_SEARCH:
                printf("struct %s not found\n",structtype);
                retval = -1;
                break;
            case STRUCT_DISPLAY:
                printf("invalid member type: %s\n",type);
                retval = -1;
                break;
            case STRUCT_ERROR:
                printf("unknown error\n");
                retval = -1;
                break;
        }
        tfsseek(tfd,curpos,TFS_BEGIN);
        if (linkname)
            memAddr = nextlink;
        castDepth--;
        return(retval);
    }
```

12.5.1 一些示例输出

用一些显示的例子来示范 cast 和 dm 命令的灵活性，先用 dm 显示内存转存，再用 cast 显示同样的内存。

假设想看到位于系统内存 0x5d000 处的结构类型 abc，如果用 dm 以原始内存

转存数据，则将显示如下：

```
uMON> dm 0x5d000 48
0005d000: 00 01 00 00 64 00 00 00     04 01 12 34 00 00 00 00     ....d.....4....
0005d010: 00 00 00 00 00 00 00 00     00 00 00 00 00 00 00 00     ................
0005d020: be ef 00 00 00 00 00 00     00 00 00 00 00 00 00 00     ................
```

只要你想读原始内存，则输出的格式是整齐且易读的，每一行均以十六进制的地址开头，然后是 16B 的 ASCII 码十六进制数据。如果 ASCII 是不可输出的字符，则会输出一个点站位。如果你进一步地看，那么能看到内存区域中的数据。更好的办法应为如下输出：

```
uMON>cast abc 0x5d000

struct abc @0x5d000:
    long        I: 65536
    char.c      c1: d
    struct def d:
        short       s1: 1025
        short.x     s2: 0x1234
        long        ltbl[5]
        short       s2: 48879
```

注意，I 被写为长整型十进制值（0x10000= =65536）

0005d000：00 01 00 00 64 00 00 00　　04 01 12 34 00 00 00 00　　....d.....4....

注意，被 c1 写为 ASCII 码字符（0x64= =d）

0005d000：00 01 00 00 64 00 00 00　　04 01 12 34 00 00 00 00　　....d.....4....

注意，三个填充字节被忽视

0005d000：00 01 00 00 64 00 00 00　　04 01 12 34 00 00 00 00　　....d.....4....

注意，s1 被显示为十进制数（0x401= =1025）

0005d000：00 01 00 00 64 00 00 00　　04 01 12 34 00 00 00 00　　....d.....4....

并且继续这个结构。正如你所看见，用 cast 观察一个结构比通过原始内存转

移更容易。

12.6 堆栈跟踪

堆栈跟踪异常处理可能是固件开发人员最有用的工具。堆栈跟踪式开发者能够在程序断点看到功能嵌套。由于堆栈跟踪可以即时看到调用的过程，由此可以节省大量的调试和分析的时间。

跟踪堆栈的能力通常只能由高级的调试环境来提供，但在这里却不是这样。完成对几种不同 CPU 的堆栈跟踪，应该说是很难的，但只要你完成之后，你就离不开它了。

本节将讨论的基于监控器的堆栈跟踪可从符号表文件中提取符号信息。虽然堆栈跟踪需要将符号按地址的升序排列，但还是需要将基于监控器的堆栈跟踪限制在其所提供功能的嵌套上。

因为在堆栈中解释变量有点复杂。想像一下，返回的地址必须很容易查找，因为硬件在每次从函数调用中返回的时候必须利用返回的地址。另一方面，在堆栈中不是硬件查找变量，因硬件是由编译器构造的虚拟机的一部分。事实上，在堆栈中，同一个 CPU、不同的编译器可以用不同的转换放置变量。因此，在特定的函数堆栈框架中，显示变量的能力对于 CPU 和监控器来说都不是必然的。任何显示变量的堆栈跟踪都需要编译器有关变量在哪里和如何在框架中存储的信息。

多数执行堆栈跟踪的代码是编译器和 CPU 的细节。一些分析符号文件是普遍的且能在若干执行中再次使用的。正如前面所提到的，堆栈跟踪虽然非常复杂，但却具有很高的应用价值。程序清单 12.16 给出了 MicroMonitor(或 PowerPC)中 strace 命令的代码。

程序清单 12.16　strace 命令

```
int
Strace(int argc,char *argv{])
{
    char      *symfile, fname[64];
    TFILE     *tfp;
    ulong     *framepointer, pc, fp, offset;
    int       tfd, opt, maxdepth;

    tfd = fp = 0;
```

```
maxdepth = 20;
pc = ExceptionAddr;
while ((opt=getopt(argc,argv,"d:F:P:rs:")) != -1) {
        switch(opt) {
        case 'd':
                maxdepth = atoi(optarg);
                break;
        case 'F':
                fp = strtoul(optarg,0,0);
                break;
        case 'P':
                pc = strtoul(optarg,0,0);
                break;
        case 'r':
                showregs();
                break;
                default:
                return(0);
        }
}

if (!fp) {
        getreg("R1", &framepointer);
}
else {
        framepointer = (ulong *)fp;
}

/* Start by detecting the presence of a symbol table file... */
symfile = getenv("SYMFILE");
if (!symfile) {
        symfile = SYMFILE;
}

tfp = tfsstat(symfile);
```

```
if (tfp)    {
        tfd = tfsopen(symfile,TFS_RDONLY,0);
        if (tfd < 0) {
                tfp = (TFILE *)0;
        }
}

/* Show current position: */
printf("     0x%08lx",pc);
if (tfp) {
        AddrToSym(tfd,pc,fname,&offset);
        printf(": %s()",fname);
        if (offset) {
                printf(" + 0x%lx",offset);
        }
}
putchar('\n');

/* Now step through the stack frame... */
while(maxdepth) {
        framepointer = (ulong *)*framepointer;

        if ((!framepointer) || (!*framepointer) || (!*(framepointer+1))) {
                break;
        }

        printf("     0x%08lx",*(framepointer+1));
        if (tfp) {
                int match;

                match = AddrToSym(tfd,*(framepointer+1),fname,&offset);
                printf(": %s()",fname);
                if (offset) {
                        printf(" + 0x%lx",offset);
                }
```

```
                if (!match) {
                    putchar('\n');
                    break;
                }
            }
            putchar('\n');
            maxdepth--;
        }

        if (!maxdepth) {
            printf("Max depth termination\n");
        }

        if (tfp) {
            tfsclose(tfd,0);
        }
        return(0);
    }
```

像大多数的 MicroMonitor 命令一样，strace（见程序清单 12.16）先进行默认的初始化及处理命令行选项和参数。由于 strace 总是在异常发生之后运行，因此当前地址从全局变量 ExceptionAddr（由异常处理者写入）中获得。命令接收几种选项，但通常均默认为适合。

选项被处理之后，代码找回堆栈指针。堆栈基本上以堆栈段的连接列表形式组织起来。然后，代码用这个指针找到开始段。

在处理堆栈之前，代码先寻找 symtbl 文件，并初始化 tfd 描述符后，在每次监控器查找到一个相对应的地址时将描述符发送到 AddrToSym（）。如果 symtbl 文件不存在，则只显示地址。处理循环历经当前嵌套的堆栈段，直到符合中止条件、空段指针或者段深的最大值。

程序清单 12.17　AddrToSym()

```
/* AddrToSym():
 *  Assumes each line of symfile is formatted as...
 *          synmame SP hex_address
 *  and that the symbols are sorted from lowest to highest address.
```

```
 *  Using the file specified by the incoming TFS file descriptor,
 *  determine what symbol's address range covers the incoming address.
 *  If found, store the name of the symbol as well as the offset between
 *  the address of the symbol and the incoming address.
 *
 *  Return 1 if a match is found, else 0.
 */
int
AddrToSym(int tfd,ulong addr,char *name,ulong *offset)
{
    int       lno;
    char      *space;
    ulong     thisaddr, lastaddr;
    char      thisline[84];
    char      lastline[sizeof(thisline)];

    lno = 1;
    *offset = 0;
    lastaddr = 0;
    tfsseek(tfd,0,TFS_BEGIN);

    while(tfsgetline(tfd,thisline,sizeof(thisline) -1)) {
        space = strpbrk(thisline,"\t ");
        if (!space) {
            continue;
        }
        *space++ = 0;
        while(isspace(*space)) {
            space++;
        }
        thisaddr = strtoul(space,0,0);   /* Compute address from          */
                                         /* entry in symfile.             */

        if (thisaddr == addr) {          /* Exact match, use this entry   */
```

```
            strcpy(name,thisline);   /* in symfile.                    */
            return(1);
        }
        else if (thisaddr > addr) { /* Address in symfile is greater   */
            if (lno == 1) {          /* than incoming address...       */
                break;               /* If first line of symfile       */
            }                        /* then return error.             */
            strcpy(name,lastline);
            *offset = addr-lastaddr; /* Otherwise return the symfile   */
            return(1);               /* entry previous to this one.    */
        }
        else {                       /* Address in symfile is less than */
            lastaddr = thisaddr;     /* incoming address, so just keep */
            strcpy(lastline,thisline); /* a copy of this line and      */
            lno++;                   /* go to the next.                */
        }
    }
    }
    strcpy(name,"???");
    return(0);
}
```

堆栈跟踪函数利用 AddrToSym（）（见程序清单 12.17）将一个地址转换成函数名。这种转换由 symtbl 文件（用 TFS API 来定义 tfsgetline（）并假定按地址的升序排列）的每一行单步执行来实现。当符号表中的地址大于等于引入的地址时，地址被当做是一对字符串和偏移量被返回。如果地址不完全匹配，则只返回偏移量。当没有完全的匹配时，最相近的匹配名字会与偏移量一起显示，例如：

```
uMON> strace
0x8018844a: errCheck() + 0x18
0x800c7f40: serialTest() + 0x40
0x8004006c: TaskAudit() + 0x88
```

此例显示了 TaskAudit（）调用 serialTest（）、serialTest（）调用 errCheck（）及在 errCheck（）中发生的中断。

注意

当客户说你的产品有时会重启时，你该做什么？如果重启只是偶然发生，则原因很难预测，可能是与客户的站点有关。那么你如何找到原因呢？你恐怕不能让一位工程师留在客户站点吧！

但是，如果利用 MicroMonitor 的堆栈跟踪能力和其他的一些能力，则完全可以解决上述问题。此时，你可以配置环境，使异常情况能自动地让监控器将堆栈跟踪转存到文件系统中的一个文件中，并重新启动程序。稍后，你可以找到这个文件，并在自己的实验室中分析。

12.7 检测堆栈溢出

程序使用了两种不同的变量，即静态变量和堆栈变量。静态变量被分派到固定的地址。不管执行什么函数，静态变量的地址总是相同的。堆栈变量常属于特定的函数，在函数声明时可以被访问，并在每次函数运行时生成。按照函数开始时堆栈深度的不同，堆栈变量可以放在内存中不同的地方。

静态变量的使用虽很方便，但却一直占用内存。堆栈变量由当前运行的函数临时指定位置，对于临时存储非常方便。当函数结束时，堆栈变量立即释放空间供给其他的函数。这两种变量类型都是必需的，每一种都有有利和不利的方面。

堆栈变量的分配和再分配由每个函数首尾的代码处理和调用，代码为 prolog 和 epilog。这个不可见的代码由编译器生成，是函数头的一部分，并不是函数的逻辑。在大多数情况下，prolog 和 epilog 给堆栈段分配，再分配足够大的空间来装载函数中声明的变量。分配调整当前的堆栈指针和段指针，提供内存空间的一段给当前运行的函数。当嵌入系统刚建立时，少量的内存分配给堆栈。如果在执行函数的过程中，一些函数嵌套使堆栈指针超过分配给堆栈的空间的结尾，则将造成堆栈溢出。这是最难发现的错误之一。

堆栈溢出发生之后的事情决定于目标内存映像图。在很多单线程系统中，堆栈指针置于内存顶端（让它向下增长），堆（由程序调用 malloc 指定的区域）[1]从文本、数据、BSS 空间的结尾开始并向上增长（如图 12.1 所示）。发生堆栈溢出的位置很

1. 对于嵌入式系统，一个好的原则只避免完全使用 malloc，然而由于一此程序需要 malloc，因此在本书，假设它只需要的。

大程度上是取决于分配给这个堆的空间大小。注意，堆是向上增长的，而堆栈是向下的，并在中间的某个地方相遇。相遇点是好是坏很难确定，因为堆栈和堆都是动态的。

多线程环境更复杂。在多线程环境中，每个任务均有单独的逻辑堆栈。通常，每个任务的堆栈大小可以因为任务的不同而由你指定。这个设计会因为你有 n 个不同的堆栈溢出而使事情变得更复杂。堆栈溢出什么决定于堆栈的位置。如果系统建立时任务堆栈从共同的内存区分配，则溢出可能会毁坏未运行的其他任务的堆栈。如果堆栈段分布于内存，则溢出会毁坏系统中其他的变量。任何一种情况下，堆栈溢出都很难查找，因为失败的代码本身可能并没有任何错误，只是因为其他任务的堆栈溢出而毁坏了代码本身的变量。

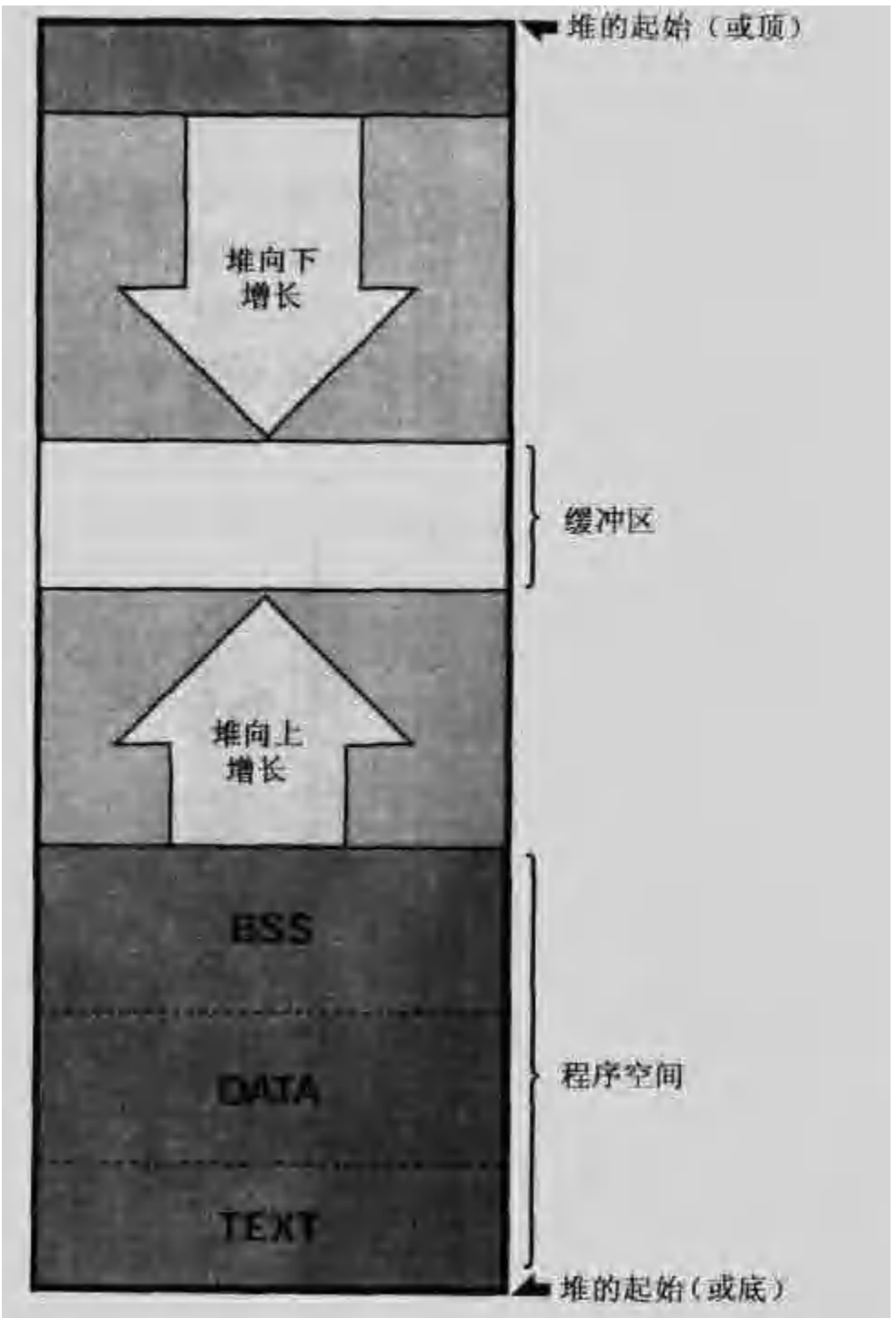


图 12.1　单线程操作存储器框图

在单线程应用程序中，堆栈和堆经常各自向未用数据区方向增长。

这种难懂的特点使堆栈溢出更难处理。几乎所有的嵌入系统都有中断。中断处理通常与运行于程序空间的内容异步，并由 C 编写。中断处理中的 C 代码与其他的并无不同。当中断处理代码通过调整当前的堆栈指针生成堆栈段时，当前堆栈指针就是被中断的任务的堆栈指针。如果当前函数嵌套的任务使它的堆栈指针接近分配给它的内存空间的结尾，则中断的出现会引发堆栈溢出，这是因为在这个堆栈中增加了中断处理代码。

如何处理这个问题呢？理想情况是给每个堆栈分配很多的内存，问题即可解决。在某些情况下可按上述方法解决，但在另一些情况下，我们就要必须花时间来计算哪些函数嵌套、任务或中断条件引发了问题。

12.7.1 预填充堆栈内存或者缓冲区

检测堆栈溢出最通常的办法是用已知的数据预填充被堆栈占用的 RAM 空间，然后检查这个数据是否被完全覆盖。例如，你可以在堆栈内存空间加载 0x55 后，生成附加的任务和一个缓冲检查任务。因其知道每个堆栈的位置，所以可用定时器或其他的事件定期地执行这个任务。如果检测到堆栈的结尾被修改，也就检测到一个堆栈溢出。

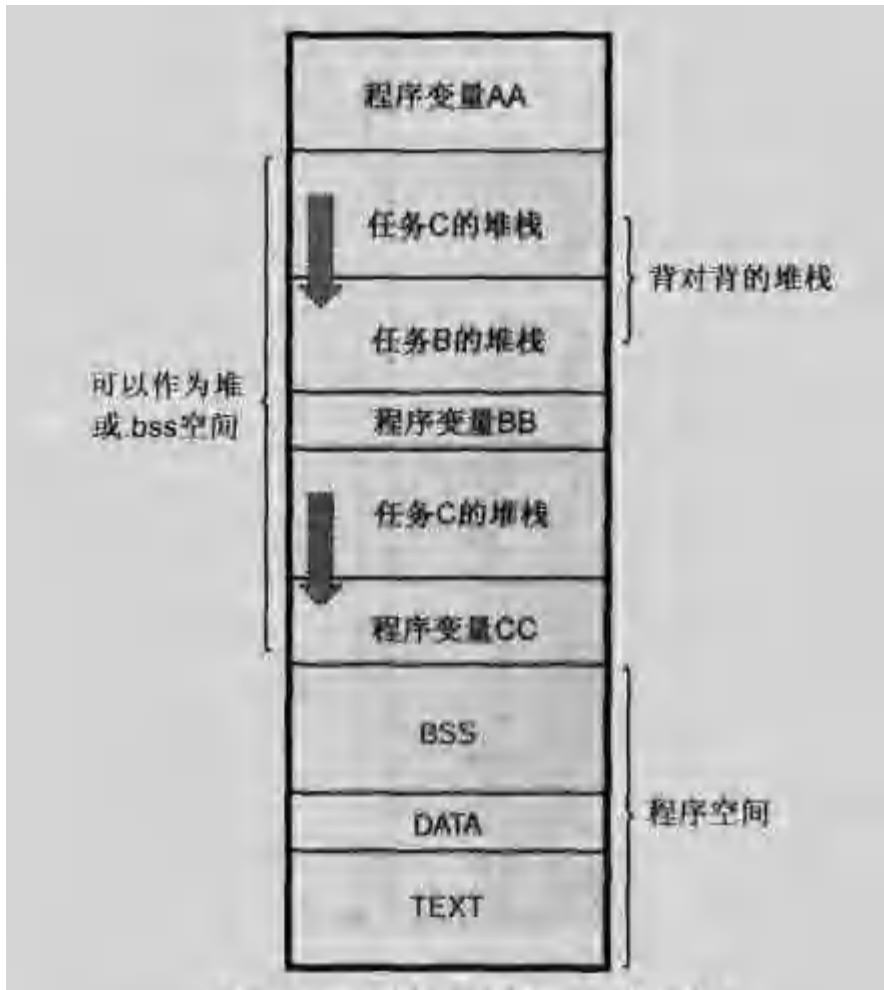


图 12.2 多线程操作存储器框图

在多线程应用程序中，每个任务都有各自的堆栈，这就给检查堆栈溢出带来了极大的困难。

注意，此例只是能够检测到溢出的存在，却不能知道错误的程序，但根据知道的堆栈溢出，你便可以知道引起问题的任务，即可检查这个任务对应的代码。如果有必要利用堆栈，则可以增加它的大小。在如图 12.1 所示的单线程程序中的内存分配情况下，你可以预填充一个缓冲区。如果被修改，你就知道发生了堆栈溢出。

你也可以用这个办法来调整通用的堆栈大小，用数据预填充所有的堆栈或者缓冲区后，根据每一个堆栈中被覆盖的数据来调整已分配的堆栈大小。但这不是最佳的解决办法，而且也很有可能使函数溢出堆栈但却不毁坏数据。这常发生在函数有未用数组的情况下，见程序清单 12.18。

程序清单 12.18　未被检测出堆栈超限

```
func(int arg)
{
    char buffer[32];
    int   val;

    if (arg > 45) {
        sprintf(buffer,"hey, arg is greater than 45!\n");
        func1(buffer);
    }
    val = func2(arg);
    return(val);
}
```

为 buffer 设置的 32B 可能不会被修改。如果接近堆栈段的结尾，并以 arg<45 调用函数，而且堆栈空间的结尾与缓冲分配的空间重叠，则 0x55 将不会被重写，但是溢出还是会毁坏堆栈结尾下面的内存空间。

注意

尽量避免堆栈中的大数组。虽然你可能需要它，但是一定要小心。

12.7.2　利用对每个函数堆栈段的检查

查找堆栈溢出的另一种技巧是在每个函数的顶端插入一个堆栈检查函数。这个

函数类似于汇编函数，能够查找当前的堆栈指针并判定它是否超出了堆栈的结尾。对于多线程系统，这个函数必须知道正在运行的任务，因为不同的任务需用不同的堆栈。

检查堆栈的函数对于捕获真实发生的溢出非常理想。你可以设计这个函数，使其能自动地生成一个异常，并执行堆栈跟踪来捕获引发溢出的函数嵌套。

但堆栈检查函数却会对运行时间产生影响。其执行时间会被添加到系统中的每个函数调用，但最好是有选择地使用它，因一些函数不会有问题。然而，越有选择性，你捕获错误的可能性就越小。如果你使用了可选性，则即使函数没有任何大的数组，也会影响函数。同样，也有可能调用当前函数的函数才是真正有问题的函数。见程序清单 12.18。如果 func2（）除了增加 arg，却不做任何事情并返回值，则也有可能是错误的函数，因为 func（）使堆栈深度加大。

堆栈检查函数在其他方面可能是不实际的。你可能并没有所有源代码，因而不能在每个地方加入这个函数。或者你有源代码，但是可能没有用堆栈检查模块，或编写了大部分的工程后，加入调用的却是既繁琐又容易出错的。

预先准备的办法

应该先提供堆栈检查，也许你需要在工程编码标准中建立一个合适的模块。因为调用的是个模块，所以将它包含在代码之中，但并不代表它被代码所使用。通常，对定义 STKCHK（）为空，此时的调用不生成任何代码。当遇到堆栈问题时，你可以通过改变它的定义来激活模块（见程序清单 12.19）。

程序清单 12.19　创建一个堆栈监测的钩子

```
#include "stackchecker.h"


func()
{
    STKCHK();


    yada, yada, yada
}
```

12.8　系统评测

假设你在做固件开发项目，并且工作在完全并成功配置的硬件平台上，但是仍有些新特性要加入。而你的工作就是要加入这些新特性，但当这些新特性添加之后，现有的特性因为没有足够的能力来支持新增加的特性而无法工作，或者你没有足够的存储空间来增加你的特性。

解决的方案就是放开手脚，并告诉硬件开发者你需要更多。这个策略通常不会很成功，所以你需要一个变通的办法。要决定你是否能够重新安排系统，从而在当前的配置中得到更多的带宽。还要确定是否能够移除一些不用的功能。这些问题可以通过一种叫做系统评测的技术（也叫运行时分析）来解决。

Function counting、task-ID activity 和 basic block coverage 是一些基本的系统评测功能。系统评测中的一些信息对于确定所有的代码被执行时是很有用的，且在你想提高速度的时候也有一些信息可用于决定该优化的代码位置。注意，一些更复杂的 RTOS 包以标准组件或者可选扩展的形式提供了不同类型的系统评测。这些工具将根据 RTOS 的固件项目清楚地被表示出来，系统评测给系统插入或者不插入附加的负担取决于你所用的工具。下面将讨论一些系统评测技巧。这对于那些不想另购插件的开发者是很有用的。

注意

通过利用外部硬件来分析和显示系统运行期的统计数据，你可以得到系统的概貌而无需任何运行期开销。这种硬件技术是很实际的，而且，如果地址线可以从外部访问，就可以对指令缓冲区进行处理。

12.8.1　使用系统节拍（tick）

如果你对一个嵌入系统进行编程，则你的目标机器很可能有一些重要的或者系统节拍性的中断。这些中断通常是系统中最高优先级的中断。你可以用这些中断记录每次中断发生时被中断的代码段或任务。

最简单的程序是收集任务统计数据。当运行任务时，系统中有一些全局指针（如 task ID 和 TID）可以由系统节拍处理句柄访问，使其知道在这个节拍上正在运行的

任务。中断处理句柄可以由计数器来自动加 1，从而知道任务的运行情况。

信息的重要与否决定于正在进行的事情，也取决于指定的任务和特定的代码段。例如，如果正在运行很多任务，且其中的大部分正调用同一个函数，则不管当前运行哪个任务，均不能使你知道应优化哪个函数。但如果每个任务主要执行一些独特的代码，那么 task ID 记录便可以为你指明方向。

另一种解决的办法是记录每个系统节拍上正在运行的函数，可采用下面两种不同的方法来记录正在运行的函数：一种是在系统节拍上加入负担；另一种决定于是否有很大的 RAM 可用。

第一种方法是先生成程序并收集函数统计。当知道函数统计信息后，在系统节拍中插入的函数调用可将系统节拍中断代码的地址传给知道地址和大小的已生成函数，通过地址的比较，可以确定正在执行的函数并增加相关的计数器。因为此方法给每个时钟节拍均增加了复杂的查找，所以可能会给处理器增加很重的负担。

第二种方法虽提供了更好的性能，但却需要更多的 RAM。采用这个方法时，你必须在有另一块与.text 空间同样大的 RAM 后，系统节拍才会将被中断的代码地址传给一个函数。这个函数将这块 RAM 当做计数器表来处理，用引入的地址当做这个表的偏移量，而且可以在这个偏移量上修改计数器。这种解决办法是快速的，并可明确地指出正在运行的那些代码。

MicroMonitor 中的 prof 命令支持前面所说的一些概念。程序清单 12.20 给出了相关的代码。

程序清单 12.20　prof 命令

```
struct pdata {
    ulong     data;        /* Start of symbol or tid. */
    int       pcount;      /* Pass count.             */
};


/* prof_FuncConfig():
 * This function builds a table of pdata structures based on the
 * content of the symbol table.
 * It assumes the file is a list of symbols and addresses listed
 * in ascending address order.
 */
void
prof_FuncConfig(void)
```

```
{
    int         tfd, i;
    struct    pdata *pfp;
    char       line[80], *space;

    tfd = prof_GetSymFile();
    if (tfd < 0) {
        return;
    }

    prof_FuncTot = 0;
    pfp = prof_FuncTbl;

    while(tfsgetline(tfd,line,sizeof(line) -1)) {
        space = strpbrk(line,"\t ");
        if (!space) {
            continue;
        }
        *space++ = 0;
        while(isspace(*space)) {
            space++;
        }
        pfp->data = strtoul(space,0,0);
        pfp->pcount = 0;
        pfp++;
        prof_FuncTot++;
}
tfsclose(tfd,0);

/* Add one last item to the list so that there is an upper limit for
 * the final symbol in the table:
 */
pfp->pdata = 0xffffffff;
pfp-pcount = 0;
```

```
        /* Test to verify that all symbols are in ascending address order...
         */
        for (i=0;i<prof_FuncTot;i++) {
            if (prof_FuncTbl[i].data > prof_FuncTbl[i+1].data) {
                printf("Warning: function addresses not in order\n");
                break;
            }
        }
        prof_FuncTot++;
    }
```

在用户想建立基于 symtbl 文件内容的 pdata 结构表时，Prof 命令调用 prof_FuncConfig（）函数。与堆栈跟踪类似，prof_FuncConfig（）假设文件中的符号地址升序排列，然后遍历文件并为符号表文件中的每一行生成 pdata 入口。表在由用户用 prof 命令建立的 prof_FuncTbl 所指定的位置产生。

程序清单 12.21　profiler()

```
#define HALF(m) (m >> 1)

static int prof_Enabled;        /* If set, profiler runs; else return.          */
static int prof_BadSymCnt;      /* Number of hit, but not in a symbol.          */
static int prof_CallCnt;        /* Number of times profiler was called.         */
static int prof_FuncTot;        /* Number of functions being profiled.          */
static int prof_TidTot;         /* Number of TIDs being profiled.               */
static int prof_TidTally;       /* Number of unique TIDs logged so far.         */
static int prof_TidOverflow;    /* More TIDs than the table was built for.      */

static struct pdata *prof_FuncTbl;
static struct pdata *prof_TidTbl;
static char prof_SymFile[TFSNAMESIZE+1];

void
profiler(struct monprof *mpp)
{
```

```
struct pdata *current, *base;
int nmem;

if (prof_Enabled == 0)
    return;

if (mpp->type & MONPROF_FUNCLOG) {
    nmem = prof_FuncTot;
    base = prof_FuncTbl;
    while(nmem) {
        current = &base[HALF(nmem)];
        if (mpp ->pc < current->data) {
            nmem = HALF(nmem);
        }
        else if (mpp->pc > current->data) {
            if (mpp ->pc < (current+1) ->data) {
                current->pcount++;
                goto tidlog;
            }
            else {
                base = current + 1;
                nmem = (HALF(nmem)) - (nmem ? 0 : 1);
            }
        }
        else {
            current->pcount++;
            goto tidlog;
        }
    }
    prof_BadSymCnt++;
}
tidlog:
    if (mpp ->type & MONPROF_TIDLOG) {
        /* First see if the tid is already in the table.  If it is,
```

```
 * increment the pcount.    If it isn't add it to the table.
 */
nmem = prof_TidTally;
base = prof_TidTbl;
while(nmem) {
        current = &base[HALF(nmem)];
        if (mpp ->tid < current->data) {
                nmem = HALF(nmem);
        }
        else if (mpp->tid > current->data) {
                base = current + 1;
                nmem = (HALF(nmem)) - (nmem ? 0 : 1);
        }
        else {
                current->pcount++;
                goto profdone;
        }
}
/* Since we got here, the tid must not be in the table, so
 * do an insertion into the table.    Items are in the table in
 * ascending order.
 */
if (prof_TidTally == 0) {
        prof_TidTbl->pcount = 1;
        prof_TidTbl->data = mpp->tid;
        prof_TidTally++;
}
else if (prof_TidTally >= prof_TidTot) {
        prof_TidOverflow++;
}
else {
        current = prof_TidTbl + prof_TidTally -1;
        while(current >= prof_TidTbl) {
                if (mpp ->tid > current->data) {
```

```
                current++;
                current->pcount = 1;
                current ->data = mpp ->tid;
                break;
            }
            else {
                *(current+1) = *current;
                if (current == prof_TidTbl) {
                    current->pcount = 1;
                    current->data = mpp ->tid;
                    break;
                }
            }
            current- -;
        }
        prof_TidTally++;
            }
        }
    profdone:
        prof_CallCnt++;
        return;
    }
```

profiler（）函数（见程序清单 12.21）将被应用程序的系统标号所调用（通过连接第 11 章提到的微型监控器的应用程序接口 mon_）。应用程序代码会开辟一个 monprof 结构并通知 profiler 去执行 TID 的与/或函数，指定 TID 为与/或路径控制程序。

函数（MONPROF_FUNCLOG）调用 prof_FuncConfig（）创建的表格。为实现这项服务，profiler（）在表格（使用二进制搜索）中搜索引入的地址及相应符号地址分配间的匹配。每找到一个匹配，匹配标识计数就会加 1。

TID 函数假定用户已经创建了最大数量预想的特殊 TID 值，一个那么大的 pdata 结构表格已经存在（初始化为空值），起始位置由 prof_TidTbl 指定。当每次调用 profiler（）时，随着预想的特殊 TID 值的插入创建表格，以便使这些条目按升序排列。在引入的 TID 和表格中，这一方法可使 profiler（）在已经存在的条目间进行

快速地二进制搜索以找到匹配。如果找到一项匹配，则标识计数会增加；若未搜索到匹配，则 TID 就会添加到表格中，计数被设置为 1。

程序清单 12.22 打印 profile 统计表

```
void
prof_ShowStats(int minhit, int more)
{
    int         i, tfd, linecount;
    ulong       notused;
    char        symname[64];
    struct  pdata     *pptr;

    printf("FuncCount Cfg: tbl: 0x%08x, size: %d\n",
        prof_FuncTbl, prof_FuncTot);
    printf("TidCount   Cfg: tbl: 0x%08x, size: %d\n",
        prof_TidTbl, prof_TidTot);

    if (prof_CallCnt == 0) {
        printf("No data collected\n");
        return;
    }
    linecount = 0;
    tfd = prof_GetSymFile();
    if ((prof_FuncTbl) && (prof_FuncTot > 0)) {
        printf("\nFUNC_PROF stats:\n");
        pptr = prof_FuncTbl;
        for(i=0;i<prof_FuncTot;pptr++,i++) {
            if (pptr->pcount < minhit) {
                continue;
            }
            if ((tfd < 0) ||
                (AddrToSym(tfd,pptr->data,symname,&notused) == 0)) {
                printf(" %08x      :   %d\n",pptr->data,pptr->pcount);
            }
```

```
                else {
                    printf(" %-12s:    %d\n",symname,pptr->pcount);
                }
                if ((more) && (++linecount >= more)) {
                    linecount = 0;
                    if (More() == 0) {
                        goto showdone;
                    }
                }
            }
        }
        if ((prof_TidTbl) && (prof_TidTot > 0)) {
            printf("\nTID_PROF stats:\n");
            pptr = prof_TidTbl;
            for(i=0;i<prof_TidTot;pptr++,i++) {
                if (pptr->pcount < minhit) {
                    continue;
                }
                printf(" %08x       :    %d\n",pptr->data,pptr->pcount);
                if ((more) && (++linecount >= more)) {
                    linecount = 0;
                    if (More() == 0) {
                        goto showdone;
                    }
                }
            }
        }
    showdone:
        putchar('\n');
        if (prof_BadSymCnt) {
            printf("%d out-of-range symbols\n",prof_BadSymCnt);
        }
        if (prof_TidOverflow) {
            printf("%d tid overflow attempts\n",prof_TidOverflow);
```

```
        }
        printf("%d total profiler calls\n",prof_CallCnt);

        if (tfd >= 0) {
            tfsclose(tfd,0);
        }
        return;
    }
```

当 profiling 完成时，这两个函数所搜集的统计资料可以通过 prof 命令转存至用户处。Prof 命令实际上调用了 prof_ShowStats（）函数。prof_ShowStats（）函数（见程序清单 12.22）将这两个 pdata 结构的表格打印至控制台。用户可以直观地看出选中的数量，只打印大的命中项。用户也可以指定更多的限制，因为若程序中用到的函数多，则输出可能相当大。

12.8.2 基本模块

基本模块（BBC）的性能取决于编译程序，且会增加应用程序的运行时间。编译程序为代码中的每个基本模块嵌入一个计数器。系统测试组用这个计数器来确认是否包括了所有的被测试代码。目标经过测试后，由 BBC 的结果可以看出哪些行的代码漏掉了。虽然此解决办法并不完美，但它却向成功迈出了第一步。BBC 仅仅能覆盖代码，并不能覆盖所有功能。

注意

基本模块是一系列代码。这些代码总是作为一个整体（亦可中断）来执行的。换句话说，如果执行模块中的一行代码，则这个模块中的每一行代码都会被执行。在 C 语言中，模块通常包含在大括号（{}）中。

程序清单 12.23 BBC 举例

```
void
funcX(void)
{
    if (aa == 15)
```

```
        {
            Block 1 of funcX().
        }
        if ((aa == 15) || (bb == 35))
        {
            Block 2 of funcX().
        }
    }
```

程序清单 12.23 的 BBC 技术表明，即便只调用 aa==15，也会使整个模块都被执行。因此，bb==35 和 aa!=15（包含在 funcX 中，但仅在 Block 2 中被调用）并没有被测试到。

12.9　小结

当然，基于监控器的调试并不能提供适应当前可能获得的一些桌面开发环境的所有功能，但却能提供一个可与应用程序一起驻留于系统中的开发环境。因此，基于监控器的调试支持现场调试，而无需扩展连接或附加一个串行口。这种便利是无价之宝。基于监控器的调试并不妨碍任何更加复杂的开发环境存在于监控器平台上。

第 13 章　将微型监控器接入 ColdFire MCF5272

本章将简述如何把微型监控器连至摩托罗拉 ColdFire MCF5272C3 评估板。假设你已经对微型监控器的命令设置比较精通，而且你有权使用原始资料代码（所有代码均在 CD 上）。在此，仅论述 ColdFire 中与连接过程有关的细节。本章论述的重点不在于 MCF5272C3 或 ColdFire 的结构，而在于其与监控器的连接过程。

选择 MCF5272C3 评估板的原因如下：

- MCF5272C3 CPU 有一套性能很好的内置外设，尺寸和复杂度对于微型监控器来说都很匹配；
- MCF5272C3 的核心与摩托罗拉 68000 系列微处理器很相似；
- 摩托罗拉的网站上提供了大量的有关 CPU 和评估板的信息，包括引导监控器的源代码；
- 尽管 ColdFire 的 BDM 接口在连接中并未用到，但它的确是这类产品中的佼佼者；
- 装载在监控器中的工具可以很容易地安装微型监控器。实际上，无需修改默认的引导监控器就可安装微型监控器。

使用评估板能够马上做出一些方便的假设，但假设不能与新硬件冲突。假设硬件已经通过测试，且得到了正确的图表，摩托罗拉引导监控器中的范例代码却能说明如何正确地操纵硬件。

提示

如果工程包括开发新硬件，则你可先编写虚拟硬件，并为 CPU 配一块评估板，使硬件设计者以评估板为向导进行设计。如果新硬件以评估板为向导，则你可以对评估板上的虚拟硬件进行操作，当真正的硬件做出以后，向真正硬件转换会很容易。

13.1　原始资料代码目录树

在为评估板编写代码之前，必须看一下 CD 上的微型监控器的原始资料代码目录树。目录树基本上有两大高级目录即公共目录和目标目录。公共目录包括能在许多不同平台间通用的代码。目标目录包括子目录，每个子目录指向一个特定的目标。在仅对代码做少量修改的情况下，这两大目录即可把监控器的原始资料用于大不相同的体系结构中。

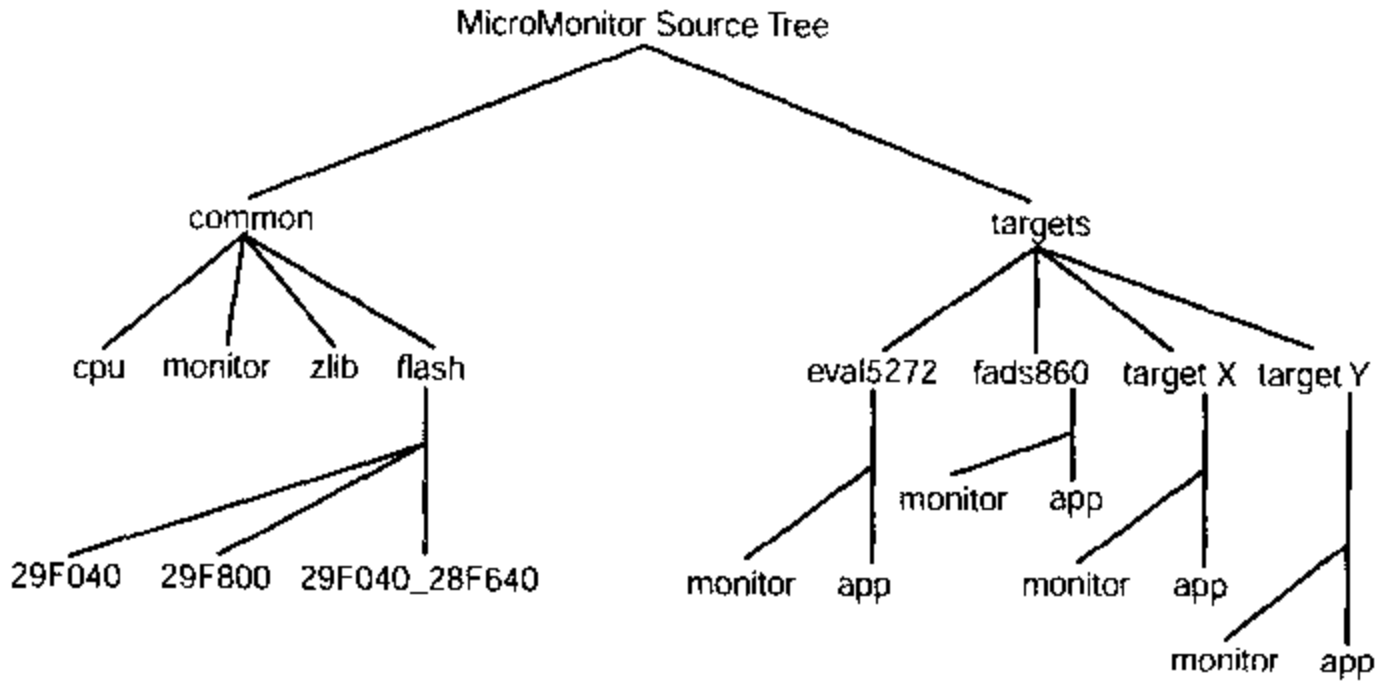


图 13.1　微型监控器原始资料树

公共目录和目标目录下的子目录如下：

- common/cpu——包括 CPU 专用而不是目标专用的代码，如反汇编程序、异常处理、为 CLI 的特定 CPU 命令等。
- common/monitor——包括大量的代码。所有 100%独立于目标/CPU 的软件都在此目录下，如以太网设备 TFS、文件编辑器、大部分的监控器命令、内存分配算符及 CLI 处理器，等等。所有监控器的核心设备都在此目录下。
- common/zlib——包括 zlib 全局代码中几乎完全没有变动的素材。为适合监控器，对 zlib 代码做了极少的改动。这些改动独立于一个文件中。
- common/flash——包括支持特定闪存结构的子目录。这一结构可能是一个 29F040 或 29F800，或者可能是一个目标。该目标带有用于 TFS 中存储的 28F640 的 29F040 引导设备。每个子目录代表一个闪存结构，每个闪存结构可能对许多目标系统都适用。
- targets/fads860——包括（同其他 targets/xxx 目录）对特定的目标系统生效

的代码。对于一个给定目标（在此情况下指 FADS860 评估板）的所有特殊的、不可再度使用的素材都在这个子目录下。

- targets/eval5272——与其他特定目标子目录一样，这里包含复位代码、通信驱动器（串行口和以太网）以及主函数 main（）。图形文件连接、编译文件以及其他与创建有关的文件也包含在此。主要的连接工作都在此目录下完成。

在每个 targets/xxx 目录下是一个 monitor 和 app 目录。monitor 目录包括所有的为监控器所用的特定目标代码。app 目录包括一个基本的应用程序举例。该程序可以在连上监控器后进入 TFS 内并执行。

13.2 编译文件

编译文件包括少量存在于公共目录下的其他.make 文件。每个特定目标编译文件包括 common/monitor/common.make、common/zlib/zlib.make 和 common/monitor/tools.make。这三个.make 文件把公共素材放在一处，使得特定目标编译文件最小。当然，做这项工作还有技巧性更强的办法，但是对于我来说，只要简便就足够了。

注意

我习惯使用 UNIX 操作系统，并且特别喜欢使用不同的基于 UNIX 的命令解释程序及许多 UNIX 的工具（有充分的理由）。尽管这样，我还是喜欢在微机上创建嵌入式开发系统，以便能够灵活操作。因为我用 UNIX 系统比较顺手，所以碰到 MicroCross GNU 十字形工具很开心，这个工具为微机提供了一个类 UNIX 的交叉开发环境。十字形工具箱包括了预置的 GNU 交叉编译程序。这个程序适用于许多 CPU 体系机构、bash shell 和其他 Win32 平台下的 UNIX 工具（find，grep，rm，cp，ls，awk，sed 等等）。在 Win32 操作系统下，以 bash shell 作为控制台可以两方面兼顾。这样，即便正在微机上工作，也可以使用 make 来运行工程文件，假设类 UNIX 工具是可用的。基本的 MicroCross 工具设置和安装示范都包括在本书的 CD 中。

common.make 文件包括了所有公共模块的生成目标，增加了用 GNU 交叉开发工具来创建各种不同的输出程序清单的便利。这些附属的目标包括符号表格、S 记录及 C/反汇编程序混合的文件。Common.make 文件还为公共的 clobber 和 clean 提

供了工具及一个为特定目标资源创建.tar 文件的目标。

要实现所有这些便利，特定目标编译文件必须遵循一些方针。在编译文件中，这些方针的主体包括初始化某些 make 变量，并被 common.make 文件调用。另一些方针是特定目标监控器目录下的所有对象模块都放置在一个 obj 目录下。

程序清单 13.1~程序清单 13.4 列出了连接中要用到的基本编译文件。

程序清单 13.1　基本编译文件

```
#############################################################################
#
# Makefile for building a monitor for the MCF5272 evaluation platform.
#
# Currently, the only target supported for 5272 platform is EVAL.  When
# the platform count increases to more than 1, the target name should be
# specified on the command line.
#
# NOTE: This port has only been tested running in RAM space of the
# MCF5272 eval board.  It is downloaded using the DBUG command "DL" or "DN".
#
PLATFORM    = CFEVAL

FLASH       = 29pl160c
TGTDIR      = eval5272

MONBASE     = ../../..
TGTBASE     = $(MONBASE)/targets/$(TGTDIR)
COMBASE     = $(MONBASE)/common
COMCPU      = $(COMBASE)/cpu
ZLIB        = $(COMBASE)/zlib
COMMON      = $(COMBASE)/monitor
FLASHDIR    = $(COMBASE)/flash/$(FLASH)
INFO        = info

TARGET      = m68k-coff
include     $(COMMON)/tools.make
```

```
CFLAGS      = -Wall -D PLATFORM_$(PLATFORM)=1 -Wno-format \
              -fno-builtin -msoft-float -g -c -m5200 -I. -I$(COMMON) \
              -I$(COMCPU) -I$(FLASHDIR) -o $@
ASFLAGS     = -m5200 -o $@
ASMCPP      = cpp -D PLATFORM_$(PLATFORM)=1 -D ASSEMBLY_ONLY \
              -I$(COMCPU) -I$(COMMON)
LDFLAGS     = -Map=$(AOUT).map
AOUT        = mon$(PLATFORM)
LIBS        = libz.a $(LIBGCC)
```

在程序清单 13.1 中，变量的最高设置（top set）是用来确认监控器原始资料代码的驻留环境的。这个环境支持包含文件，并且允许 common.make 文件完成一些基于这些变量的公共目录中的 convenient-I 变量。其他变量则是编译文件的典型变量。PLATFORM 变量的值可使这一目录和编译文件潜在地支持多个平台。下面讨论 config.h 文件时会着重讲述这一特性。

MicroCross 十字形工具箱中的预置工具是以名字来区分的。名字的形式为<目标><输出文件类型><基本工具名>。如要用 ELF 格式对象文件为 ARM 板生成代码，则需要调用名为 arm-elf-gcc 的编译程序。程序清单 13.1 包含的 tools.make 文件用变量 TARGET 的值来创建十字形工具的兼容工具名（CC，LD，ASM 等）。对 ColdFire 连接来说，其 CPU 是摩托罗拉 68KB 家族（m68k）的一员，要生成 COFF 输出文件，则 TARGET 应被设置为 m68k-coff。

程序清单 13.2　ColdFire 5272 编译文件对象清单

```
OBJS=obj/reset.o obj/start.o obj/cpuio.o obj/chario.o obj/mprintf.o \
obj/main.o obj/mstat.o obj/sbrk.o obj/malloc.o obj/docmd.o obj/cmdtbl.o \
obj/go.o obj/env.o obj/memcmds.o obj/xmodem.o obj/flash.o obj/except.o \
obj/flashpic.o obj/flashdev.o obj/ethernet.o obj/reg_cache.o obj/vectors.o \
obj/tfs.o obj/if.o obj/misccmds.o obj/genlib.o obj/edit.o obj/lineedit.o \
obj/tfsapi o obj/tfsclean1.o obj/tfscli.o obj/tfslog.o obj/symtbl.o \
obj/tfsloader.o obj/redirect.o obj/monprof.o obj/bbc.o obj/etherdev.o \
obj/icmp.o obj/arp.o obj/nbuf.o obj/dhcpboot.o obj/dhcp_00.o obj/tftp.o \
obj/tcpstuff.o obj/crypt.o obj/password.o obj/moncom.o obj/cache.o \
obj/misc.o obj/dis_cf.o
```

```
include $(ZLIB)/zlib.objlist
```

程序清单 13.2 的对象列表是基于将成为监控器组成部分的特性。这些文件的大部分是对象文件，该对象文件是从公共空间中建立起来的。这里，对象文件是指定的，以便使每个创建的目标能够仅包含该应用程序所需的特性。因此，这个清单针对的目标是详细而明确的。需要注意的重要一点，即必须假定 reset.o 是程序清单中的第一个模块。这一假定使连接器能够把 reset.s 中的代码放到内存图的开始，注意包含 zlib.objlist 文件。zlib.objlist 文件（位于 common/zlib 目录下）包括建立 zlib 压缩工具的对象。因为这个清单总是以同样的方式调用，所以它保存在公共空间。如果想使用它，则很容易调用。

程序清单 13.3　发行版与开发版

```
#######################################################################
#
# rom:
# Standard monitor build, destined for installation using newmon tool.
#
rom: $(INFO) $(OBJS) libz.a makefile
        $(LD) $(LDFLAGS) -TROM.lnk -nostartfiles -e coldstart \
        -o $(AOUT) $(OBJS) $(LIBS)
        coff -m $(AOUT)
        coff -B $(AOUT).bin $(AOUT)

#######################################################################
#
# ram:
# Version of monitor for download into RAM.
#
ram: $(INFO) $(OBJS) libz.a makefile
        $(LD) $(LDFLAGS) -TRAM.lnk -nostartfiles -e coldstart \
        -o $(AOUT) $(OBJS)
        coff -m $(AOUT)
        coff -B $(AOUT).bin $(AOUT)
```

每一个监控器的安装均需选择一个可以导出闪存的发行版（见程序清单 13.3

中的 rom 标记符)或者可以装入 RAM 的开发版(见程序清单 13.3 中的 ram 标记符)。(测试新的监控器特性时经常使用 RAM 驻留文本。对于本例中的连接，使用 ram 标记符创建一个可下载至没有被评估板的 DBUG 监控器占用的 RAM 空间的监控器)。rom 标记符在编译文件中是最高的标记符，所以它是默认的。

注意

coff 工具（程序清单 13.3 中涉及）可以被各种有着相似性能的 GNU 工具所取代。我把 coff 工具当成练习来写，为的是加深自己对 COFF 文件格式的理解。-m 选项堆存了一个可执行的内存图，-B 选项在执行的基础上建立了一个二进制图像。正如在本书前面的章节中所论述过的，同样的性能也适用于 COFF、ELF 和 A.OUT 文件格式。其原始资料和可执行文件在 CD 上。

注意$（AOUT）变量的用法。该变量用于整个目标编译文件中，common.make 文件作为不同文件的基本命名是由编译过程产生的。在这种情况下，AOUT 被设置成 mon$（PLATFORM）、PLATFORM 被设置成 CFEVAL 后，AOUT 就是 monCFEVAL。基本命名用于许多其他的输出，像原始二进制图像的 monCFEVAL.bin、S-record 中的 monCFEVAL.srec 及符号文件中的 monCFEVAL.sym，等等。这一简单的命名方案使目标之间更有条理了，再使用一个公共的基本命名，使 clobber 标记符可以更方便地移动与特定目标有关的所有文件。

程序清单 13.4　其他准则部分

```
#############################################################################
#
# Miscellaneous rules:
#
include $(COMMON)/common.make
include $(ZLIB)/zlib.make

libz.a: $(ZOBJS)
        m68k-coff-ar rc libz.a $(ZOBJS)

info:
```

```
defdate -f %H:%M:%S BUILDTIME >info.h
defdate -f %m/%d/%Y BUILDDATE >>info.h
```

准则部分（见程序清单 13.4）包括两项其他的公共.make 文件。由于包含了这些文件，因此目标的核心设置和它们的从属也被加载进来。libz.a 标记符为一个库创建了 libz 对象列表。Info 标记符创建了一个包含创建日期和时间的简单头文件（info.h）。info.h 文件被监控器的源文件之一所包含，为的是提供部分由微型监控器的 version 命令所报告的信息。

交叉平台日期工具

与 info.h 文件一起创建的另一自带工具叫做 defdate（原始资料和可执行文件在 CD 上）。之所以写 defdate，是因为我寄希望于 UNIX 和 Win32，但都未能找到创建一个可以被 #include 文件包含的数据链交叉平台的好方法。如果你看一下 CD 上的原始资料，则会发现写 defdate 大约会花费 5min 时间。

程序清单 13.5　info.h

```
info.h:

#define BUILDTIME "12:26:16"
#define BUILDDATE "12/23/2000"

function called by 'Version' command:

void
ShowVersion(void)
{
    printf("Monitor built: %s @ %s\n",BUILDDATE, BUILDTIME);
}
```

程序清单 13.5 列出了 info.h 中的典型内容（正如它会由 defdate 工具创建），并且说明了如何被监控器的 vesion 命令所引用。注意大多数编译程序提供的_DATE_和_TIME_的基本定义。当程序中用到的编译程序不支持这些定义时，defdate 就显得很便利了。

程序清单 13.6 中列出的单个模块标记符列表清楚地显示了建立在文件顶部各种命令解释的程序变量。注意，最后一条准则中把汇编过程分成两步。选择使用两步的过程，无论如何都能胜过在编译文件中的工具设置从属。注意，所有的模块都在 obj 目录下。这一划分打乱了包含原始资料代码的目录中的对象模块。

程序清单 13.6 单个模块目标

```
#########################################################################
#
# Individual modules:
#
obj/dis_cf.o:     $(COMCPU)/dis_cf.c $(COMMON)/genlib.h config.h
      $(CC) $(CFLAGS) $(COMCPU)/dis_cf.c

obj/flashdev.o: $(FLASHDIR)/flashdev.c
      $(CC) $(CFLAGS) $(FLASHDIR)/flashdev.c

obj/main.o: main.c config.h cpu.h $(COMMON)/tfs.h \
      $(COMMON)/genlib.h $(COMMON)/ether.h \
      $(COMMON)/monflags.h $(COMMON)/stddefs.h
      $(CC) $(CFLAGS) main.c

obj/reset.o:      reset.s   config.h
      $(ASMCPP) reset.s >tmp.s
      $(ASM) tmp.s
      rm tmp.s
```

编译文件最后的一部分（见程序清单 13.7）设置了一个为 common.make clobber 标记符所用的 make 标记符。马上你就会看到 common.make 有了 clobber 目标，但为了使 clobber 用户化，必须依靠由目标编译文件提供的 clobber1。

在汇编语言中调用 cpp

我觉得汇编语言之间的细微差别很讨厌，因不同的汇编语言使用不同的注释分隔符和不同指令所以处理这些细微的差别太繁琐，并用一个部分“规范化”的语法编写了汇编程序代码后，用 C 预处理程序把代码翻译成特殊汇编程序所

要求的格式。这样的翻译办法可使用所熟悉的 C 语言的注释分隔符（/*和*/）。更重要的是，可以在汇编程序文件的顶部使用 # include，使其在汇编程序和 C 语言中使用同样的头文件。在编写虚拟硬件时使用同样的头文件是很方便的。这项技术应用很广，许多编译程序都支持并通过一些命令行选项预处理汇编语言文件的功能。即使你的汇编程序并不支持这一特性，你仍可以通过两步的翻译过程达到同样的目的。在“规范化”的汇编文件上运行 CPP 后，编译预处理程序的输出结果。

程序清单 13.7　clobber1 目标

```
#############################################################################
#
# Miscellaneous utilities:
# (generic utilities are in $(COMMON)/common.make)
#
clobber1:
    rm -f $(AOUT).map
```

程序清单 13.8 中列出了认为 common.make 文件中最有意思的部分。Common.make 的第一部分（清单中未列出）由每个公共模块的标记符和一组公共工具标记符所组成。这些工具（正如 common.make 中 help 标记符中的文本所示）提供了普遍用于交叉编译过程的便利，如 S-records 或二进制的转化、符号表格的产生及同时包含 C 语言和汇编语言源程序的文件生成，等等。注意，common.make 取决于编译文件顶部设置的一些变量。

程序清单 13.8　其他公共目标

```
#############################################################################
#
# COMMON miscellaneous targets:
#
clean:
    rm -rf obj
    rm -rf libz.a symtbl
    mkdir obj
```

```
clobber:        clean clobber1
        rm -f $(AOUT)
        rm -f $(AOUT).bin $(AOUT).srec $(AOUT).fcd $(AOUT).dis $(AOUT).sym

tar:        clean
        rm -f $(AOUT).srec $(AOUT).sym    $(AOUT).fcd $(AOUT).dis $(AOUT).tar
        /bin/sh -c "cd $(MONBASE) ; \
        tar -cf $(AOUT).tar common/monitor common/zlib common/cpu \
        common/flash/$(FLASH) targets/$(TGTDIR)/app targets/$(TGTDIR)/monitor"
        mv S(MONBASE)/$(AOUT).tar .

gnusrec:
        $(OBJCOPY) -F srec $(AOUT) $(AOUT).srec

bin2srec:
        bin2srec $(AOUT).bin > $(AOUT).srec

bindump:
        $(OBJDUMP) --full-contents $(AOUT) >$(AOUT).fcd

showmap:
        $(OBJDUMP) --section-headers $(AOUT)

dis:
        $(OBJDUMP) --source --disassemble $(AOUT) >$(AOUT).dis

disx:
        $(OBJDUMP) --source --disassemble --show-raw-insn $(AOUT) >$(AOUT).dis

sym:
        $(NM) --numeric-sort $(AOUT) >$(AOUT).sym

symtbl: sym
        monsym -p0x $(AOUT).sym >symtbl
```

```
help:
    @echo "gnusrec        : use objcopy to produce $(AOUT).srec"
    @echo "bin2srec       : use bin2srec to produce $(AOUT).srec"
    @echo "showmap        : display section headers"
    @echo "dis            : source/assembly dump to $(AOUT).dis"
    @echo "disx           : like dis, but show instruction in hex & symbolic"
    @echo "sym            : numerically sorted symbol table dump to $(AOUT).sym"
    @echo "symtbl         : rearrange output of sym to create monitor's symtbl file"
    @echo "bindump        : ascii-coded hex full-content dump to $(AOUT).fcd"
    @echo "clean          : delete entire obj directory"
    @echo "tar            : create a tar file of the current source and binary."
```

上述即为编译文件的其他公共目标及其内容。很简单，其规则同样也适用于其他的目标和编译程序，适用于 Win32 和 UNIX。

13.3　头文件的结构

为维持控制监控器构造中所有文件某些类型的所有配置，应在配置中的每个文件 #include 清单的顶部放置一个名为 config.h 的文件。其他部分对 config.h 中的每个定义进行预排，并解释如何调整各种操作参数。

13.3.1　FORCE_BSS_INIT

CPU 复位时，控制权移交给位于 CPU 复位向量地址中的固定代码，reset.s 文件中的代码应该映射到该位置。完成最低水平的初始化之后，reset.s 就调用 start（）（第一个 C 语言函数）。复位代码传过来一个参数，告知 start（）函数刚才进行的是哪种类型的启动（热启或冷启）。如果是冷启（指的是由电源上电或按下系统复位按钮），则监控器的所有.bss 空间必须初始化为零。如果不是冷启，则.bss 不变。这样，在复位保持有效之前状态已被设置完毕，是否要对.bss 进行初始化，取决于由 reset.s 中的汇编语言代码设置的一个引入参数。

添加新硬件时，为保证.bss 进行初始化，应使其独立于开启类型之外。定义 FORCE_BSS_INIT 可使这种强制的.bss 初始化发生在 start（）。一旦目标引导源代码稳定了，则应移走这个定义，以便使.bss 能正确存在或调用。

13.3.2 PLATFORM_XXX

通常，目标目录下的每个目录都对应一个特定的目标系统。这种目录结构使特定目标接口在它们自己的目录空间下很有条理。但由于有时有多个目标在结构上很接近，因此用目录把它们区分开就没有意义了。config.h 中，PLATFORM_XXX 定义（见程序清单 13.9）的编译文件和任何一个特定目标文件都支持几乎完全相同的目标间的细微差别。

复位时，监控器给控制台端口加了一个冗余的头文件。头文件中的一行是平台的名字，是从 PLATFORM_NAME 定义中得到的（见程序清单 13.9）。因 ColdFire 端口仅支持一个平台，所以这个定义并不完全有必要。但随着模块的建立，它会使各端口之间比较一致。当然，如果将来出现一种与评估平台很接近的设计，则我们的设计也会与它兼容。

程序清单 13.9　PLATFORM_XXX 的定义

```
#if PLATFORM_CFEVAL
#define PLATFORM_NAME       "Coldfire 5272 Evaluation Board"
#else
#error                      "Platform name is not specified"
#endif
```

13.3.3 闪存结构

因为闪存驱动器独立于它们所使用的空间，所以必须给它们提供一些有关设备在目标存储空间中驻留地址的基本信息。这一信息是由程序清单 13.10 中的定义来提供的。

程序清单 13.10　闪存组合结构定义

```
/* Flash bank configuration:
 */
#define FLASHBANKS                  1
#define FLASH_BANK0_WIDTH           2
#define FLASH_BANK0_BASE_ADDR       0xFFE00000
#define FLASH_PROTECT_RANGE         "0–6"
```

从闪存驱动器这一章的内容可知，闪存有很多分区。这个平台仅包括一个闪存设备。如果多于一个，则在 FLASHBANKS 中会反映出来，即会有若干组 FLASH_BANKX_WIDTH（以字节为单位）和 FLASH_BANKX_BASE_ADDR（CPU 内存空间中设备的开始点）的定义。FLASH_PROTECT_RANGE 的最终定义用来告诉闪存驱动器有哪些部分需运用软件保护（参见有关闪存驱动器软件保护细节的章节内容）。这部分配置能够连接多个分区和/或设备组，并能够详细指明复合配置（用连字符配置多个部分，用逗号分隔多个范围）。

13.3.4　TFS 结构

编译 TFS 时，必须告知它在存储器中的位置及备用部分驻留在存储器中的位置。config.h 中针对 TFS 的定义使 TFS 允许任意选择复合的、不相连的存储模块（TFS 设备）。这里的结构数据需与 tfsdev.h 头文件中的信息配合起作用。如果支持多个 TFS 设备，则 tfsdev.h 头文件中就定义了用来描述多个模块的结构或者结构表。

程序清单 13.11　config.h 中的 TFS 定义

```
Portion of config.h:

/* TFS definitions:
 */
#define TFSSPARESIZE            0x40000
#define TFSSECTORCOUNT          2
#define TFSSTART                0xfff40000
#define TFSEND                  0xfffbffff
#define TFSSPARE                0xfffc0000
#define TFS_EBIN_COFF           1
#define TFSNAMESIZE             23

Portion of tfsdev.h:

/* TFS Device table:
 */
struct tfsdev tfsdevtbl[] = {
    {    "/AM29160/",
```

```
        TFSSTART,
        TFSEND,
        TFSSPARE,
        TFSSPARESIZE,
        TFSSECTORCOUNT,
        TFS_DEVTYPE_FLASH, },

    /* If there was another TFS device, an
     * additional tfsdev entry would be here.
     */

    { 0, TFSEOT,0,0,0,0,0 }
};

#define TFSDEVTOT ((sizeof(tfsdevtbl))/(sizeof(struct tfsdev)))
```

程序清单 13.11 说明了 config.h 和 tfsdev.h 间的相关联部分。config.h 中的条目被 tfsdev.h 调用，同时也用于 TFS 代码的其他部分。大多数情况下，文件系统仅由一个 TFS 设备构建起来，起始、结束、备用容量及 config.h 与 tfsdev.h 间的段计数会有一对一的映射。如果由多个 TFS 设备构建一个系统，那么每个附加设备的 tfsdevtbl[] 数组中将会有一个附加的 tfsdev 结构。config.h 中也会有附加的定义供 tfsdev.h 引用。

TFS 定义建立了编译文件系统所需的特定目标信息，参数包含了 TFS（TFSSTART 和 TFSEND）用到的闪存空间的基址和终端地址、备用部分（TFSSPARE 和 TFSSPARESIZE）的地址和容量及分配给 TFS 的存储空间（TFSSECTORCOUNT）部分的数量，但不包括备用部分。如果容量不仅仅是默认值 23，那么才会用到 TFSNAMESIZE 的定义（对迄今为止的所有系统来说，23 刚刚好。但如果需要使它大一点或小一点，则可在 TFSNAMESIZE 中修改。惟一的要求是容量值要比某个能被 4 整除的数小 1）。

一个附加的 TFS 定义，即 TFS_EBIN_COFF 规定了构建 TFS 时应涵盖的加载程序，通常支持 COFF、ELF 和 A.OUT，其他的很容易加入平台。

13.3.5 INCLUDE 列表

监控器可由相当多的选项种类构成。通常，选项的特定功能是由相应的 CLI

指令来支持的。能够有选择地设置不同功能的第一个明显的好处就是，可删去不必要的功能用于节省内存空间。其他不太明显的好处是，当连接一个新目标时，功能少可减少初始建立时的复杂度和调试工作。开始时，除了最基本的功能部件以外，其他所有的功能部件都不必配置。在工作时，再逐个地加上符合特定硬件外设的功能部件。例如，开始连接时，在并不知道如何配置 RAM 和串行口的情况下，暂不考虑 TFS、以太网或者闪存问题，所以先禁用其内容，然后再逐个启用这些功能部件。在连接过程当中，采用这种方法可一次只处理一个问题。

程序清单 13.12 中的定义演示了一个配置监控器平台的例子。其公共/监控器目录中包含的 inc_check.h 头文件用来对程序做合理性的检查。在连接开始时，这些定义中的大部分都会被设置为零后，再逐个地给它们添加功能。开始连接时，首先要做的就是设置 INCLUDE_MEMCMDS 和 INCLUDE_XMODEM，这样就可以启动位于 reset.s 和串行口的连接了。串行口目标引导起作用后，可以用一些存储 modify 和 display 指令查看地址空间。如果存储器看起来配置正确，就可以用 xmodem 下载小的测试程序到 RAM 中。

注意

INCLUDE_XXX 定义列表包含 inc_check.h，程序清单 13.12 中的定义强制要求设置成 0 或 1，此不能省略。选择这种二中择一的办法，是因为这样可以迫使用户知道这一结构，从而减少了用户仅仅因为不知道这个选项，而使一部分被遗漏掉的可能性。

程序清单 13.12

```
/* INCLUDE_XXX Macros:
 * The sanity of this list is tested through the inclusion of
 * "inc_check.h" at the bottom of this list...
 */
#define INCLUDE_MEMCMDS          1
#define INCLUDE_PIO              0
#define INCLUDE_EDIT             1
#define INCLUDE_DEBUG            0
#define INCLUDE_DISASSEMBLER     0
```

```
#define INCLUDE_UNPACK          0
#define INCLUDE_UNZIP           1
#define INCLUDE_ETHERNET        0
#define INCLUDE_TFTP            0
#define INCLUDE_TFS             1
#define INCLUDE_FLASH           1
#define INCLUDE_XMODEM          1
#define INCLUDE_LINEEDIT        1
#define INCLUDE_CRYPT           0
#define INCLUDE_DHCPBOOT        0
#define INCLUDE_TFSAPI          1
#define INCLUDE_TFSAUTODEFRAG   1
#define INCLUDE_TFSSYMTBL       0
#define INCLUDE_TFSSCRIPT       0
#define INCLUDE_TFSCLI          1
#define INCLUDE_EE              0
#define INCLUDE_GDB             0
#define INCLUDE_STRACE          0
#define INCLUDE_CAST            0
#define INCLUDE_EXCTEST         0
#define INCLUDE_IDEV            0
#define INCLUDE_REDIRECT        0
#define INCLUDE_QUICKMEMCPY     1
#define INCLUDE_PROFILER        0

#include "inc_check.h"
```

13.3.6 多样化配置

下面讲述的是论述平台时需要的两个重要定义。

ALLOCSIZE ——监控器有它专用的内存分配程序，尽管并不主张在一个嵌入式系统中到处使用 malloc，但在某些情况下灵活地使用内存分配程序是很方便的。ALLOCSIZE 的定义告诉监控器应给其中的堆栈分配多少静态内存。malloc 堆栈用到的内存模块是配置给监控器的静态内存的一部分，把内存分配给监控器.bss 空间

内的堆栈，以使使应用程序能够在监控器之后驻留于内存空间中，而不必担心它会与来自监控器的堆栈发生冲突。另外，监控器的分配程序具有支持从两个不同且不相邻的内存模块（实质上是两个堆栈）进行分配的功能，使监控器既可为保持 RAM 的低使用率而建立了一个小的堆栈，又可在偶尔运行某些应用程序时而建立一个较大的堆栈。应用程序可以使用监控器的分配程序，并且仅仅通过把堆栈扩大为由应用程序分配的一些新的地址空间就可以增加堆栈。

SYMFILE——如果处理程序变量 SYMFILE 没有被作为替换值设置，那么这个定义会被标记表文件设置默认名，通常是 symtbl。

13.4　连接步骤

现在将依靠供应商提供的开发系统信息准备开始连接，最关键的资源为：

- CPU 数据表；
- 闪存（AM29PL160C）数据表；
- 串行口（处理器部分）数据表；
- 以太网设备（处理器部分）数据表；
- 评估板图表；
- 板上监控器的源代码。

（这些文档中每一个对应的.pdf 文件都可以从摩托罗拉和 AMD 的网站上检索到）。

下面将要讲述的测试是针对 MCF5272C3 评估板的，即全面而清晰地描述步骤是如何应用于其他系统的。如果在连接其他评估板时，你计划用这个例子作为向导，那么对你来说，在开始之前熟悉板及其文件编制就很重要了。

13.4.1　下载第一个镜像

首先需要制定计划，确定如何将代码（以二进制形式存在）与目标对应起来。在前面章节讲述的内容中，提到了三种可选择的办法，即 BDM、JTAG 及由外部编程器和插拔式引导 ROM 构成的手动闪存。在当前的情况下，还有第四种选择，即评估板与一个引导监控器（并非微型监控器）一起装载在闪存中，引导监控器下载程序来进行测试。

如果发展这种方式，则有一点需要特别注意，即在编写和测试引导虚拟硬件和设备驱动程序时，必须注意不要因疏忽而使其受到原始虚拟硬件或被它落在后面的

某个值的控制。MCF5272C3 的优良特性是，其闪存可以作为两个独立的模块来配置，其中任何一个均可以映射到包含复位矢量的空间中去（取决于跳线器的设置）。这种方法可使你在装载代码时导入准备好的监控器，而不导入你的测试代码。其过程如下：

- 由处理复位的监控器导入；
- 将代码下载到 RAM 中；
- 将代码移植到备用的闪存模块中；
- 更改跳线器；
- 复位。

在复位时，系统执行（导入）最新的下载代码，这确实是测试引导程序的一个好方法。用这种方法测试时，如果你的程序起作用了，则你知道是它自己起的作用——而不是不经意地从某个由原始监控器执行的设置里获益。如果你的代码不起作用，那么你所要做的就是把跳线器变回到原来的位置，且在原始的监控器上后退。这个特性真的很方便！

向闪存移植需要程序具有下载的功能，以及在刚刚下载它的位置执行程序的功能。MCF5272C3 监控器有一个 dl 指令，该指令可使你从一台主机中移植 S-record 文件；还有一个 go 指令，该指令可使你把控制权转给内存空间中一个指定的地址。

有了监控器中的这些工具，大多数终端评估程序应该可以把一个文件从它的源代码中移植到目标上。选择使用自己的工具（叫做 com），是因为用自己写的东西可以明确地知道如何连接串行口。另外，com 存于 CD 上，也是你在配置中将会用到的工具。为准备移植，从微机的 COM 端口连一根 RS-232 电缆到终端开发板上的 DB9 连接器，并给评估板上电。这个特殊的硬件不需要 NULL 调制解调器。目标的监控器要求主机以 19200b/s 传输数据，无奇偶校验、8 位数据位及 1 位停止位，全部为标准 stuff。为了以 19200b/s 连向 COM2 端口，在控制提示中需加命令行（在主机上）：com-b19 200-p2。

注意

装载于 MCF5272CE 上的监控器也支持网络下载（dn 指令）工具，用于连接 TFTP 服务器。我们坚持串行口下载，就是为了使过程简单以及消除在该点上的以太网连接中的未知数。

com 程序在这一点等待从 COM 端口或控制端口传来的字符。如果 com 从 COM

端口接收到一个字符，就把该字符显示到控制端。如果 com 从控制端接收到字符，就把该字符传送到 COM 端界面。如果现在点击返回，就应该看到 dBUG>提示。这一提示确认了主机与目标评估板之间的连接，即可以安装监控器并逐段进行测试。

参照目标/eval5272/监控器目录下的代码，config.h 文件应该在所有的 INCLUDE_XXX 宏定义为 0，且定义过 FORCE_BSS_INIT 的情况下配置。这一过程安装了一个完全最小化的监控器。reset.s 文件以 MCF5272CE 监控器中的少数文件为基础[1]，调用外设初始化函数（也是 MCF5272CE 监控器的一部分），最后调用 start()，并在开始运行 start()以及串行口之后，便越过了一大障碍。这一评估板可很容易地为串行口编写代码，因为 MCF5272CE 监控器的代码组织得相当好。我摘录了执行串行口 putchar/getchar/gotachar 功能的函数，并把它们嵌入到微型监控器结构（cpuio.c）中。如果该程序生效，则串行口就开始工作并且产生一个初始化了的目标。所以需要安装、下载该程序，并看它是否能运行。

编译文件能够生成一个基于 RAM-或 ROM-（闪存）的镜像，但却需要一个基于 RAM-的镜像，所以需运行 make ram。make 过程会编译监控器模块的每一部分，并把所有模块连接到一个特殊的内存映射表（RAM.lnk）中后，先将 COFF 文件转换成二进制代码，再转换成 S-records（因为那是 dl 命令所要求的）。

现在准备下载第一个测试镜像。在 dBUG>提示后，键入 dl，然后按下 ctrl-x，告诉 com 程序要向目标移植哪个文件。接下来键入 monCFEVAL.srec，因为该文件就是由编译文件中的 bin2srec 工具生成的。

在微机的控制窗口上，这一过程显示了许多的点（每 S-record 行一个）。这些点表明数据正在移植。这一过程结束后，就会看到一条消息表明 S-record 文件移植成功，同时 dBUG>提示返回。

在这一点，镜像位于评估板 DRAM 存储空间的 0x80000 处（感兴趣的话请参看评估板文档了解更多细节）。现在可以用 go 指令来执行某个函数或下载空间中的批函数。由于需要确定 start 函数的地址，因此在由编译文件（monCFEVAL.sym）生成的符号文件中搜索（用 grep）该地址。然后键入 go 0xADDR，这里的 ADDR 就是 start 函数的地址。这一行为使得存于评估板 RAM 空间中的微型监控器启动。输出见程序清单 13.13。

程序清单 13.13　微型监控器的运行输出

```
dBUG> go 80450
```

1. 连接中，如果比较一下微型监控器用到的特定目标代码和用来对 DBUG 监控器进行初始化的程序，就会发现我重点使用了摩托罗拉的程序。

```
                                MICRO MONITOR
                              CPU: ColdFire 5272
                    Platform: Coldfire 5272 Evaluation Board
                          Built: 09/14/2001 @ 21:16:42
                        Monitor RAM: 0x091310-0x0a4234
                        Application RAM Base: 0x0a5000

uMON>
```

程序清单 13.13 举例说明了几个简捷的步骤。从安装在评估板引导闪存里的监控器开始（dBUG），监控器均在输出列表顶部显示 dBUG>提示。go 指令从 start（）函数的地址（在刚刚下载的监控器中）开始执行，把控制权移交给微型监控器。微型监控器输出的第一个直观的信息是打印一个头文件，该文件显示了平台和 CPU 的名称，以及有关内存位置的描述。输出列表的最后显示新的提示行 uMON>。这一提示表明微型监控器正在运行。

我已经多次成功地安装及下载了监控器镜像，且加上的安装程序也能够运行。如果程序不能执行，就不会看到微型监控器头文件和 uMON>提示。为了做一个全面的检查，在 uMON>提示后需键入 help，如程序清单 13.14 所示。

程序清单 13.14　键入 help 做全面检查

```
uMON>help
call            echo        heap        help        ?           mstat
reg             reset       set         sleep       ulvl        version

uMON>
```

注意，在这一点上并没有很多指令可用，因为所有的 INCLUDE_XXX 宏都已被设置为 0。现在可以添加一些附加的特性。下一个主要特性应该是闪存，但在设置它之前，应先开启 INCLUDE_MEMCMDS 和 INCLUDE_XMODEM。启动这两项#defines 后，像以前一样重新安装和下载。帮助列表（见程序清单 13.15）现在包含 cm、dm、fm、mt、pm 和 sm。这些指令构成了内存显示、测试及由 INCLUDE_MEMCMDS 引入的修改指令。附加的 xmodem 指令作为设置 INCLUDE_XMODEM 的结果被引入。

现在我有一个系统，它可以像函数型监控器来运行，可以发出指令、显示/修改/测试内存（cm，dm，fm，mt，pm 和 sm）、下载数据至 RAM 中（xmodem）及

跳转到已下载的数据中（call）。注意，所有这些新增指令均来自于公共目录中的资源代码，不包括任何特定目标代码。一旦启动基本串口驱动器，就可以立即插入 INCLUDE_MEMCMDS 和 INCLUDE_XMODEM。

在这一点上，如果我们不在另一个监控器顶部运行，就会开始用 xmodem 来下载使用微型监控器 xmodem 和 call 指令的小程序（与用 dl 和 go 指令下载整个监控器的方法类似）。

程序清单 13.15　扩展帮助列表

```
uMON>help
call            cm          dm          echo        fm          heap
help            ?           mstat       mt          pm          reg
reset           set         sleep       sm          ulvl        xmodem
version
uMON>
```

13.4.2　启动闪存驱动器

驱动器列表中的下一个是闪存驱动器。启动这些驱动器是一个复杂的过程，第一步需要了解设备以及设备是如何连到 CPU 上的。在这种情况下，闪存设备需通过一个 16 位的连接器与 CPU 相连，起始地址为 0xffe00000。设备由 11 区段组成，common/flash/29pl160c/flashdev.c（见程序清单 13.16）的 SectorSizes160C[ ]数组为其指定大小（以字节为单位）。因为驱动函数已复制到 RAM 中，所以必须确认在执行 FlashInit（）函数期间，指令和数据高速缓冲存储器是禁用的。

config.h 把 INCLUDE_FLASH 定义为 1，并执行重构/下载，建立第一个驱动器连接，从 common/monitor/flash.c 和 common/flash/29pl160c/*.*导入了闪存代码。

程序清单 13.16　区段容量数组

```
/* SectorSizes160C:
 *   There are a total of 11 sectors for this part.   This table reflects the
 *   size of each sector in bytes.
 */
int SectorSizes160C[] = {
        0x4000,   0x2000,   0x2000,   0x38000, 0x40000,
        0x40000, 0x40000, 0x40000, 0x40000, 0x40000,
        0x40000
```

```
};

struct sectorinfo sinfo160[sizeof(SectorSizes160C)/sizeof(int)];
```

最后，这一连接的闪存驱动器提供了设备识别、扇区删除、写入及自动擦写功能。从设备识别开始，是因为它是执行和测试用到的第一个功能。对于 AM29PL160C 来说，设备标识分为设计标识和制造标识。对它的读取包括把 AutoSelect 指令写入设备，由 common/flash/29pl160c/flashpic.c 中的 Flashtype16（）来进行处理。我知道这一标识[1]的接收正确，因为重新开始新的安装时微型监控器没有产生出错信息。如果标识与规定的数据表不匹配，则驱动器会指示设备识别失败。这就意味着代码错误或者硬件不完全支持对闪存设备的写入。

验证驱动器通过 flash info 指令正确地给设备分配了扇区容量（见程序清单 13.17），显示了监控器默认的设备扇区映像图。如果要刻写程序，则在继续进行下一步之前，需回过头来检查一下每一扇区的容量是否与数据表中指定的扇区容量一致。

按照复杂性的顺序，下一项操作为 flash erase 指令。但是如果没有向闪存中写入任何东西，就不能对 flash erase 进行测试，也不能完成写入闪存的过程。其他接口程序虽然是类似的，但涉及的内容会多一点。flashwrite 过程（common/flash/29pl160c/flashpic.c 中的 flashwrite16（））与 Flashtype16（）类似，但却用了一个不同的指令序列，并且在指令序列之后写入数据，而不是读入，正如 Flashtype16（）所示。

程序清单 13.17 设备分区图

```
uMON>flash info
Device = AMD–29PL160C
  Base addr    : 0xffe00000
  Sectors      : 11
  Bank width   : 2
```

Sector	Begin	End	Size	SWProt?	Erased?
0	0xffe00000	0xffe03fff	0x004000	yes	no
1	0xffe04000	0xffe05fff	0x002000	yes	no
2	0xffe06000	0xffe07fff	0x002000	yes	yes
3	0xffe08000	0xffe3ffff	0x038000	yes	no

1. flashdev.h 中定义为 #define AM29PL160C 0x00012245（参照数据表）。

```
    4      0xffe40000   0xffe7ffff   0x040000     yes        no
    5      0xffe80000   0xffebffff   0x040000      no        no
    6      0xffec0000   0xffefffff   0x040000      no        no
    7      0xfff00000   0xfff3ffff   0x040000      no        no
    8      0xfff40000   0xfff7ffff   0x040000      no        no
    9      0xfff80000   0xfffbffff   0x040000      no        no
   10      0xfffc0000   0xffffffff   0x040000      no        no
uMON>
```

这里有一个附加的按钮。因为我正在使用一个 16 位的设备，所以对设备的每次写入都是 16 位的，并且必须应付起始地址为奇数或结束地址为偶数的可能性。这两种情况很相似，必须由驱动器进行处理。Flashwrite16（）函数有一个处理奇数起始地址的前端和一个处理偶数结束地址的后端。程序清单 13.18~程序清单 13.20 显示了 Flashwrite16（）函数的前端和后端。

程序清单 13.18　Flashwrite16()函数的前端

```
int
Flashwrite16(struct flashinfo *fdev,uchar *dest,uchar *src,long bytecnt)
{
    ftype    val;
    long     cnt;
    int      i, ret;
    uchar    *src1;

    ret = 0;
    cnt = bytecnt & ~1;
    src1 = (uchar *)&val;

    /* Since we are working on a 2-byte wide device, every write to the
     * device must be aligned on a 2-byte boundary.  If our incoming
     * destination address is odd, then decrement the destination by one
     * and build a fake source using *dest-1 and src[0]...
     */
    if (NotAligned(dest)) {
```

```
        dest--;

        src1[0] = *dest;
        src1[1] = *src;

        /* Flash write command */
        Write_aa_to_555();
        Write_55_to_2aa();
        Write_a0_to_555();
        Fwrite(dest,src1);

        /* Wait for write to complete or timeout. */
        while(1) {
            if (Is_Equal(dest,src1)) {
                if (Is_Equal(dest,src1))
                    break;
            }
            /* Check D5 for timeout... */
            if (D5_Timeout(dest)) {
                if (Is_Not_Equal(dest,src1)) {
                    ret = -1;
                    goto done;
                }
                break;
            }
        }

        dest += 2;
        src++;
        bytecnt--;
    }
```

前端（见程序清单 13.18）用来检查未对齐。如果从奇地址开始写入，那么建立代码并写入一个两字节的伪资源缓冲器（其中一个字节来自于真正的原始资料，另一个来自于已写的闪存区）后，确认资源缓冲器的起始地址，以保证中间模块能

够接收到起始于对齐界线的模块。

程序清单 13.19　Flashwrite16()函数的中间模块

```
    /* Each pass through this loop writes 'fdev->width' bytes...
      */

    for (i=0;i<cnt;i+=fdev->width) {

        /* Flash write command */
        Write_aa_to_555();
        Write_55_to_2aa();
        Write_a0_to_555();

        /* Just in case src is not aligned... */
        src1[0] = src[0];
        src1[1] = src[1];

        /* Write the value */
        Fwrite(dest,src1);

        /* Wait for write to complete or timeout. */
        while(1) {
            if (Is_Equal(dest,src1)) {
                if (Is_Equal(dest,src1))
                    break;
            }
            /* Check D5 for timeout... */
            if (D5_Timeout(dest)) {
                if (Is_Not_Equal(dest,src1)) {
                    ret = -1;
                    goto done;
                }
                break;
            }
```

```
        }
        dest += fdev->width;
        src += fdev->width;
    }
```

代码的中间模块（见程序清单 13.19）完成闪存写入的主要工作（取决于写入模块的容量）。

程序清单 13.20 Flashwrite16()函数的后端

```
/* Similar to the front end of this function, if the byte count is not
 * even, then we have one byte left to write, so we need to write a
 * 16-bit value by writing the last byte, plus whatever is already in
 * the next flash location.
 */
 if (cnt != bytecnt) {
      src1[0] = *src;
      src1[1] = dest[1];

/* Flash write command */
Write_aa_to_555();
Write_55_to_2aa();
Write_a0_to_555();
Fwrite(dest,src1);

/* Wait for write to complete or timeout. */
while(1) {
    if (Is_Equal(dest,src1)) {
        if (Is_Equal(dest,src1))
            break;
    }
    /* Check D5 for timeout... */
    if (D5_Timeout(dest)) {
        if (Is_Not_Equal(dest,src1)) {
            ret = -1;
            goto done;
```

```
                    }
                    break;
                }
            }
        }

done:
    /* Read/reset command: */
    Write_f0_to_555();
    return(ret);
}
```

函数的后端（见程序清单 13.19）处理在循环的结尾处剩余一个字节的情况（用同样的“伪缓冲器”技术）。

可以用 flash write 0xfff00000 0x80000 16 指令测试闪存。这一指令告诉微型监控器从 0x80000 处复制 16B 后，写入闪存区 0xfff00000。如果你用一个实际的评估板来完成这一过程，那么 0xfff00000 处很可能存在数据，调用 flash erase 7 指令删除该数据。在进行自己的连接时，你应该设计独立的测试，并确认写入程序的前端和后端都正常运行（用一个奇目的地址和一个偶字节计数器）。

描述删除和擦写的代码很相似，完整的细节可以到 CD 上查阅。

13.4.3　启动 TFS

现在已经完成了一些闪存驱动步骤，且可以转向闪存文件系统。假设已经正确地写入了 Flashwrite16( )和 Flasherase16( )函数，那么 TFS 只要被配置好了，就可以在启动时立即运行。

为启动 TFS，config.h 文件中的 INCLUDE_TFSXXX 宏都被设置成 1，这样就可以启动这些宏，并需要为目标工作平台配置 TFS（config.h 中更多的宏）。闪存设备占据了 0xffe00000~0xffffffff 段（2MB）。已安装了的监控器存于 0xffe00000，且并不使用闪存空间的上半部分，保留第 8 和第 9 扇区（0xfff40000~0xfffbffff），并分配 5MB 给微型监控器用来存储 TFS 文件。最后一扇区（10）分配给 TFS SPARE，第 7 扇区（0xfff00000~0xfff3ffff）作为安装 ROM 版的双微型监控器[1]。调整 config.h

1. 这一过程证明了板上闪存中 TFS 的多样性。在这种情况下，TFS 仅用 3 段闪存空间就与工作平台配合得很好。

中的 TFS 配置项目（见程序清单 13.11）以反映这些参数。

注意

切记，我现在正通过评估板的监控器运行微型监控器。这一过程有点非同寻常，但是针对这一论述，它还是很便捷的。这一方案使我们不必移走已安装在引导监控器中的引导部分即可测试代码。在最终的系统里，这一双监控器工作平台将转换成仅有一个微型监控器的工作平台。

建立一个新的镜像之前，也要打开 config.h 中的 INCLUDE_EDIT，这样就会有一个简单的方法来创建简易文本文件（INCLUDE_EDIT 参数能够启动监控器中的一个简单而有效的 ASCII 码文件编辑器）。现在安装及下载后，启动新的监控器，包括 TFS 和文件编辑器。

和以前一样，首先检查指令。这些指令是由于 TFS 的存在而被包含进去的（见程序清单 13.21），且注意，不仅存在 tfs 和 edit 指令，而且还包含许多有关 TFS 命令过程中的指令。

程序清单 13.21　包含 TFS 的微型监控器指令集

```
uMON>help
argv        call        cm          dm          echo        edit
exit        flash       fm          gosub       goto        heap
help        ?           if          item        mstat       mt
pm          read        reg         reset       return      set
sleep       sm          ulvl        tfs         xmodem      version

uMON>
```

依靠分配给 TFS 闪存空间的状态，复位时微型监控器可能会报告检测到不可靠的文件空间。这样的结果很可能是因为分配给 TFS 的闪存空间未被清空。你可以用以下方法来清空闪存空间，即用 flash erase 8~10 或者 tfs init 来删除被 TFS 占用的扇区（8~10）。

清空 TFS 空间后，文件可以加入，试用 tfs add 和 edit 创建文件，再删除一些文件后，试着用 tfs clean 清空。如果这些试验在运行时没有出错，你就可以确认 TFS 安装正确且运行正常。tfs stat 的输出有助于调试配置错误的 TFS。

13.4.4　启动以太网

现在所有的基本资源已经就位，该研究以太网驱动器了。除初始引导程序以外，最难处理的是构成新连接的过程。以太网设备比串行口稍微复杂一点。另外，如果出错，要是没有一个良好的以太网协议分析器（或检测头），就很难知道出错的是什么部位。因为微型监控器不需要任何中断，所以该驱动器可以做得尽可能简单。

为加入以太网支持程序，我修改了 config.h 中 INCLUDE_ETHERNET 的定义。这一变动启动了所有的基本代码，如 ARP、ICMP Echo（ping）以及一个运行于 UDP 端口 777 的简单 CLI 服务器。如果 ping 起作用了，就可以知道以太网传送和接收路径正在运行，所有其他的资源应该就位了。

注意

你可以把目标直接或者通过集线器连接到主机上，即用电缆把它连起来，所以需要尽量巩固你的接线工作。例如，使用同样的连接结构 ping 一些别的设备。用集线器连接目标（假设你没有协议分析器）的一项好处是，通常集线器会给你一些直观的反馈，使你了解线路上正在通信，每个集线器中都有一些发光二极管，使用户可以知道（以很高的水平）连接过程中发生了什么。

需要给目标分配一个 MAC 和 IP 地址。如果你工作于对等网络或者工作于孤立的集线器，则可以虚构一个 MAC 地址来测试，但是要确保你已分配到一个正式的地址。对这个例子来说，ColdFire 板已经配置了一个 MAC 地址，所以就使用那个地址。在此处的列表中，也可使用 00：60：1d：02：0b：08 得到该地址。

同样，必须建立一个 IP 地址。创建地址的规则同样适用于 MAC 和 IP 地址。再次假设一个孤立的网络，但必须确保目标 IP 地址与主机一样在同一个子网上。把微机的 IP 地址设为 135.3.94.39，微机的子网掩码是 255.255.255.0，所以设目标 IP 地址为 135.3.94.40，使两个设备位于同一个子网中。为使微型监控器用这些值进行自我配置，可创建一个 monrc 文件。其内容如下：

```
set IPADD 135.3.94.40
set ETHERADD 00:60:1d:02:0b:08
set NETMASK 255.255.255.0
```

记住，monrc 文件必须可执行，所以创建时就必须设置 e 标记。

现在，双监控器的 INCLUDE_ETHERNET 已经启动，存有 IP 和 MAC 地址的 monrc 文件也已建立起来。通过集线器或直接把微机连向目标，指令 ping 能够执行。如果一切都连接正确，则目标就会响应，ping 成功。

13.5 小结

上述内容介绍了目标结构、特定目标编译文件、包含监控器所需的 config.h 文件及连接过程的主要模块。完成所有的步骤后，可以看出，config.h 文件也可用做控制连接复杂性的机构。你可以用 config.h 每次加上一个子系统，用以实现一定的功能而不需一次处理所有的事情。

这个例子在某种程度上是独特的，因为它在已有一个监控器的系统中又加入一个监控器。如果调试监控器被删掉，就不得不用 BDM 或 JTAG 下载器或内存仿真器来运行初始版本。在最坏的情况下，就只能自己重新烧写引导闪存。

前面提到过，我重点使用了摩托罗拉调试监控器的代码。尽管我对 MCF5272C3 不太了解，却完成了整个连接。除此之外，我剪切了调试监控器的片断，并粘贴到微型监控器之中，很快地把它组织起来并运行。可从中获得两点启示：

- 摩托罗拉的员工组织调试源代码的工作干得很漂亮；
- 如果目标的引导代码已经存在（尽管不像调试程序做得那么好），那么连接微型监控器就很简单了。

结 束 语

从本书前面的内容中，读者已经完成了一个基本 CPU 的图解，并且学到了关于微处理器、RAM、ROM 一起工作的基本知识。你已经成功地将源代码转化到熟悉且易懂的环境（PC 或 Unix 工作站），并在电路板上将它做成（通过交叉编辑）CPU 所需要的二进制数据的模型。传送给设备程序员最终文件中的字节就是 CPU 想看到的字节，它们就是 CPU 在实现 C 语言程序的任务时所需要的指令和数据。

读者已经学会了采用比较简单的步骤解决第一次导入嵌入式系统的难题。本书中所介绍的步骤对于多种 CPU 都适用。微监视器提供了一套很好的开发工具，并且适用于不同种类的 RTOS 和硬件平台。读者还学会了一些不同类型的调试方法，且逐渐可以看到，把监视器用到新的平台会涉及到什么。

在大学第二学期的微积分学课上，我曾问过老师“在这之后还有其他的数学资料吗？”如果当时她嘴里有咖啡的话，那么咖啡一定会被喷出来。

与这种问题相同，本书所讲内容仅是一个开始。嵌入式系统的设计涉及了许多的决策，其中的一些解决方法在某些情况下作用很显著，但在其他情况下，可能不一定奏效。尽管没有单独的课本教你如何解决所有问题，但是本书所讲内容却提供了一些帮助解决所有问题的方法。这个行业将不断发展，不适合那些害怕改变的人。因为一方面你不想让自己在新的工艺和技术出现时被淘汰；而另一方面你还不想被不断的更新打断计划。所以，我们要随时注意行业的动向，看看哪些方面才能适合自己，不要跟着广告走，应不断研究新的工具和技术，思考新的策略，考虑适合你的选择。

附录 A　建立基于主机的工具箱

嵌入式系统不仅仅是嵌入式系统，不要让自己有“我只做硬件”的想法。尽管最终的产品均隐藏在让用户无法直观地看到内部部件的外壳中，但在开发过程中仍很难使 PC 上有一些工具来做不同的事。通常，为 PC 写程序并不太难（不论你用什么主机），因为有时只需要写一些仅仅作为主机程序的程序。一些工具可用于修改生成的可执行二进制文件，而另一些工具则可用于通过 RS-232 或以太网与目标机通信。附录 A 将描述一些我所写过的工具，我发现它们在连接主机运行监视器时很有用，且这些工具大部分都可独立于平台，具有很强的实用性。

本附录将介绍三种实用的代码片段。这些片段可提供以下功能：

- 与主机的二进制文件连接；
- 与 PC 的 COM 接口连接；
- 与 PC 的 UDP 插槽连接。

代码不是 C++的，你没有任何的 GUI——我写这些工具时从没有用过 GUI。我使用了微软 Visual C++的命令行工具和 nmake（没什么特殊）。以下所述内容仅可作为一个示例，其全部的工作过程请参见 CD。

A.1　与主机文件连接

在嵌入式系统程序中，你经常需要执行不同文件格式的转化。例如，你可能需要一个由二进制到 S-record 的转化（CD 中的 bin2srec），也可能需要一个工具，把二进制文件转换成 C 数组，并将它包含在 C 文件中（CD 中的 bin2array）。

前不久，我用 64 位的 CPU 设计一块板，在这块板上有两个启动设备，该板上的 CPU 要先从第一个设备上取 4 个字节，然后再从第二个设备上取 4 个字节。这个要求意味着我必须得到从 ELF 格式连接器输出生成的二进制文件，并把它分成独立的两个文件，每个文件包含要交换的那个二进制的 4 个字节块。为了能够自动分割，我写了一个叫做 chunker 的基于主机的工具。我可能不再使用这个工具，但它在我需要时却很方便。Chunker 工具很容易编写，但是如果我没有像目标机那样开发主机的能力（或者有一个相当的工具），那也会遇到麻烦的！程序清单 A.1 和程序清单 A.2 说明了 chunker 代码的大小，并给出与二进制文件连接的示例。

程序清单 A.1　chunker

```
main(int argc,char *argv[])
{
    char    *ofile, *buf1, *buf2, *bp1, *bp2, *fname, ofilename[128];
    struct  stat    stat;
    int     ifd, ofd, i, j, opt, size1, size2;

    debug = 0;
    ofile = "chunk";

    while((opt=getopt(argc,argv,"do:V")) != EOF) {
        switch(opt) {
        case 'd':
            debug = 1;
            break;
        case 'o':
            ofile = optarg;
            break;
        case 'V':
            showVersion();
            break;
        default:
            exit(1);
        }
}
if (argc != optind +1)
    usage(0);

fname = argv[optind];

/* Open a binary file, determine its size and
 * allocate two buffers of that size to contain the "chunks".
 * This is an over allocation, but who cares!
 */
```

```
ifd = open(fname,O_RDONLY | O_BINARY);
if (ifd == -1) {
        perror(fname);
        exit(1);
}

fstat(ifd,&stat);
if (stat.st_size % 4) {
        fprintf(stderr,"Input file must be mod 4\n");
        exit(1);
}

buf1 = bp1 = malloc(stat.st_size);
buf2 = bp2 = malloc(stat.st_size);
if ((!buf1) || (!buf2)) {
        perror("malloc");
        exit(1);
}
```

chunker 程序从处理一些命令行开始，使用 getopt（）和输入的参数列表。要被处理的文件的文件名是在 getopt（）参数列表中的第一个参数。代码先存储文件名，然后在决定了文件大小后打开它（通过调用 fstat）。代码分配两个缓冲区，在缓冲区中存放输入文件的数据块（见程序清单 A.2）。

程序清单 A.2　填充缓冲区

```
/* Now read in 4 bytes at a time from the binary file and
 * feed 4 bytes to one buffer then 4 bytes to the other buffer.
 * Keep this up till the input file is exhausted...
 */
size1 = size2 = 0;
while(1) {
        if (read(ifd,bp1,4) != 4)
                break;
        bp1 += 4;
        size1 += 4;
```

```
        if (read(ifd,bp2,4) != 4)
                break;
        bp2 += 4;
        size2 += 4;
}

/* Create 2 new files that represent the two "chunks" of the
 * original file:
 */
sprintf(ofilename,"%s1.bin",ofile);
ofd = open(ofilename,O_WRONLY|O_BINARY|O_CREAT|O_TRUNC,0777);
if (ofd < 0) {
        fprintf(stderr,"Can't open %s\n",ofilename);
        exit(1);
}
if (write(ofd,buf1,size1) != size1) {
        fprintf(stderr,"Can't write file %s\n",ofilename);
        exit(1);
}
close (ofd);

sprintf(ofilename,"%s2.bin",ofile);
ofd = open(ofilename,O_WRONLY|O_BINARY|O_CREAT|O_TRUNC,0777);
if (ofd < 0) {
        fprintf(stderr,"Can't open %s\n",ofilename);
        exit(1);
}
if (write(ofd,buf2,size2) != size2) {
        fprintf(stderr,"Can't write file %s\n",ofilename);
        exit(1);
}

/* Close, free and exit.
 */
```

```
        close (ofd);
        close (ifd);
        free(buf1);
        free(buf2);
        printf("Files %s1.bin (%d) & %s2.bin (%d) created\n",ofile,size1,size2);
        exit(0);
}
```

程序清单 A.2 中的主要循环使用了 read（），同时找回 4B，把这 4B 块分别放在两个不同的缓冲区中。缓冲区被装载之后，生成两个新的文件，其中的数据被存入这些文件中。最后，文件和缓冲区被释放，状况信息被打印，显示从一个输入文件生成了两个块文件。

A.2 与计算机串口的接口

在几乎所有的情况下，当与计算机的串口进行连接时，计算机自带的工具完全可以满足需要，但是也存在超级终端不能满足需要的情况。例如，如果我们需要观察串行传输线上的二进制数据，或者需要在某个对象上实现特定的串行传输协议，那么这时我们将会发现直接对计算机的 UART 进行操作将会很容易地实现。程序清单 A.3 是个人编写的串口接口程序的主要部分，我们可以在需要的时候用它来代替默认的 Windows 工具。

程序清单 A.3 串口的接口程序的主要部分

```
HANDLE
comOpen(int portnum, int baud)
{
    DCB dcb;
    HANDLE hCom;
    char portname[16];

    sprintf(portname,"COM%d",portnum);
    hCom = CreateFile(portname,
        GENERIC_READ | GENERIC_WRITE,        // Open for read/write.
        0,                                   // Sharing flags.
```

```
        NULL,                                   // Security attributes.
        OPEN_EXISTING,                          // Required for serial port.
        0,                                      // File attributes (NA).
        NULL);                                  // Required for serial port.

    if (hCom == INVALID_HANDLE_VALUE) {
        ShowLastError("comOpen() CreatFile()");
        return(INVALID_HANDLE_VALUE);
    }

    // Fill in the DCB...
    if (!GetCommState(hCom,&dcb)) {
        ShowLastError("comOpen() GetCommState()");
        return(INVALID_HANDLE_VALUE);
    }
    dcb.BaudRate = baud;
    dcb.ByteSize = 8;
    dcb.Parity = NOPARITY;
    dcb.StopBits = ONESTOPBIT;
    dcb.fDtrControl = DTR_CONTROL_ENABLE;
    dcb.fDsrSensitivity = FALSE;
    dcb.fOutX = FALSE;
    dcb.fInX = FALSE;
    dcb.fNull = FALSE;
    if (!SetCommState(hCom,&dcb)) {
        ShowLastError("comOpen() SetCommState()");
        return(INVALID_HANDLE_VALUE);
    }

    return(hCom);
}

void
comClose(HANDLE hCom)
```

```
{
    CloseHandle(hCom);
}

int
comRead(HANDLE hCom,char *buf,int count)
{
    DWORD    bytesread;
    DWORD    tot;

    tot = (DWORD)count;
    while(tot) {
        if (ReadFile(hCom,buf,(DWORD)tot,
            &bytesread,NULL) != TRUE) {
            ShowClearCommError(hCom);
            ShowLastError("comread ReadFile()");
            return(-1);
        }
        tot -= bytesread;
        buf += bytesread;
    }
    return(count);
}

int
comWrite(HANDLE hCom, char *buffer,int count)
{
    DWORD    byteswritten;

    if (WriteFile(hCom,buffer,(DWORD)count,
        &byteswritten,NULL) != TRUE)
            return(-1);
    return((int)byteswritten);
}
```

程序清单 A.3 中的代码依赖于某些 Windows 的服务，即 CreateFile（）、GetComm State（）、SetCommState（）、Readfile（）及 WriteFile（）。如果需要有关这些系统函数的细节，请查阅 Visual C++ CD documentation。comOpen（）函数为 CreateFile（）、GetCommState（）和 SetCommState（）等函数的封装，是用来打开一个公共端口，从而产生一个配置波特率、奇偶校验及其他串口参数的方便的钩子。comRead（）和 comWrite（）函数产生了对 ReadFile（）和 WriteFile（）系统调用的封装，并被用来从串口发送和接收数据。

A.3　基于 PC 的 UDP 处理：moncmd

基于以太网可兼容性的 MicroMonitor 包括了一个小型服务程序。它可以处理被 UDP 端口 777 接收到的输入数据包，且简单地对输入数据包进行处理，并假定其内容是去往 MicroMonitor CLI 的。服务程序将整个字符串传递给 docommand（）函数，并在 MicroMonitor 内部设置了一个标志位。这个标志告知 putchar 来构建 UDP 的信息包（每包一行），并将信息包返回给发布这条命令的客户机。当设定了标志位时，命令的结果将被送至目标控制台（如果存在的话）和发布这条命令的远程 UDP 端口。当输出的最后一行完成之后，MicroMonitor 清除标志位并输出一个结束信息包。这个信息包只包含一个 NULL 字节，以用来通知远程客户端对于命令的应答已经完成。

moncmd 工具（监视命令）是一个特殊的运行于 PC 上的 UDP 客户程序。它是一个简单的应用程序，允许用户与运行 MicroMonitor 的目标系统进行远程对话。程序清单 A.4 说明了当使用一台基于 Win32 的主机时，在 UDP 接口上发送和接收所需要的基本函数。注意，程序清单 A.4 是一个非常基础的程序，其意义仅在于举例说明 socket（）、sendto（）和 recvfrom（）等函数的使用方法（完整的应用程序请见本书所附的 CD）。

程序清单 A.4　do_moncmd()

```
/* do_moncmd():
 *   Open a socket and send the command to the specified port of the
 *   specified host.    Wait for a response if necessary.
 *
 *   hostname:
 *         Character string IP address
 *   command_to_monitor:
```

```
 *          Character string command destined for target monitor.
 *   portnum:
 *          Port number that the UDP packet is to be sent to.
 */
int
do_moncmd(char *hostname, char *command_to_monitor, short portnum)
{
      int i, lasterr;
      int msglen;
      ulong     inaddr;
      struct    hostent *hp, host_info;
      char      rcvmsg[1024];
      WSADATA WsaData;
      DWORD     tid;
      HANDLE    tHandle;

    if (WSAStartup (0x0101, &WsaData) == SOCKET_ERROR)
          err("WSAStartup Failed");

 targetHost = hostname;

 /* Build the UDP destination address:
  * Accept target name as string or internet dotted-decimal address.
  */
 memset((char *)&targetAddr,0,sizeof(struct sockaddr));
 if ((inaddr = inet_addr(targetHost)) != INADDR_NONE) {
      memcpy((char *)&targetAddr.sin_addr,(char *)&inaddr,sizeof(inaddr));
      host_info.h_name = NULL;
 }
 else {
      hp = gethostbyname(targetHost);
      if (hp == NULL)
           err("gethostbyname failed");
      host_info = *hp;
      memcpy((char *)&targetAddr.sin_addr,hp->h_addr,hp->h_length);
```

```
}
targetAddr.sin_family = AF_INET;
targetAddr.sin_port = htons(portnum);

/* Open the socket:
 */
socketFd = socket(AF_INET,SOCK_DGRAM,0);

if (socketFd == INVALID_SOCKET)
    err("socket failed");

/* Send the command string to the target:
 */
msgString = command_to_monitor;
if (sendto(socketFd,msgString,(int)strlen(msgString)+1,0,
    (struct sockaddr *)&targetAddr,sizeof(targetAddr)) < 0) {
    close(socketFd);
    err("sendto failed");
}

/* Now wait for the response:
 */
while(1) {
    int j;

    /* Wait for incoming message: */
    msglen = sizeof(struct sockaddr);
    i = recvfrom(socketFd,rcvmsg,sizeof(rcvmsg),0,
        (struct sockaddr *)&targetAddr,&msglen);

    if (i == 0) {
        fprintf(stderr,"Connection closed\n");
        close(socketFd);
        exit(EXIT_ERROR);
    }
```

```
        /* If size is 1 and 1st byte is 0 assume that's the target */
        /* saying "I'm done". */
        if ((i==1) && (rcvmsg[0] == 0))
            break;

        /* Print the received message: */
        for(j=0;j<i;j++)
            putchar(rcvmsg[j]);
        fflush(stdout);
    }

    close(socketFd);
    return(EXIT_SUCCESS);
}
```

来自前线

这里，我将诚实地声明其实我并不知道函数 WSAStartup（）的实际功能。但事实上，这也没有什么关系，就像运动鞋的广告上所说的“just do it!”

函数 do_moncmd（）对 IP 地址或命令行所提供的主机名进行处理后，使用 SOCK_DGRAM 打开了一个端口并声明为 UDP 的端口。当端口被打开之后，代码将利用函数 sendto（）和 recvfrom（）来发送和接收 UDP 数据包。对函数 sendto（）的调用可将命令行上的字符串传送到在远程目标上运行的 MicroMonitor 服务任务。最后，while（）循环处理应答，接收的每一行都将在控制台上打印输出。为了支持接收多行应答的功能并且仍然能够知道最后一行被接收的时刻，monitor 的服务程序利用一个仅包含 NULL、大小为 1B 的数据包来结束消息。远程终端的 moncmd 检测到这个数据包后将在适当的时候退出。

A.4 小结

在固件开发库里的其余几个工具也是很方便的。当在一个基于某一网络平台的终端上开发程序时，TFTP 客户终端服务程序和 DHCP/BOOTP 服务程序是很有帮

助的。计算机上的服务程序允许你检测接入网络的设备，而不需要系统管理员在 Unix 主机上安装服务程序。

使用自己开发工具的缺点是，如果你对某件事情并不清楚的话，那将会把工具里的错误传播到目标固件上去；而这种做法的优点是，其中的某些工具开发起来并不是很复杂，且这些工具公开的源代码可允许我们在其中增加代码以仿真协议的错误或丢失的数据包。

利用能够分析可执行图形（如 coff 或者 elf）的工具，我们可以将符号和内存映射信息转储为其他的格式，这将比编译器附带的工具所支持的格式更加方便。同时，在 MicroMonitor 和 TFS 的情况下，由于 TFS 解压缩时可采用非标准的处理步骤，因此对某些基于主机的工具是非常方便的。

这些工具完整的源程序和其他部分内容请见本书所附的 CD。

附录 B　RTOS 概述

通常，实时操作系统（RTOS：Realtime Operating System）指的是一种操作环境，运行于嵌入式系统上，并在可预测的时间间隔内，使编写正确的程序能够对特定激励做出反应。事实上，我所说的 RTOS 是指一种运行于嵌入式系统上的操作环境，可以提供至少具有建立多线程的执行（通常指任务）能力的 API。多任务操作系统就是指能够同时执行多个任务的操作系统。

RTOS 为每个任务建立了一个可执行的环境，可以很方便地在任务之间传递消息，在一个中断处理程序和任务之间传递事件，区分任务执行的优先级，并协调多个任务对同一个 I/O 设备的调用。尽管这些功能对大多数的多任务操作系统是很普通的，但作为 RTOS 来说，必须设计这些功能，才能使其在某些最大的可预测的时间内对输入的激励做出反应。RTOS 的名字已经很清楚地指明了实时性，但是准确地说很大一部分对 RTOS 的应用并没有太多对实时性的要求。

然而，如果使用 RTOS 的话，某些实时性的应用将会变得更加简单。当嵌入式系统变得越来越复杂的时候，固件工作的复杂性也随之增加。通常，一个很大很复杂的嵌入式系统可以分解为一系列较小、较简单的并行任务来实现，各个任务之间互不干扰。大多数情况下，这种多线程的解决方案可以使代码排除了并行任务中的人为因素，使复杂度更低，更模块化，使工程由更简易和标准化的模块组成，处理起来更轻松，而且更快捷。

本书所提到的许多其他问题完全可以写一本题目为 RTOS 的书了。但这里只局限于讨论一些基本的定义。

注意

任何多任务操作系统的基本模块都是一个个的任务。一个任务就是一个函数，一定数量的资源或时间被分配给它来完成一定的功能。对于大多数的 RTOS，每个任务都有一定的优先级。RTOS 的优点就在于你可以让几个不同优先级的任务同时运行，每个任务并不需要知道是否正在和其他任务并行执行。

B.1　调度程序

RTOS 的核心就是调度程序。调度程序是一块代码，它指定下一个被执行的任务。通常在每个系统调用结束的时候，通过系统唤醒调度程序来决定当前正在运行的任务是否需要挂起以使执行其他任务。

操作系统认为每个任务都处于几个执行状态之一。每种 RTOS 都有自己的执行状态定义，但有三种状态几乎总是相同的，即运行态、就绪态及阻塞态。在每个时刻只有一个任务可以处于运行态，该任务被称为当前执行活动任务。一个任务可以被执行的条件是处理器没有被处于就绪态的高优先级任务占用。如果一个任务需要等待一些系统调用时才能执行，那么它就处于阻塞态。当一个任务脱离阻塞态时就进入就绪态。

如果当前任务结束或是失去对处理器的占用（可能是由于更高优先级的任务处于就绪态了），操作系统就执行环境转换。其过程为：RTOS 将与当前活动任务对应的 CPU（或环境）状态保存在一些结构中（实际上叫做状态控制模块——TCB）。然后，RTOS 从新任务的 TCB 中找到它的状态并把它载入 CPU 作为新的环境。如果调度程序是基于优先级的，那么新的任务一定比前一任务有更高的优先级。调度程序总是选择处于就绪态的优先级最高的任务，这一点很重要。通常，系统中存在几个优先级比当前任务高的任务没有执行，其原因是它们没有处于就绪态。

对于特定的 RTOS，调度程序可能不在当前运行任务进行系统调用的时候被唤醒。如果 RTOS 支持抢占，则环境转换有可能因为中断的边际效应而发生。例如，中断有可能在一个外设得到输入的时候发生。中断处理程序捕获了这个输入，把它保存为一个监听任务（通过系统调用）后，唤醒调度程序作为它的返回协议的一部分。如果监听任务是因为等待输入而被阻塞，那么它现在处于就绪态。如果监听任务（或其他就绪的任务）比中断任务有更高的优先级，那么调度程序执行任务转换，将中断任务悬挂并且载入高优先级的任务。

如果系统调用导致当前任务被挂起，则这种环境转换称为同步的，因为它是通过内联代码实现的。如果一个中断发生并且被中断的任务被挂起，则这种环境转换称为异步的，因为它是在被中断任务执行过程的任意点发生的。

B.2　任务、线程和过程

线程是指在单个过程的环境中同时执行的几个潜在的任务之一。过程是指在一

个平台上运行的几个潜在程序之一，该平台能够利用内存管理单元（MMU）。这里提到 MMU 的利用是很重要的，因 MMU 是更强大的 CPU 硬件扩展，这种 CPU 允许程序在被保护的内存中运行。

来自前线

实际上，任务和线程是同一样东西。我们通常认为任务是在嵌入式系统中，而线程是在大型机里。

MMU 提供的保护可将一个过程与一个线程区分开来。和线程不一样，不管过程多么复杂，它都不可能破坏其他过程的内存空间。由于 MMU 提供的保护，各个过程彼此是安全的。如果当前运行的过程有一个迷失（stray）指针开始废弃内存，则它只能废弃自己的空间，因为在它能够访问其他过程的内存空间之前会导致一些与 MMU 相关的操作系统异常。

你可能会奇怪，为什么在一个基于 MMU 的多处理环境中仍然需要多线程（或多任务），因为 MMU 提供的保护在环境转换时增加了很多的系统开销。回忆一下前面内容中提到的，当前运行任务的环境就是 CPU 的状态，此环境包括寄存器组和几个与任务相关的变量。而对于一个过程来说，环境是指所有上面提到的再加上重新配置 MMU 的开销和把新的过程从次要存储器推入 RAM 的可能开销。

这些开销可能会令人大吃一惊，因为看起来平淡无奇的转移可能会导致整个过程的转换。例如，因为一个过程不能访问其他过程的存储空间，所以一个过程与其他过程对话的惟一途径就是通过某些类型的消息传递系统调用——这种调用可同样导致一个环境的转换。

因为过程不可避免地存在额外开销，所以对于实时环境的多任务系统来说线程是更好的选择。其环境转换的开销最小，并且线程之间的数据共享也简单得多。但运行线程也是有危险的，如果多个线程在同一块内存空间中运行，那么其中一个线程的坏指针或堆栈溢出可能会破坏其他线程的空间。正如前面的章节中提到的，发现那样一个错误是很有挑战性的。

B.3 抢占、时间分割和中断

RTOS 允许并发任务且可以忽略彼此的存在，但并不允许程序员忽略与并发有

关的棘手问题。虽然 RTOS 内部的机制可以很宽松，但是开发者必须知道应怎样正确利用这些机制。

例如，如果系统中有 3 个任务，每个任务可完成各自的任务，那么只有优先级最高的任务会得到执行。为了避免这种情况和其他潜在的问题，必须为每个任务编写代码，以使在当前任务没有执行任何有用的工作时，使它返回调度者并让其他任务执行。一个任务可以同步或异步地交出对处理器的控制权。对于同步的情况，任务应该产生一些系统调用以通知操作系统任务正在等待某事发生。如果操作系统知道任务只是在等待，就可以调度一些其他任务来做一些实际的工作。让操作系统等待通常叫做阻塞，每个任务的目标应该是尽快地完成要完成的工作，然后再阻塞。

另外，如果一个任务不是为了阻塞，那么就需要一些其他的技术来造成环境转换的发生。这就是中断。中断造成当前运行代码异步地暂停。这种异步地中断可以导致当前任务被其他一些高优先级的任务抢占。

一个 RTOS 的中断处理程序通常在头和尾加上与 RTOS 相关的代码。这部分代码的结束部分被称为调度程序。那些在一个中断处理程序中的合法系统调用可以检测他们什么时候在中断中被调用。系统调用中的代码知道，如果它处于一个中断处理程序中，则通常不会调用调度程序，除非中断处理程序中存在对调度程序的调用。当中断处理程序结束，它唤醒调度程序，这时如果有一个拥有更高优先级的任务处于就绪态，转换就发生了。中断服务程序返回到另一个任务。

如果中断处理程序中没有系统调用来导致环境转换，则会怎么样了？如果多个任务拥有相同的优先级并且都不愿意阻塞，则一个简单的以 tick 为单元的时间分割系统仍然可以保证系统运转。RTOS 的时间分割是指在多个任务拥有相同优先级的时候可分配给每个任务一定的 CPU 时间以让他们同时执行的能力。如果你的系统有 3 个任务（优先级都相同），且它们都想占用处理器，则 RTOS 的时间分割能力允许每个任务占用一定固定数量 tick 的 CPU 时间。当一个任务的时间到了，调度程序会自动地将环境转换至下一同优先级的任务。这一过程通常被称为自动时间分割轮换。

B.4　信号机、事件、消息和定时器

现在你对 RTOS 的核心部分已经有了一个基本的了解。我已经不只一次地提到系统调用了，却一直没有解释到底什么是系统调用。就像线程和任务，系统调用的定义取决于实际系统。一般而言，系统调用是指一种操作系统的功能一旦被唤醒，就会导致一个进入核心的环境转换，因为系统调用涉及的系统资源只有在核心模式

下才可以达到。对于嵌入式系统，系统调用一般就是指一个 RTOS 的 API 函数，我在这本书里就是这样定义系统调用的。就算 RTOS 拥有了到目前为止我所说的所有功能，它还是不能真正做什么。这与一台拥有笨重马达和油料却没有挂任何器具的拖拉机类似。RTOS 需要挂的器具包括允许我们在任务间通信、在中断处理程序和任务间通信、确定在同一时间只有一个任务在运行及设置定时器等等的系统调用。每个 RTOS 都有一套系统调用，但是所有的系统调用都有一些通用的基本函数。

“信号量”在共享资源时可以被用来使访问同步。通常的同步形式是相互异或，也就是说任务要协调它们对同一资源的访问，这样在同一时间内只有一个任务操作这个资源。例如，如果一个被信号量保护的函数正在开始执行，而一个环境转换开关又在向另一块代码传输指令，并且就是要通过调用这个被信号量保护的函数，那么后面的这个调用请求就会被阻止。在某些系统中，多于一个的任务可以对同一资源进行访问，这时互斥性的访问是必要的。

例如，假设一个系统有一些存储器映射寄存器，它们是 8 位的，每一位都可以控制一个 LED（1 为高）。程序清单 B.1 就是使用这个函数来修改这些位的。

程序清单 B.1　LedOn()

```
unsigned char
LedOn(unsigned char onbits)
{
    unsigned char current_bits;

    current_bits = *(unsigned char *)LED_PORT;
    current_bits |= onbits;
    *(unsigned char *)LED_PORT = current_bits;
    return(current_bits);
}
```

参考程序清单 B.1，就在变量 current_bits 从 LED_PORT 的地址得到值之后，一个异步的环境转换开关出现了，同时其他的任务正在调用这个函数，这时向函数传送一个不同的值，会发生什么呢？回答是第二次任务建立的设置会丢失。让我们假设在第一次调用 LedOn（）时，onbits 的值是 0x01，而现在 LED_PORT 的值是 0x80。如果没有环境转换开关出现，则结果是 LED_PORT 为 0x81，即意味着两个 LED 会被点亮。

如果这时出现一个环境转换开关，则结果就会不同。在这一行之后第一次调用

LedOn（）函数

```
current_bits = *(unsigned char *)LED_PORT;
```

current_bits 中存储的值是 0x80。现在假设环境转换开关在这一行之后出现，而另一个任务在调用这个函数时得到的 onbits 值是 0x02。第二次请求在完成之后离开，LED_PORT 会在一段时间保持值为 0x82，且环境存储的是原来的任务。这个例子中的函数会在这一行之后继续执行

```
current_bits |= onbits
```

在原来的环境中，current_bits 被设成 0x80、在 onbits 等于 0x01，并且 current_bits 的值是 0Red 时，0x81 就会被写到 LED 灯上，原来的任务建立的值（0x82）就会丢失。

这个问题的解决办法是相互异或。如果这些 LED 的操作是由恰当的信号量操作完成的（在顶部取得，在底部释放），那么在环境转换开关出现时，先前的任务就不会被允许操作 LED，直到控制信号量的这个任务完成之后。

事件是系统提供的一个标志，当它被一个任务或中断处理程序设置之后，会唤醒某些其他的任务（如进入准备运行状态）。特别是，当事件同时关联两个系统时，则发出两个信号：一个表示事件的发送；另一个表示事件的阻塞等待。事件的特点是不排队。在某个任务在等待事件保持阻塞时，如果同一个事件在得到这个任务的承认之前被发送了几次，那么这些发送会丢失。因为事件只是一个标志，在这个标志被设置之后，重新设置一遍是没有意义的。通常事件比其他形式的内部进程通信开销更小，因此它通常被用在中断服务程序和任务之间的通信上。

消息提供了一个机构，允许任务（或中断处理程序）发送一些数据给其他任务。不像事件，消息是要排队的。当多重的消息被送到某些被阻塞的任务时，每一条消息都被操作系统排队，来延迟消息接收的开销。

大多数实时操作系统支持把发送的消息发送到队尾或队头。在某些情况下，将消息送到队头来加速其执行是很方便的，但是这个特点不能被滥用。消息应该在任务之间或任务与中断之间传送，而且消息通常意味着比事件更多的开销。

计时器可能是实时操作系统（RTOS）中最重要的一项功能了。不仅一些系统对计时器有特别的设置，而且调用很多其他的系统通常也提供一些超时的机构。最通常的计时器函数就是使一个任务“睡眠”一段时间，或可在将来某天某时唤醒某个任务。最简单的例子就是使一个 LED 不停地闪烁（见程序清单 B.2）。

程序清单 B.2 task_BLINKER()

```
void
task_BLINKER(void)
{
    while(1)
    {
        Turn_On_Led();
        GoToSleep(1000);
        Turn_Off_Led();
        GoToSleep(1000);
    }
}
```

系统调用的 GoToSleep（）函数是一个计时函数，可周期性地唤醒任务。除了 GotoSleep（）函数，其他的系统调用本质上都使用计时器来完成超时中断（time-out）功能。例如，一个任务可能想等待一个即将到来的事件，但如果事件在 30s 之内没有发生，那么一些其他的动作就要发生。在这种情况下，系统可调用一个事件机制，但在本质上，这个事件机制使用的还是计时器。

B.5 重入

虽然我们没有正式命名它，但我们实际上已经讨论过了重入问题。我们讨论了中断和抢占，还讨论了信号量怎样帮助用户写一段极微小的代码。中断和复合的线性编程环境很好，但它们增加了代码的复杂性，因为它们允许一个函数在多于一段的程序中同时运行。在 LED 那个例子中，错误（程序清单 B.1 中一个函数的结果丢失，导致点亮错误的 LED）的发生是函数本身不能处理重入问题。只有一个函数可以同时在几个线程和进程中被安全地调用时，这个函数才可以被认为是可重入的。上述情况（见程序清单 B.1）解决的办法是使用信号量来禁止重入的发生。在其他情况下，代码就会允许重入。

函数使用全局和静态变量可能也会导致程序不能重入。与 LED 的情况相同，如果一个函数的进程正在修改一个全局变量，而其他任务也在操作这同一个变量，这样函数在修改的中途就会被其他任务打断，那这个数据可能就会作废。通常，重入函数可以在任何时候都使用局部（堆栈）变量（堆栈中的变量一定是创建它的线程中的局部变量）。当一个函数需要对一个全局资源进行操作，而且要进行重入时，

它可能是需要互斥地对一个全局资源进行连续地访问。

B.6　好的并行和差的并行

这里用一个好的设计和差的设计对比的例子来说明差的代码是怎样破坏一个好的 RTOS 的。假如在程序清单 B.2 中的函数 task_BLINKER（）是一个基于 RTOS 应用的任务。这个任务要完成的工作很简单：每秒唤醒一次，并且触发 LED 的状态。LED 被调整，并且系统调用 GoToSleep（）被立即执行。这个任务只做这些后，就阻塞了。现在看看程序清单 B.3 中这个替代的函数。

程序清单 B.3　BLINKER_BAD()

```
void
task_BLINKER_BAD(void)
{
    while(1)
    {
        int timeout;

        Turn_On_Led();
        for(timeout=0;timeout<500 000;timeout++);
        Turn_Off_Led();
        for(timeout=0;timeout<500 000;timeout++);
    }
}
```

如果这个超时循环需要 1s，则程序清单 B.3 中的这种近似可能算是一种可以接收的替代了。但是，不幸的是，程序清单 B.3 离可用还相去甚远。先不说这个循环计时的不精确性，程序清单 B.3 没有想到系统中还有其他的任务在运行。因为它没有发出任何阻塞系统的指令，且将系统中分给它的资源全部占用了。在特定的环境中这个不好的设计可能还可以结束执行，但是它确实是一个不好的设计，因为它根本就没有利用 RTOS 提供给它的资源。

在很多情况下，程序清单 B.3 根本就不会工作。假如由于某种原因，一定要使 LED 的持续闪烁。要确保按时地执行，任务可以在一个高优先级下运行。如果真的这样做了，则抢占方式也不能使其他任务获得 CPU。因为只有当一个任务已经处于

准备执行的状态，并且现在正在执行的任务的优先级比它低时，抢占才能使这个任务获得执行。在这一点上，RTOS 并不能克服一个蹩脚的设计！程序开发者仍必须编写清晰而明确的代码。

B.7 小结

尽管并不是每一个实时的应用都需要 RTOS 的开销，但是如果作为几个独立的多线程或任务来实现的话，目前大多数的嵌入式应用的设计与编程将会变得很简单。本附录阐述了 RTOS 的概念及工作过程，并解释了在通常的情况下，调度程序是如何工作的，以及我们可以在一般的 RTOS 上找到什么样的服务。现在，我们应该能够理解任务和进程之间及获取优先控制权与取得中断之间的区别，希望这些总结能够帮助你决定是否需要查阅更多的关于 RTOS 的详细信息。

如果使用得当的话，RTOS 可以成为一个有效的工具，但它并不能解决实时性编程中的所有问题，也不是能够将应用程序开发员转化为实时性程序开发员的权威工具。一个好的 RTOS 可以允许一名天才的实时性程序开发员能够在一个问题中更加清晰地表达客观存在的并发性，并可以提供封装了许多重要的实时性的基础元素（如标志、事件、消息，等等）的结构。需要着重指出的是，并不存在可以消除并行程序设计内在的复杂性的 RTOS，即使应用程序将在 RTOS 的顶层上运行，程序设计者仍然必须清楚地理解相互排斥性、可重入性及死锁等问题的含义。如果对这些问题并不很清楚的话，那么程序设计者必须从基础的并行性的概念开始 RTOS 的学习。如果能够完全理解这些问题，那么程序设计者使用特定的 RTOS 来处理这些问题时，将会相对简单多了。

[General Information]
书名=嵌入式系统固件揭秘
作者=
页数=328
SS号=11130024
出版日期=